Methoden und Werkzeuge zum Einsatz von rekonfigurierbaren Akzeleratoren in Mehrkernsystemen

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Fabian Nowak

aus Gießen

Tag der mündlichen Prüfung:	06.02.2015
Erster Gutachter:	Prof. Dr. Wolfgang Karl
Zweiter Gutachter:	Prof. Dr.-Ing. Jörg Henkel

Für meine Kinder Leonie und Julius.

Danksagung

Zuallererst gebührt mein größter Dank meinem Betreuer Professor Dr. Wolfgang Karl für seine langjährige Unterstützung und nicht gescheuten Mühen. Er lehrte mich die Durchführung wissenschaftlicher Arbeiten, gab mir ein angenehmes und produktives Arbeitsumfeld und gewährte große Freiräume bei der Gestaltung und Durchführung meiner Arbeit. Den dabei beschrittenen, sehr lehrreichen Weg möchte ich rückblickend nicht missen. Mit ihm habe ich insbesondere einen Betreuer gefunden, der sich meines Geistes annahm und mir zeigen konnte, wie sich die Vermittlung von Inhalten an andere Menschen bewerkstelligen lässt.

Im universitären Umfeld möchte ich zunächst Professor Dr. Jörg Henkel für die Übernahme des Korreferats sowie für seine wertvollen Anregungen danken und aufgrund der erfolgreichen Zusammenarbeit im Umfeld numerischer Berechnungen Professor Dr. Vincent Heuveline und Dr. Jan-Philip Weiß Dank aussprechen. Für die Idee mit dem exakten Skalarprodukt auf FPGAs danke ich ausdrücklich Professor em. Ulrich Kulisch.

Für das produktive Arbeitsumfeld zeichnen auch sehr stark meine direkten Kollegen verantwortlich. Besonders möchte ich mich bei Dr. David Kramer für konstruktive, inhaltliche Diskussionen bedanken, bei Dr. Rainer Buchty für die initialen Ideen und Förderung, bei Dr. Lars Bauer für die Gespräche über partielle dynamische Rekonfiguration auf FPGAs von Xilinx, bei Dr. Martin Schindewolf für ergiebige Diskussionen über die Durchführung wissenschaftlicher Arbeiten und bei Dr. Mario Kicherer für die gemeinsam durchgeführten Arbeiten sowie bei allen weiteren Kollegen des Lehrstuhls für Rechnerarchitektur und Parallelverarbeitung (CAPP), des Chairs for Embedded Systems (CES) und des Chairs of Dependable Nano Computing (CDNC).

Desweiteren haben fachlich die Kollegen Mareike Schmidtobreick und Dr. Florian Wilhelm seitens numerischer Verfahren beigetragen. Mit Michael Rückauer konnte ich besonders wertvolle Gespräche über Hardwareentwicklung führen. Besondere Beiträge haben ferner die Studenten Ingo Besenfelder, Michael Bromberger, Jiefei Chen und Andreas Hampp geleistet (in alphabetischer Reihenfolge). Ihnen allen gebührt dafür ein großes Dankeschön.

Mit Alastair Reed von ARM Ltd., Cambridge, UK konnte ich erkenntnisreiche Gespräche über das wissenschaftliche Arbeiten und insbesondere die Literaturrecherche führen, wovon ich sehr profitiert habe.

Meine Vorgesetzten und Kollegen bei der SEW-EURODRIVE GmbH & Co KG haben mich bei der Fertigstellung der Arbeit insofern unterstützt, als dass sie über gelegentliche persönliche Erschöpfungsanzeichen hinweggesehen haben, wofür ich ihnen großen Dank aussprechen möchte.

Anonym danken kann ich nur all jenen Personen, die ich auf wissenschaftlichen Veranstaltungen wie Konferenzen gesprochen habe, mit ihnen meine Arbeit diskutiert habe, dabei weiteres Wissen erworben, neue Aspekte entdeckt habe.

Der Weg hin zu dieser Arbeit wurde maßgeblich durch meine Eltern unterstützt. Ich schätze sie und ihre Unterstützung sehr, konnte immer auf sie bauen. Meinem Freundeskreis aus Schwäbisch Hall ist mein besonderes Interesse an Rechnern, Informatik und insbesondere Hardware zuzuschreiben, aus dem zunächst das Informatik-Studium resultierte und jetzt die Beschäftigung mit Soft- und Hardware.

Abschließend danke ich meiner eigenen Familie mit Anna-Lena, Leonie und Julius für die Ausdauer und das entgegengebrachte Verständnis.

Zusammenfassung

Rechensysteme mit Mehrkernprozessoren werden häufig um eine Beschleunigerkomponente wie *Field Programmable Gate Arrays* (FPGAs) zu heterogenen Systemen erweitert. FPGAs zeichnen sich auch dadurch aus, dass sie auf Bitebene Daten verarbeiten und speichern können. Sie eignen sich damit zur Implementierung und Ausführung von Spezialoperationen in Hardware und vermögen so, Anwendungsteile und Anwendungen zu beschleunigen. FPGAs sind nicht nur leistungsfähig, sondern können auch energiesparender rechnen und dadurch die Leistungsaufnahme sowie die Abwärme reduzieren.

Die Verlagerung von Anwendungsteilen in Hardware wird meist von Hardwarespezialisten vorgenommen. Für viele Anwendungen wurden dedizierte Beschleuniger entwickelt. Für numerische Verfahren bietet sich ein Skalarprodukt als zu verlagernde Spezialoperation an. Die Durchführung des Skalarprodukts in exakter Arithmetik kann aufwendige Reorthogonalisierungen verhindern oder allgemein die Stabilität sowie die Konvergenz verbessern. Ein exaktes Skalarprodukt ist in Software sehr aufwendig, kann aber effizient in FPGAs implementiert werden aufgrund der Verwendung von Bitoperationen. Deutliche Beschleunigung durch die Verlagerung von Software in Hardware lässt sich in der Bioinformatik erzielen. In dieser Arbeit wird beschrieben, wie die erste Stufe der Vorfilterung der bioinformatischen Anwendung HHblits in rekonfigurierbare Hardware ausgelagert wird.

Der Anwender kann von diesen Beschleunigern nur dann profitieren, wenn die zu beschleunigende Anwendung genau diese implementierte, vorhandene Funktionalität benötigt und wenn die umgebende Rechnerarchitektur die effiziente Nutzung gestattet.

Damit Anwender selbst rekonfigurierbare Hardware programmieren können, existieren hochsprachliche Ansätze. Dabei entfällt die mühsame Entwicklung von Hardwareschaltungen weitgehend. Aus der hochsprachlichen Formulierung wird automatisiert eine Hardwarebeschreibung erzeugt. Aus jeder Beschreibung muss zeitaufwendig eine FPGA-Schaltung synthetisiert werden, was bei der Entwurfsraumexploration und bei der Fehlersuche hinderlich ist und so die Entwicklung deutlich erschwert und verlangsamt.

Mein Beitrag ist die komponentenbasierte Programmierung und Verwendung von rekonfigurierbarer Hardware in Mehrkernsystemen mit automatischer Beachtung der Datenlokalität. Für sowohl den Koprozessor als auch den Prozessor stellen Hardware- und Softwareentwickler domänenspezifische Komponenten bereit. Die Komponenten sollen in ihrer Domäne häufig wiederverwendbar sein, damit der Syntheseaufwand für den Koprozessor weitestgehend entfällt. Es wird ein mikroprogrammbasierter Ansatz verwendet, der die Ausführung der Komponenten auf einem Koprozessor und den Datenfluss steuert. Die Anwendungsentwickler müssen keine Anwendungsbeschleuniger mehr selbst entwickeln. Der Aufwand bei der Entwicklung von Anwendungsbeschleunigern ist so für die Anwendungsentwickler merklich gesenkt. Die Komponenten arbeiten nach dem Datenflussprinzip, um die Wiederverwendung der Daten auf dem Koprozessor zu erhöhen, indem unnötige Kommunikation mit dem externen Speicher vermieden wird. Indem unterschiedliche Knoten im Datenflussgraphen einer Anwendung nebenläufig zueinander ausgeführt werden, wird mehr Parallelität ausnutzbar, sogar wenn Abhängigkeiten zwischen den Knoten vorliegen. Die Komponenten sind dazu im Koprozessor untereinander mittels eines großen Puffersatzes verbunden, der ebenso wie die Komponenten über Mikroprogramme angesteuert wird. Der zentrale Puffersatz löst das Problem des Datentransfers auf Koprozessorseite, das insbesondere bei der Verwendung von Hochsprachen sonst besteht, da nur dann Ein- und Ausgangsdaten vom und zum Host übertragen werden müssen, wenn der FPGA-interne Speicher diese Daten noch nicht beinhaltet oder insgesamt nicht mehr ausreicht.

Dieser komponentenbasierte Bibliotheksansatz bietet klare Verteilung der Aufgaben zwischen Anwendungsentwickler und Hardwarespezialist. Die Entwicklungsdauer lässt sich merklich senken. Zeitgleich lässt sich aufgrund der hohen Datenwiederverwendung auch in datenintensiven Anwendungsgebieten wie dem numerischen Rechnen höherer Nutzen aus der Koprozessorverwendung erzielen.

Veröffentlichungen

Verwendete Veröffentlichungen

[BKNK09] Rainer Buchty, David Kramer, Fabian Nowak, and Wolfgang Karl: A Seamless Virtualization Approach for Transparent Dynamical Function Mapping Targeting Heterogeneous and Reconfigurable Systems. In: *Reconfigurable Computing: Architectures, Tools and Applications*, Band 5453 der Reihe *Lecture Notes in Computer Science*, Seiten 362–367. Springer Berlin Heidelberg, März 2009.

[BN13] Michael Bromberger and Fabian Nowak: Parallel Prefiltering for Accelerating HHblits on the Convey HC-1. In: *Mitteilungen (25. PARS-Workshop)*, Band 30, Seiten 47–57. Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, September 2013.

[BNK14] Michael Bromberger, Fabian Nowak, and Wolfgang Karl: Combined Hardware-Software Multi-Parallel Prefiltering on the Convey HC-1 for Fast Homology Detection. In: *Parallel Computing*, Band 40, Nummer 3. Elsevier, 2014.

[HKN+11] Vincent Heuveline, Wolfgang Karl, Fabian Nowak, Mareike Schmidtobreick, and Florian Wilhelm. Employing a high-level language for porting numerical applications to reconfigurable hardware. In: *Preprint Series of the Engineering Mathematics and Computing Lab (EMCL)*, Band 13. KIT, Karlsruhe, Deutschland, 2011. Fakultät für Mathematik; Institut für Angewandte und Numerische Mathematik.

[KNBK10] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl: Extending a Light-weight Runtime System by Dynamic Instrumentation For Performance Evaluation. In: *ARCS 2010 Workshop Proceedings: Workshop on Parallel Programming and Run-time Management Techniques for Many-core Architectures (2PARMA)*, Seiten 279–284. VDE, Februar 2010.

[KNBK12] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl: Seamlessly portable applications: Managing the diversity of modern heterogeneous systems. In: *Transactions on Architecture and Code Optimization (TACO)*, Band 8, Nummer 4, Seiten 42:1–42:20. ACM New York, NY, USA, Januar 2012. Zugehörige Veröffentlichung zu Konferenzbeitrag auf HiPEAC 2012.

[KVB+09] David Kramer, Thorsten Vogel, Rainer Buchty, Fabian Nowak, and Wolfgang Karl: A general purpose HyperTransport-based Application Accelerator Framework. In: *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA2009)*, Seiten 30–38. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg, Februar 2009.

[NB10] Fabian Nowak and Rainer Buchty: A tightly coupled accelerator infrastructure for exact arithmetics. In: *Architecture of Computing Systems (ARCS '10)*, Band 5974 der Reihe *Lecture Notes in Computer Science*, Seiten 222–233. Springer Berlin Heidelberg, Februar 2010.

[NBK09] Fabian Nowak, Rainer Buchty, and Mario Kicherer: Providing Guidance Information for Application-Mapping on Heterogeneous Parallel Systems. In: *Mitteilungen (22. PARS-Workshop)*, Seiten 115–122. Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, Dezember 2009.

[NBK+13] Fabian Nowak, Ingo Besenfelder, Wolfgang Karl, Mareike Schmidtobreick, and Vincent Heuveline: A Data-driven Approach for Executing the CG Method on Reconfigurable High-Performance Systems. In: *Architecture of Computing Systems (ARCS '13)*, Band 7767 der Reihe *Lecture Notes in Computer Science*, Seiten 171–182. Springer Berlin Heidelberg, Januar 2013.

[NBK14] Fabian Nowak, Michael Bromberger, and Wolfgang Karl: An Architecture Framework for Porting Applications to FPGAs. In: *Mitteilungen (11. PASA-Workshop)*, Band 31, Seiten 61–67. Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, Dezember 2014. Auch vom VDE unter den ARCS 2014 Workshop Proceedings, ISBN 978-3-8007-3579-2, veröffentlicht.

[NBKK09] Fabian Nowak, Rainer Buchty, David Kramer, and Wolfgang Karl: Exploiting the HTX-Board as a Coprocessor for Exact Arithmetics. In: *Proceedings of the First International Workshop on HyperTransport Research and Applications (WHTRA2009)*, Seiten 20–29. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg, Februar 2009.

[NBSK13] Fabian Nowak, Michael Bromberger, Martin Schindewolf, and Wolfgang Karl: Multi-Parallel Prefiltering on the Convey HC-1 for Supporting Homology Detection. In: *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, Seiten 169–174. ACM, September 2013. International Workshop on Parallelism in Bioinformatics (PBio 2013).

[NKBK10] Fabian Nowak, Mario Kicherer, Rainer Buchty, and Wolfgang Karl: Delivering Guidance Information in Heterogeneous Systems. In: *Mitteilungen (23. PARS-Workshop)*, Seiten 84–90. Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, 2010. Auch vom VDE unter den ARCS 2010 Workshop Proceedings, ISBN 978-3-8007-3222-7, auf Seiten 95–102 veröffentlicht.

[Now14] Fabian Nowak: Evaluating the Energy Efficiency of Reconfigurable Computing Toward Heterogeneous Multi-Core Computing. In: *Mitteilungen (11. PASA-Workshop)*, Band 31, Seiten 73–78. Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen, Dezember 2014. Auch vom VDE unter den ARCS 2014 Workshop Proceedings, ISBN 978-3-8007-3579-2, veröffentlicht.

Weitere Veröffentlichungen

[GNRH05] Henning Groenda, Fabian Nowak, Patrick Rössler, and Uwe D. Hanebeck: Telepresence Techniques for Controlling Avatar Motion in First Person Games. In: *Intelligent Technologies for Interactive Entertainment, First International Conference (INTETAIN '05)*, Band 3814 der Reihe *Lecture Notes in Computer Science*, Seiten 44–53. Springer Berlin Heidelberg, 2005.

[MNBK09] Oliver Mattes, Fabian Nowak, Rainer Buchty, and Wolfgang Karl: Augmenting the Curriculum targeting Hardware-aware System Design. In: *CDNLive! EMEA 2009*, Seite 66. Cadence Design Systems, Inc., Mai 2009.

[NBK08a] Fabian Nowak, Rainer Buchty, and Wolfgang Karl: A Run-time Reconfigurable Cache Architecture. In: *Parallel Computing: Architectures, Algorithms and Applications (ParCo 2007)*, Band 15 der Reihe *Advances in Parallel Computing*, Seiten 757–766. IOS Press, 2008. Veröffentlichung zum gleichnamigen Workshop.

[NBK08b] Fabian Nowak, Rainer Buchty, and Wolfgang Karl: Adaptive cache infrastructure: supporting dynamic program changes following dynamic program behavior. In: *Proceedings of the 9th Workshop on Parallel Systems and Algorithms (PASA 2008)*, Band 124 der Reihe *Lecture Notes in Informatics*, Seiten 59–68. Gesellschaft für Informatik, Februar 2008.

[TKN+08] Jie Tao, Marcel Kunze, Fabian Nowak, Rainer Buchty, and Wolfgang Karl: Performance Advantage of Reconfigurable Cache Design on Multicore Processor Systems. In: *International Journal of Parallel Programming*, Band 36, Nummer 3, Seiten 347–360. Springer US, Juni 2008.

Studentenarbeiten

Betreute Studentenarbeiten mit Beitrag zu dieser Arbeit

[Ahu09] Bernd Ahues. Managing prepartitioned Hardware Resources dynamically. Diplomarbeit, Universität Karlsruhe, Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Karlsruhe, Deutschland, 30. September 2009.

[Bes12] Ingo Besenfelder. Evaluation moderner FPGA-basierter Koprozessoren für numerische Algorithmen. Diplomarbeit, Karlsruher Institut für Technologie, Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Karlsruhe, Deutschland, 29. Juni 2012.

[Bro10] Michael Bromberger. Entwicklung einer mikroprogrammierbaren Koprozessorsteuerung. Studienarbeit, Karlsruher Institut für Technologie, Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Karlsruhe, Deutschland, 7. Oktober 2010.

[Bro13] Michael Bromberger. Beschleunigung der Homologie-Suche in Proteindatenbanken mithilfe der Convey HC-1. Diplomarbeit, Karlsruher Institut für Technologie, Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Karlsruhe, Deutschland, 31. März 2013.

[Che09] Jiefei Chen. Verwendung von ELF zur Unterstützung heterogener Programmausführung. Studienarbeit, Universität Karlsruhe (TH), Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Karlsruhe, Deutschland, 14. September 2009.

[Ham10] Andreas Hampp. Mikroprogrammierte Steuerung einer Einheit für exakte Arithmetik. Studienarbeit, Karlsruher Institut für Technologie, Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Karlsruhe, Deutschland, 7. Oktober 2010.

[Kna12] Jonathan Knam. Arithmetisch exakte Operationen auf der Convey HC-1. Bachelorarbeit, Karlsruher Institut für Technologie, Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Karlsruhe, Deutschland, 14. Dezember 2012.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Der Bauingenieur Konrad Zuse war es leid, numerische Berechnungen von Hand durchführen zu müssen, und entwickelte eine Maschine, welche ihm die Berechnungen durchführen sollte. Rechner werden seitdem in einer Vielzahl an Bereichen genutzt, angefangen bei der Berechnung von großen linearen Gleichungssystemen über das numerische Lösen von Differentialgleichungen, wie sie in vielen Ingenieurwissenschaften vorkommen, hin zu vollständigen Simulationen beispielsweise der Belastungssituationen von Deichen, die in Katastrophensituationen benötigt werden. Immer wichtiger wird auch die Bioinformatik, welche sich unter anderem mit Problemstellungen der Genforschung und Medikamentenentwicklung beschäftigt. Denn aufgrund der zunehmenden Anzahl an Genen, Proteinen, Merkmalen und Wirkstoffen erhöht sich der Rechenbedarf fortlaufend. Sollen die Simulationen weiterhin an Genauigkeit gewinnen unter Einbeziehen verbesserter, detaillierterer Umgebungsmodelle bei gleichzeitigem Einhalten der für den Katastrophenschutz benötigten Zeitschranken oder unter Einbeziehung neuer Gensequenzen und Wirkstoffe, so wird mehr Rechenleistung benötigt. Zwar nimmt die Rechenleistung unserer Rechner – im Kleinen zuhause wie auch im Großen auf Großrechnern – beständig zu, allerdings stellen die Energieversorgung sowie der Abwärmetransport ein zunehmendes Problem dar. Aufgrund der aktuellen Entwicklung bei den Strukturbreiten nehmen die Leckströme zu, und noch immer wird Leistungssteigerung durch Frequenzsteigerung erreicht und damit einhergehend die Leistungsaufnahme erhöht, da die Taktfrequenz proportional in die gesamte Leistungsaufnahme eingeht. Der Einsatz von Spezialeinheiten ist ein von eingebetteten Systemen her bekannter, bewährter und vielversprechender Ansatz, Operationen schneller, genauer oder energieeffizienter[1] durchzuführen. Solche Spezialeinheiten werden als Akzeleratoren[2] bezeichnet und kommen häufig als Koprozessoren[3] zum Einsatz, indem sie über einen Bus mit dem Prozessor verbunden werden. In dieser Arbeit werden vor allem Field-Programmable Gate Arrays (FPGAs) als Akzeleratoren betrachtet.

Vorteile von FPGAs sind die Möglichkeiten der Datenmanipulation auf Bitebene und der massiven internen Parallelität, wodurch sehr viele Bitoperationen parallel ausführbar sind. FPGAs stellen gerade für bioinformatische Anwendungen eine vielversprechende Beschleunigerkomponente dar, da Leistungssteigerungen bis zu Faktor 200 erwartbar sind aufgrund der für die Verarbeitung in Standardmikroprozessoren häufig ungeeigneten Datenstrukturen sowie massiver Parallelität in den Algorithmen. FPGAs werden auch anwendungsorientiert eingesetzt, zum Beispiel zur Speicherung und zum Vergleich von medizinischen Bildaufnahmen [59]. Auf rekonfigurierbaren Knoten oder Einheiten ausgeführte Funktionen versprechen nicht nur, schneller ausgeführt zu werden, sondern aufgrund der zehnfach geringeren Taktrate rekonfigurierbarer Logik auch nur ein Zehntel der Leistung aufzunehmen. Das Ziel ist, durch die Erweiterung von herkömmlichen Mehrkernsystemen um rekonfigurierbare Akzeleratoren zusätzliche Rechenleistung und möglicherweise auch Energieeinsparungen dynamisch erhalten zu können. Bei Bedarf soll die laufende Anwendung unterstützt werden können, beispielsweise eine zeitkritische Hochwassersimulation. So wird das Problem der Leistungsaufnahme zukünftiger Exascale-Systeme adressiert und zugleich Leistungssteigerung gewonnen.

Ein weiterer, entscheidender Vorteil von FPGAs liegt nämlich darin, dass Algorithmusimplementierungen zur Laufzeit gegen andere Implementierungen ausgetauscht werden können. Um eine

[1] z.B. bewertet anhand des Energy-Delay-Products (EDP) [114]: Produkt von Energie und Verzögerung
[2] lat. accelerare = beschleunigen
[3] lat. cum+procedere = zusammen voranschreiten

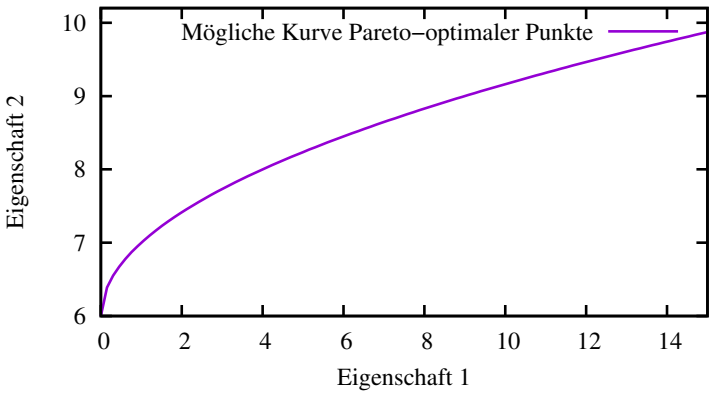


Abbildung 1.1: Effizienz anhand von Pareto-Optimalität festgemacht.

Anwendung maximal zu beschleunigen, muss möglichst jeder Teil einer Anwendung durch Hardwareschaltungen unterstützt werden. Da es schlichtweg unmöglich ist, dass jede erdenkliche Funktion für die benötigte Schaltung einen eigenen Teil der Chipfläche besäße, stellen FPGAs bzw. rekonfigurierbare Logik also eine geeignete Möglichkeit dar, solche Schaltungen zur Laufzeit entsprechend den Anwendungsanforderungen zur Verfügung zu stellen.

Hochleistungsrechensysteme mit Mehrkernprozessoren werden häufig um FPGAs ergänzt zu heterogenen, beschleuniger-erweiterten Systemen. Die Verlagerung von Anwendungsteilen in die FPGA-Hardware wird zumeist von Hardwarespezialisten mittels einer Hardwarebeschreibungssprache vorgenommen, denn die hardwareorientierte Entwicklung und dadurch nutzenbringende Ausnutzung der Hardwarestrukturen von FPGAs ist eine große Herausforderung.

Die Effizienz im Allgemeinen lässt sich anhand des (annähernden) Erreichens Pareto-optimaler Werte festmachen, wie in Abbildung 1.1 exemplarisch für zwei beliebige Eigenschaften skizziert. Werte oberhalb (oder unterhalb, je nach Auslegung) der Kuve sind nicht möglich; je geringer der Abstand zur Kurve ist, umso effizienter; unterschiedliche Punkte auf der Kurve sind gleichermaßen als effizient einzustufen und somit Pareto-optimal. Als Eigenschaften werden hier die Verhältnisse der Ausführungsdauern im Kontext der Gesamtanwendung und der benötigten Entwicklungs- bzw. Integrationsdauern genommen. Die Energieeffizienz im Besonderen kann anhand des Energy-Delay-Products (EDP) [114] definiert werden. Bei einem „energieeffizienten“ heterogenen System übertrifft die erzielbare positive Steigerung der Rechenleistung die Steigerung der Leistungsaufnahme (Power-Delay-Product (PDP)) bzw. des EDPs gegenüber der nicht beschleunigten Ausführung. Der Begriff wird im Laufe des Kapitels in Abschnitt 2.1.6 mittels Abbildung 2.11 genauer festgemacht. Bei abweichender Verwendung der Begrifflichkeiten wird die Verwendung explizit geklärt.

Damit auch Anwender anstelle von Hardwarespezialisten rekonfigurierbare Hardware programmieren können, existieren hochsprachliche Ansätze. Die Programmierung von Graphics-Processing Units (GPUs) zum Zwecke des Allzweckrechnens musste zunächst ebenso hardwarenah über die Grafikprogrammierungsroutinen aus OpenGL oder DirectX erfolgen [227][4]. Für die Programmierung von Grafikkarten haben sich im Verlauf des letzten Jahrzehnts bequemere und leistungsfähigere Programmiermodelle entwickelt und etabliert, und im Umfeld der FPGA-Entwicklung werden eben Werkzeuge zur Konvertierung von hochsprachlichem Anwendungscode zu Schaltungsbeschreibungen von FPGAs angeboten, um die Entwicklung zu erleichtern und zu beschleunigen. Dabei entfällt die mühsame Entwicklung von Hardwareschaltungen weitgehend. Zahlreiche Anwendungen sind so bereits erfolgreich auf rekonfigurierbare Hardware portiert worden. Aus der hochsprachlichen Formulierung wird in einem Zwischenschritt automatisiert eine Hardwarebeschreibung erzeugt. Aus jeder Beschreibung muss zeitaufwendig eine FPGA-Schaltung synthetisiert werden, was bei der Entwurfsraumexploration und bei der Fehlersuche hinderlich ist und die effiziente Entwicklung deutlich erschwert.

[4] Rumpf und Strzodka haben die Implementierung der mathematischen Funktionen in Grafikhardware als Aufgabe des Informatikers betrachtet und die Zuständigkeit des Mathematikers auf das Verwenden der Funktionalitätsimplementierungen abstrahiert [249].

Die Verwendung der Hochsprachenkonverter erfordert jedoch weiterhin sehr tiefgehendes Verständnis der zu verwendenden Hardware und dient daher mehr den Hardwareexperten zur schnelleren Schaltungsentwicklung als den Anwendungsexperten, da zwar nicht mehr zwangsweise hardwarenah, aber noch immer hardwareorientiert entwickelt werden muss. Wesentliche Schwierigkeiten stellen die Datenübertragung an sich und ihre Optimierung dar, die mit der intern parallelen Ausführung abgestimmt sein müssen. Die nötige Ausnutzung der Datenlokalität ist mit Hochsprachen daher nur schwer erreichbar, und zum effizienten Einsatz benötigt man gleichermaßen umfassendes Verständnis von Hardware wie beim manuellen Entwurf. Zudem wird die interne Ablaufsteuerung meistens mit Zustandsautomaten umgesetzt. Diese bedingen für jede neue FPGA-Konfiguration neue Synthesen, wodurch der zeitliche Entwicklungsaufwand wieder erhöht wird. Oftmals reicht der Anteil an ausnutzbarer Datenparallelität nicht aus, die wesentlich geringere Taktrate der FPGAs gegenüber gewöhnlichen Mehrkernprozessoren zu kompensieren, so dass sich der hohe Aufwand für den Anwender gar nicht mehr lohnt. In diesem Kontext entsteht unmittelbar die Frage, ob es sich für den Anwendungsexperten nicht eher lohnt, spezifische Hardware des verwendeten Allzweckprozessors, wie etwa SIMD-Einheiten, auszunutzen. Daher sind Methoden zu finden und Werkzeuge zu entwickeln, welche den Einsatz von FPGAs für den Anwender attraktiver machen als die Nutzung von Standardprozessoren.

Deutliche Beschleunigung durch die Verlagerung von Software in Hardware lässt sich in der Bioinformatik erzielen. Diese Arbeit beschreibt daher, welche hardwarespezifischen Umformulierungen des Algorithmus nötig sind, um die erste Vorfilterungsstufe der bioinformatischen Anwendung HHblits [222] mittels rekonfigurierbarer Hardware zu beschleunigen. Die Umformulierung des Smith-Waterman-Algorithmus nach Farrar [96] erlaubt bereits, 128 gleiche Spezialoperationen zur Berechnung der Matrix intern parallel zueinander durchzuführen. Erst mit weiteren Umformulierungen sind problemangepasste Speicherstrukturen zur Erhöhung der Datenlokalität möglich, wodurch die Anzahl externer Speicherzugriffe reduziert wird. Damit können dann zwölf Sequenzvergleiche gleichzeitig ausgeführt werden. Zusätzlich sind besondere Reduktionsschaltungen implementiert. Beide implementierten Spezialoperationen existieren in gewöhnlichen Mehrkernprozessoren nicht. Durch die massive Parallelität und die schnelle Bereitstellung von ausreichend vielen Daten wird die geringere Taktrate der FPGAs kompensiert.

Für numerische Verfahren bietet sich das Skalarprodukt als zu verlagernde Spezialoperation an. Die Durchführung des Skalarprodukts in exakter Arithmetik kann aufwendige Reorthogonalisierungen verhindern oder allgemein die Stabilität sowie die Konvergenz verbessern. Die Notwendigkeit zur Verwendung einer exakten Skalarproduktimplementierung sieht man im Rahmen dieser Dissertation am Konjugierte-Gradienten-Verfahren sowie am Lanczos-Verfahren. Ein exaktes Skalarprodukt ist in Software sehr aufwendig, kann aber effizient[5] in FPGAs implementiert werden aufgrund der Verwendung von Bitoperationen. Um Beschleunigung aus dem exakten Skalarprodukt in rekonfigurierbarer Hardware zu erzielen, muss jedoch zunächst das Problem der Datenlokalität gelöst werden. Diese Lösung muss die koprozessorinterne Datenhaltung gleichermaßen wie den effizienten[6] und nahtlosen Datenaustausch mit dem umgebenden Mehrkernsystem angehen.

Eine diesbezüglich zu betrachtende Fragestellung ist daher, wie die Hardware-Software-Schnittstelle zu FPGA-basierenden Akzeleratoren und die Verwendung selbiger für Programmierer von Anwendungen für heterogene Mehrkernsysteme aussehen muss, wenn einerseits der Anwendungsentwickler von der hardwarenahen Programmierung befreit oder zumindest entlastet werden soll und gleichzeitig aber Vorteile durch die Nutzung des Akzelerators erzielbar sein sollen. Das dazu entstehende Programmiermodell muss zudem erlauben, die entstandenen Anwendungen innerhalb eines gewissen Rahmens für FPGAs portabel zu halten. Dabei ist die Integration in aktuelle Mehrkern- und künftige Vielkernsysteme als Randbedingung zu beachten. Beispielsweise verfügt bereits die RAW-Architektur [275] als Mehrkernprozessorarchitektur über rekonfigurierbare Logik, Intels Atom E6x5C[7] mit zwei Prozessorkernen birgt einen FPGA von Altera zur Auslagerung rechenintensiver Anwendungsteile, und es ist davon auszugehen, dass in zukünftigen gitterbasierten Vielkernprozessoren einige Knoten keine Allzweckprozessoren sein werden, sondern viel mehr Spezialeinheiten und rekonfigurierbare Logik [197]. Nur in einem Verbund zusammen mit effizienten[6] Schnittstellen und Programmiermodellen ist es denkbar, beschleuniger-erweiterte Exascale-Systeme eines Tages erfolgreich betreiben zu können [86].

[5] im Hinblick auf Ressourcennutzung und erzielbare Leistungssteigerung

[6] unnötigen Mehraufwand vermeidend

[7] URL: `http://www.altera.com/devices/processor/intel/e6xx/proc-e6x5c.html`

In der Regel bestehen unterschiedliche Anwendungen aus unterschiedlichen Funktionen, und zudem besitzen sie unterschiedliche Aufrufhäufigkeiten dieser Funktionen. Das Verwenden eines einzelnen anwendungsspezifischen Akzelerators ist für andere als die ursprünglich beschleunigte Anwendung dementsprechend nur bedingt sinnvoll. Daher sind weitere Funktionsverlagerungen auf den Akzelerator nötig, um die restlichen Funktionen zu unterstützen. Die bestmögliche Ausnutzung von Beschleunigerhardware bedingt auch die zeitlich überlappte Ausführung unterschiedlicher Funktionalitäten. Für jede Anwendung stellt wiederum eine andere Komposition der feingranulareren Akzeleratoreinheiten zu einem neuen anwendungsspezifischen Akzelerator die bestmögliche Lösung dar. Um mehrere Anwendungen aus einer gemeinsamen Anwendungsdomäne unterstützen zu können, müssen daher mehrere feingranulare Akzeleratoreinheiten gleichzeitig vorhanden und geeignet miteinander verbunden sein.

Der dieser Disseration zugrunde liegende Ansatz zur Lösung beider oben genannter Probleme ist die komponentenbasierte Programmierung und Verwendung von rekonfigurierbarer Hardware in Mehrkernsystemen. Dies ermöglicht auch die automatische Beachtung der Datenlokalität. Für sowohl den Prozessor als auch den Koprozessor können Hardware- und Softwareentwickler domänenspezifische Komponenten bereitstellen. Die Komponenten müssen dann in ihrer Domäne häufig wiederverwendbar sein, damit der Syntheseaufwand für die rekonfigurierbaren Akzeleratoren entfällt. Die Anwendungsentwickler sollen weder Hochsprachen zur Entwicklung von Anwendungsbeschleunigern verwenden noch sollen die Komponenten über Zustandsautomaten ansgeteuert werden, wie sie von Quelltextkonvertern erzeugt werden. Ein mikroprogrammbasierter Ansatz wird daher eingesetzt, um die Ausführung der Komponenten auf einem Koprozessor und den Datenfluss zu steuern. Die Entwicklung eines Mikroprogramms kann dann in späteren Schritten aufgrund der datenflussorientierten Ansteuerung und Ausführung mit einer grafischen Schnittstelle wesentlich vereinfacht werden. Der Aufwand bei der Entwicklung von Anwendungsbeschleunigern sollte so für die Anwendungsentwickler spürbar gesenkt sein. Denn durch die Mikroprogrammansteuerung des Koprozessors entfällt auch der Konfigurationsaufwand zur Laufzeit. Die Komponenten arbeiten nach dem Datenflussprinzip, um die Wiederverwendung der Daten auf dem Koprozessor zu erhöhen, indem unnötiges Rückschreiben in den und Neueinlesen aus dem externen Speicher vermieden wird, und um bei Ausführung in Software der Verdrängung der Daten aus den Prozessorcaches zuvorzukommen. Wenn unterschiedliche Knoten im Datenflussgraphen einer Anwendung nebenläufig zueinander ausgeführt werden, ist mehr Parallelität ausnutzbar, sogar wenn Abhängigkeiten zwischen den Knoten vorliegen. Die Komponenten sind dazu im Koprozessor untereinander geeignet zu verbinden. Ein zentraler Puffersatz löst das Problem des Datentransfers auf Koprozessorseite, das insbesondere bei der Verwendung von Hochsprachen sonst besteht, da nur noch Ein- und Ausgabedaten vom und zum Host übertragen werden müssen. Mit einem solchen komponentenbasierten Bibliotheksansatz mit klarer Verteilung der Aufgaben zwischen Anwendungsentwickler und Hardwarespezialist lassen sich bei zeitgleich höherem Nutzen der Koprozessorverwendung die Kosten der Entwicklung senken, denn die Entwurfsraumexploration ist schneller und bequemer durchführbar aufgrund der entfallenden Synthesezeiten.

Kapitel 2

Verwendung von rekonfigurierbarer Beschleunigerhardware zur Verlagerung von Algorithmen

Gemäß Moore's Law verdoppelt sich die Anzahl an Transistoren pro Chip bei gleichbleibender Chipfläche etwa alle 18 Monate[1]. Über die letzten 40 Jahre hinweg wurden Möglichkeiten der Nutzung der zusätzlichen Transistoren ausgiebig erforscht, geprüft und mitunter etabliert. Beispiele dafür sind etwa die Superskalartechnik, welche Parallelität auf Befehlsebene bei voneinander unabhängigen Befehlen auszunutzen vermag, oder auch die simultan mehrfädige Ausführung (Simultaneous Multi-Threading (SMT)), wobei die internen Funktionseinheiten abwechselnd mit Befehlen und Daten aus verschiedenen Threads versorgt werden, und so von außen betrachtet mehrere Threads parallel verarbeitet werden. Die Taktfrequenzen können jedoch nicht im gleichen Rahmen steigen wie die Transistoranzahl. Auch der Abwärmetransport vom Chip weg gestaltet sich problematisch, denn die Leistungsaufnahme nimmt beständig zu. Die gestiegene Taktfrequenz geht kubisch in die Leistungsaufnahme ein, und aufgrund der geringen Strukturgrößen haben die Leckströme mittlerweile einen beachtlichen Anteil an der gesamten Aufnahme gewonnen. Bis zu 50 % der Chipfläche eines Prozessors werden zudem für Caches verwendet, weil durch die damit erhaltene Erhöhung der Datenlokalität wesentlich mehr Leistungsgewinn erwartbar ist als durch andere Techniken.

Aufgrund der zunehmend geringer ausfallenden Leistungssteigerungen bei Verbesserungen auf Mikroarchitekturebene haben sich daher seit etwa Mitte der 2000er Jahre Mehrkernprozessoren gegenüber den Einkernprozessoren etabliert. Mehrkernprozessoren liefern ein Vielfaches der Leistung durch eine höhere Anzahl an Kernen anstatt durch rechnerarchitektonische Änderungen. So können sie die Forderung der Verbraucher nach beständiger Leistungserhöhung weiterhin erfüllen. Durch die Erhöhung der Kernanzahl kann Leistungssteigerung jedoch auch nur in einem begrenzten Maß erreicht werden.

Spezialhardware wie SIMD Streaming Extensions (SSE), Grafikkarten mit General-Purpose Graphics-Processing Units (GPGPUs), Gleitkommakoprozessoren und schlussendlich auch rekonfigurierbare Logik bieten gegenüber Standardmikroprozessoren hohe Leistungssteigerungen für verschiedene Algorithmen bei gleichzeitig geringerem Energiebedarf (aber eventuell kurzfristig höherer Leistungsaufnahme). Entsprechend wird Spezialhardware bereits seit vielen Jahren in eingebetteten Systemen eingesetzt, um sehr dedizierte Aufgaben äußerst effizient, also ressourcenschonend und schneller, zu erfüllen. Seit einigen Jahren wird sie zum Zwecke der Beschleunigung und Erhöhung der Energieeffizienz auch in Rechensysteme, die auf Mehrkernprozessoren basieren und im Umfeld des wissenschaftlichen oder technischen Rechnens zum Einsatz kommen, integriert.

Solche Systeme, die aus einem oder mehreren Mikroprozessoren und mindestens einer Beschleunigereinheit bestehen, werden als „heterogene Systeme“ bezeichnet. Sie zeichnen sich unter anderem dadurch aus, dass unterschiedliche Befehlssatzarchitekturen auf den unterschiedlichen Einheiten zum Einsatz kommen. Aufgrund ihrer besonderen Hardwareeigenschaften stellen heterogene Systeme eine Möglichkeit dar, mit geringem Kostenaufwand dynamisch ein hoch leistungsfähiges, aber dennoch energieeffizientes System zur Verarbeitung spezieller Aufgaben zu erhalten. Es wurden bereits Untersuchungen hinsichtlich der Eignung rekonfigurierbarer Logik zum Einsatz in heterogenen Systemen durchgeführt [120].

[1] zunächst nur für 24 Monate vorhergesagt, später korrigiert auf 18 Monate

Kritisch beim Einsatz rekonfigurierbarer Logik ist die Entwicklung effizienter, d.h. leistungsteigernder, Programme bzw. Schaltungen bei FPGAs und die bezüglich der Vermeidung unnötigen Mehraufwands effiziente Integration in die Hostsysteme. Zur Entwicklung von FPGA-Schaltungen werden Hardwarebeschreibungssprachen wie VHDL eingesetzt und mit diesen manuell die Schaltungen spezifiziert. Eine hinsichtlich dieser Aspekte effiziente Implementierung ist sehr schwer zu erreichen. Dazu sind Kenntnisse im Entwurf von Schaltungen, in Hardwareeigenschaften und FPGA-Spezifika nötig. Neben dem manuellen Entwurf existieren auch Werkzeuge, die aus hochsprachlichen Beschreibungen dann Schaltungen für FPGAs erstellen. Damit ist jedoch noch keine effiziente Anbindung an den Prozessor oder an das umgebende System gegeben. Die Hardwareschaltungen werden von den Hochsprachen zugehörigen Quelltextkonvertern mittels formaler Transformationen erstellt und enthalten daher sehr viele Zustandsautomaten. Ein Großteil des nötigen Aufwands wird bei der Hardwareentwicklung daher von den langen Synthesezeiten für die komplexen Beschreibungen verursacht. Die Synthesezeiten werden mit hochsprachlichen Ansätzen also nicht reduziert. Die Programmierung heterogener Systeme, insbesondere mit FPGAs als Beschleunigerhardware, ist dementsprechend zu aufwendig, so dass sie sich für Anwendungsexperten nur dann lohnt, wenn sie mit einem Programmier- und Ausführungsmodell ausgestattet werden, welches die bequeme Programmierung des Akzelerators und die nahtlose Integration in das bestehende System bietet.

Es müssen also Konzepte und Methoden untersucht werden, welche die beschleunigte Ausführung von Anwendungsteilen durch die Nutzung rekonfigurierbarer Hardware schneller und bequemer für Anwendungsentwickler zugänglich machen, damit zukünftige Systeme über die enge Einbindung von rekonfigurierbarer Hardware anwendungs- und bedarfsorientiert Leistungssteigerungen bieten können. Die vorliegende Dissertation untersucht daher einige heterogene Systeme mit unterschiedlichen Anbindungen von FPGA-basierenden Akzeleratoren als Koprozessoren. Das entwickelte, portierbare und adaptierbare Rahmenwerk zur Nutzung von FPGAs als Akzeleratoren beruht auf dem komponentenbasierten Programmiermodell und setzt Mikroprogrammierung zur Ansteuerung und Verschaltung der Komponenten ein. Dieses Rahmenwerk ist gleichermaßen in Hardware wie in Software ausführbar und ermöglicht damit die schrittweise Verlagerung von Funktionalität aus dem Mikroprozessor hinaus auf die rekonfigurierbare Hardware. Die Komponenten werden von Hardwarespezialisten in hardwarenahen Sprachen entwickelt.

Dieses Kapitel beschreibt daher die eingangs genannten heterogenen Systeme, die nutzbaren Programmiermodelle und die auftretenden Probleme, wenn Algorithmen in rekonfigurierbare Akzeleratorhardware verlagert werden sollen. Da wesentliche, in der Dissertation verwendete Komponenten digitale Arithmetik implementieren, werden Aspekte bei der Nutzung von rekonfigurierbarer Hardware für arithmetische Berechnungen und potentiell gewinnbringende Ansätze zur Implementierung aufgeführt. Ebenso wird kurz die Idee der Mikroprogrammierung beschrieben.

2.1 Heterogene Systeme

Als *heterogen* bezeichnet man solche Systeme, die eine weitere Verarbeitungseinheit mit einer anderen Befehlssatzarchitektur[2] in das System integrieren und nahtloses Umschwenken zwischen den Verarbeitungseinheiten erlauben. Das Ziel solcher Systeme ist, durch die Ausführung bestimmter Anwendungsteile auf der zusätzlichen Einheit Einsparungen in der Ausführungsdauer oder auch in der Leistungsaufnahme zu erhalten. Heterogene Systeme können auch aus mehr als nur zwei Verarbeitungseinheiten mit unterschiedlichen ISAs bestehen (Abb. 2.1). Im Folgenden werden Grafikkarten und FPGAs als wesentliche Möglichkeiten zur Anwendungsbeschleunigung sowie einige Ansätze zu Aufbau, Entwicklung und Nutzung heterogener Systeme vorgestellt, mit Unterscheidung zwischen Systems on Chip (SoCs) und busbasierten Systemen. Der Fokus liegt vor allem auf der Integration rekonfigurierbarer Logik in ein System mit Mehrkernrozessor.

2.1.1 Grafikkarten und -Prozessoren

Zur Darstellung von Text und Bildern wurden bereits frühzeitig dedizierte Grafikchips über einen Prozessor- oder Systembus an die CPU angebunden, so dass die CPU entlastet werde von der Auf-

[2] engl. Instruction Set Architectures (ISAs)

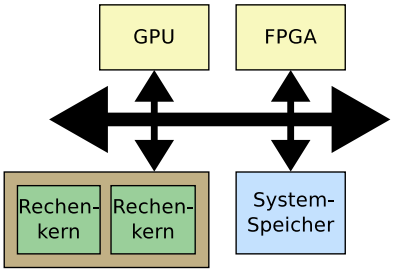


Abbildung 2.1: Heterogenes System mit CPU, GPU und FPGA.

gabe der Grafikdarstellung. Aufgabe von Grafikkarten ist es, modellierte Welten auf die verfügbaren Pixel des Monitors des Benutzers abzubilden. Dabei muss das vorgegebene 3D-Modell in mehreren Schritten transformiert werden, was im Wesentlichen die Polygonerzeugung beinhaltet, das Versehen der Polygone mit Texturen und Farben sowie abschließend die Belichtung mit indirektem und direktem Licht. Es entsteht somit für jeden Pixel eine Sequenz an auszuführenden Operationen, um den endgültigen Farbwert zu errechnen, die sogenannte Grafikpipeline. Dies lässt sich parallelisieren, indem mehrere Rechenkerne jeweils nur einen Teil des Bilds berechnen, oder alternativ die Rechenkerne unterschiedliche benötigte Operationen ausführen und die berechneten Daten untereinander weiterreichen (Pipelining). Es wurden immer mehr Recheneinheiten nötig, um für anspruchsvollere Belichtungsverhältnisse, Reflektionen, höhere Auflösungen, kompliziertere Modelle usw. die benötigte Rechenleistung zur Verfügung stellen zu können.

Im Jahre 1999 waren Grafikchips bereits programmierbar geworden. Zunächst mussten OpenGL und Cg zur Programmierung verwendet werden, was im Wesentlichen grafikorientierte Sprachen sind, bevor Brook/BrookGPU[3]/Brook+ [2], Stream [1] und Compute Unified Device Architecture (CUDA)[4] [131, 206] von Akademia und Industrie entwickelt wurden.

Die Verwendung von programmierbaren Grafikrecheneinheiten zum Lösen nicht-grafischer Probleme wurde von Rumpf und Strzodka begonnen [227]. Motiviert über die verfügbare Speicherbandbreite, die mögliche Anzahl parallel ausführbarer Vektoroperationen und die bereits vorab erzielte immense Leistungssteigerung bei Bildoperationen verwendeten sie Festkommaarithmetik und die OpenGL-API zum Lösen von Hitzeverteilungsproblemen mittels des Jacobilösers und des CG-Verfahrens (vgl. Abschnitt 5.8.2). Die Berechnung auf der Grafikkarte erfolgte wesentlich schneller, dafür war allerdings auch die Genauigkeit auf festgelegte Intervalle $[-\rho, \rho]$ mit acht Bit Auflösung beschränkt, während die CPU mindestens mit einfach genauer Gleitkommaarithmetik rechnete. Ferner wurden die Datenübertragungszeiten sowie die nötigen Konvertierungen in andere Formate und Datenstrukturen nicht berücksichtigt. Dieser Zustand hält bis heute an: Die Daten müssen vorab der Nutzung modifiziert und umstrukturiert werden, dann werden sie erst zur Grafikkarte übertragen und abschließend zurückgeschrieben. Genau diesem Ausführungsmodell versucht die vorliegende Dissertation beizukommen.

Stratton et al. haben untersucht, was essentiell beim Allzweckrechnen auf Grafikkarten ist [248]. Da die sieben von ihnen extrahierten Techniken nicht vollständig orthogonal zueinander sind, können sie keine konkrete Anleitung geben. Damit bleibt die GPU-Programmierung ein langwieriger Prozess mit abwechselndem Messen der Kernellaufzeit und Code-Veränderungen, um sukzessive die unterschiedlichen Techniken einzubringen. Dieser Prozess muss für jede neue, zu beschleunigende Anwendung wiederholt werden. Um Anwendungsentwickler und Anwender bei der Verwendung rekonfigurierbarer Logik zu unterstützen, muss eine Lösung gefunden werden, wie rekonfigurierbare Logik einfach und effizient programmierbar ist, so dass ohne diesen langwierigen, mühsamen Prozess Leistungssteigerung erzielbar ist.

2.1.2 Field-Programmable Gate Arrays

Neben GPGPUs bieten FPGAs herausragendes Leistungspotential [124], das sie vornehmlich über massiv datenparallele Ausführung erlangen, aber nicht ausschließlich, da beispielsweise für den

[3]URL: `http://graphics.stanford.edu/projects/brookgpu/`
[4]URL: `http://www.nvidia.com/cuda/`

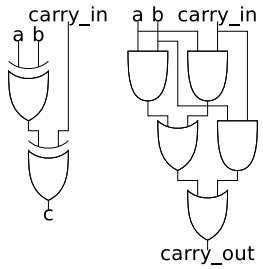


Abbildung 2.2: Verschaltung mehrerer Gatter zur Umsetzung eines Volladdierers.

a	b	carry_in	c	carry_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabelle 2.1: Belegung einer Look-Up Table für einen Volladdierer.

Kontrollfluss ausschlaggebende Berechnungen gleichzeitig durchgeführt werden können sowie nahezu beliebig breite Datentypen und Operationen auf ihnen implementierbar sind.

FPGAs wurden etwa im Jahr 1984 erfunden von Ross Freeman und Bernard Vonderschmitt. Ihr erstes Produkt war der Xilinx XC2064, der im November 1985 auf den Markt kam. Mit der XC3000-Serie begann dann die erfolgreiche Verbreitung von FPGAs [103]. Weitere Hersteller kamen entsprechend im Laufe der 80er Jahre hinzu, und so gibt es mittlerweile eine ganze Reihe an Herstellern von FPGAs, die sich durch unterschiedliche Produkt- sowie Leistungsmerkmale voneinander abgrenzen. Eine ausführliche Darstellung in Bezug auf Konfigurationsspeicherung, Kontrollprozessoren, Rekonfigurierbarkeit u.ä. bieten Fons et al. in ihrer Arbeit [99] bezüglich Kontrollstrukturen zur Ausnutzung partieller dynamischer Rekonfiguration. Programmiert werden FPGAs mit Harwarebeschreibungssprachen wie Verilog oder VHDL oder einigen wenigen weiteren. Aus den Beschreibungen erzeugen, wie später beschrieben werden wird, Synthesewerkzeuge die exakten Konfigurationen für die einzelnen Bestandteile der FPGAs.

Aufbau und Technologie von FPGAs

FPGAs sind gitterartig aufgebaut. An jedem Knoten befindet sich mindestens ein Element, wobei die Gitterlinien durchaus mehrere Signalleitungen umfassen können wie bspw. eine Datensignalleitung und eine für ein Taktsignal. Schaltmatrizen[5] an den Knoten sind dafür zuständig, die Ausgangssignale eines Elements weiterzureichen an das Zielelement. Zum Weiterreichen gibt es spezifische Verbindungsleitungen innerhalb der Nachbarschaft (short wires) und solche, die Verbindungen über größere Distanzen ermöglichen (long wires).

Look-Up Tables (LUTs) sind ebenso Elemente an den Gitterknoten. Sie sind programmierbare Logiktabellen, mithilfe derer sich einfache bool'sche Funktionen auf den Eingangssignalen realisieren lassen [102]. Beispielsweise kann anstatt aus mehreren zusammengeschalteten (Abb. 2.2) Gattern ein 1-Bit-Volladdierer in zwei LUTs einbeschrieben sein mit drei Eingängen a, b und $carry\_in$ sowie zwei Ausgängen c und $carry\_out$, wie Tabelle 2.1 veranschaulicht. Mehrere Look-Up Tables (LUTs) können in einem Configurable Logic Block (CLB) enthalten sein und zu einer LUT mit bis zu 11 Eingängen kombiniert werden.

An gewissen Stellen können in FPGAs nutzerverwendbare Speicher untergebracht sein. Diese Block RAMs (BRAMs) können ähnlich einem Scratchpadspeicher für Zwischenwerte verwendet werden. Sie zeichnen sich insbesondere dadurch aus, dass das Datum unter Umständen noch im selben Takt, in dem es angefordert wurde, weiterverwendet werden kann.

Digital Signal Processing (DSP)-Einheiten werden benötigt zur Durchführung arithmetischer Operationen wie Addition oder Multiplikation, gleichermaßen in Ganzzahl- wie in Gleitkommaarithmetik. Sie können beispielsweise in der Ausprägung als DSP48 18 Bit vorzeichenbehaftete Ganzzahlen multiplizieren und auf den alten Wert bis zu einer Gesamtlänge von 48 Bit akkumulieren. Abbildung 2.3 illustriert die Verwendung am Beispiel der Addition.

[5] engl. switch matrices

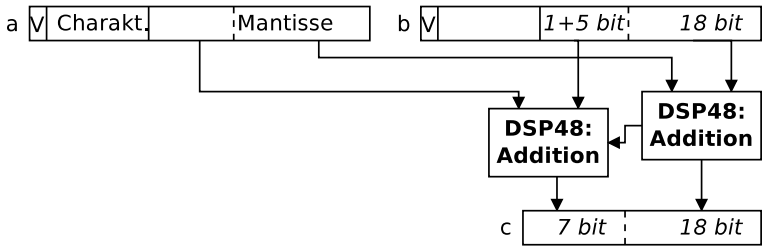


Abbildung 2.3: Addition der Mantissen zweier Gleitkommazahlen mittels DSP-Einheiten.

Die Konfiguration einer einzelnen Look-Up Table wird in SRAM-Zellen abgelegt. Ähnlich werden auch die Schaltmatrizen konfiguriert, um so die Verdrahtung verschiedener Logik- und Funktionsblöcke miteinander zu erreichen. Die Programmierung erfolgt anhand eines Bitstreams, welcher im Platzierungs- und Routing-Prozess der synthetisierten Netzliste entstanden ist. Zum Programmieren des Bitstreams auf einen FPGA gibt es unterschiedliche Möglichkeiten über USB-Verbindungen mit JTAG-Chips seitens der FPGAs oder FPGA-intern mit dem Internal Configuration Access Port (ICAP). Diesen Vorgang des Programmierens bezeichnet man als *Konfigurieren des FPGAs*.

Neben der Konfiguration eines FPGA vorab der Ausführung hat sich die Möglichkeit aufgetan, momentan unbenutzte FPGA-Teilressourcen zur Laufzeit anderweitig zu konfigurieren, indem beispielsweise die ICAP-Schnittstelle verwendet wird. Dies wird als *dynamische partielle Rekonfiguration* bezeichnet. Damit bietet sich bei JPEG2000-Beschleunigern die Möglichkeit, die verfügbaren Ressourcen nur für den gerade benötigten Teilschritt wie diskrete Wavelet-Transformation[6] oder Entropie-Kodierung[7] zu nutzen oder sogar nur Teile der EBCOTs zu konfigurieren, so dass in der Summe mehrere EBCOTs parallel verwendbar sind [3].

FPGAs verfügen über eine hohe Anzahl an Pins (wenige Hundert bis fast 2000), die zur Bereitstellung der Versorgungsspannung, Masseverbindung und Taktsignale sowie der zu verarbeitenden Nutzerdaten dienen. Ein eingehender Signalpegel von beispielsweise einem Tastendruck ist kein sauberes Signal, sondern wird erst mittels des IO Blocks (IOBs) als logische 1 definiert, wenn das Ausgangssignal des Tasters sich stabilisiert hat und als 1 zu erkennen ist. Serializer-Deserializers (SERDESs) dienen dem Anschluss von Hochgeschwindigkeitsdatenübertragungskomponenten an langsamere Verarbeitungseinheiten. Die vorhandenen Daten werden dazu sequentiell, seriell (und eventuell differentiell) mit hoher Taktrate übertragen und auf der Empfängerseite in einem SERDES gesammelt, von wo aus sie parallel anderen Teilen der konfigurierten Schaltung mit weitaus niedrigerer Taktung zur Verfügung gestellt werden. Die frei verwendbaren Pins können daher zum Anschließen von FPGAs (bzw. ihren Trägerboards) an Prozessorbusse (vgl. Convey HC-1 am Intel Front-Side Bus) oder Systembusse (vgl. DRC AC2030 im Opteron-Sockel mit HyperTransport oder UoH HTX-Board [104]) verwendet werden, wie es im weiteren Verlauf beschrieben wird.

Einsatz und Verwendung von FPGAs

FPGAs sind deshalb interessant für die Anwendungsbeschleunigung, weil von Akzeleratoren Anpassbarkeit hinsichtlich der Anwendungsanforderungen gefordert wird – die eine Anwendung benötigt zur Videodekodierung schnelle Fouriertransformationen, die nächste Anwendung möchte Reduktionen beschleunigt durchführen. Dazu sind jeweils unterschiedliche Spezialschaltungen nötig. FPGAs erfüllen die Anforderung der Anpassbarkeit aufgrund ihrer Rekonfigurierbarkeit [142]. Die in den FPGAs vorhandenen physischen Schaltungen werden geeignet parametriert, so dass die benötigte Spezialschaltung für die jeweilige Anwendung nutzbar ist. So kann der Anwender im selben System zu beliebigen Zeitpunkten beliebige Anwendungen mittels FPGAs beschleunigt ausführen. Somit sind FPGAs geeignet, als Anwendungsbeschleunigunger eingesetzt zu werden.

FPGAs dienten ursprünglich dem Rapid Prototyping von anwendungsspezifischen Schaltungen[8], digitalen Signalprozessoren[9] sowie chipbasierten Rechensystemen (SoCs). Ebenso dienen sie seit

[6] engl. Discrete Wavelet Transformation (DWT)
[7] engl. Embedded Block Coding with Optimized Truncation (EBCOT)
[8] Application-Specific Integrated Circuits (ASICs)
[9] DSPs

Langem in eingebetteten Systemen zur Umsetzung von Algorithmen in Hardware, da bei geringen Stückzahlen der Einsatz von FPGAs preisgünstiger ist als die Entwicklung und Fertigung von anwendungsspezifischen Chips (ASICs). In den letzten zehn Jahren haben sich FPGAs weiter verbreitet und wurden unter anderem als Bestandteile von Koprozessoren in einer Vielzahl unterschiedlicher Systeme integriert.

Traditionell werden FPGAs im Bereich der Signalverarbeitung eingesetzt [274], um gepipelinte, parallele Schaltungen in Festkommaarithmetik umzusetzen. Beispielsweise lässt sich in eingebetteten Systemen die Auswertung von Antennensignalen beschleunigen gegenüber der CPU-Ausführung [282]. Im Rahmen des HADES-Experiments werden in einem mehrstufigen Verfahren bis zu 10 GByte/s an Ereignissen auf FPGAs verarbeitet und danach vorgefiltert, ob es wirklich einen Lichtkegel eines auf Lichtgeschwindigkeit beschleunigten Teilchens gibt [186], und daraus dann nach Möglichkeit Masse, Energie und Ladung errechnet anhand der Ablenkung, die durch das Magnetfeld erzeugt und über die *Tracking Processing Unit* erkannt wird.

FPGAs eignen sich auch vorzüglich für bildverarbeitende Algorithmen [59, 73, 124, 132, 159, 184, 50, 138, 216] wie Kantendetektion, Kameraüberwachung, Sortieranlagen und Robotersysteme sowie ferner auch in der Medizintechnik [159, 73, 292]. Die drei Farben im RGB-Farbmodell (gegebenenfalls zuzüglich des Alpha-Kanals) können parallel berechnet werden, und ein zur Anwendung und zum Filter (Faltung, Gauß-Filter o.ä.) passender Zwischenspeicher kann implementiert werden, der die Applikation der Filter auf die Bilddaten in Streaming-Manier ermöglicht. Ist der Eingabedatenstrom breit genug, so sind auch mehrere Filter parallel applizierbar. Außerdem ist das Problem in mehrere Gebiete zerlegbar, so dass mehrere Exemplare der implementierten Kernels oder mehrere FPGAs datenparallel rechnen können. So werden kompakte Kamerasensoren mit integrierten FPGAs angeboten, welche die Ausrichtung von Objekten auf Fließbändern mittels Kontrasterkennung durch Einsatz von Kantendetektionsfiltern (Prewitt- und Sobel-Filter) bestimmen und dabei eine Erkennungsrate von 50 Bildern pro Sekunde liefern können [92].

Ebenso lohnen sich FPGAs im Bereich der Verschlüsselung. Ein FPGA-basierter Koprozessor kann mehrere AES-Einheiten enthalten und dadurch die Taktfrequenz der Einheiten senken, um die Leistungsaufnahme zu senken [246]. Der optimierte Einsatz von Zwischenspeichern und die Optimierung der Schnittstellenbreiten tragen wesentlich zur Anwendungsbeschleunigung bei.

In der Bioinformatik kann der Vergleich des menschlichen Genoms mit einem Xilinx Virtex-II Pro beschleunigt werden auf einer Cray XD1 [247]. Dabei ist das Ausrichten der Anfrage- und Referenzsequenzen mittels des Smith-Waterman-Algorithmus die zu beschleunigende Funktionalität, was als systolisches Array in Hardware umgesetzt werden kann. In einer weiteren Arbeit [39] wurde die Ausrichtung als erster Teil von DIALIGN sogar 300- bis 400-fach beschleunigt. BLAST ist ein weiteres Werkzeug, um die Ausrichtung von Sequenzen aneinander zu bestimmen. Eine FPGA-Implementierung von BLAST ist vielversprechend [203]. Auch die vorhersagende Berechnung der dreidimensionalen Struktur von Proteinen anhand der zwischen den Atomen vorherrschenden Bindungen, Öffnungswinkel und Van-der-Waals-Kräfte ist mit FPGAs möglich und verspricht Speedups von Faktor 5 je FPGA [155].

Gebietet das Einsatzziel nicht die Verwendung von Mikroprozessoren, so kann ein FPGA das zentrale Verarbeitungselement darstellen. Nebst Entwicklungs-/Evaluationsplattformen, wo es um den prototypischen Aufbau des beabsichtigten Systems und seiner Komponenten geht, sind die entwickelten Systeme zur Verarbeitung von Eingabedaten wie Temperaturwerten, Tastendrücken, massiv paralleler (Vor-)Verarbeitung ermittelter Daten wie beim ALICE-Projekt am CERN [7, 100, 139] und dergleichen FPGA-zentrisch.

Neben den bereits genannten Bereichen wurden und werden FPGAs zur Beschleunigung von numerischen Funktionen eingesetzt. In der Literatur werden Implementierungen von Multiplikationen dünnbesetzter Matrizen mit einem Vektor vorgestellt [165, 200, 201, 291] oder Cholesky-Zerlegungen [29, 285, 286], CG-Verfahren [20, 89, 90, 187, 226], Stencil-Berechnungen [13, 59, 135] und weitere. FPGAs sind dem alleinigen Anwendungszweck des Rapid Prototypings also entwachsen und stellen neben dem Einsatz in eingebetteten Systemen auch im Bereich des Hochleistungsrechnens Möglichkeiten und Kandidaten zur Anwendungsbeschleunigung dar.

Es sind die Ausnutzung mehrerer Arten von Parallelismus, das Optimieren von Datentypen, die beiderseitige Anpassung von Datentransfer und Berechnung, die Datenwiederverwendung und interne Datenbereitstellung, welche es in den hier vorgestellten Fällen möglich machen, FPGAs zur

Anwendungsbeschleunigung qua Verlagerung eines Algorithmus von Software in Hardware einzusetzen (vergleiche Parks Kriterien zur Verlagerung eines Algorithmus in Hardware [212] und Strattons für essentiell befundene Techniken beim Einsatz von sehr vielen Verarbeitungseinheiten [248]). Die Entwicklungszeit für Beschleuniger auf Basis von FPGAs fällt entsprechend sehr hoch aus, da im FPGA alles direkt so umgesetzt werden kann und muss, wie es der Algorithmus vorsieht, mit Parallelismus von der Bitebene bis hinauf zur Funktionsebene, was überdies durch die implizit parallelen Entwicklungssprachen zusätzlich erschwert wird. Die Kriterien zur erfolgreichen Beschleunigung einer Anwendung sind also, dass wesentliche Anteile der Ausführungszeit auf eine einzelne Funktion entfallen, dass diese parallel ausführbar ist, dass die Implementierung nicht (mehr) durch die zur Verfügung stehende Speicherbandbreite beschränkt sein wird, und schließlich damit einhergehend, dass die interne Speicherkapazität ausreichend ist, dies zu gewährleisten [247, 212]. Um FPGAs als mögliche Akzeleratoren in heterogenen Systemen zu programmieren, sind also detaillierte Kenntnisse sowohl der Hardware als auch des zu verlagernden Algorithmus' nötig.

Rechnen auf FPGAs

Zur erfolgreichen Umsetzung von Algorithmen auf FPGAs müssen die ausgewählten FPGAs und die sie umgebenden Systeme unbedingt eine gewisse Anzahl an Voraussetzungen erfüllen (vergleiche auch die Anforderungen an GPU-Portierungen [212, 248]). Die Nichterfüllung einzelner Bedingungen hat unmittelbare, gravierende Auswirkungen auf die erzielbare Leistungsfähigkeit einer Implementierung.

DSP-Einheiten werden zur Addition und Multiplikation von vorzeichenbehafteten sowie vorzeichenlosen ganzzahligen Werten benötigt, was die Basis für weitere Operationen wie Division sowie für weitere Zahlenformate wie Gleitkommadarstellung bildet. Für die Multiplikation zweier 64-Bit-Gleitkommazahlen sind für die Multiplikation der 53 Bits der zwei Mantissen neun DSP-Einheiten nötig, für die Akkumulation weitere vier sowie eine für die Addition der Exponenten. Diese 14 DSP-Einheiten reichen erst zur Implementierung einer einzelnen Multiplikationseinheit. Um Nutzen gegenüber herkömmlichen CPU-Systemen zu haben, benötigt man mindestens zehn Multiplikationseinheiten, um auf 2 GFLOPS bei einer Taktrate von 200 MHz zu kommen.

Hohe Geschwindigkeit des FPGAs ist unerlässlich, wenn an entfernten Stellen des FPGAs liegende Funktionseinheiten miteinander verschaltet werden müssen, um ein größeres Ganzes zu erreichen. Zwar verfügen FPGAs heutzutage über dedizierte Verbindungen für längere Entfernungen (*long wires*), dennoch sind für komplexe Schaltungen mehr Verbindungen nötig, so dass die *long wires* definitiv notwendig, aber noch lange nicht hinreichend sind. Auf einem langsamen FPGA wird die Platzierung der Einheiten problematisch, da sie häufig nicht derart miteinander verschaltet werden können, dass die Verbindungen die Signale ausreichend schnell transportieren würden und die gewünschte Taktrate erreicht würde. Die Auswirkungen sind, dass weitere Taktraten und damit Taktdomänen eingeführt werden müssen, deren Anzahl aber hardwaretechnisch limitiert ist und die zusätzlicher Synchronisation an den Übergängen bedürfen.

Viele und große BlockRAMs sind nötig, um die internen Einheiten mit Daten versorgen zu können, da die externen Schnittstellen nicht dazu ausreichen, die benötigte Datenwiederverwendung für zehn oder mehr Einheiten sicherzustellen. Die BlockRAMs können dazu verwendet werden, Caches, Scratchpad-Speicher oder FIFOs und ähnliche Zeilenpuffer zu implementieren [201].

Viele CLBs sind zur Umsetzung von Registern und für die Verfügbarkeit von Look-Up-Tabellen, die auch für Multiplexer und damit für Kontrollfluss genutzt werden, unbedingt nötig.

FPGA-nahe schnelle, große Speicher mit großer Bandbreite wie SRAM mit 64-Bit Datenbreite oder mehrere DDR3-Module werden benötigt, so dass diese unterschiedliche Datenströme für die Verarbeitungseinheiten stellen können, da der FPGA-interne BlockRAM nur für wenige Tausend Zahlenwerte ausreichend ist. Der Nutzen von FPGAs stellt sich häufig aber erst bei großen Datenmengen heraus, da dann CPUs weniger von ihren Caches profitieren können, während anwendungsspezifische Implementierungen auf FPGAs dann die vorhandene Speicherarchitektur gezielt ausnutzen können.

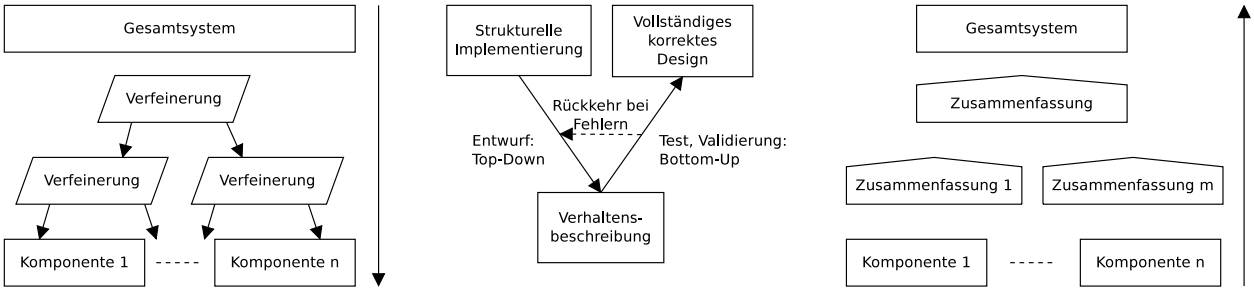


Abbildung 2.4: Entwurfsprozess bei der Entwicklung von Hardware mit niedrigebenigen Hardwarebeschreibungssprachen. Links: Top-Down-Zerlegung im strukturierten Entwurf; Mitte: V-Modell [37] mit Zerlegung, mit Implementierung des Verhaltens auf unterster Ebene und mit Testen und Zusammensetzen; Rechts: Zusammenfassen implementierter und getester Komponenten zu größeren Einheiten.

Integrierte PowerPC-Kerne wie beim FPGA des UoH HTX-Boards (Abschnitt 2.1.4) können aufgrund ihrer Allzweckarchitektur zur Ansteuerung, Organisation und Rekonfiguration von Hochleistungsdesigns auf FPGAs eingesetzt werden. Anstelle des Hardcores kann auch ein zweckspezifisch ausgewählter Softcore eines Mikroprozessors für Verwaltungsaufgaben eingesetzt werden.

Sollen Anwendungen oder zumindest Teile im Einsatzgebiet bioinformatischer und numerischer Anwendungen beschleunigt werden, so ist von Datenmengen im Bereich mehrerer Gigabyte auszugehen. Schnittstellen mit guter Bandbreite sind unabdingbar, um die für hohe Rechenleistung benötigten vielen Verarbeitungseinheiten mit ausreichend vielen Daten versorgen zu können. Entsprechend der zu verarbeitenden Datenmengen müssen die FPGAs also nah an den Speicher angebunden sein. Besonders geeignet sind die Ansätze von DRC und Convey, die FPGAs als Koprozessoren über einen weiteren CPU-Sockel anzubinden. Ihnen beiden ist gemeinsam, dass koprozessornah sehr viel Speicher zur Verfügung steht, auf den schnell zugegriffen werden kann, um nicht auf weiter entfernten Speicher zugreifen zu müssen, der erst auf der Hostplattform angeschlossen ist und dort schlimmstenfalls nicht primär dem Koprozessor, sondern einem anderen Prozessor zugehörig ist.

Entwurf digitaler Schaltungen für ASIC-Bausteine und FPGAs

Zum Entwurf digitaler Schaltungen sind insbesondere bei rekonfigurierbarer Hardware mehrere Schritte nötig, die aufgrund ihrer Komplexität und Wichtigkeit hier beschrieben werden, um die eingangs geschilderte Motivation weiter zu stützen. Wie in Abbildung 2.4 dargestellt, sind die wichtigsten Schritte beim Entwurfsprozess von Hardware die Zerlegung in Komponenten, die schrittweise Verfeinerung in Unterkomponenten, die Implementierung selbiger, und das schrittweise Testen und Zusammenfügen der implementierten Komponenten.

Zur technischen Umsetzung auf rekonfigurierbare Systeme besteht dann eine Reihe von Werkzeugen [161]. Die manuell oder mit Hilfsmitteln erstellte Hardwarebeschreibung in meist Verilog, VHDL, oder gelegentlich System C muss in mehreren Schritten zu einer auf einen FPGA aufspielbaren Konfiguration verarbeitet werden[10]. Der eigentliche Syntheseprozess besteht genauer aus dem Erzeugen mehrerer Kontrolldatenflussgraphen (CDFGs) oder Operationsgraphen ausgehend von der Hardwarebeschreibung. Die beschriebene Schaltung wird zunächst in die Register-Transfer-Ebene (RT-Ebene) gebracht und zu logischen Gattern transformiert. Die physische Umsetzung (vgl. Y-Diagramm oder Gajski-Diagramm [105]) besteht dann aus drei Teilen. Zuerst werden die logischen Gatter oder Kombinationen solcher auf die dem FPGA zur Verfügung stehenden Ressourcen wie LUTs, Blockspeicher und arithmetische Einheiten abgebildet (*Bindung / Mapping*). Im Zuge dessen wird auch die Belegung der LUTs errechnet. Dann erfolgt die Platzierung (*Placing*): Wo auf dem FPGA liegen die zu verwendenden Einheiten, ist die Verbindung zusammengehöriger physischer Einheiten schnell genug machbar, so dass die angestrebte Taktrate erzielbar ist? Abschließend

[10] Synthese=künstliche Erzeugung

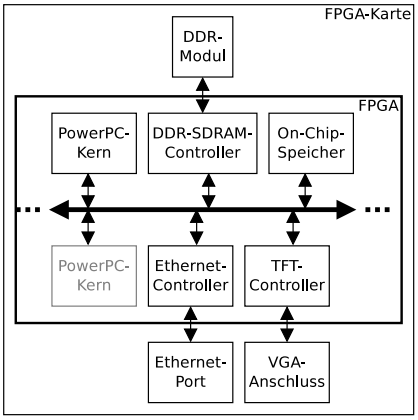


Abbildung 2.5: PowerPC-basiertes heterogenes SoC auf Xilinx-FPGA.

werden die Einheiten miteinander verdrahtet (*Routing*). Sämtliche Konfigurationen für die Belegungen der LUTs und der Schaltmatrizen müssen hinterher zusammengeführt werden, so dass sie mittels JTAG o.ä. auf den FPGA gespielt werden können (*Bitstream Generation*).

Insgesamt beläuft sich der vollständige Prozess für komplexe Designs auf großen, leistungsfähigen FPGAs auch unter Verwendung höhergetakteter Prozessoren mit vielen Kernen auf mehrere Stunden[11]. Für Anwendungsentwickler und Domänenexperten ist somit das Erzeugen eines FPGA-Bitstreams viel zu langwierig. Benötigt wird eine Methodik, die

1. einfach hinsichtlich der Werkzeugkette ist, z.B. ähnlich dem Übersetzen eines C-Programms,
2. zügig vonstatten geht innerhalb weniger Sekunden bis Minuten,
3. ohne detaillierte Kenntnisse des Hardwareentwurfs bereits die Nutzung von Schaltungen in rekonfigurierbarer Hardware ermöglicht, z.B. indem sie Hardwareimplementierungen bietet, um so letzten Endes die gewünschte Beschleunigung zu erhalten,
4. das schrittweise Entwickeln und Verbessern eines Algorithmus mit sofortiger Validierung in Hardware ermöglicht, ohne die Schritte über Software-Simulation, Emulation der Hardware, zeitaufwendige Synthese und abschließende Validierung in Hardware zu gehen,
5. von spezifischen Randbedingungen des Systems und der Nutzung rekonfigurierbarer Hardware zu abstrahieren vermag.

Neben den zuvor betrachteten Grafikkarten wurden jetzt Field-Programmable Gate Arrays (FPGAs) genauer eingeführt. Dazu wurden der Aufbau und die dahinterliegende Technologie genauer vorgestellt. Aufgrund der Möglichkeiten zur Verlagerung von Funktionen auf FPGAs wurde anschließend auf den Einsatz von FPGAs eingegangen. Dabei hat sich auch gezeigt, dass das Erzielen von Leistungssteigerung mittels FPGAs einige Voraussetzungen hat, insbesondere dann, wenn es um das Durchführen arithmetischer Berechnungen auf FPGAs geht. Abschließend wurde ersichtlich, dass der Schaltungsentwurf für FPGAs nicht nur komplex hinsichtlich der unterschiedlichen Anforderungen und Voraussetzungen ist, sondern darüberhinaus auch sehr zeitaufwendig.

2.1.3 Systems on Chips (SoCs)

FPGAs werden häufig im Kontext eingebetteter Systeme eingesetzt und dort als SoC (vgl. Abb. 2.5) oder innerhalb eines solchen. Bei diesen sind viele Elemente wie Prozessoren, Spezialeinheiten, Speichercontroller und kleine Speicher auf einem Chip integriert. Weitere Beispiele für SoCs sind Chips für Mobilfunktelefone [281], die zumeist als vollkundenspezifische Designs gefertigt werden.

[11] Dabei hat das System ausreichend viel Arbeitsspeicher zur Verfügung gestellt, so dass kein Swapping den Prozess unnötig verlängert hat.

Mikroprozessoren und FPGAs lassen sich auf drei unterschiedliche Arten direkt und unmittelbar miteinander kombinieren. Bei FPGAs der Reihe Xilinx Virtex sind häufig festverdrahtete PowerPC-Kerne enthalten (sogenannte *Hard Blocks*), die vom Entwickler durchaus zur Ausführung der Hauptaufgabe genutzt werden können und Unterstützung durch in rekonfigurierbarer Logik implementierte Erweiterungen erfahren [106], oder aber auch nur Verwaltungsaufgaben wie Datentransfers und Ausgabe übernehmen. Befehlssatzerweiterungen in der nutzerkonfigurierbaren Logik setzt man über die APU-Schnittstelle um; ganze Funktionseinheiten werden über das enthaltene Bussystem an die Prozessoren angeschlossen. Viele Arbeiten zielen auf die dynamische Anpassbarkeit eines Prozessors für quasi beliebige Anwendungen ab, wie RISPP [22, 24] und andere [191, 272], indem instruktionsspezifische Erweiterungen in der konfigurierbaren Logik im Sinne von Beschleunigereinheiten verfügbar gemacht werden, beispielsweise im Kontext eingebetteter Systeme und SoCs, aber auch bei der Convey HC-1, wo der in den Application Engine Hub eingebettete Superskalarprozessor 32 frei belegbare Befehle hat. Erst die weiteren Hardwarebausteine wie beispielsweise SD-RAM oder A/D-Wandler sitzen auf einer den FPGA umgebenden Platine. Anstatt Mikroprozessoren direkt fest mit dem FPGA zu verdrahten, können geeignet große (kleine) Mikroprozessoren auch als sogenannte *Soft-Cores* auf den FPGA gebracht werden. Beispiele sind MicroBlaze[12], PicoBlaze[13], quelloffene Implementierungen von AVR (Atmel Atmega 103[14]) und SPARC (Leon3[15]), OpenRISC[16]. Auf diese Weise lässt sich insbesondere Prototyping von Mehrkern- und Vielkern-Architekturen durchführen, aber auch VLIW-Prozessoren erzeugen. Intel hat mit dem Atom E6x5C[17] die dritte Variante vorgelegt, bei welcher der Koprozessorgedanke vorherrscht. Über die PCI-Express-Schnittstelle hat der Prozessor Zugriff auf ein im selben Multichippaket befindlichen FPGA von Altera. Dieser Ansatz ist deshalb sehr interessant, weil sich sowohl Befehlssatzerweiterungen auf dem FPGA implementieren lassen als auch grobgranularere Funktionsblöcke, beispielsweise ein Skalarprodukt. Darüberhinaus lässt sich die (Re-)Konfiguration vom Mikroprozessor aus steuern, so dass der FPGA nicht die Verwaltung von sich selbst übernehmen muss, wie dies bislang häufig in der Literatur in Ermangelung von Alternativen der Fall ist [23, 256, 257, 266, 267].

2.1.4 Busbasierte Systeme

Standardsysteme mit Prozessorarchitekturen wie der Intel Core i7-Architektur[18] oder AMD Phenom II[19] können über Systembusse wie PCI Express[20], den HyperTransport-Bus[21] oder Intel QuickPath Interconnect (QPI)[22] mit einem oder mehreren zusätzlichen Prozessoren oder Beschleunigern ausgestattet werden, anstatt FPGAs in demselben Paket wie den Mikroprozessor unterzubringen. Der Systembus HyperTransport und die darauf basierenden Plattformen UoH HTX-Board sowie DRC AA2311 werden im Folgenden beschrieben, bevor das Multi-FPGA-System Convey HC-1, das auf dem Intel Front-Side Bus (FSB) basiert, skizziert wird. Diese drei Systeme sind wesentlich für die vorliegende Dissertation.

HyperTransport ist ein Konsortialstandard des gleichnamigen Konsortiums, welchem AMD, die Universität Heidelberg, Cray, DRC, Dell, HP, Xilinx, Oracle, XtremeData und weitere angehören. Ziel ist es, ein paketbasiertes Netzwerk zu haben, das gut skaliert. Angefangen bei den direkten Verbindungen zwischen CPUs, die als sogenannter „Tunnel“ implementiert sind, weiter über den Anschluss von Speicher hin zur Einbindung von Peripheriekomponenten wie Netzwerkkarten und FPGA-Karten, die als „Cave“ ein Endgerät darstellen und vor allem zum Prototyping von Netzwerkprotokollen und Partitioned Global Address Space (PGAS)[23] [287] dienen, erstreckt es sich

[12]URL: http://www.xilinx.com/tools/microblaze.htm
[13]URL: http://www.xilinx.com/products/intellectual-property/picoblaze.htm
[14]URL: http://opencores.org/project,avr_core
[15]URL: http://gaisler.com/index.php/products/processors/leon3?task=view&id=13
[16]URL: http://openrisc.net/
[17]URL: http://www.altera.com/devices/processor/intel/e6xx/proc-e6x5c.html
[18]URL: http://www.intel.com/cd/products/services/emea/deu/processors/corei7ee/overview/406055.htm
[19]URL: http://www.amd.com/de/products/desktop/processors/phenom-ii/Pages/phenom-ii.aspx
[20]URL: http://www.pcisig.com/specifications/pciexpress/
[21]URL: http://www.hypertransport.org/default.cfm?page=Technology
[22]URL: http://www.intel.com/technology/quickpath/
[23]URL: http://pgas.org/

über zahlreiche unterschiedliche Geräte und Zwecke.

HyperTransport ist auf physischer Ebene über „Lanes“[24] organisiert, welche die 8-Bit-Daten-, Steuerungs-, Takt- und Systemsignale zusammenfassen. Mehrere Lanes werden zusammengefasst zu einem „Channel“[25], wobei ein Gerät ebenso wie ein Mikroprozessor mehrere Channels besitzen kann: zu vier weiteren Prozessoren und zu dem lokalen Speichercontroller.

Daten auf den Lanes bzw. Channels bilden Pakete vom Typ Information, Anfrage, Antwort oder Daten. Auf ein *Kontrollpaket* für eine Anfrage kann eine weitere Anfrage folgen, oder es können die benötigten Daten bei einer Schreibanfrage folgen. Auf eine erhaltene Leseanfrage ist mit einem *Antwortpaket* und angehängtem *Datenpaket* zu antworten. *Informationspakete* können für das Abrufen von Geräteinformationen und -zustand versandt werden. Zum Versenden der Daten wurden drei unterschiedliche „Queues“[26] eingeführt: „Posted“[27] ist für solche Anfragen mit angehängten Daten wie einem Schreibzugriff, „Non-Posted“[28] für Anfragen ohne Daten wie eine Leseanfrage, und „Response“[29] dient unter anderem Antwortpaketen auf Leseanfragen. Der Hardwareentwickler muss bei der Verwendung von HyperTransport also genau beachten, welchen Typ auf welcher Schlange mit oder ohne Daten er erhält und ggf. beantworten muss. Bei HyperTransport bestehen zwei unterschiedliche Möglichkeiten zum Anschluss eines FPGAs: Die eine ist über einen freien Prozessorsockel, die andere ist über den sogenannten HyperTransport Expansion Socket (HTX), also eine Steckkartenschnittstelle.

UoH HTX-Board. Um HyperTransport zu entwickeln, testen und weiterzuentwickeln, hat das HyperTransport Center of Excellence[30] ein FPGA-Board für den HTX-Slot entwickelt [42, 104]. Die Anbindung über den HTX-Slot und HyperTransport an den Prozessor ist in Abb. 2.6 dargestellt. Dieses Board ist primär zur Implementierung vom HT-Core[31] [43, 240, 241] auf FPGAs entwickelt worden und desweiteren zum Aufbau von PGAS-Systemen gedacht, indem die bis zu sechs optischen Gigabit-Schnittstellen für ein Netzwerk genutzt werden. Es kann aber auch als spezifischer oder generischer Anwendungsbeschleuniger genutzt werden [NBKK09, KVB+09].

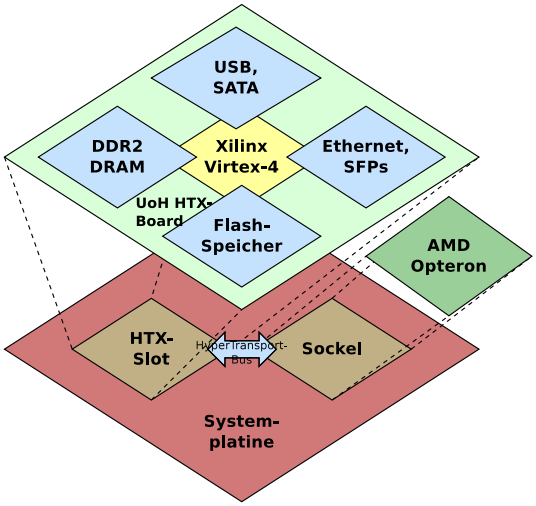
DRC Accelium AA2311. Direkt am HyperTransport-Sockel ist hingegen die in Abb. 2.7 abgebildete RPU (Reconfigurable Processing Unit) „DRC AC2030“ [88] von DRC Computer[32] im System DRC AA2311 angeschlossen. Die AA2311 ist ein NUMA-System mit zwei Sockeln und 8 GB RAM installiert am CPU-Sockel und weiteren 4 GB RAM am Koprozessorsockel, so dass man einen CPU-Knoten und einen Koprozessor-Knoten vorliegen hat, die jeweils eine Verarbeitungseinheit und Speicher besitzen.

Ein Platform Support Package (PSP) von Synective Labs[33] ermöglicht die Verwendung von Impulse C als Programmiersprache für den DRC Koprozessor. Impulse C selbst verwendet das Prinzip der Communicating Sequential Processes (CSPs) (Abb. 2.8). Das PSP setzt dazu das Application Programming Interface (API) von Impulse C auf das DRC-API (Abb. 2.9) um. Anwendungsentwickler sehen sich hier bereits mit einer Menge an Schnittstellen und Ebenen konfrontiert, welche die Fehlersuche und -behebung komplizierter gestalten sowie zusätzliche Ineffizienz hinsichtlich der benötigten Entwicklungsdauer mitbringen könnten.

2.1.5 Multi-FPGA-Systeme

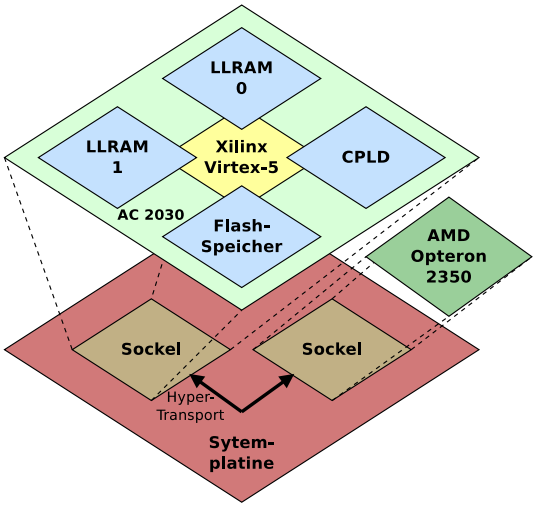
Die bereits genannte, für diese Dissertation wichtige Convey HC-1 hebt sich von den vorgestellten busbasierten Systemen dadurch ab, dass sie über mehrere FPGAs verfügt. Vorab einer genaueren

[24] engl. Lane = Spur
[25] engl. Channel = Kanal
[26] engl. Queue = Schlange
[27] engl. posted = angehängt, gesandt
[28] engl. non-posted = ohne Anhang
[29] engl. response = Antwort
[30] URL: `http://ra.ziti.uni-heidelberg.de/coeht/`
[31] URL: `http://ra.ziti.uni-heidelberg.de/coeht/?page=projects&id=htcore`
[32] URL: `http://www.drccomputer.com/`
[33] URL: `http://synective.se`



Xilinx Virtex-4	
Modell	FX100
4-Input LUTs	94.896
Digital Clock Managers	12
AMD Opteron	
Modell	Dual-Core 870
Kerne	2
Kerngeschwindigkeit (MHz)	2.000
Systembusgeschwindigkeit (MHz)	1.000
L1-Cachegröße (KB pro Kern)	I$: 64, D$: 64
L2-Cachegröße (KB pro Kern)	1.024

Abbildung 2.6: Das UoH HTX-Board in einem AMD Opteron-System.



Xilinx Virtex-5	
Modell	LX330
6-Input LUTs	207.360
Digital Clock Managers	12
DSP48E-Einheiten	192
AMD Opteron	
Modell	2350
Kerne	4
Kerngeschwindigkeit (MHz)	2.000
Systembusgeschwindigkeit (MHz)	1.000
L1-Cachegröße (KB pro Kern)	128
L2-Cachegröße (KB pro Kern)	512
L2-Cachegeschwindigkeit (MHz)	2.000
L3-Cachegröße (KB pro Die)	2.048
Geschwindigkeit des integrierten Speichercontrollers (MHz)	1.800

Abbildung 2.7: Das Koprozessorsystem DRC AA2311 mit dem Koprozessor DRC AC2030.

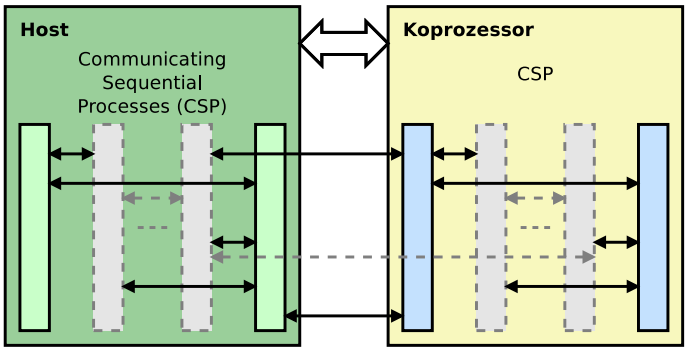


Abbildung 2.8: Einsatz von *Communicating Sequential Processes* bei Impulse C.

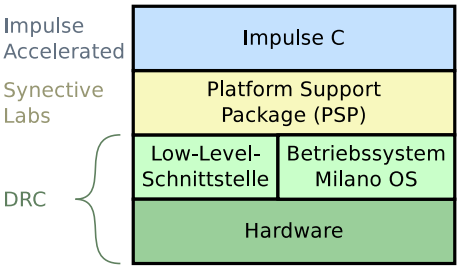
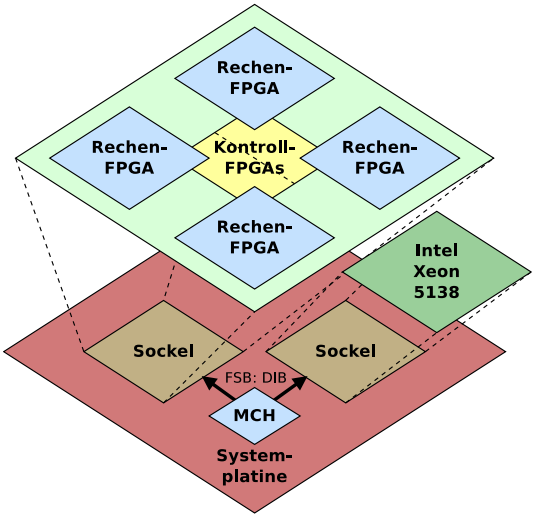


Abbildung 2.9: Anschluss von Impulse C mittels des PSP von Synective Labs und des Milano OS an die Hardware von DRC.



4 × Xilinx Virtex-5	
Modell	LX330
6-Input LUTs	207.360
Digital Clock Managers	12
DSP48E Slices	192
Intel Xeon	
Modell	5138
Kerne	2
Kerngeschwindigkeit (MHz)	2.136
FSB-Geschwindigkeit (MHz)	1.066
L1-Cachegröße (KB pro Kern)	I$: 32, D$: 32
L2-Cachegröße (KB pro Die)	4.096
Geschwindigkeit der Speicherschnittstelle (MHz)	533

Abbildung 2.10: Das Koprozessorsystem Convey HC-1 (ohne koprozessorseitige Speicher).

Vorstellung der Convey HC-1 werden daher Multi-FPGA-Systeme (MFSs) [133] beschrieben. Die universitäre Forschung sowie mehrere Hersteller von FPGA-Systemen liefern Systeme mit mehreren FPGAs aus mehreren Gründen aus. Einerseits können mehr FPGAs auch mehr Leistung bieten. Andererseits lassen sich auf MFSs größere Anwendungen in ihrer Gesamtheit beschleunigen anstelle einzelner ausgewählter Kernel, da mehr Platz vorhanden ist zur Umsetzung großer Designs, die auf einem einzelnen FPGA nicht mehr platzierbar wären. Multi-FPGA-Systeme sind mit leistungsfähigen Speicherstrukturen konfiguriert und können eine große Bandbreite an Anwendungen teilweise erheblich unterstützen. Allen voran steht die *Berkeley Emulation Engine* (BEE), die seit 2009 schon in der dritten Generation [77] hergestellt und auch weiterverkauft wird.

Die Verwendung von Multi-FPGA-Systemen ist von wissenschaftlichem Interesse [230] bei der Erforschung von Rechnerarchitekturen [77], Vielkernarchitekturen [46] und Programmiermodellen, beispielsweise bei der Entwicklung von Chip-Multiprozessoren mit Transactional Memory [207]. Dabei stand die erzielte Leistung der Multiprozessorsysteme überwiegend im Hintergrund. Es finden sich einige Portierungen verschiedener Anwendungen auf MFSs mit positivem Ergebnis [32, 112, 294, 282, 292].

Das Multi-FPGA-System Convey HC-1 [61] ist eine sobezeichnete *Hybridkern-Rechenplattform* und besteht, wie in Abb. 2.10 zu sehen, aus einem Mainboard mit zwei Sockeln, wo ein Sockel mit einem Intel Xeon Zweikernprozessor ausgestattet ist und der andere Sockel die Koprozessorplatine trägt. Auf dieser Platine sind mehrere FPGAs für unterschiedliche Aufgaben vorhanden.

Convey[34] hat sich bei der HC-1 während des Entwurfs und der Implementierung für die Nutzung des Intel FSB mit einer maximalen Bandbreite von 8,5 GB/s zwischen zwei Sockeln entschieden, aber bei dem Nachfolgemodell HC-2[35] bereits Intel QPI als Systemschnittstelle verwendet, so dass bis zu 64 GB/s Bandbreite zwischen Host-CPU und Koprozessorboard ausnutzbar ist.

Die zentralen Kontroll-FPGAs implementieren den Application Engine Hub (AEH) und sind verantwortlich für die Handhabung des FSB-Protokolls, die Speicherverwaltung, die Speicheraktualisierung nach hostseitigem Schreibzugriff mittels des Cachekohärenzprotokolls und nicht zuletzt für die Durchführung skalarer Operationen auf koprozessorseitig liegenden Daten. Dazu enthalten sie leistungsfähige Superskalarprozessoren als sogenannte „Soft Cores“. Sie bieten ferner die Anbindung der Rechen-FPGAs („Application Egines“, AEs) über eine Registerschnittstelle an. Seitenzugriffsfehler werden gleichermaßen auf der Host-CPU und dem Skalarprozessor entsprechend der Speicheradresse behandelt und führen ggf. zum Migrieren von Seiten. Dieses Konzept wurde auch von Intel im Rahmen von EXOCHI verwendet [276]. Das Konzept der virtuellen Speicherverwaltung ist von großer Bedeutung bei der Implementierung der Convey HC-1.

[34] URL: `http://www.conveycomputer.com`
[35] URL: `http://www.conveycomputer.com/index.php/download_file/view/44/157/`

Bei der Convey HC-1 ist der 16 GB große Speicher des Koprozessors folgendermaßen eingebettet: Auf der Hostplattform sind 24 GB gewöhnlicher Arbeitsspeicher installiert. Über ein „Fenster“ von 16 GB des Hauptspeichers adressiert die CPU indirekt die 16 GB des Koprozessors („1-zu-1-Abbildung“). Alternativ kann bei einer dynamischen Abbildung mittels eines Fensters von nur 4 GB Größe auf dem Host auf die 16 GB des Koprozessors zugegriffen werden. Die Organisation dieses virtuellen Adressraums erfolgt bei x86-basierenden Architekturen mittels Seiten. Die mehrstufige Umsetzung von virtuellen Seitenadressen zu physischen Adressen wird, abgesehen von nötigen helfenden Zwischenspeichern (Translation-Lookaside Buffer (TLB)), von der Memory Management Unit (MMU) vorgenommen. Je eine MMU muss im Host- sowie im Koprozessor vorhanden sein.

Schreibzugriffe des Hosts in diesen Fensterbereich führen zu Cachekohärenznachrichten an den Koprozessor. Die MMU des Koprozessors ist in der Lage, Zugriffe auf hostseitige, also noch nicht auf den Koprozessor verlagerte, Speicherbereiche aus dem angegebenen Fenster durchzuführen. Diese MMU ist in einem der zwei FPGAs des AEH implementiert. Aufgrund der seitenbasierten Speicherverwaltung führt ein solcher Zugriff dazu, dass die MMU die gesamte Seite gegebenfalls verlagert, so sie noch auf der jeweils anderen Seite liegt. Dies kann hostseitig zwar sehr schnell erfolgen, nicht jedoch koprozessorseitig. Bei fehlender Sorge um die Datenhaltung und den Datentransfer migriert Convey dynamisch die CPU-seitigen Speicherinhalte seitenweise zum Koprozessor mittels des AEHs anstatt der 10-20 mal schnelleren CPU, so dass ohne Berücksichtigung der Speichertransfers keine gute Leistung mit dem compiler-basierenden Ansatz erzielbar ist, sondern erst unter Beachtung der Datenallokation und -übertragung [15]. Daher ist zu berücksichtigen, dass vom Koprozessor zu lesender Speicher direkt koprozessorseitig alloziert wird oder die Daten vorab zum Koprozessorspeicher verschoben bzw. kopiert werden. Es müssen ferner auch solche Aspekte wie Geschwindigkeit arithmetischer Operationen in Abhängigkeit der Speicherörtlichkeit berücksichtigt werden. Ein geeigneter Ansatz wird in dieser Schrift erarbeitet, mittels dessen der Anwendungsentwickler von den dazu benötigten hardwarespezifischen Codeformulierungen weitgehend befreit werden kann.

Convey bietet die Nutzung sogenannter „Personalities“ an, die eine Sammlung von Befehlssatzerweiterungen[36] für jeweils eine bestimmte Domäne, beispielsweise Finanzmathematik, umfassen. Das Problem daran ist jedoch, dass sich keine zwei Personalities gleichzeitig nutzen lassen. Lässt sich eine Anwendung nicht ausreichend mittels einer einzigen Personality beschleunigen, bedarf es der Neukonfiguration des FPGAs zur Laufzeit mit einer anderen Personality, was die weitergehende Erforschung von Scheduling- und Mapping-Problemen bedingt, oder einer gänzlich neuen, nutzer- und anwendungsspezifischen Implementierung einer eigenen Personality bedarf.

2.1.6 Leistungsaufnahme und Energiebedarf von FPGA-Systemen

FPGA-basierte Systeme warten je nach Betrachtungsweise mit guter oder schlechter Energieeffizienz auf. Unter Energieeffizienz wird häufig verstanden, mehr Operationen pro Zeiteinheit bei gleicher oder geringerer Leistungsaufnahme durchzuführen. Diese Definition kann umgestellt werden auf die Ausführung eines bestimmten Algorithmus in möglichst kürzerer Zeit bei weniger Energieverbrauch, um auch der Ausführung von Spezialoperationen, wie sie in rekonfigurierbarer Logik implementierbar sind, Rechnung zu tragen. Beim Vergleich mehrerer Systeme bietet es sich an, das Verhältnis der Ausführungszeiten bzw. die Beschleunigung gegenüber einem Referenzsystem zu der Leistungsaufnahme zu betrachten:

$$\text{Energieverhältnis} = \frac{\text{Referenzdauer}}{\text{Ausführungsdauer}} \cdot \frac{\text{Referenzleistungsaufnahme}}{\text{Leistungsaufnahme}}$$

Sind mehrere Einheiten in einem heterogenen System bei der Ausführung beteiligt, so bietet es sich an, konkret die Energieeffizienz zu betrachten. Dabei gilt für die benötigte Energie in einem heterogenen System mit n Ausführungseinheiten: $E_{\text{heterogenes System}} = \sum_{i=1}^{n} P_i * t_i = \sum_{i=1}^{n} E_i$. Dabei können mehrere Einheiten $i \neq j$ zeitgleich zueinander genutzt werden, also $t_{\text{heterogenes System}} < \sum_{i=1}^{n} t_i$. Die Energieeffizienz kann prinzipiell durch das Energy-Delay-Product (EDP) [114] beschrieben werden, wobei beim EDP Energie hinsichtlich des Wechsels der kapazitiven Last beim Schalten eines Pfads in einer CMOS-Schaltung gemeint ist mit $E = \frac{1}{2} \cdot C_L \cdot U_{DD}^2$ ($\hat{=}$ Power Delay Product (PDP)), und die Zeitdauer sich auf die Ausführung eines Schaltkreises oder gar einer Anwendung

[36] engl. Instruction Set Extensions

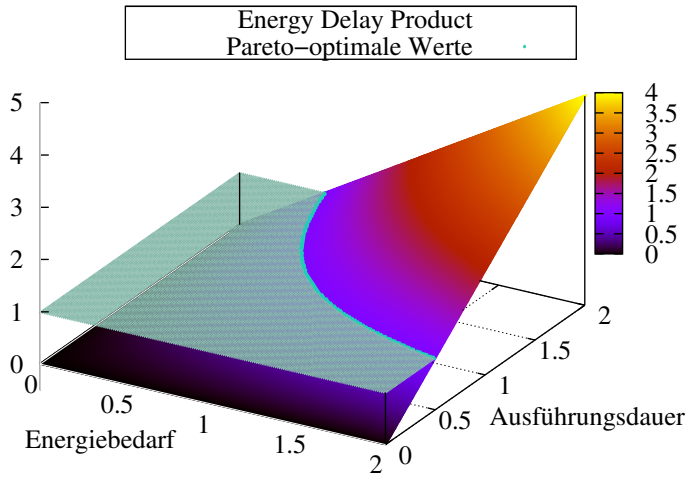


Abbildung 2.11: Energy-Delay-Product (EDP) in Abhängigkeit zu normiertem Energiebedarf und normierter Ausführungszeit. Die Pareto-optimalen Punkte befinden sich auf der Schnittkurve zwischen der Ebene $z = 1$ und den EDP-Werten.

bezieht. Für die CMOS-Schaltungen wurde ein optimales physikalisches Modell angenommen, um so eine untere Schranke für die Energieeffizienz zu haben. Anhand dieser Festlegung lässt sich eine Pareto-Kurve nachvollziehen beim Auslegen von Schaltkreisen und der erzielbaren Ausführungszeit: „Kompliziertere“ Schaltkreise müssen das Mehr an benötigter Energie durch schnellere Ausführung der Anwendung wettmachen.

Bezeichne E_X den Energiebedarf eines Systems X, dann ist:

$$EDP = \frac{E_{\text{heterogenes System}}}{E_{\text{Referenz}}} \cdot \frac{t_{\text{heterogenes System}}}{t_{\text{Referenz}}}.$$

Ein System gilt als energieeffizienter im Vergleich zu einem anderen System $\Leftrightarrow EDP < 1 \Leftrightarrow 1/EDP > 1$. Ausgehend von einem optimal ausgelegten System hinsichtlich Energiebedarf und Ausführungszeit stellt Abb. 2.11 dar, in welchem Rahmen sich das EDP bewegen kann: Es ist physikalisch unmöglich, $EDP < 1$ zu erreichen, was durch die Ebene $z = 1$ beschrieben wird; größere Werte sind jedoch möglich, d.h. ein anderes System benötigt mehr Energie oder mehr Zeit für die gleiche Aufgabe und ist dementsprechend nicht effizient. Die Menge der Pareto-optimalen Punkte ist genau die hervorgehobene Schnittmenge der Ebene $z = 1$ mit den EDP-Werten. Zu dieser Metrik ist jedoch anzumerken, dass die Ausführungszeit quadratischen Beitrag hat und daher ein (homogenes) System mit der vierfachen Leistungsaufnahme nur doppelt so schnell sein muss, um $EDP = 1$ noch zu erfüllen, wohingegen bei einer Verlangsamung von Faktor 10 ebenso $EDP = 1$ gilt, wenn das System nur ein Hundertstel der Leistung aufnimmt. Effizienz ist aus Sicht eines Anwenders allerdings nur dann gegeben, wenn die Verlangsamung sehr gering ausfällt ($0 \ll$ Speedup $S \leq 1$) oder Beschleunigung erzielt wird ($S > 1$). Daher wird in dieser Dissertation zusätzlich zum EDP stets die Beschleunigung S angegeben.

Für grundsätzliche Vergleiche der Leistungsaufnahme, des Energiebedarfs und der Energieeffizienz der Verarbeitungseinheiten ist es ausreichend, wenn sämtliche beteiligte Verarbeitungseinheiten berücksichtigt werden, insbesondere auch die CPU bei der hybriden Ausführung eines Algorithmus, allerdings weitere beiderseits involvierte Komponenten wie Speicher und Controller als sehr ähnlich gewertet werden und daher nicht weiter mitberechnet werden. Die Ermittlung der Leistungsaufnahme sollte dabei jedoch nicht anhand der angeschlossenen Netzteile oder der maximalen Leistungsaufnahme von FPGAs erfolgen, sondern so gut wie möglich anhand der tatsächlich benötigten bzw. belegten Ressourcen. Bei Xilinx FPGAs bedeutet dies, dass bei fehlenden Möglichkeiten zur direkten Ermittlung der Leistungsaufnahme die platzierte und geroutete Netzliste genommen werden sollte zusammen mit einem repräsentativen, aussagekräftigen Schaltverhalten der Gatter, das in Simulation mit Modelsim ermittelt werden kann. Damit kann man den *Power Analyzer* (XPA) die Leistungsaufnahme errechnen lassen, anstatt die maximale Leistungsaufnahme des FPGAs zu seinen Ungunsten zu betrachten.

Das Problem bei solchen Vergleichen bleibt, dass ein Koprozessor wie eine FPGA-Karte oder Grafikkarte nicht einfach ohne weiteres Zutun das Problem berechnen kann, sondern eingebettet in ein Umfeld aus externem Speicher und Dateneingangsschnittstelle für die zu verarbeitenden Daten betrachtet werden muss.

2.1.7 Behandlung der Datentransferproblematik bei non-uniformen Speicherarchitekturen in heterogenen Systemen

Nutzen ist von FPGAs vor allem dann erzielbar, wenn die Eingangs- und Ausgangsdaten gestreamt werden können bzw. hohe Datenlokalität vorliegt und die Daten nahe der Ausführungseinheiten vorgehalten werden können. Die Ausführungszeit erhöht sich entsprechend um die Dauer, bis das erste Datum übertragen ist, und die Dauer, bis das letzte Datum vom Koprozessor zum Host zurückübertragen ist, analog dem Füllen und Entleeren einer Pipeline:

$$t_{\text{total}} = 2 * t_{\text{transport latency}} + t_{\text{compute}}$$

Bei ausreichend hohem Berechnungsaufwand ist der nötige Transportaufwand dann nicht mehr ausschlaggebend. Die Datentransferproblematik bei der Programmierung von heterogenen Systemen ist von mehrerlei Seiten aus lösbar.

Self-aware Memory (SaM) [45] versucht, die Datenlokalität anhand von dezentraler, lokaler Überwachung der Speicherzugriffe und Zugriffsquellen und durch die Migration von Daten in einen anderen Speicher zu erhöhen. Derart wird die Datentransferproblematik autonom gelöst ohne jegliche Anweisung seitens des Entwicklers. Entscheidungen zur Migration werden ebenso dezentral gefällt, wie auch das Monitoring erfolgt. Modelle wie SaM können ferner dazu dienen, Prefetching einzuleiten, z.B. bei der seitenweisen, MMU-gesteuerten Migration des zugegriffenen Speichers auf der Convey HC-1 vom Host-Speicher in den Koprozessorspeicher. Denn diese seitenweise Migration sollte automatisch von Hostseite aus initiiert werden, wodurch insgesamt der Entwickler vom speicherorientierten Allozieren und Verwalten der Objektstrukturen entbunden würde.

Bekannterweise sind cachekohärente Speichermodelle schneller gegenüber nicht-kohärenten Zugriffen oder dem Kopieren von Daten in einen anderen Speicher(-bereich) [115, 276]. Dazu müsste auf der Convey HC-1 aber auch die MMU des Koprozessors gleich schnell wie die der Host-CPU sein, was mit FPGAs als Plattform zur Implementierung des Superskalarprozessors auf dem Koprozessorboard der HC-1 nicht machbar ist. Daher müsste eigentlich auf eine nicht-kohärente Variante oder das manuelle Kopieren zurückgegriffen werden. Die Convey HC-1 ist jedoch cachekohärent implementiert. Insbesondere für größere Datenmengen ist das manuelle, hostinitiierte Kopieren der Daten über die von Convey bereitgestellte Methode zum Kopieren von Daten zum Koprozessor allerdings schneller, als wenn der Koprozessor die Seitenmigrationen selbst vornimmt, was bei Verwendung der Standardroutinen der GNU C-Bibliothek erfolgt. Die in Abb. 2.12 dargestellten Ergebnisse zu dazu durchgeführten Messungen veranschaulichen, dass nur bei kleinen Datenmengen die MMU des Koprozessors der automatischen Seitenmigration nachkommen kann.

Insgesamt zeigen diese Untersuchungen bereits, dass heterogene, rekonfigurierbare Systeme sehr hardware- und problemspezifisch zu nutzen sind, beispielsweise mit unterschiedlichen Kopierroutinen. Der Anwender soll von solchen Kenntnissen und Anforderungen befreit werden.

Interessant ist, die Datentransfers zusätzlich hinter Streaming zu verstecken, wie bereits angesprochen. Unter Streaming wird das kontinuierliche Versenden sowie Empfangen und dazu gleichzeitige Verarbeiten von Daten verstanden. Dies wird beispielsweise von StreamX10 erreicht [279]. Streaming wird auch ferner von Impulse C, das auf Communicating Sequential Processes [144] aufbaut, bei der hochsprachlichen Entwicklung von FPGA-Schaltungen eingesetzt, indem hostseitig einzelne Daten asynchron übertragen werden, während weitere Berechnungen auf der CPU ausgeführt werden können. Bei Streaming erfolgt die Ausführung weitestgehend *datengetrieben*. Datengetriebene Ausführung wird später gesondert in Abschnitt 6.1.2 behandelt.

Streaming wurde für Multi-Prozessor-Systeme auf einem Chip mit gemeinsamem Speicher untersucht [118]. Dazu wurde eine Middleware eingeführt, welche die Zugriffskontrolle auf den gemeinsamen Speicher in Hardware durchführt. Der Vorteil ergibt sich daraus, dass die Prozesse *busy*

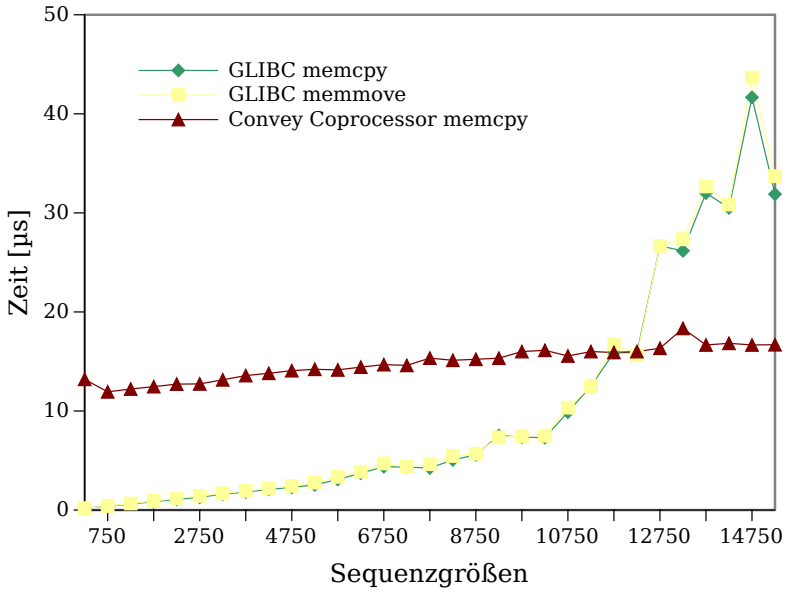


Abbildung 2.12: Ausführungsgeschwindigkeit unterschiedlicher Möglichkeiten zum Transportieren von Sequenzdaten auf der Convey HC-1. Bei den GLIBC-Routinen migriert der Koprozessor die Daten.

waiting auf Daten betreiben müssen und daher der Aufwand zum Aufwecken entfällt, was im spezifischen Kontext eingebetteter Systeme für Telefon- und Videoanwendungen geeignet scheint im Gegensatz zu herkömmlichen Desktop- und Serversystemen, wo Schlafenlegen und Aufgewecktwerden eingesetzt werden sollten. Die Nutzung von zusätzlichen Hardwareressourcen, etwa FPGAs, als Berechnungseinheiten wird explizit vorgeschlagen, da die Middleware keinerlei Beschränkungen diesbezüglich auferlegt und der potentielle Nutzen rekonfigurierbarer Architekturen als gegeben angenommen wird. Dabei sind FPGAs als zusätzliche Recheneinheiten angedacht im Gegensatz zur Nutzung als spezielle Beschleuniger.

Erste Arbeiten in die Richtung eines Programmiermodells für Streaming-Anwendungen auf beliebigen Funktionseinheiten wurden über Pipelining erreicht [254]. Ein Programm wird während der Entwicklungsphase vom Programmierer mit Makros instrumentiert, mittels einer erweiterten Version von Valgrind auf Pipeline-Möglichkeiten untersucht und abschließend automatisch anhand der detektierten Streams in eine Pipeline-Version transformiert. An dieser Stelle kann es sich anbieten, Teile der Pipeline in weitere Beschleuniger zu verlagern oder heterogene CMPs zu betrachten, insbesondere aufgrund der Verwendung von (Doppel-)Puffern zum Datenaustausch zwischen den Pipelinestufen in der zitierten Arbeit, was sich auch für Beschleuniger ohne Kopplung über gemeinsamen Speicher anbietet.

2.2 Digitale Arithmetik

Im Bereich des wissenschaftlichen Rechnens sind sehr häufig sehr viele arithmetische Operationen auf den Arbeitsdaten auszuführen. Für digitale Rechenanlagen stellte sich die Frage, wie man Zahlen mittels Bits und Bytes in Rechnern darstellen, speichern und damit rechnen kann. Es haben sich drei wichtige Zahlendarstellungen in der Rechnerarithmetik etabliert: das Festkommaformat, das Binary Coded Decimal (BCD)-Format[37] und die Gleitkommadarstellung. Die verwendeten, genormten Zahlenformate sind mitunter nicht optimal geeignet, ein Problem zu bearbeiten, so dass es sich lohnen kann, von den bestehenden Normen abzuweichen, um ein Problem mit einer geringeren Anzahl an Operationen oder Iterationen zu lösen. Mit Standardprozessoren muss ein solches Abweichen emuliert werden, was in der Regel mehr Operationen bedeutet und damit zunächst eine verlängerte Ausführungszeit. Mit rekonfigurierbaren Akzeleratoren bietet sich die Möglichkeit,

[37] engl., binär kodierte Dezimalzahlen

von den Normen abweichende Operationen direkt zu implementieren und den Anwendungen anzubieten. Auf den folgenden Seiten sind einige wesentliche Methoden und Aspekte des Rechnens in digitaler Hardware zusammengefasst, um einerseits die Normen und andererseits ihre Probleme sowie mögliche Abweichungen zu verstehen. Hinsichtlich möglicher Modifikationen werden dazu ferner besondere Zahlenformate vorgestellt.

2.2.1 IEEE 754

In der Gleitkommadarstellung nach dem bekannten Vorzeichen-Exponent-Mantissen-Schema werden Zahlen folgendermaßen zusammengesetzt:

$$Zahl = (-1)^{Vorzeichen} \cdot 2^{Exponent} \cdot Mantisse, \tag{2.1}$$

wobei m die Länge des Felds der Charakteristik sei, n die Länge der Mantisse, und der Exponent sich aus der Charakteristik berechnet: $Exponent = Charakteristik - 2^{m-1} + 1$ (die Charakteristik ist also der Exponent plus die benötigte Verschiebung). Die Mantisse enthält dabei das Komma an einer bestimmten, festzulegenden Stelle, z.B. nach dem ersten Bit.

Das Gleitkommaformat nach IEEE 754 [151] ist ein sehr kompaktes Format, das im Vergleich zu Festkommaarithmetik einfach zu verwenden ist und die Verschiebung des Wertebereichs bei arithmetischen Operationen mittels der Anpassung des Exponenten zu behandeln vermag. Es ist folgendermaßen spezifiziert: Gleitkommazahlen nach IEEE 754-2008 liegen einfach genau[38] (32 Bit), doppelt genau[39] (64 Bit) oder vierfach genau[40] (128 Bit) vor. Die Kardinalität bezieht sich dabei auf eine spezifizierte Wortlänge von 4 Bytes. Dabei teilen sich die verfügbaren Bits für die Charakteristik und Mantisse für die unterschiedlichen Breiten wie folgt auf: 8 Bits + 23 Bits, 11 Bits + 52 Bits, 15 Bits + 112 Bits. Darüberhinaus gibt es die erweiterten einfach und doppelt genauen Formate, die über einen etwas größeren Exponentenbereich verfügen (mind. 11 bzw. 15 Bits) und mehr Genauigkeit (mind. 32 bzw. 64 Bits) und hardware-intern eingesetzt werden. Sie erreichen unter anderem die genauere Rundung durch die „Schutzziffern“[41] [113].

Bei der Gleitkommaarithmetik fällt der Speicherbedarf sehr gering aus. Allerdings ist im Gegensatz zu Festkommaarithmetiken und insbesondere dem BCD-Format stets nur genau ein Bereich präzise. Dahinter befindliche Stellen werden irreversibel abgeschnitten, weshalb die Genauigkeit nicht durchweg gegeben bleibt. Heutige Mikroprozessoren rechnen intern mit längeren Mantissen und verschieben dadurch dieses Problem etwas.

Kahan stellt fest [162], dass das Format über zu viele Eigenschaften und Merkmale wie Rundungsmodi, Sonderwerte, denormalisierte Zahlen und Exceptions verfügt. Nicht alle werden weder von Compilern und Programmiermodellen unterstützt, weshalb sie kaum verwendet werden und somit noch weniger Berechtigung für die Integration in Compiler oder Programmiermodelle erhalten.

2.2.2 Arithmetik auf FPGAs

Spezialoperationen und solche Operationen, die nicht nach dem Standard IEEE 754 arbeiten wie beispielsweise solche mit besonderer Arithmetik, stellen Kandidaten zur Verlagerung auf Akzeleratoren dar, so sie nicht mittels Verwendung von mehreren Prozessorkernen beschleunigbar sind. Daher werden nun auf rekonfigurierbaren Akzeleratoren implementierte Arithmetiken sowie die Implementierungen besonderer Operationen kurz vorgestellt.

[38] engl. single precision
[39] engl. double precision
[40] engl. quad(ruple) precision
[41] engl. guard digits, sticky bit

Arithmetische Operationen und Kernel

FPGAs eignen sich zum Rechnen auf bestimmten Gruppen wie Galois-Feldern $GF(p^m)$ im Rahmen kryptographischer Anwendungen [30, 98, 224]. Galois-Felder sind definiert als Ring über einer Wertemenge mit zwei Operationen (Addition und Multiplikation) und zugehörigen neutralen Elementen. Für $k \in \mathbb{N}$, k fest, ist $z \in GF(2^m)$ mit $z^k = 1$ eine k-te Einheitswurzel. Ist k im Sinne eines privaten Schlüssels nicht bekannt, so kann nicht auf z rückgeschlossen werden. Das Addieren in Galois-Feldern ist sehr schnell in Hardware umsetzbar und ausführbar, nicht jedoch die Multiplikation und darauf basierende Operationen oder Polynomauswertungen. Ferrer et al. haben einen FPGA-basierten Koprozessor zum Rechnen in Galois-Feldern entwickelt, wo die Multiplikation mittels einer Matrixvektormultiplikation erfolgt mitsamt der Reduktion; die Akkumulation erfolgt über eine FIR[42]-Schaltung [98]. Die Implementierung von Non-Standard-Arithmetiken auf FPGAs ist also eine Möglichkeit, FPGAs zur Anwendungsbeschleunigung einzusetzen.

Nach Arbeiten [189] im Bereich der FPGA-gestützten Beschleuniger für Gleichungslöser auf Basis des CG-Verfahrens haben Lopes und Constantinides ihre Arbeit auf exakte Akkumulations- und Skalarprodukteinheiten zusätzlich ausgeweitet [188]. Der Platzbedarf ihrer hybriden Implementierung bestehend aus interner Festkommadarstellung reduziert sich auf weniger als ein Drittel gegenüber dem Einsatz vefügbarer Gleitkommaeinheiten zur Multiplikation und Addition.

Die Multiplikation dichtbesetzter Matrizen mittels Akkumulations- und Skalarprodukteinheiten in 64-bit-Gleitkommaarithmetik kann auf FPGAs von hoher Datenlokalität und FIFO-Puffern in Hardware profitieren [87]. Die Addition stellt den kritischen Pfad dar, der sich über die parallele Berechnung des vorderen Teils jeweils mit und ohne Übertrag von der hinteren Addition auflösen lässt. Aufgrund der gewonnenen Erkenntnisse sollten Matrixmultiplikationen auf FPGAs auch nach der gewöhnlichen Variante erfolgen, anstatt wie bei Strassen Multiplikationen durch weitere Additionen zu ersetzen [158]. [75] Mittels passender Parametrierung der Puffer lassen sich effiziente[43] FPGA-Designs auch trotz geringfügiger Bandbreite erstellen [181].

Operationen der BLAS-Bibliothek wurden auf einen Virtex-II Pro abgebildet [293]. Dabei wurden jedoch die Bandbreite und Pinanzahl entsprechend so definiert, dass der Durchsatz maximiert wird bei maximalem Ressourcenverbrauch. Dazu wurde eine Parametrisierung eingeführt über Bandbreite, Anzahl Einheiten und Ressourcen. Gängige Systeme besitzen im Gegenteil fest definierte Schnittstellen, etwa mit einer Bandbreite von 64 Bit pro Takt, so dass eine gänzlich andere Parametrierung passend zur Architektur des Systems anstelle der optimalen Parametrierung verwendet werden muss. Die Berechnungen der Latenzen und Bandbreiten ergaben bereits, dass für geringe Bandbreiten wenige Verarbeitungseinheiten ausreichen, um die maximale Leistung zu erzielen. Die wichtige Fragestellung, wie bei vorgegebener Schnittstelle (beispielsweise einem Bus) ein optimales FPGA-Design für BLAS-Operationen aussehen kann, welche Optimierungen dazu nötig sind, wurde somit nicht gelöst.

In nachfolgenden Arbeiten [294] wurde die obige Methodik der Parametrisierung von BLAS-Routinen bezüglich der Aufteilung auf ein MFS angewandt sowie auf die LR-Zerlegung übertragen. Es wurde ferner gezeigt, dass die Aufteilung der LR-Zerlegung auf mehrere FPGAs skaliert.

Die Berechnung der Hessenberg-Matrix zu einer gegebenen Matrix ist der erste Schritt der QR-Zerlegung, wie sie unter anderem bei der Bestimmung von Eigenwerten mittels des Lanczos-Verfahrens benötigt wird (vgl. Abschnitt 5.1.2). Sie ist mit doppelt genauer Gleitkommaarithmetik trotz starker Optimierung 3-4 mal langsamer auf einer SGI Altix RASC RC100 mit einem Xilinx Virtex-4 LX200 als auf einem System mit zwei Intel Xeon E5520, während eine GPU wie die Nvidia GTX 480 19,6 mal schneller sein kann [149]. Die QR-Zerlegung nach Givens ist ebenso auf FPGAs implementierbar [236]. Wie aus den Arbeiten erkennbar wird, hängt es von den Anwendungen und Hardwareeigenschaften wie insbesondere der möglichen Datenlokalität und damit einhergehend der ausnutzbaren, effektiven Speicherzugriffsbandbreite ab, ob und auf welchen Systemen Anwendungen von der Verlagerung auf FPGAs profitieren können.

Häufig wird versucht, die Verarbeitung dünnbesetzter Matrizen aufgrund ihres besonderen Formats zu beschleunigen. Für Bandmatrizen gelingt dies mit dem CG-Verfahren nur in dem Rahmen, wie

[42] Finite Impulse Response
[43] hinsichtlich Ressourcenaufwand und erzielbarer Leistung

die Qualität der verglichenen Softwarebibliothek ATLAS miteinspielt und nur sequentielle CPU-Ausführungen betrachtet werden [189]. Ein wesentliches Problem bei iterativen Verfahren ist die ständige Übertragung von Daten zum Akzelerator, so dieser selbige nicht mehr intern speichern kann [89]. Alternativ können viele Verfahren blockweise ausgeführt und die Blöcke parallel zueinander berechnet werden. Unter anderem ist für $128 \leq n < 500$ die LR-Zerlegung von $n \times n$-Matrizen mit doppelt genauen Werten auf einem Xilinx Virtex-II Pro XC2VP100 schneller als auf einem 2,2 GHz AMD Opteron [294].

Das iterative SRT-Verfahren, mit dem Divisionen und Wurzelberechnungen durchgeführt werden können, kann zwar platzsparend auf FPGAs implementiert werden [190], benötigt dann aber viele Iterationen sowie zusätzliche Takte zum Laden, Normalisieren und für weitere Operationen. Bei IBM beläuft sich dies auf insgesamt 39 Takte für Operanden mit doppelter Genauigkeit, wobei auch nur alle 34 Takte eine neue Division angestoßen werden kann [110]. Das SRT-Verfahren ist dennoch hervorragend zur Implementierung in FPGAs geeignet und kann auch unter heutigen Gesichtspunkten ausreichende Leistung durch Implementierung als Pipeline erzielen, wobei der Platzbedarf dann 10-20 mal höher ausfällt [278]. Die Hardwarebibliothek VFloat stellt Blöcke zur Division und Wurzelberechnung zur Verfügung, die mittels Taylorreihenentwicklung als Pipeline in VHDL implementiert sind [277]. Altera hat sich der Rundungsproblematik bei FPGA-Implementierungen von Divisionen angenommen [215].

Insgesamt erkennt man anhand der hiesigen Beispiele, dass die Implementierung von Standardoperationen für und in Standardarithmetik auf FPGAs nicht gewinnbringend ist. Am ehesten lassen sich Vorteile durch intern abweichende Implementierungen wie Taylorreihenentwicklung statt SRT-Verfahren oder interne Rechnung in Festkommadarstellung erzielen. Stencil-Operationen als zusammengesetzte Operationen außerhalb des Basisgleitkommabefehlssatzes eignen sich aufgrund ihrer vielen Berechnungen für ein einziges Ergebnisdatum sehr gut zur Verlagerung in Hardware. Indem die Genauigkeit durch kürzere Mantissen der Stencil-Koeffizienten geringfügig reduziert wird, können mehr Einheiten integriert werden für mehr Parallelismus und mehr Berechnungen auf den zwischengespeicherten Daten [136].

Erst bei Arithmetiken, die wie Galois-Felder von den Gleitkommastandards abweichen und somit in Standardprozessoren nicht vorhanden sind, nützt rekonfigurierbare Hardware. Rekonfigurierbare Hardware sollte daher zur Implementierung exakter Arithmetik geeignet sein. Dies wird im Folgenden genauer betrachtet sowie weitere erfolgreiche Beispiele zur Anwendungsbeschleunigung durch Verlagerung arithmetischer Spezialoperationen in die rekonfigurierbare Hardware.

Anwendungsbeschleunigung

Ist eine gute Speicherhierarchie des Akzeleratorknotens vorhanden, durch welche das ständige Übertragen von Daten zum Koprozessor entfällt, so kann ein einzelner Akzeleratorknoten für Finite-Differenzen-Probleme vergleichbare Leistung zu einem Cluster von 30 PCs liefern [91]. Wichtig dabei ist stets auch die optimierte interne Datenhaltung, die letztendlich die Möglichkeit zur Beschleunigung ausmacht [137].

Molekülsimulationen können von anpassbaren Formaten und Genauigkeiten auf FPGAs profitieren sowie von Wertetabellen, paralleler Berechnung, angepassten Architekturen mit Ausführung in einer Pipeline und nicht zuletzt auch von anwendungsspezifischen Reduktionseinheiten [120, 121, 141]. Dies ermöglicht bis zu 90-fache Geschwindigkeit gegenüber der Ausführung auf GPUs.

Das Lanczos-Verfahren kann auf FPGAs von der hohen Datenlokalität durch interne Speicherung und von vielen parallel ausgeführten Gleitkommaeinheiten wie n Skalarprodukten, die jeweils als Reduktionsbaum implementiert sind, profitieren [221]. Es erzielt durch die Spezialimplementierungen gegenüber sequentieller und paralleler Ausführung auf einem Prozessor wesentliche Geschwindigkeitssteigerungen. Solange bei GPUs die Daten mit CUBLAS vor und nach jeder Matrix-/Vektor-Operation neu übertragen werden müssen, übertrifft der ausgewählte und verwendete Xilinx Virtex-6 SX475T nicht nur Mehrkernprozessoren, sondern sogar die GPUs. Ein weiterer Ansatz, das Lanczos-Verfahren mithilfe von FPGAs zu unterstützen, ist die Implementierung in Festkommaarithmetik, so dass mehr Einheiten auf dem FPGA untergebracht werden können [157]. Dazu

wenden Jerez et al. einen Vorkonditionierer an, der die Wertebereiche der Matrix stark einschränkt und so überhaupt erst die Ausführung mit Festkommaarithmetik möglich macht.

2.2.3 Exakte Arithmetik

Beim Rechnen mit Kommazahlen in digitalen Rechnern gibt es eine Reihe von zu beachtenden Dingen [113], wie sie im vorherigen Unterabschnitt bereits angedeutet wurden. Insbesondere ist die Auflösung der Zahlen nur beschränkt möglich. Beispielsweise sind ohne Zusatzhardware keine periodischen Zahlen speicherbar; die maximale Auflösung beträgt 52 binäre Stellen im doppelt genauen Gleitkommaformat. Das standardmäßig eingesetzte, sehr leistungsfähige Gleitkommaformat nach IEEE 754 (vgl. Abschnitt 2.2.1) hat genau den einen Nachteil, dass die Mantisse, und damit die Auflösung, für die gesamte Zahl verwendet werden muss anstatt nur für die Nachkommastellen.

Die Wichtigkeit dieser Problematik zeigt sich darin, dass 2012 von Intel als Anmerkung zum Rechnen mit Vielkernprozessoren und damit einhergehend auch der Parallelisierung von Operationen wie Skalarprodukten die Thematik aufgegriffen und mit einem Beispiel illustriert wurde [69]. Ähnliche Probleme entstanden auch in den Anfängen des Rechnens auf Grafikkarten mit 8 Bit Auflösung, beispielsweise kann im Intervall $[-1;1]$ die Division $\frac{x}{y}$ zum Ergebnis $\frac{1}{y}$ haben [227]. Hilfreiche Abweichungen von den Standards, die nicht effizient in Mikroprozessoren ausführbar sind, wo also der Mehraufwand zur Ausführung den erzielbaren Nutzen übersteigt, können für rekonfigurierbare Hardware implementiert werden und dem Anwender dynamisch zur Verfügung gestellt werden. Folgend werden bekannte Verbesserungsansätze geschildert, diskutiert und verglichen sowie abschließend die Ausführungszeiten evaluiert.

Verbesserungsansätze

Einerseits beschäftigen sich Arbeiten mit der Angabe von Periodizitäten, wie $0,\overline{3}$ [52], andererseits gibt es Erweiterungen zu den Standard-Gleitkommaformaten und alternative Darstellungen, die im Folgenden beschrieben werden.

GNU Multiple Precision (GMP) ist ein bibliotheksbasierter Ansatz, bei welchem der Programmierer einen Gleitkommaformatdatentyp mit bestimmter Breite definiert, bspw. 128 Bit mit 11 Bit Charakteristik, 116 Bit Mantisse, also um 64 Bit gegenüber dem Standardformat mit 52 Mantissenbits verlängert, und dem Vorzeichenbit [117]. Auf dem Datentyp angewandte Operationen werden dann als mehrere auf den Standardtypen aufbauende Operationen durchgeführt. Neben GMP ist MPFR [101] recht bekannt [18].

„Staggered Arithmetik“[44]: GMP benötigt Platz auch für diejenigen Fenster, die vollständig mit Nullen belegt sind. Bei der staggered Arithmetik werden nur die nichtüberlappenden Zahlenbereiche, in denen von Null verschiedene Werte liegen, durch jeweils eigene Gleitkommazahlen ausgedrückt. So kann noch weitaus mehr Genauigkeit erzielt werden. Implementiert ist staggered Arithmetik beispielsweise in der Softwarebibliothek XSC [33].

Intervallarithmetik geht das spezifische Problem an, dass Gleitkommaformatimplementierungen nicht nur einige berechnete Bits verwerfen müssen, sondern auch das Ergebnis auf- oder abrunden, zu Null, zum nächstkleineren Wert, zum nächstgrößeren Wert oder zum betragsmäßig größeren Wert hin. Dabei kann es je nach Algorithmus und Daten geschickter sein, einen spezifischen dieser sechs Rundungsmodi einzusetzen. Dennoch ist der Programmierer und damit auch der Anwender nie davor gefeit, mehrfach hintereinander ungeschickt gerundet zu haben und damit ein vom nominellen Wert stark abweichendes Ergebnis zu erhalten [134]. Hier setzt die Intervallarithmetik an, welche bei jeder Operation die Streubreite des Ergebnisses als ein Intervall I von kleinstmöglichem Wert u und größtmöglichem Wert o angibt: $I = [u, o]$. Das Anwendungsgebiet ist vielseitig von ingenieursmäßigen Berechnungen über das Lösen von linearen Gleichungssystemen hin zur exakten

[44] engl., gestaffelt, abgestuft

Berechnung [225]. Das Produkt $L := I \cdot K$ errechnet sich vereinfacht dargestellt (von den zahlreichen Sonderfällen abgesehen) als

$$u_L := \text{abrunden}(u_I \cdot u_K)$$
$$o_L := \text{aufrunden}(o_I \cdot o_K)$$

Damit kann der Programmierer bereits bei der Entwicklung eines Programms Aufschluss über die Qualität seines Algorithmus oder Implementierung erlangen und ggf. daran arbeiten, die Intervallgröße durch Änderungen am Programmcode zu verringern und somit die Qualität zu verbessern, um z.B. numerische Stabilität oder geringere Laufzeiten zu erreichen, oder aber diese Methode zur Berechnung des Fehlers zur Laufzeit [199] einsetzen und entsprechend nötige Maßnahmen wie etwa die Neuberechnung mit anderen Auflösungen oder Verfahren durchführen lassen [205]. Dem Anwender hingegen kann ein solches Ergebnis Auskunft über die Qualität eines errechneten Werts geben [146] und über den möglichen Ergebnisbereich.

Der GAMM-Fachausschuss regte die Integration von Intervallarithmetik zur Verbesserung des IEEE-754-Standards für Rechnen im Gleitkommaformat an [107]; diese Anregungen wurden schließlich im Standard IEEE 754-2008 übernommen. Hardwareimplementierungen in Prozessoren benötigen etwa doppelt so viel Ausführungszeit für Intervallarithmetik wie für normale Gleitkommaoperationen [234, 9]. Kulisch gibt weitere Ideen und Hinweise zu möglichen Implementierungen der Intervallarithmetik [178].

Kahan warnt jedoch davor, Intervallarithmetik als Lösung für die Rundungsproblematik zu betrachten, und führt an, wie unterschiedliche Implementierungen und Compileroptimierungen unter Verwendung regulärer Rundungsmodi zu sehr genauen Ergebnissen kommen können oder zu äußerst falschen [163]. Mit Intervallanalyse lassen sich solche Fehler nicht erkennen. Daher kann es geschickter sein, gewisse Berechnungen ganz exakt durchzuführen.

Hardware-Implementierungen

An der George-Washington-Universität wurde untersucht, wie sich exaktere (nicht absolut exakte) Arithmetik in Hardware bringen lässt [93]. Untersucht wurde auf der SRC-6 MapStation mit Xilinx EDK 8.2 und VHDL auf FPGAs der Serie *Virtex XC2V6000-4*, getaktet mit 100 MHz, gegenüber der Software-Bibliothek GMP. Ihre Implementierung addiert fünf mal schneller als GMP und multipliziert neun mal schneller. Operationen wie die Division wurden bei beiden Arbeiten jedoch nicht betrachtet, sind allerdings ebenso von Bedeutung, wie die spätere Analyse des Lanczos-Verfahrens in Abschnitt 5.1.3 zeigen wird.

FloPoCo [80] erlaubt breitere Akkumulatoren, die auf Festkommaarithmetik basieren und frei parametrierbar sind, so dass sie ebenso zur Implementierung von Kulischs exakter Additionseinheit dienen können.

Kulischs Konzept des exakten Skalarprodukts [172] hat Bierlox mittels der Hardwarebeschreibungssprache VHDL auf einem Xilinx XCV800-4 mit vertretbarem Hardwareaufwand implementiert [31], um exakte Berechnungen an sich (oder wahlweise exakte Berechnungen des Fehlers) durchzuführen [177]. Die damalige Implementierung [31] war über den PCI-Bus als Koprozessor verwendet worden. Sie ermöglichte die exakte Multiplikation-Akkumulation von einfach genauen Operanden mittels einer internen Repräsentation als 768 Bits breites Festkommaregister, wobei die Quadrate der betragsmäßig kleinsten und größten darstellbaren Gleitkommazahlen (bei einfacher Genauigkeit) speicherbar und sogar im Gegensatz zum direkten Rechnen nach IEEE-754 deren Summen korrekt speicherbar sind. Die Assoziativität der Addition bleibt somit beim Rechnen erhalten; der Anwender muss keine aufwendige Quellcodeanalyse oder Codeumstellung durchführen.

	GMP	C-XSC	DP
Laufzeit (Ticks)	566.481 K	640.446 K	6.404 K
Laufzeit (μs)	177.089	200.211	2.001
M MACs/sec	5,921[1]	5,237[1]	523,810

Tabelle 2.2: Vergleich zwischen GMP mit 256 Bits, C-XSC, und Ausführung mit doppelter Genauigkeit. [1] Exakte MAC-Operationen.

Evaluation und Leistungsbewertung

Ein ausführlicher Vergleich unterschiedlicher Ansätze mitsamt Diskussion der zugrundeliegenden Ansätze wurde von Gowland und Lester durchgeführt [116], insbesondere mit dem Fokus auf Genauigkeit. Tabelle 2.2 der durchgeführten Untersuchungen zeigt, dass GMP und die Softwareimplementierung des exakten Skalarprodukts etwa zehn Mal langsamer sind als das Rechnen mit doppelter Genauigkeit auf derselben CPU.

2.3 Zusammenfassung

Vor dem Hintergrund der Verlagerung vornehmlich bioinformatischer und numerischer Anwendungen wurden in diesem Kapitel die Verwendung von rekonfigurierbarer Hardware und die dabei auftretenden Probleme sowie zu beachtenden Dinge beschrieben. Die Probleme liegen vornehmlich in der maximalen Ausnutzung der Bandbreite durch Datenwiederverwendung, in der Ausreizung der Bandbreite durch massiv parallele Berechnung sowie in der Verrichtung maximal vieler Operationen auf möglichst wenig Daten.

Lösungsansätze finden sich unter anderem mit datengetriebener und taskparalleler Ausführung von Anwendungen, welche komponentenbasiert entwickelt werden. Weitere Ansätze sind die Verlagerung von Spezialoperationen, welche die Daten vielfach wiederverwenden können, wie Stenciloperationen, oder welche sehr viele Operationen ausführen mit nur wenig Eingabedaten, wie etwa exakte Skalarproduktimplementierungen.

In den nächsten Kapiteln werden Anwendungen unter Einsatz des in diesem Kapitel erworbenen Wissens vereinfacht und umgestaltet, um erfolgreich in rekonfigurierbare Hardware ausgelagert werden zu können. Die Beschleunigung beliebiger Anwendungen durch Verlagerung aufwendiger Teile in rekonfigurierbare Hardware soll auch ohne Anwendung jenes Wissens möglich werden, damit Domänenexperten auf schnelle und bequeme Weise jeweils passende Akzeleratoren für unterschiedlichste Anwendungsfälle einsetzen können. Basis dafür kann ein Bibliotheksansatz mit vorgefertigten Akzeleratorportierungen sein, die von Hardwarespezialisten entwickelt wurden. Zur Ausführung werden FPGA-basierende heterogene Systeme verwendet, wobei die FPGAs die Bibliotheksbausteine vorhalten sollen und über Mikroprogramme die Verschaltung erreichen sollen. Um von verwandten Arbeiten abzugrenzen, beschreibt das nächste Kapitel 3 den Stand der Forschung in diesen Gebieten.

Kapitel 3

Verwandte Arbeiten

In dieser Schrift werden bioinformatische und numerische Anwendungen hinsichtlich der Verlagerung in rekonfigurierbare Hardware analysiert. Im vorangegangen Kapitel wurden bereits einige wesentliche Aspekte zur Verlagerung von Algorithmen in Beschleunigerhardware beschrieben. Bevor in Kapitel 4 die geschilderten Lösungsansätze und auftretenden Probleme bei der Verlagerung eines Algorithmus in Hardware anhand der Portierung eines bioinformatischen Kernels in rekonfigurierbare Hardware aufgezeigt werden, zeigt dieses Kapitel den Stand der Forschung auf. So grenzt es auch von den verwandten Arbeiten ab, um den individuellen Beitrag aufzeigen und die Motivation zu der vorliegenden Arbeit festigen zu können. Der Stand der Forschung umfasst hochsprachliche Konvertierungswerkzeuge ebenso wie Alternativen zu FPGAs sowie Programmiermodelle und Laufzeitsysteme. Eine Diskussion der hinter dieser Arbeit stehenden Motivation, der verwandten Arbeiten und der bleibenden Fragestellung schließt dieses Kapitel ab.

3.1 Konvertierung von hochsprachlichen Beschreibungen zu hardwarenahen Beschreibungen

Detaillierte Hardwarekenntnisse sollten nicht von einem Entwickler von beschleunigernutzenden Anwendungen („Anwendungsexperten“) verlangt werden, da dies nicht seinen Aufgabenbereich darstellt, wie es Strzodka sieht [249]. Ein möglicher Ansatz dazu ist, die Programmierung von Hardware über eine Hochsprache zu gestalten.

3.1.1 Quelltextkonverter von einer Hochsprache zu einer Hardwarebeschreibung

Hardwarebeschreibungssprachen wie Verilog oder VHDL sind im Gegensatz zu herkömmlichen Programmiersprachen wie C, Java, Python usw. inhärent parallel, aber wiederum explizit sequentiell. Dem versuchen Hochsprache-zu-Beschreibung-Konvertierer[1] entgegenzuwirken, indem der Nutzer in einer C-ähnlichen Programmiersprache wie ArchC [17], Impulse C, *Riverside Optimizing Compiler for Configurable Computing* (ROCCC) [47, 48, 122] oder gar wie bei Maxeler komplett in Java den Algorithmus angibt. Zur jeweiligen Hochsprache zugehörige Werkzeuge konvertieren dann die hochsprachliche Formulierung zu VHDL oder Verilog. Anschließend lässt sich der Code eventuell sogar in Softwaresimulationen und Emulationen testen.

Das Prinzip der Konverter ist, voneinander unabhängige Operationen parallel auszuführen oder teilabhängige Operationen teilüberlappt, etwa in einer Pipeline mit Weiterleitung[2] der zuletzt berechneten Werte. Allgemein anerkannt ist, dass das Programmieren von parallelem Code, was bei der Hardwareentwicklung ja vorherrschend ist, weitaus schwieriger als das schon nicht einfache Programmieren von sequentiellem Code ist [41]. Zustandsautomaten stellen ein Mittel zur Zwangssequentialisierung von Operationen dar. Die Überlappung der Ausführung erfolgt bei Impulse C und ROCCC daher nicht ausschließlich datengetrieben, sondern wird zusätzlich durch einen sehr großen Zustandsautomaten gesteuert.

[1] engl. HLL-to-HDL, C-2-RTL u.ä.

[2] engl. „Forwarding“ im Kontext von Pipelines; bei iterativen Kernels als „Feedback“ bezeichnet

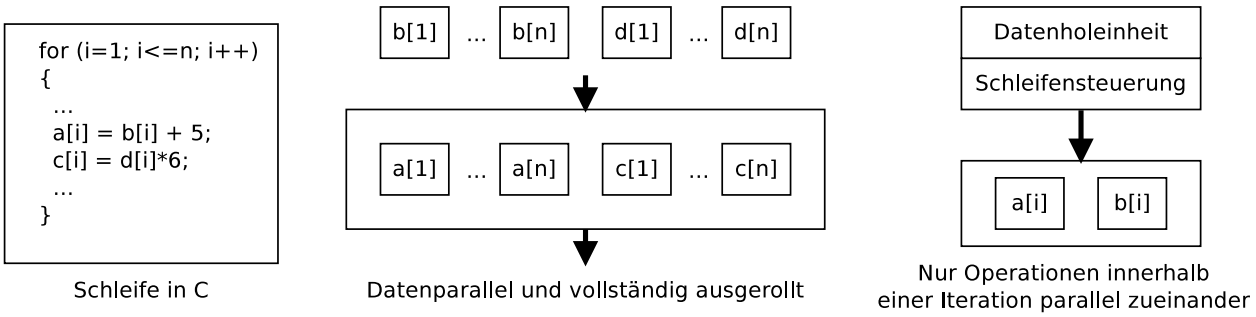


Abbildung 3.1: Eine in hochsprachlichem Code spezifizierte Schleife kann zu mehreren feingranulareren Rechneinheiten konvertiert werden oder zu einer Rechenpipeline.

Eine Iteration einer in C spezifizierten Schleife (links) könnte parallel ausgeführt werden oder als Pipeline mit Ergebnisweiterleitung, wie Abbildung 3.1 veranschaulicht. Da größere Datenmengen überwiegend nicht adhoc bereitstehen, wird eher eine Pipeline synthetisiert als dass mehrere Exemplare zur Ausführung des Schleifenrumpfes genutzt würden. Dies hat negative Auswirkungen auf die in rekonfigurierbarer Hardware unbedingt auszunutzende Parallelität und erfüllt nicht das unausgesprochene Ziel der Autoparallelisierung von Code. Damit ist nun als Forderung aufgestellt, dass mehrere arithmetische Einheiten parallel arbeiten müssen, indem z.B. Daten aus unterschiedlichen Quellen bezogen werden.

Altera ermöglicht Entwicklern, mit OpenCL beschriebene Kernel zu Verilog zu konvertieren, und erzielt bereits für FPGAs hohe Taktraten von mehr als 160 MHz [71]. Um OpenCL erfolgreich einzusetzen, sind allerdings hardware-bewusste Konfiguration und Programmierung nötig.

Die HoneyComb-Architektur stellt eine generische Architektur zur Erforschung neuartiger Rechnerorganisationen dar, in der Einheiten wabenartig arrangiert sind und vergleichbar der DodOrg-Architektur [26] unterschiedliche Aufgaben erfüllen. Im Rahmen von HoneyComb wurde die „HoneyCombLanguage“ [259] entwickelt. Diese Sprache basiert auf VHDL und hat unterschiedliche Arten von Prozessen, die auf die entsprechenden Hardwareeinheiten wie Verarbeitungseinheiten oder Speicherelemente dann abgebildet werden. Eine derartige Beschreibung kann anschließend für einen FPGA oder ein Netzwerk von FPGAs synthetisiert werden.

Dime-C wurde zur Auswertung von Antennensignalen auf Nallatech FPGA-Boards eingesetzt [282]. Die gewonnenen Erkenntnisse sind, dass es zur FPGA-Nutzung einerseits passender Anwendungen bedarf und andererseits der Ausnutzung der verfügbaren Speicherstrukturen mittels doppelter Speicherung[3], was laut den Autoren mit Dime-C keine einfache Aufgabe sei.

Durch einen auf CoSy[4] basierenden Ansatz können 4,4 mal schnellere Hardware-Beschreibungen aus regulärem C-Code erzeugt werden [202].

LALP [195] geht das Problem an, ausreichend Parallelismus im erzeugten RTL-Code auszunützen. Dazu wurde einerseits eine domänenspezifische Sprache entwickelt, und andererseits aggressives Pipelining von Schleifen versucht, indem auf große Zustandsautomaten verzichtet wird. Allerdings ist LALP sehr hardwareorientiert bis zur Abfrage von Aktivierungssignalen hin und somit für Anwendungsentwickler nicht geeigneter als die anderen bislang vorgestellten Sprachen.

Zwar wird durch den Ansatz der Quelltextkonverter das Erzeugen der Hardwarebeschreibungen abgenommen, der gesamte Prozess ist aber mitunter langwieriger, und keineswegs ermöglicht er, hardware-unbewusst[5] zu entwickeln. Denn die Verwendung von Konverterwerkzeugen bedeutet, dass das V-Modell mehrfach durchlaufen wird: Zunächst muss die Funktionalität verifiziert werden mittels Simulation der hochsprachlichen Beschreibung, dann wird der Prozess für die Emulation der Hardware auf Basis der Hardwarebeschreibungssprache wiederholt, und abschließend erfolgen erst die Hardwareerzeugung und der Test mit echter Hardware.

[3] engl. double buffering
[4] URL: `http://www.ace.nl/compiler/cosy`
[5] engl. hardware-unaware – im Gegensatz zu hardware-aware

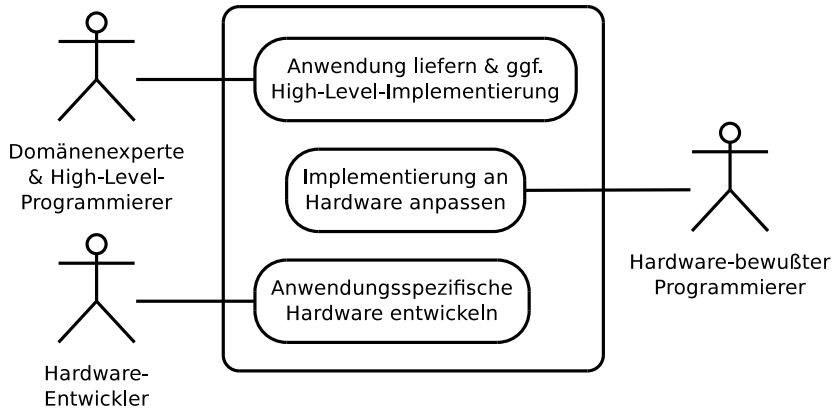


Abbildung 3.2: Kollaborationsdiagramm beim Verwenden rekonfigurierbarer Hardware mittels hochsprachlicher Programmierungsansätze.

Domänenexperten (vgl. Abb. 3.2) können rekonfigurierbare Hardware also mittels solcher Konverter programmieren. Wie sich bereits in der aufgeführten Literatur zeigte, bedarf es dazu jedoch erst der ganzheitlichen Berücksichtigung der Merkmale und Eigenschaften eines Systems wie Speicherstrukturen, Speicheranbindung, NUMA-Aufbau, Speichergrößen, Bandbreiten und anschließend der passenden Ausnutzung selbiger, bevor auch mit solchen Konvertern Beschleunigung erzielbar ist. Dies ist Aufgabe der Entwickler mit Hardwareverständnis bzw. der Hardwareentwickler selbst (vgl. Abb. 3.2). Als Hardwareentwickler wiederum, so hat sich gezeigt [159], ist nach Entfernen von Kontrollfluss, Verringern der Auflösung der Gleitkommazahlen, Vorberechnung von Werten, Einsatz von Pipelining und schließlich Verwenden mehrerer Exemplare der Rechenkernels z.B. mit Handel-C als Hochsprache zur Hardwarebeschreibung durchaus Leistungssteigerung unter Verwendung von Quelltextkonvertern erzielbar. Die zu lösende Frage ist jedoch, wie Domänenexperten ohne dieses Spezialwissen und ohne aufwendige Verfahren Nutzen aus rekonfigurierbarer Hardware ziehen können.

Ein Ansatz mit Java und in Java implementierten Ergänzungen als rückwärtskompatibler Bibliotheksansatz wird mit Liquid Metal [150] verfolgt. Der Java-Code wird für PowerPC und Beschleuniger übersetzt, wobei für die Beschleunigereinheiten weitere Compiler den von Limeade erzeugten Hardware-Zwischencode erst noch zu einer Netzliste übersetzen müssen. Das Ziel dieser Arbeit ist, Just-in-Time-Compilation zu erreichen, dass also zur Laufzeit Code für die Ausführung als dedizierte Hardware-Schaltung generiert wird. Ähnliches wurde ebenso von Lysecky et al. mit dem WARP-Prozessor untersucht [191] mit dem Ergebnis, ressourcenschonenender, energieffizienter und schneller zu sein als herkömmliche HW-SW-Partitionierungsansätze und Prozessorarchitekturen. Diese beiden Ansätze sind sehr attraktiv, da sie eine Methodik bieten, die Akzeleratoren zielgerichtet und gewinnbringend einzusetzen.

Maxeler hat auf Basis seines Quelltextkonverters ein weiteres Werkzeug entwickelt, das domänenspezifisch den Code übersetzen helfen soll[6], um den Abstraktionsgrad zu erhöhen und FPGA-Hardware direkt vom Domänenexperten effizient programmierbar werden zu lassen. Dadurch sollten sich unterschiedliche Stencil-oder Faltungs-Implementierungen als hinsichtlich der Koeffizienten parametrierbare Funktionen aufrufen lassen. Die alleinige Auslagerung einer Stencil-Operation beschleunigt jedoch nur unter wie von Maxeler geschilderten Voraussetzungen wie besonders großer Datenmengen die Gesamtanwendung, ansonsten ist der Mehraufwand durch die Datenübertragung zu hoch und das Verhältnis von Rechenoperationen zu Speicheroperationen auch bei Stencils bzw. Faltungen zu gering.

Mit TANOR [171] wurde ein domäneneingeschränktes Hardware-Übersetzungswerkzeug für MATLAB[7]-Code geschaffen, das zudem in der Lage ist, numerische Instabilitäten zu berücksichtigen und den entstehenden Hardware-Code mit mehr Bits für erhöhte Genauigkeit auszustatten, während gleichzeitig die Hierarchie zum Zwecke des Debuggings erhalten bleibt. Neben MATLAB hat der

[6] URL: `http://www.maxeler.com/products/software/maxgenfd/`
[7] URL: `http://www.mathworks.de/products/matlab/`

Hersteller MathWorks auch sein Produkt Simulink[8] um Fähigkeiten zur Erzeugung von HDL-Code erweitert.

Der Stil der von Quelltextkonvertern synthetisierten Hardwarebeschreibungen ist *kombinatorisch*, d.h. zentral sind ein oder mehrere Zustandsautomaten, deren Übergänge durch Logik oder Arithmetik berechnet werden. Ihre Zustände dienen als Eingänge an Multiplexern zwischen Datenleitungen und Registern und steuern derart den Daten- und auch Kontrollfluss. Ist ein Zustand redundant, so lässt er sich kaum händisch entfernen; noch lässt sich ein neuer Zustand bequem einführen oder sonstige Änderungen durchführen. Die Zustände sind aufgrund der automatischen Erstellung auch fortlaufend benannt, so dass die Hardwarebeschreibung nicht mehr verständlich ist. Dies hat zur Folge, dass Simulation ausschließlich softwarebasiert vorgenommen werden kann und muss, bevor die Hardwarebeschreibung erstellt wird, denn ab der Konvertierung ist keine Überprüfung des erzeugten Codes mittels Emulation durchführbar, da die Variablennamen und Zustände dem ursprünglichen Code nicht für Menschen zuordenbar sind. Statt der Emulation ist nach der Konvertierung der vollständige Prozess der Bitstreamerzeugung mit einer Dauer von häufig mehreren Stunden nötig, bevor in echter Hardware die erzeugte Schaltung auf Korrektheit überprüft werden kann. Damit fällt der Schritt der a-priori-Validierung durch Hardwaresimulation aus. Somit besteht keine nahtlose Methodik mehr.

Der aktuelle Stand bei den gängigen Werkzeugen bietet das benötigte mehrfache Unterbringen von Funktionseinheiten nicht an. Deshalb ist von automatisierten Werkzeugen keine hohe Rechenleistung und somit kein Nutzen erwartbar, sondern erst wieder unter Mitwirkung des Anwendungsentwicklers – der aber an der konkreten Hardware-Umsetzung eigentlich nicht beteiligt sein soll. Durch die großen Zustandsautomaten zur Steuerung des Datenholens, der Einheitenaktivierung und der Schleifenzähler wird sehr viel zusätzliche Logik synthetisiert. Damit eignen sich Konverter primär, wenn bitlastige Operationen auf wenig Daten duchzuführen sind, wo FPGAs ohnehin prinzipiell geeigneter sind als herkömmliche CPUs. Erste Weiterentwicklungen zeichnen sich dahingehend ab, dass nach dem Datenflussmodell Daten transportiert werden und darüber auch Kontrollfluss besser abgebildet werden soll [72]. Der hohe Platzbedarf automatisch erzeugter Schaltungen kann auch reduziert werden, indem das Initiierungsintervall etwas größer wird oder die Schaltung nicht vollständig als Pipeline umgesezt wird [239].

Für den sich in die Konvertierung einarbeitenden Hardwareexperten bzw. für solche Entwickler, die Hardware bislang mittels herkömmlicher Hardwarebeschreibungssprachen entwickelten, ist dieses Vorgehen in der Summe weitaus zeitaufwendiger. Für Domänenexperten erfüllt die Konvertierung von Hochsprachen zu Hardwareeschreibungen zwar den Zweck, dass die rekonfigurierbare Hardware nutzbar wird, die Fehlersuche und -behebung sind aber kaum ohne Fachwissen über Schaltungen und Systemaufbau und Funktionsweise der eingesetzten Übersetzungswerkzeuge durchführbar [29]. Entsprechend lässt sich Beschleunigung nur sehr schwer erhalten.

AutoESL AutoPilot hilft über das Ziel der Vereinfachung der Hardwareentwicklung hinaus, 11 %-31 % der FPGA-Ressourcen einzusparen. Mit drei Mannmonaten alleine für die Optimierung des mit AutoPilot implementierten Algorithmus zur Bereichserkennung von Funksignalen ist aber die Behauptung, die Produktivität werde gegenüber manuellem Entwurf mit einer Hardwarebeschreibungssprache erhöht, nicht haltbar [58]. Mit Impulse C als Hochsprache ist eine Implementierung eines Stencil-basierten Vorkonditionierers [HKN+11] 100-200 mal langsamer als die Softwarevariante. Erst wenn Optimierungen verwendet werden wie Wiederverwendung alter Werte (Feedback-Variablen), wenn Pipelining gezielt angegangen wird, wenn Schleifen ausgerollt werden, wenn der Datentransfer berücksichtigt wird, wenn Transfer und Berechnung ausreichend überlappen, wenn langsame arithmetische Operationen mit schnelleren ausgetauscht werden (Division anstelle Multiplikation mit inversem Element), wenn die Kommunikation gezielt minimiert wird, wenn die Indizierung von Speicherzugriffen effizient, d.h. mit Minimalaufwand zur Berechnung der Indizes, durchgeführt wird, und wenn konstante Startwerte bei den Schleifendurchläufen existieren, ist Beschleunigung möglich. All diese notwendigen Optimierungen werden nicht von den Werkzeugen geleistet. Dies wird genauer in Abschnitt 5.8.4 beleuchtet. Bedenkt man, dass die erreichbare Leistung des eingesetzten Systems etwa Faktor 10 über der Softwarevariante liegt, so ist eine Diskrepanz von Faktor 2000 zwischen der Entwicklung über den Konvertierungsansatz ohne Optimierungen und der hardwarebewussten Entwicklung eines Designs feststellbar.

[8] URL: `http://www.mathworks.de/products/simulink/`

Erste Ansätze für hochsprachliche Werkzeuge helfen also durchaus, eine geeignete Systemarchitektur für eine FPGA-Schaltung zu entwerfen [288]. Sie benötigen aber detaillierte Angaben über die umgebende sowie die zu entwerfende Hardware. Daher sind wieder umfangreiche Hardwarekenntnisse und entsprechende Erfahrungen trotz der Verwendung einer Hochsprache von Nöten, um keine Verlangsamung zu erreichen. Konvertierungswerkzeuge für Hochsprachen zu Hardwarebeschreibungssprachen erfüllen daher noch nicht die Anforderung, den Anwendungsentwickler von der Hardwareprogrammierung zu entlasten.

Es ist nötig, Anwendungs- und Domänenexperten für den Einsatz rekonfigurierbarer Akzeleratorhardware Möglichkeiten an die Hand zu geben, mittels derer sie abstrahiert Hardwareschaltungen entwickeln können, bei denen die geschilderten notwendigen Optimierungen sowie Hardwareangaben nicht mehr manuell zu spezifizieren sind. Der vorgeschlagene Ansatz ist, Bibliotheken mit Funktionsimplementierungen sowohl in Software als auch in Hardware anzubieten, die nahtlos ausgetauscht werden können [BKNK09]. In Hardware kann eine geschickte pufferbasierende Verbindung mehrerer Recheneinheiten für die datengetriebene und taskparallele Ausführung sorgen [NBK+13, NBK14].

3.1.2 Automatisierte Erzeugung von Gleitkommaschaltungen

Nebst der Erzeugung von allgmeinen, beliebigen Schaltungen aus hochsprachlich formulierten Beschreibungen von Algorithmen oder gar ganzen Anwendungen ist die Erzeugung von Schaltungen für Gleitkommaberechnungen besonders interessant. Die Generierung parametrierbarer Multiplizier-Einheiten wurde angepasst an die Architektur und Eigenschaften eines Xilinx XC4000 [180]. Die in dieser Arbeit um 1995 erreichten Bitbreiten der Eingabedaten lagen bei 8 Bit, die Latenz lag dabei zwischen 80 ns und 120 ns. Heutzutage sind Eingabebreiten von 53 Bit (Breite der Mantisse im „IEEE-754 Double Precision“-Format) mit 200 MHz (5 ns) in 10 – 20 Takten auf einem Xilinx Virtex-6 problemlos umsetzbar.

FloPoCo generiert Gleitkommaschaltungen mit der benötigten Genauigkeit, vor allem hilft dies bei der Akkumulation von Zahlen und Produkten, wie sie bei Monte-Carlo-Simulationen bzw. Matrix-Vektor-Multiplikationen vorkommen [80]. Einzelne Operationen lassen sich zu einer größeren ganzen Schaltung verbinden, bei der intern auf Runden/Abschneiden weitestgehend verzichtet wird und dadurch höhere Genauigkeit bei gleichzeitig vergleichsweise geringem Platzbedarf erreicht wird [81]. Schlussendlich ist FloPoCo hervorragend dazu geeignet, für eine gewisse Menge an verschiedenen FPGAs in VHDL beschriebene Schaltungen zur Berechnung von Termen wie $x^2 + y^2$ zu erzeugen, anstatt sie mühsam selbst durch das Verdrahten mehrerer Gleitkommablöcke zu erstellen [82].

Der Trident Compiler ist ein Framework [262, 263], dessen wesentliche Leistung in der prädikativen Ausführung unterschiedlicher Kontrollpfade bei zusammengesetzten Gleitkommaschaltungen besteht, so dass der Schleifenrumpf in einer Pipeline ausgeführt werden kann. Trident ermöglicht die Verwendung quasi beliebiger Gleitkommabibliotheken. Beispielsweise könnte VFloat [277] eingesetzt werden, das sich durch gepipelinte Implementierungen von Divisions- und Multiplikationseinheit sowie durch eine Akkumulatorimplementierung auszeichnet.

Den Ansatz, Gleitkommaoperationen miteinander zu einer großen Pipelineschaltung zu verschmelzen, verfolgt auch Alteras Fused Datapath [213]. Der Unterschied ist jedoch, dass Altera intern in Festkommadarstellung rechnet und obendrein compiler-basierte Analyse des Genauigkeitsbedarfs liefert, so dass die Schaltungen wesentlich kleiner und damit auch energieeffizienter werden können. Die festkommabasierte Rechnung kann zwar durchaus mehr Platz hinsichtlich Datenwortbreiten benötigen, aber dafür lassen sich die Operationen wesentlich einfacher implementieren mit besseren Laufzeiten.

Diese Werkzeuge können insbesondere von Hardwareentwicklern dazu genutzt werden, schneller bessere Hardwarekomponenten zu entwickeln. Zu lösen bleiben jedoch die effiziente nahtlose Verbindung mit anderen Einheiten, die effiziente Integration in das Hostsystem, die Wiederverwendbarkeit und der effiziente Einsatz durch Anwendungsentwickler. Eine geeignete Verbindung kann ein zentraler Puffersatz sein, der wie eine Crossbar mehrere Funktionseinheiten entsprechend der programmierbaren Konfiguration verbindet [NBK+13, NBK14].

3.1.3 Compiler-basierende Extraktion von Operationen

Alternativ zum Versuch, die implizite Parallelität der Hardwareprogrammierung mittels eines sequentiellen Programmiermodells zu verstecken, kann auch der Ansatz gewählt werden, die Parallelität eher komponentenbasierend zu nutzen. Dabei erkennt ein Compiler, dass er Operationen verlagern kann und dass diese als Operationen eines Schleifenrumpfs unabhängig voneinander sind über die Iterationen hinweg, und erstellt Koprozessorcode auf Basis einer Bibliothek von unterstützten Funktionen. Der generierte Code wird dann auf vielen Koprozessoren gleichzeitig ausgeführt (vgl. Abbildung 3.1 mittig). Dies ist möglich, wenn ausreichende Speicheranbindung vorhanden ist, wie auf der Convey HC-1 zum Beispiel acht separate Speichercontroller mit je zwei Kanälen und einer maximalen Bandbreite von 76,8 GB/s, wenn alle vier FPGAs über alle 16 Kanäle auf den Speicher zugreifen. Entsprechend bietet Convey eine Konfiguration für die vier FPGAs („Application Engines“) ihres Koprozessors an, welche Vektorverarbeitung unterstützt. Die Operationen sind dann Vektoroperationen auf möglichst großen Vektoren.

Ähnlich diesem Ansatz kann ein Laufzeitsystem basierend auf Funktionsattributen [BKNK09, NBK09] wie Methodenname und Signatur alternative Funktionsimplementierungen aussuchen [KNBK12], solange sie die notwendigen Eigenschaften erfüllen [NKBK10]

3.1.4 Eignung zur Verlagerung eines Algorithmus in rekonfigurierbare Hardware

Die genannten Quelltextkonverter sowie gleichermaßen die vielen weiteren Quelltextkonverter können allesamt einen Hardwareentwickler dabei unterstützen, schneller zu einem ersten Prototyp einer Schaltung zu kommen und gegebenfalls sogar das finale Design zu entwickeln. Sie können bereits bei arithmetischen Schaltungen Logik durch Analyse der benötigten Genauigkeit einsparen, betreiben sehr viel Pipelining, erkennen bzw. nutzen Muster wie systolische Arrays und erzeugen passend für FPGAs kombinatorische Schaltungen, die von Zustandsautomaten aus gesteuert werden. Im nächsten Kapitel wird ersichtlich, dass die Verlagerung erst dann Vorteile bringen kann, wenn auch die zu verlagernden Algorithmen angepasst und umgestellt werden. Quelltextkonverter vermögen daher also nicht, einen auf einen Allzweckprozessor zugeschnittenen, hochsprachlich formulierten Algorithmus zu einer Hardwarebeschreibung zu konvertieren, durch welche Leistungssteigerung erzielbar sein wird.

3.2 Alternativen zu FPGAs: Coarse-Grained Reconfigurable Arrays

Coarse-Grained Reconfigurable Arrays (CGRAs) sind Architekturen aus mehreren miteinander verbundenen Elementen, die häufig gitterartig angeordnet sind. Die Elemente sind im Gegensatz zu FPGAs Logik- oder Arithmetikelemente mit erhöhter Funktionalität wie Addition von Festkommazahlen. CGRAs eignen sich gut zur Ausführung von Datenflussgraphen bzw. -subgraphen.

PACT-XPP [25] ist eine einst kommerziell erhältliche Architektur, bei der die Elemente hierarchisch angeordnet und organisiert sind. Sie zeichnet sich durch die Behandlung der Konfigurationen hinsichtlich Vorladen, Zwischenspeichern und Nebenläufigkeit mit anderen Berechnungen oder Konfigurationsvorgängen auf anderen sogenannten Clustern aus. Der Entwurfsfluss bei der Entwicklung und Programmierung von Anwendungen für XPP bedingt die Partitionierung der Anwendung, da XPP zumeist von einem Hostprozessor aus angesteuert wird. Aufgrund der verhältnismäßig kleinen Speicherzellen (je Cluster 16 Speicherzellen mit nur je 1 KB beim XPU128-ES) müssen bei größeren Anwendungen mit größeren Datenmengen häufig Daten zum Prozessor zurückgeschrieben werden. Somit sind geschickte Datenhaltung, Optimierung der Zugriffe und Übertragungen bei PACT-XPP besonders wichtige Aspekte. FPGAs hingegen verfügen über weit mehr internen Speicher und die Möglichkeit, viele Schnittstellen zu externen Speichern wie SRAMs, bei denen die Latenz nur einen Takt beträgt, zu integrieren.

ADRES [40, 193] ist ein weiteres Konzept für einen CGRA. Die Architektur besteht aus einem 8×8-Gitter von Funktionseinheiten. Die erste Reihe verfügt zudem über eine Speicherzugriffsschnittstelle. ADRES kann als VLIW-Prozessor eingesetzt werden, dann sind die weiteren sieben Zeilen der „rekonfigurierbaren Matrix“ allerdings nicht aktiviert. Unterstützung für den Entwickler ist sowohl

für die sogenannte rekonfigurierbare Matrix als auch für den VLIW-Prozessor bereits vorhanden. Parameteraustausch erfolgt über wiederverwendete Register. Die Architektur ist allerdings nicht skalierbar, da die Leistungssteigerung mit Faktor 2 bis 6,4 bei Verwendung der achtfachen Menge an Ressourcen gering ausfällt. Im Gegensatz zu XPP ist ADRES nicht als Chip implementiert.

Eine FPGA-Implementierung eines CGRAs ist die Arbeit von Ghazanfari et al. [111]. Gegenüber PACT-XPP und ADRES sind aber zahlreiche FIFOs vorhanden, um den Datentransport und die -haltung für Streaming-Anwendungen wesentlich zu verbessern. Da bis zu vier FIFOs verwendet werden können, sind auch Operationen mit drei Operanden wie Multiplizieren-Addieren (d:= a*b+c) in einem Knoten umsetzbar.

$(GECO)^2$ stellt schließlich eine grafische Programmierumgebung für CGRAs [217]. Sie erzeugt eine Konfiguration, mittels derer die verfügbaren Funktionsblöcke auf den CGRA gebracht und parametriert werden, sowie eine zugehörige eindeutige Konfigurationsnummer. Anwendungen, welche diese Konfigurationsnummer benötigen, sind nun ausführbar. Die grobgranularen Funktionsblöcke werden mittels Mikroprogrammen angesteuert und so unterschiedliche Funktionalitäten in den Blöcken ausgeführt sowie die Ein-/Ausgaben passend verbunden. Damit ist weder die geschickte Datenhaltung bzw. Datenwiederverwendung noch die Minimierung des Datenaustauschs mit externen Speichern gelöst.

Multi-Purpose Processor Arrays (MPPAs) wie der Kalray MPPA 256 gelten als flexibler als CGRAs, da die Prozessoren unabhängig voneinander andere Unterprogramme oder Threads ausführen können. Bei CGRAs werden die Verarbeitungseinheiten im „Lock-Step“-Modus ausgeführt, was bedeutet, dass zum gleichen Zeitpunkt auf allen Einheiten der gleiche Thread und der gleiche Basisblock (z.B. Knoten im Datenflussgraphen) ausgeführt werden. Die Kombination von MPPAs und CGRAs derart, dass die CGR-Einheiten voneinander unabhängiger werden, verspricht die Programmierung zu vereinfachen und die Ressourcennutzung zu verbessern [211]. Der Zusatzaufwand in Hardware betrage dabei lediglich 2 %. Der in dieser Dissertation gewählte Ansatz ist daher auch die gleichzeitige Ausführung mehrerer Einheiten, die unterschiedliche Funktionalität bieten.

Es ist also nicht zwangsläufig nötig, unmittelbar die Schaltungen für die rekonfigurierbare Logik eines FPGAs zu entwickeln, sondern es kann auch einen gangbaren Weg darstellen, gezielt Einheiten von größerer Granularität als CLBs anzusteuern, die auf einem FPGA bereits von einem Hardwareentwickler aus untergebracht wurden. Reine CGRAs hingegen sind hingegen nicht ausreichend zur Beschleunigung vielfältiger Anwendungen in unterschiedlichen Domänen, da nicht ausreichend viel Rekonfigurierbarkeit gegeben ist, sich stets den aktuellen Anforderungen anzupassen mit neuen Funktionseinheiten und anderen Verdrahtungen.

3.3 Programmiermodelle und -sprachen für heterogene Systeme

Neben reinen Quelltextkonvertern gibt es auch zahlreiche andere Ansätze, die Programmierung heterogener Systeme zu vereinfachen und zu verbessern, damit Anwendungsentwickler schnell und bequem zu leistungsfähigen Gesamtlösungen unter Einsatz von Beschleunigerhardware gelangen. Die Vielzahl von Programmiermodellen für das Programmieren von Systemen mit Akzeleratoren und für das Rechnen mit diesen gebietet die Einteilung der Programmiermodelle in verschiedene Rubriken. Das große Interesse und die Notwendigkeit haben dazu geführt, dass diverse (Konsortial-)Standards entstanden sind und auch weiterhin entwickelt werden. Zum ersten werden daher Konsortialstandards für die Programmierung von heterogenen Systemen betrachtet, zum zweiten dann kommerzielle und industrielle Programmiermodelle für heterogene Systeme. Abschließend werden akademische Ansätze diskutiert. Wie man sehen wird, haben sie alle bereits einige wesentliche Lösungen für die im letzten Kapitel aufgezeigten Probleme gefunden. Sie sind jeweils aber noch nicht ausreichend, das erklärte Ziel maßgeblich als erfüllt betrachten zu können.

3.3.1 Konsortialstandards für die Programmierung von heterogenen Systemen

OpenCL (Open Computing Language)[9], das von der Khronos Group entwickelt wurde, ist ein Ansatz zur Verbesserung der Portabilität. Die Sprache soll Abhilfe dabei schaffen, dass vorhandene Codes nicht auf jeder GPU gleichermaßen effizient laufen und darüberhinaus auch nicht auf alternativen Architekturen wie ClearSpeed, Cell B.E. oder FPGAs sofort ausführbar sind. Dies soll erreicht werden, indem ein hardware-unspezifisches, grafikkartennahes Programmiermodell eine gewisse Abstraktion sowie Befehle zur Handhabung des Kernels bietet, so dass die Kernels portierbarer werden können [167]. Dabei werden Kernels dynamisch für die ausgewählte Zielarchitektur zur Laufzeit kompiliert, Ein- und Ausgabepuffer definiert und die übersetzten Kernels auf der Zielhardware unter Verwendung der Puffer im gemeinsamen Speicher ausgeführt [56]. Dennoch liefern in OpenCL implementierte Algorithmen die Maximalleistung nur auf wenigen oder gar einer einzelnen Plattform, da auch mit OpenCL die Spezifika jeder Plattform noch bei der Programmierung berücksichtigt werden müssen wie etwa die Anzahl und Aufteilung der einzelnen Recheneinheiten oder der gemeinsame Scratchpadspeicher für unterschiedlich viele Einheiten. Ein weiterhin ungelöstes Problem bleibt auch mit OpenCL die datengetriebene Ausführung von Kernels, denn zum Streaming von Eingabedaten an einen Kernel müssen diese Daten bereits vollständig berechnet sein [166], da keine noch feingranulareren Synchronisationsmechanismen angeboten werden. Nachteilig am Ansatz von OpenCL sind ferner die Verwendung von gemeinsamem Speicher sowie die Übersetzung zur Laufzeit für die Zielhardware, die ja je nach Zeitpunkt der Ausführung eine andere sein kann, weil gerade eine Ressource belegt ist. Da der Syntheseprozess für rekonfigurierbare Hardware sehr zeitaufwendig ist, ist OpenCL nicht sonderlich gut geeignet zur Programmierung von und Ausführung in rekonfigurierbarer Hardware. Die Portabilität von Hardwareimplementierungen kann wesentlich verbessert werden, wenn die Programmierung und Ausführung jeweils unabängig von der spezifisch eingesetzten Hardware sind, indem beispielsweise ein Rahmenwerk für eine Infrastruktur sorgt und indem die Ausführung unabhängig von der spezifischen Implementierung von für den gleichen Algorithmus entwickelten Programmen ist [NBK14].

OpenACC (Open Accelerator)[10] ist ein weiterer, sehr aktueller Standard. Aus HMPP [36, 84] von CAPS Entreprise[11] wurde dieser Standard [208], bei dem neben CAPS auch Nvidia, Cray und die Portland Group (PGI)[12] mitwirken, entwickelt. Der Standard wird maßgeblich von CAPS vorangetrieben und liefert eine Reihe von Compiler-Direktiven, mittels derer Schleifen und Code-Abschnitte auf verfügbare Akzeleratoren ausgelagert werden können oder der Datentransfer angegeben werden kann. Dabei werden vergleichbar zur Behandlung von mit OpenMP-Pragmas annotiertem Code die pragma-annotierten Funktionsimplementierungen übersetzt, um auf Akzeleratoren ausgeführt zu werden, nachdem die vorliegenden Implementierungen hinsichtlich der Eignung analysiert wurden. Das Konfigurieren der für den Akzelerator benötigten Laufzeitumgebung wie CUDA erfolgt dabei automatisch. Dieser Ansatz soll sich für Anwendungsentwickler eignen, welche die Hardware und ihre Anbindung nicht gesondert berücksichtigen sollen. Ferner sind die Programme dadurch in einem gewissen Rahmen portabel geworden. Die Besonderheit daran ist also, dass der Programmierer sich nicht mehr um Initialisierung, Programmierung und Verwendung des Beschleunigers sowie den Datentransfer zu kümmern hat. Fokussiert auf die Erweiterung von Multicore-Plattformen um Grafikprozessoren wurde mit HMPP [84] mittels Direktiven zur Angabe von Datentransfers untersucht, inwieweit unterschiedliche Kommunikationsstrategien beim Verwenden zusätzlicher Hardware von Vorteil sein können. Es kann dabei also dennoch sehr hilfreich sein, die Datenplatzierung explizit anzugeben. Da immer eine spezielle Schaltung anhand der Pragmas synthetisiert werden muss, ist keine Wiederverwendbarkeit der Schaltung oder von Teilen für andere Anwendungsteile gegeben. Indem Komponenten eingesetzt und zur Laufzeit parametriert sowie verbunden würden, könnten die Komponenten für unterschiedliche Anwendungsteile wiederverwendet werden ohne erneute Synthese und ohne Laufzeitrekonfiguration, die neben technischen Aspekten auch weitere Verwaltungsaspekte miteinbringen würde. Diesen Ansatz verfolgt daher die vorliegende Schrift.

[9] URL: `http://www.khronos.org/opencl/`
[10] URL: `http://www.openacc-standard.org/`
[11] URL: `http://www.caps-entreprise.com/`
[12] URL: `http://www.pgroup.com/`

3.3.2 Kommerzielle und industrielle Programmiermodelle für heterogene Systeme

AMD hat *Heterogeneous Parallel Primitives* (HPP) [108] entwickelt. Gaster und Howes erkennen [108], dass die ausschließlich datenparallele Ausführung von Kernels auf GPUs nicht so gewinnbringend ist, wie wenn zusätzlich Taskparallelität ausgenutzt wird. Daher haben sie mehrere Konstrukte entwickelt, so dass z.B. für Iterationen der äußeren Schleife bei verschachtelten Schleifen Tasks gebildet werden, die innere Schleife aber datenparallel läuft. Tasks hängen von der Verfügbarkeit von Eingabedaten ab. Der Einsatz von sogenannten Channels[13] dient dazu, auch feingranular die Daten auszugeben und an andere Tasks weiterzureichen. Die „verflochtene“[14] Ausführung könnte zu Verklemmungen[15] führen, was die Autoren durch die Verwendung von `sleep` anstelle von `wait` lösen.

IBM *Liquid Metal* [150, 147] basiert auf Java und fügt Erweiterungen hinzu, um zusätzlich auch auf Bitebene hardwareorientiert Programme zu entwickeln. Es zielt darauf ab, basierend auf dem Kontrolldatenflussgraphen einer Anwendung möglichst zusammenhängende Teile der Anwendung auf den FPGA zu bringen, um die Kommunikation gering zu halten. Der Anwendungscode kann sowohl mit einer modifizierten Java Virtual Machine (JVM) ausgeführt werden als auch im Ganzen oder in Teilen zu einer FPGA-Schaltung konvertiert und synthetisiert werden. Die erzielte FPGA-Implementierung ist jedoch 14× langsamer als ein Intel Core 2 Duo.

Intel untersucht im Projekt *EXOCHI* die Steuerung von zusätzlichen Grafikeinheiten über einen Ansatz bestehend aus Laufzeitsystem, Compiler und Spracherweiterung auf Basis von Direktiven, wie Beschleuniger von Anwendungen aus verwaltet werden können [276]. Dabei wird der Code für die zu verwendenden Funktionseinheiten in ein sogenanntes „Fat Binary“, also eine „große“ ausführbare Datei, zusammengeführt. Dies hält zwar den Code einigermaßen portabel, eleganter sind jedoch Ansätze, dies über die dynamischen Linker der Betriebssysteme erledigen zu lassen und entsprechend nur Funktionsaufrufe zu enthalten [KNBK12, KNBK10, BKNK09]. Ferner ist das Projekt auf die Nutzung von Grafikeinheiten beschränkt.

Bei der Programmierung eines Larrabee-erweiterten X86-Systems setzt Intel auf die programmierergesteuerte Nutzung von virtuellem gemeinsamem Speicher [229]. Der Compiler erzeugt zwei unterschiedliche ausführbare Dateien mit den gleichen virtuellen Adressen für die gemeinsamen Objekte, so dass der x86-Prozessor und der Larrabee kein Fat Binary mehr bekommen. Der Anwendungsentwickler muss selbst für Konsistenz der im virtuellen Speicher verwalteten Daten sorgen. Das Konzept der unterschiedlichen Ausführungsdateien ist interessant und kapselt gewissermaßen jeden Ausführungseinheitstyp. Die Evaluation zeigt jedoch, dass bei sechs Larrabee-Kernen mitunter noch kein Speedup erzielt wird und bei 24 Kernen maximal 7-facher Speedup, also lediglich eine Effizienz $E(24) = S(24)/24 = 7/24 \approx 0,292$. Indem der Anwender explizit die Arbeitsdaten in den Akzelerator lädt [NBK+13, NBK14], können die Aspekte der Konsistenz weitestgehend entfallen, ohne dass sich der Anwender des Datenmigrierens sonderlich bewusst sein muss.

Mit Dryad[16] [153] bringt Microsoft Task- und Pipeline-Parallelismus auf bequeme Art in die Programmierung von Clustern. Bei Dryad ist die Integration von Akzeleratoren und damit die Ausnutzung von heterogenen Systemen nur derart sinnvoll, dass wichtige Funktionen innerhalb der Anwendungen auf Akzeleratoren transparent für den Anwender ausgeführt werden. Einzig der Entwickler dieser Anwendungen, der idealerweise über ausreichendes Hardwareverständnis verfügt, ist damit in die Akzeleratorprogrammierung involviert. Dies wiederum ist genau der Ansatz der komponentenbasierten Programmierung, wo der Anwendungsexperte für die Implementierung der einzelnen von ihm verwendeten Komponenten und damit für deren Ressourcenbedarf sowie Leistung nicht mehr verantwortlich zeichnet.

Die genannten Lösungsansätze zeichnen sich dadurch aus, dass sie Parallelismus auf Task- und auf Datenebene gleichzeitig nutzen. Sie integrieren die Ausführung auf einem Akzelerator nahtlos in die zu unterstützende Anwendung. Zur Überlappung der ausgelagerten Berechnungen mit der Kommunikation setzen sie Streaming ein. Der Anwendungsentwickler hat entweder keine Möglichkeiten, auf die Verlagerung einzuwirken, oder er muss mittels Pragmas die auszulagernden Teile spezifizieren.

[13] engl., Kanäle
[14] engl. braided
[15] engl. Deadlocks
[16] URL: `http://research.microsoft.com/en-us/projects/dryad/`

Ersteres leidet darunter, dass wie bereits geschildert die Werkzeuge eine sehr schwierige Aufgabe bei der Codekonvertierung zu bewältigen haben. Letzteres ist bei der Entwurfsraum hinderlich, wenn die Verlagerung auf feinerer oder groberer Ebene untersucht werden soll; insbesondere aufgrund der nötigen neuen Synthesen oder Konvertierungen. Obendrein ist die erwartbare Leistung eher als gering einzustufen.

3.3.3 Akademische Ansätze

Bereits 2004 beschäftigten sich viele Arbeiten damit, die Programmiermodelle für CPUs und FPGAs zu verknüpfen [12]. Für eng gekoppelte Systeme, wie etwa einen FPGA mit integriertem Prozessor oder einen im System über gemeinsamen Speicher an die CPU gekoppelten FPGA, wird häufig ein thread-basierter Ansatz vorgeschlagen. Ein oder mehrere Threads laufen dann auf dem FPGA als Hardware-Threads ab unter Nutzung des gemeinsamen Speichers zusammen mit der Anwendung. Der Vorschlag der Autoren [12] ist das Hardware Thread Interface (HTI), das sich durch eine Unterscheidung von Hardware- und Software-Threads auszeichnet sowie durch einen Thread-Scheduler in Hardware, der obendrein mit einem IP-Core zur Synchronisation mittels Semaphoren in Hardware verbunden ist. Ein Wechsel zwischen den Ausführungseinheiten ist nicht möglich, obwohl bereits geringfügig die Thread-Zustände erfasst werden. Ferner ist die Synchronisation nun auf die Ausführung von Hardware-Threads optimiert, da jeder Software-Thread mit dem in Hardware implementierten IP-Core kommunizieren soll und dadurch Mehraufwand hat. Es wird also implizit davon ausgegangen, dass die Hardwareimplementierungen leistungsfähiger seien als die Ausführung in Software. Wie bereits angesprochen, darf nicht implizit davon ausgegangen werden, die explizite Nutzung der Hardwareimplementierungen sei leistungsfähiger. Stattdessen muss es dem Anwendungsentwickler möglich sein, die Zielimplementierung auszuwählen [NKBK10], und das heterogene System muss zur Laufzeit selbst die beste Implementierung auswählen können [KNBK12].

Im Gegensatz dazu ist der Wechsel von ausgeführten Threads auf andere Ausführungseinheiten bei der Self-distributing Virtual Machine (SDVM) [129, 130] und der rekonfigurierbaren Variante $SDVM^R$ [145] explizit angedacht. Hieraus ergibt sich als großer Forschungsbereich die programmierertransparente Programmentwicklung auf einem Thread-Modell, welches, gesteuert durch Laufzeitsysteme in Hardware und Software, die Anwendung auf beliebigen Ressourcen auszuführen vermag, ohne dabei jedoch noch weitere Programmiermodelle und Systeme als die bereits verwendeten Posix-Threads, OpenMP und MPI zu verlangen. Dies ist durch komponentenbasierte Programmierung und Ausführung möglich, wo die Funktionen vorab der Ausführung anhand von Anforderungen [NBK09] genauer spezifiziert werden und mit den verfügbaren Funktionsimplementierungen bezüglich deren Eigenschaften [NKBK10] abgeglichen werden [KNBK12].

Auch für Multi-FPGA-Systeme wurde an Programmiermodellen geforscht, wie möglichst viele heterogene Prozessoren und Recheneinheiten miteinander verbunden und so programmiert werden können [230], dass mit wenig Entwicklungsaufwand viel Beschleunigung erzielbar ist. Die Autoren untersuchen dazu MPI und implementieren Hardwareschnittstellen mit FIFOs, um das MPI-Protokoll außerhalb der Verarbeitungseinheiten handzuhaben. Dies ermöglicht, für bestimmte Funktionalitäten die Prozessoren durch effiziente Hardwareimplementierungen wie etwa die eines Stencils auszutauschen. Für den Programmierer ist transparent, ob die Ausführung durch einen Prozessor oder eine spezielle Hardwareimplementierung erfolgt. Die Hardwareunterstützung von MPI ist auf möglichst hohe Datentransferraten ausgelegt. Insgesamt wurde dadurch eine leistungsfähige Abstraktion geschaffen, die Anwendungsentwicklern die Nutzung von FPGAs und insbesondere heterogenen Recheneinheiten ermöglicht.

Bei TMD bzw. TMD-MPI der Universität von Toronto [231] werden bereits die FPGAs als selbständige Recheneinheiten betrachtet, die nicht mehr von einem Software-Thread, der selbst einen Slave gegenüber der Anwendung als Master darstellen würde, gesteuert werden, sondern direkt aufgerufen werden und deren Ausführung dann einen Hardware-Thread darstellt. Dies steht beispielsweise im Gegensatz zu H-MOL [273], das genau diesen Doppel-Slave-Ansatz umgesetzt hat, wo FPGAs oder deren Ressourcen noch als Koprozessoren bzw. Koprozessoreinheiten betrachtet werden. Dies wird aber wiederum genau dann attraktiv, wenn der FPGA als Koprozessor grobkörnigere Funktionalität bietet, und nur intern mehrere Einheiten gezielt zur Berechnung der Funktionalität verwendet, wie etwa in einem mikroprogrammbasierten Rahmenwerk mit mehreren unterschiedlichen, domänenspezifischen Einheiten [NBK+13, NBK14].

OpenMP wurde untersucht hinsichtlich der Angabe möglicher Ausführungseinheiten [16]. Dabei lag der Fokus zunächst auf CUDA und der RASC Library[17] als Programmiermodellen sowie dem IBM Cell B.E.-Prozessor [119] als Hardware, die über die OpenMP-Pragmas angesprochen werden sollten. In späteren Arbeiten [49] wurden auch gezielt FPGAs als mögliche Akzeleratoren untersucht. Die Bitstreams zur Konfiguration des/der FPGAs sollen dabei vorgefertigt in einem Cache vorliegen und bedarfsorientiert konfiguriert werden. Aufgrund der geringen Bandbreite zur Datenübertragung ist der Aufwand zur Kommunikation so hoch, dass die Ausführung blockweise erfolgen muss und erst ab einer Größe von 128×128 gewinnbringend ist. Demzufolge sollten die zum Koprozessor übertragenen Daten nach Möglichkeit häufig wiederverwendet werden. Indem grobkörnigere Algorithmen auf Koprozessoren ausgeführt werden, besteht die Möglichkeit, bei komponentenbasiertem Design und komponentenbasierter Ausführung die Daten zwischen den Komponenten auszutauschen und somit wiederzuverwenden [NBK+13]. Insgesamt lässt sich somit das wichtige Verhältnis von Berechnung zu Kommunikation steigern.

Das „vielseitige“ (polyedrische oder polytope) Modell dient zur Bestimmung von guten Ablaufplänen[18] bei verschachtelten Schleifen, wobei häufig die Optimierung der Speicherzugriffe [219] im Vordergrund steht. Zur Bestimmung der besten Ablaufpläne wird *Integer Linear Programming* (ILP) eingesetzt. Außer bei der Optimierung von verschachtelten Schleifen kann das polyedrische Modell auch eingesetzt werden, wenn Verzweigungen von den Eingabedaten abhängen [28]. Es wird dazu mit Pragmas annotiert, und mittels eines Compilers lässt sich dann Autoparallelisierung erreichen. Weitere akademische direktivenbasierte Ansätze finden sich mit Sequoia++ [97] und CellSs [27] als Umsetzung des generischeren StarSs [218].

Wie auch bei den kommerziellen Programmmiermodellen ist eine Alternative zu Direktiven die Programmierung mit Streams. Wissenschaftliche Ansätze diesbezüglich finden sich beispielsweise mit „Extending a Stream Programming Paradigm to Hardware Accelerator Platforms“ [51] für beliebige Akzeleratoren oder auch konkret für FPGAs [147].

StreamIt [255, 265] ist eine auf Java basierende, stream-orientierte Sprache, mit der Datenflussarchitekturen auf gitterbasierte Mehr-/Vielkernprozessoren abgebildet werden. StreamIt kann dazu verwendet werden, den Raw Microprocessor zu programmieren [252]. Eine Laufzeitumgebung für StreamIt auf dem IBM Cell-Processor berücksichtigt die verteilten Speicher auf der PPE und den SPEs, kann aber nur mit festen Ein- und Ausgabedatenraten arbeiten, wodurch der Nutzen bislang sehr begrenzt und der Mehraufwand sehr hoch ist [290]. Dieser Mehraufwand kann verringert werden, wenn vorab der Ausführung mittels eines analytischen Modells eine Belegung[19] der vorhandenen Kerne und ein Scheduling errechnet werden [95].

Für den IBM Cell-Prozessor gelang es Chellappa et al. mittels des Programmerzeugungssystems „Spiral“ und zusätzlicher Erweiterungen für Parallelisierung und Speicher-Streaming, Code zu erzeugen [55], der mehr Nutzen aus der verfügbaren Hardware zu ziehen vermag als vorgefertigte Implementierungen. Donaldson et al. haben untersucht, inwieweit die Autoparallelisierung für den Cell-Prozessor unterstützt werden kann, wenn die verwendete Programmiersprache um ein Konstrukt zur Einhaltung der Datenlokalität erweitert wird [85]. In diesen als Sieb[20] gekennzeichneten Regionen werden Schreibzugriffe der verwendeten Cell SPEs auf den Hauptspeicher verzögert und die Daten damit nicht kohärent mit anderen SPEs gehalten. Der Begriff des Siebens rührt daher, dass die Daten aufgeteilt werden und von verschiedenen SPEs bearbeitet werden. Dadurch wird die Speicherschnittstelle nicht überlastet, sondern kann in Streaming-Manier die bereits berechneten Daten nebenläufig weiteren Berechnungseinheitenen zuführen, und z.B. bei Stencil-Verfahren sind die vorab verwendeten und berechneten Daten für die nächsten Berechnungen noch im SPE-eigenen Speicher verfügbar. Abschließend werden alle Schreibzugriffe wieder gültig gemacht und der Speicher zwischen PPE und SPEs wird kohärent. Der Cell PPE darf in der Zeit der Sieb-Ausführung nur von den SPEs unabhängige Berechnungen machen, da der Speicher während der Ausführung ja nicht kohärent gehalten ist.

Existierende heterogene Rechensysteme wie die Convey HC-1 lösen das Problem der Kohärenz, indem der Koprozessor seinen Speicher wie einen weiteren Cache in ein SMP-System einbettet. Somit sind dann Berechnungen sowohl seitens des Koprozessors als auch seitens des Hostprozessors

[17] URL: `http://www.mitrion.se/?document=RASC_white_paper.pdf`
[18] engl. Schedule
[19] engl. Mapping
[20] engl. sieve

auf den gleichen Daten zum gleichen Zeitpunkt durchführbar. Dadurch können von der auf dem Koprozessor ausgeführten Funktion abhängige Funktionen (und umgekehrt) überlappt ausgeführt werden und so weiterer Nutzen aus dem heterogenen System gezogen werden [Now14, BNK14].

Besonders wurde in den genannten akademischen Programmiermodellen und -sprachen die Lösung des Kommunikationsproblems in heterogenen Systemen angegangen. Neben Streaming wurde versucht, Datenabhängigkeiten gezielt zu berücksichtigen und dadurch die Systemleistung zu verbessern. Die Programmiererfreundlichkeit wurde wie auch bei den kommerziellen und industriellen Programmiermodellen über den Einsatz von Pragmas sowie darüber hinaus des bekannten Programmiermodells MPI adressiert. Die Synthesezeiten, die allgemeine Vorgehensweise bei der Verlagerung eines Algorithmus in Akzeleratoren, die hohe Einarbeitungszeit, die hardwareorientierte Programmierung wurden dabei jedoch nicht ausreichend verbessert. Der Anwendungsentwickler hat damit noch kein nennenswertes Mittel an die Hand bekommen, welches ihn auf bequeme, schnelle Weise leistungsfähige Akzeleratoren ohne hardwareorientierte Denkweise nutzen lassen würde.

3.4 Laufzeitsysteme für heterogene Systeme

Eine wichtige Fragestellung bei der Verwendung heterogener Systeme ist, wie die verfügbaren Einheiten insgesamt bestmöglich eingesetzt werden können. Laufzeitsysteme betreiben Scheduling von Threads zum Lastausgleich[21] und bieten Schnittstellen für die Ausführung von Threads in Software oder Schaltungen in Hardware an. In Abhängigkeit der Bandbreite, der Speicherzugriffslatenzen sowie der Art der gemeinsamen Nutzung von Speicher ergeben sich unterschiedliche Notwendigkeiten und Möglichkeiten, geeignete Laufzeitsysteme für diese heterogenen Systeme zu entwickeln.

Liquid Metal [150] ist nicht nur ein Programmiermodell, sondern bringt auch ein Laufzeitsystem in Software und Hardware mit, das im Wesentlichen aus Puffern und Registern zum Datenaustausch besteht. Die zu verlagernden Algorithmen sind für einen Domänenexperten sehr hardwarenah und bitlastig zu formulieren, so dass kaum Vereinfachung durch die Verwendung von Liquid Metal entsteht. Zudem müssen zunächst die Hardwarebeschreibungen generiert und nach weiteren Zwischenschritten daraus die Hardwareschaltungen synthetisiert werden vorab der Verwendung; sie sind somit nicht sofort vom Anwender verwendbar und die Werkzeugkette laut Angabe der Autoren noch sehr komplex. Wenn vorhanden, so werden die von Limeade erzeugten FPGA-Bitstreams vom Laufzeitsystem in die rekonfigurierbare Logik geladen und dort mittels der Puffer nach dem Datenflussmodell ausgeführt. Der „Aufruf“ der FPGA-Bitstreams erfolgt über den Austausch von Code-Objekten, an welchen Daten hängen, mittels des Laufzeitsystems. Das heißt wiederum, dass aufgrund dieser Anbindung in jedem Takt neue Daten nur an wenige Subgraphen übergeben werden und so kaum mehrere Subgraphen völlig autonom voneinander ausgeführt werden können; ein Anwender kann nicht gezielt mehrere nicht in Zusammenhang stehende Funktionalitäten auslagern und daher auch weder die Hardware noch die Schnittstellen maximal ausnutzen. Möchte ein Anwendungsentwickler mehrere, etwa durch Funktionen, Methoden oder Klassen implementierte Subgraphen mehrfach in unterschiedlichen Datenflussgraphen zeitgleich oder zu einem späteren Zeitpunkt innerhalb eines anderen Graphens nutzen, muss wieder ein neues Programm formuliert, eine Zwischenbeschreibung erstellt und daraus eine neue Schaltung synthetisiert werden, welche dann wieder alle Subgraphen enthält und wieder durch die Datenschnittstelle des Laufzeitsytems limitiert ist. Der Ansatz von Liquid Metal ist also auch noch nicht ausreichend wiederverwendbar für Anwendungsentwickler. Entsprechend können Domänenexperten als Anwendungsentwickler noch keinen für sie ausreichenden Nutzen durch Liquid Metal erhalten.

RISPP [22, 23, 24] ist eine Architektur für einen zur Laufzeit rekonfigurierbaren Prozessor im Umfeld eingebetteter Systeme. Der Befehlssatz enthält dazu benutzerspezifizierbare, anwendungsanpassbare Befehle, die aus Atomen zu Molekülen zusammengesetzt sind. Dies hat den Vorteil, dass die Atome austauschbar sind und die zur Verfügung gestellte Funktionalität unabhängig von den Positionen der Atome auf dem FPGA ist, wodurch die Rekonfiguration eines Moleküls lediglich aus dem Konfigurieren der noch nicht vorhandenen Atome besteht. Solange die aus dem Anwendungsumfeld extrahierten Spezialbefehle für einen Anwendungsentwickler ausreichend sind,

[21] engl.: Load Balancing

kann RISPP sehr hohe Leistung liefern. Genügen die Spezialbefehle nicht, so hat ein Domänenexperte neue Spezialbefehle zu definieren, ggf. in die Werkzeugkette einzubringen, Moleküle zu formulieren und voraussichtlich sogar das gesamte Design neu zu synthetisieren. Die sehr effizienten Atome sind, um für mehrere unterschiedliche Spezialbefehle und für deren Molekülimplementierungen nutzbar zu sein, aus einer sehr eng eingegrenzten Domäne. Sie werden sehr eng miteinander verbunden, so dass ein spezifischer Datenpfad für den zu beschleunigenden Kernel bzw. Befehl entsteht und daher das Problem der Kommunikation mit dem umgebenden System nicht gelöst werden muss. Da bei RISPP und Spezialbefehlen je Aufruf nur geringe Datenmengen jeweils im Rahmen der Prozessorpipeline behandelt werden, wurde wenig Fokus auf Datenvorhaltung und -wiederverwendung gelegt. Dies ist im Umfeld wissenschaftlicher Anwendungen aus dem numerischen oder bioinformatischen Umfeld aber unerlässlich, um Leistungssteigerung aus der Verlagerung grobgranularer, datenintensiver Algorithmen in Beschleunigerhardware zu erhalten. Denn dort kann die Ausführung der Prozessorpipeline nicht für einige Millionen bis Milliarden Takte angehalten werden, sondern die Akzeleratornutzung muss gezielt in die Ausführung weiterer Recheneinheiten wie Prozessorkerne oder anderer Beschleunigereinheiten integrierbar sein, um Wartezeiten auf Ergebnisse größtmöglich zu vermeiden. Es muss unbedingt dafür explizit Sorge getragen werden, dass sehr große Datenmengen von der Beschleunigerhardware je Aufruf verarbeitet werden können, ohne dabei die Ausführung des Prozessors zu beeinträchtigen oder gar explizit den Prozessor zu benutzen. Entsprechend sollte auch Kontrollfluss innerhalb der Moleküle unterstützt werden, um komplexere Funktionalität als aneinandergereihte Filteroperationen auszulagern. Als Erweiterung von RISPP wurde betrachtet, dass die Rekonfigurationsdauer von großen rekonfigurierbaren Flächen ein Problem darstellt und daher die Nutzung von weniger oder einfacheren Atomen nutzbringender sein kann, insbesondere wenn diese dann später im Zusammenspiel mit anderen Molekülen genutzt werden können [4]. Bei heterogenen Mehrkernsystemen mit rekonfigurierbaren externen Akzeleratoren hingegen sollte ausreichend Fläche für alle benötigten Funktionalitäten in der rekonfigurierbaren Hardware vorhanden sein. Zudem sollte die Verwendung der Funktionalitäten derart lang andauern, dass die Rekonfigurationsdauer nicht ins Gewicht fällt; und damit kann die Rekonfiguration schlichtweg entfallen, indem nach der Verwendung des Beschleunigers dieser vollständig neu rekonfiguriert wird. Das Rekonfigurieren kann auch auf das Einsparen von Energie hin zugeschnitten werden [237]. Dabei werden abhängig von den zur Laufzeit dynamisch veränderlichen Anforderungen einige Atom-Container mittels Power-Gating deaktiviert. So kann die durch Leckströme und dynamisch verursachte Leistungsaufnahme verringert werden, und die Rekonfiguration kann ebenso entfallen. Wenn Domänenexperten rekonfigurierbare Hardware zur beschleunigten Ausführung ihrer Anwendungen auf Mehrkernsystemen einsetzen wollen, spielt die Leistungsaufnahme bzw. der Energiebedarf keine wesentliche Rolle, zumal sich dynamisch keine Vorgaben ändern, da in der Regel nur ein Anwender exklusiven Zugriff auf ale Ressourcen hat, diese für nur eine Anwendung nutzt und das System weder ein eingebettetes System ist noch batteriebetrieben eingesetzt wird. Insofern benötigen Anwendungsentwickler bzw. Domänenexperten solche Rekonfigurationsstrategien und den damit verbundenen Aufwand gar nicht.

Im MORPHEUS Runtime System [256] wird mit ISRC (Intelligent Services for Reconfigurable Computing) [257] ein Laufzeitsystem bereitgestellt, das in der Lage ist, MOLEN-artige [271] Beschleuniger zu allozieren und das für die Verwendung benötigte dynamische Scheduling zu betreiben. MORPHEUS [258] selbst ist zunächst ein SoC bestehend aus Prozessor und drei unterschiedlichen rekonfigurierbaren Bereichen, die für unterschiedliche Granularität geeignet sind; zur Verbindung dient ein Network on Chip (NoC). Um eine Anwendung zu unterstützen, kann nach dem MOLEN-Paradigma jeder der drei Bereiche angesprochen werden, und nach Konfiguration mit einem zuvor statisch erzeugten Bitstream kann dann ein Kontroll-Datenflussgraph in einem solchen Bereich ausgeführt werden. Die über die Bitstreams konfigurierten Schaltungen in diesen Bereichen können miteinander über Datenaustauschpuffer[22] kommunizieren und zusammen so einen größeren Datenflussgraphen abbilden und ausführen. Die Bitstreams werden anhand einer durchgängigen Werkzeugkette aus dem C-Code des Anwenders erzeugt, wobei zur Hardware-Implementierung über Subfunktionen zwar eine grafische Schnittstelle angeboten wird, aber bereits Hardwareexpertenwissen nützlich sein kann. Ein einmalig erzeugter Bitstream kann zwar für eine andere Anwendung unverändert wiederverwendet werden, nicht jedoch Teile davon, welche mehrere Subfunktionen umfassen. Insgesamt ist somit für den Anwender häufig neue Synthese nötig, was hohen zeitlichen Aufwand und mitunter sogar zusätzlich mehrfache Entwurfsraumexploration bedeutet, und er ist auf die Verwendung von nur drei Bitstreams/Funktionalitäten zeitgleich beschränkt. Wird

[22] engl. Data Exchange Buffer, DEB

eine Funktionalität nicht mehr benötigt, so kann in den Bereich dynamisch zur Laufzeit ein anderer Bitstream konfiguriert werden, wodurch zumindest zeit-gemultiplext mehr Funktionalitäten bereitgestellt werden können. Engpässe bestehen in der Datenübertragung zwischen den Einheiten und zu weiteren Speichern; datengetriebene Ausführung bzw. Überlappung von Datenübertragung und Berechnung fehlen, weshalb wie angegeben die Ausführung nur halb so schnell wie eigentlich erwartbar ist. Die ISRC regeln die Verwendung der Hardwareressourcen für mehrere unabhängige, zeitgleich ausgeführte Anwendungen; sie suchen die geeigneteste Implementierung aus, programmieren die DMA-Einheiten und verschalten die Puffer korrekt. Die ISRC erreichen jedoch nicht die nahtlose Kommunikation der Funktionalitäten mit dem externen Speicher und anderen Funktionalitäten. Es steht aus, die Überlappung von Kommunikation und Berechnung durch datengetriebene Ausführung der Funktionalitäten auf feiner Granularität zu erreichen, um so die Schnittstellen maximal auszunutzen. Ferner fehlt es an ausreichend vielen Speichern, um Daten temporär ablegen und später wiederverwenden zu können; dadurch werden die Speicherschnittstellen unnötig stark zusätzlich belastet. Vor dem Hintergrund des Verlagerns datenintensiver Anwendungen in Beschleunigerhardware kann es zwar sinnvoll sein, mehrere unterschiedlich granular rekonfigurierbare Hardware zu haben. Allerdings benötigt ein Anwender in der Regel nicht mehrere Anwendungen gleichzeitig und bei Verfügbarkeit eines dedizierten Akzelerators auch kein dynamisches Ersetzen von Funktionalitäten, weshalb der Aufwand in diesem Umfeld nicht gerechtfertigt scheint. Ferner erwartet ein Anwender mehr Wiederverwendbarkeit der erzeugten Elemente wie Subfunktionsverschaltungen oder der Bitstreams, um nicht für jede neue Anwendung aus seinem Umfeld dem hohen zeitlichen Aufwand erneut begegnen zu müssen.

Ähnlich zu MORPHEUS' ISRC ist der *Hardware Component Manager (HCM)* [284], der zum Ziel hat, über die Abstraktion mit Komponenten eine wiederverwendbare, abstrahierte Schnittstelle für Anwendungen in heterogenen, FPGA-erweiterten Systemen zu bieten. Demzufolge könnte bei gleichbleibender Schnittstelle und gleichen Hardwareressourcen im Idealfall ein generierter Bitstream für eine Komponente in ein anderes System, auf einen anderen FPGA übertragen werden, und man könnte die gleiche Komponente nutzen. Der Aspekt der Portabilität ist so erfüllt, und gleichermaßen der der Wiederverwendbarkeit, da eine Komponente für Folgeaufrufe im gleichen oder in anderen Kontexten, allein oder in Kombination mit weiteren, wiederverwendbar ist. Dies erfüllt schon bereits länger das am Lehrstuhl entwickelte H-MOL [KVB+09]. Hat man dann allerdings hinreichend viele Komponenten zur Verfügung, entsteht das Problem, diese als Anwendungsentwickler auch geeignet ansteuern zu wollen. Außerdem sollen die Komponenten auch Daten austauschen können, um so einen möglichst großen Kontrolldatenflussgraphen auf der rekonfigurierbaren Hardware ausführen zu können. Mit HCM wird noch keine Kommunikationsstruktur für die Komponenten bereitgestellt, so können nicht mehrere Komponenten gemeinsam ein Problem bearbeiten und der FPGA wird so nicht maximal ausnutzbar. Auch fehlen Möglichkeiten, welche die Datenwiederverwendung explizit unterstützen würden. Wesentliche Probleme treten dann ferner auf, wenn nicht ausreichend viele rekonfigurierbare Bereiche für die Anzahl an benötigten Komponenten zur Verfügung stehen: Anhand welcher Kriterien entscheidet wer, welche Bereiche mit welcher Funktionalität konfiguriert werden sollen? Diese Problemstellung geht konkret RISPP mit seinem mRTS an [4, 23]; die Forschergruppe vom HCM selbst hat dazu für das konkrete Gebiet von Funkanwendungen den Flexible Radio Kernel (FRK) [148] als Hardwarelaufzeitsystem entwickelt. Demzufolge handelt es sich bei HCM und FRK auch eher um Ansätze für eingebettete Systeme und SoCs als um Möglichkeiten der Anwenderunterstützung bei der Verlagerung von Software in rekonfigurierbare Hardware.

RAMPSoCVM [128] virtualisiert die Hardware innerhalb eines Mehrprozessorsystems auf einem Chip[23], so dass von einer auf dem MPSoC laufenden Anwendung aus die rekonfigurierbaren Knoten und Prozessorkerne gleichermaßen zum gemeinsamen Lösen eines Problems miteingebunden werden können. Die zugrundeliegende Plattform „Runtime Adaptive Multi-Processor System-on-Chip“ (RAMPSoC) [125, 126] zeichnet sich dadurch aus, dass mehrere Bestandteile wie Prozessoren und rekonfigurierbare Fläche über ein NoC namens Star-Wheels [127] verbunden sind. Sie ist vor allem für eingebettete Anwendungen entwickelt, bei denen mehrere echtzeitkritische Anwendungen zeitgleich weitere Prozessor- oder Beschleunigerressourcen benötigen und zugleich Zeitschranken einzuhalten haben. RAMPSoCVM definiert in Anlehnung an den MPI-Standard für RAMPSoC eine Funktion zum Initialisieren von Clients bzw. zum Laden von Anwendungen oder Bitstreams, wobei in einer Datei der auszuführende Taskgraph hinterlegt sein muss. Desweiteren definiert es

[23] engl. Multi-Processor System-on-Chip (MPSoC)

je eine Sende- und eine Empfangsfunktion zur Kommunikation bzw. zum Datenaustausch, und eine Funktion, um die Ausführung als beendet und damit die Hardwareressource als unbelegt zu deklarieren. Die Umsetzung erfolgt über Gerätetreiber im verwendeten Betriebssystem. Anwendungen kommunizieren mit dem Treiber über diese Funktionen und können dadurch auf mehrere Client-Ressourcen zugreifen. Dazu muss die Anwendung als Fat-Binary ausgeliefert werden und alle Daten, Programmcodes, Bitstreams für die Clients enthalten, um diese auch laden zu können. So können auch gleichzeitig mehrere Anwendungen über die an MPI angelehnten Mechanismen über das NoC die benötigten Ressourcen verwenden. Für die Datenkommunikation zwischen dem sogenannten Server und den Clients wird Star-Wheels genutzt. Ein Anwendungsentwickler hat so ein sehr bequem nutzbares System, wenn die Programme bzw. Bitstreams alle vorliegen; er muss dann den Taskgraphen formulieren auf Basis der verfügbaren Funktionalitäten. Da er jedoch nur eine Anwendung zu einem Zeitpunkt beschleunigt ausführen will, ist bereits der durch die vier Funktionen eingebrachte Aufwand zu groß. Wie die Ergebnisse zeigen, ist das Übertragen von Daten daher auch etwa sieben mal langsamer als die direkte Verwendung von RAMPSoC und Star-Wheels ohne das Laufzeitsystem RAMPSoCVM.

BORPH [244] steht für „Berkeley Operating system for ReProgrammable Hardware“ und soll Anwendungsentwicklern ermöglichen, FPGAs zu nutzen, indem Funktionen mittels eines veränderten Linux-Systems als Hardwareprozesse auf dem FPGA laufen. Zur Erzeugung dieser Hardware-Prozesse muss eine erweiterte Werkzeugkette verwendet werden, welche die Anschlusslogik für die Nutzerlogik einbettet und rekonfigurierbare Bitstreams aus dem z.B. in Simulink angegebenen Anwenderprogramm erzeugt [245]. Dies ist zunächst sehr zeitaufwendig. Die Hardware- und Softwareprozesse kommunizieren dann über ein virtuelles Dateisystem zur Angabe von Parametern und zur Synchronisation. Der Datenaustausch kann über Pipes erfolgen, indem ein Hardwareprozess programmatisch über `popen` aus der Software heraus aufgerufen wird bzw. auf einer Shell Soft- und Hardwareprozesse mittels des Pipe-Zeichens („|“) verbunden werden. Maximal ein Hardwareprozess wird auf der rekonfigurierbaren Teilfläche eines User-FPGAs ausgeführt. Auf der Plattform BEE2 sowie auf der Convey HC-1 sind somit maximal vier Hardwareprozesse gleichzeitig nutzbar, in der Regel jedoch weit weniger. Die Hardwareprozesse auf mehreren FPGAs kommunizieren bei Verfügbarkeit über dedizierte Verbindungen, ohne den Software-Prozess zu involvieren. Aus Anwendersicht ist dies ein sehr bequemer und guter Ansatz, solange die vier Hardwareprozesse ausreichend sind. Werden weitere Komponenten benötigt, so dauert es etwa 900 ms, einen neuen Hardwareprozess zu laden, konfigurieren und starten. Es ist nötig, dass die Granularität der Hardwareprozesse sehr hoch gewählt ist, um möglichst viel Leistung aus der Verlagerung in rekonfigurierbare Hardware zu erhalten, da die Datenübertragung sehr langsam ist und dementsprechend die Daten möglichst oft auf dem FPGA wiederverwendet werden müssen. Demzufolge würde man eher kein einzelnes Design für einen Sequenzvergleich nach dem Smith-Waterman-Verfahren auslagern und ebensowenig eine Matrixmultiplikation, sondern eher ein Multiple-Sequence-Alignment, das intern mehrere Vergleiche parallel implementiert und durchführt, bzw. einen Gleichungslöser oder ein anderes komplexeres mathematisches Verfahren, welches weitere Operationen als lediglich eine Matrixmultiplikation durchführt. Somit sind die Hardwareprozesse bereits von einer derart hohen Granularität, dass sicherlich die direkte Einbindung in einem größeren Kontext gewinnbringend sein kann. Aber andererseits fehlt den Anwendern die Möglichkeit, mit wenig zeitlichem Aufwand anhand vorgefertigter Komponten selbst eine nutzbringende Verlagerung eines (Teil-)Algorithmus in die rekonfigurierbare Hardware zu erreichen. Taskparallelismus wird bei BORPH nicht gezielt angegangen, sondern lediglich Pipeline-Parallelismus, um sozusagen Ketten von Prozessen/Anwendungen teilweise hardwarebeschleunigt auszuführen. Die Hardwareprozesse laufen ganz normal wie softwareimplementierte Funktionen und müssen über den Prozessor des Kontroll-FPGAs auf den Speicher zugreifen, anstatt mit hardwarenahen, wirksameren Mitteln wie DMA-Transfers oder Streaming über weitere Hardware die Eingabedaten vom zentralen Kontroll-FPGA bereitgestellt zu bekommen. Bei BORPH besteht also insgesamt das Problem, die FPGAs gut auszunutzen, indem möglichst schnell neue Daten verarbeitet werden können und indem möglichst viele Berechnungen auf den erhaltenen Daten durchgeführt werden.

InvasIC [140] entwickelt viele der genannten Themen weiter hingehend zu Manycore-Architekturen, die vor besonderen Ressourcen- und Energieproblemen stehen. Basierend auf dem zuvor beschriebenen RISPP fokussiert es vornehmlich auf die Anwendung im Rahmen eingebetteter Systeme oder der Multimediadatenverarbeitung. Eine Anwendung kann bei InvasIC angeben, wieviele Ressourcen sie an einer bestimmten Stelle nutzen kann und welchen Nutzen dies verspricht. Wenn dann mehrere Anwendungen ausgeführt werden, wägt ein Laufzeitsystem, das zu Teilen in Hard-

ware und in Software implementiert ist, ab, welche Anwendung welche Anzahl Ressourcen nutzen darf. Die Anwendungen müssen dementsprechend auch damit zurecht kommen, dass ihnen Ressourcen entzogen werden. Die verfügbaren Ressourcen sind Kacheln, welche mehrere RISC-Kerne oder „Tightly Coupled Processor Arrays“ (TCPAs) oder Speicher enthalten, und über ein Network on Chip (NoC) miteinander verbunden. Die RISC-Kerne können auch Befehlssatzerweiterungen ausführen (*Infection*), welche von der Anwendung aus zuvor mittels einer sogenannten *Invasion* angefordert wurden. Die möglichen Implementierungen müssen dem Laufzeitsystem vorab bekannt sein. Solange die Implementierungen nicht auf die rekonfigurierbare Hardware ladbar sind, wird alternativ eine Software-Implementierung ausgeführt und später nahtlos auf die Verwendung der speziellen Hardware-Implementierungen umgestellt. Innerhalb einer Kachel können Threads auf den (erweiterten) RISC-Kernen über einen Kachel-lokalen Speicher kommunizieren; zur Kommunikation zwischen mehreren Rechen-Kacheln dienen das NoC und die expliziten Speicherkacheln; und zur Nutzung des externen Speichers werden Eingabe-/Ausgabe[24]-Kacheln genutzt. Die nahtlose Datenweitergabe von einem Verarbeitungselement an das nächste wird lediglich innerhalb eines TCPAs, welcher ebenso nach dem Invade/Infect-Prinzip programmiert und genutzt wird, erreicht. Also muss ein Anwendungsentwickler genau angeben, welche Teile auf welchen Ressourcen ausgeführt werden müssen, z.B. Streaming-orientierte, gut parallelisierbare Teile eben auf den TCPAs. Dementsprechend muss er sich auch damit befassen, welche Ressourcen überhaupt zur Verfügung stehen. Ein Anwendungsentwickler findet somit mit InvasIC keinen Ansatz vor, mit dem er datenintensive Anwendungen durch Nutzung rekonfigurierbarer Hardware beschleunigen kann, ohne sich um Details der Hardware und ihrer Programmierung zu sorgen, zumal der Datenaustausch zwischen mehreren Kacheln über das NoC (etwa 100 Takte pro Zugriff angegeben) und ggf. den gemeinsamen Speicher erfolgen muss, der aber nicht von beliebig vielen Kacheln gleichzeitig angesprochen werden kann.

Das grundlegende Problem, die Kommunikation in heterogenen Systemen aufwandsarm zu gestalten mit niedrigen Verzögerungszeiten und maximiertem Durchsatz, wird in den genannten Arbeiten nicht gelöst, wodurch ein Anwendungsentwickler aus dem Umfeld des wissenschaftlichen Rechnens keinen Nutzen aus der Verlagerung der datenintensiven Anwendungen erzielen kann. Ein möglicher Ansatz liegt in der datengetriebenen Ausführung [NBK+13] und wird daher im Verlauf dieser Arbeit betrachtet.

3.5 Diskussion, Abgrenzung und Aufbau der eigenen Arbeit

Neben der Zunahme der Zahl der Prozessorkerne bei Mikroprozessoren haben sich in den letzten Jahren heterogene Systeme weit verbreitet, die neben einem oder mehreren Ein- oder Mehrkernprozessoren über Beschleunigereinheiten verfügen, um so einerseits zusätzliche Beschleunigung sowie andererseits verbesserte Energieeffizienz zu bieten. Eine große Anzahl wissenschaftlicher Arbeiten hat bereits die Portierung einzelner Anwendungsteile auf unterschiedliche Beschleuniger wie GPUs, FPGAs oder ClearSpeed erreicht. Nur wenige Arbeiten koppeln mehrere Beschleuniger zum Erreichen größerer Gesamtleistung, z.B. BORPH. Entsprechend arbeiten zumeist selbst bei Multi-FPGA-Systemen (MFS) die auf einer Karte oder in einem System verfügbaren FPGAs lediglich zusammen als eine einzelne große rekonfigurierbare Recheneinheit (übliches Modell bei der Convey HC-1). Die gleichzeitige Ausführung vieler unterschiedlicher Operationen nebeneinander wurde jedoch selten angegangen (bei RISPP innerhalb von Molekülen; bei HCM ohne Kommunikation; bei RAMPSoCVM Auf Knotenebene erreicht). Ein Ziel dieser Arbeit ist es daher, viele unterschiedliche Operationen gleichzeitig auszuführen.

Ein großes Problem beim Verwenden von Beschleunigern stellt der Datentransfer bzw. dessen Optimierung dar. Sollen mehrere verschiedene Recheneinheiten auf einem Beschleuniger verwendet werden, so werden die Daten zumeist hin- und herübertragen, und dieses Problem multipliziert sich daher. Einige Programmiermodelle gehen diese Problematik bereits an, indem Ein- und Ausgaberaten spezifiziert werden oder kommunizierende Prozesse eingesetzt werden. Bei den Laufzeitsystemen geht lediglich MORPHEUS' ISRC dieses Problem an, ohne dabei jedoch eine ausreichende Lösung zu stellen für die benötigte nahtlose Kommunikation.

[24] engl. Input/Output, I/O

Die vorherrschende Aufgabe zur Erhöhung der Nutzbarkeit von FPGA-erweiterten heterogenen Mehr- oder Vielkernsystemen ist, die Anwendungsexperten sowie die Entwickler von Beschleunigereinheiten („Hardwareexperten“) beim aufwandsarmen, zielführenden Entwickeln möglichst portabler Anwendungen und nicht zuletzt deren beider erfolgreiche Zusammenarbeit mittels geeigneter Methoden und Werkzeuge zu unterstützen. LiquidMetal, MORPHEUS mit seinem zugehörigen Runtime System, der HCM, BORPH und in gewissem Rahmen auch RAMPSoCVM bieten Lösungen an oder integrieren bestehende, um dies zu erreichen. Hat ein Hardwareexperte eine Speziallösung gefunden, mit der ein bestimmter Algorithmus auf ein bestimmtes System gebracht werden kann, so ist diese Speziallösung in der Regel nicht auf anderen Systemen einsetzbar, also nichtportabel. Lediglich beim HCM wird Portabilität gezielt angegangen und erreicht. Die synthetisierte Sepziallösung ist häufig auch nicht für andere Anwendungen wiederverwendbar, am ehesten die eher feingranularen Komponenten beim HCM und desweiteren noch die Atome bei RISPP für weitere Moleküle oder ein Molekül für weitere Anwendungsteile. Der angefallene Aufwand sollte sich aber rentieren durch höhere Wiederverwendbarkeit; die Spezialimplementierung muss also portabel sein. Hinsichtlich der Portierbarkeit muss sowohl im Chip integrierte rekonfigurierbare Logik nutzbar sein als auch ein auf rekonfigurierbarer Logik basierender Koprozessor; dies bietet RAMPSoCVM. Der Anwendungsentwickler wiederum möchte eine einfache und bequeme Herangehensweise, die auf bekannten Prinzipien und Sprachen aufbaut. Der Ansatz über Hochsprachen zur Hardwarebeschreibung ist generell vielversprechend und entsprechend zu überprüfen; er wird u.a. von Liquid Metal mit Java, von MORPHEUS und RAMPSoC mit C und von BORPH mittels Simulink genutzt.

Desweiteren gilt es herauszufinden, welche Arten von Algorithmen oder Anwendungen zur Ausführung in rekonfigurierbarer Hardware geeignet sind. Die im vorigen Abschnitt aufgeführten Plattformen und Laufzeitsysteme gehen allesamt feingranulare Algorithmen an (Feistel-Funktion bei Liquid Metal, Punkt-Filterung bei RISPP, Merkmalsextraktion bei RAMPSoCVM, schnelle Fourier-Transformation bzw. Entfernen von Rauschen bei MORPHEUS, Funk-Kernel bei HCM, Sortieren bei BORPH). Ausgewählte Möglichkeiten zur Nutzung von heterogenen Systemen und insbesondere rekonfigurierbaren Beschleunigern im Bereich bioinformatischer Anwendungen und numerischer Berechnungen sowie die dabei auftretenden Probleme werden dementsprechend in den Kapiteln 4 und 5 betrachtet. Das betrachtete Spektrum möglicher Ansätze reicht von Koprozessoren auf der untersten Instruktionsebene bis hin zur hochsprachenbasierten Portierung einer Anwendung. Als bioinformatische Anwendung wurde HHblits ausgewählt. Angefangen mit einer leistungsschwachen Software-Parallelisierung der Intervallarithmetik wird zur Unterstützung numerischer Berechnungen dann das exakte Skalarprodukt auf Festkommabasis betrachtet und mehrere Koprozessoren dazu entworfen. Anhand der Hochsprache Impulse C und eines Koprozessors zur Lösung der Poissongleichung werden verschiedene Probleme bei der Verwendung von Quelltextkonvertern für Hochsprachen illustriert. Dies sind jedoch alles nur Speziallösungen, die noch nicht allgemein nutzbar sind. Als Ergebnis all dieser Untersuchungen muss ein leicht, bequem und schnell programmierbares, portables Programmier- und Ausführungsmodell mit Anleitung zum sinnvollen Vorgehen zur Nutzung von heterogenen, FPGA-gestützten Systemen hervorgebracht werden.

Auch mit OpenCL bleibt ein weiterhin ungelöstes Problem die datengetriebene Ausführung von Kernels zur verbesserten Ausnutzung der verfügbaren Hardware und damit zur Steigerung der erzielbaren Leistung mittels Ausnutzung von Task- oder gar Pipelineparallelismus. Denn sind die Daten einmal im Level-2-Cache bzw. auf dem Koprozessor vorhanden, sollten sie solange, früh und oft wie möglich verwendet werden durch weitere Tasks, indem z.B. Ergebnisse an andere Speicherstellen geschrieben werden und entsprechend keine Kohärenzprobleme auftreten können (Erhöhung der Datenlokalität). Bei dem GPU-Taskmodell von OpenCL überlappen sich abhängige Tasks nicht, so dass kein internes Streaming oder positive Cache-Effekte ausgenutzt werden. Zum Streaming von Eingabedaten an einen Kernel müssen diese Daten gleichermaßen wie bei CUDA vollständig im Voraus berechnet sein [166]. Die Firma Maxeler löst das Problem der Datenanbindung über die Ausführung nach dem Datenflussprinzip. Die Algorithmen werden in einem Java-Dialekt beschrieben und ähnlich weiteren Hochsprachen und Quelltextkonvertern wie Impulse C und ROCCC zu einer Hardwarebeschreibung konvertiert. Damit sind bereits die FPGAs programmiert, zudem sind sie statisch eng verbunden. Es mangelt jedoch daran, dass der gesamte FPGA mit einem Design fest programmiert ist, bis es überschrieben wird. Dies bedingt dann häufig, dass ein neues Design erst zeitintensiv synthetisiert, platziert und geroutet werden muss.

Mit dem Hintergrund des Lösens numerischer Probleme auf rekonfigurierbaren Mehrkernsystemen

fällt der Fokus stark auf das Rechnen in digitaler Arithmetik, vor allem mittels Gleitkommaformaten. Zahlreiche numerische Operationen und Algorithmen wurden bereits auf Akzeleratoren wie FPGAs verlagert unter Verwendung von gekürzten und regulären Gleitkommadarstellungen. Diese sind jedoch nicht völlig gefahrlos verwendbar, wenn es sich um kritische Berechnungen wie Turbinenschwingungen in Atomkraftwerken handelt. Die genauere, sichere Berechnung ist sehr rechenintensiv und zeitaufwendig und stellt damit einen ersten Kandidaten zur beschleunigten Implementierung mittels FPGAs in rekonfigurierbaren Rechnern dar. Der Algorithmus für ein exaktes Skalarprodukt [164, 172], das solche exakten Berechnungen unterstützt, birgt eine Reihe von Datenabhängigkeiten, welche die uneingeschränkte interne Parallelisierung nicht möglich machen. Das Skalarprodukt eignet sich dennoch gut zur Verlagerung in Hardware, da das sehr breite Register zur internen Speicherung des Werts gut umgesetzt werden kann und da mehrere Einheiten gleichzeitig angesprochen werden können. Vorteil lässt sich aufgrund der zehn- bis zwanzigfach niedrigeren Taktrate der FPGAs allerdings erst dann aus solchen Implementierungen ziehen, wenn das Problem der Datenübertragung gelöst ist.

Zur Lösung dieses Problems setzt die Hochsprache Impulse C bei der Programmierung heterogener, FPGA-erweiterter Systeme bekannte Programmiermodelle, welche genau das Kommunikationsproblem angehen, ein. Allerdings vermögen Hochsprachen nicht die Details der FPGA-Hardware auszunutzen, sondern müssen auf Grundbausteinen wie Gleitkommaoperationen aufbauen. Diese wiederum sind nicht auf die Anwendungsdomäne zugeschnitten. Und da sie allgemein gehalten sind und überdies eventuell noch parametrierbar, sind sie zumeist platzineffizient, so dass weniger rekonfigurierbare Fläche für weitere Logik übrig bleibt. Um auf die Hardware zugeschnittene Kernel-Portierungen zu haben, greift man vorzugsweise auf vorgefertigte Routinen und Bibliotheken zurück, wie etwa CUBLAS zum Rechnen auf GPUs. Jetzt steht man aber wieder vor den Problemen, die Datenübertragung und Ausführung optimieren sowie entscheiden zu müssen, wann sich der Einsatz von CUBLAS anstelle einer Softwareroutine lohnt. Häufig genügt dabei obendrein die vorliegende Kernelimplementierung nicht den Anforderungen und muss entsprechend erweitert oder gar neugeschrieben werden. Als Lösungsansatz dazu bietet es sich wiederum an, feingranularere Recheneinheiten oder Konfigurationen bereitzustellen, die z.B. über eine Befehlssatzerweiterung ansteuerbar sind [4, 23].

Im Rahmen der hiesigen Arbeit wird unter anderem untersucht, inwieweit komponentenbasierte Programmierung einerseits für numerische und bioinformatische Programme geeignet ist, und andererseits wie die Integration von Hardwareimplementierungen in solchen Programmiermodellen erreichbar ist. Abstrahiert man nämlich die Grundbausteine von Instruktions-/Operationsebene weiter zu vollständigen Funktionen bzw. Einheiten im Sinne der komponentenbasierten Programmierung, macht dann die Kommunikation (insbesondere den Datenfluss) transparent für den Programmierer und sorgt schließlich für effiziente[25], automatische Behandlung selbiger, so sollte sich ein leistungsfähiges Modell zur Entwicklung aktueller und zukünftiger heterogener Mehr- und Vielkernsysteme herausstellen. Somit stellt sich die Frage, ob und wie Koprozessorressourcen in ein solches Programmiermodell für Streaming-Anwendungen integriert werden können. Basierend auf den aus den Arithmetikkoprozessoren gewonnenen Erkenntnissen werden mehrere Rahmenwerke und Schnittstellen in Kapitel 6 untersucht, mit denen eine oder mehrere Koprozessoreinheiten auf einem FPGA integriert und von einer oder mehreren Anwendungen gleichzeitig genutzt werden können. Das besondere Problem dabei ist, eine Möglichkeit zu finden, den Koprozessor eng und nahtlos so einzubinden, dass die Gesamtausführungszeit minimiert wird, wozu es der bereits mehrfach aufgeführten Betrachtung der Datentransferproblematik auch von Host- bzw. Anwendungsseite aus bedarf.

Das Ziel ist, eine Schnittstelle für Anwendungsentwickler zu FPGAs zu schaffen, die über Komponenten funktioniert. Die Verschaltung der Komponenten miteinander soll über Mikroprogrammierung erfolgen. Dies bietet zweierlei Vorteile. Einerseits sind direkt vom Hardwarespezialisten entwickelte Schaltungen nutzbar, andererseits kann der FPGA einfach vom Anwendungsentwickler neu programmiert werden, indem ein anderes Mikroprogramm übergeben wird. Der FPGA soll über DMA-Einheiten an den Host angeschlossen werden, die zeitgleich zueinander und zu den übrigen Komponenten arbeiten. Derart werden die Daten gestreamt. Intern erfolgt die Datenverbindung der Komponenten über einen großen Puffersatz, um datengetriebene Ausführung zu erreichen.

[25] mit geringstmöglicher Verzögerung, mit größtmöglichem Durchsatz

Aus den Entwürfen und Implementierungen zur exakten Arithmetik und zu speziellen Anwendungen wird erkennbar, dass Koprozessoren und FPGAs im Speziellen nur dann sinnvoll nutzbar sind, wenn die genauen *Eigenschaften* der Implementierung auch bekannt sind und mit den *Anforderungen* der Anwendungen auch in Einklang stehen. Aus diesem Grund wird in Kapitel 7 das Konzept der Attributierung entwickelt, die implementierte Werkzeugkette evaluiert sowie in ein Laufzeitsystem zur leistungsorientierten Verwendung von Implementierungsalternativen aus einer Bibliothek eingebettet und ebenso evaluiert. Die Attributierung ist orthogonal zu den Schnittstellen und Rahmenwerken, da die Verwendung der Hardware über Funktionsaufrufe aus Software erfolgt, die wiederum von dem Laufzeitsystem ausgewählt werden.

Dem Anwendungsentwickler soll die Möglichkeit gegeben werden, die rekonfigurierbare Hardware auf Funktions- beziehungsweise Taskebene auszunutzen und dabei die Attributierung einzusetzen. Dazu wird in Kapitel 8 unter anderem eine grafische Programmierumgebung skizziert, mit der Anwendungsgraphen erstellt werden. Daraus könnten sich die Verschaltungen der Funktionsimplementierungen erzeugen lassen, so dass die Funktionsimplementierungen dann datengetrieben überlappt ausgeführt werden können. Dies wird beispielsweise mit $(\text{GECO})^2$ [217] bereits erreicht. Führt auch der Koprozessor eine datengetriebene Komponente oder mehrere aus, so können die Daten über einen Stream zwischen Komponenten auf Host- und FPGA-Seite ausgetauscht werden. Dies behebt das Datentransferproblem und ermöglicht zusätzlich, unterschiedliche Hardware-Software-Partitionierungen einfach und bequem zu testen.

In Kapitel 9 werden die entwickelten Methoden evaluiert. Es wird geprüft, wie Stencil-basierte und bioinformatische Anwendungen, die mitunter wiederum Operationen der Linearen Algebra verwenden, von der Convey HC-1 und vom datengetriebenen Programmiermodell profitieren können. Ein datengetriebenes und vom Nutzer mikroprogrammierbares Steuerwerk wird auf der Convey HC-1 dazu eingesetzt, das Konjugierte-Gradienten-Verfahren zur Lösung des Poisson-Problems zu beschleunigen. Da die dabei benötigte Skalarproduktoperation zunächst ressourcensparend ohne dedizierte Akkumulatorpipeline implementiert ist, lässt sich leicht feststellen, dass der dabei auftretende Genauigkeitsverlust über exaktere Arithmetik zu beheben ist. Auf der Convey HC-1 ist ein Prozessorkern mit der Ansteuerung des Koprozessors beschäftigt. Das datengetriebene Programmiermodell ermöglicht es, den unbeschäftigten zweiten Kern gewinnbringend zur Verrichtung weiterer Rechenaufgaben einzusetzen. Dies wird anhand einer datengetriebenen Softwareimplementierung der bioinformatischen Anwendung HHblits gezeigt.

Kapitel 10 fasst die Arbeit zusammen. Dabei wird der individuelle Beitrag herausgestellt. Ferner sind mögliche zukünftige Weiterentwicklungen der vorgestellten Implementierungen und Verfahren skizziert. Der Ausblick auf den möglichen Beitrag hinsichtlich der Entwicklung FPGA-erweiterter Systeme schließt diese Dissertation ab.

Im folgenden Kapitel ist zunächst dargestellt, wie eine Anwendung in rekonfigurierbare Hardware verlagert wird. Dadurch wird das generelle Vorgehen bei der Verlagerung erkennbar und im Besonderen auch, welche analytische Leistung seitens des Hardwareentwicklers vollbracht werden muss, um Nutzen aus der Verlagerung zu erzielen. Damit begründet sich, warum Hochsprachen für die Verlagerung ungeeignet sind, wenn wie in der Motivation für diese Arbeit Anwendungsentwickler anstelle von Hardwareexperten die Verlagerung vornehmen sollen oder wollen. Ebenso zeigt sich, wie wesentlich die Beachtung der Kommunikation ist. Auch diese Aufgabe vermögen Hochsprachen nicht automatisiert vorzunehmen.

Kapitel 4

Verlagerung einer bioinformatischen Anwendung in rekonfigurierbare Hardware

Anwendungen der Bioinformatik können hervorragend von Hardwareunterstützung profitieren (vgl. Abschnitt 2.1.2). In diesem Kapitel wird anhand einer ausgewählten bioinformatischen Anwendung gezeigt, wie das prinzipielle Vorgehen beim Verlagern von Funktionalität in rekonfigurierbare Hardware ist, welche Leistung durch Hardwareunterstützung erzielbar ist und was dabei zu beachten ist [BN13, NBSK13]. So wird erkennbar, warum die verwandten Arbeiten nicht ausreichend sind, Anwendungsentwickler bei der Nutzung rekonfigurierbarer Hardware zur Beschleunigung von Anwendungen ausreichend zu unterstützen. Die bei der Verlagerung gewonnenen Erkenntnisse sollen später angewandt und übertragen werden sowie Lösungen für die aufgezeigten Probleme entwickelt werden. Das Kapitel gliedert sich in einen einleitenden, vom Stand der Forschung abgrenzenden Teil, in die Beschreibung des Vorgehens bei der Implementierung eines Anwendungsbeschleunigers in rekonfigurierbarer Logik sowie in die Zusammenfassung der gewonnenen Erkenntnisse.

4.1 Bioinformatische Anwendungen

Zur Krebsforschung und in anderen medizinisch-biologischen Forschungsgebieten ist der DNS[1]-Vergleich ein gutes, aber sehr zeitaufwendiges Hilfsmittel. Dabei wird prinzipiell nach dem Prinzip des dynamischen Programmierens vorgegangen, wo die Editierdistanz zwischen zwei Zeichenketten ermittelt wird. Als Besonderheit im Bereich der Genanalyse werden auch mehrere Sequenzen miteinander verglichen, was gegenüber paarweisem Vergleich exponentiellen Aufwand hat. Genau hier können massiv parallele Vektoreinheiten oder -prozessoren ihre Leistung ausspielen, indem sie nicht nur einzelne Vergleiche intern parallel ausführen, sondern mehrere gleichzeitig. Die hohe erzielbare Beschleunigung ist in der Regel auf sehr kleine Datentypen zurückzuführen. Für Datentypen unterhalb der Wortebene und für Verfahren wie dynamisches Programmieren eignen sich FPGAs hervorragend.

4.1.1 Beschleunigung bioinformatischer Anwendungen durch rekonfigurierbare Akzeleratoren

Wie in Abschnitt 2.1.2 beschrieben wurde, konnten bereits zahlreiche bioinformatische Anwendungen durch rekonfigurierbare Akzeleratoren maßgeblich unterstützt werden. Grundlage vieler bioinformatischer Anwendungen ist der Sequenzvergleich. Beispielsweise werden Teilseqenzen aus der DNS extrahiert und in Datenbanken mit Millionen von Sequenzeinträgen wird gesucht, in welchen Sequenzen die extrahierten Teilsequenzen enthalten sind. Eine Schwierigkeit dabei ist, geringe Abweichungen an einigen wenigen Stellen tolerieren zu müssen aufgrund von möglichen Mutationen. Um herauszufinden, ob eine Teilsequenz in einer Datenbanksequenz enthalten ist, wird häufig das Verfahren von Smith und Waterman gewählt [243]. Das von ihnen entwickelte Verfahren ermöglicht, diejenigen Stellen zu finden, an denen die Teilsequenz maximal übereinstimmt, und einen zugehörigen Pfad. Dazu wird auf das Prinzip des dynamischen Programmierens zurückgegriffen, was in rekonfigurierbarer Hardware häufig mittels systolischer Arrays umgesetzt wird [39, 247]. Für eine

[1] Desoxyribonukleinsäure

Teilsequenz mit m Zeichen reichen beim Vergleich gegen eine n Zeichen lange Referenzsequenz aus der Datenbank dann m Elemente im Array aus. Durch die parallele Berechnung im systolischen Array reichen dann $m+n-1$ Schritte aus, um den gesamten Vergleich durchzuführen. Anschließend müssen die Maxima und die Positionen der Maximalwerte festgestellt werden.

4.1.2 Bioinformatik auf der Convey HC-1

Auf das heterogene, FPGA-erweiterte System „Convey HC-1“ wurde schon eine Reihe von bioinformatischen Anwendungen portiert („Convey Bioinformatics Suite“ [62, 228, 268]). BLAST [8, 67] und das Smith-Waterman-Verfahren [62, 65, 242, 270] wurden verlagert. Bei der anzugehenden Anwendung HHblits aus der HH-suite wurde das Verfahren von Smith und Waterman so verändert, dass durch die bestehenden FPGA-Implementierungen keine Leistungssteigerung erzielbar sein wird. HMMER wurde schon untersucht [196], aber auch HMMER3 soll kommen. Die InsPecT Proteomics-Personality [60, 251] ist bereits vorhanden, ebenso wie der Burrows-Wheeler-Aligner (BWA) [63, 66, 185]. Die Neuzusammensetzung von Sequenzen („Short-Read de-novo Assembly“) und die Unterstützung durch Sequenzausrichten sowie Histogramme [64, 269, 220, 235, 289], die Ausrichtung kurzer Sequenzen zueinander mittels minimalem perfekten Hashing in einer Hardwarepipeline [204] und MrBayes-Phylogenetik [295] vervollständigen den Stand der Forschung mit Implementierungen auf der Convey HC-1.

4.1.3 HH-suite

Die HH-suite [250] ist ein Werkzeugkasten eines Forschungsprojekts des Genzentrums der Ludwig-Maximilian-Universität München[2] und des Max-Planck-Instituts für Entwicklungsbiologie in Tübingen[3]. Die in der HH-suite enthaltene Anwendung *HMM-HMM-based lightning-fast iterative sequence search* (HHblits) [222] dient dazu, gemeinsame Vorgänger zwischen unterschiedlichen Gensequenzen zu finden. Darüber können sich dann gewisse Aussagen treffen lassen hinsichtlich Aufbau und Struktur, Wirkung von Medikamenten u.ä. Das Besondere an HHblits gegenüber anderen Sequenzvergleich-Anwendungen ist, dass es zum Suchen und Finden homologer Sequenzen auf Hidden-Markov-Models (HMM) arbeitet. Es fasst mehrere hintereinander auftretende Zeichen in einem Zustand eines HMMs zusammen und vergleicht anstelle von Zeichenvergleichen nun die HMMs und ihre Zustände. Nach einem Lauf kann das HMM der Anfragesequenz um die bereits gefundenen homologen HMMs erweitert werden, um so in weiteren Läufen noch mehr Homologien zu finden. Zudem wurden die Bewertungsfunktion beim Sequenzvergleich vereinfacht mit zwei Operanden weniger: der linke und obere Nachbar müssen nicht mehr ausgewertet werden bei der Berechnung eines Elements. HHblits ist dadurch bis zu doppelt so schnell wie PSI-BLAST [222]. Die Beschleunigung von HHblits erfolgte bislang ausschließlich über die Verwendung von SSE nach M. Farrar [96]. Bestehende Implementierungen zur Beschleunigung des allgemeinen Smith-Waterman-Verfahrens können aufgrund der vorgenommenen Optimierungen des Verfahrens keine Beschleunigung für diese HMM-Vergleiche liefern. Mit der Verwendung von HMMs stellt HHblits einen interessanten Kandidaten zur Verlagerung in Akzeleratorhardware dar.

Der Ablauf der Anwendung ist in Abb. 4.1 dargestellt. Für Sequenzketten wurden vorab des Vergleichs zunächst HMMs mittels weiterer Anwendungen erstellt. Zueinander ähnliche Sequenzen wurden in der Datenbank zu Clustern zusammengefasst, gegen die verglichen werden soll anstatt gegen die Sequenzen selbst, um so die Laufzeit zu verbessern. Entsprechend wird auch aus der Anfragesequenz, zu der homologe Sequenzen (eigentlich: Cluster) aus der Datenbank zu finden sind, bei der Ausführung von HHblits zuerst ein HMM erzeugt (Anfrage-HMM, vgl. Abb. 4.1, Schritt 1). Danach wird die Vorfilterung initialisiert und alle Sequenzen bzw. ihre zugehörigen HMMs aus der Datenbank geladen. Nach der Initialisierung kann die Hauptschleife ausgeführt werden, und zwar so oft, wie der Nutzer dies wünscht, wobei mit jedem Lauf mehr Ergebnisse mit entfernterem Verwandtschaftsverhältnis gefunden werden. Aus dem Anfrage-HMM wird ein einfaches Profil erstellt (Schritt 2). Mit diesem werden in Schritt 3 unpassende HMMs aus der Proteindatenbank herausgefiltert (`prefilter_db()`), um weniger HMMs später miteinander vergleichen zu müssen.

[2] URL: `http://www.genzentrum.lmu.de/`
[3] URL: `http://www.eb.tuebingen.mpg.de/de.html`

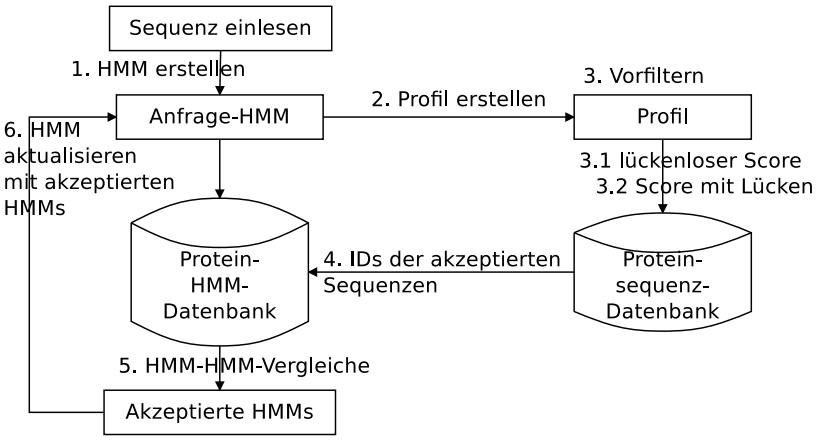


Abbildung 4.1: Ablauf von „HMM-HMM-based lightning-fast iterative sequence search (HHblits)“ (angelehnt an [222]).

Dieser Vorfilter (HHPrefilter) ist ein Bestandteil von HHBlits, der unmittelbar aus der Methode `main` aufgerufen wird. Die zugehörige Methode `prefilter_db()` (vgl. Abb. 4.2) ruft mehrere projekteigene Methoden auf. Die zeitaufwendigste und damit wichtigste ist `ungapped_sse_score`. Sie berechnet für das Anfrageprofil einen lokalen Ähnlichkeitswert ohne Lücken (und ohne Einfügungen) gegen jeden Eintrag in der Datenbank (Schritt 3.1). Ist der Wert oberhalb eines festgelegten Schwellwerts, so wird im zweiten Schritt (3.2) ein lückenbehafteter Wert errechnet unter Zuhilfenahme einer streifenorientierten Implementierung des Smith-Waterman-Algorithmus. Bei allen vom Vorfilter akzeptierten Einträgen (Schritt 4) werden die zugehörigen HMMs mit dem Anfrage-HMM mittels einer Viterbi-Suche verglichen (Schritt 5). Wird bei der Viterbi-Suche ein Ergebnis oberhalb eines weiteren Schwellwerts erreicht, so werden die Eingabesequenz und die Sequenz aus der Datenbank als zueinander homolog erachtet. Abschließend werden die akzeptierten HMMs noch aneinander ausgerichtet und das Anfrage-HMM um die akzeptierten HMMs erweitert (Schritt 6). Der nächste Lauf, insofern überhaupt erwünscht, verwendet dann dieses aktualisierte HMM und findet damit auch entfernter homologe Sequenzen. Die Anzahl zeitintensiver Viterbi-Vergleiche nimmt aufgrund der höheren Anzahl akzeptierter Sequenzen in weiteren Läufen entsprechend zu.

HHBlits ist insgesamt sehr zeitaufwendig. Mehrkernprozessoren können diese Verfahren teilweise datenparallel ausführen. Zu untersuchen ist, wieweit die Convey HC-1 zur weiteren Beschleunigung geeignet ist. Dazu eignen sich verschiedene Techniken. Eine Technik basiert auf dem Einsatz von Compilern und arbeitet mit einer massiv parallelen Vektoreinheit. Eine andere Technik ist die gezielte Beschleunigung geeigneter Teile über handentwickelte Spezialeinheiten unter Berücksichtigung der Speicherhierarchie, Kommunikation sowie Hardwareeigenschaften wie Taktfrequenzen und Busbreiten.

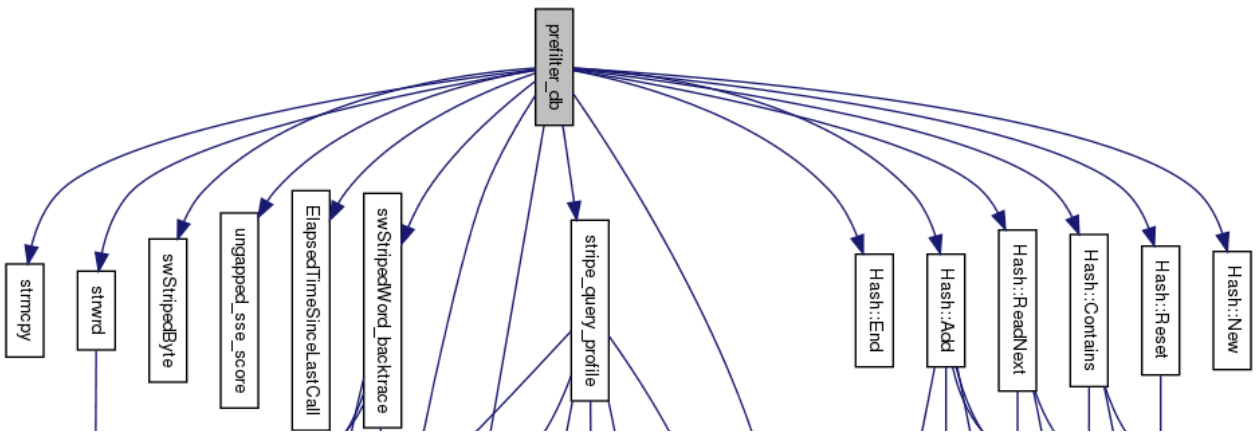


Abbildung 4.2: Callgraph der Vorfilterung prefilter_db() in HHblits.

4.2 Homologiesuche in Sequenzdatenbanken mit HHblits auf der Convey HC-1

Um die Vorgehensweise sowie den zeitlichen Aufwand beim Verlagern von Algorithmen aus Software in Hardware zu beschreiben und zu illustrieren, präsentiert dieser Abschnitt, wie HHblits auf dem heterogenen System Convey HC-1 beschleunigt werden kann [BN13, NBSK13, BNK14]. Dazu ist zunächst nötig, ein Laufzeitprofil der Anwendung zu erstellen, um die beschleunigenswerten Programmteile zu identifizieren. Hat man dies erreicht, so werden diese Teile genauer betrachtet. Ausgehend von der aktuellen Implementierung in Software wird ein erstes Hardware-Design erstellt. In diesem Design werden Engpässe gesucht und nach Möglichkeit behoben, wobei es vorteilhaft sein kann, die erzielten Unterschiede in der Ausführungszeit bzw. im Durchsatz durch Messreihen nachvollziehbar zu quantifizieren. Auf Basis dessen wird ein zweiter Ansatz vorgestellt, welcher die detektierten Schwachstellen behebt.

4.2.1 Laufzeitanalyse von HHblits hinsichtlich Beschleunigbarkeit

Um eine Aussage über den Laufzeitbedarf der Anwendung HHblits und die Aufteilung auf unterschiedliche Teile zu erhalten, wurde HHblits beim Übersetzen mit GCC[4] mittels des Compilerschalters `-pg` zum Erzeugen von Laufzeitprofilen instrumentiert sowie OpenMP weggelassen. Durch Ausführung der Befehlszeile

```
time hhblits -cpu 1 -i /path/to/hhsuite-2.0.15/data/query.seq
    -d /path/to/hh/databases/uniprot20/uniprot20_02Sep11 -oa3m query.a3m -n 1
```

wurde ein Laufzeitprofil für den Intel Xeon 5138 @ 2.13 GHz erhalten, der auf der Convey HC-1 installiert ist. Die Vorfilterung wird lediglich ein Mal pro Durchlauf ausgeführt (Parameter `-n 1`), die Bewertung unter Zuhilfenahme von SSE (`ungapped_sse_score`) jedoch mehr als sechs Millionen mal für alle Einträge in der verwendeten Datenbank uniprot20. Die Vorfilterung hat einen Anteil von 67,37 % an der Gesamtausführungszeit und ist daher beschleunigenswert. Geringfügig beschleunigenswert sind außerdem noch die Viterbi-Suche (bzw. der -Algorithmus) mit 7,01 % sowie `CalculatePosteriorProbs` mit 13,13 %, und die Smith-Waterman-Implementierung `swStripedByte` mit 4,00 % der Gesamtzeit von 74,08 Sekunden. Lässt man das Verfahren zwei Mal laufen (`-n 2`), so verschieben sich die Werte leicht zum Viterbi-Algorithmus hin, und die Ausführungszeit steigt auf 189,65 Sekunden.

Vorabschätzung zur Eignung des Vorfilterungsschrittes zur Beschleunigung. HHblits arbeitet auf kleinen Daten von nur 8 Bit Größe, von denen mittels Multimediaerweiterungen (SSE2) 16 parallel zueinander in der vorliegenden Implementierung von HHblits verarbeitet werden. Nach der Anwendung arithmetischer Operationen auf diese Daten werden sie reduziert durch die Bestimmung des globalen Maximums, was in $log_2(n)$ Schritten bei Farrars Implementierung mit SSE2 erfolgt. Der FPGA-basierte Koprozessor auf der Convey HC-1 hingegen bietet 8 Speichercontroller, die bei 300 MHz 64 Bit pro Takt liefern bzw. 2 mal 64 Bit bei 150 MHz-Taktung des Koprozessors. Somit können 2*8 Einheiten je Speichercontroller und damit 16*8=128 Einheiten je FPGA die 8-Bit-Daten mit 150 MHz verarbeiten.

Mit vier FPGAs auf der HC-1 stehen also theoretisch bis zu 512 Einheiten zur Verfügung. Das ist zwar das 16-fache des mit 2136 MHz getakteten Xeon-Prozessors der Convey HC-1, aber mit 512 Zeichen/Takt * 150 MHz = 25.600 M Zeichen/s anstelle von 32 * 2.136 MHz = 68.352 M Zeichen/s ist nur ein Drittel des Durchsatzes zunächst zu erwarten. Es muss eine gute Hardware-Pipeline entwickelt werden und in die gesamte Anwendung möglichst nahtlos integriert werden. Daher ist insbesondere zu untersuchen, ob die Speicherarchitektur der Convey geeignet ist, die Ausführungseinheiten schneller mit Daten zu versorgen, als dies bei SSE2 bzw. bei über den Intel Front-Side Bus (FSB) an den Speicher angeschlossenen Mikroprozessoren der Fall ist. Weiteres Potential liegt in der Erhöhung der Taktraten der Verarbeitungseinheiten. Dies ist deshalb denkbar, weil es sich

[4] Die HH-suite bietet explizit auch die Nutzung vom ICC an.

um einfache Byte-Operationen handelt und weil die Menge an Daten im Verlauf der Vorfilterung stark reduziert wird. Darüberhinaus würde die ohnehin kaum genutzte Schreibbandbreite besser ausgenutzt, wenn in einem kürzeren Zeitraum mehr Daten reduziert würden.

Zur genaueren Bestimmung des tatsächlich erzielten Durchsatzes von `ungapped_sse_score` wurden einzelne Durchläufe für jede Sequenzvorfilterung um Zeitmessung auf Basis von `gettimeofday` und damit auf Millisekundenbasis erweitert. Diejenigen Ausführungen mit sehr kurzer Dauer wurden dabei nicht berücksichtigt. Das erhaltene Ergebnis ist ein Durchsatz von 7,178 GB/s bis 7,757 GB/s. Das Anfrageprofil mit 220 Zeilen und 512 Byte pro Zeile benötigt lediglich 110 KB an Speicherplatz und sollte daher vollständig im Cache gespeichert sein, insofern keine ungeschickten Speicherzugriffsmuster zu wiederholter Verdrängung führen. Die Speicherbandbreite von 8,5 GB/s der Intel-FSB-Schnittstelle ist also die limitierende Komponente, denn eine einzelne SSE-Einheit könnte bei einer Taktrate von 2136 MHz mit 16 Byte je Takt insgesamt 34,176 GB/s an Durchsatz erzielen. Der FPGA-basierte Koprozessor hat hingegen eine Speicherbandbreite von 76,8 GB/s und könnte daher zehnfache Beschleunigung erzielen. Überdies kann die interne Datenvorhaltung in den FPGAs weiter bei der Optimierung des Datentransfers und der -wiederverwendung helfen.

Der erste Schritt der Vorfilterung scheint es also wert, auf den Koprozessor der Convey HC-1 verlagert zu werden. Es ist jedoch zu beachten, dass der zu erwartende Speedup $S(n = 10,\ a = 1 - 0,6737) = 2,54$ nicht sonderlich hoch ist. Werden ferner die reduzierten Daten bereits innerhalb des FPGAs weiterverarbeitet, kann evt. zusätzliches Optimierungspotential freigelegt werden und höhere Beschleunigung erzielt werden.

Eignung vom Smith-Waterman-Algorithmus zur Beschleunigung. Der Smith-Waterman-Algorithmus benötigt etwa 4% der Laufzeit und wurde bereits von Convey Computer und Partnern als Personality verfügbar gemacht.

Beschleunigung von CalculatePosteriorProbs. Die Funktion `CalculatePosteriorProbs` ist nicht beschleunigenswert, da die Funktion stark von der Berechnung der Exponential- und Logarithmus-Funktion abhängt, wobei die beste, als Pipeline implementierte FPGA-Implementierung [5] bei über 300 MHz FPGA-Taktfrequenz für 10^8 Berechnungen dennoch langsamer ist als in Software auf einem Intel Core2Duo T9600.

4.2.2 Ungapped SSE Score

Es geht in HHBlits generell und in der Vorfilterung mittels Heuristiken darum, herauszufinden, ob zwei Sequenzen sich ähnlich sind und daher die gleichen Vorfahren haben könnten. Bei der Bewertung der einzelnen Datenbanksequenzen gegenüber der Anfragesequenz hinsichtlich der möglichen Homologie, wie in `ungapped_sse_score` implementiert, geht es darum, die Hidden-Markov-Modelle der zwei Sequenzen miteinander grob abzugleichen. Dazu modellieren die HMMs die Übergangswahrscheinlichkeiten vom Zustand eines Zeichens z_i zum nächsten Zeichen z_{i+1}. In der HMM-Anfragesequenz steckt für jedes mögliche, gerade in der anderen Sequenz gefundene Zeichen (=Zustand) die Übergangswahrscheinlichkeit zu den nächsten Zeichen. Diese Wahrscheinlichkeiten lassen sich unterschiedlich genau modellieren und berechnen. Beim lückenlosen Übereinstimmungswert soll einfach nur das Maximum einer Evaluationsfunktion in der gesamten Ergebnismatrix ermittelt werden. Dazu muss zu jedem Zeichen der Referenzsequenz das entsprechende Band (=Zeile) des Profils der Anfrage elementweise verglichen werden. Mit der Methode des dynamischen Programmierens wird der Funktionswert berechnet und für das nächste Element weiterverwendet (Subterme). Dabei handelt es sich jedoch um eine Spezialimplementierung, die mittels streifenbasierter Berechnung die Ausnutzung von Parallelität erlaubt und zudem im Gegensatz zum allgemeinen Smith-Waterman-Algorithmus nur den linken oberen, diagonalen Nachbarn benötigt. Daher ist sie also nicht mit der Smith-Waterman-Personality beschleunigbar.

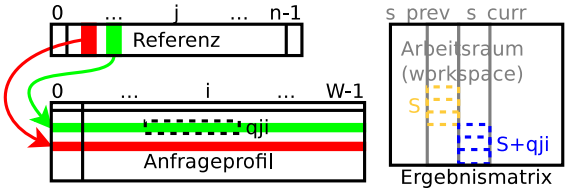


Abbildung 4.3: Berechnung eines lückenlosen Scores.

```
1: W = (query_length+15) / 16 /* 16-Byte block count */
2: uint128 workspace[2*W]
3: s_prev = &workspace[W]
4: s_curr = &workspace[0]
5: for j=0, ..., n-1 do
6:     qji = query_profile [ sequence[j]*W ]
7:     S = s_curr[W-1]
8:     SWAP(s_prev, s_curr)
9:     for i=0, ..., W-1 do
10:        S = S + qji[i] - Soffset
11:        Smax = max(Smax, S)
12:        s_curr[i] = S
13:        S = s_prev[i]
14:    end for
15: end for
16: return component-wise_max(Smax)
```

Algorithmus 1: Berechnung eines lückenlosen Scores.

SSE-Implementierung in HHblits. Der Implementierung von `ungapped_sse_score` werden ein (2-dimensionales!) Anfrageprofil als `char*` (die erste Dimension enthält eine Sequenz für jedes Alphabetzeichen in der zweiten Dimension, Abb. 4.3) und die Länge dieses in Zeichen/Bytes, eine (1-dimensionale) Referenzsequenz aus der Datenbank als `char*` und die Länge dieser in Zeichen/Bytes sowie ein Arbeitsraum der Länge 3*Anfragesequenzlänge als Parameter übergeben.

Die gesamte Implementierung wie in Algorithmus 1 arbeitet auf dem aus 16-Byte-Blöcken aufgebauten Arbeitsraum. Zunächst wird der Arbeitsraum mit 0 initialisiert. Anschließend werden Zeiger auf den Arbeitsraum angelegt – `s_curr` auf den Anfang und `s_prev` um die Anfragesequenzlänge weiter, also in die nächste Spalte des Arbeitsraums, ausgerichtet an der Anfragesequenz (vgl. Abb. 4.3). Dann beginnt die äußere Schleife, über die Referenz Zeichen für Zeichen zu iterieren, wobei `s_curr` und `s_prev` jeweils geschickt miteinander vertauscht werden, um auf den drei Spalten des Arbeitsraums den gesamten Algorithmus durchführen zu können.

Innerhalb der Schleife wird zunächst ein Zeiger `qji` initialisiert, mit dem über die zum Zeichen der Referenzsequenz passende Profilzeile der Anfrage iteriert werden kann (s. Abb. 4.3). Die Werte der aktuellen Spalte werden berechnet als $S_{\text{current}} := S_{\text{prev}} + *qji - \text{Soffset}$. Zeitgleich wird fortlaufend das maximal vorgekommene S ermittelt und festgehalten.

4.2.3 Eine Convey-Personality für die Berechnung des lückenlosen Scores

Nach dem Einarbeiten in Organisation und Ablauf der Anwendung sowie nach der Analyse der Anwendung hinsichtlich der Ausführungszeit der Bestandsteile erfolgt beim Entwickeln eines Anwendungsbeschleunigers zunächst ein gezielter Entwurf. Dieser wird dann schrittweise implementiert und validiert, um anschließend sukzessive verbessert zu werden (vgl. V-Modell, Abschnitt 2.1.2, Abb. 2.4). Der erste Ansatz ist entsprechend geradlinig in Anlehnung an die bestehende Implementierung in SSE. Der zweite Ansatz unterscheidet sich vom ersten Ansatz im Wesentlichen dadurch, dass Daten im Voraus in den Koprozessor geladen und durchweg vorgehalten werden. Die zugehörige Implementierung kann überdies besser skaliert werden hinsichtlich der internen Verarbeitungseinheiten, da FPGA-Spezifika wie Speicher mit zwei Ports[5] ausgenutzt werden können.

Erster Ansatz auf Basis der vorliegenden SSE-basierenden Implementierung. Für die Convey HC-1 kann anhand der Implementierung in Software mit SSE und anhand der bereits erlangten Erkenntnisse aus der Analyse ein Koprozessor zur Beschleunigung des ersten Vorfilterungsschrittes von HHblits wie in Abb. 4.4 veranschaulicht aufgebaut werden. Es wird mit dem Speicherzugriffsmodus „binary interleaved“ gearbeitet; die Eingabedaten werden also auf acht Spalten verteilt.

[5] engl. Dual-Port RAMs (DPRAMs)

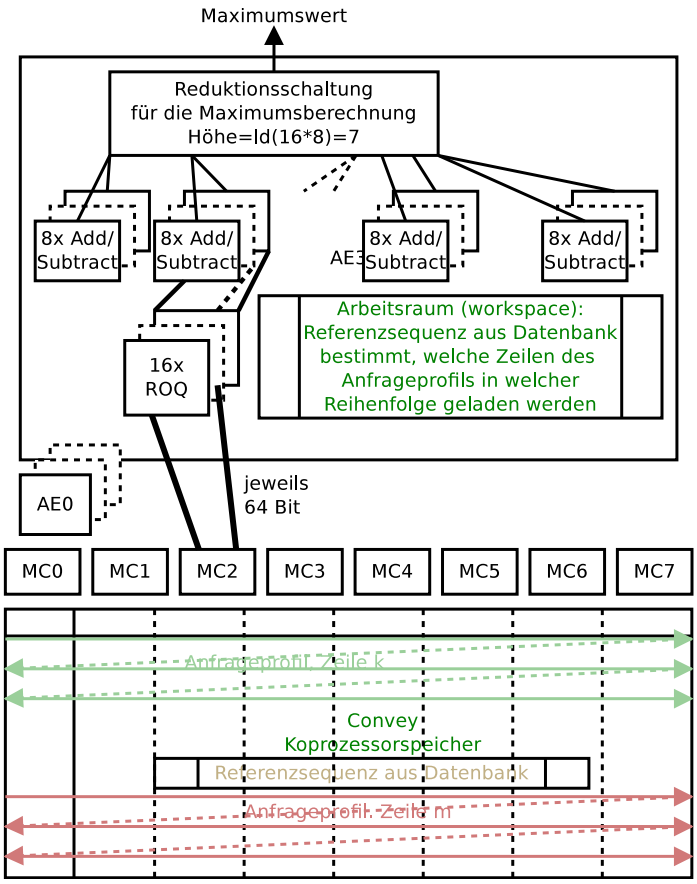


Abbildung 4.4: Möglicher Aufbau des Koprozessors für den lückenlosen Score.

Jeder der acht Speichercontroller (MC0-7)[6] greift auf genau eine solche Spalte zu. Der Einsatz der zwei Speicherports je MC verdoppelt virtuell die Anzahl an MCs; mit zwei Lesezugriffsumordnungseinheiten[7] wird auf diese Ports lesend zugegriffen. Jede Application Engine (AE0-3) erhält die gleiche Hardware-Beschreibung; Parallelisierung erfolgt dementsprechend über mehrfache gleichzeitige Anfragen. Je AE sind 16 ROQs untergebracht, um alle Ports zu nutzen und die Datenparallelität größtmöglich auszunutzen. Über die MCs werden die Referenz und die Profile der Anfrage eingelesen genau wie in der SSE-Implementierung. An jede ROQ sind aufgrund der Datenbreite 8 Einheiten mit je 8 Bit Datenbreite angeschlossen, um die 64 Bit Daten je Port gleichzeitig zu verarbeiten. Diese Einheiten implementieren die Operation S(i,j) = func(S(i-1,j-1)) (s. Abb. 4.5) und müssen entsprechend das berechnete Datum der Vorgängereinheit aus der letzten Iteration erhalten. Die produzierten Werte werden im Gegensatz zur SSE-Implementierung nicht nur fortlaufend und sequentiell, sondern gleichzeitig und baumartig zur Berechnung des globalen Maximums reduziert. Das Maximum wird nach vollständiger Durchführung des Abgleichs der Anfrage mit einer Referenz über ein Koprozessorregister zurückgeben. Der Transport der Eingabedaten vom Hostspeicher zum Koprozessorspeicher erfolgt nicht überlappt zu den Berechnungen, weil dazu je Übertragung die individuelle Größe und damit einhergehend die beste Übertragungsvariante zu ermitteln wäre, wobei die Kommunikation bei größeren Datenmengen wesentlich effizienter[8] erfolgt (vgl. Abschnitt 2.1.7).

Der zeitliche Ablauf ist bei diesem ersten Ansatz wie folgt: Der Koprozessor wird mittels eines Opcodes vom Application Engine Hub gestartet, die Speicheradressen von Anfrage- und Referenzsequenz sowie die jeweiligen Längen werden dazu in Registern übergeben. Die Referenzsequenz wird dann in einen internen Speicher eingelesen. Anschließend wird ein Profil der Anfrage anhand des ersten Werts in der Referenz eingelesen. Ab Erhalt des ersten Datums wird auf dem Workspace der S-Wert in jeder Verarbeitungseinheit berechnet und der Maximumsreduktionsschaltung übergeben.

[6] engl. Memory Controllers (MCs)
[7] engl. Read-Order Queues (ROQs)
[8] mit weniger Verzögerung pro Datum und somit größerem Durchsatz

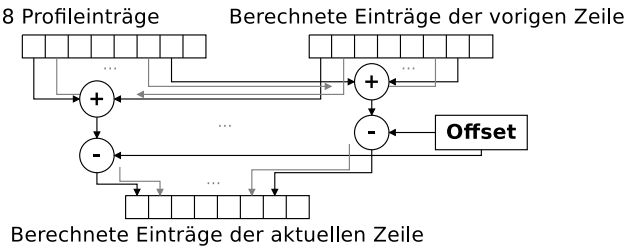


Abbildung 4.5: Intern parallele Verarbeitungseinheit (VE).

Währenddessen werden weitere Werte der aktuellen oder nächsten Anfrageprofilzeile eingelesen.

Der von Convey gestellte Compiler (cnycc), welcher die *Single-Precision Vector Personality* unterstützt und den Ansteuerungscode des Koprozessors übersetzen muss, basiert auf dem Open64-Compiler [57] als Frontend. Als Backend wird der GNU C Compiler (GCC) verwendet. Prinzipiell sollte sich jeder reguläre Code anstelle mit dem Intel C Compiler (ICC) oder GCC auch mit dem cnycc übersetzen lassen. Mit dem cnycc könnte später für geeignete Code-Regionen die Vektor-Personality genutzt werden. Manche SSE-Befehle wurden jedoch nicht akzeptiert beim Kompilieren von HHblits mit cnycc. Die nahtlose Nutzung von OpenMP war ebensowenig möglich. Daher wird einzig der Teil zur Ansteuerung des Koprozessors als separate Datei mit cnycc übersetzt und das erzeugte Objekt gegen die übrigen mit GCC übersetzten Dateien gelinkt. Hier kann für Anwendungsentwickler bereits eine entscheidende Hürde liegen, wenn die Entwicklungswerkzeuge des Zielsystems nicht naht- und problemlos auf die zu portierende Anwendung appliziert werden können.

Die Sequenzen der Datenbank werden sowohl zur Berechnung des lückenlosen Scores benötigt als auch für weitere Teile von HHblits. Daher werden diese Daten von HHblits zunächst von der Festplatte gelesen und in den Hauptspeicher geschrieben, müssen aber zusätzlich vorab der Koprozessorausführung auch noch auf selbigen kopiert werden, um nicht unter der vergleichsweise langsamen Auflösung von Seitenzugriffsfehlern auf dem Koprozessor zu leiden. Als schnellste Möglichkeit hat sich dazu die Verwendung der von Convey bereitgestellten Kopierroutine herausgestellt (vgl. Abb. 2.12). Da der Binary-Interleave-Modus der Convey HC-1 verwendet werden soll für mehr erzielbare Leistung, müssen die Zeilen des Profils eine Länge von 512 Byte aufweisen und mussten entsprechend dort verlängert werden, wo sie kürzer waren.

Die endgültige Umsetzung [BN13] in VHDL auf Basis des obigen Entwurfs ist in Abb. 4.6 illustriert. An jeden Speichercontroller wurde zunächst eine, dann im Zuge verbesserter Ressourcenausnutzung zwei Datenumordnungseinheiten angeschlossen, die jeweils eine auf 8 Byte Daten arbeitende Verarbeitungseinheit bedienen, so dass insgesamt 128 Elemente der Matrix gleichzeitig berechnet werden. Dies ist möglich, da die Zeilen der zwei Spalten des Arbeitsbereichs nur jeweils vom linken diagonalen Nachbarn abhängen und daher auch 16 Zeilen (oder mehr) gleichzeitig berechnet werden können. Die Verarbeitungseinheiten berechnen in Sättigungsarithmetik innerhalb eines Takts die Addition und Subtraktion für jedes der acht Elemente. Das Maximum jeder Verarbeitungseinheit wird in die Maximumsreduktionsschaltung weitergeleitet. Selbige ist baumartig aufgebaut und hat nach sieben Takten das lokale Maximum errechnet und nach einem weiteren das globale Maximum aktualisiert. Aufgrund der Implementierung als Pipeline wird jeden Takt das Maximum aktualisiert, wenn die Pipeline vollständig gefüllt ist, und es sind sieben zusätzliche Takte bis zum vollständigen Leeren der Pipeline nötig, bis das echte globale Maximum erhalten wird.

Der Ressourcenverbrauch ist mäßig, wie man aus Tabelle 4.1 für das vollständige Design mit acht sowie 16 Verarbeitungseinheiten und rechnerisch für eine einzelne Einheit erkennen kann. Zur Evaluation wurden die Laufzeiten der FPGA-Implementierung des Kernels für die Berechnung des lückenlosen Scores von HHblits gemessen und gemittelt. Die in Tabelle 4.2 aufgeführten Ergebnisse zeigen, dass die erhoffte Beschleunigung von Faktor 10 noch nicht erreicht wird, selbst wenn die SSE-Implementierung fairerweise ebenso auf 512 Bytes langen Profilzeilen rechnet („angepasst“).

Aus diesem Grund müssen die Ursachen für die unerwartet geringe Beschleunigung genauer ermittelt werden. Eine mögliche Ursache ist ein unzureichendes Verhältnis von Berechnung zu Kommunikation, zumal beide noch nicht überlappt zueinander folgen. Zur genaueren Ursachenermittlung wurden die unterschiedlichen Sequenzen in Klassen nach ihrer Länge in Blöcken eingeteilt. Wie

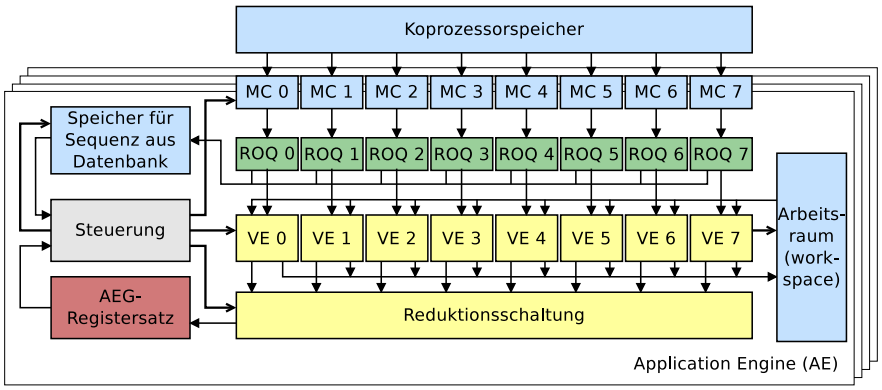


Abbildung 4.6: Erster Ansatz einer parallelen Koprozessorarchitektur (hier mit acht Verarbeitungseinheiten (VEn)) zum Berechnen eines lückenlosen Scores zwischen dem Profil und einer Sequenz (nach [BN13]).

Ressource	**mit 8 VEn**	**mit 16 VEn**	**Durchschnitt 1 VE**
Slices	21406 / 41 %	25701 / 49 %	537 / 1 %
Register	52478 / 25 %	68002 / 32 %	1940 / 0,875 %
LUTs	53133 / 25 %	70432 / 33 %	2162 / 1 %
BRAMs	58 / 20 %	74 / 25 %	2 / 0,625 %
Max. Freq.	201,014 MHz	198,196 MHz	

Tabelle 4.1: Ressourcenbedarf der FPGA-Implementierung für eine Application Engine auf der Convey HC-1 mit je acht bzw. 16 Verarbeitungseinheiten (VEn) und durchschnittlicher Bedarf.

Sequentiell		**SSE**		**Convey HC-1**	
Original	Angepasst	Original	Angepasst	1 AE	4 AEs
512,7 μs	607,2 μs	16,1 μs	18,9 μs	31,8 μs	11,8 μs

Tabelle 4.2: Durchschnittliche Ausführungszeit des Kernels über die gesamte Datenbank uniprot20. („Angepasst“: 512 Byte lange Profilzeile statt „Original“ mit nur 256 Byte langer Profilzeile)

Abb. 4.7 erkennbar macht, sind einerseits sehr viele Sequenzen nur sehr kurz, und andererseits ist die FPGA-Portierung erst ab Sequenzlängen ab 3500 Bytes schneller als die SSE-Version. Betrachtet man nur diese langen Sequenzen und mittelt darüber die durschnittliche Ausführungszeit eines Sequenzvergleichs in Tabelle 4.3, so erhält man bereits eine Beschleunigung von mehr als Faktor 4. Dennoch führt die bloße Verarbeitung langer Sequenzen zu nur geringfügiger Beschleunigung der gesamten Anwendung (Tabelle 4.4). Aus diesen Untersuchungen muss man folgern, dass der Datentransfer vom Host zum Koprozessorspeicher einserseits und andererseits vom Koprozessorspeicher zu den Application Engines (AEs) optimiert bzw. verringert werden muss. Desweiteren bieten die FPGAs Platz für mehrere gleichzeitige Sequenzvergleiche, wie es bereits mit mehreren AEs erfolgt.

Zweiter Ansatz mit Beachtung der Datenlokalität und -wiederverwendung. Die vom ersten Ansatz erzielbare Leistung ist als unzureichend zu bewerten aufgrund der in anderen Arbeiten erzielten weit besseren Ergebnisse und aufgrund der Diskrepanz zu den bei der Analyse ermittelten Möglichkeiten. Die Bandbreite muss mehr ausgenutzt werden, indem alle Speichercontroller je Takt neue Werte anfordern bzw. erhalten [NBSK13]. Ferner wurden nur die zumeist eher kurzen Sequenzen vorgeladen. Lädt man hingegen das Anfrageprofil vor, so hat dies eine Länge von 110 KBytes gegenüber wenigen Hundert Bytes der Sequenzen und kann für alle Datenbankeinträge wiederverwendet werden. Weiterhin sollte jeder Speichercontroller mit zwei Umordnungseinheiten auch zwei

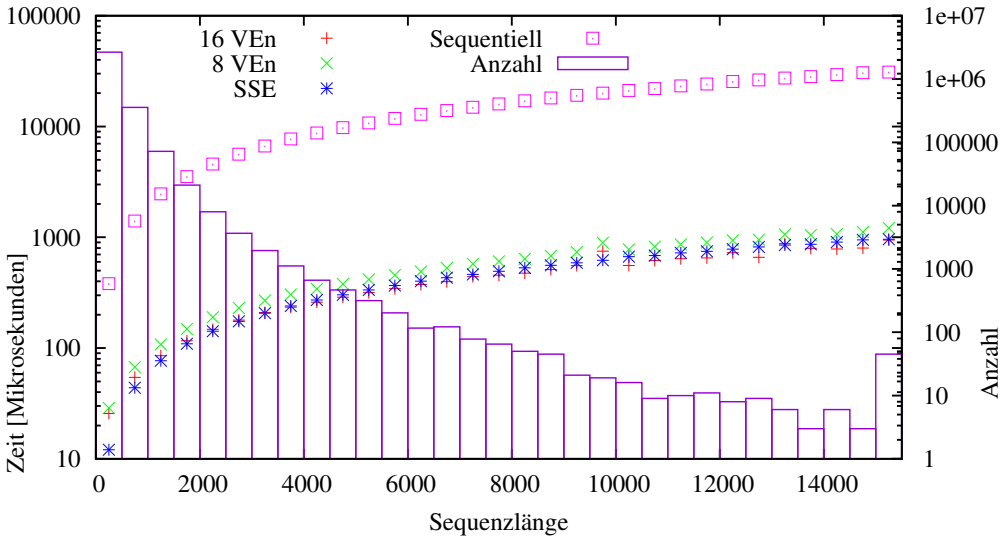


Abbildung 4.7: Durchschnittliche Ausführungszeit des Kernels auf einer AE in Abhängigkeit der Sequenzlängen.

Sequentiell	**SSE**	**Convey HC-1** 1 AE	4 AEs
10731,9 μs	371 μs	311,9 μs	91,3 μs

Tabelle 4.3: Durchschnittliche Ausführungszeit des Kernels bei Sequenzlängen ab 3500 Bytes.

	Sequentiell	**SSE**	**Convey HC-1** 1 AE	4 AEs
Vollständige Datenbank	2343,02 s	97,0 s	155,8 s	91,0 s
Nur lange Sequenzen	64,7 s	22,4 s	22,0 s	21,7 s

Tabelle 4.4: Berechnungszeit von HHblits, kompiliert mit cnyCC und -O3-Flag.

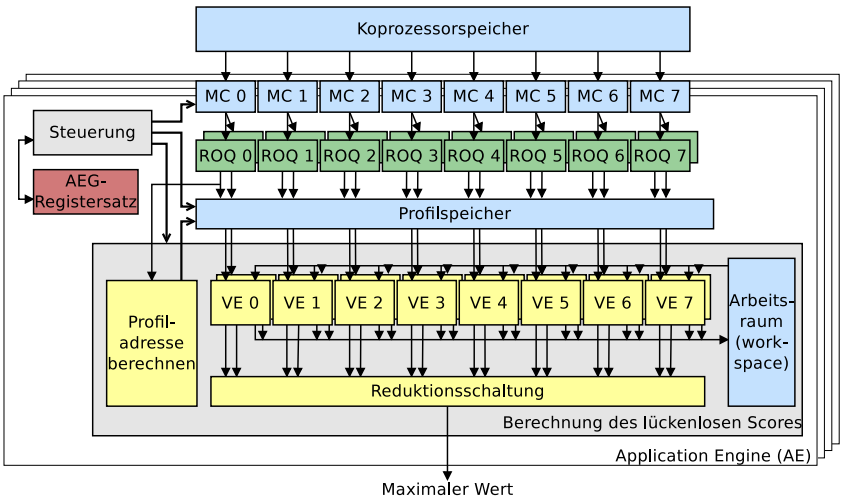


Abbildung 4.8: Zweiter Koprozessoransatz zur Berechnung lückenloser Scores auf 16 Verarbeitungseinheiten (VEn) nach [NBSK13].

Ressource	Anzahl	Anteil
Slices	25.701	49 %
Register	68.002	32 %
LUTs	70.432	33 %
BRAMs	74	25 %
Max. Freq. (Synthese)	5,046 ns	198,196 MHz
Max. Freq. (P&R)	6,658 ns	150,20 MHz

Tabelle 4.5: Ressourcenbedarf des Koprozessors mit einem einzelnen Exemplar zur Berechnung des lückenlosen Scores.

2 Einheiten		3 Einheiten	
42.575 /	82 %	46.098 /	88 %
124.691 /	60 %	140.951 /	67 %
112.835 /	54 %	126.745 /	61 %
138 /	47 %	170 /	59 %
4,801 ns /	208,29 MHz	4,801 ns /	208,29 MHz
6,644 ns /	150,51 MHz	6,666 ns /	150,02 MHz

Tabelle 4.6: Ressourcenbedarf des Koprozessors mit zwei und drei Exemplaren.

Sequenzen zeitgleich einlesen. Dazu ist zwar die Verwendung eines Kreuzschienenverteilers[9] nötig, der einige Hardwareressourcen benötigt, dafür kann jeder Speichercontroller aber Sequenzen von jeder Speicherstelle einlesen.

Die zum zweiten Ansatz zugehörige Implementierung [NBSK13] ist in Abb. 4.8 skizziert. Die Profildaten werden 16-fach parallel eingelesen, indem jede ROQ ihre 32 Bytes des mit einer weiteren ROQ gemeinsam verwendeten Speicherbereichs von 64 Bytes einliest. Die Einheit zur Berechnung des lückenlosen Scores berechnet die Zeilen 5-15 von Algorithmus 1. Die wie vorab implementierten, intern achtfach parallelen Verarbeitungseinheiten (VEn) berechnen den eigentlichen Kern (Zeilen 9-10). Die Reduktionsschaltung ist gegenüber dem ersten Ansatz unverändert. Bandbreite und Zugriffszeit stellen nun kein Problem mehr dar, da für jedes Zeichen der Referenz (1 Byte) in Zeile 5 die entsprechende 512 Zeichen lange Zeile des Profils verarbeitet werden muss (Zeile 6), wobei jede der 16 VEn 32 Elemente (Zeile 9) in vier Iterationen verarbeitet, je acht Berechnungen parallel. Die von den VEn verwendeten Daten entsprechen genau den von den zugehörigen ROQs eingelesenen Daten. Die Speicher in FPGA-Schaltungen haben zumeist nur ein oder zwei Ports und daher kann nicht von mehreren Einheiten aus auf sie zugegriffen werden. Da jede ROQ jedoch in Unterbereiche des Profilspeichers schreibt, die nur von der zugehörigen Verarbeitungseinheit gelesen werden, wurde der gemeinsame Profilspeicher aufgeteilt und auch dieses Problem gelöst. Somit können 16 ROQs gleichzeitig den Profilspeicher befüllen.

In Tabelle 4.5 sind die Implementierungsergebnisse aufgeführt. Aufgrund der fehlenden Möglichkeit zum Zurücksetzen von BRAMs wurden zunächst normale Slices für viele Speicher verwendet, wodurch der Bedarf an Slices recht hoch ist. 66 der verwendeten 74 BRAMs benötigen die ROQs und die Crossbar. Das auf Korrektheit überprüfte Design und die Ansteuerungssoftware wurden anschließend dahingehend verändert, dass die Speicher nicht mehr zurückgesetzt werden müssen, wodurch sich insgesamt drei Einheiten für drei unterschiedliche Sequenz-Profil-Vergleiche pro AE unterbringen lassen (Tabelle 4.6). Das Profil wird nun in 32 BRAMs gespeichert. Zum Einlesen der Zeichen der Referenzsequenz benötigt jede der drei Einheiten nur jeweils eine ROQ. Das Design könnte somit auf den größeren Virtex-6-FPGAs der ex-Serien weiter skaliert werden hin zu 4 bis 16 Einheiten, da das koprozessorseitige Datentransferproblem durch Umstellung des Algorithmus bzw. der Implementierung gelöst wurde. Desweiteren können die Profilspeicher für zwei Einheiten gemeinsam genutzt werden, wodurch der verfügbare Speicher auf dem FPGA optimal ausgenutzt würde.

Es liegt noch weiteres Optimierungspotential darin, den Koprozessor nur genau einmal zu starten, damit dieser sämtliche Datenbanksequenzen selbständig vergleiche, oder aber zumindest mehrere Sequenzen hintereinander berechnen zu lassen und zeitgleich erst die Eingabedaten für die nächsten Aufrufe zu übertragen. Abbildung 4.9 zeigt für die ersten neun Sequenzen aus uniprot20, dass die Berechnung weitaus länger dauert als das Kopieren der nachfolgenden Sequenz. Dieser Schritt ist also im Allgemeinen durchaus gangbar, leidet im Besonderen allerdings an der fehlenden Sortierung der Sequenzen der Länge nach, so dass dies nicht für alle Sequenzen gleichermaßen ausgenutzt werden kann.

Im ersten Ansatz wird nur ein Thread für die Ausführung auf der Convey HC-1 genutzt. Durch die Angabe, HHblits mit zwei Threads auszuführen, wird für den Viterbi-Algorithmus ein weiterer Thread auf dem Zweikern-Prozessor ausgeführt. So wird die Anwendung zusätzlich beschleunigt.

[9] engl. Crossbar

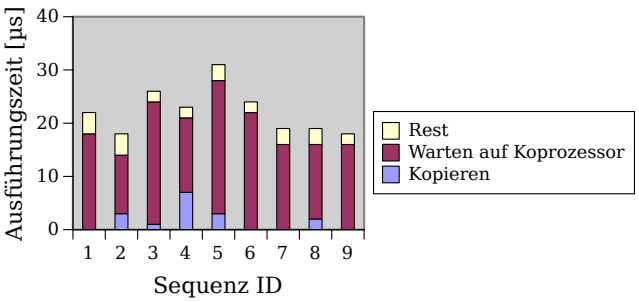


Abbildung 4.9: Überlappung von Datentransfer und Ausführung auf dem Koprozessor.

Hardware	**CPU**	**CPU**		**Convey HC-1**	
Implementierung	**sequentiell**	**SSE**		**3 Einheiten**	
Kerne/AEs	1	1	2	1	4
Erster Vorfilterungsschritt	1601,9 s	50,0 s	25,1 s	28,6 s	13,1 s
Anwendung	1626,2 s	74,1 s	44,5 s	56,8 s	41,3 s

Tabelle 4.7: Durchschnittliche Laufzeit über 30 Läufe für den ersten Vorfilterungsschritt und für die gesamte Anwendung HHblits, jeweils mit der gesamten Datenbank uniprot20.

Die Laufzeit der unterstützten Vorfilterungsstufe und der gesamten Anwendung wurde jeweils sequentiell, mit ein und zwei Threads mit SSE und mit dem Koprozessor mit drei Einheiten, zwei Softwarethreads und einer bzw. 4 AEs gemessen. Tabelle 4.7 zeigt, dass Unterstützung des Verfahrens durch SSE oder durch den Koprozessor wirklich nötig ist. Der Koprozessor ist sogar $\frac{25,0}{13,1} = 1,91$ mal schneller bei der Vorfilterung. Da der Koprozessor auf 512 Byte langen Profilen arbeitet, erhöht sich die zusätzliche Ausführungszeit für die weiteren Routinen jedoch auch etwas. Dennoch ist der Koprozessor auch für die gesamte Anwendung etwas schneller als die SSE-Implementierung mit zwei Threads und erzielt eine geringfügige Beschleunigung von $\frac{44,5}{41,3} = 1,08$.

Damit ist es mit diesem Ansatz gelungen, die Berechnung eines lückenlosen Scores so an die Hardware anzupassen, dass die Durchführung allein durch die Rechenkapazität und die maximale Anzahl an untergebrachten Einheiten beschränkt ist. Gegenüber der Ausführung in reiner Software ist die Ausführung sogar 39,39 mal schneller.

Teilt man die Sequenzen nach ihrer Länge in 31 Klassen ein, so kann Beschleunigung bereits ab Sequenzlängen von 500 Bytes erreicht werden mit nur einer Einheit zur Berechnung des lückenlosen Scores auf nur einer AE, wie in Abb. 4.10 illustriert.

Bei genauerer Betrachtung der Ausführungszeiten skaliert SSE von einem auf zwei Threads mit 1,995 von 27,47 s (2662819 Sequenzen) auf 13,77 s im Bereich der Sequenzen bis zu 500 Zeichen. Der Koprozessor mit einer Einheit auf einer AE benötigt für diesen Bereich 46,45 s, skaliert mit Faktor 2,18 bis 2,35 auf bestenfalls 21,32 s für vier AEs und mit Faktor 2,06 von einer Einheit auf drei Einheiten, wodurch die Ausführungszeit nur 10,33 s betragen würde.

Um sich darauf einzurichten, müssten die Sequenzen ihrer Länge nach vorsortiert sein, damit ohne Zusatzaufwand je Koprozessoraufruf 4*3 Sequenzen mit ähnlicher Länge eingelesen und verarbeitet werden könnten, denn die längste der zwölf gleichzeitig verarbeiteten Sequenzen bestimmt die Ausführungszeit eines Koprozessoraufrufs. Dadurch ließe sich nach ersten Messungen zusätzlich Beschleunigung von $1,17$ erzielen gegenüber der unausgeglichenen Berechnung von 12 gemischt langen Sequenzvergleichen gleichzeitig. Sieht man von dem zusätzlichen Sortieraufwand ab, bedeutet dies bereits Beschleunigung von $4,46$ insgesamt gegenüber der Ausführung mit SSE auf einem Kern. Idealerweise wird die Datenbank also gleich vorsortiert ausgeliefert.

4.2.4 Zusammenfassung der Beschleunigung von HHBlits auf der Convey HC-1

In diesem Abschnitt wurde die Nutzung der Convey HC-1 als heterogenes System mit anwendungsspezifischen Koprozessor beschrieben. Für die HC-1 wurde ein FPGA-Design auf Funktionsebene

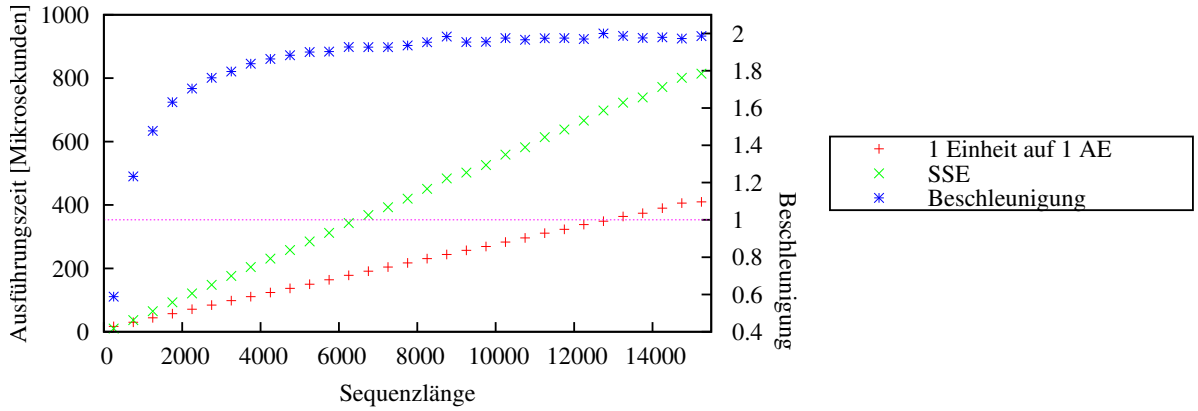


Abbildung 4.10: Kernel-Laufzeit der SSE-Implementierung mit einem Thread und einer Einheit auf einer AE des Koprozessors und die erzielte Beschleunigung für verschiedene Sequenzlängen.

für die Berechnung des lückenlosen Scores entwickelt. Nach dem Betrachten der zu beschleunigenden Anwendung hinsichtlich beschleunigenswerter Kandidaten wurde das Potential solcher Kandidaten analysiert, um eine erste Abschätzung der maximal erzielbaren Beschleunigung zu erhalten. Ausgehend von den Erkenntnissen aus der Analyse wurde ein ersten Ansatz für einen Koprozessor in rekonfigurierbarer Logik entworfen und dabei der große Parameterraum bei der Nutzung der Convey HC-1 erheblich eingeschränkt. Entsprechend dem V-Modell wurden die einzelnen Teile **hardwarenah von einem Hardwarespezialisten implementiert** [Bro13], validiert und evaluiert, um dann eine Designänderung auf höherer Ebene vorzunehmen und einige Parameter anders zu setzen.

Die Einheiten zur Durchführung der Bewertung sind die intern achtfach parallelen Verarbeitungseinheiten, die eine Addition zusammmen mit einer datenabhängigen Subtraktion in einem Takt berechnen. Die Ergebnisse werden einer Reduktionsschaltung übergeben, welche ihr reduziertes Ergebnis in ein Register weitergibt. Somit wurde eine Spezialschaltung **anstatt auf niedrigebener Befehlsebene auf Funktionsebene** entwickelt, die Granularität also hoch gehalten. Insgesamt werden zwölf Bewertungen unterschiedlicher Sequenzen gleichzeitig ausgeführt. Die Bewertungen sind keine einzelnen Operationen auf Befehlsebene, sondern unterstützen in Form eines datenparallel ausgeführten Konglomerats die Anwendung. Das Prinzip der **massiv parallelen Ausführung** wurde also umgesetzt.

Hoher Nutzen des Koprozessors ergibt sich erst dann, wenn Daten sehr viel wiederverwendet werden, da für die geringen Datenmengen die Speichercontroller mit ihren hohen Latenzen ineffizient sind. Im zweiten Ansatz wird dies gezielt angegangen, indem das Profil intern gespeichert wird zur **Ausnutzung von Datenlokalität und -wiederverwendung**, wodurch die Berechnung nicht mehr speichergebunden ist. Daher kann zudem die Bandbreite maximal ausgenutzt werden im Rahmen der maximal möglichen Verarbeitungseinheiten.

Unter Beachtung der in Kapiteln 2 und 3 aufgestellten Forderungen lässt sich also Nutzen aus rekonfigurierbarer Hardware und heterogenen Systemen erzielen. Weiterer Nutzen könnte erzielbar sein, wenn die Übertragung der Arbeitsdaten überlappt mit der Verarbeitung erfolgte, wozu allerdings die Systemaufrufe auf der Convey HC-1 bislang nicht die benötigte Leistung liefern, so dass eine einzelne, gesamte Kopieroperation noch am effizientesten[8] ist.

4.3 Zusammenfassung zur Verlagerung von Anwendungen in rekonfigurierbare Hardware

Für bioinformatische Anwendungen werden in der Literatur Geschwindigkeitssteigerungen von ein bis zwei Größenordnungen durch Verlagerung in rekonfigurierbare Hardware berichtet.

In diesem Kapitel wurde am Beispiel der Anwendung HHblits beschrieben, welches Vorgehen bei der Hardwareentwicklung für rekonfigurierbare Akzeleratoren nötig ist. Indem der erste Vorfilterungsschritt mehr als 122-fach durch die Verlagerung in Hardware beschleunigt wird, ist die Anwendung HHblits etwa 39-fach gegenüber sequentieller Ausführung beschleunigbar. Insgesamt ist die Hardware sogar schneller als die Implementierung mit SSE nach Farrar. Dazu ist es nötig, den Algorithmus gegenüber der Softwareimplementierung etwas verändert zu implementieren. Das Ergebnis ist eine datengetriebene, als Pipeline arbeitende Spezialschaltung für die Bestimmung eines maximalen lückenlosen Übereinstimmungswerts.

Mit dieser Studie der Unterstützung einer gängigen Anwendung durch den FPGA-basierten Koprozessor der Convey HC-1 sind bereits folgende Erkenntnisse gewonnen:

- Software ist nicht so portabel, wie dies für die Portierung in Hardwareschaltungen wünschenswert ist. Deshalb sind die softwarebasierten Werkzeuge für Anwendungsentwickler nicht ohne Weiteres einsetzbar.
- Die verfügbare Bandbreite zum Koprozessor bzw. zu einem FPGA muss maximal ausgenutzt werden. Die Art und Größe der Daten haben dabei großen Einfluss. Die Übertragung kleiner Datenmengen ist nicht lohnenswert, so auch nicht die kurzen Sequenzen bei HHblits.
- Wesentlich ist die geschickte Übertragung der Daten und Vorhaltung zur Wiederverwendung. Die Nutzung muss datengetrieben erfolgen.
- Zur Entwicklung einer Hardwareunterstützung muss ein Algorithmus, gerade im Hinblick auf die Datenverwendung, eventuell umgestellt werden, um die Datenlokalität zu erhöhen. Dies bedarf unter Umständen der Interaktion von Anwendungsentwickler und Hardwareentwickler.
- Einige wenige Spezialkomponenten machen die Umsetzung und die Leistungsfähigkeit aus, bspw. das Zusammenlegen mehrerer voneinander abhängiger Operationen in einen Takt, um die Länge der entstehenden Pipeline zu kürzen, was bei geringen Arbeitsdaten besonders hohe Auswirkungen hat.

Diese Erkenntnisse lassen sich folgendermaßen ausnutzen:

- Es bietet sich an, bereits entwickelte Portierungen eines Verfahrens in Hardware vorzuhalten, um sie zu gegebenem Zeitpunkt wiederverwenden zu können.
- Möglichst viele Einheiten (im Rahmen der Möglichkeiten eines erfolgreichen Routings und Platzierung) werden helfen, die Bandbreite maximal auszunutzen. Bei geringen Datenmengen sollten Koprozessoren außer acht gelassen werden.
- Streaming von Daten zwischen erzeugenden und verbrauchenden Einheiten, auch über die Grenze zwischen Software und Hardware hinweg, bringt die Datenwiederverwendung sowie die datengetriebene Ausführung und ist daher einzusetzen.
- Hardwareentwickler stellen idealerweise die Hardwareimplementierungen bereit und darüberhinaus den Anwendungsentwicklern eine geeignete Schnittstelle zur Verfügung, mittels welcher letztgenannte aufgrund ihrer Domänenkenntnisse die Hardware optimal nutzen können.
- Nicht nur die massiv parallele Ausführung von Algorithmen sollte mittels FPGAs bei der Hardwareverlagerung angegangen werden, sondern die Implementierung von Spezialoperationen, die mehrere Operationen pro Takt ausführen oder auf anderen Datentypen arbeiten.

Im folgenden Kapitel werden unterschiedliche Möglichkeiten zur Beschleunigung numerischer Anwendungen auf heterogenen FPGA-basierenden Mehrkernsystemen betrachtet. Hervorhebenswert sind dabei Spezialoperationen wie exakte Skalarprodukte auf Befehlsebene, Stencil-Operationen auf Funktionsebene, vollständige Verfahren auf Funktions- oder Anwendungsebene. Der bereits aufgeführte Komponentengedanke kommt dabei ebenso zum Tragen, da der Komponentenansatz ermöglicht, auf Basis feingranularer Beschleuniger komplexere Beschleuniger zu erhalten, die leistungsfähiger sind als die voneinander unabhängige Verwendung einzelner feingranularer Beschleuniger.

Kapitel 5

Ansätze zur Beschleunigung numerischer Anwendungen mit Mehrkernsystemen und FPGAs

Im vorherigen Kapitel wurde stellvertretend an einer bioinformatischen Anwendung beschrieben, welche Schritte und Umformulierungen eines Algorithmus nötig sind, um echten Nutzen für den Anwender aus einem rekonfiguierbaren Koprozessor herauszuholen. In diesem Kapitel werden hingegen mehrere Ansätze untersucht, wie im Bereich numerischer Anwendungen für unterschiedliche Granularitäten, mit unterschiedlichen Werkzeugen und Techniken Mehrwert aus Mehrkernsystemen mit rekonfigurierbaren Akzeleratoren erhaltbar ist. Bei numerischen Anwendungen kann Unterstützung durch rekonfigurierbare Hardware beispielsweise mit Spezialschaltungen für Erweiterungen im Bereich des Festkomma- und Gleitkommarechnens oder für weitere Arithmetiken erfolgen. Daher wurden weitere Spezialoperationen untersucht, die so in Standardhardware nicht vorhanden sind und deren Softwareemulation auch unter Verwendung mehrerer Rechenkerne sehr zeit- und ressourcenaufwendig ist.

Zunächst werden die für die eigentlichen Untersuchungen durchgeführten Vorarbeiten hinsichtlich einer Matrixbibliothek (Abschnitt 5.1.1) und des Lanczos-Verfahrens (Abschnitte 5.1.2 und 5.1.3) beschrieben. Die Matrixbibliothek stellt unterschiedliche Funktionalitäten mit unterschiedlich hohen Granularitäten zur Verlagerung in Hardware zur Verfügung. Anhand der Genauigkeitsanforderungen des Lanczos-Verfahrens motiviert sich der Einsatz von Spezialoperationen. Als mögliche Spezialoperationen wird zuerst Intervallarithmetik in Abschnitt 5.2 betrachtet, deren Ausführung häufig zwei gleichzeitig zu berechnenden Datenpfade für jeweils das untere und das obere Ende eines Intervalls enthält. Mittels datenparalleler Ausführung auf mehreren Kernen eines Prozessors kann keine nennenswerte Beschleunigung der Intervallarithmetik erhalten werden; die gewählte Granularitätsebene ist zu niedrig. Wie sich mittels einer intern genaueren Einheit zur Berechnung eines Skalarprodukts in Hardware in Abschnitt 5.3 zeigt, ist eine Alternative zur Implementierung von Spezialoperationen die Spezialimplementierung von Standardoperationen wie Multiply-Accumulate (MAC)-Operationen, Skalarprodukten oder Matrixmultiplikationen. Dies motiviert eine Implementierung eines exakten Skalarprodukts in rekonfigurierbarer Hardware und die dabei aufgekommenen Probleme in Abschnitt 5.4. Mittels einer alternativen, hardwarenäheren Implementierung mit Pipelining und ausgeglicheneren Stufen könnte man grobgranularere Operationen wie Matrixmultiplikationen schneller exakter ausführen als mit Softwareimplementierungen für exakte Arithmetik. Diese Analyse wird in Abschnitt 5.5 durchgeführt. Wie aus den Vorarbeiten ersichtlich wurde, ist die Matrixmultiplikation allein nicht ausreichend dafür, beispielsweise das Lanczos-Verfahren hinreichend zu unterstützen hinsichtlich der Genauigkeitsanforderungen. Daher wird in dieser Dissertation eine auf Mikroprogrammierung basierende Implementierung einer exakten Arithmetik-Einheit entwickelt, die intern feingranulare Komponenten wie Addierer und Multiplizierer verwendet (Abschnitt 5.6). In Abschnitt 5.7 wird demonstriert, wie diese komponentenbasierende Implementierung der mikroprogrammierten Arithmetik-Einheit für weitere Operationen als das Skalarprodukt genutzt werden kann, um exakt auf den internen Datentypen zu rechnen. Abschließend wird von der anderen Seite der Domänenexperten aus die Verwendung heterogener Systeme beleuchtet. Diesbezüglich werden die Untersuchungen zur Verwendung von Hochsprachen zur Hardwarebeschreibung für numerische Anwendungen in Abschnitt 5.8 am Beispiel des CG-Verfahrens zur Lösung der Poisson-Gleichung gezeigt, wozu Stenciloperationen und darauf aufbauende Vorkonditionierer eingesetzt werden.

5.1 Vorarbeiten

Zur besseren Durchführung der Folgearbeiten mussten einige Vorarbeiten durchgeführt werden. Mittels diser Vorarbeiten lässt sich auch die durchgängige Anwendbarkeit der vorgeschlagenen und evaluierten Konzepte und Implementierungen zeigen. Am wichtigsten ist eine anpassbare Matrixbibliothek, die für die untersuchten numerischen Anwendungen zum Einsatz gekommen ist.

5.1.1 Eine Matrixbibliothek

Es stellt sich unter Leistungsaspekten als nicht ausreichend heraus, einzelne Operationen exakt durchzuführen, sondern wie von Kulisch angedacht [172, 177] muss etwas grobkörnigere Funktionalität wie ein Skalarprodukt angeboten werden, in dessen Rahmen die exakte Arithmetik nahtlos integriert ist. Dazu wird vorab der Weiterentwicklung ein Rahmenwerk in Software benötigt, das die Implementierung von Anwendungen maßgeblich erleichtert, in denen exaktere Arithmetik zum Einsatz kommen kann, und das ebenso die Ausnutzung von Mehrkernsystemen und FPGA-erweiterten Systemen mit dem datengetriebenen Rahmenwerk ermöglicht. Prinzipiell stellen die *Basic Linear Algebra Subroutines* (BLAS) eine Möglichkeit dazu dar. Die Verwendung von BLAS ist allerdings nicht durchweg leicht eingängig. So wird bei der Operation $a \cdot x + y$ (axpy) das Ergebnis in der Variablen y gespeichert, bei Vektoradditionen ist gar die Verwendung der Matrix-Vektor-Multiplikation (GEMV) nötig: $y := \alpha Ax + \beta y$. Dies ist insbesondere dann problematisch, wenn die Einbindung von Streaming auf Plattformen wie DRC Accelium AC2030 unter Zuhilfenahme von Impulse C, den MaxWorkstations mit MaxCompiler oder von zusätzlichen Speichern wie auf der Convey HC-1 angedacht ist, wo die benötigten zusätzliche Kopieroperationen zur zwischenzeitlichen Trennung von Eingaben und Ausgaben bei solchen Operationen besonders nachteilhaft wären. Obendrein ist die Bibliothek in Umfang und Funktionalität sehr groß. Daher kann es insgesamt als nicht zielführend betrachtet werden, die gesamte BLAS-Bibliothek um Schnittstellen oder Implementierungen für C-XSC, GMP, die entwickelten Hardwareimplementierungen und Softwareemulationen zu erweitern. Für die direkte Nutzung von spezieller Hardware stellen Hersteller wie Convey und Nvidia ohnehin hauseigene Implementierungen mit den Convey Mathematical Libraries (CML) und CUBLAS zur Verfügung. Der Fokus dieser Arbeit liegt hingegen auf portablen Lösungen der Funktionalitäten mit entsprechend eher wenigen hardwarespezifischen Anpassungen.

Aus diesem Grund werden Neuimplementierungen der jeweils benötigten Funktionalität geschaffen, die von vornherein für die zu verwendenden Arithmetik-Bibliotheken entwickelt werden. Somit wird auch dem Konzept der Verlagerung von Softwarekomponenten in Hardware Rechnung getragen.

Entwurfsziele

Die folgenden Entwurfsziele standen aufgrund des geschilderten Hintergrundes und weiterer Vorhaben fest.

Modularität: Der modulare Aufbau ermöglicht die Wiederverwendung von Modulen in neuen Modulen sowie natürlich von Anwendungsseite her. Die Funktionalität der Module ist eindeutig.

Erweiterbarkeit: Die Bibliothek ist so zu entwerfen und implementieren, dass das Hinzufügen neuer Module jederzeit gewährleistet ist, ohne dabei die formulierten Entwurfsziele grundlegend zu verletzen. Ferner soll es möglich sein, die Bibliothek um neue interne Möglichkeiten und Merkmale zu erweitern.

Flexibilität: Die Schnittstelle ermöglicht Implementierungen für unterschiedliche Ziele wie bestimmte Hardware oder weitere Softwarebibliotheken.

Änderbarkeit: Abgesehen von der Möglichkeit, eventuell vorhandene Fehler auszubessern, sind beispielsweise Optimierungen wie eine ikj-Multiplikation anstelle einer ijk-Multiplikation leicht vorzunehmen.

Verwendbarkeit: Für einen externen Entwickler, der vielleicht aus einer spezifischen Domäne kommt, ist es ohne große Mühen möglich, die Module und Funktionen aufzurufen und im Rahmen eines größeren Ganzen zu verwenden und dabei auch von besonderer Software oder Hardware zu profitieren.

Evaluation. Im Rahmen gemeinschaftlicher Projekte mit der Shared Research Group „New Frontiers in High Performance Computing Exploiting Multicore and Coprocessor Technology“ wurde die Matrixbibliothek für mehrere Projekte eingesetzt. Die äußerst zügig umgesetzte Implementierung des Lanczos-Verfahrens sowie die hinzugefügten Funktionen wie Givens-Rotation, QR-Zerlegung, Stencilimplementierungen und symmetrische Gauss-Seidel-Vorkonditionierer deuten darauf hin, dass die Entwurfsziele erreicht wurden. Insbesondere die Aspekte der Modularität, Erweiterbarkeit und Verwendbarkeit wurden dadurch gezeigt.

Schnittstelle

Ziel der Matrixbibliothek ist eine klare Programmierschnittstelle, wie sie der Anwender beim Formulieren auf Papier oder mittels funktionaler Hochsprachen wie Matlab automatisch verwendet. Entsprechend wird eher auf zusammengesetzte Operationen wie *axpy* und dergleichen verzichtet.

Die wichtigsten Bestandteile der Schnittstellenspezifikation sind die Festlegung von Datentypen für Skalare, Vektoren und Matrizen sowie eine gewisse Aufrufkonvention. Bei dieser war es wichtig, die Trennung von Eingaben und Ausgaben zu erreichen sowie intuitiv verständliche und leicht zu erkennende Funktionsnamen und Signaturen zu verwenden. Der grundlegende Aufbau ist der folgende:

```
int funktionsname (datatype *eingabe1, ... datatype *eingaben,
                   datatype *ausgabe);
```

wobei der Rückgabewert Information über die Anzahl oder Art an aufgetretenen Fehlern enthält, der Datentyp `datatype` ein Platzhalter für o.g. Datentypen ist, und der Funktionsname gemäß folgendem Muster aufzubauen ist:

funktionsname := <Aktion>_<Involvierter Datentyp>{_<Ggf. weiterer Datentyp>}

Ein Beispiel für die Erfüllung der Vorgaben ist

```
int multiply_matrix_scalar (matrix_t *mat_in, scalar_t *scal_in,
                            matrix_t *mat_out);
```

Das vorab angedachte Zurückgeben der Ergebnismatrix als Pointer über den Rückgabewert erwies sich als nachteilhaft, da bereits bestehende Matrizen nicht für die Ergebnisse genutzt werden konnten, sondern der anwendende Entwickler zusätzliche Matrizen allozieren müsste, sowie weitere Mechanismen zur Fehleranzeige nötig wären. Daher werden der Zeiger auf die Ergebnisdatenstruktur und ihre Inhalte auf Gültigkeit überprüft und gegebenenfalls Speicher in der passenden Größe alloziert. Ist der Zeiger hingegen gültig, so werden die Spalten- und Zeilenzahl mit den Anforderungen der Operationen abgeglichen und die Operation je nach Ausprägung der Implementierung entweder auf dem bereits allozierten Datenbereich durchgeführt oder abgebrochen oder bereits allozierter Matrixspeicher freigegeben und die Matrixstruktur neu initialisiert. So wird immer die Ausführbarkeit der Operationen gewährleistet.

print_matrix	read_from_file	allocate_matrix
clear_matrix	copy_matrix	free_matrix
multiply_matrix	add_matrix	transpose_matrix
change_row_col_order	stencil_2d_matrix	scalar_product
matrix_norm_2	multiply_matrix_scalar	

Tabelle 5.1: Grundlegender Funktionsumfang der Matrixbibliothek.

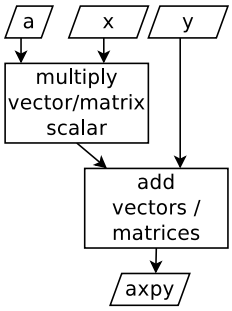


Abbildung 5.1: Implementierung der Funktion axpy als Folge von Skalierung und Addition.

Einbindung von exakter Arithmetik

Zusätzliche arithmetische Hard- und Software ist wie folgt eingebunden. Die Matrix- und Vektortypen verwenden anstelle eines Grunddatentyps wie `float` oder `double` einen Platzhhalter, der entsprechend der zu verwendenden Bibliothek gesetzt wird. In den Implementierungen der Funktionen wird an den entscheidenden Stellen mittels einer Compilerdirektive zwischen den verschiedenen Zielen unterschieden. So verwenden Funktionen mit Skalarprodukten im Fall von C-XSC als Zielbibliothek das exakte Skalarprodukt von C-XSC, während das Standardziel bestehend aus Gleitkommaarithmetik auf dem Host regulär als Schleife über die Eingabeoperanden Multiplikationen und Akkumulationen auf die Zielvariable durchführt.

Eingebunden sind folgende Ziele: exakte Multiplikations-Akkumulationseinheit für einfach genaue Daten auf dem UoH HTX-Board, Emulation selbiger in Software, GNU Multiple Precision Library, C-XSC, einfach genau auf der Host-CPU sowie doppelt genau.

Funktionsumfang

Einen Teil des Funktionsumfangs der Bibliothek gibt Tabelle 5.1 wieder. Die Operation *axpy* ist nicht darinnen enthalten; sie kann als neues Modul formuliert werden wie in Abb. 5.1 oder muss in der Anwendung in zwei Teilen durchgeführt werden. Die Einhaltung der vorgegebenen Nomenklatur gestaltet sich unter Nutzbarkeitsaspekten sehr schwierig. Zu lange Funktionsnamen wurden daher dort vermieden, wo der Datentyp unbedeutend oder selbsterklärend ist. Matrizen können auf zwei unterschiedliche Arten gespeichert werden. Bei „row major“ wird zuerst über die Spalten innerhalb einer Zeile iteriert; bei „column major“ zuerst über die Zeilen innerhalb einer Spalte. Dies ist besonders praktisch bei Matrixmultiplikationen hinsichtlich der cacheeffizienten Speicherung und ermöglicht daher, wesentlich mehr Nutzen durch Streaming zu erzielen.

Anwendung der Bibliothek für exakte arithmetische Operationen

Als Anwendungsfall der Bibliothek wurde die Berechnung eines synthetischen Benchmarks bestehend aus 2^{20} Akkumulationen des Werts v^2 für $v = 2^{-20}$ auf einen mit 2^{20} initialisierten Akkumulator herangezogen. Die Berechnung der Gleitkommaoperationen erfolgt dabei entweder mit der nativen Hardware, oder arithmetisch exakt mit GMP oder C-XSC. Einfache und doppelte Genauigkeit reichen hierbei nicht mehr für ein korrektes Ergebnis aus, da die Differenz der höchsten Stelle von a und der von v^2 $20 - (2 * (-20)) = 60$ beträgt. Zieht man den Initialisierungswert vom Endergebnis wieder ab, so erhält man mit GMP und C-XSC den korrekten Wert 2^{-20}. GMP wurde

	GMP	C-XSC	DG
Laufzeit (Ticks)	566.480.613	640.445.526	6.403.510
Laufzeit (μs)	177.089	200.211	2.001
M eMACs/sec	5,921	5,237	523,810

Tabelle 5.2: Vergleich zwischen GMP mit 256 Bits, C-XSC und doppelter Genauigkeit (DG) für 2^{20} MAC-Operationen: $a = 2^{20} + 2^{-20} \cdot 2^{-20} + \ldots + 2^{-20} \cdot 2^{-20}$.

mit 256 Bits parametriert und C-XSC verwendet ein 4288 Bits breites Festkommaakkumulatorregister. Die Laufzeiten wurden auf einem Intel Pentium 4 mit 3,2 GHz gemessen über 128 Läufe. Alle Programme wurden mit `-O2` übersetzt. Wie Tabelle 5.2 entnehmbar ist, ist die Ausführung von exakterer Arithmetik mit den Bibliotheken GMP und C-XSC etwa 100 mal langsamer als die doppelt genaue Rechnung mit inkorrektem Ergebniswert.

Zusammenfassung der Matrixbibliothek

Die vorgestellte Matrixbibliothek ist ähnlich zu BLAS und anderen Bibliotheken zur Umsetzung numerischer Verfahren geeignet. Die arithmetischen Anforderungen des Lanczos-Verfahrens können jetzt mit einer um exakte Arithmetiken erweiterten Version untersucht werden.

5.1.2 Eigenwertbestimmung mittels des Lanczos-Verfahrens

Der Lanczos-Algorithmus [182] ist eine iterative Methode zur Bestimmung einer orthogonalen Matrix $\mathbf{Q}$, mittels derer eine symmetrische Eingabematrix $\mathbf{A}$ transformiert werden kann zu einer symmetrischen Tridiagonalmatrix $\mathbf{T}$:

$$\mathbf{T} := \mathbf{Q}^{\mathrm{T}} \cdot \mathbf{A} \cdot \mathbf{Q}$$

Eine solche tridiagonale Matrix ermöglicht die einfache Bestimmung der Eigenwerte. Man schreibt die Matrix $\mathbf{T}$ folgendermaßen:

$$\mathbf{T} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \ldots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \beta_{n-1} \\ 0 & \ldots & 0 & \beta_{n-1} & \alpha_n \end{pmatrix}$$

Also müssen die Vektoren α und β bestimmt werden. Diese Vektoren lassen sich erhalten, indem iterativ die Spalten $\mathrm{q_i}$ von $\mathbf{Q}$ bestimmt werden, welche abschließend die α_i und β_i liefern [78, 209]:

$$\alpha_i = \mathrm{q_i^T} \mathbf{A} \mathrm{q_j},$$

und indem die i-te Spalte $\mathbf{A}\mathrm{q_i}$ von $\mathbf{AQ} = \mathbf{QT}$ betrachtet wird:

$$\mathbf{A}\mathrm{q_i} = \beta_{i-1}\mathrm{q_{i-1}} + \alpha_i \mathrm{q_i} + \beta_i \mathrm{q_{i+1}} \qquad | - \beta_{i-1}\mathrm{q_{i-1}} - \alpha_i \mathrm{q_i}. \tag{5.1}$$

Damit lässt sich $\mathrm{r_i} = (\mathbf{A} - \alpha\mathbf{I})\mathrm{q_i} - \beta_{i-1}\mathrm{q_{i-1}} = \beta_i \mathrm{q_{i+1}}$ setzen. β_i kann also anhand von $\mathrm{r_i}$ durch $\beta_i = \frac{\|r_i\|}{\|q_{i+1}\|}$ bestimmt werden, und da die Länge des Vektors $\mathrm{q_i}$ genau 1 ist, ist auch $\beta_i = \|\mathrm{r_i}\|_2$. Die Iterationsvorschrift ist nun gegeben durch $\mathrm{q_{i+1}} = \mathrm{r_i}/\beta_i$.

Innerhalb jeder Iteration wird eine Spalte q_i von $\mathbf{Q}$ berechnet, was auch α_i und β_i liefert.

Gleichung 5.1 kann geschrieben werden als

$$\alpha_i \mathrm{q_i} + \beta_i \mathrm{q_{i+1}} = \mathbf{A}\mathrm{q_i} - \beta_{i-1}\mathrm{q_{i-1}} \qquad | \cdot \mathrm{q_i^T}$$

```
 1 # Initialization
 2 r[0:SIZE] := 1.0 / sqrt(SIZE)
 3 q_0[0:SIZE] := 0.0
 4 beta := 1.0
 5 i := 1
 6 while i<=SIZE and beta!=0:
 7   alpha := q_i_T * A * q_i
 8   r := (A - alpha * I) * q_i - beta * q_i-1
 9   beta := length (r)
10   if beta=0: break
11   q_i-1 := q_i
12   q_i := r / beta
```

Listing 5.1: Initialisierung und Iteration des Lanczos-Algorithmus.

$$\alpha_i = \mathrm{q_i^T}(\mathbf{A}\mathrm{q_i} - \beta_{i-1}\mathrm{q_{i-1}}),$$

denn wegen der Orthogonalität aller Vektoren $\mathrm{q_i}$ zueinander ist $\mathrm{q_i} \bullet \mathrm{q_j} = 0 \ \ \forall i \neq j$.

Damit berechnet man α_i zuerst anhand von $\mathrm{q_{i-1}}$ aus der letzten Iteration, dann erhält man $\mathrm{r_i}$, was dann wiederum nach Normalisierung den neuen Vektor $\mathrm{q_i}$ darstellt. Die Auflistung 5.1 gibt den Algorithmus noch einmal zusammen mit der Initialisierungsphase wieder. Die endgültigen Eigenwerte der Matrix $\mathbf{T}$ können dann über iterative Gleichungslöser wie den QR-Algorithmus bestimmt werden.

5.1.3 Anwendung der Bibliothek zur Bedarfsanalyse arithmetischer Genauigkeit beim Lanczos-Verfahren

Die Anwendbarkeit der Matrixbibliothek zur Umsetzung von numerischen Verfahren wird anhand des in Abschnitt 5.1.2 beschriebenen Lanczos-Verfahrens gezeigt. Ferner wird damit die Einsetzbarkeit unterschiedlicher Arithmetik-Bibliotheken vorab der Integration von Software-Hardware-Schnittstellen zu Koprozessoren oder Rahmenwerken untersucht. Schlussendlich wird auch untersucht, ob es wirklich ausreichend ist, lediglich das Skalarprodukt exakt zu berechnen.

Die Eigenwerte für die Eingabematrix sind exemplarisch verteilt. Die äußeren Werte weichen am sichtbarsten ab und sind daher zum besseren Abgleich mit den von den unterschiedlichen Implementierungen errechneten Werten in Tabelle 5.3 aufgeführt.

Im Gegensatz zu dem in Listing 5.1 geschilderten grundlegenden Algorithmus ist Folgendes geändert: α_i wird zunächst durch $\langle \mathrm{q_i}, \mathbf{A}\mathrm{q_i} \rangle$ berechnet; dann wird r durch Subtrahieren von $\alpha \mathrm{q_i}$ und $\beta \mathrm{q_{i-1}}$ vom bereits bestimmten $\mathbf{A}\mathrm{q_i}$ erhalten; schließlich wird β bestimmt, und q_i ist dann der skalierte Vektor r. Die Schleife bricht ab, wenn $|\beta| < 10^{-8}$ gilt.

Doppelte Genauigkeit

Die mit doppelter Genauigkeit errechneten Eigenwerte sind in Abb. 5.2 dargestellt; die beiden äußersten Eigenwerte werden zweimal gefunden und somit völlig falsche Eigenwerte ermittelt.

GMP

Zur Nutzung der Matrixbibliothek mit GMP muss lediglich die Bibliothek entsprechend parametriert werden, um den GMP-spezifischen Datentyp mit einer Auflösung von in diesem Falle 256 Bits zu verwenden. Die in Abb. 5.3 abgebildeten, mit GMP errechneten Eigenwerte sind identisch mit den spezifizierten Eigenwerten. Die Implementierung mit GMP wird also als korrekt eingestuft.

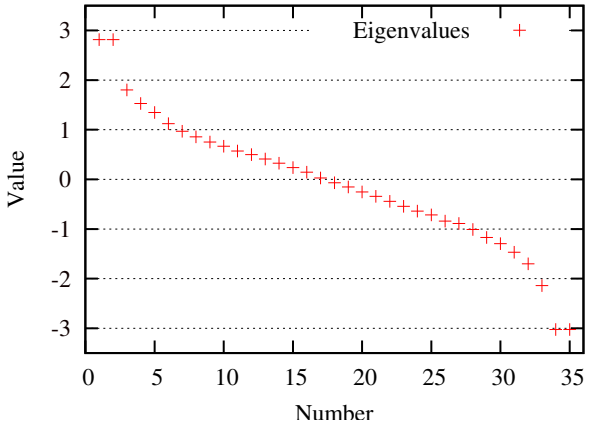


Abbildung 5.2: Errechnete Eigenwerte bei Implementierung mit doppelter Genauigkeit.

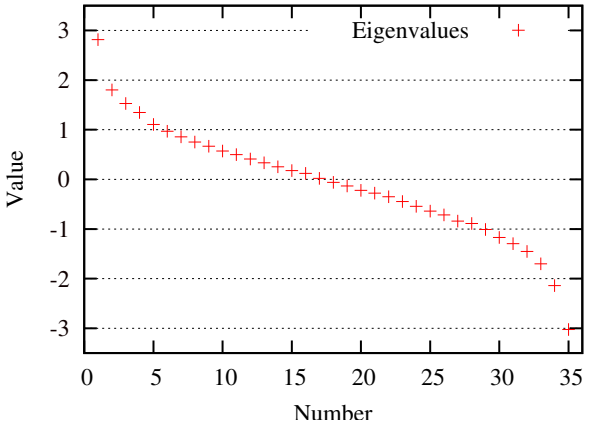


Abbildung 5.3: Errechnete Eigenwerte bei Implementierung mit GMP (256 Bits).

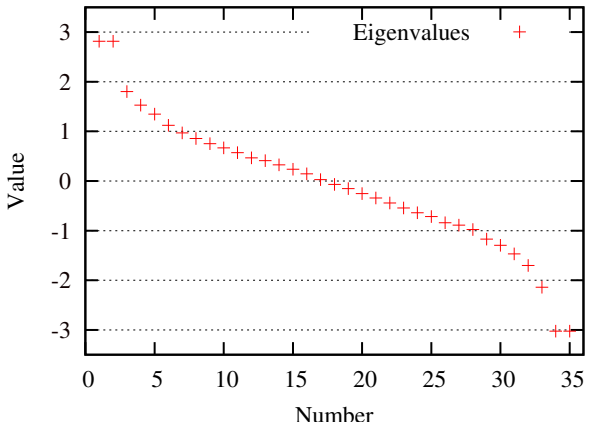


Abbildung 5.4: Errechnete Eigenwerte bei Implementierung mit naiver C-XSC-Implementierung.

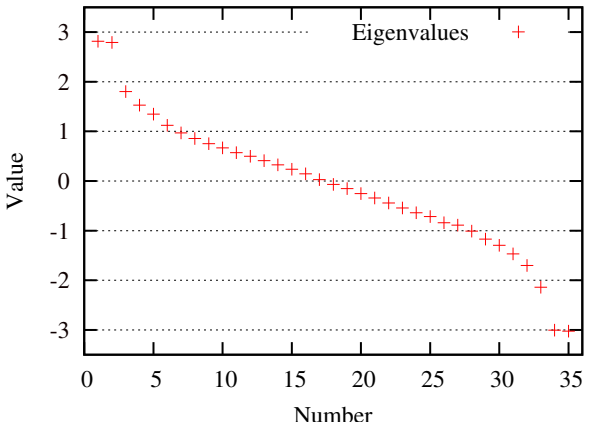


Abbildung 5.5: Errechnete Eigenwerte bei Implementierung mit verbesserter C-XSC-Implementierung.

C-XSC

Die Erweiterung der Matrixbibliothek hingehend zur Unterstützung und Verwendung von C-XSC bedingt auch teilweise die Verwendung von C++. Mit C-XSC kommen auch eigene Matrix- und Vektortypen sowie die wichtige `accumulate()`-Operation.

Der Ansatz mit der ausschließlichen Verlagerung der Skalarprodukte in exakte Arithmetik erzielte die in Abb. 5.4 dargestellten Werte als Ergebnisse für die Eigenwerte. Er lässt erkennbar werden, dass mehr als lediglich das Skalarprodukt genau berechnet werden muss.

Vergleich der Genauigkeit

In Tabelle 5.4 werden die Unterschiede bei den äußersten Eigenwerten ersichtlich und die absoluten und relativen Fehler in Tabelle 5.5 angegeben. Wie bereits angemerkt, genügen doppelt genaue Implementierungen und ein exakt berechnetes Skalarprodukt nicht bei der korrekten Berechnung der inneren 33 Eigenwerte. Die mit GMP erzielten Fehler hingegen sind lediglich auf die nötigen Formatkonvertierungen und das damit verbundene Abschneiden/Runden zurückzuführen.

Nr.	Wert
1	2,814172761275914
2	1,800713681981315
34	-2,141856786835564
35	-3,026665892769740

Tabelle 5.3: Eigenwerte der Eingabematrix.

Nr.	Doppelt genau	GMP	C-XSC
1	2.8141727612720642	2.814172761275883	2.81417276127319
2	2.8141727599325881	1.800713681981300	2.81417214146501
34	-3.0266654324701134	-2.141856786835524	-3.02666588474666
35	-3.0266658927695635	-3.026665892769714	-3.02666589276698

Tabelle 5.4: Doppelt genau, mit GMP oder C-XSC errechnete Eigenwerte.

Nr.	Doppelt genau		GMP		C-XSC	
	Absoluter Fehler	Relativer Fehler	Absoluter Fehler	Relativer Fehler	Absoluter Fehler	Relativer Fehler
1	3,84981e-12	1,36801e-12	3,10862e-14	1,10463e-14	2,72404e-12	9,67973e-13
2	-1,01346	-0,56281	1,48770e-14	8,26172e-15	-1,01346	-0,56281
34	-0,88481	-0,41310	-3,99680e-14	1,86605e-14	0,88481	-0,41310
35	-1,76303e-13	5,82500e-14	-2,57572e-14	8,51008e-15	-2,76001e-12	9,11899e-13

Tabelle 5.5: Absolute und relative Fehler zwischen doppelt-genauer Implementierung, der GMP-Version und der C-XSC-Implementierung.

Fehlersuche und -analyse

Anhand eines einfachen Fehlermodells können die möglichen Fehlerquellen nachvollzogen werden. Sei e_x der relative Fehler zwischen dem Wert x und seiner Darstellung im Rechner. Dann ist

$$e_{a+b} = \frac{a}{a+b}e_a + \frac{b}{a+b}e_b$$
$$e_{a*b} \approx e_a + e_b$$
$$e_{a/b} = e_a - e_b$$

Wie sich einfach erkennen lässt, sollten solche Werte a und b mit ähnlich großem Betrag und ungleichem Vorzeichen nicht addiert opder subtrahiert werden, da dann der Nenner nahe Null wird und der Fehler somit stark ansteigen kann. Bei gleichem Vorzeichen tritt dieses Problem jedoch nicht auf. Gleichermaßen sollte bei der axpy-Operation $y := a * x + y$ nicht nahe Null sein. Multiplikation und Division sind hingegen weitestgehend unproblematisch nach obigen Formeln.

Die meisten Werte bei der Berechnung der oben spezifizierten Eingabewerte liegen im Intervall $[-4,0;\ 4,0]$, Daher sind die Chancen groß, dass genau solche Auslöschungen erfolgen, was letztendlich bis zu vertauschten Vorzeichen von α und β führt. Das Skalarprodukt bei C-XSC hingegen verwendet die exakten Multiplizier-Akkumulier-Operationen auf dem exakten Festkommaregister, so dass weder Vektor- noch Matrixmultiplikationen eine Fehlerquelle darstellen können. Damit sind die bleibenden möglichen Fehlerquellen die Subtraktion des Vektors r und die Division beim Normalisieren von $\mathrm{q_i}$.

Verbesserte Verwendung von C-XSC

Bislang konnten zweierlei Erkenntnisse erhalten werden: Einerseits müssen die Skalarprodukte unbedingt genauer berechnet werden, wozu exakte Arithmetik genutzt werden kann; andererseits müssen die Matrix-Additionen bzw. -Subtraktionen auch genauer durchgeführt werden. Dazu kann C-XSC durch Skalierung der Summanden mit 1,0 und anschließende Akkumulation ebenso eingesetzt werden. Also ist das Skalarprodukt $< u, v >$ der beiden Vektoren $u := (1,0; 1,0)$ und $v := (a_{i,j}; b_{k,l})$ exakt zu berechnen. Die Subtraktion von $\mathbf{A}$ und $\alpha\mathbf{I}$ wurde nach diesem Ansatz exakter durchgeführt. Dann wurde die Berechnung von r umgestellt, um diesen Ansatz ebenso auszunutzen. Die Ergebnisse in Abb. 5.5 und Tabelle 5.6 lassen zwar geringfügige Verbesserungen erkennen, dafür haben sich andere Eigenwerte sogar verschlechtert.

Weitere Möglichkeiten zur Verbesserung der Genauigkeit sind, wie bereits erwähnt, die Division bei der Skalierung von r zu $\mathrm{q_i}$. Da die Division hinsichtlich obigen Fehlermodells schon genau ist, verbleibt die Bestimmung der Länge $\sqrt{< \mathrm{r}, \mathrm{r} >}$.

Nr.	Errechneter Wert	Absoluter Fehler	Relativer Fehler
1	2,814172759342681	1,93323e-09	6,86963e-10
2	2,791845578330389	-0,99113	-0,55041
34	-3,004338709824201	0,86248	-0,40268
35	-3,026665469180980	-4,23589e-07	1,39952e-07

Tabelle 5.6: Eigenwerte und die absoluten und relativen Fehler bei der verbesserten Berechnung mit C-XSC.

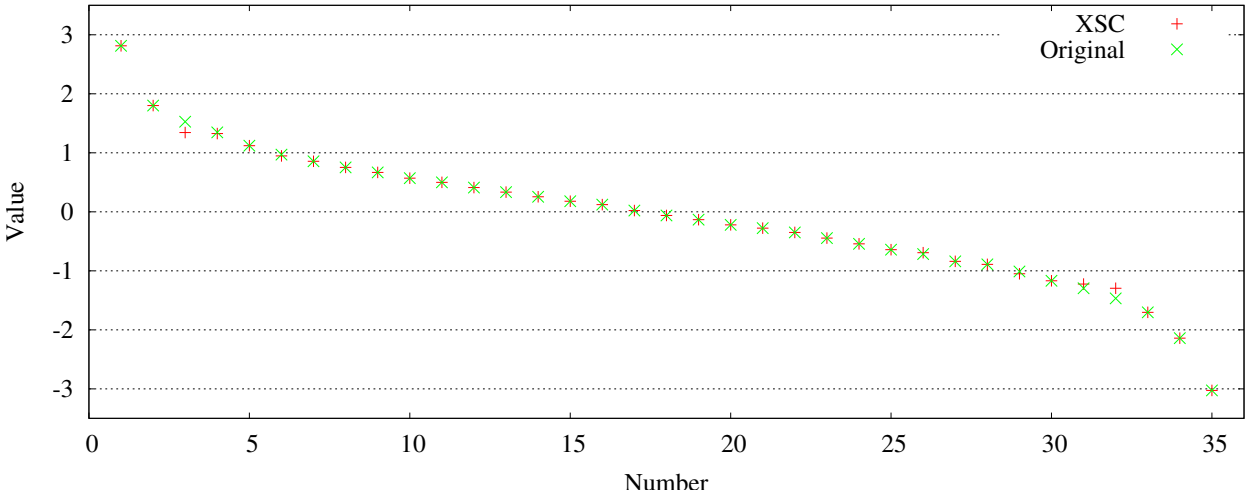


Abbildung 5.6: Mit verbesserter C-XSC-Implementierung errechnete Eigenwerte im Vergleich zu echten Eigenwerten: Vektoraddition, -subtraktion, Normberechnung exakt durchgeführt; Werte exakt ausgelesen und zwischengespeichert für unmittelbar folgende Operationen.

Das Problem des zwischenzeitlichen Rundens blieb natürlich bei der exakteren Akkumulation bestehen, kann aber durch genauere Zwischenformate, z.B. staggered Arithmetik oder vollständige Speicherung wie bei GMP, weitestgehend umgangen werden. Der Speicherbedarf des 4288 Bits = 67 * 64 Bits = 536 Bytes breiten Festkommaformats scheint für $n \times n$-Matrizen gar nicht übermäßig hoch mit $536\,\mathrm{B} * n^2 + 3 * 536\,\mathrm{B} * n + 2 * 536\,\mathrm{B} = (n^2 + 3n + 2) * 536\,\mathrm{B}$, z.B. für $n = 35$ sind dies lediglich 713.952 Bytes. Um die oben formulierten Behauptungen der Anwendbarkeit der Matrixbibliothek und der Einsatzbarkeit unterschiedlicher Arithmetik-Bibliotheken zu beweisen, wurden folgende Verbesserungen angewandt: Alle Werte werden mittels C-XSC in Staggered Arithmetik zwischengespeichert. Weitere arithmetische Operationen, wie die Normierung, werden in Staggered Arithmetik oder das Multiplizieren-Akkumulieren mit exaktem Festkommaakkumulator ausgeführt. Die Berechnung von r wurde umgestellt und dadurch exaktes Multiplizieren-Akkumulieren angewandt für diesen Teilschritt. Mit diesen Änderungen haben sich erst einigermaßen korrekte Eigenwerte errechnen lassen, wobei noch immer einige wenige Eigenwerte falsch sind oder fehlen. Die Werte sind in Abb. 5.6 abgebildet.

Es lässt sich schlussfolgern, dass die zwischenberechneten Werte auch exakt gespeichert werden müssen, um später wiederverwendet werden zu können, z.B. für die zweite Vektormultiplikation im Teil $\mathrm{q}^T \mathbf{A}\mathrm{q}$. Ferner ist die Berechnung der Division und Wurzel auf exakten Darstellungen nötig.

Zusammenfassung

Das Lanczos-Verfahren zum Bestimmen von Eigenwerten ist ein effektives, aber auch numerisch sehr empfindliches Verfahren. Unterschiedliche Möglichkeiten zur exakten Berechnung wurden untersucht und verglichen, welche die korrekten Eigenwerte ermitteln können. Der Nutzen der exakten Arithmetiken kann in Abhängigkeit der Daten darin liegen, die sonst nötigen Reorthogonalisierungen der Vektoren nach jeder Iteration zueinander zu vermeiden und dadurch insgesamt Laufzeit

einzusparen. Aufgrund ihrer langen Ausführungsdauer liefert die exakte Berechnung einen hervorragenden Kandidaten zur Beschleunigung mittels Hardwareverlagerung. Die in Abschnitt 5.4 geschilderten Ansätze zeigen, dass grobgranularere Ansätze verwendet müssen als solche auf Befehlsebene, um vor allem Daten vielfach zu verwenden, unnötige Datentransfers und natürlich auch zusätzlichen Kontrollaufwand in Form von Kontrollpaketen oder Steuerwerken in Hardware zu vermeiden. Die Untersuchungen in diesem Abschnitt haben ferner ergeben, dass ohne tiefgehendes Verständnis der dem Lanczos-Verfahren zugrundeliegenden Numerik weitere Operationen exakt durchgeführt werden müssen und die Daten ferner auch exakt vorgehalten werden müssen. Sollte sich das exakte Skalarprodukt in rekonfigurierbare Hardware verlagern lassen, so muss später eine gänzlich andere Implementierung eines Koprozessors zum Rechnen auf exakten Datendarstellungen entwickelt werden, welche mehr Operationen als das exakte Skalarprodukt bereitstellt. Weiter entsteht aufgrund der gewonnenen Erkenntnisse die Forderung, dass beispielsweise ähnlich der prozessorinternen Aufweitung von IEEE-754 von 54 Bits auf 80 Bits auch in Hardware verlagerte numerische Operationen bzw. Algorithmen mehr Genauigkeit aufweisen müssen. Dadurch sollten unerwünschte Nebeneffekte wie ungenauere Ergebnisse als bei Softwareausführung vermeidbar sein und die Fehlersuche bei der Implementierung von koprozessornutzender Software vereinfacht werden.

Um einerseits die bereits erworbenen Erkenntnisse weiter zu stützen und um andererseits außer dem exakten Skalarprodukt eine weitere Arithmetik zu betrachten, wird im folgenden Abschnitt die Intervallarithmetik untersucht. An ihr zeigt sich auch, dass der Einsatz von Mehrkernprozessoren im Falle von Spezialoperationen auf geringer Granularität nicht gewinnbringend ist.

5.2 Intervallarithmetik

Die *Intervallarithmetik* ist eine hilfreiche Arithmetik (vgl. Abschnitt 2.2.3), da sie den möglichen Bereich des Ergebnisses genau angibt und somit die Fehlerberechnung und eventuell auch die anschließende Behebung ermöglicht. Die genauere oder exakte Berechnung des Skalarprodukts hilft bei numerischen Instabilitäten, ist aber rechenaufwendiger und daher ein guter Kandidat zur Beschleunigung. Bei der Intervallarithmetik werden Zahlen z als Zahlenbereiche in Form von Tupeln $[u_z, o_z]$, bestehend aus unterer Grenze u_z und oberer Grenze o_z, dargestellt. Beim Anlegen einer Zahl kann diese exakt darstellbar sein ($u_z \equiv o_z$) oder im Rahmen einer Konvertierungsoperation beispielsweise aus dem Dezimalsystem als Tupel aus nächstkleinerer und nächstgrößerer Zahl angegeben werden ($|o_z - u_z| > 0$). Beim Durchführen arithmetischer Operationen auf Intervalldaten werden die neuen Grenzen durch reguläre arithmetische Operationen auf den Operandengrenzen und Auf- oder Abrunden ermittelt, wie auch in Abb. 5.7 illustriert ist. Besonders einfach ist dabei die Multiplikation $z := a \cdot b$ für acht von neun Fällen zu implementieren:

$$z = [u_z; o_z] := [abrunden(u_a \cdot u_b); aufrunden(o_a \cdot o_b)] \quad \text{(für positive Zahlen } a \text{ und } b\text{)}$$

Sie erfordert jedoch die Unterscheidung von neun Fällen für die Operanden und im neunten Fall gar die Berechnung von vier Produkten sowie die Minimums- und Maximumsberechnung. Weitaus aufwendiger in Hardware ist zwar die Addition/Subtraktion, benötigt aber keine unterschiedliche Fallbehandlung im Rahmen der Intervallarithmetik. Aufgrund des im Gegensatz zu regulären Gleitkommaoperationen weitaus höheren Rechen- und Kontrollaufwands stellt sich die Frage, ob Intervallarithmetik durch Mehrkernsysteme ohne Einsatz zusätzlicher Beschleunigereinheiten unterstützbar ist. Die zwei Operationen zur Bestimmung der unteren und oberen Grenze könnten parallel auf mehreren Kernen berechnet werden. Die Untersuchung dieser Idee einschließlich der erhaltenen Erkenntisse beschreibt dieser Abschnitt.

5.2.1 Eine parallele Implementierung auf einem Mehrkernsystem

Die Möglichkeit einer Implementierung mittels zweier Hardware-Threads soll betrachtet werden. Dazu werden die obere und untere Grenze des Ergebnisses jeweils auf unterschiedlichen Hardware-Threads errechnet sowie gegebenenfalls Operanden oder bereits berechnete Produkte ausgetauscht. Dabei entsteht Zusatzaufwand durch das Binden von Software-Threads auf die Hardware-Threads (*CPU-Pinning*) sowie durch die Synchronisation vor dem Variablenaustausch. Das Binden ist nötig, damit für jeden Software-Thread gewährleistet ist, dass separate Rundungseinstellungen genutzt

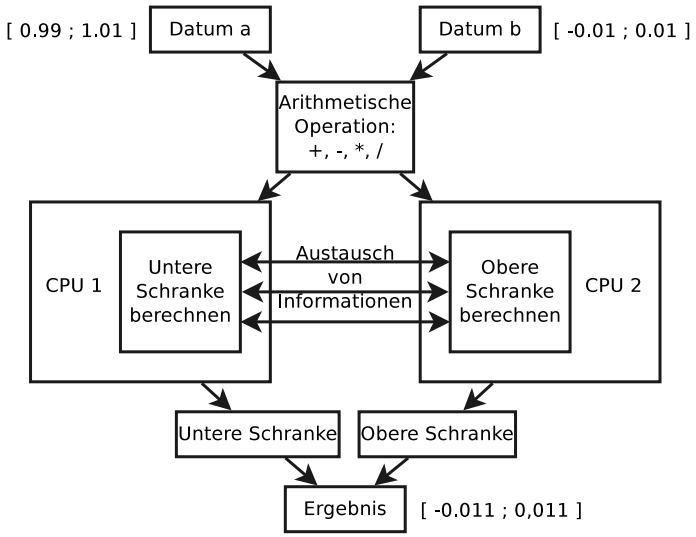


Abbildung 5.7: Intervallarithmetik: Implementierung und Ausführung der Addition nach [178] mittels zweier CPU-Kerne.

werden. Ohne das CPU-Pinning würden beliebige Prozessorkerne oder jeweils gar derselbe Kern verwendet und vielmehr noch, die Rundungseinstellungen für die FPUs wären nicht die garantiert richtigen.

Die Parallelisierung wurde dabei wie folgt umgesetzt. Zunächst wird für jede Operation ein Thread für die obere Grenze gestartet, die Berechnung der unteren Grenze wird vom Hauptprogramm ausgeführt. Mittels einer Vielzahl logischer Operationen werden für die unterschiedlichen möglichen Operationen die Unter- und Obergrenze des Intervalls bestimmt und nach Synchronisation mittels Semaphoren zwischen den beiden Threads ausgetauscht. Als Threadmodell wurden erst GLIB GThreads eingesetzt, die jedoch nur Wrapper um POSIX Threads sind und zusätzlichen Mehraufwand einführen, weshalb später direkt Pthreads verwendet wurden. Die mit den POSIX Threads erzielten Ergebnisse sind in Zeile 2 („je Iteration 2 neue Threads mit Datenaustausch“) in Tabellen 5.7- 5.8) für einen Zweikern- und einen Vierkernprozessor aufgeführt. Gegenüber der sequentiellen Implementierung ist eine Verlangsamung auf etwa 1/80 feststellbar. Die massiv vielen Threadstarts und benötigten Sychronisationen bei der Berechnung eines Skalarprodukts, das CPU-Pinning und nicht zuletzt auch die ineffektive Cache-Nutzung beim Schreiben des berechneten Intervallwertes fallen derart stark ins Gewicht, dass die Parallelisierung nicht lohnt und anstelledessen eine geradlinige, sequentielle Implementierung zu bevorzugen ist oder eben die Nutzung eines Akzelerators bzw. rekonfigurierbarer Hardware.

Die Synchronisation für den Austausch der Ober- und Untergrenzen kann mehr Aufwand bedeuten als die eventuell überflüssige, doppelte Berechnung innerhalb jeden Threads und wurde entsprechend entfernt (Zeile 3 „je Iteration 2 neue Threads, redundante Berechnungen“). Allerdings hat sich das Gegenteil in diesem Falle herausgestellt.

Daher wurde die Parallelisierung um eine Granularitätsstufe angehoben auf ein Skalarprodukt sowie mit nur insgesamt zwei Threads gearbeitet, die sich nach der Berechnung der jeweiligen Unter -und Obergrenzen des Produkts bzw. der Zwischensumme mittels Semaphoren synchronisieren. Diese Variante „Master- und Worker-Thread“ ist insgesamt deutlich schneller geworden, allerdings noch immer um Faktor 4-20 langsamer als die rein sequentielle Implementierung.

Aus hiesiger Studie wird noch einmal ersichtlich, dass die gewählte Granularität der Instruktionsebene zu gering ist, um Nutzen aus Parallelisierung ziehen zu können. Insbesondere lässt sich durch Parallelisierung in Software mittels Threads kein Nutzen erzielen. Aufgrund des Aufbaus der Intervallarithmetik aus Standard-Einheiten wie Gleitkommamultiplizierern lohnt sich daher die Implementierung selbiger in Hardware anstatt der Ausführung in Software. Die Implementierung einer Spezialeinheit aus Standardkomponenten erzielt entsprechend die beabsichtigte Wirkung [9]. Intervallarithmetik wurde in den erneuerten Standard IEEE 754-2008 aufgenommen und steht in

Vektorenlänge	1		100000		Statischer	Dauer pro
	Gesamt	Kern	Gesamt	Kern	Mehraufwand	Iteration
Sequentiell	1.000	36	115.000	107.472	7.628	1,075
je Iteration 2 neue Threads mit Datenaustausch	2.000	146	8.912.000	8.901.796	10.204	89,018
je Iteration 2 neue Threads, redundante Berechnungen	3.000	234	12.905.000	12.891.425	13.575	128,914
Master- und Worker-Thread	2.000	171	2.234.000	2.226.399	7601	22,264

Tabelle 5.7: Skalarprodukt in Intervallarithmetik implementiert mittels zweier Hardware-Threads (Zeitangaben in μs, gemessen auf einem Intel Core 2 Duo T7400 mit 2,16 GHz, kompiliert mit GCC 4.7 und -O3 für die Core2-Architektur).

Vektorenlänge	1		100000		Statischer	Dauer pro
	Gesamt	Kern	Gesamt	Kern	Mehraufwand	Iteration
Sequentiell	2.000	37	97.000	84.522	12.478	0,845
je Iteration 2 neue Threads mit Datenaustausch	2.000	158	6.778.000	6.768.350	9.650	67,684
je Iteration 2 neue Threads, redundante Berechnungen	2.000	153	7.429.000	7.416.828	12.172	74,168
Master- und Worker-Thread	3.000	103	1.107.000	1.096.493	10.507	10,965

Tabelle 5.8: Skalarprodukt in Intervallarithmetik implementiert mittels zweier Hardware-Threads (Zeitangaben in μs, gemessen auf einem Intel Core 2 Quad Q6600 mit 2,4 GHz, kompiliert mit GCC 4.6 und -O3 für die Core2-Architektur).

modernen Gleitkommaeinheiten in Prozessoren nun zur Verfügung. Daher wird sie fortan nicht weiter betrachtet, sondern stattdessen wird alternativen Arithmetiken auf Basis von Skalarprodukten zugewandt.

5.3 Exakteres Skalarprodukt in Hardware

Über die der Gleitkommaarithmetik inhärente Problematik der fensterbasierten Genauigkeit wird man sich besonders bewusst, wenn man neben den in Abschnitt 2.2.3 formulierten Darstellungen versucht, eine mittels OpenMP parallelisierte Variante des Skalarprodukts zweier Vektoren über mehrere Läufe hinweg auf CMP- oder SMP-Systemen zu analysieren. Dabei hat sich in verschiedenen Messungen gezeigt, dass insbesondere bei größer dimensionierten Problemen die Zahl benötigter Iterationen zum Lösen eines linearen Gleichungssystems unter Verwendung jenes parallelisierten Skalarprodukts stark schwankt, was vor allem auf unterschiedliche, ungünstige Ausführungen des Skalarprodukts zurückzuführen ist. Selbstverständlich lässt sich dies mittels Zerlegung in Teilsummanden beheben, die dann alle in einer ähnlichen Größenordnung liegen mögen, so dass ihre Summe exakter sei als das Ergebnis bei der sequentiellen Akkumulation. Genau dieses Phänomen ist aber auch die Ursache dafür, dass mittels OpenMP parallelisierte Implementierungen eines Skalarprodukts zwar eine gewisse Varianz hinsichtlich des Ergebniswerts aufgrund des geschilderten Indeterminismus in der Bearbeitungsreihenfolge aufweisen, dafür aber aufgrund der Summation der in verschiedenen Threads errechneten Teilsummen wesentlich exakter sind. Die Varianz in den benötigten Iterationen bei paralleler Ausführung kann natürlich durch Festlegen einer statischen Bearbeitungsreihenfolge[1] eingeschränkt oder gar vollständig verhindert werden. Dieser Ansatz kann selbstverständlich von einem hardwarenahen Bibliotheksentwickler genutzt werden. Es soll jedoch vornehmlich betrachtet werden, welche Möglichkeiten man Anwendungsentwicklern und Domänenexperten bieten kann, dass sie ohne Berücksichtigung von Hardwareeigenschaften Leistungssteigerungen durch den Einsatz von heterogenen Mehrkernsystemen erfahren können. Tabelle 5.9 gibt einen Überblick über die Laufzeit in Mikrosekunden und benötigte Anzahl an Iterationen für ein vorkonditioniertes CG-Verfahren, wenn man sequentielle Implementierungen mit mehrfädigen und diese gegenüber Hardwareimeplementierungen mit einfacher Genauigkeit jeweils vergleicht sowie

[1] #pragma omp schedule static

Größe	32×32	64×64	128×128	256×256	512×512	1024×1024	2048×2048	4096×4096
					Iterationen			
Sequentiell	29	60	130	310	749	1958	5875	26308
2 Threads	24	64	127	272	653	1750	3690	
4 Threads	24-29	49-60	115-128	246-273	544-692	1415-1722	4803-12638	11579-11853
24 Threads	24-29	49-49	100-127	232-256	515-589	1191-1392	2964-2992	9835-10284
Hardware	24	60	128	284	712	2079	5132-5160	37253-39332
32 zusätzliche Bits	24	49	100	232	512	946	2085	
					Laufzeit (μs)			
Sequentiell	739,033	5916,633	51840,867	488942,531	6547319,000	93162760,000	1172235607,000	31996284965,000
2 Threads	703,033	4498,567	31148,199	252894,500	4077900,500	70472840,000	615067088,000	
4 Threads	940,027	3405,120	17514,926	123078,773	1355587,750	32114384,000	105869040,000	194770272,000
24 Threads	1565,650	4068,720	12439,463	54065,086	372004,188	5158851,000	77629640,000	164665358,236
Hardware	1850,000	7623,000	43030,000	336029,000	7883096,000	85452171,000	360251361,000	10721530963,333
32 zusätzliche Bits	1713,000	6157,000	33475,000	271683,000	2285635,000	16630197,000	145596258,000	

Tabelle 5.9: Einfluss der Genauigkeit eines Skalarprodukts bzw. seiner Berechnung auf ein vorkonditioniertes CG-Verfahren. Sequentiell und 2 Threads auf Intel Xeon 5138 gemessen, 4 Threads auf Intel Core2 Quad, 24 Threads auf Intel Xeon X5670, Hardwaremessungen mit dem mikroprogrammierbaren Steuerwerk (vgl. Abschnitt 6.4).

eine Implementierung in rekonfigurierbarer Hardware um zusätzliche 32 Bits für die Mantisse bei den Operationen des Skalarprodukts erweitert, ohne dabei die Bearbeitungsreihenfolge statisch und vorab festzulegen.. Sowohl die Laufzeit insgesamt als auch die Anzahl an Iterationen können durch eine exaktere Implementierung drastisch verkürzt werden, was bei den OpenMP-Implementierungen als sogenannter „superlinearer Speedup“ herüberkommt, aber allein von der geschickteren Akkumulation herrührt.

Es ist also sinnvoll, Skalarprodukte aufgrund der möglichen Erhöhung der numerischen Stabilität genauer zu betrachten und dedizierte Akzeleratoren dafür zu entwickeln. Eine besondere Möglichkeit bietet sich mit einem exakten Skalarprodukt auf Festkommabasis, denn das dazu nötige breite Akkumulatorregister könnte ebenso wie die bit-basierende Verwaltung gut in rekonfiguierbarer Hardware umgesetzt werden. Daher werden in den folgenden drei Abschnitten unterschiedliche Ansätze geschildert, exakte Skalarprodukte und weitere Operationen zu implementieren und für Anwendungsentwickler nutzbar zu machen.

5.4 Koprozessor für exaktes Skalarprodukt

Bei exakter Arithmetik nach Abschnitt 2.2.3 und 2.2.3 wird intern ein Festkommaformat mit Zweierkomplementdarstellung verwendet, das den gesamten darstellbaren Zahlenraum des zu verwendenden Formats abzdeckt. Bei einfacher Genauigkeit werden 640 Bits zur genauen Akkumulation benötigt, zur Verwendung doppelt genauer Daten werden für das Skalarprodukt 4288 Bits benötigt, die jeweils in 32- oder 64-Bit-Blöcken arrangiert werden.

Eine Addition oder Subtraktion eines eingehenden Datums erfolgt dabei wie in Abb. 5.8 dargestellt: Anhand des Exponenten wird derjenige Block errechnet, der die letzten Mantissenbits des Datums aufnehmen wird; diese Bits werden addiert. Zusammen mit dem eventuell aufgetretenen Überlaufbit werden die vorderen Mantissenbits auf den nächsthöheren Block addiert. Tritt auch hier ein Überlauf auf, so erlaubt die schnelle *Carry-Propagation*-Logik nach Kulisch die Bestimmung desjenigen Blocks, der den Überlauf aufzunehmen imstande ist. Dazu werden zwei Register mit Anzahl Bits entsprechend der Anzahl an Blöcken mitgeführt, die angeben, ob ein Block nur gleiche Werte enthält (`all_mask`) und welcher Wert dies ist (`all_value`). Der erste nächsthöhere Block, dessen `all_mask`-Flag nicht gesetzt ist oder dessen `all_value`-Flag anzeigt, dass der Überlauf aufgenommen werden kann, wird erhöht. Dies hat Ähnlichkeit zu dem bei Carry-Lookahead zugrundeliegenden Prinzip, erfordert aber weitaus weniger Aufwand.

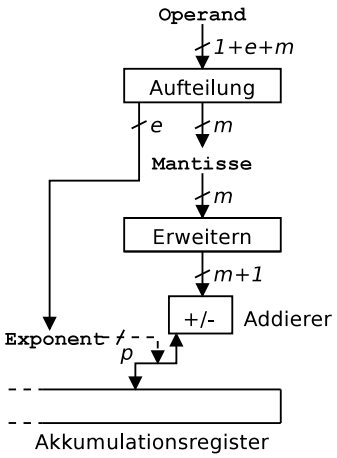


Abbildung 5.8: Exakte Akkumulation.

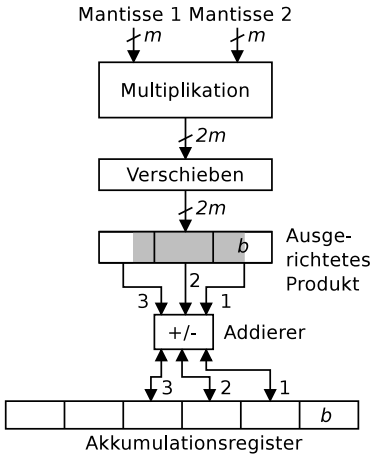


Abbildung 5.9: Exaktes Multiplizieren-Akkumulieren.

Das von Kulisch und Bierlox entworfene Konzept wurde auf die an den HTX-Bus angeschlossene FPGA-Karte (vgl. Abschnitt 2.1.4) portiert [83] und so an ein HyperTransport-basiertes AMD Opteron-System angebunden. Im Gegensatz zum originalen Konzept implementiert es weder Pipelining (Parallelismus auf Befehlsebene) noch Multiplikations-Akkumulations-Operationen, sondern lediglich exakte Akkumulationen. Anstelledessen verfügt es aber über mehrere Einheiten (Task-Parallelismus). Wesentlich ist jedoch für exakte Skalarprodukte, dass die Produkte zweier einfach oder doppelt genauer Zahlen einen mehr als doppelt so großen Bereich abdecken, und auch die vorhandene Genauigkeit der Produkte der beiden Mantissen kann zusätzlich berücksichtigt werden. Im Gegensatz zu IEEE-754-konformen Gleitkommaimplementierungen lassen sich dann alle Produkte sämtlicher darstellbarer Zahlen akkumulieren wie in Abb. 5.9 ersichtlich, wodurch sich die Entwicklung von Anwendungen mit Skalarprodukten für den Anwendungsentwickler wesentlich vereinfacht. Wie in der Veröffentlichung zum exakten Skalarprodukt auf dem UoH HTX-Board [NBKK09] beschrieben, wurde die Einheit zur Umsetzung des exakten Skalarprodukts für einfach genaue Operanden erweitert und als Koprozessor eingebunden.

5.4.1 Interne Zahldarstellung

Wie bereits erwähnt, wird intern eine andere Zahldarstellung als die vom IEEE-Gleitkommaformat her bekannte Darstellung „Vorzeichen-Charakteristik-Mantisse“ ($Zahl = 2^{Exponent} * 1, Mantisse$; mit m Bits für den Exponenten und n Bits bei der Mantisse) benötigt und verwendet. Mittels einer Festkommadarstellung mit Zweierkomplement lassen sich hervorragend gebrochen-rationale Zahlen angeben und zeitgleich sowohl betragsmäßig sehr kleine als auch sehr große Zahlen darstellen:

$$Zahl = Ganzzahlanteil.Nachkommastellen \tag{5.2}$$

mit jeweils einer Länge von k und l Stellen. Dabei rangiert die Ganzzahl im Bereich von -2^{k-1} bis $2^{k-1} - 1$ und die Nachkommastellen zwischen 2^{-l} und $\sum_{i=1}^{l} 2^{-i} = 1 - 2^{-l}$ mit sowohl positiven als auch negativen Werten. Zusätzlich ist die wichtige Möglichkeit zur Repräsentation der Null gegeben. Bei IEEE-754-konformen einfach genauen Zahlen mit $m = 8$ und $n = 23$ ist $0x00000001$ die betragsmäßig kleinste darstellbare Zahl mit Wert $2^{-2^{m-1}+2-n} = 2^{-149}$, wohingegen die größte $0x7FFFFFFF$ ist mit Wert $2^{2^{m-1}} - 2^{-n} < 2^{128}$. Daher müssen zum genauen Rechnen mit einfach genauen Operanden folgende Ungleichungen erfüllt sein:

$$k >= 128; l >= 149 \tag{5.3}$$

Für doppelte Genauigkeit mit $m = 11$ und $n = 52$ sind die Anforderungen sehr hoch:

$$k >= 1024; l >= 1074 \tag{5.4}$$

Damit kann man zwar bereits innerhalb dieses Bereichs korrekt akkumulieren, es kommt aber schnell zu Überläufen. Daher sind zusätzliche Bits empfehlenswert, und obendrein benötigt man zur Akkumulation eines Produkts bereits die doppelte Anzahl an Bits. Kulisch geht davon aus, dass nicht mehrere Milliarden Akkumulationen erfolgen werden, sondern lediglich einige Millionen, und schlägt daher 86 weitere Bits zur Behandlung von Überläufen vor bzw. 92 Bits im Falle von doppelt genauen Operanden. Somit kommt man auf eine Anforderung von $2 * (128 + 149) + 86 = 640$ Bits bzw. $2 * (1024 + 1074) + 92 = 4288$ Bits für die Breite des Festkommaakkumulatorregisters.

5.4.2 Implementierung

Wie in Abb. 5.10 rechts dargestellt, ist der Koprozessor innerhalb einer speichereingebundenen[2] HT-Schnittstelle an die HyperTransport-Queues mittels einer zusammenfassenden „Simplify Unit“ über die physischen IO-Puffer angebunden. Die Speichereinbindung des Koprozessors ist in Abb. 5.11 dargestellt. Anfragen an verschiedene Adressen in der 4 KB großen Speicherseite führen zu unterschiedlichen Befehlen wie dem Auslesen von Daten, dem Schreiben von Flags oder dem Multiplizieren-Akkumulieren zweier Daten, indem an zwei aufeinanderfolgende Speicherstellen, beispielsweise 136 und 140, die Operanden geschrieben werden. Beim Auslesen des Akkumulatorregisters werden die fünf Rundungsmodi nach dem IEEE-754-Standard unterstützt; dazu dienen die ersten fünf Adressen 0, 4, 8, 12 und 16. Zwar sind die Akkumulationseinheiten völlig unabhängig voneinander, aber über Hypertransport und das Memory-Mapping kann immer nur eine Einheit zu einem Zeitpunkt angesprochen werden. Diese Schnittstelle handhabt dann mittels der Systemtreiber für den PCI-Bus und des HyperTransport-Protokolls das Versenden von Befehlen, Daten und Adressen sowie das Empfangen von Antwortpaketen.

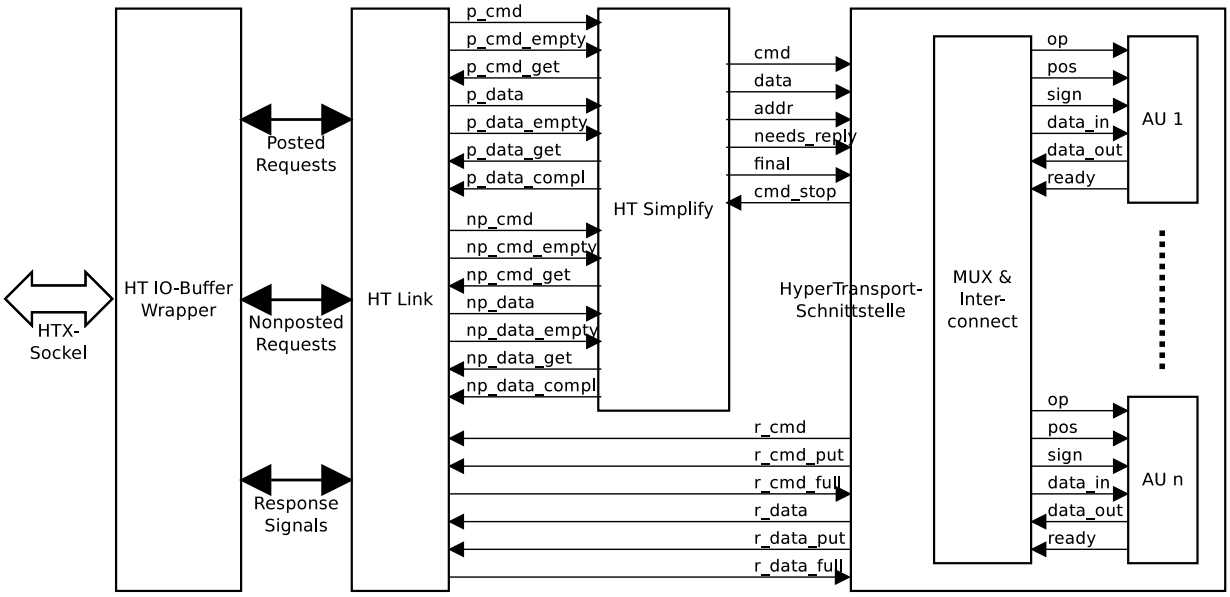


Abbildung 5.10: Hardwarearchitektur des exakten Akkumulators auf dem UoH HTX-Board.

Ohne die Multiply-Accumulate-Logik passen 16 Einheiten (*Exact Accumulation Units, EAUs*) ohne jegliche Optimierungen auf den zugrundeliegenden Xilinx Virtex-4FX100-FPGA. Nach Optimierungen am Design sind ebenso 16 Einheiten mit MAC-Unterstützung nutzbar. Da eine Taktrate von 100 MHz erreicht werden konnte, kann das System theoretisch einen unidirektionalen Durchsatz von 800 MB/s erzielen, wenn bei jeder Taktflanke und 200 MHz-HyperTransport-Takt je 16 Bits übertragen werden.

Der Koprozessor würde aufgrund des darin enthaltenen FPGAs bei 64-Bit-Designs für doppelt genaue Operanden keine ausreichenden Taktraten erreichen und nicht mehrere MAC-Einheiten parallel zueinander ermöglichen. Aufgrund dieser durch die Hardware vorgegebenen Einschränkung arbeitet das System also nur mit einfach genauen Operanden, also auf 32-Bit-Daten. Intern wird

[2] engl. memory-mapped

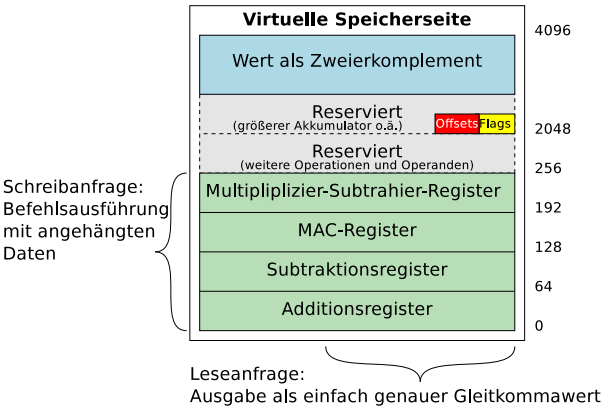


Abbildung 5.11: Speichereinbindung der Exakte-Arithmetik-Einheiten.

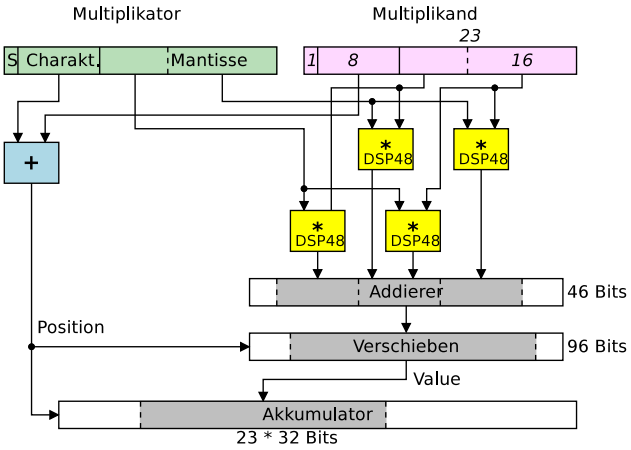


Abbildung 5.12: Multiplizier-Akkumulier-Operation auf dem Koprozessor.

daher auch auf 32-Bit-Blöcken des Akkumulatorregisters gearbeitet. Anhand des bzw. der Exponenten wird sowohl der erste Zielblock bestimmt als auch die notwendige Ausrichtung des/der Operanden auf den Zielblock. Indem sogenannte Dual-Port-Speicher auf dem FPGA verwendet werden, kann der nächste Block im Register gleichzeitig ausgelesen werden, während das Ergebnis einer Teilakkumulation in den passenden Block geschrieben wird. Mittels des Konzepts der schnellen Auflösung möglicher Überträge werden lediglich sechs Takte nach der Datenübertragung benötigt, um ein Datum zu akkumulieren. Trotz fehlenden Pipelinings darf immerhin im letzten Takt bereits die Bearbeitung eines neuen Befehls begonnen werden, so dass bereits mit sechs Einheiten die Latenz komplett überbrückbar sein sollte unter der Annahme, dass der Datentransfer lediglich einen Takt benötigt. Das Multiplizieren-Akkumulieren benötigt insgesamt 18 Takte, da wie in Abb. 5.12 erkennbar die Latenz der Multiplizierer und Additionen mit vier Takten zu Buche schlägt, da drei Blöcke akkumuliert werden müssen sowie der Übertrag aufgelöst werden muss. Dann sind acht Takte dem internen Datentransfer zuzuschreiben, wovon sechs zwischen dem HT-Link und dem IO Buffer Wrapper ganz links in Abb. 5.10 verloren gehen und nicht mittels Verwendung mehrerer Einheiten verdeckt werden können, da zu einem Zeitpunkt immer nur eine HyperTransport-Verbindung bestehen kann. Immerhin kann wie oben beschrieben ein Takt dadurch eingespart werden, dass bereits während der letzten Stufe ein neuer Befehl ausgeführt wird, was somit zu einer Latenz von 17 Takten führt.

Syntheseergebnisse

In Tabelle 5.10 sind die Syntheseergebnisse für den Koprozessor mit variierender Anzahl an exakten Akkumulatoren angegeben. Tabelle 5.11 gibt die Ergebnisse für unterschiedlich viele Multiplikations-Akkumulationseinheiten wieder. Um 16 Einheiten unterzubringen, waren starke weitere Optimierungen und Veränderungen an den Werkzeugeinstellungen nötig, weshalb die Ergebnisse mitunter besser ausfallen als bei den bloßen Akkumulationseinheiten. Hardware-Implementierungen benötigen also insbesondere bei FPGAs weitläufige Kenntnisse der Hardware (FPGA-Architektur) sowie der eingesetzten Entwicklungswerkzeuge, um nicht nur funktionstüchtig, sondern auch effizient[3] zu sein. Aus den Ergebnissen wird ersichtlich, dass die Logik des HyperTransport-Kerns und die Schnittstellen nur vergleichsweise wenige Ressourcen verwenden. Die maximal erzielbare Taktrate beläuft sich für die erste Variante ohne Multiplikation auf 120 MHz, während sie bei der multiplikations-erweiterten Variante nur 100 MHz beträgt. Insgesamt konnte mittels dieser Implementierung die Spezifikation des HTX-Bus eingehalten werden ohne Taktveränderungen und ohne damit einhergehende Nachteile. Den kritischen Pfad verursacht dabei das Auswählen aus einem großen Register mit 23+1 Blöcken zu je 32 Bit anstatt nur 16+1 Blöcken. In beiden Fällen reicht die Geschwindigkeit jedoch nicht für die Verwendung des HT400-Protokolls aus, wozu der Koprozessor mit 200 MHz betrieben werden müsste.

[3] hinsichtlich Ressourcennutzung und Zielerfüllung

Ressource	**Konfiguration**					**verfügbar**
	1 EAU	2 EAUs	4 EAUs	8 EAUs	16 EAUs	
Slice Flip-Flops	4.835	5.350	6.409	8.536	12.783	84.352
Belegte Slices	7.795	9.501	13.701	19.254	31.923	42.176
4-Eingaben-LUTs	10.324	13.274	19.312	31.477	55.788	84.352
Logik	10.066	12.888	18.654	30.291	53.562	
Route-thru	234	362	634	1.162	2.202	
Shift-Register	24	21	24	24	21	
RAMB16s	26	27	29	33	41	
Anz. äquiv. Gatter	1.811.662	1.902.807	2.085.366	2.451.628	3.184.303	

Tabelle 5.10: Implementierungsergebnisse für variierende Anzahl von exakten Akkumulationseinheiten (EAUs).

Ressource	**Konfiguration**					**verfügbar**
	1 EAU	2 EAUs	4 EAUs	8 EAUs	16 EAUs	
Slice Flip-Flops	5.085	5.890	7.505	10.728	17.167	84.352
Belegte Slices	7.955	10.718	14.846	24.008	39.141	42.176
4-Eingaben LUTs	11.355	15.535	23.868	40.570	73.868	84.352
Logik	11.059	15.077	23.084	39.134	71.129	
Route-thru	272	434	760	1.412	2.715	
Shift-Register	21	21	21	21	24	
RAMB16s	26	27	29	33	41	
DSP48s	4	8	16	32	64	160
Anz. äquiv. Gatter	1.821.103	1.923.091	2.126.585	2.533.808	3.346.401	

Tabelle 5.11: Implementierungsergebnisse für variierende Anzahl von exakten Multiplikations-Akkumulationseinheiten (EAUs).

5.4.3 Evaluation

Die Evaluation exakter Arithmetik im Allgmeinen und des implementierten Koprozessors im Speziellen besteht aus Messungen zur Laufzeit, Betrachtung der benötigten Iterationen bis hin zur Konvergenz eines Verfahrens und natürlich aus der erzielbaren Genauigkeit. Zur Überbrückung der für die Berechnung benötigten Latenz sollten drei exakte Skalarprodukteinheiten ausreichen. Dennoch reduziert sich der maximal erreichbare Durchsatz von 800 MB/s auf weniger als 50 MB/s.

Zur Ermittlung des Zeitbedarfs gegenüber Standardimplementierungen wurden Laufzeitmessungen von Matrixmultiplikationen durchgeführt für das IJK- sowie das IKJ-Schema. Die Ergebnisse sind in Abb. 5.13 dargestellt. Probleme mit der Systemhardware machten es nötig, die HyperTransport-Kanäle zu entlasten, beispielsweise über hinreichend langes Warten, weshalb die Ausführungszeit mit nur einer MAC-EAU besonders hoch ausgefallen ist. Allgemein fällt die Laufzeit unter Verwendung exakter Arithmetik weitaus höher aus. Der Nutzen muss daher also über verringerte Anzahl an Iterationen bei iterativen Verfahren erzielt werden.

Um dies zu überprüfen, wurden unterschiedliche Routinen zur Berechnung eines Sinuswerts einander gegenüber gestellt. Die zu untersuchende Fragestellung dabei ist, ob unter Verwendung der exakten Arithmetik das Verfahren dermaßen schneller konvergiert, dass sich der Einsatz exakter Arithmetik lohnt. Die erste Routine ist die von der GNU C Library zur Verfügung gestellte Implementierung. Sie kann mit einfach oder doppelt genauen Argumenten nach dem IEEE-754-Format verwendet werden. Die zweite Routine basiert auf einer Look-Up-Tabelle und ist entsprechend auch nur für einfach und doppelt genaue Argumente geeignet, da aufgrund der nötigen Diskretisierung des Intervalls $[-\pi;\pi]$ in eine feste Anzahl von Tabelleneinträgen kein Vorteil durch genauere Einträge erzielbar ist. Look-Up-Tabellen sind besonders schnell und werden daher in zeitkritischen Anwendungen ohne besonders hohe Anforderungen an Genauigkeit eingesetzt wie etwa in Computerspielen. Die dritte und letzte Routine implementiert die Taylor-Reihenentwicklung, bis Konvergenz erreicht ist:

$$sin(x) = \sum_{i=1}^{n} (-1)^{i+1} \frac{x^{2i-1}}{(2i-1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \tag{5.5}$$

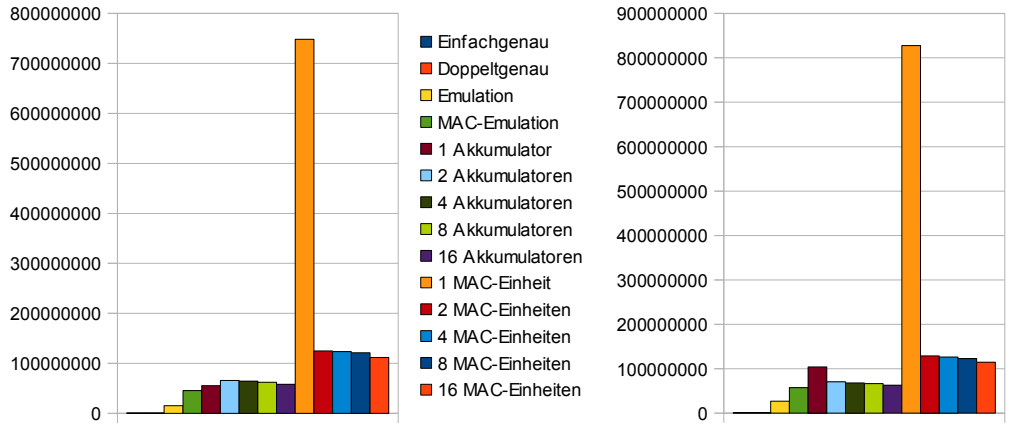
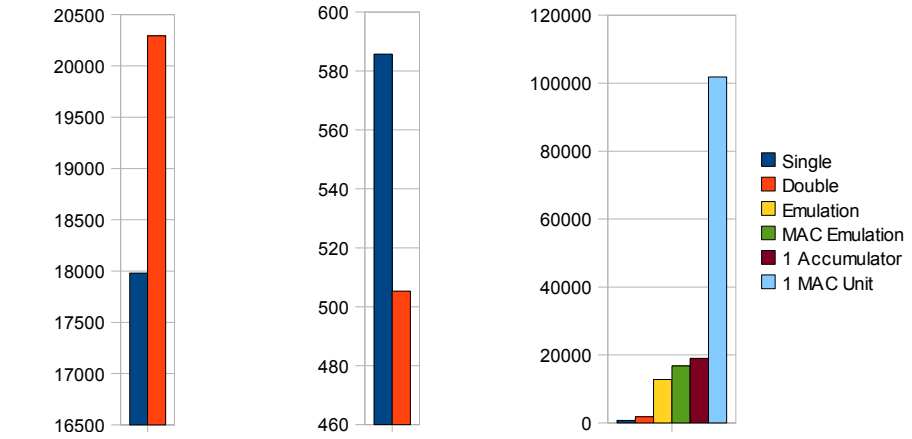


Abbildung 5.13: Vergleich der Laufzeiten in CPU-Takten für Matrixmultiplikationen ohne und mit exakter Arithmetik. Links: IJK-Multiplikation, rechts: IKJ-Multiplikation.



(a) GNU C Library (b) Look-Up-Tabelle (c) Taylor-Reihe

Abbildung 5.14: Laufzeit in CPU-Takten bei der Berechnung des Sinus von 1.0.

Zwar ist die Berechnung über die Taylor-Reihe an sich wesentlich langsamer, sie erlaubt aber das Einbinden des externen Koprozessors. Die Ergebnisse für die drei Routinen sind in Abb. 5.14 illustriert. Auch hier ist festzustellen, dass der Mehraufwand der vorliegenden Hardware-Implementierung, aber auch der emulierten Version, zu groß ist, obwohl die emulierte Version mit vier Iterationen am schnellsten konvergierte, die Hardware-Variante sowie die einfach genaue nach fünf Iterationen ein Ergebnis lieferten, die doppelt-genaue Variante allerdings erst nach neun Iterationen. Auch hier musste bei der Verwendung einer einzelnen exakten MAC-Einheit in Hardware zusätzliche Verzögerung zwischen jede Iteration eingefügt werden, so dass die Messwerte für die Ausführungszeiten nicht aussagekräftig sind. Zu erwarten ist etwa die doppelte Laufzeit gegenüber der exakten Akkumulation ohne die Multiplikationseinheit. Insgesamt ist die Verlagerung exakter Arithmetik in einen Koprozessor mit rekonfigurierbarer Hardware noch nicht als vorteilhaft zu bewerten.

Aus diesem Grund wird abschließend die erzielbare Genauigkeit anhand der Berechnung der Euler-Zahl betrachtet. Aus Tabelle 5.12 wird ersichtlich, dass mittels exakter Arithmetik das Ergebnis näher am echten Ergebnis liegt als bei Verwendung einfacher Genauigkeit. Auch hier beeinflussen besondere Gegebenheiten der Hardware wieder die Laufzeit der Multiplizier-Akkumuliereinheit negativ; allerdings kommt auch hinzu, dass nach jeder Iteration das exakte Ergebnis ausgelesen und als einfach genauer Wert ausgegeben wird. Nach bereits zehn Iterationen ist das einfach genaue Ergebnis unverändert, bei der doppelt genauen Variante nach 17 Iterationen. Nach 39 bzw. 178 Iterationen ist das zu akkumulierende Produkt nicht mehr als einfach bzw. doppelt genaue Zahl darstellbar. Die MAC-Einheit könnte aufgrund der Verwendung von mehr internen Bits zur Zwischenspeicherung des Produkts auch weiter akkumulieren. Ein genaueres Ergebnis nach dem Prinzip der „Stagge-

Variante	Ergebnis	Ausführungszeit (in ms)	Iterationen Ergebnis unverändert	Iterationen Summand ! = 0
Einfach genau	2,7182819843292236328125	3.150	10	39
Doppelt genau	2,71828182845904553488480.. ..14849026501178741455078125	6.566	17	178
Emulation ohne MAC	2,71828174591064453125	19.935,9	10	39
Emulation mit MAC	2,71828174591064453125	26.171,3	10	39
Akkumulator	2,71828174591064453125	31.486,2	10	39
Multiplizer-Akkumulator	2,71828174591064453125	188.945,7	10	39

Tabelle 5.12: Ergebnis, Laufzeit und Anzahl benötigter Iterationen der Berechnung der Euler-Zahl.

red Arithmetik“ kann mittels Subtraktion des ausgelesenen Werts und erneutem Auslesen erhalten werden. Er beträgt dann 2,71828183528879208097350783646106719970703125 für den Akkumulator und 2,71828183518901056459071696735918521881103515625 für die MAC-Einheit. Gleichermaßen sind auch bei der Taylorreihe für $\sin(x)$ die Ergebnisse unter Verwendung der exakten Arithmetik um einiges genauer. Insgesamt wird somit der Vorteil der internen genauen Produktberechnung (2*54 Bits = 108 Bits) bei der MAC-Einheit deutlich ersichtlich.

5.4.4 Zusammenfassung

Mittels der vorliegenden Implementierung wurde gezeigt, dass rekonfigurierbare Logik prinzipiell für exakte Arithmetik einsetzbar ist. Die untersuchten Verfahren konvergieren schneller und liefern genauere Ergebnisse als die auf einem Prozessor ausgeführten Varianten mit einfacher oder doppelter Genauigkeit. Wichtig sind bei der Verwendung rekonfigurierbarer Hardware leistungsfähige und gut nutzbare Hardware- und Software-Schnittstellen sowie passende Anwendungen. Es hat sich mit der vorliegenden Implementierung auch gezeigt, dass Hardwareentwickler benötigt werden, um effiziente[3] und funktionsfähige Hardwareschaltungen mittels Hardwarebeschreibungssprachen zu entwickeln.

Die bislang gewählte Granularität der Beschleunigung auf Operationsebene und damit Unterstützung von einfachen Funktionen wie Matrixmultiplikation, Sinus- oder Eulerzahl-Berechnung erwies sich aufgrund des hohen Aufwands zur Ansteuerung des Koprozessors und zur Datenübertragung als nicht sinnvoll. Die Berechnung ist wesentlich schneller, wenn doppelte Genauigkeit verwendet wird. Das wiederholte Auslesen des aktuellen Akkumulatorwerts sollte daher unterlassen werden, der Koprozessor in Streaming-Manier die Daten zur Verarbeitung erhalten und die Konvergenzprüfung selbst vornehmen.

Aus diesen Gründen müssen im Weiteren folgende zwei wesentliche Aspekte angegangen werden: Die **Erhöhung der Granularität auf Funktionsebene** und die **Erweiterung der Implementierung auf doppelte Genauigkeit**.

5.5 Ein Hardwarekonzept zur exakten Matrixmultiplikation

Zur Beschleunigung numerischer Anwendungen mittels heterogener Mehrkernsysteme wurde in dieser Schrift bislang die Intervallarithmetik betrachtet, welche sich nicht mittels mehrfädiger Ausführung beschleunigen lässt. Hingegen konnte festgestellt werden, dass exaktere Implementierungen von Operationen wie dem Skalarprodukt durchaus nutzbringend sind und die Beschleunigung von Spezialoperationen somit einen sinnvollen Ansatz darstellt. Die implementierte und untersuchte exakte Arithmetikeinheit zur Unterstützung einfacher Funktionen hat sich jedoch als nicht ausreichend erwiesen. Ulrich Kulisch hat das Konzept ohnehin mehr als exaktes Skalarprodukt im Kontext von Vektor- und Matrixmultiplikationen gesehen, bei denen Daten als Block gelesen oder gestreamt würden. Die Granularität oder Abstraktion wird daher nun zu Koprozessorentwürfen auf Funktionsebene erhöht, um vollständige Verfahren zu unterstützen.

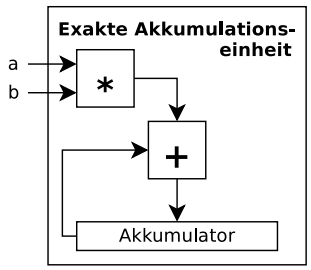


Abbildung 5.15: Eine Einheit, die zwei Operanden zeitgleich entgegennimmt und deren Produkt akkumuliert.

Normalerweise werden numerische Verfahren mit erhöhten Genauigkeitsanforderungen unter Zuhilfenahme weiterer Bibliotheken wie QD [19], GMP [117], XSC [173, 174, 175, 176] implementiert, die intern andere Zahlendarstellungen verwenden, dadurch aber auch erheblich langsamer sind, wie bereits gezeigt wurde. Sind diese Bibliotheken nicht einsetzbar, so besteht eine weitere Lösung aus der numerischen/algorithmischen Behandlung der möglichen eingehandelten Instabilitäten, z.B. Reorthogonalisierung der erzeugten neuen Basis bei Krylov-Unterraum-Methoden. Letztendlich können auch schlichtweg weitaus mehr Iterationen nötig sein, bis ein ausreichend genaues Ergebnis gefunden ist. Im schlimmsten Fall wird aber tatsächlich kein Ergebnis gefunden, weil das Verfahren nicht konvergiert. Als abstrakteres Verfahren wird nun eine exakte Matrixmultiplikationseinheit unter Verwendung doppelt genauer Operanden betrachtet. Folgend sind Analyse, Entwurf und Implementierung eines exakten Skalarprodukts für doppelt genaue Operanden, ein Konzeptentwurf zur Ausnutzung der verfügbaren Bandbreite sowie Evaluation und Skalierung beschrieben auf Basis einer Veröffentlichung [NB10].

5.5.1 Zerlegung der Matrixmultiplikation

Matrixmultiplikationen können als Reihe von Matrix-Vektor-Multiplikationen betrachtet werden, die wiederum selbst aus einer Reihe von Vektor-Vektor-Multiplikationen, also Skalarprodukten[4], aufgebaut sind:

$$C = A \cdot B = \begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix} \cdot \begin{pmatrix} b_1 & \cdots & b_o \end{pmatrix} = \begin{pmatrix} A \cdot b_1 & \cdots & A \cdot b_o \end{pmatrix} = \begin{pmatrix} a_1 \bullet b_1 & \cdots & a_1 \bullet b_o \\ \vdots & \ddots & \vdots \\ a_m \bullet b_1 & \cdots & a_m \bullet b_o \end{pmatrix} \tag{5.6}$$

Dabei bezeichnet A eine m×n-Matrix mit den Zeilen a_1 bis a_m, B eine n×o-Matrix mit den Spalten b_1 bis b_o, und C ist dann entsprechend eine m×o-Matrix. Der Operator $\bullet$ bezeichnet hier das Skalarprodukt. Ein einfaches System bestehend aus lediglich einem Akkumulator (anstatt mehreren) für die exakte Berechnung dieses Skalarprodukts ist in Abb. 5.15 skizziert. Die Integration in eine Hardwareumgebung wie das UoH HTX-Board ist nicht dargestellt, muss aber selbstverständlich zusätzlich geleistet werden. Sie wird später betrachtet.

Bei näherer Betrachtung von Gleichung 5.6 kann man leicht sehen, dass es zur Wiederverwendung von Daten nötig ist, eine der zwei Matrizen zu streamen und die andere anderweitig, beispielsweise von lokal verfügbarem Speicher aus, bereitzustellen. Die Zeile a_i wird dann dazu verwendet, die gesamte Zeile i von C zu berechnen. Daher können also die Zeilen von A gestreamt werden und eine beliebige, aber feste Zahl R von Vektoren der Matrix B gleichzeitig elementweise gelesen werden entsprechend der Spaltenposition in a_i. Entsprechend sind auch R Elemente von C auf einmal fertigberechnet, wenn die $n \cdot R$ Produkte akkumuliert sind. Es wäre nicht sonderlich sinnvoll, mehr als ein Element eines Spaltenvektors b_j auf einmal zu lesen. Zum Erzielen von Beschleunigung wird also vorgeschlagen, mehrere exakte Akkumulatoreinheiten gleichzeitig zu betreiben; je eine für eine Spalte b_j der R gleichzeitig verwendeten Spalten. Dieses Schema ist in Abb. 5.16 skizziert.

Dieser Ansatz kann dahingehend erweitert werden, auch mehrere Zeilen a_i, a_{i+1}, $\ldots a_{i+S-1}$, $S \geq 1$, gleichzeitig zu streamen, so ausreichend Bandbreite vorhanden ist, wie in Abb. 5.17 illustriert ist.

[4] engl. dot product oder inner product

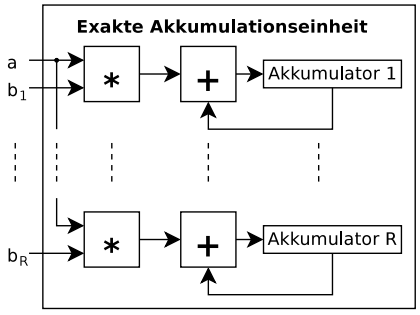


Abbildung 5.16: Parallele Architektur mit R Akkumulatoren.

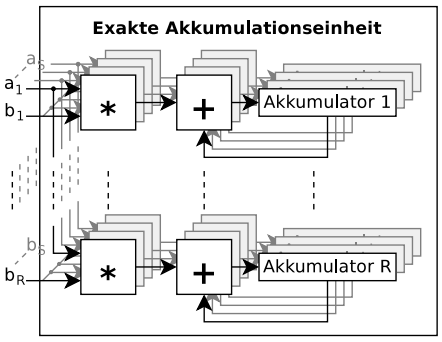


Abbildung 5.17: Zweidimensional mit $R * S$ Akkumulatoren.

Dabei sind also die Vektoren b_j die gleichen für eine vektorartige Berechnung von mehreren Spalten der Ergebnismatrix mit mehreren Zeilen a_i.

Im ersten Koprozessorentwurf aus Abschnitt 5.4 wurden die Werte von A und B einzeln gesendet oder höchstens als Tupel $(a_{i,k}, b_{j,k})$, wodurch zumindest kein Aufwand für die Datenhaltung oder -organisation entstand. Werden mehrere Vektoren mittels hiesigen parallelisierten Ansatzes multipliziert, so erfolgt der Zugriff auf B immer nur innerhalb der R aktuellen Spalten, also nicht fortlaufend über die Matrix hinweg. Daher sollte B derart im Speicher abgelegt werden, dass aufeinanderfolgende Speicherzugriffe genau die benötigte Reihenfolge wiedergeben bzw. angepasst an die speicherinterne Struktur aus Bänken. Ist $R == o$, so kann die Reihenfolge natürlich prinzipiell beibehalten werden.

Werden mehrere Zeilen von A zeitgleich gestreamt wie in Abb. 5.17, so sollte A transponiert vorliegen, um die Daten rechtzeitig an die Streams zur Verfügung stellen zu können, wobei analog zu obigem die Blockgröße S auch zu berücksichtigen ist.

5.5.2 Eine Hardwarearchitektur auf dem UoH HTX-Board

Wie im vorigen Abschnitt gesehen, sollten zur parallelen Berechnung mehrerer Skalarprodukte die Daten wiederverwendet werden und Daten aus möglichst mehreren Quellen zeitgleich verfügbar gemacht werden. Für die folgenden Überlegungen wird das UoH HTX-Board (Abschnitt 2.1.4 zugrundegelegt, das über zwei DDR2-Speicher verfügt. Diese Speicher können zur Bereitstellung mehrerer Spaltenwerte von B genutzt werden. Der dazu notwendige DDR-Controller kann mit bis zu 266 MHz getaktet werden. Entsprechend wird dann die Nutzerlogik mit bis zu 133 MHz getaktet. Ein möglicher Systementwurf ist in Abb. 5.18 vorgestellt, wobei die Taktung auf 130 MHz bzw. 260 MHz verringert ist, da ersten Untersuchungen zufolge eine verbesserte Akkumulationseinheit noch mit 130 MHz zu betreiben sein könnte. Der HyperTransport-Anschluss kann mit entweder 200 MHz oder 400 MHz betrieben werden. Der Bus kann mehr ausgelastet werden, wenn zu der 400-MHz-Geschwindigkeit passend die Nutzerlogik mit 200 MHz betrieben wird, der kritische Akkumulatorkern mit 100 MHz und entsprechend die DDR-Schnittstelle mit nur 200 MHz. Dazu müssen zusätzliche Puffer eingebracht werden, welche die unterschiedlichen Taktraten miteinander in Einklang zu bringen vermögen. Derart kann die Systemanbindung sowie die Hardwarearchitektur des HTX-Boards besser im Zusammenspiel ausgenutzt werden.

5.5.3 Implementierung

Im Folgenden werden die erzielbaren Taktraten sowie spezifische Probleme und weitere Abhängigkeiten beim Implementieren der exakten Einheiten und der Integration selbiger in eine komplette Hardware-Architektur diskutiert.

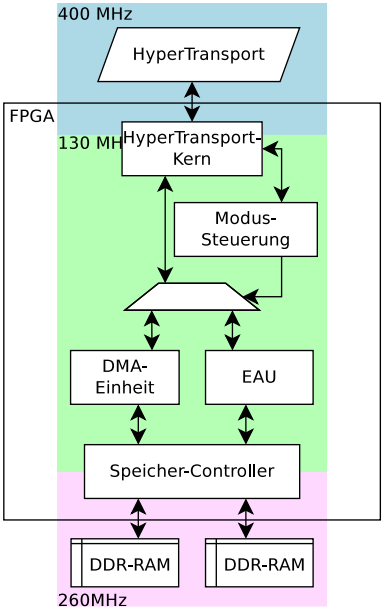


Abbildung 5.18: Integration einer EAU in das UoH HTX-Board.

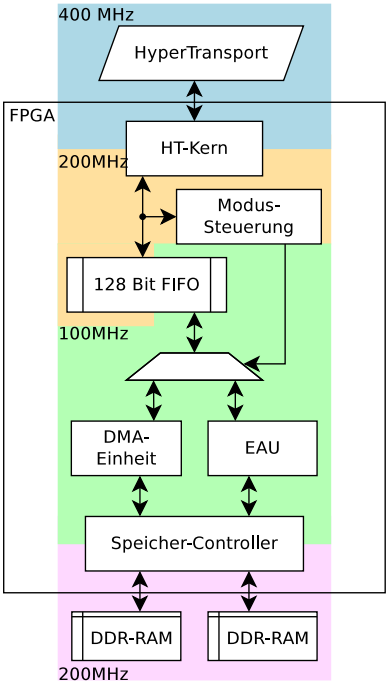


Abbildung 5.19: Verbesserte Integration mit nötiger Zwischenspeicherung.

Ressource	Verwendung	Prozentsatz
Slices	1.946	4%
Flip-Flops	2.669	3%
4-Eingaben-LUTs	2.874	3%
DSP48s	16	10%

Tabelle 5.13: Ressourcenbedarf der Multiplikationseinheit nach der Synthese.

Multipliziereinheit

Die Einheit zur Berechnung des Produkts zweier Komponenten, wie in Abb. 5.15-5.17 jeweils links zu sehen, ist für IEEE-754-konforme Operanden entworfen und gibt konforme Ergebnisse aus. Allerdings muss der Einfachheit der Einheit halber davon ausgegangen werden, dass keine Werte wie `Infinity`, `Not a Number` zu behandeln sind, da die Daten von Software aus gesendet werden, wo vorab die notwendigen Überprüfungen und Fehlerbehandlungen erfolgt sein sollten. Andererseits können manche Sonderwerte aufgrund des geschickten Designs des Standards IEEE-754 auch einfach durchgereicht werden, da z.B. eine Multiplikation des Werts `Infinity` mit einer Zahl wieder `Infinity` ergibt. Die Einheit selbst gibt aus, ob die Verwendung der Operanden Sonderfälle erzeugt hat. Im Unterschied zum Standard ist die Mantisse des Ergebnisses allerdings 106 Bits breit, um keine Genauigkeit zu verlieren.

Besonderer Wert beim Entwurf und der späteren Implementierung wurde auf möglichst hohe erzielbare Taktraten gelegt. Dazu wurde die Multiplikation der je 53 Mantissenbits miteinander in 14 verschiedene Pipeline-Stufen zerlegt. Dadurch ließ sich bei der in Tab. 5.13 angegebenen Ressourcenverwendung eine maximale Taktrate von 176 MHz für den Xilinx Virtex-4 FX100-11 FF1152 erzielen, dessen Leistung der auf dem UoH HTX-Board befindliche FX100-10 mittels eines Lüfters erreichen könnte.

Die Implementierung folgt dem Schema in Abb. 5.20: Zuerst werden die beiden Operanden in ihre Bestandteile Charakteristik, aus der später der Exponent errechnet wird, erweiterte 53-Bit-Mantisse und Vorzeichen zerlegt. Die Breite des neuen Exponenten erhöht sich um ein Bit auf zwölf Bits; die Mantisse des Ergebnisses benötigt wie schon erwähnt 106 Bits. Die Mantissen werden zeitgleich in k Blöcke zerlegt, die anschließend zu k^2 Multiplikationen führen. Nichtüberlappende Multiplikationen führt man nach Möglichkeit parallel zueinander aus, damit diese einfach konkateniert

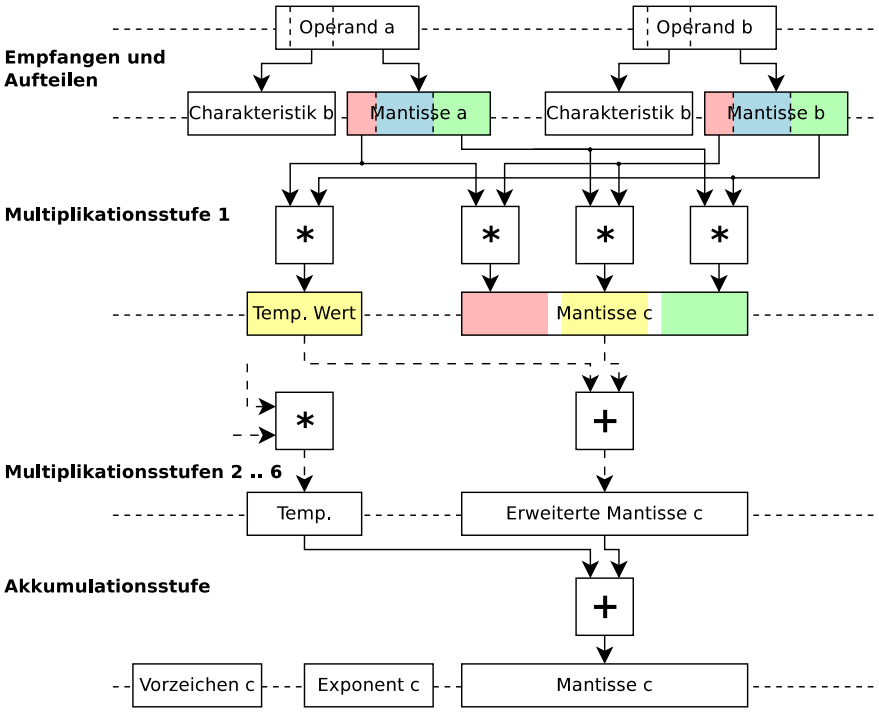


Abbildung 5.20: Multiplizierer für doppelt genaue Operanden.

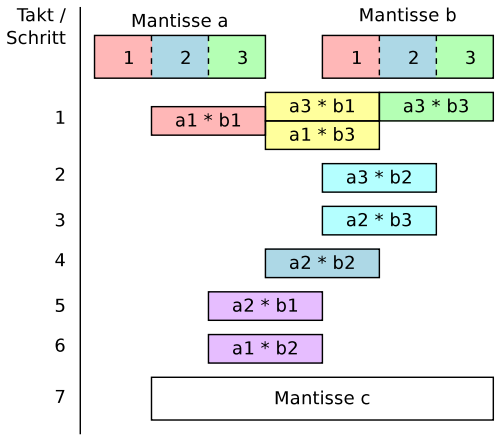


Abbildung 5.21: Pipeline einer Multiplikationseinheit unter Annahme einer DSP-Breite von 19 Bits.

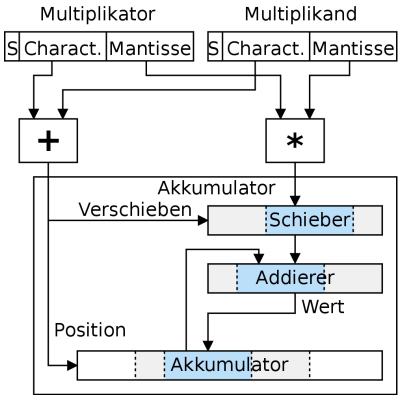


Abbildung 5.22: Ausrichtung und Akkumulation des Produkts an die korrekte Position im breiten Akkumulator.

werden können zu einem Teilergebnis. Angenommen, man kann 19-bit-Multiplizierer verwenden, dann braucht man $k = 3$ Blöcke pro Mantisse. Drei von vier Multiplikationen überlappen sich dann im ersten Schritt nicht. Das vierte, überlappende Produkt darf erst im nächsten Takt akkumuliert werden, während zeitgleich weitere Produkte berechnet werden. Dies wiederholt sich dann für die verbleibenden Teilprodukte, was zu insgesamt sechs Multiplikationsstufen führt. Zum Schluss wird das letzte Produkt akkumuliert, das Vorzeichen und der berechnete Exponent zusammengefügt und ausgegeben. Das Pipelining und der Gesamtablauf über sieben Pipelinestufen hinweg sind in Abb. 5.21 illustriert. Den DSP48-Einheiten auf den verwendeten FPGAs entsprechend sollte man jedoch 18-Bit-Multiplizierer verwenden, was zu $k = 4$ Blöcken mit jeweils Länge 13 oder 14 führt und somit insgesamt 16 Multiplikationen und zwölf Stufen benötigt.

Ressource	Verw.	Prozent
Slices	17.094	40%
Flip-Flops	5.808	6%
4-Eingaben-LUTs	32.481	38%
DSP48s	1	1%

Tabelle 5.14: Ressourcenbedarf der Akkumulationseinheit nach der Synthese.

Ressource	Verw.	Prozent
Slices	17.838	42%
Flip-Flops	8.449	10%
4-Eingaben-LUTs	34.207	40%
DSP48s	16	10%

Tabelle 5.15: Ressourcenbedarf der Multiplikations-Akkumulationseinheit.

Akkumulationseinheit

Das Ergebnis aus der Multiplikation bestehend aus Vorzeichen, größerem Exponenten und doppelt so großer Mantisse mit seinen insgesamt $1 + 12 + 106 = 119$ Bits wird an die Akkumulationseinheit weitergereicht, wo es ausgerichtet wird und im passenden Block akkumuliert wird, wie in Abb. 5.22 zu sehen. Dabei geht keine Genauigkeit verloren, da das Akkumulatorregister mit $64*67 = 4288$ Bits breit genug gewählt ist, sowohl das Produkt der betragsmäßig größten als auch kleinsten Zahlen aufzunehmen, und obendrein über zusätzliche Überlaufbits verfügt [177]. Beim Akkumulieren des Produkts kann ein Übertrag auftreten, der in höheren Blöcken aufgenommen werden muss.

Die Akkumulationseinheit ist als Pipeline aufgebaut. Die Daten werden in mehreren Schritten akkumuliert auf die 64-Bit-Blöcke des Akkumulatorregisters. Zusätzlich werden nebenbei Register zur schnellen Auflösung des Übertrags mitgeführt [177]. Allerdings ist zur korrekten Durchführung ein Lese-Schreib-Zyklus atomar zu gestalten, der sich von der Akkumulation des kleinsten Blocks bis hin zur Auflösung des Übertrags zieht. Kulisch hatte dies seinerzeit dadurch umgangen, dass zeitgleich langsam die Daten über die Systemschnittstelle übertragen wurden. Als Alternative wurde eine intern höhere Taktrate vorgeschlagen. Aufgrund der Taktung des verwendeten FPGAs und der Systemschnittstelle sowie der Anwenderlogik mit möglichst hoher Frequenz nahe dem Maximum ist keine Vervielfachung der Taktrate möglich. Ebensowenig ist eine Teilung der weiteren Taktraten sinnvoll. Die Implementierung dieses Zyklus innerhalb eines Takts würde die erzielte Taktrate von etwa 105 MHz massiv reduzieren. Stehen ausreichend viele Hardware-Ressourcen zur Verfügung, so könnte der Übertrag in einem weiteren Speicher mitgeführt werden und dadurch dieser Zyklus zumindest teilweise aufgelöst werden. Allerdings wären dann dieser Speicher und das Akkumulatorregister zum Auslesen erst zusammenzuführen. Um die Taktrate also möglichst hoch und damit den Synthesewerkzeugen auch mehr Spielraum zu lassen, wird nach jedem Additions-/Subtraktionsbefehl ein atomarer Lesen-Modifizieren-Schreiben-Zyklus durchgeführt. Dadurch erhöht sich allerdings der Cycles per Instruction (CPI)-Wert um zwei Takte, so dass insbesondere nur alle drei Takte effektiv eine neue Akkumulation angestoßen werden kann bzw. eine Akkumulation vollendet wird.

Nach der Akkumulation wird im letzten Schritt der Pipeline bestimmt, wo sich im Akkumulator die höchstwertigen Daten befinden. Von dieser Stelle werden die zu konvertierenden 53 Bits sowie ggf. zusätzliche sogenannte *Guard-Bits* ausgelesen und als doppelt genauer Gleitkommawert ausgegeben. Die Ressourcenverwendung der Akkumulationseinheit allein ist in Tabelle 5.14 zusammengestellt; die der MAC-EAU in Tabelle 5.15. Entsprechend der Aufteilung der Mantisse in $k = 4$ Teilblöcke sind $k^2 = 16$ DSP-Einheiten vom Synthesewerkzeug korrekt eingesetzt worden.

Kommunikation und Steuerung

Der Akkumulator erhält nicht nur die Mantisse und den Exponenten, sondern auch eine Identifikationsnummer zum Operationscode dazu, mittels derer der Wert des Akkumulators genau nach Durchführung der Operation erkannt werden kann. Diese Identifikationsnummer kann im Rahmen der Adressierung und der Speichereinbindung mitgegeben werden. Zudem muss angegeben werden, ob der anliegende Befehl an der jeweiligen Einheit bzw. Stufe gerade gültig ist. Dadurch kann die MAC-EAU auch für gewöhnliche Akkumulationen genutzt werden. Die Operation `CLEAR` setzt den internen Zustand zurück sowie den Akkumulator selbst; der Akkumulator kann anschließend für die nächste Reihe an exakten MAC-Operationen, z.B. für ein neues Skalarprodukt, verwendet werden.

Der andere Teil der Einheitensteuerung besteht aus der Rückmeldung der Einheit an den Dateneingang über die Bereitschaft zum Datenempfang (ready for data, `rfd`), da nur alle drei Takte eine Operation begonnen werden kann aufgrund der Ressourcenabhängigkeit beim Beschreiben des Akkumulators. Ist `rfd=0` gesetzt, so wird ein eventuell bereits am Eingang angelegtes Datum zwischengespeichert und direkt im Anschluss an die aktuelle Operation verarbeitet, bevor die nächste neue Operationen angenommen (bei `rfd=0`) und ausgeführt werden kann.

Überlegungen zu Datenübertragung und -speicherung

Vorab dem Start müssen die häufig wiederzuverwendenden Daten möglichst nahe zum FPGA gebracht werden. Ein DDR-Speicher wie auf dem HTX-Board stellt in der Regel eine Möglichkeit zur FPGA-nahen Speicherung dar. Zwar sei der DDR-Speicher langsamer von Latenz und Durchsatz her als die Nutzung des Hauptspeichers über HyperTransport, es sollen jedoch möglichst viele Daten gleichzeitig erhalten werden und entsprechend mehrere Datentransfers zeitgleich von und mit verschiedenen Quellen bzw. Senken geschehen. Dazu wird beispielsweise die rechte Matrix B in den DDR-Speicher übertragen. Ist die Matrix übertragen, so kann die eigentliche Operation beginnen: Die Daten der Matrix A werden dann über HyperTransport mittels DMA vom Hostsystem zum HTX-Board gestreamt, wo sie dann mit den Daten der Matrix B aus dem board-eigenen DDR-Speicher multipliziert und akkumuliert werden entsprechend der vorangegangen Überlegungen. Das Ergebnis aus dem Akkumulator muss abschließend ins doppelt genaue Gleitkommaformat konvertiert und zum Hostsystem übertragen werden, wo es weitergenutzt werden kann.

Aufgrund dieser Vielzahl an unterschiedlichen nötigen Aufgaben, die von dem Gesamtdesign geleistet werden müssen, ist eine interne Steuerung nötig, die zwischen verschiedenen Zuständen für das Beschreiben der Matrix B, für das Empfangen von A über HyperTransport bei zeitgleichem Lesen von B aus dem DDR-Speicher und der Multiplikation-Akkumulation sowie dem Rückschreiben des Ergebnisses zu unterscheiden vermag. Ferner müssen vier unterschiedliche Operationen, Akkumulieren, subtraktives Akkumulieren, Multiplizieren-Akkumulieren und subtraktives Multiplizieren-Akkumulieren unterstützt werden, wobei bereits abzusehen ist, dass weitere Operationen folgen werden.

Um Platz auf dem FPGA zu sparen, wird das Übertragen der Matrizen B und A bevorzugterweise der Anwendung auferlegt. Diese kann nämlich nach dem Mitteilen einer mittelgranularen Operation wie dem Skalarprodukt die Operanden selbst übertragen und gegebenenfalls für eine Umsortierung entsprechend den Blockgrößen R und S sorgen. Dazu lässt sich hervorragend die Einbindung des HTX-Boards in den virtuellen Speicherraum verwenden.

Das Ergebnis eines Skalarprodukts könnte explizit auf Anfrage über HyperTransport und die Posted-Queue als Antwort gesendet werden. Dies lässt sich mit dem Pipeline-Design des Akkumulators allerdings schlecht vereinbaren. Alternativ kann wie bereits vorgestellt einfach nach jeder Operation automatisch im Rahmen der Pipeline-Implementierung der aktuelle Wert konvertiert und ausgegeben werden. Dies scheint insbesondere unter Verwendung des Konzepts der Identifikationsnummern ein guter Ansatz. Bereits drei EAUs reichen dabei aus, die verfügbare Schreibbandbreite auszulasten, da jede EAU alle drei Takte ein 64-Bit-Datum ausgeben kann. Bei jeder der EAUs wird jedoch nur der zuletzt berechnete Wert benötigt. Daher sind diese zwei Ansätze dahingehend zu kombinieren, dass der automatisch von den EAUs ausgegebene Wert nur dann übertragen wird, wenn er vollständig berechnet ist und eine Leseanfrage mit der passenden Identifikationsnummer vorliegt. Derart wird die Schreibbandbreite nicht unnötig ausgelastet. Es bleibt noch die Notwendigkeit, die Akkumlationseinheit während der Konvertierung anzuhalten, damit das Ergebnis nicht von nachfolgenden Operationen in der Pipeline verändert ist. Mit bereits vier Bits für die Identifikationsnummer lassen sich die unterschiedlichen Befehle und eingehenden Daten hervorragend voneinander separieren. Der Lesebefehl muss dann genau die zu den letzten Operanden zugehörige Identifikationsnummer verwenden. Aufgrund vorangegangener Erfahrungen kann die Zahl an exakten Einheiten auf 16 beschränkt werden [NBKK09]. Zur Adressierung benötigt man also vier weitere Bits in den Adressen bzw. 16 eigene Speicherseiten. Dieses Schema lässt sich noch erweitern und vereinfachen, indem anhand einer für die letzten Operanden zuvor festgelegten Identifikationsnummer automatisiert der finale Wert über die DMA-Einheit geschrieben wird.

Gesamtentwurf

Wie erkennbar wurde, kann die EAU nur einen Befehl alle drei Takte bearbeiten, damit keine Lese-Schreib-Konflikte auftreten; der CPI-Wert beträgt also 3. Die Multiplizierer sind jedoch vollständig gepipelinet. Daher kann ein Multiplizierer drei Akkumulationseinheiten bedienen. Werden wie in Abb. 5.23 drei Konglomerate aus je einem Multiplizierer und drei EAUs verwendet, so können mit drei aus dem DDR-Speicher gelesenen Werten von B sowie mit in jedem Takt erhaltenen Wert von A neun Akkumulationseinheiten betrieben werden und ein Durchsatz von 300 M exakten MAC-Operationen pro Sekunde (eMACs/s) erzielt werden bei einer Taktfrequenz von 100 MHz für das Akkumulatordesign. Wie die Ergebnisse des Mappings von einer bzw. zwei MAC-EAUs in Tabelle 5.16 zeigen, ist auf dem HTX-Board jedoch kaum Platz für drei solche Einheiten.

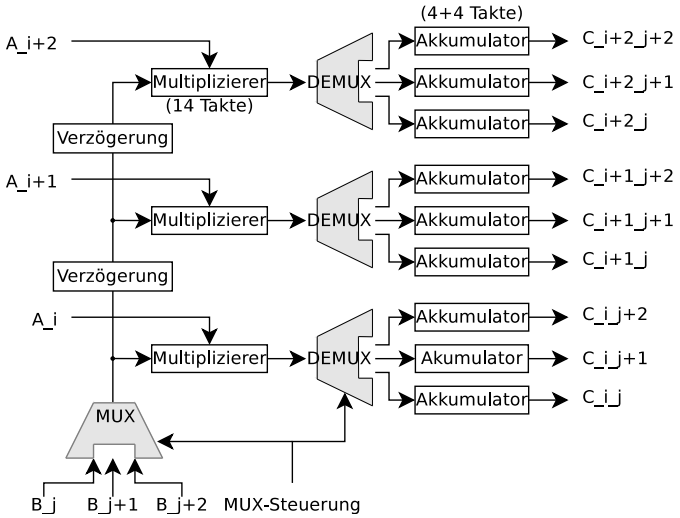


Abbildung 5.23: Demultiplexen der Produkte von drei Multiplizierern an jeweils drei Akkumulationseinheiten.

	1 Mult., 1 Akku		2 Mult., 2 Akkus	
Ressource	**Verw.**	**Prozent**	**Verw.**	**Prozent**
Belegte Slices	19.973	47%	36.315	86%
Mit verwandter Logik	19.973	100%	36.315	10%
Mit nicht verw. Logik	0	0%	0	0%
Gesamtzahl 4-Eingaben-LUTs	34.336	40%	68.892	81%
Für Logik	33.701		61.480	
Für Routing	551		7.248	
Als Schieberegister	84		184	
BUFG/BUFGCTRLs	1	3%	2	6%
Davon als BUFGs	1		2	
DSP48-Einheiten	16	10%	32	20%

Tabelle 5.16: Ressourcenbedarf der Multiplikations- und Akkumulationseinheiten auf dem Virtex-4 FX100 nach dem Mapping.

5.5.4 Evaluation

Aufgrund der massiven Ressourcennutzung (vgl. Tabellen 5.13-5.16) des vorgeschlagenen Systems kann die Evaluation nur als Leistungsabschätzung und Technologieskalierung erfolgen.

Die Berechnung der Gesamtausführungszeit wird anhand einer Formel vorgenommen, welche die Übertragungsdauer und den Aufwand für die Berechnung berücksichtigt. Anhand dieser in Tabelle 5.17 aufgeführten Formel lassen sich die Ausführungszeiten der Matrixmultiplikation für unterschiedlich große Matrizen errechnen. Zur Ausführung muss zuerst die eine Matrix in den boardlokalen Speicher übertragen werden, dann wird die andere Matrix gestreamt und währenddessen die

Transfer zu RAM	$n * o * 8\text{B}/bw$
Transfer vom RAM	$m * o * 8\text{B}/bw$
Berechnung	$m * n * o/tp$
Gesamtzeit	$(m+n) * o * 8\text{B}/bw + m * n * o/tp = (2m + m * n + 2n) * o/tp$
130-MHz-Design	$(2m + m * n + 2n) * o/260M/s$
100-MHz-Design	$(2m + m * n + 2n) * o/400M/s$

Tabelle 5.17: Übersicht der verschiedenen Komponenten und ihres Einflusses auf die maximal erzielbare Ausnutzung der Bandbreite in Abhängigkeit der Matrixdimensionen n, m, o, des Durchsatzes tp und der Hyper-Tranport-Bandbreite bw.

Dimension	**130 MHz**	**100 MHz**
1000	3,86 s	2,51 s
4000	246 s	160 s
8000	3847 s	2501 s

Tabelle 5.18: Bestmögliche Ausführungszeiten zweier Architekturen für exakte Arithmetik auf dem HTX-Board.

Skalarprodukte berechnet. Das Rückschreiben der Werte erfolgt ebenso überlagert mit der Berechnung. Die Zeitspanne zwischen Vollendung des letzten Akkumulationsbefehls und dem Schreiben des letzten Datums hat vernachlässigbare Auswirkungen auf die Ausführungszeit.

Aus Tabelle 5.18 wird ersichtlich, dass die 100-MHz-Variante das zu favorisierende Design ist. Für die dort aufgeführte Matrixdimension $m = n = o = 8000$ ergibt sich lediglich ein Speicherbedarf von 512 MB je Matrix, was für das HTX-Board eine mögliche Konfiguration mit zwei 256 MB großen Speicherchips darstellt. Dabei ist ferner zu beachten, dass beim 100-MHz-Design die erzielbare Leistung sehr stark vom Streaming der Daten abhängt und so weit mehr Anspruch an die Handhabung und Implementierung des Datentransports stellt.

Aufgrund der Ressourceneinschränkungen auf dem ausgewählten FPGA, insbesondere bei Verwendung des HT-Cores mit 20 % bis 40 % Ressourcenbedarf, passen lediglich zwei Akkumulationseinheiten mit nur einem Multiplizierer auf den FPGA. Somit sind nur noch zwei exakte MAC-Operationen in drei Takten durchführbar, was bei der 130-MHz-Variante (Abb. 5.18) zu 88M eMACs/s und bei der 100-MHz-Variante (Abb. 5.19) zu 66M eMACs/sec führen würde.

Der Vergleich mit Tabelle 5.2 lässt ersichtlich werden, dass die vorgeschlagene Architektur mit bis zu 400 eMACs/s eine ernsthafte Möglichkeit zur Beschleunigung der Softwarebibliothek C-XSC um Faktor $\frac{400}{5{,}237} = 76,38$ stellt und auch schneller als GMP wäre. Selbst die wegen Ressourcenproblemen weitaus schwächere Variante mit einem Multiplizierer und zwei Akkumulatoren würde eine Beschleunigung von $\frac{88}{5{,}237} = 16,80$ als 130-MHz-Variante liefern.

Die Integration der vorgeschlagenen Architektur auf dem HTX-Board mit Xilinx Virtex-4 FX-100-FPGA ist nicht sonderlich vielversprechend. Anstelle einer vollständigen, mühsamen Implementierung und Evaluation wird daher ein Xilinx Virtex-6 LX240T, wie er z.B. auf dem Entwicklungsboard Xilinx ML605 verfügbar ist, betrachtet. Ähnlich der Implementierung eines exakten Akkumulators von Bierlox auf einer PCI-Karte [31] kann das ML605 über PCI Express angesprochen werden oder alternativ über optisches Gigabit-Ethernet. Auf diesem FPGA lassen sich sechs exakte Skalarprodukte unterbringen, so dass mindestens zwei exakte MAC-Operationen je Takt gestartet und vollendet werden können. Mit der schnelleren Logik und Look-Up-Tabellen mit sechs statt vier Eingängen lassen sich nicht nur höhere Taktraten erzielen, sondern auch Pipelinestufen zusammenlegen und zudem der Platzverbrauch reduzieren, wodurch eventuell noch weit mehr Einheiten untergebracht werden können. Ferner ermöglichen die breiteren DSP-Einheiten, die Länge der Multiplikationspipeline zu verringern auf weniger als zwölf Stufen.

5.5.5 Zusammenfassung der bisherigen Ergebnisse

In diesem Abschnitt wurde ein Konzept vorgestellt und eine Architektur entwickelt, die unter der Annahme hinreichend großer FPGAs imstande ist, bis zu 400 Millionen exakter Multiplikationen-

Akkumulationen pro Sekunde durchzuführen auf einem HyperTransport-basierenden, heterogenen System. Dazu müssen so viele Multiplikations-Akkumulations-Einheiten genutzt werden, dass die verfügbare Bandbreite vollständig ausgenutzt wird. Einschränkungen durch die ausgewählte Hardware erlauben jedoch nur sehr wenige Exemplare an MAC-Einheiten, so dass auf dem UoH HTX-Board nur 66 bis 88 Millionen Operationen pro Sekunde als Maximum erwartbar sind.

Das vorliegende Design der Einheit zur Berechnung eines exakten Skalarprodukts besteht aus einem breiten Festkommaregister und vorgeschalteteten Multiplikationseinheiten. Anhand einer groben Projektion wird erkennbar, dass weitere Studien auf dem UoH HTX-Board mit dem dort vorhandenen, vergleichsweise kleinen und langsamen FPGA aufgrund der Ressourcen- und Hardwareeinschränkungen nicht weiter zielführend sind. Die Architektur lässt sich zwar auf größeren und schnelleren FPGAs umsetzen, etwa mit der Convey HC-1. Die vorherrschende Fragestellung ist jedoch, welche alternativen Implementierungen möglich sind, die volles Pipelining bieten und somit für kleinere FPGAs geeignet sind. Kleinere Einheiten sind einfacher zu verschalten und einfacher in Rahmenwerke zu integrieren, wodurch sie für Anwendungsentwickler bequemer nutzbar werden. Ferner kann der gewonnene Platz auf dem FPGA dann für andere Einheiten genutzt werden, um Vor- oder Nachverarbeitung der Arbeitsdaten durchzuführen und so das Verhältnis von Kommunikation zu Berechnung zu verbessern und letztendlich noch mehr Nutzen aus der Verlagerung in Hardware zu ziehen.

Daher wird im folgenden Abschnitt eine alternative Implementierung für ein exaktes Skalarprodukt beschrieben, welche Mikroprogrammierung einsetzt, um Basiseinheiten so zu verschalten, dass zahlreiche Operationen auf breiten Festkommaregistern durchgeführt werden koennen. Das Skalarprodukt kann dadurch vollständig als Pipeline implementiert und ausgeführt werden.

5.6 Mikroprogrammierte exakte arithmetische Einheit

Aus den Implementierungsergebnissen und Leistungsabschätzungen zu einer Architektur für exakte Matrixmultiplikation im vorigen Abschnitt 5.5 lassen sich folgende Anforderungen an eine exakte Skalarprodukteinheit ableiten: Verwendung doppelt genauer Operanden, Pipelining mit einem CPI-Wert von 1, hohe Taktung, nahtlose und extern gesteuerte Datenversorgung sowie Erweiterbarkeit für eventuell nötige zuätzliche Operationen. Vor allem gilt es jedoch, den Nutzer von der Erzeugung von hardwarenahem Code zu befreien und stattdessen ein alternatives Modell anzubieten, das die komfortable Nutzung der Vorteile rekonfigurierbarer Hardware bietet.

Es kann bei Hardwareschaltungen hilfreich sein, Mikroprogrammierung zu verwenden [74]. Dies spart Platz bei der Implementierung der Zustandsautomaten, lässt aber auch die Unterkomponenten gezielt wiederverwenden, anstatt sie mehrfach anzulegen. Nach außen hin kann unterschiedlich granulare Funktionalität von der Instruktionsebene eines MAC-Befehls bis hin zur algorithmischen Ebene mit einer Matrixmultiplikation angeboten werden, so dass verschiedenerlei Algorithmen und Anwendungen die exakte Einheit auszunutzen vermögen. Der Anwender profitiert dadurch von mehreren unterschiedlichen Funktionalitäten, die allesamt ohne Verwendung einer Hardwarebeschreibungssprache nutzbar sind. Die genauere Zielsetzung ist schließlich, Anwendungen mit hohen Anforderungen an die Genauigkeit, wie etwa das Lanczos-Verfahren (Abschnitt 5.1.2), vollständig zu unterstützen. Die Idee hinter dem Einsatz der Mikroprogrammierung für das exakte Skalarprodukt in rekonfigurierbarer Hardware ist, dass viele verschiedene Unterkomponenten zur Umsetzung der exakten Akkumulation vorhanden sein müssen wie Splitter, unterschiedlich breite Addierer und Multiplizierer (vgl. Abb. 5.8-5.9). Die Multiplikationseinheit und einer der zwei Splitter werden dabei für reguläre Akkumulationen nicht benötigt. Mittels der Mikroprogrammierung lässt sich auch der Befehlssatz erweitern unter Konfiguration eines anderen Datenpfads bzw. nach Hinzunahme anderer Einheiten. Für weitere zukünftige Operationen wie Skalierung des Akkumulators o.ä. müsste nur ein weiterer Multiplizierer hinzugefügt werden, der mittels der Mikroprogrammierung nahezu aufwandslos integrierbar ist. So lassen sich effektiv viele verschiedene Befehle anbieten, welche nur die unbedingt benötigten Einheiten verwenden und dadurch energieeffizient ablaufen. Zur Implementierung einer mikroprogrammierten Pipeline, die leicht zu erweitern sein soll, bietet sich die horizontale Mikroprogrammierung an (Abb. 5.24) anstelle der vertikalen Mikroprogrammierung. Ein von außen eingehender Befehl führt dann zur Ausführung eines Mikroprogramms, bei dem jeder einzelne Befehl zeitgleich alle Einheiten anzusprechen vermag. Dennoch werden mehrere

Mikrobefehle hintereinander benötigt, um eine Funktionalität wie eine exakte Akkumulation eines Produkts auszuführen.

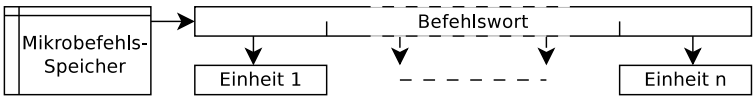


Abbildung 5.24: Horizontale Mikroprogrammierung zur Steuerung von bis zu n Einheiten.

Eine solche mikroprogrammierte Einheit für ein exaktes Skalarprodukt, das den oben formulierten Anforderungen genügt, wurde entworfen und implementiert [Ham10]. Dabei wurde gegenüber den exakten Skalarprodukteinheiten aus den Abschnitten 5.4 und 5.5 ein gänzlich anderer Ansatz verfolgt, nämlich die Auflösung des Lese-Schreib-Problems durch Hinzunahme eines weiteren Akkumulatorspeichers für die auftretenden Überträge. Die Einheit soll auch in ein datengetriebenes, mikroprogrammierbares Steuerwerk integrierbar sein, um mehrfach gleichzeitig unterzubringen und nutzbar zu sein sowie um noch mehr Nutzen durch die Einbindung in größere Kontexte bieten zu können. Das mikroprogrammierbare Steuerwerk wird später vorgestellt im Rahmen der Untersuchungen zu Ansteuerungen von FPGAs (Kapitel 6).

5.6.1 Datenaustausch und Synchronisation

Zum Austausch von Daten zwischen Stufen in einer Pipeline, die mittels unterschiedlicher Einheiten implementiert sind, werden Pipelineregister eingesetzt und besondere Verzögerungseinheiten. Solche Verzögerungseinheiten dienen dazu, bereits früher berechnete Teilergebnisse später zeitgleich zu anderen Ergebnissen zur Verfügung zu stellen.

Sie werden als reguläre Funktionseinheiten zur Verfügung gestellt, die als Teil des Mikrobefehls angesteuert und ferner auch hinsichtlich der Anzahl zu verzögernder Takte parametriert werden. Ihr Entwurf ist in Abb. 5.25 skizziert. Ein Beispiel ist in Abb. 5.26 mit der Verzögerung der schnell errechneten Summe der Exponenten gegenüber des aufwendiger zu berechnenden Produkts der Mantissen gegeben.

Damit die gesamte Einheit sehr flexibel hinsichtlich Erweiterbarkeit ist, erhält jede Einheit eine Reihe an Eingängen, die von unterschiedlichen Einheiten fest belegt sind, wie in Abb. 5.27 dargestellt. Die generisch gehaltene Schnittstelle zu den Einheiten enthält die ausgehenden Daten mitsamt Gültigkeitsanzeige aller passenden und sinnvollen Erzeugereinheiten. Die Datentypen der Schnittstelle sind in Form von Records der 64-Bit-basierte `data_type` und der 4*64-Bit-basierte `data4_type`, der im Wesentlichen ein Array von vier Blöcken des ersteren ist. Neben den reinen Daten sind Statusanzeigen enthalten. Überflüssige Leitungen, die von der Hardware nicht genutzt würden, entfernt das Synthesewerkzeug teilweise selbständig. Ein Teil des Opcodes jeder Einheit steuert den nötigen Multiplexer, um einen bzw. zwei der Eingänge auszuwählen. Die Anzahl an Verbindungen mit anderen Einheiten darf nicht überhand nehmen, da aufgrund der mikroprogrammgesteuerten Verwendung das Synthesewerkzeug nicht alle unbenötigten Leitungen wegoptimieren kann. Jede Einheit kann eine unterschiedliche Belegung der verfügbaren Eingänge haben, um dadurch die Anzahl an Verbindungen auf das Nötigste zu reduzieren. Der `data4_type` ist wichtig bei allen auf dem Akkumulator agierenden Operationen, wie später ersichtlich wird. Mehrere

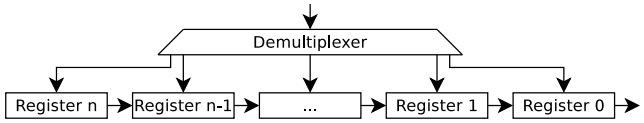


Abbildung 5.25: Aufbau der Verzögerungseinheit als Reihe von Registern mit Demultiplexer zum passenden Initialisieren für beliebig lange Verzögerungen $\leq n$.

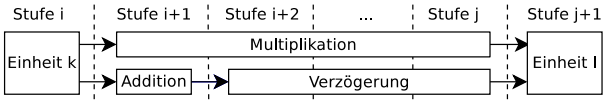


Abbildung 5.26: Verzögerung zum Ausgleich unterschiedlich langer Ausführungszeiten der Einheiten bei der Multiplikation zweier Gleitkommazahlen.

dieser beiden Records machen dann die Schnittstelle zu einer Einheit aus. Es ist dabei Aufgabe des Entwicklers der Mikroprogramme, dafür zu sorgen, dass auch gültige Daten anliegen. Der Anwendungsentwickler ruft über die zur Verfügung stehenden Systemschnittstellen oder Hardware-Rahmenwerke die Mikroprogramme auf und ist somit von der Erstellung der Hardwarebeschleuniger vollständig entbunden.

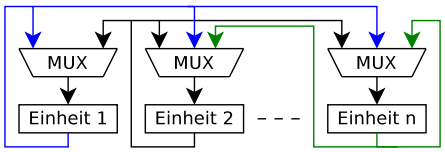


Abbildung 5.27: Ausgangsports der Einheiten werden ausgewählten Einheiten zur Verfügung gestellt und auf den Eingang gemultiplext.

Um die Anzahl an direkten Verbindungen zwischen den Einheiten etwas zu reduzieren, sind zusätzliche Multiplexereinheiten hinzugefügt, die einen von mehreren Eingängen entsprechend dem aktuell anliegenden Befehl bzw. Opcode an die Eingänge anderer Einheiten weiterreichen. Die ausschließliche Verwendung von (meistens zwei) Multiplexern für die Eingänge der Einheiten wäre viel zu aufwendig, da jeder Multiplexer angesteuert werden müsste über das Befehlswort, was sich dadurch verlängerte und auch die Programmierung zusätzlich erschwerte. Somit ist ein guter Mittelwert zwischen Flexibilität und direkter Verdrahtung gefunden worden, der in weiteren Arbeiten bei Bedarf system- und anwendungsspezifisch angepasst werden kann.

5.6.2 Arithmetische und funktionale Einheiten

Aufgabe der mikroprogrammierten Einheit für ein exaktes Skalarprodukt ist die genaue Multiplikation zweier Operanden aus dem IEE-754-Format. Dazu müssen die verschiedenen Teile aus den 64-Bit-Daten innerhalb eines Splitters herausgelöst werden, passend miteinander addiert bzw. multipliziert werden und abschließend an die korrekte Stelle im breiten Festkommaregister geschrieben werden.

Entsprechend dem Typ der beim Splitter eingehenden Daten ist die Aufsplittung in Vorzeichen, Exponent und Mantisse oder aber das Setzen des Exponenten auf Null und Ausgabe des Eingangswerts als Mantisse vorzunehmen. Damit sind bereits wesentliche Voraussetzungen dafür gegeben, das Akkumulatorregister mit Integer-Werten und ebenso Gleitkommawerten zu skalieren. Ein 64-Bit-Addierer dient dazu, die Exponenten der Operanden einer Multiplikation zu addieren, Akkumulatorregisterblöcke zu addieren beim Auflösen des Übertrags oder zwei exakte Werte aus den Akkumulatorregistern zu addieren. Die zum Lesen und Schreiben des Akkumulatorregisters korrekte Position wird anhand des eingehenden Exponentenwerts mittels folgender Berechnungsvorschrift bestimmt: Verschiebung $= \lfloor \frac{\text{Charakteristik}+1022}{64} \rfloor$. Die in das Akkumulatorregister zu schreibenden Daten müssen an den Blockgrenzen ausgerichtet werden. Dies erfolgt anhand des Exponenten und benötigt dazu die unteren sechs Bits gemäß folgender Berechnungsvorschrift bei einem doppelt genauen Operanden: Verschiebung $= \lfloor (\text{Charakteristik} + 1022) \mod 64 \rfloor$. Für andere Formate wird die Vorschrift entsprechend angepasst. Die Kombination aus Adressberechnung und Verschiebung ist in Abb. 5.28 dargestellt.

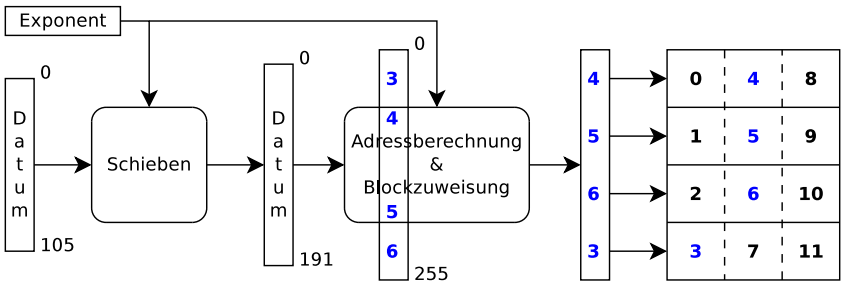


Abbildung 5.28: Verschieben und Ausrichten des Operanden passend zur parallelen Akkumulation auf vier nebeneinanderliegende Akkumulatorregister.

Zwei Dinge hatten sich bei dem vorherigen Entwurf als negativ erwiesen. Einerseits bringt die „schnelle“ Übertragsauflösung eine echte Datenabhängigkeit mit sich, deren Konflikt nur schwer zu beheben ist im Rahmen der zur Verfügung stehenden Hardware. Andererseits brachte das kontinuierliche Konvertieren sowohl eine weitere echte Abhängigkeit mit als auch noch vier zusätzliche Pipeline-Stufen. Zur Behebung des Datenkonflikts bei der Übertragsauflösung wird auf die schnelle Auflösung verzichtet und anstelle dessen ein separates Überlaufsregister eingesetzt (Abb. 5.29). Zusätzlich ist es nötig, aus dem Akumulatorregister vier 64-Bit-Register gleichzeitig auszulesen und zu schreiben, um das innerhalb von 192 Bits ausgerichtete Ergebnis der Multiplikation der zwei 53-Bit-Mantissen akkumulieren zu können (Abb. 5.28 und 5.30). Das Akkumulatorregister ist daher aus vier Zweiport-Speichern aufgebaut, deren Einträge mittels eines intern vierfach parallelen Addierers zeitgleich mit dem 4*64 Bits großen passend ausgerichteten Wert addiert und zurückgeschrieben werden. Der zwischen Lesen und Schreiben der modifizierten Daten bestehende echte Datenkonflikt wird mittels Forwarding aufgehoben (Abb. 5.31).

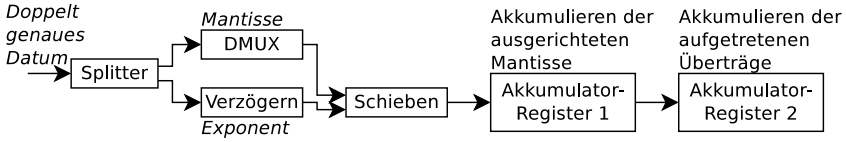


Abbildung 5.29: Akkumulation eines Datums und Behandlung der auftretenden Überläufe mittels eines zweiten Akkumulatorregisters.

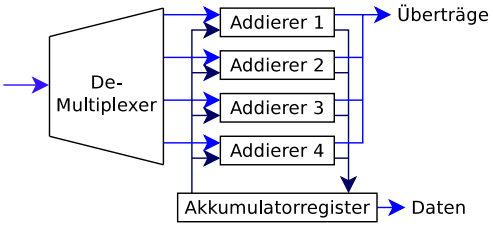


Abbildung 5.30: Verteilung eines eingehenden Vierfach-Datentyps auf vier separate Addierer, die parallel auf dem viergeteilten Akkumulatorregister arbeiten.

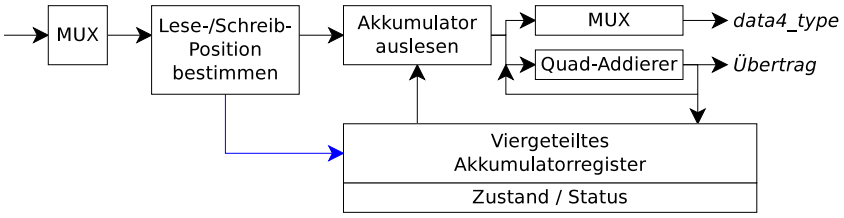


Abbildung 5.31: Involvierte Einheiten zur Akkumulation eines Operanden als Pipeline mit Durchsatz von 1 unter Verwendung von Result Forwarding.

Zur Berechnung eines exakten Skalarprodukts müssen die beiden 53 Bits langen Mantissen der Operanden miteinander multipliziert werden. Um den exakten Akkumulator mittels eines doppelt genauen Operanden zu skalieren, müssen die 64-Bit-Blöcke des Akkumulatorregisters mit der 53-Bit-Mantisse multipliziert werden. Bei der Multiplikation zweier Akkumulatorregister miteinander müssen gar je 64 Bits miteinander multipliziert werden. Daher erhält das Design eine 64x64-Bit-Multiplikationseinheit.

Eine sogenannte Konvertierungseinheit dient zum einen zur Zusammenführung des eigentlichen Akkumulatorregisters mit dem Überlaufregister sowie zum anderen zur Ausgabe konvertierter Daten, wie etwa Ausgabe als doppelt genaue Gleitkommazahl. Um dies zu erreichen, wird zunächst das Akkumulatorregister und einen Takt versetzt das Übertragsregister ausgelesen, miteinander addiert und das Resultat ins Akkumulatorregister zurückgeschrieben. Die dabei auftretenden Überträge werden berücksichtigt. Die höchstwertigen 53 bzw. 24 Bits sowie die zugehörige Blocknummer ermöglichen die sofortige Konvertierung zu einem doppelt bzw. einfach genauen Wert sowie die Ausgabe von Null oder Unendlich, wenn alle Bits bis zum allerersten ungesetzt bzw. gesetzt sind.

	Ohne Rahmenwerk		Mit Rahmenwerk	
Ressource	**Bedarf**	**Anteil**	**Bedarf**	**Anteil**
Slices	11.127	26 %	22.531	53 %
Flipflops	11.710	13 %	21.793	25 %
Look-Up-Tabellen	17.515	20 %	38.324	45 %
Frequenz (in MHz)	207,125	96,5 %	207,125	96,5 %

Tabelle 5.19: Ressourcenverbrauch der exakten arithmetischen Einheit ohne und mit Rahmenwerk zur Ansteuerung für das HTX-Board nach der Synthese.

Ist das Register negativ, so wird dabei automatisch das Zweierkomplement gebildet für die Ausgabe einer vorzeichenbehafteten Zahl. Einen besonderen Multiplexer stellt die Ausgabeeinheit aufgrund der Verwendung am Ausgangsport der gesamten Einheit dar.

5.6.3 Implementierungen

Implementierung auf dem UoH HTX-Board

Nach Entwicklung, Implementierung und Test der Komponenten sowie der Verschaltung zu einer gesamten exakten arithmetischen Einheit ließen sich die in Tab. 5.19 aufgeführten, vielversprechenden Syntheseergebnisse nach aufwendigem Explorieren des Entwurfsraums der Syntheseeinstellungen wie Verwendung gemeinsamer Ressourcen, Entfernen äquivalenter Register, Registerduplikation und Registerbalanzierung erzielen. Mittels des verwendeten Xilinx ISE 10.1 war keine erfolgreiche Platzierung weder mit der schnellen Einstellung des Nutzerdesigns auf 200 MHz (HT400-Standard) noch mit der langsamen Einstellung auf 100 MHz (HT200-Standard) für das HTX-Board erzielbar, da der gesamte Flächenverbrauch mitsamt einem benötigten Rahmenwerk bereits zu groß war, als dass der HT-Core im Rahmen seiner Anforderungen von etwa 40 % der Ressourcen auch noch Platz gehabt hätte. Daher wird später nur eine modellbasierte Leistungsanalyse für das UoH HTX-Board vorgenommen. Für die Convey HC-1 liegt hingegen eine vollständige, lauffähige Portierung vor.

Portierung der mikroprogrammierten exakten Skalarprodukteinheit auf die Convey HC-1

Da die Grenzen des Xilinx Virtex-4 FX100 auf dem UoH HTX-Board bereits mit Rahmenwerken zur Ansteuerung erreicht werden, wird zur weiteren Evaluation die Convey HC-1 mit Xilinx Virtex-5 LX330 verwendet [Kna12]. Dass überhaupt von einer Zielplattform auf eine andere gewechselt werden kann, belegt bereits die Portierbarkeit der Implementierung und des gesamten Ansatzes, domänenspezifische Funktionalitäten innerhalb von Rahmenwerken anzubieten.

Über die Koprozessorschnittstelle wird pro Application Engine nur eine exakte Skalarprodukteinheit angesprochen. Selbige enthält zwei exakte Register, von welchen das eine in der Regel zur Behandlung von Überläufen dient. Über die Speicherschnittstelle werden die Operanden von der Personality aus eingelesen. Ebenso kann der konvertierte Wert oder auch die exakte Darstellung in den Koprozessorspeicher geschrieben werden. Zwei Speicherports dienen dem Einlesen zweier Vektoren; ein weiterer Port dem Schreiben des Festkommaregisters in den Koprozessorspeicher. Wie die Syntheseergebnisse in Tabelle 5.20 belegen, sind mit dem Design auch beim Virtex-5-FPGA bereits viele Ressourcen belegt. Die Convey HC-1 sieht für die Nutzerlogik auf den FPGAs eine Taktfrequenz von 150 MHz vor, daher unternimmt das Place&Route-Werkzeug auch keine weiteren Anstrengungen, eine wesentlich höhere Taktrate zu erzielen als die 150,2 MHz. Aufgrund des Ressourcenverbrauchs wurde auf das Einbringen weiterer Einheiten verzichtet. Dies ist aber auch aufgrund der Registerschnittstelle der Convey HC-1 sinnvoll, da sich mehrere Parameter für unterschiedliche Einheiten nicht voneinander trennen lassen, solange nicht unterschiedliche AEs verwendet werden. Mehrere AEs können bereits in der vorliegenden Version unabhängig voneinander über die AE-eigenen, lokalen Register angesteuert werden.

Um die exakte Skalarprodukteinheit überhaupt auf den FPGAs der Convey HC-1 platzieren und routen zu können, waren weitergehende Kenntnisse in der Entwicklung von Hardwareschaltungen nötig. Dies betrifft im Wesentlichen die Verringerung der Verzögerungen auf den längsten Pfaden

Ressource	Bedarf	Anteil
Slices	28.906	55 %
Flipflops	75.914	36 %
Look-Up-Tabellen	72.463	34 %
Frequenz (in MHz)	202,82 (Synthese)	150,20 (Place&Route)

Tabelle 5.20: Ressourcenverbrauch einer Personality mit exakter arithmetischer Einheit auf der Convey HC-1 nach Place&Route.

durch Verbesserung der Logik auf abstrakter Ebene, die Minimierung der Fanouts[5]; das manuelle Balanzieren von Berechnungen über Pipelinestufen hinweg, das Entfernen unnötiger Multiplexer und die Entwurfsraumexploration hinsichtlich der Einstellungen für die Werkzeuge des FPGA-Herstellers. Zudem kamen sogenannte „Hold"-Fehler hinzu, die es zu beseitigen galt. Die Beseitigung selbiger erfolgte nicht durch Verbessern des längsten Pfads, sondern im Gegenteil durch Verlangsamung der zu schnellen Multiplikation zweier 64-Bit-Blöcke in zehn Takten, indem diese nun innerhalb von neun Takten erfolgen muss und dazu kompliziertere Verdrahtung benötigt. Dadurch konnte der Hold-Fehler beseitigt werden. Diese Techniken und Arbeitsaufwände können von Anwendungsentwicklern nicht verlangt werden. Die Entwicklung von Spezialeinheiten in rekonfigurierbarer Hardware muss also von Hardwareexperten übernommen werden.

5.6.4 Evaluation der mikroprogrammierten EAU

Modellbasierte Leistungsanalyse

Das Design wurde für das UoH HTX-Board auf eine Taktung mit 200 MHz und Nutzung des HT400-Standards ausgelegt. Daher wird die modellbasierte Leistungsanalyse auch auf dieser Basis durchgeführt.

Das Zurücksetzen des Akkumulatorregisters auf Null verwendet einen Demultiplexer mit zwei Pipeline-Stufen, einen Shifter mit sechs Stufen und den Akkumulator mit fünf Stufen. Der Datenpfad hat somit eine Länge von 13 Stufen bzw. Takten. Das Mikroprogramm hat einen Prolog von zwei Takten Dauer für den Demultiplexer, dann werden 17 gepipelinete Schreibe-Operationen auf dem Register durchgeführt (das Register ist viergeteilt, jedes Register hat 17 Blöcke), und schließlich muss entsprechend der Berechnungsvorschrift zur Ausführungslänge t einer Folge von n Befehlen in einer k-stufigen Pipeline von $t = n + k - 1$ ein Epilog von $k - 1 = 13 - 1$, also zwölf Takten Dauer ausgeführt werden. Somit ergibt sich eine Gesamtdauer von $2 + 17 + 12 = 31$ Takten bzw. 155 ns.

Beim Multiplizieren-Akkumulieren werden zunächst die beiden Operanden „gesplittet" (ein Takt), dann die Operanden multipliziert (zwölf Takte), zurechtgeschoben (sechs Takte) und im Akkumulatorregister akkumuliert sowie die Überträge aufgenommen (je fünf Takte). 1000 MAC-Operationen können somit in $1000 + 29 - 1 = 1028$ Takten durchgeführt werden, also kann das exakte Skalarprodukt zweier Vektoren mit 1000 Elementen in 5,14 μs berechnet werden. Zum Auslesen des exakt akkumulierten Werts müssen zunächst die zwei Akkumulatorregister zusammengeführt werden, wozu eine 14 Takte lange Pipeline eingesetzt wird, wovon vier Takte zum Auslesen der Register, fünf zum Zusammenführen im Konverter, und fünf weitere zum Schreiben verwendet werden. Der Konvertierer berechnet zeitgleich den doppelt genauen Wert, so dass diese weitere Konvertierung die Dauer nicht erhöht. Die 68 Blöcke der Akkumulatoren müssen sequentiell ausgelesen werden, um die Überträge korrekt aufzurechnen, wodurch sich das Auslesen auf $68 + 14 - 1 = 81$ Takte beläuft, also 405 ns.

Eine exakte Matrixmultiplikation zweier 1000×1000-Matrizen würde somit $10^6 * (155\,\text{ns} + 5.140\,\text{ns} + 405\,\text{ns}) = 5.700\,\text{ms}$ benötigen, wenn die Daten immer in den Eingangspuffern vorliegen und der Ausgangspuffer nie volläuft. Die Akkumulation von 2^{20} Werten wie in Abschnitt 5.5.4 und Tabelle 5.2 würde in 17 Takten die Initialisierung einschreiben und folglich in $155\,\text{ns} + 17 * 5\,\text{ns} + (2^{20} + 29 - 1) * 5\,\text{ns} + 405\,\text{ns} = 5.243.665\,\text{ns} \approx 5.244\,\mu\text{s}$ erfolgen, also etwa dem 2,6-fachen der Dauer bei Ausführung

[5] engl., Verzweigung eines Signals

	Rück-setzen	$\sum_{i=1}^{1000} v_i \cdot v_i$	**Auslesen**	$2^{20} + \sum_{i=1}^{2^{20}} 2^{-20} \cdot 2^{-20}$	**Matrixmultiplikation** (4 AEs) 1000×1000	1024×1024
C-XSC	–	0,12 ms	–	101 ms	119,4 s	158,6 s
EAU	0,1 ms	0,05 ms	0,01 ms	15 ms	15,8 s	16,7 s

Tabelle 5.21: Vergleich zwischen C-XSC in Software und der exakten arithmetischen Skalarprodukteinheit (EAU) auf der Convey HC-1.

mit doppelter Genauigkeit entsprechen, aber nur einem Bruchteil der exakten Ausführung in Software.

Evaluation der Implementierung auf der Convey HC-1

Auf der Convey HC-1 wird die Anwenderlogik mit einer Taktfrequenz von 150 MHz betrieben anstelle von 200 MHz auf dem UoH HTX-Board. Das Rücksetzen der vier Register benötigt 0,000087 bis 0,000110 Sekunden. Das Multiplizieren-Akkumulieren von 1000 Werten erfolgt in 0,000049 bis 0,000053 Sekunden anstelle von 5,1 Mikrosekunden. Es ist also etwa zehn mal langsamer als errechnet; allerdings werden die Daten dabei auch wirklich aus dem Arbeitsspeicher eingelesen. Das Auslesen eines Werts in doppelt genauer Darstellung ist in 0,000012 bis 0,000015 Sekunden möglich. Die Akkumulation von 2^{20} Werten ist nicht direkt vorgesehen, da die Längenangabe der Vektoren intern auf 16 Bit beschränkt ist. Sie lässt sich dennoch erreichen, indem $16 = 2^4$ Akkumulationen von je $2^{16} = 65.536$ Werten durchgeführt werden. Letzteres benötigt 0,002155 bis 0,002225 Sekunden, während 16 aufeinanderfolgende Akkumulationen dann 0,015379 bis 0,015453 Sekunden benötigen. Damit ist die Implementierung etwa drei mal langsamer langsamer als erhofft, dennoch mehr als zwölf mal schneller als die Ausführung mittels C-XSC. Die Verlagerung eines exakten Skalarprodukts in rekonfigurierbare Hardware ist somit erreicht. Die Multiplikation zweier Matrizen der Dimension 1000×1000 benötigt 15,80 bis 15,82 Sekunden; in 16,65 bis 16,86 Sekunden sind zwei 1024×1024-Matrizen exakt miteinander multipliziert unter Verwendung von vier Skalarprodukteinheiten (je eine pro AE). Tabelle 5.21 vergleicht die Koprozessorausführung der mikroprogrammierten exakten arithmetischen Einheit mit der Ausführung äquivalenter Implementierungen in Software, die auf C-XSC basieren. Dazu wurde auf dem Intel Xeon 5138 der Convey HC-1 mit der Optimierungsstufe `-O2` übersetzt. Es ergibt sich eine Beschleunigung zwischen 6-fach und 9-fach für die unterschiedlichen gemessenen Fälle.

Mit diesen Ergebnissen wurden Konzept und Implementierung einer erweiterbaren Architektur für exakte Arithmetik auf einem breiten Festkommaregister positiv bestätigt. Darauf aufbauend lassen sich Erweiterungen hinzufügen, die aufgrund des modularen Aufbaus und der mikroprogrammierten Ansteuerung nun im Gegensatz zu den ersten Entwürfen (Abschnitte 5.4 und 5.5) umsetzbar sind.

5.7 Erweiterung der exakten Arithmetikeinheit

Ausgehend von der Bedarfsanalyse des Lanczos-Verfahrens (Abschnitt 5.1.3) wird die mikroprogrammierte exakte Recheneinheit erweitert [Kna12]. Nötig sind neben der exakten Berechnung des Skalarprodukts auch exakte Wurzelberechnung und Division.

Akkumulation exakter Festkommaregister

Mit der Portierung auf die Convey HC-1 wurde es möglich, direkt Festkommaregister miteinander zu addieren. Dies erfolgt, indem die Personality aus dem Koprozessorspeicher einen im breiten Festkommaformat gespeicherten Wert einliest und intern addiert, so wie auch Produkte zum Festkommaregister addiert werden oder die Auflösung der Überträge erfolgt. Dabei sind wiederum keine Überträge zu beachten, sondern das zweite interne Festkommaregister wird ebenso wie beim Skalarprodukt zur Auflösung verwendet.

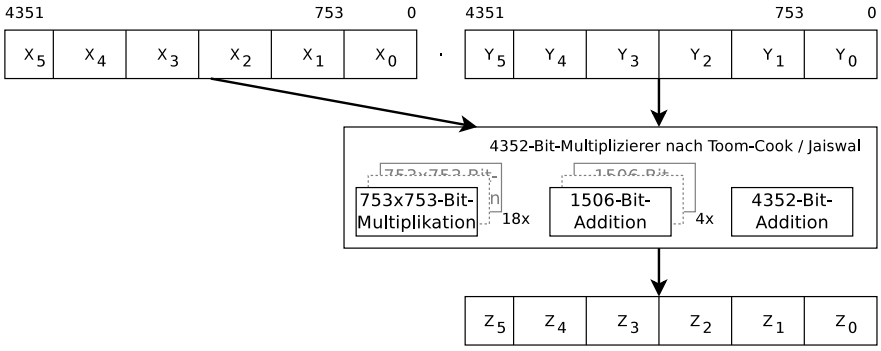


Abbildung 5.32: Einheit zur exakten Multiplikation zweier Zahlen in exakter Festkommadarstellung mittels 753-Bit-Multiplizierern.

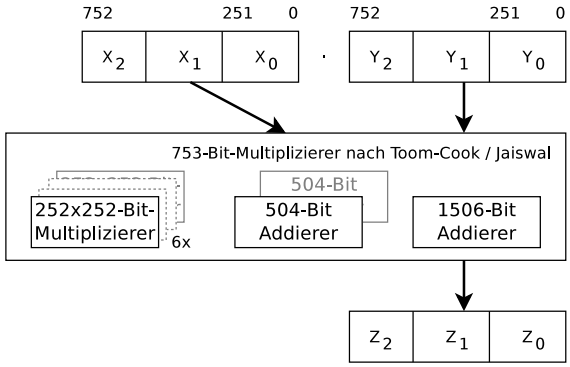


Abbildung 5.33: 753-Bit-Multiplizierer zum hierarchischen Aufbau einer exakten Multiplikationseinheit.

Multiplikation exakter Festkommaregister

Allgemein werden zwei Festkommaregister X und Y, die aus n Blöcken mit m Bits Breite bestehen, nach der Formel

$$X \cdot Y = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} X_i \cdot Y_j \cdot b^{(i+j)\cdot m}$$

miteinander multipliziert. Aufgrund der hohen Laufzeit für $n = 67$ Blöcke und $m = 64$ Bits (Basis $b = 2$) wurde nach einer alternativen Lösung gesucht, die zeitsparender sei. Nach dem Toom-Cook-Algorithmus [68] und nach Jaiswal [156] lässt sich die Anzahl benötigter Multiplikationen durch geschickte Umformungen reduzieren, indem künstlich eingeführte Zwischenterme α und β mehrfach wiederverwendet werden. Weitere Multiplikationen können unter geringfügiger Verschlechterung der Genauigkeit des Ergebnisses dadurch eingespart werden, dass nur solche Blöcke miteinander multipliziert werden, bei denen das Produkt im Festkommaregister speicherbar ist. Mit dieser Zerlegung lässt sich ferner ein hierarchischer Aufbau einer Multiplikationseinheit erreichen, die intern idealerweise achtzehnfach parallel mit 753-Bit-Multiplizierern aufgebaut ist, um vollständig parallel arbeiten zu können (Abbildung 5.32). 18 solche Multiplikationen sind insgesamt durchzuführen und die Ergebnisse miteinander und an den passenden Stellen des breiten Festkommaregisters zu summieren mittels geeigneter Addierer. Die 753-Bit-Multiplizierer sind intern wiederum sechsfach parallel aufgebaut aus 252-Bit-Multiplizierern (Abbildung 5.33). Letztgenannter Multiplizierer verwendet eine Aufteilung in vier Blöcke, aber nur eine 64-Bit-Standardmultiplikationseinheit, mit der die benötigten 10 Multiplikationen in einer Pipeline berechnet werden.

Division

Um das Newton-Verfahren zur Berechnung einer Division $q = a/d$ einsetzen zu können, wird auf Basis der bereits vorhandenen und neu entwickelten Komponenten lediglich eine Einheit zur

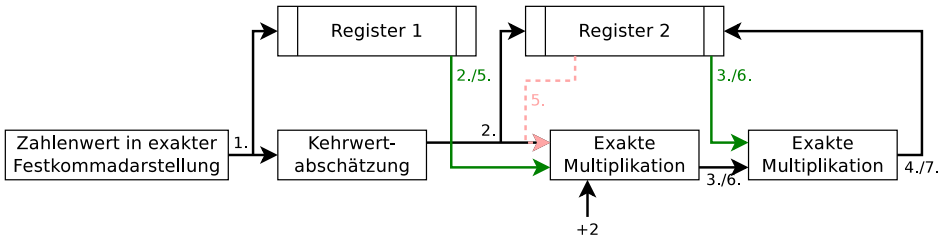


Abbildung 5.34: Division zweier exakter Werte in Festkommadarstellung nach dem Newton-Verfahren unter Verwendung der entwickelten Komponenten. Die Schritte 5-7 sind zu wiederholen, bis das Verfahren konvergiert

ersten Abschätzung des Kehrwerts des Divisors benötigt. Die Division soll auf Werten in exakter Festkommadarstellung erfolgen. Daher kann der Startwert recht einfach anhand der ersten von Null (bei einer positiven Zahl) verschiedenen Stelle n sowie der darauffolgenden 16 Stellen ermittelt werden. Dazu wird eine vorgefertigte Divisionsschaltung eingesetzt, welche $2^{53}/accu[n : n-16]$ berechnet und damit bereits eine sehr exakte Annäherung schafft, so dass nur acht Iterationen benötigt werden. Die Steuerung der Division kann unter Zuhilfenahme dieser Einheit allein über ein Mikroprogramm erfolgen. Zunächst muss in das zweite Akkumulatorregister die Abschätzung des Divisors geschrieben werden, dann kann das Newton-Verfahren bestehend aus $x_{k+1} = x_k \cdot (2 - d \cdot x_k)$ durchgeführt werden, wie in Abbildung 5.34 illustriert ist. Dabei ist zu beachten, dass die Multiplikationen $y := d \cdot x_k$ und $x_{k+1} = x_k \cdot (2 - y)$ exakt durchzuführen sind. Abschließend ist noch die exakte Multiplikation mit dem Dividenden a durchzuführen.

Berechnung der Quadratwurzel

Die Berechnung der Quadratwurzel von a lässt sich ebenso mittels des Newton-Verfahrens unter Verwendung der vorhandenen Komponenten zur Addition und Multiplikation auf Festkommaregistern durchführen. Sie basiert auf der Vorschrift $x_{k+1} = \frac{x_k}{2}(3 - a \cdot x_k^2)$. Die Division durch 2 ist als Schiebeoperation umsetzbar. Gegenüber der oben erwähnten Umsetzung der Division wird eine weitere exakte Multiplikation benötigt zum Quadrieren von x_k.

Zur Approximation des Startwerts x_0 wird eine weitere Komponente benötigt. Diese verwendet zwei Tabellen, die anhand der Stelle der vordersten Eins im Festkomaformat (bei positiven Zahlen) und anhand der nächsten drei Bits die erste Abschätzung in acht Bits ausgeben.

Implementierungs- und Syntheseergebnisse

Erste Syntheseergebnisse zeigten einen Ressourcenbedarf von 22 % der Slice-Register und 25 % der Lookup-Tabellen für die 753-Bit-Multiplikationseinheit. Die breiten Addierer für die Teilergebnisse benötigen zusätzlich je 2 % bis 4 % der Ressourcen. Daher musste die Erweiterung der exakten Skalarprodukteinheit hin zu einer exakten Arithmetikeinheit, die mittels Mikroprogrammen ihre Subkomponenten ansteuert, fallengelassen werden. Anhand der Erweiterungen zur Division und Quadratwurzelbestimmung ist aber erkennbar, dass die Mikroprogrammierung ein aprobates Mittel ist, um komplexe arithmetische Einheiten zu steuern und neue komplexe Operationen zu integrieren, die mitunter sogar iterativer Natur sein können wie eben das Newton-Verfahren.

5.7.1 Zusammenfassung

Es wurde eine leistungsstarke, voll pipeline-fähige Einheit für exakte arithmetische Operationen entwickelt, deren Einzelkomponenten und Verdrahtung derart gehalten sind, dass die Einheit mehr als 200 MHz in der Synthese erreicht. Die Ansteuerung der einzelnen Komponenten erfolgt über vorab festgelegte Mikroprogramme. Die Convey HC-1 bietet für die mikroprogrammierte EAU ausreichend große FPGAs an und ermöglicht so die Unterbringung einer exakten Skalarprodukteinheit pro Application Engine. Der mikroprogrammbasierte Ansatz zum Aufbau der Einheit hat sich als

gangbar erwiesen und das Hinzufügen weiterer arithmetischer Komponenten und die Wiederverwendung bestehender Komponenten ermöglicht. Für weitere Berechnungen auf exakten Darstellungen sind jedoch die eingesetzten Virtex-5 LX330-FPGAs zu klein und zu langsam, als dass der Einsatz der Einheiten durch die interne Ausnutzung von Datenparallelität als effizient betrachtet werden könnte. Das Ergebnis ist, dass sich gewisse Spezialoperationen sehr gut in rekonfigurierbare Hardware bringen lassen und Beschleunigung zu liefern vermögen. Weiter zeigt sich, dass der Ansatz über Mikroprogrammierung geeignet ist, portierbare Hardwareschaltungen für rekonfigurierbare Logik zu entwickeln; vor allem aber sind die Hardwareschaltungen erweiterbar für mehr Funktionalität. Auch komplexe, iterative Operationen wie das Newton-Verfahren für die Division oder Wurzelberechnung sind auf diese Weise aus Software in rekonfigurierbare Hardware verlagerbar.

Bis jetzt wurde erreicht, dass Spezialoperationen von Hardwareexperten unter Einsatz von Hardwarebeschreibungssprachen in rekonfigurierbare Hadware ausgelagert werden können und Nutzen bringen können gegenüber der Verwendung von Softwareimplementierungen der Spezialoperationen. Der Anwender muss damit einerseits auf die korrekte Implementierung der Spezialoperation vertrauen, und andererseits muss sie genau den Bedürfnissen entsprechen, da sie sonst keinen Nutzen mehr verspricht. Der Anwender sollte daher Möglichkeiten erhalten, auf die in rekonfigurierbarer Logik ausgeführten Programmteile mehr Einfluss zu nehmen. Dies verspricht der Einsatz von Hochsprachen zur Hardwarebeschreibung, bei denen der Anwender abstrakt seinen vollständigen Algorithmus in einer Hochsprache formuliert. Anschließend erzeugt ein Konvertierungswerkzeug aus der hochsprachlichen Beschreibung eine Hardwarebeschreibung, die synthetisiert und auf einen FPGA gespielt werden muss, um abschließend Teile der Anwendung in rekonfigurierbarer Hardware auszuführen. Für den Anwender verspricht dies einerseits die volle Kontrolle und Erfüllung der besonderen Bedürfnisse der entwickelten Anwendung, und andererseits entfällt der „Umweg“ über den Hardwareentwickler. Daher beschreibt der folgende Abschnitt, wie Hochsprachen zur Hardwarebeschreibung eingesetzt werden können am Beispiel eines Stencil-basierenden numerischen Problems.

5.8 Hochsprachen zur Hardwarebeschreibung

Nachdem Beschleunigung auf Instruktionsebene betrachtet wurde, die Granularität auf Funktionsebene angehoben wurde aufgrund nicht zufriedenstellender Ergebnisse, und die Implementierungen dabei überwiegend hardwarenah und in VHDL erfolgt sind, soll ein gegensätzlicher Ansatz betrachtet werden: Die Beschreibung von Anwendungen oder wichtigen Kernels in einer Hochsprache und der Einsatz von Quelltextkonvertern, um für die leistungsrelevanten Teile Hardwareschaltungen zu erhalten (vgl. Abschnitt 2.1.2).

Bei der Verwendung von Hochsprachen zur Beschleunigung von Anwendungen oder Anwendungsteilen durch FPGA-Ausführung sind einige Probleme zu lösen. An Beispielen wird klar werden, dass auch Hochsprachen gute Hardwarekenntnisse bedingen, aber aufgrund des hohen Aufwands bei der Entwurfsraumexploration auch keineswegs für den Hardware-Entwickler ein geeignetes Werkzeug über die Prototypenphase hinaus darstellen. Die Portabilität der Schaltungen stellt ein weiteres Problem dar, da die Daten an mehreren und unterschiedlichen Stellen vorgehalten, umsortiert, gepackt oder entpackt werden müssen.

Die Untersuchungen zur Verwendung vom Compiler ROCCC [123] und von Impulse C [152] zur Erzeugung von Hardwarebeschreibungen werden konkreter an Stencil[6]-basierten Verfahren durchgeführt. Ergänzend dazu wird ein compiler-basierter Ansatz von Convey zur Beschleunigung des CG-Verfahrens evaluiert. Zuvor muss noch das als Beispielanwendung eingesetzte Verfahren der konjugierten Gradienten mit dem Beispielproblem der Poisson-Gleichung im zweidimensionalen Raum beschrieben werden, um daran einige Aspekte bei der Implementierung von Stencil-Berechnungen in Software und Hardware näher beleuchten zu können.

[6] engl., Stempel

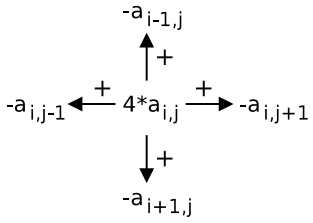


Abbildung 5.35: 2-dimensionaler Stempel mit den Koeffizienten $(-1, -1, 4, -1, -1)$.

5.8.1 Die Poisson-Gleichung

Die Lösung der Poisson-Gleichung ist ein Standard-Problem, das häufig zur Abschätzung oder zum Vergleich von unterschiedlichen Lösungsverfahren eingesetzt wird. Dabei geht es darum, die gleichmäßige Verteilung und Anpassung der Werte unter Beeinflussung und Berücksichtigung der Nachbarwerte zu errechnen, wie dies bei Hitzeverteilungsproblemen in Metallen oder Strömungssimulationen nötig ist.

Die Poisson-Gleichung ist eine partielle Differentialgleichung für zeitunabhängige Probleme. Gegeben sei eine Funktion $f : \Omega \to \mathbb{R}$ auf einem Gebiet $\Omega \subset \mathbb{R}^d$, dann ist eine Funktion $u : \overline{\Omega} \to \mathbb{R}$ gesucht, wobei $\overline{\Omega} = \Omega \cup \partial\Omega$, die

$$-\triangle u = -\nabla \cdot (\nabla u) = -\nabla \cdot (\partial_1 u, \ldots, \partial_d u)^T = -\sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2} u = f \quad \text{innerhalb eines Gebiets } \Omega,$$

$$u = 0 \text{ auf dem Rand } \partial\Omega \quad \text{(homogene Dirichlet-Bedingung)}$$

erfüllt, wobei Ω auf $(0, 1)^2$ eingeschränkt wird und $f : \Omega \to \mathbb{R}$, $f \in C(\Omega)$ verlangt wird. u ist weiter spezifiziert durch $u \in C^2(\Omega) \cap C(\bar{\Omega})$.

Um auf numerischem Wege eine Lösung finden zu können, wird die Domäne Ω diskretisiert, im zweidimensionalen Falle zu $\Omega_h = \{(x, y) \in \Omega \mid \exists k, l \in \mathbb{Z} : x = k \cdot h, y = l \cdot h\}$ indem ein gleichmäßiges Gitter mit Abstand $h > 0$ auf Ω gelegt wird. Daher ist der Differenzenquotient gegeben durch $\frac{\partial}{\partial x} u(x, y) \approx \frac{u(x+h,y)-u(x,y)}{h}$. Zählt man die diskreten Punkte x_i in Ω_h durch und bezeichnet sie $u(x_j + he_i) =: u_{j+e_i}$, wobei e_i der i-te Einheitsvektor ist, so kann die Finite-Differenzen-Methode (FDM) nun angewandt werden. Auf einem eindimensionalen Gitter erhält man, wenn man die Vorwärts- und Rückwärtsdifferenzenquotienten miteinander verknüpft im Sinne einer doppelten Ableitung, $\partial_i \overline{\partial_i} u_j = \partial_i(\frac{u_j - u_{j-e_i}}{h}) = \frac{u_{j+e_i} - u_j}{h^2} - \frac{u_j - u_{j-e_i}}{h^2} = \frac{u_{j+e_i} - 2u_j + u_{j-e_i}}{h^2}$. Die Koeffizienten der u_i hierbei werden häufig als Stencil-Werte behandelt, in diesem Fall also $(-1, 2, -1)$.

Somit kann für $d = 2$ die Gleichung $-\Delta u$ folgendermaßen mittels finiter Differenzen approximiert werden:

$$-\Delta u = \frac{-u_{j+e_1} - u_{j-e_1} + 4u_j - u_{j+e_2} - u_{j-e_2}}{h^2} + \mathcal{O}(h^2)$$

Aus obiger Gleichung lässt sich leicht ersehen, dass man mit dem Stempel $(-1, -1, 4, -1, -1)$ (vgl. Abb. 5.35) über die Matrix iterieren kann, um das Problem zu lösen.

Das entstehende lineare Gleichungssystem (LGS) $A_h u_h = b_h := h^2 \cdot f((x, y))$, wobei A_h die Koeffizienten $\partial_i \overline{\partial_i} u_j$ der FDM enthält, kann nun mittels passender Gleichungslöser gelöst werden. Eine Lösung für die Poisson-Gleichung im zweidimensionalen Raum ist in Abbildung 5.36 illustriert.

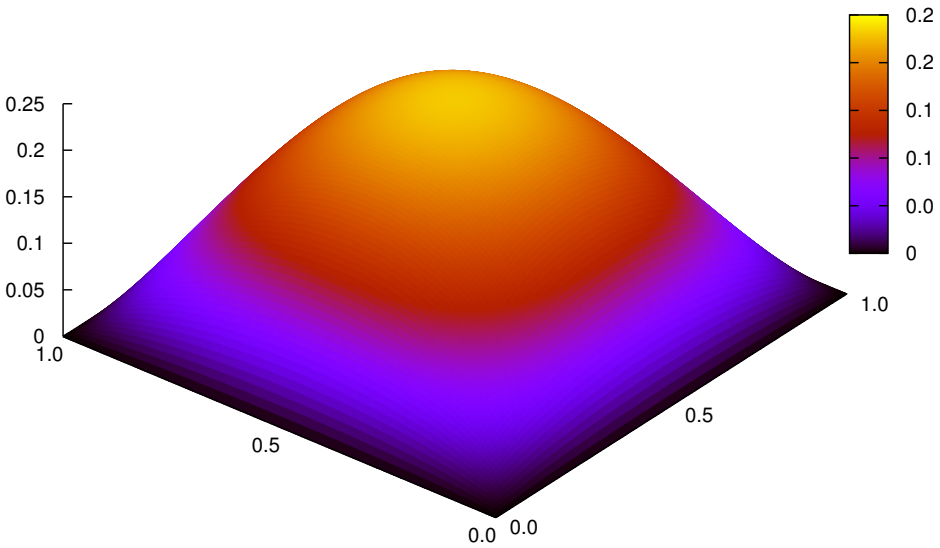


Abbildung 5.36: Lösung für die Poisson-Gleichung mit Diskretisierung h=128 auf $\bar{\Omega} = [0,1]^2$.

```
1  read_right_hand_side (vector b);
2  x0=stencil_op(x); residuum=sub_matrix(b, x0);
3  p = precondition (residuum);
4  rz=scalar_product(residuum, p); tol=1e-12 *rz;
5  for  (k=0; k<max_iteration_count; k++) {
6    Ap = stencil_op(p);
7    alpha = rz / scalar_product(p, Ap);
8    alphaP = multiply_matrix_scalar(p, alpha);
9    x = add_matrix(x, alphaP);
10   alphaP = multiply_matrix_scalar(Ap, alpha);
11   residuum = sub_matrix(residuum, alphaP);
12   z = precondition (residuum); rz_old = rz;
13   rz = scalar_product(residuum, z);
14   if (rz < tol) break;
15   beta = rz / rz_old;
16   xnew = multiply_matrix_scalar(p, beta);
17   p = add_matrix(z, xnew);
18 }
```

Listing 5.2: Pseudo-Code des vorkonditionierten CG-Lösers.

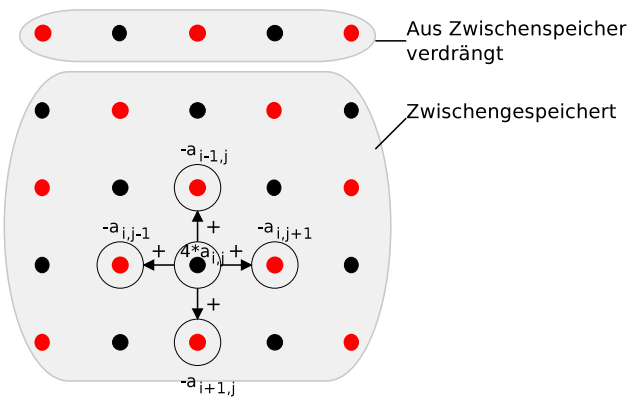
5.8.2 Lösen von linearen Gleichungssystemen mit dem Verfahren der konjugierten Gradienten

Das soeben erhaltene LGS kann nun mittels des Verfahrens der konjugierten Gradienten[7] (CG-Verfahren) [143] gelöst werden, wobei zusätzlich Vorkonditionierungsmethoden zum Einsatz kommen können. Auflistung 5.2 zeigt die wesentlichen Schritte des CG-Verfahrens auf, wenn ein Vorkonditionierer eingesetzt wird. Die Matrix A_h ist sehr dünn besetzt und wird daher implizit beschrieben durch `stencil_op`, um keine Matrixmultiplikation duchführen zu müssen. Ament et al. geben eine tiefgehende Beschreibung der Poisson-Gleichung und des Lösens mittels des vorkonditionierten CG-Verfahrens [10].

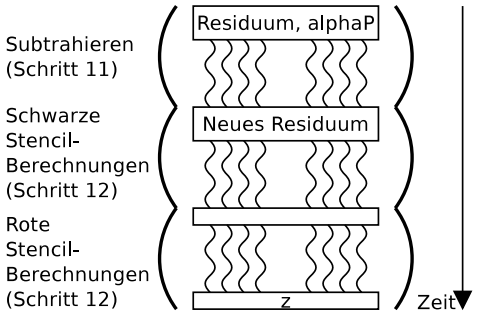
Symmetrischer Gauss-Seidel-Vorkonditionierer nach dem Rot-Schwarz-Schema

Als Löser des Vorkonditionierungsproblems im obigen LGS lässt sich die Gauss-Seidel-Methode einsetzen. Das Lösen von $Ax = b$ wird in iterativer Manier vollzogen als $x^{(k+1)} = Tx^{(k)} + c$, wobei

[7] Conjugate Gradient Method (CG-method)



(a) Anwendung des 2D-Stempels für die FDM.



(b) Mehrfädige, aber nicht nebenläufige Ausführung der schwarzen und roten Stempel-Berechnungen auf dem Ergebnis der vorhergehenden Matrixsubtraktion.

Abbildung 5.37: Rot-schwarze Stempelberechnungen.

$T = -(D+L)^{-1}U$ und $c = (D+L)^{-1}b$ ist für eine in untere, diagonale und obere Matrix zerlegte Matrix $A = L + D + U$ (Splitting-Verfahren). Legt man ein Schachbrettmuster oder Rot-Schwarz-Muster auf $r^{(k)}$ und Δx, so können die roten Punkte unabhängig voneinander parallel berechnet werden, indem jeder Punkt durch den mittleren Stempelwert wie in D vorgegeben geteilt wird. Danach können die schwarzen Punkte berechnet werden, indem $(D+L)\Delta x = r^{(k)}$ gelöst wird, was als $\Delta x_{i,j} = (r^{(k)}_{i,j} - (-\Delta x_{i-1,j} - \Delta x_{i+1,j} - \Delta x_{i,j-1} - \Delta x_{i,j+1}))/4$ formuliert werden kann. Es werden also die vorher errechneten neuen roten Punkte wiederverwendet. Dieser Berechnungsschritt ist darüber hinaus hochgradig parallelisierbar in $d/2$ parallel ausführbare Operationen aufgrund der nun nicht mehr vorliegenden Abhängigkeiten. Dieser Methode fehlt die Symmetrie. Anstelledessen kann die symmetrische Gauss-Seidel-Methode verwendet werden, die zuerst mit den schwarzen Punkten des Residuums beginnt, wobei sie einen etwas veränderten Stempel verwendet, und danach mit den roten Punkten auf den schon berechneten schwarzen Punkten wie oben fortfährt. Sobald die ersten Stempel auf den schwarzen Punkten berechnet sind, könnten einige „rote“ Stempel schon theoretisch berechnet werden, z.B. in Abb. 5.37(a) der rote Punkt oberhalb des derzeit berechneten schwarzen Punkts. Die benötigten Daten sollten sogar noch in den Caches liegen. Mit datenparalleler Programmierung ist dies jedoch nicht direkt ausnutzbar. Die Berechnungen müssen daher bei datenparalleler Programmierung dann mehrfädig, aber „rot“ und „schwarz“ zeitlich voneinander getrennt ablaufen, wie es in Abb. 5.37(b) dargestellt ist. Es ist weitergehend denkbar, die roten und schwarzen Operationen auf unterschiedlichen Zeilen zeitgleich durchzuführen; dann sind die Arbeitsdaten aber eventuell schon aus dem Cache verdrängt oder liegen in den Caches anderer Prozessorkerne oder gar anderer Prozessoren.

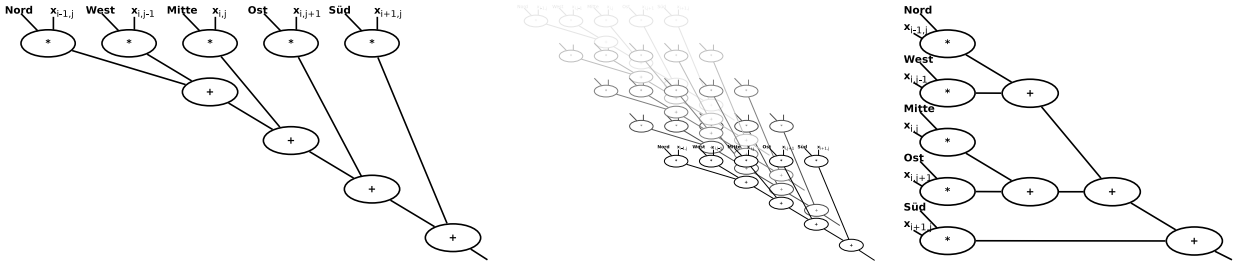
5.8.3 Analyse des Stencil-Kernels hinsichtlich effizienter Implementierungen

Eine genauere Betrachtung möglicher Stencil-Implementierungen wird jetzt vollzogen. Das Ziel ist eine möglichst effiziente Implementierung, deren Pipeline kurz ist, deren Ressourcenbedarf gering ausfällt und die mit einer hohen Taktrate genutzt werden kann. Es lassen sich einige Anforderungen aufdecken und Möglichkeiten besonders für Stencils nach dem Rot-Schwarz-Schema festhalten.

Ein Fünfpunktstencil auf zweidimensionalen Daten lässt sich folgendermaßen definieren und implementieren:

$$result_{ir,j} = x_{i-1,j} * factor_{\text{north}} + x_{i,j-1} * factor_{\text{west}} + x_{i,j} * factor_{\text{center}} + x_{i,j+1} * factor_{\text{east}} + x_{i+1,j} * factor_{\text{south}}.$$

Die Darstellung des Datenflusses in Abb. 5.38(a) zeigt, dass die einzelnen Multiplikationen und Additionen hervorragend in einer Pipeline ausgeführt werden können wie in Abb. 5.38(b). Wenn

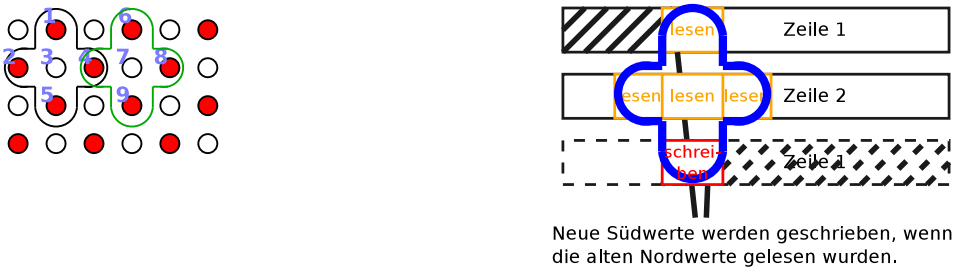


(a) Datenfluss eines einzelnen Stencils. (b) Ausführung in einer Pipeline. (c) Baumbasierter Aufbau.

Abbildung 5.38: Reihenfolge bei Stenciloperationen und Anordnung in einer Pipeline.

alle Daten auf einmal verfügbar sind, etwa weil oben beschriebene Zeilenpuffer eingesetzt werden, so kann der Stencil mit kürzerer Latenz mittels einer Baumstruktur implementiert werden (Abb. 5.38(c)).

Wenn Stencil-Operationen ausgeführt werden, müssen Daten von nichtkonsekutiven Adressen im Speicher bezogen werden, wie anhand Abb. 5.37(a) ersichtlich wird. Bei nichtkonsekutiven Zugriffen braucht man mehr Hardware-Ressourcen zur Ansteuerung als bei konsekutiven Zugriffen und benötigt zudem mehr Zeit, da keine Blocktransfers gemacht werden können. Somit ist der nichtkonsekutive Zugriff nicht ausreichend effizient. Es ist daher nötig, in einem gewissen Rahmen die Datenzugriffe und Berechnungen voneinander zu entkoppeln. In Software sollten die Daten vorgeladen[8] und vorgehalten werden, damit sie möglichst schnell bei der aktuellen Berechnung bereitstehen und bei den Berechnungen des direkt folgenden Stencils sowie denen der nächsten Zeile wiederverwendet werden können. Damit kann bereits hohe Leistung erreicht werden. Wie man in Abb. 5.39(a) sieht, kann zunächst der alte Ost-Wert der Eingabematrix als neuer Westwert des nächsten Stencils verwendet werden, wodurch sich bereits die Anzahl der Speicherzugriffe um ein Fünftel verringert. Ein weiterer Verbesserungsansatz ist, die ersten zwei Zeilen der Matrix vorzuladen in interne Speicher wie in Abb. 5.39(b) angedeuet, wodurch bis auf den Südwert alle benötigten Daten für eine Stencil-Operation bereits vorliegen. Der neue Südwert ersetzt dann den alten Nordwert. Dadurch lässt sich die Anzahl Speicherzugriffe weiter senken auf nur noch ein Fünftel.



(a) Die Wiederverwendung des alten Ostwertes als neuer Westwert bei einem Stencil nach dem Rot-Schwarz-Schema reduziert die Speicherzugriffe auf 4/5.

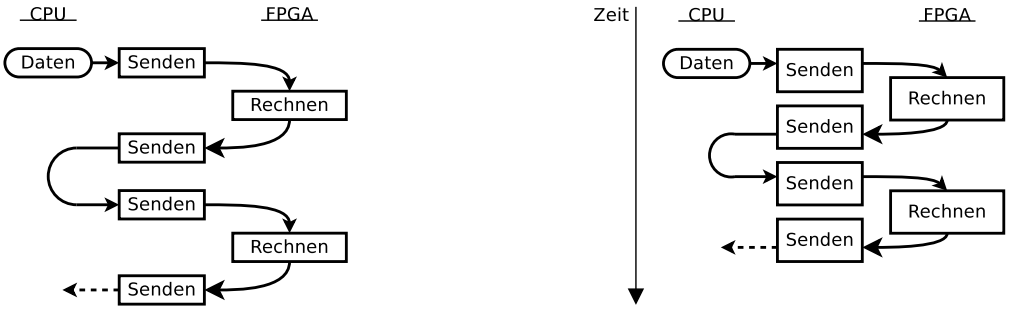
(b) Vorladen und Bereitstellen der letzten Daten reduziert die Speicherzugriffe auf 1/5.

Abbildung 5.39: Die Anzahl an nötigen Speicherzugriffen lässt sich durch Datenwiederverwendung auf das Minimum reduzieren.

Bei Verwendung von Koprozessoren bedeutet die Vorhaltung und Wiederverwendung von Daten, dass nicht jedes einzelne von der Stencil-Operation benötigte Datum separat übertragen werden darf. Die einfachste, in Abb. 5.40(a) dargestellte Möglichkeit ist, die Arbeitsdaten komplett auf den Koprozessor zu übertragen, dann den Stencil berechnen zu lassen, dann die Daten zurückzuübertragen und von vorne zu beginnen. Auf dem Koprozessor benötigt man dazu dann einen schnellen Speicher mit wahlfreiem Zugriff, etwa durch Block-RAMs implementiert. Insbesondere bei wiederholter Ausführung des Stencils ist es jedoch wesentlich sinnvoller, die Daten wie in Abb. 5.40(b) zu streamen und die Verarbeitung zu beginnen, sobald die ersten benötigten Daten vorliegen, während

[8] engl. prefetching

der Transfer nebenläufig erfolgt. Gleichermaßen kann das Rückschreiben schon während der Berechnungen erfolgen. Dies hat den Nachteil, mehr Steuerungslogik und damit mehr Platzbedarf zu benötigen.



(a) Getrennte Übertragung und Berechnung.

(b) Überlappte Übertragung und Berechnung.

Abbildung 5.40: Das Überlappen von Transfer und Berechnung kann die Gesamtausführungszeit wesentlich verkürzen.

Bei großer Speicherbandbreite können unter Umständen mehrere Operationen gleichzeitig angestoßen werden, z.B. zwei Operationen bei 32-Bit-Daten und einer 64-Bit-Schnittstelle. Weitere Steigerungen lassen sich erzielen, wenn mehrere Speicher oder Speicherbänke gleichzeitig verwendet werden. Neben der zeilenbasierten Aufteilung einer Matrix auf mehrere Speicher wie in Abb. 5.41(a) ist es gerade für das Rot-Schwarz-Schema praktisch, die Matrix zweispaltenweise (Abb. 5.41(b)) aufzuteilen, um unter Verwendung der oben beschriebenen Zeilenpuffer je Takt zwei Stenciloperationen anstoßen zu können.

(a) Zeilenbasierte Aufteilung einer Matrix auf zwei Speicher.

(b) Spaltenbasierte Aufteilung einer Matrix auf zwei Speicher.

Abbildung 5.41: Einsatz mehrerer Speicher(-bänke) zur Verbesserung des Datenzugriffs.

5.8.4 Stencil-basierter Vorkonditionierer mit Impulse C auf der DRC AC2030

Zur Untersuchung [233], ob Hochsprachen von Domänenexperten prinzipiell nutzbar sind und welche Leistung erzielbar ist, wurde Impulse C (vgl. Abschnitt 2.1.2) verwendet, um eine Beschleunigereinheit für die DRC AC2030 (vgl. Abschnitt 2.1.4) zu entwickeln. Aufgrund der vorherrschenden Annahme, der Vorkonditionierer bei iterativen Lösern sei der beschleunigenswerteste Teil, und weiter aufgrund der Annahme, Hochsprachen würden dazu führen, dass mehrere Iterationen automatisch parallel zueinander auf dem FPGA berechnet würden, wenn nur keine Datenabhängigkeiten vorliegen, wurde die Architektur aus Abb. 5.42 für das CG-Verfahren mit SSOR-Vorkonditionierer entwickelt [HKN+11]. Jene Publikation [HKN+11] dient als Grundlage für diesen Abschnitt 5.8.4.

Der Software-Prozess zur Ansteuerung des Koprozessors stellt einen Stellvertreter für den Aufruf eines Vorkonditionierers bzw. eines SSOR-Verfahrens dar. Er überträgt zuallererst die Koeffizienten des Stencils, um für andere Probleme als das Poisson-Problem einsetzbar zu sein. In jeder Iteration des CG-Verfahrens wird der Software-Prozess aufgerufen und ihm dabei ein Zeiger auf das aktuelle Residuum übergeben, das dieser in den Koprozessorspeicher schreibt. Daraufhin wird ein Start-Signal an den Koprozessor gesendet und von dort an auf das Rücksenden von $z_{k+1} = M^{-1}r_{k+1}$ seitens des Koprozessors an den Host gewartet, wie in Listing 5.3 gezeigt.

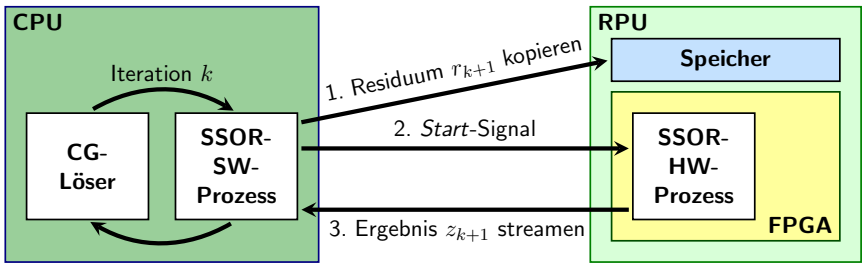


Abbildung 5.42: CG-Löser mit Vorkonditionierer auf DRC RPU.

```
for (i=0; i<sdim; i++) {
 for (j=0; j<sdim; j++) {
  mem_r[i][j] = residuum[i][j]
 }
}
// tell HW that it can begin
signal_post(rStored, 1)
for (i=1; i<=sdim-2; i++) {
 //start each row with black color
 start = (i+1)%2 + 1
 for (j=start; j<=sdim-2; j+=2) {
  //read next z_{k+1}[i][j] from HW
  stream_read(z[i][j])
 }
}
for (i=1; i<=sdim-2; i++) {
 //start each row with red color
 start = i%2 + 1
 for (j=start; j<=sdim-2; j+=2) {
  //read next z_{k+1}[i][j] from HW
  stream_read(z[i][j])
 }
}
```

Listing 5.3: Software-Prozess des SSOR-Verfahrens nach Rot-Schwarz-Schema in Pseudo Impulse C.

```
co_int32 aD, aN, aE, aS, aW, sdim;
// iterate only over black inner
//  supporting points
for (i=1; i<=sdim-2; i++) {
 start = (i+1)%2 + 1;
 for (j=start; j<=sdim-2; j+=2) {
  // read element r[i][j] and neighbors
  r        = mem_r[i  ][j  ]
  deltaxN = mem_r[i-1][j  ]
  deltaxS = mem_r[i+1][j  ]
  deltaxW = mem_r[i  ][j-1]
  deltaxE = mem_r[i  ][j+1]
  dx = omega * (2-omega) *
  ( r - omega * (aN*deltaxN
     + aS*deltaxS + aE*deltaxE
     + aW*deltaxW) / aD ) / aD
  mem_r[i][j] = dx
  stream_write(dx)
 }
}
// iterate only over red inner
//  supporting points
for (i=1; i<=sdim-2; i++) {
 start = i%2 + 1;
 for (j=start; j<=sdim-2; j+=2) {
  // read element r[i][j] and neighbors
  r        = mem_r[i  ][j  ]
  deltaxN = mem_r[i-1][j  ]
  deltaxS = mem_r[i+1][j  ]
  deltaxW = mem_r[i  ][j-1]
  deltaxE = mem_r[i  ][j+1]
  dx = ( omega * (2-omega) * r
  - omega * (aN*deltaxN
  - aS*deltaxS - aE*deltaxE
  - aW*deltaxW) ) / aD
  stream_write(dx)
 }
}
```

Listing 5.4: Hardware-Prozess des SSOR-Verfahrens in Pseudo Impulse C.

Der Koprozessor führt wie in Listing 5.4 den schwarzen und den roten Stencil hintereinander aus, wobei aufgrund der Dirichlet-Bedingung ein mit Null initialisierter Rand vorhanden ist, der die explizite Behandlung von Randwerten unnötig macht und stattdessen nur auf den inneren Werten gerechnet wird. Die schwarzen Stencil-Werte werden in dem nachfolgenden Teil zur Berechnung der roten Werte benötigt und daher in den koprozessoreigenen Speicher geschrieben. Sie werden gleichzeitig zur Berechnung des jeweils nächsten Werts zum Host übertragen, um dem Prinzip der Überlappung von Transfer und Berechnung, das mit dem Prinzip der CSPs einhergeht, gerecht zu werden. Die roten Werte werden ebenso in Streaming-Manier überlagert mit der Berechnung in den

Auflösung	Zeit SSOR auf CPU	Erwartete Zeit auf FPGA	Verhältnis „erwartet“ zu „CPU“	Gemessene Zeit auf FPGA	Verhältnis „gemessen“ zu „erwartet“
500×500	0,003518	0,073	20,75	0,974298	13,32
1000×1000	0,013678	0,292	21,35	3,880566	13,26
2000×2000	0,055825	1,170	20,96	15,60444	13,34
4000×4000	0,257733	4,680	18,16	61,91578	13,23

Tabelle 5.22: Laufzeit eines SSOR-Vorkonditionierers in Software auf der DRC AC2030 gegenüber erwarteter und gemessener Laufzeit auf dem FPGA mit 100 MHz (jeweils in Sekunden), jeweils gemittelt über zehn Aufrufe bei neun Iterationen des CG-Verfahrens.

Hostspeicher übertragen. Wenn alle Werte berechnet und übertragen sind, ist die Vorkonditionierung abgeschlossen und das CG-Verfahren kann fortfahren.

Leistungsabschätzung a priori

Anhand eines einfachen Modells kann die erwartbare und erwartete Leistung der Hardware-Software-Partitionierung und -Implementierung abgeschätzt werden. Da $\omega = 1$ verwendet wird[9], da es für diesen Anwendungsfall die beste Konvergenz verspricht, ist davon auszugehen, dass für Probleme der Größe $dim \times dim$ genau dim Iterationen bis zur Konvergenz nötig sind.

Pro Taktzyklus der mit 400 MHz getakteten HyperTransport-Anbindung kann ein Gleitkommawert übertragen werden, daher ist $t_{\text{Transfer}} = 2,5\,\text{ns}$. (Zusätzlicher Mehraufwand durch das Protokoll ist bei großen Datenmengen einigermaßen vernachlässigbar.) Pro Stencilberechnung werden fünf Werte aus dem Koprozessorspeicher ausgelesen, und vier Additionen, ein oder zwei Divisionen sowie ein Schreibzugriff ausgeführt. Die entstehende Berechnungsdauer beträgt daher 4 Additionen * 5 Takte/Addition + 2 Divisionen * 29 Takte/Division = 78 Takte. Dadurch sollten sich die Speicherzugriffe überdecken lassen, und bei einer Taktung der Schaltung von 100 MHz bzw. Zyklusdauer von 10 ns ergibt sich die Berechnungsdauer eines Stencils bei gefüllter Pipeline einzig in Abhängigkeit der Dauer einer Division zu $t_{\text{Rechnen}} = 29 * 10\,\text{ns} = 290\,\text{ns}$. Zum Ende hin muss die Pipeline geleert werden in $78 - 1 = 77$ Takten und das letzte Datum zurückgeschrieben werden: $t_{\text{Rückschreiben}} = 2,5\,\text{ns}$. Das Rückschreiben der vorher berechneten Werte erfolgt überlappt. Für eine $n \times n$-Matrix bedeut dies $t_{\text{Gesamt}}(n) = n^2 * t_{\text{Transfer}} + n^2 * t_{\text{Rechnen}} + 77 * 10\,\text{ns} + t_{\text{Rückschreiben}} = n^2 * 2,5\,\text{ns} + n^2 * 290\,\text{ns} + 770\,\text{ns} + 2,5\,\text{ns}$. Somit ist $t_{\text{Gesamt}}(4000) = 4680\,\text{ms}$. Die erwarteten Ausführungszeiten sind in Tabelle 5.22 aufgeführt. Es ist davon auszugehen, dass der FPGA für dieses Design etwa 20 mal langsamer sein wird.

Messergebnisse

Den erwarteten Ausführungszeiten sind die Werte für die Ausführung auf der Host-CPU sowie auf dem FPGA der DRC AA2311 in Tabelle 5.22 gegenübergestellt. Um die FPGA-Programmierung auf Basis von Hochsprachen nicht nur als bequemen, sondern auch als effektiven Ansatz betrachten zu können, sollte die erwartete Ausführungszeit erreicht werden. Tatsächlich ist der Vorkonditionierer auf dem FPGA 13 mal langsamer als erwartet und führt damit zu erheblicher Verschlechterung der Gesamtlaufzeit gegenüber einem Vorkonditionierer in Software auf der CPU.

Bei diesen Messungen wurden allerdings das gesamte Programmiermodell und die Hardware-Plattform mit der Zwischenschicht DRC Milano OS auf dem FPGA evaluiert. Für einen direkteren Vergleich sollte die koprozessorgestützte Ausführung daher gegenüber einer softwarebasierten Ausführung des Impulse-C-Designs verglichen werden, welches ebenso CSPs und mehrere Speicher nutzen müsste. Aus Sicht eines Anwenders zählt jedoch nur die insgesamt erzielbare Beschleunigung ohne besondere Beachtung des Programmiermodells oder Laufzeitsystems.

Es hat sich gezeigt, dass es gegenüber der Verwendung von RTL-Sprachen mittels Impulse C als Vertreter der Hochsprachen zur Hardwarebeschreibung möglich ist, als Anwendungsexperte FPGAs

[9] Damit ist das SOR-Verfahren zu einem Gauss-Seidel-Verfahren vereinfacht

zu nutzen. Dennoch lässt sich aus den Ergebnissen bereits klar erkennen, dass Hochsprachen zur Nutzung von rekonfigurierbarer Hardware nicht den gewünschten Nutzen für Domänenexperten wie etwa Mathematiker liefern können, solange diese nicht Eigenschaften der Rechner- und Systemarchitektur (Caching, Pipeling, Implementierung digitaler Arithmetik, Ausrollen von Schleifen) sowie von rekonfigurierbarer Hardware (kritische Pfade, implizite und explizite Parallelität) und letztendlich auch über die Besonderheiten vom Programmiermodell (CSPs) und Laufzeitsystem (DRC Milano OS) gezielt berücksichtigen wollen. Die nötigen und im Folgenden beschriebenen Erweiterungen können nur unter Berücksichtigung der genannten Punkte geleistet werden.

Ursachensuche und -behebung

Die Verwendung des FPGAs mit Impulse C zur Beschleunigung des SSOR-Vorkonditionierers gegenüber CPU-Ausführung benötigt den Einsatz weiterer Strategien und Technologien. Folgend ist die schrittweise Untersuchung der Ursachen für die erzielten Ergebnisse dargestellt und das Design wird den Erkenntnissen entsprechend verbessert.

Viele der in Abschnitt 5.8.3 genannten Aspekte müssen betrachtet und gegebenenfalls umgesetzt werden. Pipelining, Nutzung mehrerer gleichzeitig agierender Stencileinheiten oder -datenpfade, Datenwiederverwendung und Speicherzugriffsoptimierung, bis die Implementierung die Speicherschnittstelle völlig auslastet, sind Möglichkeiten zur Leistungssteigerung des FPGA-Vorkonditionierers. Desweiteren kann das Senden eines Signals eingespart werden, das zum Starten und Synchronisieren der Iterationen eingesetzt wurde. Tabelle 5.23 fasst die möglichen Verbesserungsansätze zusammen. Sie werden im Folgenden näher beleuchtet.

Entfernen von ω aus den Stencil-Berechnungen. Zwar ist die Überrelaxationsmethode hochgenerisch und für eine Vielzahl an Anwendungen anwendbar, aber das CG-Verfahren konvergiert für das betrachtete Problem der Poisson-Gleichung am schnellsten mit $\omega = 1.0$ für das Rot-Schwarz-Schema. Daher sollten die Übertragung von ω und die Multiplikation des Stencils mit ω entfernt werden, wodurch man die symmetrische Gauss-Seidel-Methode erhält. Aufgrund der Verkürzung des Datenpfads ist bereits etwas Leistungssteigerung zu erwarten. Außerdem muss auch ein Parameter weniger übertragen werden. Der Laufzeitgewinn der in Tabelle 5.24 aufgeführten Messungen gegenüber denen aus Tabelle 5.22 fällt nur sehr gering aus.

Pipelining. In Impulse C kann über das Pragma `#pragma CO PIPELINE` und gegebenenfalls `#pragma CO stagedelay 64` die innerste Schleife zu einer Pipeline in VHDL transformiert werden. Dabei kann es passieren, dass das Übersetzungswerkzeug Variablen als Feedbackvariablen erkennt, wodurch kein Pipelining möglich ist. Desweiteren könnten die Pipeline-Stufen zu viele arithmetische und logische Operationen erhalten, wodurch die maximale Taktrate gesenkt wird. Um dies zu lösen, soll das oben genannte Pragma der Stufendauer eingesetzt werden; es ließen sich allerdings

Datentransfer	**Datenwiederverwendung**	**Weitere Optimierungen**
• Softwareseitiger Speicherzugriff in Blöcken • Größere Stream-Puffer • Streaming der rechten Seite b und des Residuums r	• Ausnutzung cache-ähnlicher Effekte bei hardwareseitigem Speicherzugriff in Blöcken • Alter Ostwert als neuer Westwert • Zeilenpuffer für die zuletzt verwendeten Daten • Sofortige Wiederverwendung in rotem Stencil	• Unnötige Signale entfernen • Division durch Multiplikation ersetzen • Modulo-Berechnung aus Schleifenindizierung entfernen • Schnellere Berechnung der Speicherindize • ω aus Stencil-Berechnungen entfernen
Ausführung in Pipeline	**Speicherbandbreite ausnutzen**	
• Pipelining mittels Impulse-C-Pragmas aktivieren	• Beide LLRAMs nutzen für datenparallele Ausführung	

Tabelle 5.23: Maßnahmen zur Behebung der Ursachen für die geringer ausgefallene Leistung als erwartet.

Iterationen	Auflösung		
	10×10	100×100	1000×1000
1	0,000509	0,041666	4,137024
10	0,002841	0,370814	39,790556
100	0,026159	3,771276	386,636393
1000	0,257077	36,56551	–

Tabelle 5.24: Durchführungszeit (in Sekunden) unterschiedlich vieler SGS-Iterationen für unterschiedliche Problemgrößen auf der AC2030.

Iterationen	Auflösung		
	10×10	100×100	1000×1000
1	1,04980842911877	1,06186289979212	1,09897263338543
10	1,10843833524246	1,11224629214987	1,08353554769228
100	1,07701900748258	1,07193784666087	1,08807295907816
1000	1,09856709199424	1,10266924503429	–

Tabelle 5.25: Beschleunigung der Impulse-C-Implementierung mittels Blocktransfers.

keine Werte unterhalb von 72 erhalten. Dadurch ließen sich die Designs nicht für die angestrebte Taktfrequenz von 100 MHz routen und platzieren, obwohl die ersten Syntheseergebnisse noch 130 MHz versprochen hatten, sondern benötigen hingegen Taktdauern von mehr als 12 ns. Daher war das vollständig platzierte Design nur für sehr geringe Eingabegrößen und wenige Iterationen funktionsfähig und führte ansonsten zu falschen Ergebnissen. Daher werden hier auch keine Meßergebnisse angegeben und auf Pipelining vorerst verzichtet. Auch erfahrene Hardware-Entwickler erreichen an dieser Stelle somit die Grenzen des eingesetzten Werkzeugs „Impulse CoDeveloper“ in Kombination mit der verfügbaren Hardware eines Virtex-5 LX330.

Blockweise Datenübertragung zum Speicher. Schreiben von Daten in den Koprozessorspeicher kann auf zwei unterschiedliche Weisen beschleunigt werden, entweder über Blocktransfers oder über Streaming, so dass gleichzeitig die ersten Berechnungen durchgeführt werden können. Entsprechend wird zu Beginn des CG-Verfahrens die rechte Seite b (als Residuum $r := Ax - b$ für $x = \vec{0}$) in den Koprozessorspeicher geschrieben mittels

```
co_memory_writeblock(mem, 0, b[0][0], dim*dim*sizeof(float));
```

Das Residuum ist ein dynamisch alloziertes eindimensionales Feld, um die Indizierung zu vereinfachen, bspw. mittels einer einfachen Zählvariablen. Blöcke könnten beliebiger, fester Länge sein, was aber zu unnötigem Steuerungsaufwand führen kann. Besser ist die Festlegung einer Blockgröße genau auf die Zeilenlänge. Den Messungen zufolge wird am meisten Leistungssteigerung erzielt, wenn die Residuumsmatrix als Ganzes übertragen wird. Diese Steigerung ist in Tabelle 5.25 ablesbar und liegt zwischen 4 % und 11 %. Weitere Messungen konnten aufgrund unbehebbaren Hardwareversagens nicht mehr durchgeführt werden. Folgend sind die weiteren theoretischen Überlegungen aufgeführt, wie noch mehr Leistungssteigerung erzielbar sein sollte.

Große Streaming-Puffer sollten gerade das Rückschreiben der berechneten Stencil-Werte etwas von der Berechnung entkoppeln können. Gewählt worden war eine Puffergröße von nur zwei Elementen. Angebracht sind Puffer in der Größe von 256 bis 2048 Elementen.

Langsame Operationen ersetzen. Die Division hat eine Dauer von 29 Takten ohne Pipelining und kann durch Multiplikation mit dem Kehrwert ersetzt werden, die als Pipeline mit wenigen Stufen implementiert wird. Neben der Ersetzung der Divisionen stellt gerade die Berechnung der Startpositionen der inneren Schleife für das Rot-Schwarz-Schema eine sehr langsame Operation in der Hardwareumsetzung durch Impulse C dar, wie sich durch Analyse der erzeugten Quelltexte und Zwischenformate sowie unter Verwendung der bereitgestellten Werkzeuge von Impulse CoDeveloper zeigte. Die für die Berechnung eingesetzte Modulo-Operation zwischen der inneren und äußeren

Schleife, welche die Startposition für das Rot-Schwarz-Schema bestimmt, benötigt mehr als einen Takt. Es genügt hingegen, die zu erwartenden Werte vorab auszurechnen und das Schema zu kodieren, dass die Startposition bei 1 oder 2 beginnt und dann abwechselnd den jeweils anderen Wert annimmt. Dies lässt sich einfach lösen durch entweder Code-Duplikation (einmaliges Ausrollen der Schleife) oder durch eine Zustandsvariable wie folgt:

```
start = 2;
for (i...)
  if (start==2)
    start = 1;
  else
    start = 2;

  for (j=start...)
```

Noch einfacher wäre dies direkt in Hardware umzusetzen durch Negieren der hintersten zwei Bits `"10"` der Startadresse.

Blockzugriff auf den Koprozessorspeicher von Hardware aus ist ebenso möglich wie von Software aus. Die drei mittleren Werte des Stencils können auf einmal aus dem Speicher ausgelesen werden, anstatt drei einzelne Transfers durchzuführen. Ähnlich Caches können unbenötigterweise eingelesene Daten später verwendet werden, wie folgend beschrieben.

Datenwiederverwendung. Der Ostwert des letzten Stencils ist auch gleichzeitig der neue Westwert des neuen Stencils und kann daher in einer/einem Variable/Register zwischengespeichert werden. Weitergehend ist auch der neue Ostwert der Südwert des Stencils vor $(dim - 1)/2$ Iterationen und der neue Nordwert der Westwert jenes älteren Stencils. Diese Werte sind also alle bereits geladen worden und könnten in Zeilenpuffern vorgehalten sein. Somit ist nur der Südwert noch nicht angefasst worden und muss als einziger aus einem externen Speicher bezogen werden.

Schnelle Indexberechnung. Wie vom Impulse StageMaster Explorer berichtet und verifizierbar, erfolgt die Berechnung der Speicherindize weder nebenläufig noch zeit- noch ressourceneffizient, sondern involviert Multiplikationen und Additionen und hat damit ungerechtfertigt hohen zeitlichen und ressourcentechnischen Aufwand. Die Lösung liegt in der Kombination und Erweiterung verschiedener Techniken. Mit den Zeilenpuffern wird bereits viel Adressberechnung gespart, unter anderem, weil diese eindimensional sind. Desweiteren entfällt etwas Adressberechnungsaufwand durch die Wiederverwendung des letzten Ostwertes. Zum Schreiben wird derselbe Index wie zum Lesen des Zentrums verwendet. Schließlich können die Indize für die Zugriffe auf den externen Speicher einfach per fortlaufender Inkrementierung berechnet werden. Es ist davon auszugehen, dass langfristig die Codekonvertierungswerkzeuge mittels statischer Codeanalyse die Berechnungen selbst zusammenfassen und vereinfachen werden, so dass die Anwender sich um solche Optimierungen zukünftig nicht mehr sorgen müssen.

Überlappte Berechnung der roten und schwarzen Stencils. Ursprünglich war die Verwendung des Rot-Schwarz-Schemas angedacht, um automatisierte Parallelisierung aufgrund der dann nicht mehr vorhandenen Datenabhängigkeiten zu erlauben, damit nicht der nächste Stencil auf die Vollendung des aktuellen warten müsste. Aufgrund des Streaming-Modells von Impulse C und der begrenzten Anzahl an Speichern und damit auch der begrenzten Speicherbandbreite wird der sequentielle Code in Impulse C ebenso zu sequentiellem Code in RTL übersetzt. In Hardware können voneinander unabhängige Operationen nebenläufig ausgeführt werden. Wurde der Stencil an Position `r[i][j]` berechnet, so kann für $i > 1,\ j > 1$ auch der Stencil der jeweils anderen Farbe an Position `r[i-1][j]` daraufhin berechnet werden.

Ausnutzen der Speicherbandbreite. Die Schnittstelle zum Systemspeicher ist bei der DRC AA2311 64 Bit breit. Ferner gibt es zwei sogenannte Reduced-Latency-RAMs bzw. Low-Latency-RAMs direkt auf der Koprozesorkarte mit 32 Bit Breite. Somit stehen insgesamt 128 Bit an Speicherbandbreite pro Taktzyklus zur Verfügung. Damit lassen sich zwei einfach genaue Gleitkommawerte pro Takt aus einem Speicher laden und die neuen Residuumswerte in den anderen Speicher schreiben ohne Ressourcenkonflikte und ohne damit einhergehende Wartezeiten. Da sich gleichzeitige Lese- und Schreibzugriffe auf den Speicher nicht gegenseitig beeinträchtigen, ist zu überlegen, ob auch die Eingabematrix in zwei Bereiche aufgeteilt werden kann, die parallel zueinander bearbeitet werden. Jede Einheit ist dann mit je einem Speicher für die Eingangsdaten und je einem für die Ausgangsdaten zu verbinden. Somit wird zusätzlich Parallelismus auf Datenebene ausnützbar.

Streaming anstelle direkter Schreibzugriffe auf den Koprozessorspeicher. Schließlich ist es gar nicht nötig, das Residuum r vorab in den Koprozessorspeicher zu kopieren. Viel mehr genügt es, die Werte zu streamen. Dadurch entfällt nicht nur die Kopieroperation, sondern auch die Indizierung des Koprozessorspeichers zum Lesen. Die Adressberechnung für Speicherzugriffe ist somit ausschließlich für das Schreiben nötig. Ab der dritten gestreamten Zeile der Eingabematrix kann mit der Stencil-Berechnung begonnen werden. Der Hardware-Prozess beschreibt dann den Koprozessorspeicher als einziger.

Ergebnis und Zusammenfassung

Impulse C ist eine Hochsprache, mittels derer es Domänen- und Anwendungsexperten möglich wird, rekonfigurierbare Hardware einzusetzen. Die ohne weitere Hardwarekenntnisse erzielbare Leistung liegt mindestens eine Größenordnung unter dem Erwarteten. In der Tat ist die zu erwartende Ausführungszeit eines nicht-parallelisierten Stencils auf einem FPGA auch etwa 20 mal geringer als die auf einer CPU, so dass beispielsweise ab 20 parallel rechnenden Stencil-Einheiten erst Leistungssteigerung durch den FPGA erwartbar ist.

Durch korrekten Einsatz von Pipelining unter Zuhilfenahme von Multiplikationen und schneller Speicherindizierung sowie Datenwiederverwendung lässt sich der Durchsatz auf max(Verzögerung Speicheranfragen, Berechnungsdauer) erhöhen, mindestens jedoch um Faktor 29. Durch die Überlappung der schwarzen und roten Stencil-Berechnungen ist asymptotisch Faktor 2 als Steigerung erzielbar. Die zweifach datenparalle Ausführung kann zu einer weiteren Geschwindigkeitssteigerung von Faktor 2 führen. Insgesamt beträgt der Unterschied zwischen der direkten, hardware-agnostischen Entwicklung und der Entwicklung durch einen erfahrenen Hardwareentwickler also zwei Größenordungen hinsichtlich der erzielbaren Leistung.

Um nicht auszuschließen, dass die Erfahrungen und Resultate ausschließlich dem verwendeten Werkzeug Impulse C zuzuschreiben sind, wurde ferner mit Riverside Optimizing Compiler for Configurable Computing (ROCCC) ein weiteres bekanntes Werkzeug zur Konvertierung von in C formulierten Quelltexten zu Hardwarebeschreibungen hinzugenommen. Die dabei gewonnenen Erfahrungen und Resultate sind im Folgenden geschildert.

5.8.5 Verwendung von ROCCC zur Erzeugung von Hardwarebeschreibungen für Stencil-basierte Verfahren

Die hiesigen Studien wurden mit dem Riverside Optimizing Compiler for Configurable Computing (ROCCC) in den Versionen 0.5.1 und 0.5.2 durchgeführt, während mittlerweile Version 2.0 und neuer aufwarten. ROCCC unterscheidet zwischen Systemen und Modulen. Systeme können aus mehreren Modulen bestehen. Beispielsweise lässt sich eine Matrixmultiplikation als Serie von Matrix-Vektor-Multiplikationen oder Skalarprodukten implementieren. Letztere werden also als Module implementiert. Module bestehen aus mehrfach verschachtelten Schleifen, wobei Programmcode nur innerhalb der innersten Schleife stehen darf.

Ein Stencil-Modul besteht aus der Spezifikation der Eingabe- und Ausgabematrizen und läuft geradlinig über die gesamte Eingabe hinweg, wobei jedesmal explizit beim zweidimensionalen Stencil

die dafür benötigten fünf Werte eingelesen werden. Wiederverwendung zuletzt verwendeter Werte über eine sogenannte Feedback-Variable ist zwar prinzipiell möglich, scheitert im Falle des Stencils aber daran, dass das Zurücksetzen der Feedback-Variable in der äußeren Schleife erfolgen müsste, was von ROCCC her nicht übersetzbar war. Die Alternative, mittels Überprüfung des äußeren Schleifenzählers in der innersten Schleife dasselbe zu erreichen, führte jedoch dazu, dass die Feedback-Variable nicht mehr als solche erkannt wurde, unter anderem da nun mehrere Pfade existierten. Somit gelangt der ROCCC-verwendende Entwickler zum Partitionierungsproblem: Das Stencil-Modul muss explizit mit diesen benötigten fünf Operanden versorgt werden von einem System aus, welches sich um die Matrixzugriffe und Datenwiederverwendung kümmert.

Die Fehlersuche gestaltet sich recht aufwendig und bedarf häufig der Betrachtung des Zwischencodes im SUIF-Format oder des erzeugten RTL-Codes in VHDL. Beide sind aufgrund der automatischen Variablenbenennung kaum verständlich. Der VHDL-Code lässt sich zwar mit Modelsim simulieren; die Simulation ist allerdings auch die einzige Möglichkeit zu schauen, wie ROCCC die Bestandteile der Stenciloperation zeitlich angeordnet hat, wie hoch der Durchsatz und die Latenz sind. Das händische Erstellen von Testbenches oder -skripten ist unumgänglich. Damit ist ROCCC nicht mehr für Anwendungsentwickler wie etwa Domäneneexperten als Werkzeug zur Hardwarebeschreibung sinnvoll, zumal es weitläufiger Anstrengungen hinsichtlich Software-Partitionierung und Hardware-Software-Codesign bedarf.

Ein weiteres Problem ist, dass für jedes unterschiedliche FPGA-System ein Hardwareexperte die Anbindung von ROCCC an die Systeminfrastruktur bestehend aus Firmware, HyperTransport-Anbindung, Schnittstelle zum Application Engine Hub oder ähnlichem entwickeln muss, bevor die generierte Hardwarebeschreibung produktiv verwendet werden kann. Entsprechend beschränkt sich die Zahl der mit ROCCC programmierbaren FPGA-Systeme auf einige wenige Fälle. ROCCC bleibt eine gute Wahl für hardwarenahe Entwickler, schnell eine Hardwarebeschreibung im Sinne des Rapid Prototypings zu entwerfen oder sich auf abstrakter Ebene bereits um Partitionierung zu kümmern, bevor die händische Erstellung erfolgen wird.

5.8.6 Ergebnis und Zusammenfassung der Untersuchungen zur Hardware-Implementierung von Stencils mittels Hochsprachen

Um performante Hardwareschaltungen, die wenig Rechenzeit und wenige Ressourcen benötigen sowie Beschleunigung liefern, mithilfe von Hochsprachen zu erstellen, bedarf es vier Dingen. Erstens muss der zu beschleunigende Algorithmus die wesentlichen Kriterien nach Park et al. erfüllen [212]: Bei einem für Rot-Schwarz-Verfahren eingesetzten Stencil sind mindestens zwei Stencil-Berechnungen parallel auf einem Bereich durchführbar, und ein Bereich kann in mehrere Unterbereiche aufgeteilt werden, um parallel mehrere Bereiche zu berechnen; durch geschickte Zwischenspeicherung der Daten können neun Gleitkommaberechnungen pro einem einzelnen Speicherzugriff durchgeführt werden; pro berechnetem Pixel sind genau diese neun Gleitkommaoperationen nötig; pro Pixelberechnung sind ein Lese- und ein Schreibzugriff auf den Speicher nötig; der Kontrollfluss kann vollständig eliminiert werden durch einen mit Nullwerten initialisierten Rand (Dirichlet-Bedingung) bis auf die einzuhaltenden Schleifengrenzen; bei mehreren parallel berechneten Unterbereichen bestehen Abhängigkeiten hinsichtlich des Austauschs der Randwerte, eine Vereinfachung ohne Synchronisation würde das Block-Jacobi-Verfahren anstelle des Gauß-Seidel-Verfahrens als Vorkonditionierer implementieren. Folglich sind die Kriterien nach Park et al. erfüllt oder zumindest erfüllbar.

Zweitens benötigt der Entwickler tiefgehendes Verständnis von Hardwarestrukturen und -implementierungen, insbesondere um die Kriterien wie genannt zu erfüllen. Wesentlich ist bei der Entwicklung und Implementierung die Behandlung der Daten (vgl. Strattons et al. „Technique 1: Data layout transformation“ [248]): Transport, Datenfluss, Wiederverwendung, zeitliche und räumliche Lokalität, geeignete Speicherstrukturen.

Drittens ist iterative Entwurfsraumexploration für jeden einzelnen Schritt nötig, was die zeitaufwendige Synthese und Vermessung am echten System miteinschließt. Beim manuellen Entwurf mit VHDL/Verilog kann aufgrund der Möglichkeit zur Hardwaresimulation mittels Software sehr lange auf die Synthese verzichtet werden. Demgegenüber kann bei hochsprachlichen Ansätzen lediglich die abstrakte hochsprachliche Beschreibung nativ übersetzt und emulierend ausgeführt werden.

Viertens muss die Schnittstelle der erzeugten Schaltungen an die verfügbare Hardware vorhanden und so implementiert sein, dass mit wenig Hardwareaufwand seitens dieser Schaltungen hohe Datenübertragungsraten zustande kommen können. Für Schaltungen, die mit ROCCC generiert wurden, muss eine Anbindung an die zur Verfügung stehende Hardware häufig erst geschaffen werden. Für Impulse C stellen Drittfirmen wie Synective Labs PSPs für FPGA-Systeme zur Verfügung.

Hochsprachen zur Hardwarebeschreibung eignen sich aufgrund dieser Anforderungen und erkennbar an den erzielten Ergebnissen nicht für Anwendungsexperten, schnell und bequem Nutzen aus rekonfigurierbarer Hardware zu ziehen.

5.8.7 Compiler-basierter Ansatz für stencil-basierte Vorkonditionierer

Auf der Convey HC-1 (vgl. Abschnitt 2.1) gibt es drei verschiedene Möglichkeiten, die FPGAs zur Unterstützung numerischer Anwendungen zu untersuchen und auszunutzen. Erstens können FPGA-Entwickler ihren Code in herkömmlichen Hardwarebeschreibungssprachen wie Verilog oder VHDL implementieren und eine eigene Personality erzeugen. Die zweite ist die Programmierung mit einer Hochsprache, die zu einer RTL-Beschreibung konvertiert wird, wie dies im vorangegangenen Abschnitt untersucht wurde. Mitrion-C wird unterstützt und mittlerweile von akademischer Seite aus auch ROCCC. Schließlich liefert Convey einen Compiler aus, der Programmcode in Hochsprachen wie C oder Fortran und insbesondere Schleifen zu Vektorcode transformiert, der auf einer Convey-eigenen Vektor-Personality ausgeführt wird. Ähnlich dazu können auch die Convey BLAS-Routinen mittels der zugehörigen Software-API verwendet werden, welche ebenso die Vektor-Personality aufrufen. Ergänzend zu den vorab geschilderten Untersuchungen zur Eignung von Konvertierungswerkzeugen wurde der dritte, compiler-gestützte Ansatz untersucht hinsichtlich seiner Eignung für Anwendungsentwickler zur Nutzung rekonfigurierbarer Hardware. Als Hochsprache wurde dabei C verwendet, die seitens Convey unterstützte Alternative ist Fortran.

Portierung des Vorkonditionierers

Gemäß Conveys Beispiel zur Durchführung von Stencil-Operationen wurde der Rot-Schwarz-Vorkonditionierer für das Poisson-Problem implementiert. Die Stencil-Koeffizienten sowie die Adressen der Matrizen werden als Datenwerte übergeben. Beide sollten anschließend in Registern des Skalarprozessors des Convey AE Hubs oder gar der Vektorprozessoren liegen. Die inneren Schleifen der Berechnungen der roten und schwarzen Werte wurden einmal ausgerollt, um die Modulo-Operation oder Zustandsabfragen einzusparen. Die Divisionen bei der Berechnung von dx können wie folgt zu einzelnen Koeffizienten zusammengerechnet werden:

$$\begin{aligned} dx = (r - (aN * deltaxN + aS * deltaxS + aE * deltaxE + aW * deltaxW)/aD)/aD \\ = 1/aD * r + (-aN)/aD^2 * deltaxN + (-aS)/aD^2 * deltaxS \\ +(-aE)/aD^2 * deltaxE + (-aW)/aD^2 * deltaxW, \end{aligned}$$

wobei r das alte Residuum an der aktuellen Stelle ist, $a\{D/E/N/S/W\}$ die aus der Poisson-Gleichung ermittelten Gewichte, und $deltax\{E/N/S/W\}$ diejenigen Werte, die r umgeben und entweder Residuumswerte oder bereits aktualisierte neue Werte sind. Damit können echte Stencil-Operationen für die Berechnungen der roten und schwarzen Elemente durchgeführt werden.

Alle beim CG-Verfahren benötigten Matrizen außer der Ergebnismatrix $z = dx$ aus der Vorkonditionierung werden auf dem Host alloziert. Die Eingabematrizen müssen zum Koprozessor migriert oder kopiert werden, um nicht unter Leistungseinbußen durch einzelne Seitenzugriffsfehler leiden zu müssen. Dazu stellt Convey eigene Pragmas und Laufzeitroutinen bereit (vgl. Abschnitt 2.1.7). Die Ergebnismatrix wird vom Koprozessor auf den Host automatisch migriert. Da diese Rückmigration in der MMU des Hostprozessors behandelt wird, ist dies ohne nennenswerte zeitliche Nachteile möglich.

Die Verwendung von OpenMP auf dem Zweikernprozessor der Convey HC-1 erzielt eine Beschleunigung von 1,49. Die Geschwindigkeitssteigerungen durch Koprozessorausführung gegenüber mehrfädiger Ausführung auf dem Host mit OpenMP sind in Tabelle 5.26 aufgeführt. Die Beschleunigung von 2,05 für die Gauß-Seidel-Vorkonditionierung bei 100 Iterationen ist beachtlich, wenn

Iterationen	Auflösung (einzelne Seite)				
	512	1024	2048	4096	8192
Rot-Schwarz-SGS					
1	0,28	0,54	1,17	1,63	1,87
10	0,51	1,04	1,95	1,58	1,57
100	0,90	1,32	2,05	1,52	1,56
Jacobi					
1	0,37	0,54	1,14	1,58	1,70
10	0,51	0,97	1,69	1,42	1,37
100	1,03	1,36	1,93	1,34	1,33
Identität (Matrix kopieren)					
1	0,28	0,46	0,81	1,05	1,01
10	0,46	0,72	1,03	0,90	0,90
100	0,81	0,84	1,11	0,90	0,88

Tabelle 5.26: Beschleunigung unterschiedlicher Vorkonditionierer durch Koprozessorausführung gegenüber mehrfädiger Ausführung auf einer CPU.

man bedenkt, wie einfach der Ansatz ist. Dennoch wird in der Regel noch mehr Nutzen von rekonfigurierbarer Hardware erwartet als lediglich $2,05 \cdot 1,49 \approx 3,05$-fache Beschleunigung gegenüber einfädiger Ausführung.

Portierung des gesamten CG-Verfahrens

Bei der Verwendung des compiler-basierten Ansatzes von Convey wurden die Aspekte der Auslagerung rechenintensiver Anwendungsteile in rekonfigurierbare Hardware, der Einbindung der Hardware in die Anwendung, der geeigneten Granularität und der Verringerung des Datentransfers untersucht. Nicht untersucht wurde daher die überlagerte Berechnung der roten und schwarzen Werte, da diese Optimierung auch in parallelisierten Implementierungen anwendbar ist. Ebensowenig wurde untersucht, wie die Implementierung des CG-Verfahrens direkt auf die Convey HC-1 und die Vektorpersonality zugeschnitten werden könnte, da dies ebenso gegenüber einer direkt auf OpenMP zugeschnittenen Implementierung verglichen werden müsste und nicht mehr dem komponentenbasierten, bibliotheksnutzenden Ansatz gerecht werden würde.

Mittels folgender Entscheidungen war es möglich, das gesamte CG-Verfahren unter Verwendung der Jacobi- und Gauß-Seidel-Vorkonditionierer compiler-gestützt auf die Convey HC-1 zu portieren:

- Daten mittels `cny_cp_malloc` koprozessornah alloziert.
- Koprozessorregionen ausgewiesen; Nichtvorhandensein von Datenabhängigkeiten erklärt (wo angebracht); Variablen als ein-elementige Felder deklariert.
- Die CPU handhabt nur die Koprozessorsteuerung, ohne die Daten auf dem Koprozessor anzufassen und dadurch eventuell automatisch zu migrieren.

Der Convey-Compiler kann versuchen, den Anwendungscode automatisch zu vektorisieren. Darüber lässt sich auch herausfinden, welche Angaben des Entwicklers nicht vielversprechend sind und daher entfernt werden sollten oder wo der Entwickler Code-Umstellungen untersuchen sollte hinsichtlich Ausführbarkeit mit der Vektorpersonality. In hiesigem Falle stellte sich heraus, dass die Funktion zur Skalierung von Matrizen aus der in Abschnitt 5.1.1 vorgestellten Matrixbibliothek aufgrund vorhandener Sprünge nicht beschleunigt auf dem Koprozessor ausführbar sein würde. Durch Entfernen der Abfragen und Sprünge sowie durch sequentielle Adressierung der Elemente konnte auch diese Funktion schlussendlich vektorisiert werden. Dadurch können alle Daten auf Koprozessorseite gehalten werden; es müssen keine unnötigen Datentransfers zwischen Host und Koprozessor mehr stattfinden. Wie aus Abb. 5.43 erkennbar wird, können die Convey HC-1 und der Compiler ab Datengrößen von etwa 1024×1024 ihre Leistung voll ausspielen. Die Version mit vollständig portiertem CG-Verfahren ist nahezu doppelt so schnell wie die mit lediglich ausgelagertem Vorkonditionier. Ohne Vorkonditionierer ist das Verfahren am langsamsten, mit dem Rot-Schwarz-Vorkonditionierer und vollständiger Auslagerung auf den Koprozessor am schnellsten. Mit der Verlagerung auf den

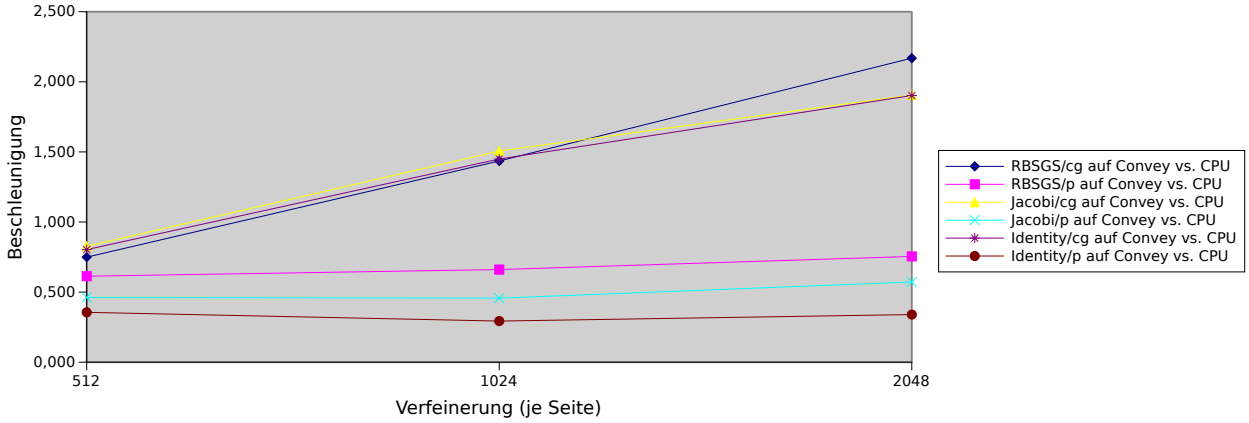


Abbildung 5.43: Beschleunigung des CG-Verfahrens durch Auslagerung des Rot-schwarzen Gauß-Seidel-Vorkonditionierers (RBSGS), Jacobi-Vorkonditioniers und ohne Vorkonditionierer (Identity), jeweils nur Vorkonditionier ausgelagert (/p) oder gesamtes Verfahren (/cg).

Koprozessor werden deutlich mehr Iterationen benötigt, bis das Konvergenzkriterium erreicht ist. Auch hier ist der rot-schwarze Gauß-Seidel-Vorkonditionierer am besten mit nur halb so vielen Iterationen wie die übrigen Vorkonditionierer auf dem Koprozessor.

Zusammenfassung des compiler-basierten Ansatzes

Der compiler-basierte Ansatz ermöglicht ebenso wie der Ansatz über Quelltextkonverter, rekonfigurierbare Hardware als Anwendungsentwickler einzusetzen. Der dazu nötige Aufwand ist durch die Annotation mit Pragmas und durch die Analyse anhand der Autovektorisierung wesentlich geringer als bei Hochsprachen und Quelltextkonvertern. Vor allem entfallen erneute Synthesen und die Konfiguration der Hardware. Bestehender Programmcode kann nahezu unverändert verwendet werden ohne Zerlegung in Hardware- und Softwareteile, ohne Streaming, ohne weitere Software-Hardware-Schnittstelle. Wie schon mehrfach beleuchtet, leidet die Genauigkeit insbesondere beim Skalarprodukt unter der Auflösung der Mantisse mit nur 23 bzw. 24 Bits, wodurch die Anzahl an Iterationen steigt. Die Erkenntnis aus dieser Untersuchung ist, dass Anwendungsentwicklern ein Programmiermodell zur Verfügung gestellt werden muss, welches sowohl die Datenübertragung nahezu vollständig selbst übernimmt, wenige weitere Werkzeuge benötigt als auch direkt in bestehende Anwendungen integrierbar ist.

5.8.8 Gewonnene Erkentnisse aus dem Einsatz von Hochsprachen

Die Entwicklung von Schaltungen benötigt auch beim Einsatz von Hochsprachen noch die Transformation eines Algorithmus bzw. Verfahrens in für Hardware geeignetere Strukturen, z.B. Zusammenfassen arithmetischer Operationen, Ersetzen arithmetischer Operationen, Entfernen unnötiger Berechnungen und unnötigen Kontrollflusses, Ausrollen von Schleifen, Überlappen von Datentransfers und Berechnungen sowie mehrfache Unterbringung von Rechenkernels wie etwa einer Multiplizieren-Akkumulieren-Einheit oder einer Stencil-Einheit. Der Anwendungsentwickler sollte daher von der Entwicklung von Hardwareschaltungen vollständig entlastet werden. Dazu muss die Nutzung rekonfigurierbarer Hardware vereinfacht werden hingehend zum bloßen Aufruf der Hardware, wie es beispielsweise mit der BLAS-Implementierung von Convey möglich ist. Um dennoch gewisse Flexibilität hinsichtlich der Unterstützung mehrerer Anwendungen zu bieten, empfiehlt es sich, mehrere Spezialoperationen zusammen anzubieten anstelle einer generischen Vektor-Personality, welche im Wesentlichen auf die Durchführung arithmetischer Operationen auf Vektordaten zugeschnitten ist. Der Aufruf der Spezialoperationen ist über geeignete Schnittstellen, wie etwa über einen Funktionsaufruf, anzubieten. Die Spezialoperationen sind als Komponenten zu implementieren und unter anderem für möglichst optimalen Datentransfer und die Datenhaltung selbst zuständig. Die

Komponenten müssen dann nicht mehr jedes mal neu synthetisiert werden. Bei der Entwurfsraumexploration ist daher lediglich die Programmierung in Software zu ändern wie beim Einsatz des Convey-Compilers für die Vektor-Personality. Die Schnittstelle der Komponenten zur Hardware und letzendlich auch zum Prozessor muss von den Systemherstellern oder Hardwareentwicklern geliefert werden.

5.9 Zusammenfassung und Erkenntnisse

Es wurden mehrerlei Ansätze auf unterschiedlichen Granularitätsstufen untersucht, wie heterogene Systeme mit Mehrkernprozessoren oder rekonfigurierbarer Hardware für numerische Anwendungen nutzbar sein können. Auf der untersten Stufe können einzelne Instruktionen verlagert werden. Das exakte Skalarprodukt nach Kulisch mit einem 768 bzw. 4288 Bit breiten Festkommaregister scheint zunächst hervorragend zur Verlagerung in rekonfigurierbare Hardware geeignet. Da jedoch eine lange Pipeline mit zahlreichen, über mehrere Takte hinweg bestehenden Konflikten bei meist nur drei oder vier parallelen Operationen pro Pipelinestufe in der Umsetzung entsteht, lässt sich Nutzen erst mit höherer Granularität auf der Funktionsebene erzielen, z.B. mit Matrix-Vektor-Multiplikationen oder gar Matrixmultiplikationen. Der Platzbedarf dafür ist allerdings enorm, und die nahtlose Datenübertragung kritisch. Dennoch lässt sich das exakte Skalarprodukt durch Verlagerung in rekonfigurierbare Hardware auf leistungsfähigen heterogenen Systemen gegenüber der Ausführung in Software stark beschleunigen.

Unter dem Gesichtspunkt der Programmierbarkeit wurden verschiedene hochsprachliche Ansätze untersucht, Leistungssteigerung dadurch zu erzielen, dass rechenintensive Anwendungsteile in rekonfigurierbare Hardware ausgelagert werden, z.B. auf Funktionsebene. Der Ansatz mit Quelltextkonvertern funktioniert prinzipiell; es ist jedoch sehr aufwendig, auf diese Weise auch Leistungssteigerungen zu erzielen. Der compiler-basierte Ansatz bedeutet hingegen weit weniger Aufwand und erzielt auch die gewünschte Leistungssteigerung. Dies wurde unabhängig von Meyer et al. bestätigt, die mit Conveys Vektorpersonality vergleichbare Leistung zu einem Achtkern-Server erreichen [198]. Es zeigte sich dabei aber auch, dass aufgrund des für die Datenübertragung nötigen Zusatzaufwands die Verlagerung größerer Anwendungsteile bzw. der gesamten Anwendung in Hardware wesentlich vielversprechender ist, so dass eine auf Vektoroperationen ausgelegte Personality nicht genügt.

Diese erhöhte Granularität ließe sich auch hervorragend durch ein Rahmenwerk zur Ausführung von Funktionsbausteinen wie Stencils und Vorkonditionierern erreichen. Dabei ist das Problem der Datenübertragung zu lösen und eine geeignete Methode zur Ansteuerung einzelner oder mehrerer Spezialeinheiten zu finden, die von Anwendungsentwicklern leicht beherrschbar ist. Im folgenden Kapitel werdem daher die Untersuchungen zu geeigneten Rahmenwerken und die finale Implementierung eines geeignet scheinenden Rahmenwerks auf der Convey HC-1 vorgestellt sowie die zugehörigen Syntheseergebnisse.

Kapitel 6

Steuerungen von FPGAs und Anbindungen an Anwendungen und Programmierumgebungen

Im vorigen Kapitel wurde ersichtlich, dass direkte Ansätze, bei denen der Entwickler bestehenden Code nur um Angaben erweitert oder mittel- bis grobgranulare Koprozessoren in die Anwendung einbindet – möglichst datengetrieben –, vielversprechender sind als Quelltextkonvertierer und feingranulare Koprozessoren auf Instruktions- oder Funktionsebene. Es werden also Rahmenwerke benötigt, die höhere Funktionalität dem Anwender bereitstellen. Innerhalb der Rahmenwerke sollen Komponenten mit Spezialoperationen wie beispielsweise einem exakten Skalarprodukt bereitgestellt werden.

6.1 Grundlagen

Die Grundlagen für die in diesem Kapitel vorgestellten Konzepte und Architekturen sind die *Datengetriebene Programmausführung und das Datenflussprinzip*, die *komponentenbasierte Entwicklung* und der *Task-Parallelismus*.

6.1.1 Vergleich einiger Programmiermodelle

Einen Vergleich einer Vielzahl an kommerziellen Programmiermodellen und Programmierstandards stellen Membarth et al. an [194]. OpenMP und Intel TBB sind ihren Untersuchungen nach besser benutzbar als Cilk, RapidMind oder OpenCL, auch wenn sie für den von ihnen gewählten Anwendungsfall der Bildregistrierung keine explizite Unterstützung bieten. Der Mehraufwand bei taskparalleler Ausführung fällt durchweg geringer aus als bei feingranular datenparalleler Ausführung. TBB bringt am wenigsten Aufwand mit und erzielt auch die besten Ausführungszeiten.

Charm, Cilk und OpenMP werden verglichen von Xia et al. gegenüber ihrem eigenen Ansatz [283]. Besonders an ihrem Ansatz ist die Verwaltung vieler Threads in Multi- und Manycore-Rechnern auf hierarchische Weise mittels Bündelung in Gruppen und Teilung derselben zum Zwecke der taskparallelen Ausführung von Anwendungen, die mittels gerichteter, azyklischer Graphen[1] beschrieben sind. Je Gruppe gibt es einen Manager-Thread, der die Synchronisation mit einem Super-Thread und die Verteilung der gruppeneigenen Last vornimmt. In jeder Gruppe gibt es Queues mit Arbeitsaufträgen, die von den Arbeitern[2] nacheinander oder auch parallel zueinander ausgeführt werden, wenn keine Abhängigkeiten vorliegen. Zur Beachtung der Abhängigkeiten wird ein Zähler eingesetzt, der Null sein muss, bevor ein Auftrag verarbeitet werden darf. Ferner enthält ein Auftrag Verweise auf von ihm abängige weitere Aufträge, deren Zähler nach vollständiger Ausführung vom Manager-Thread dekrementiert werden. Anhand eines Indikators wird laufend die Arbeitslast ermittelt und bei zu hoher Arbeitslast die Gruppe mit einer anderen zusammengelegt bzw. bei zu geringer Last aufgeteilt in zwei Gruppen, so dass die zweite Gruppe eine neue Queue mit Aufträgen bearbeiten kann. Die auszuführende und mit Charm, Cilk und OpenMP zu vergleichende Anwendung wird von ihnen in voneinander abhängige Tasks aufgeteilt und nach unterschiedlichen

[1] engl. directed, acyclic graph (DAG)
[2] engl. workers

Scheduling-Strategien ausgeführt, unter anderem zentral vs. verteilt, dediziert verwaltet vs. gemeinsam zugreifbar, und schließlich mit Work-Stealing. Insgesamt schneidet ihr Ansatz des hierarchischen, gruppen-basierten, adaptiven Schedulings wesentlich besser ab. Die beste Anzahl an Threads bzw. Gruppen hängt allerdings stark von den Taskgraphen bzw. Anwendungen ab. Zur Bestimmung der Arbeitslast müssen die Knoten des DAGs jedoch vorab schon annotiert sein mit der voraussichtlich erzeugten Arbeitslast. Außerdem ist es mittels der Zähler nicht möglich, Pipeline-Parallelismus auszunutzen, da immer gewartet wird, bis der erzeugende Auftrag vollständig ausgeführt ist. Gerade unter der Annahme von gemeinsamen Caches ist es sinnvoll, die erzeugten Daten auch gleich weiterzunutzen, was jedoch in diesem strengen Datenflussausführungsmodell nicht möglich ist.

Mittels Benchmarksuites lassen sich umfangreichere Aussagen über die Leistungsfähigkeit von Systemen und auch Programmiermodellen erhalten. Das taskparallele OMPSs ist etwa zwei Mal schneller als die Verwendung von Posix-Threads, gemessen mit einer zur Bewertung von parallelen Programmiermodellen entwickelten Benchmarksuite [11].

Den zitierten Artikeln zufolge sind taskparallele, abstrakte Programmier- und Ausführungsmodelle bevorzugenswert sowohl im Hinblick auf Entwicklungsaufwand als auch auf erzielbare Leistung. Die vorliegende Dissertation zielt daher darauf ab, zur Nutzung von heterogenen, rekonfigurierbaren Mehrkernsystemen ein abstraktes, taskparalleles Modell zu nutzen, das über die Software-Hardware-Grenze hinweg nutzbar ist.

6.1.2 Datengetriebene Programmausführung und das Datenflussprinzip

Die datengetriebene Ausführung von Operationen und Algorithmen wird laufend in Programmiermodellen, Anwendungen und Hardware eingesetzt. Sie ist wesentlich bei der Verlagerung von Algorithmen in Koprozessoren oder Beschleunigerhardware, denn sie ist dazu geeignet, das angesprochene Problem der Datenübertragung anzugehen, indem Berechnungen ab Verfügbarkeit der Daten erfolgen. Ferner muss die Datenlokalität nicht gezielt angegangen werden. Dieser Ansatz ist daher sehr anwenderfreundlich, also auch für Domänenexperten als Anwendungsentwickler geeignet.

Auf der datengetriebenen Ausführung basierend wurde das Datenflussprinzip entwickelt, das über die letzten Jahrzehnte hinweg immer wieder umgesetzt wurde. Diesem Prinzip liegt zugrunde, dass die Daten von Knoten zu Knoten fließen und ausführbereite Knoten dann auf einer freien Ausführungseinheit ausgeführt werden, wobei die Knoten des Graphens entsprechend der gewählten Granularität einzelne Mikroprozessorinstruktionen sein können bis hin zu komplexen Funktionen. Das Modell des Datenflusses ist darüber hinaus auch geeignet, für die Ausführungszeit von Anwendungen untere Schranken zu bestimmen, indem unabhängige Graphen optimal geschedult werden und die Startzeitpunkte eines Subgraphens durch die Ankunfts- und Berechnungszeiten in vorhergehenden Subgraphen bestimmt werden [109].

Bereits John von Neumann hat seine zu programmierenden Algorithmen zunächst als Flussdiagramm formuliert [94] und dann daraus in mehreren Schritten das Programm entwickelt. Aufgrund der zugrundeliegenden, nach ihm benannten Architektur des verwendeten IAS-Computers wurden die Daten jedoch einzeln per Ladebefehl aus dem Hauptspeicher geholt und daraus später die kontrollflussorientierten Architekturen entwickelt. Treleaven et al. stellen einige wichtige Unterscheidungsmerkmale von Kontrollflussrechnern und Datenflussrechnern auf [261]. Im Textbuch „Processor Architecture: From Dataflow to Superscalar and Beyond“ [238] beschreiben die Autoren die wichtigsten Prinzipien des Datenflusses einhergehend mit der Entwicklung von Datenflussarchitekturen. Neben den von Treleaven et al. aufgestellten Klassifikationen beschreiben sie ebenso die zum damaligen Zeitpunkt vorhandenen Datenflussarchitekturen, ihre Herkunft, Entwicklung und Programmierung.

Datengetriebene Ausführung

Operationen datengetrieben auszuführen, bedeutet, auf das Vorhandensein der Operanden zu warten und ab Verfügbarkeit der Operanden auch sofort die abhängige Operation auszuführen. Die Auflösung von Abhängigkeiten nach Tomasulos Algorithmus [260] in Superskalarprozessoren ist

gleichermaßen datengetrieben wie die Zwischenspeicherung bzw. Bereithaltung von Daten in Hardwareschaltungen, wie es z.B. bei der Beschleunigung von Algorithmen mittels FPGAs benötigt wird. Demgegenüber steht der bedarfsorientierte Ansatz, bei dem ein Knoten erst dann aktiviert wird, wenn seine Ausgabe benötigt wird.

Daten bzw. sogenannte *Tokens*[3] werden von *Produzenten* erzeugt und von *Verbrauchern* verbraucht. Im Falle einer 1:1-Beziehung muss der Verbraucher wissen, ob das ihm überreichte Datum auch wirklich gültig ist, und umgekehrt muss der Verbraucher den Produzenten benachrichtigen, wenn er dieses Datum aufgenommen hat und ein neues Datum verarbeiten könnte. Ein Produzent kann auch Daten für mehrere Verbraucher erzeugen bzw. ein Verbraucher Daten von mehreren Produzenten verwenden. Dies ist solange unproblematisch, solange sich nicht zwei Verbraucher um dieselben Daten bemühen[4], da in diesem Fall sichergestellt werden muss, dass nur genau ein Verbraucher atomar und exklusiv alle Daten verbraucht, bevor der andere Verbraucher dies versuchen würde.

Datenfluss

Bei der Berechnung von Algorithmen oder Anwendungen nach dem Datenflussprinzip werden Anweisungen (Knoten im zugehörigen Datenflussgraphen) nur dann ausgeführt, wenn gültige Eingangsdaten vorliegen, die als Tokens zwischen den Knoten transportiert werden [160]. Damit kann auf ein zentrales Taktsignal verzichtet werden, und die Arithmetik und Logik können asynchron ausgeführt werden unter Verwendung von Bestätigungssignalen[5] nach dem Handschlagprotokoll[6]. Die Modellierung ähnelt der von Petri-Netzen. Es gibt meistens eine *Übereinstimmungseinheit*[7], die dafür zuständig ist, eingehende Tokens mit auszuführenden Operationen abzugleichen. Über Warteschlangen[8], insbesondere FIFO-Speicher, werden die auszuführenden Operationen mitsamt benötigten Operanden an die Ausführungseinheiten übermittelt. Dennis, der zu den ersten Forschern im Bereich des Datenflusses gehört, gibt einen Rückblick dazu [79].

6.1.3 Komponentenbasierte Entwicklung

Software besteht häufig aus mehreren Komponenten, die nach Möglichkeit voneinander unabhängig sind und separat entwickelt und getestet werden. Programme lassen sich auf der Grundlage von Komponenten auf abstrakter Ebene beschreiben und anschließend auf Basis dieser Beschreibung entweder Syntheseprogramme für ausführbare Dateien erhalten [264] oder die ausführbaren Dateien selbst [21]. Die Komponenten bündeln intuitiv greifbare Funktionalität. Um portabel zu sein, ist die Implementierung der Komponenten für verschiedene Architekturen nötig sowie ein Mechanismus, mit dem zur Übersetzungs- oder zur Laufzeit die jeweils passende Implementierung gewählt werden kann. Im Kontext paralleler Programmierung wurde dieser Ansatz von Bonorden et al. weitergeführt [38]. Die Nutzung des komponentenbasierten Konzepts wurde von Palatin et al. dazu eingesetzt, den Grad der parallelen Ausführung von Komponenten zur Laufzeit anzupassen [210]. Dabei werden jedoch ausschließlich Prozessoren mit Simultaneous Multi-Threading (SMT) berücksichtigt. In einer komponentenbasierten Architektur für wissenschaftliches Rechnen [6] verwenden die Komponenten eine wohldefinierte Schnittstelle und können auf unterschiedlichen Teilnehmern in einem Cluster ausgeführt werden. Dabei dürfen auch unterschiedliche Implementierungen der gleichen Funktionalität vorliegen. Derart könnten sich auch besondere Funktionseinheiten einbetten lassen. Aus Anwendungssicht muss nur eine Stellvertreterfunktion mit einer bestimmten Schnittstelle (Port) aufgerufen werden; die passenden Implementierungen können in beliebigen Sprachen für beliebige Ausführungseinheiten entwickelt sein. Die komponentenbasierte Programmierung und Entwicklung von heterogenen Systemen konnte mit Gezel bereits erfolgreich durchgeführt werden [232]. Dies wurde weitergehend mit FPGAs untersucht [54]. Dabei wurde festgestellt, dass die Änderung einer Software zu einer komponentenbasierten Architektur hin bei Ausführung auf der CPU bereits einen Nachteil von 9 % zur Laufzeit einbringt, die Ausführungszeit des Kernels

[3] Datensatz mit Gültigkeitsanzeige und ggf. Zieladresse
[4] engl. work stealing
[5] engl. Acknowledgement Signals
[6] engl. Handshaking
[7] engl. Matching Unit
[8] engl. Queues

auf dem FPGA steigt sogar um 10 %. Dennoch ist der komponentenbasierte Ansatz attraktiv, da die Hardwareinitialisierung und das Übertragen der Arbeitsdaten in Konstruktor und Destruktor versteckt werden können.

6.1.4 Task-Parallelismus

Ein bereits länger vorhandenes Programmiermodell ist die Thread-Programmierung. Kapselt man Funktionalitäten in Threads, so lässt sich einerseits Wiederverwendbarkeit und andererseits hohe Parallelisierbarkeit (Funktions-/Task-Parallelismus) erreichen, insofern die Threads bzw. Kapselungen thread-sicher sind, also mehrfach gleichzeitig aufgerufen und ausgeführt werden können[9] [192]. Auch bei nicht thread-sicheren Implementierungen lässt sich über das Ausführen mehrerer unterschiedlicher Threads noch teils erheblicher Nutzen aus mehreren Prozessoren oder Prozessorkernen im Sinne von erzielbarem Speedup ziehen. Diese Funktionalitäten können als Komponenten auf Funktionsbene betrachtet werden. Die Nutzung rekonfigurierbarer Hardware als spezielle Komponente stellt eine attraktive Möglichkeit dar, die interne Implementierung zu verstecken und die zur Verfügung gestellte Funktionalität optimal in die Gesamtausführung zu integrieren.

Cilk [34]/Cilk++/Intel Cilk Plus [183, 223], Intel Threading Building Blocks (TBB) und Open MP 3.0 sind Beispiele für taskparallele Programmiermodelle. Ein weiteres taskparalleles Programmiermodell ist Concurrent Collections [53], mit dem sich Operationen der Linearen Algebra verknüpfen lassen. Aufgerufene Operationen oder Funktionen werden parallel zueinander ausgeführt, wenn keine Abhängigkeiten zwischen ihnen vorliegen.

Liegen Abhängigkeiten vor, so ist es mit Pipeline-Parallelismus als Sonderform von Task-Parallelismus möglich, die von einer Funktion erzeugten Daten in einer davon abhängigen Funktion aufzugreifen.

6.1.5 Mikroprogrammierung

Bereits im Jahre 1951 hat Wilkes die Mikroprogrammierung erfunden [280]. So wurde es möglich, Prozessoren auf Basis sehr einfacher Funktionseinheiten zu entwickeln. Die Verbindung der Einheiten zu einem Datenpfad für einen Befehl wurde dann nicht mehr statisch festgelegt, sondern zur Laufzeit anhand eines oder mehrerer Mikrobefehle vorgenommen. Man spricht dann von horizontaler oder vertikaler Mikroprogrammierung. Die zur Umsetzung eines Befehls der Befehlssatzarchitektur[10] verwendete Folge an Befehlen wird als Mikroprogramm bezeichnet.

Mit der Mikroprogrammierung entfällt als eine von mehreren möglichen Fehlerquellen bei der Prozessorentwicklung die Verschaltung der einfachen Funktionseinheiten zu Datenpfaden während der Schaltungsentwicklung. Das Mikroprogramm kann hingegegen in einem Simulator sorgfältig entwickelt, simuliert und validiert werden. Sollten dennoch Fehler in der Implementierung der Mikroprogramme vorhanden sein, so können diese nachträglich behoben werden. Wird die Befehlssatzarchitektur erweitert, so können neue Mikroprogramme dem Prozessor hinzugefügt werden, um die neuen Befehle zu implementieren.

Eine formale Beschreibung einschließlich vieler Beispiele findet sich unter anderem im Textbuch „Mikroarchitekturen und Mikroprogrammierung: Formale Beschreibung und Optimierung“ [35] (A. Bode).

6.2 Konzept

Der Anwender soll die Möglichkeit erhalten, die Komponenten sowohl einzeln als auch gemeinsam zu verwenden. Das Rahmenwerk muss entsprechend eine Verbindungsstruktur bereitstellen, mittels

[9] engl. reentrant = wiedereintrittsfähig
[10] engl. instruction set architecture (ISA)

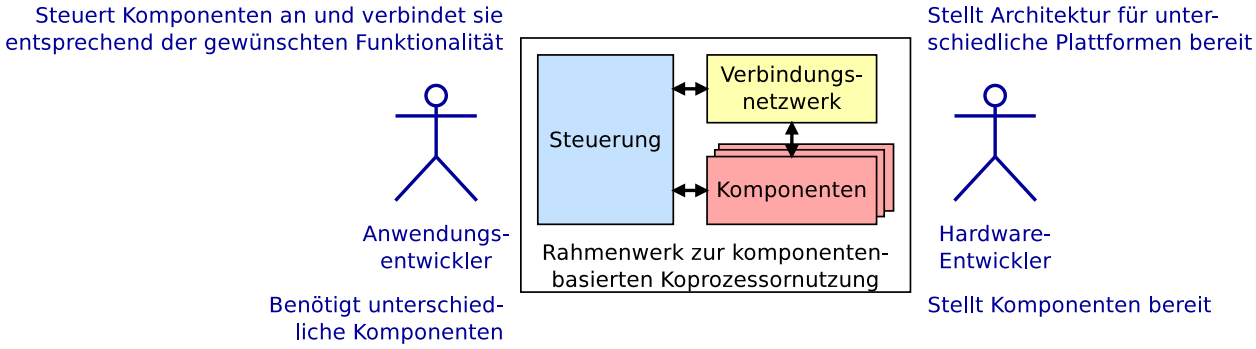


Abbildung 6.1: Hardwareentwickler stellen die Funktionsblöcke und portieren die Umgebung auf die Zielplattform, während Anwender für die Ansteuerung entsprechend dem Ziel der Anwendung sorgen müssen.

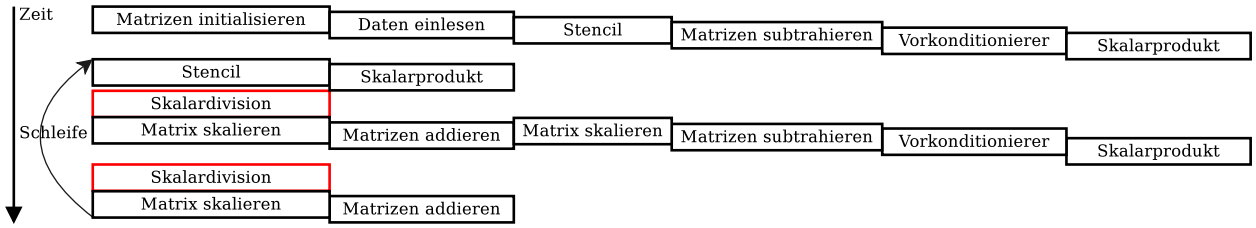


Abbildung 6.2: Komponentenbasierte Ausführung des CG-Verfahrens mit zeitlicher Überlappung unterschiedlicher Komponenten.

derer Daten zwischen den Komponenten ausgetauscht werden können. Zum Aufrufen der Komponenten und Konfigurieren des Datenflusses ist ein geeigneter Mechanismus zu finden und zu implementieren. Die Komponenten sollen von Hardwareexperten bereitgestellt werden, das Rahmenwerk vom Hardwarehersteller oder Hardwareexperten. Die Aufgabe des Anwenders bzw. Anwendungsentwicklers beschränkt sich auf die Ansteuerung der Komponenten. Diese angedachte Kollaboration ist in Abb. 6.1 illustriert. Der Anwendungsentwickler zerlegt die Anwendung in die benötigten Komponenten, so wie in Abb. 6.2 die benötigten Komponenten in graphenähnlicher Darstellung für das CG-Verfahren aufgeführt sind. Der Hardwareexperte implementiert fehlende Komponenten gegebenenfalls und integriert sie im Rahmenwerk. Aus einem ersten, händisch erstellten Kontrolldatenflussgraphen aufbauend auf Abb. 6.2 kann der Anwendungsentwickler die Ansteuerung der Komponenten umsetzen. Idealerweise ermöglichen die interne Steuerung und das Verbindungsnetzwerk die in der Abbildung angedeutete zeitlich überlappte Ausführung der Komponenten angelehnt an das Datenflussprinzip. Die Notwendigkeit, die Daten wiederholt einzulesen und in den Speicher zurückzuschreiben, entfällt, indem berechnete Daten gleich weiterverwendet werden in nachfolgenden Stufen. Die Ausnutzung der verfügbaren Speicherbandbreite kann somit verbessert werden. Dabei ist es sinnvoll, nicht auf die vollständige Bearbeitung der gesamten Eingangsdaten zu warten, sondern die Daten auf einer niedrigeren Granularitätsebene bereits zu verbrauchen, beispielsweise auf Zeilenebene bei Matrizen. Dies stellt einen klaren Gegensatz zum Datenflussprinzip dar, bei welchem auf die vollständige Berechnung eines Operanden wie etwa einer gesamten Matrix gewartet wird oder aber eine gesamte Matrixoperation als Folge von MAC-Operationen zu sehen ist, die einzelne Skalare verbrauchen und produzieren.

6.3 Hardware-MOLEN & Slotextensions

Das Rahmenwerk „Hardware-MOLEN“ [273, KVB+09] (HMOL) zielte zunächst auf die Virtualisierung von FPGAs, damit bis zu sechs Anwendungen zeitgleich den FPGA nutzen können. Der FPGA war dazu in eine Schnittstelle und sechs sogenannte *Slots* aufgeteilt worden, von denen jeder im Sinne einer Komponente unterschiedliche Funktionalität – oder auch die gleiche – im Rahmen der verfügbaren Hardwareressourcen bereitstellen soll. Als Komponenten wurden neben einer

Testeinheit die Implementierung eines exakten Akkumulators [83] und eine 3DES-Implementierung gestellt.

Wie allerdings in Abschnitt 5.3 erkennbar wurde, ist diese Ebene unzureichend zur Anwendungsbeschleunigung. Anstelle von einzelnen Skalarprodukten sollten diese unter massiver Datenwiederverwendung gemeinsam eine Matrix-Vektor-Multiplikation durchführen. Bei HMOL obliegt es dem Anwendungsentwickler, mittels sechs separater Threads aus der Anwendung heraus die in den Slots bereitgestellten Komponenten zu nutzen. Somit kann ein hohes Maß an Kommunikation der Daten und Synchronisation der Threads innerhalb der Anwendung nötig sein. Die Threads erfragen von einer Verwaltungsschnittstelle die geforderte Komponente in Form einer Identifikationsnummer. Ist die Komponente vorhanden und nicht bereits anderweitig vergeben, so werden die benötigten Verwaltungsstrukturen übergeben. Es ist jedoch unklar, in welchem Slot genau die Komponente liegt. Ist die Komponente hingegen nicht vorhanden, so war zwar angedacht, die partielle dynamische Rekonfiguration eines freien Slots über die Verwaltungsschnittstelle anzustoßen. Diese wurde jedoch nicht implementiert. Denn jeder Slot sieht physisch auf dem FPGA anders aus und hält andere Ressourcen bereit, so dass für jeden physischen Slot jede Komponente separat platziert und geroutet werden müsste. Somit hat sich das Problem der Slot-Rekonfiguration sogar dahingehend verschärft, dass die Komponente für genau diesen Slot vorliegen müsste. Dieser Weg benötigt zu hohen Aufwand bei der Entwicklung und Bereitstellung des Rahmenwerks und der Komponenten.

Ein weiteres Problem ist, dass vorab nicht bekannt ist, ob nicht das Beibehalten der aktuellen Slot-Belegungen vorzuziehen ist, da weitere Aufgaben später mehr von der aktuellen Belegung profitieren könnten, als dass die Rekonfiguration eines Slots mit neuer Komponente und daraufhin wieder mit der alten Komponente helfen würde. Dieser Problemstellung widmet sich verstärkt RISPP [23]. Dieses Planungsproblem wird später mit dem Konzept der Attributierung separat unterstützt (Kapitel 7). In weiteren Arbeiten wird das Planungsproblem in heterogenen Systemen konkret angegangen [170].

Für den Austausch von Daten musste eine geschickte Möglichkeit zur Verbindung der Slots untereinander gefunden werden [Ahu09], welche die Kommunikation aller Slots miteinander ermöglicht und gleichzeitig hohe Datenübertragungsraten bietet. In Anlehnung an die IBM Cell B.E. wurde ein ringartiger Bus gewählt, da eine Kreuzschaltung zur Verbindung zu ressourcenaufwendig wäre und bei einem gewöhnlichen Bus der Durchsatz zu gering. Die entwickelte Architektur ist in Abb. 6.3 dargestellt und wird als HMOL+Slotextensions bezeichnet. Mit dieser Architekturerweiterung ist es prinzipiell möglich, über einen Slot eine Zeile einer linken Matrix einzulesen und an mehrere Skalarprodukteinheiten in weiteren Slots weiterzureichen zum Zwecke der Matrixmultiplikation. Diese weiteren Slots lesen die Spalten der rechten Matrix selbständig ein und erhalten die Zeilendaten über die interne Verbindungsschnittstelle. Aufgrund des bei HMOL implementierten Konzepts der freien Belegbarkeit der Slots mit unterschiedlichen Komponenten muss allerdings zum Versenden der Zeilendaten bekannt sein, in welchen Slots genau die zu verwendenden Skalarprodukteinheiten untergebracht sind, damit diese korrekt adressiert werden können. Die Slot-Adressen müssen der Einheit zum Einlesen der Zeilendaten mitgeteilt werden auf geeignete Weise. Diese Einheit wird somit zu einer parametrisierten Steuerung. Über das implementierte Verbindungsnetzwerk können nur wenige Daten ausgetauscht werden, so dass eine komplexere Verschaltung mehrerer unterschiedlicher Komponenten zur Ausnutzung von Task- bzw. Pipeline-Parallelismus nicht mehr möglich ist. Aufgrund der Probleme bei der Nutzung des slot-basierenden Ansatzes muss eine andere Möglichkeit gefunden werden, wie Anwendungsentwickler bequem und effizient, also mit wenig Aufwand zur Programmierung der Kommunikation zwischen den Komponenten und ohne zeitlich hohen Aufwand für die Design-Synthese, mehr Nutzen aus rekonfigurierbarer Hardware durch Verlagerung von mittel- bis grobgranularer Funktionalität ziehen können.

6.4 Mikroprogrammierbares FPGA-Rahmenwerk für eine datengetriebene Architektur

Der Aspekt der Abbildung eines Datenflussgraphen auf den Koprozessor und der Verbindung der Komponenten wurde näher untersucht. Werden Hochsprachen und Quelltextkonverter genutzt, um die Ansteuerung von Komponenten wie z.B. Gleitkommaeinheiten in einer Hardwarebeschreibungsprache zu erzeugen, so werden in der Regel riesige, schwer wartbare und ineffizient in FPGAs

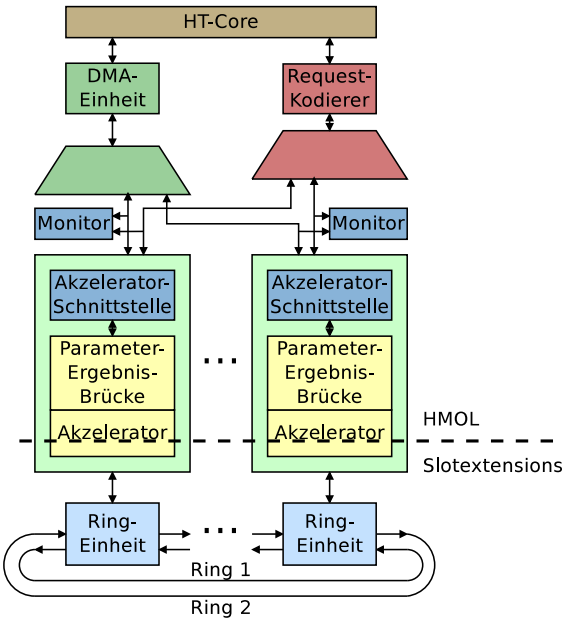


Abbildung 6.3: HMOL+Slotextensions.

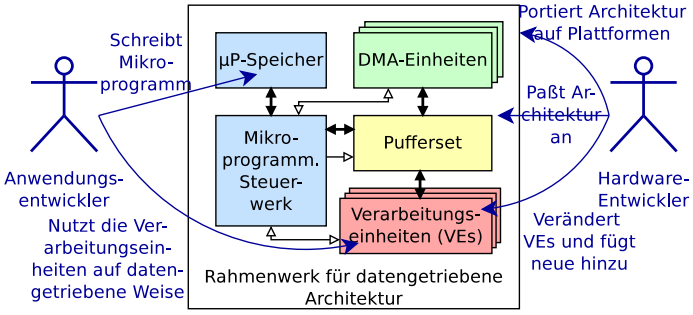


Abbildung 6.4: Datengetriebene, mikroprogrammierbare Umgebung für FPGAs. Hardwareentwickler stellen die Funktionsblöcke und portieren die Umgebung auf die Zielplattform, während Softwareentwickler die Mikroprogramme schreiben und für die Datenbereitstellung sorgen müssen.

umsetzbare[11] Zustandsautomaten generiert. Pipelining ist darüberhinaus häufig nur begrenzt mit solchen Automaten umsetzbar. Gerade auf dem UoH HTX-Board gibt es ferner große Ressourceneinschränkungen, weshalb dieser Ansatz nicht geeignet ist. Ein Problem bei der Verwendung von HMOL+Slotextensions mit domänenspezifischen Komponenten als Alternative zu den hochsprachlichen Ansätzen liegt darin, dass in Abhängigkeit der jeweiligen Belegung der Slots die Slotnummer eines Empfängers sich ändert und dem sendenden Slot erst mitgeteilt werden muss. Ein weiteres Problem besteht in der effektiven Verschaltung der Komponenten miteinander, um die Ausführung eines Datenflussgraphen erreichen und damit den Pipeline-Parallelismus ausnutzen zu können.

Der in dieser Dissertation verfolgte Ansatz zur Lösung der genannten Probleme besteht aus einem zentralen Puffersatz, der einem Kreuzschienenverteiler ähnlich ist, sowie einer mikroprogrammierbaren Steuerung [280]. Die zugehörige Veröffentlichung [NBK+13] dient teilweise als Grundlage dieses Abschnitts.

Wenn auf einem FPGA bekannte Komponenten geladen sind und ein Mikroprogrammsteuerwerk zur Ansteuerung selbiger vorhanden ist, kann ein Anwendungsentwickler schnell und bequem in einem neuen Mikroprogramm andere Algorithmen umsetzen und damit andere Anwendungen unterstützen, ohne in die Hardwareentwicklung oder -einschränkungen involviert zu sein. Der Datenaustausch zwischen den Komponenten soll über Puffer entkoppelt sein, um unnötige Wartezyklen zu vermeiden und unterschiedliche Latenzen von Hostspeicher, Board-Speicher oder FPGA-internem Speicher zu überbrücken, und steuerbar sein, um die Komponenten beliebig und der umzusetzenden, grobgranulareren Funktionalität entsprechend miteinander kommunizieren lassen zu können. Dieser gesamte Ansatz ist in Abb. 6.4 illustriert. Obendrein stellen sich hervorragende Debugging-Möglichkeiten, da die Mikroprogramme schrittweise entwickelt werden können mit Ausgabe der Zwischenwerte sowie mit zunächst nur der Auslagerung einzelner Teile in die Hardware und anschließend der sukzessiven Verlagerung weiterer Anwendungsteile. Dabei ist keine erneute Synthese nötig; stundenlanges Warten entfällt somit im Gegensatz zu der Verwendung von Hochsprachen zur Hardwarebeschreibung.

6.4.1 Mikroprogrammierbares Steuerwerk

Für diese Architektur wurde eine Systemarchitektur mitsamt einer Hardware-Software-Schnittstelle zum Aufrufen von Mikroprogrammen und auch ein zugehöriges mikroprogrammierbares Steuerwerk

[11] Dies ist auf zuviele Zustände und auf die gewählte Vermischung von Zustandsautomaten mit kombinatorischer Logik und zahlreichen Signalen sowie Registern zurückzuführen.

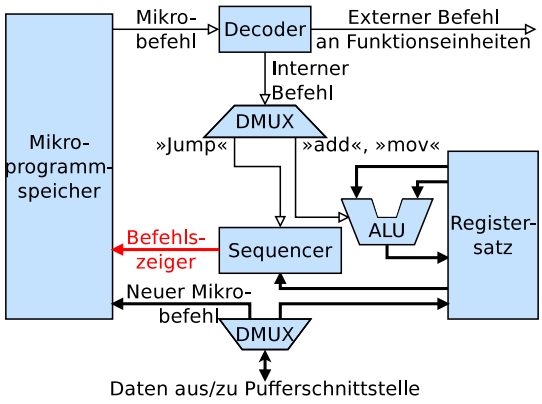


Abbildung 6.5: Mikroprogrammierbares Steuerwerk (μp SW).

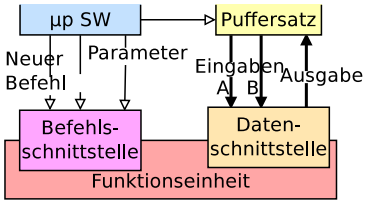


Abbildung 6.6: Befehls- und Datenschnittstelle für Funktionseinheiten.

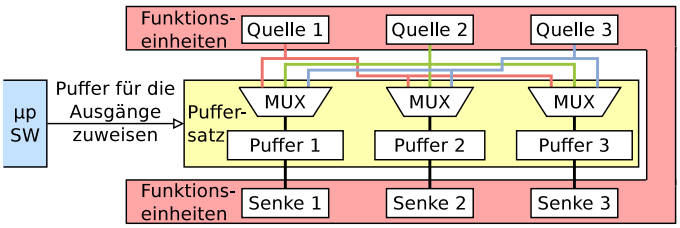


Abbildung 6.7: Puffersatz und Schnittstelle zwischen Funktionseinheiten.

```
regw R10, 200
buf_assign dma1, buf1
buf_assign dma2, buf2
dma1r 0x1add, R10
dma2r 0x2add, R10
buf_assign vadd1, buf3
vadd1 R10
dma3w 0x3add, R10
j 0
```

Abbildung 6.8: Mikroprogramm zum Addieren zweier Vektoren.

(μp SW) entwickelt. Das Steuerwerk ist in Abb. 6.5 dargestellt. Ein vom Anwender initiierter Aufruf eines Mikroprogramms führt dazu, dass der Befehlszeiger im Sequencer auf diejenige Stelle im Mikroprogrammspeicher gesetzt wird, wo das aufgerufene Mikroprogramm steht. Die Mikrobefehle aus dem Speicher werden zunächst als interne oder externe Befehle dekodiert, die externen an die Funktionseinheiten (Umsetzungen von Komponenten) weitergereicht und die internen Befehle zur Ansteuerung der ALU oder des Sequencers verwendet. Der Sequencer unterstützt Sprungbefehle und auch Schleifen. Die ALU führt einfache Operationen auf dem Registersatz aus, die vor allem der Parametrierung der Einheiten und Schleifensteuerung dienen.

Funktionseinheiten erhalten zwei wesentliche Schnittstellen, wie sie in Abb. 6.6 zu sehen sind. Die linke ist zuständig zur Entgegennahme von dekodierten Befehlen und Parametern, die auch aus dem Registersatz stammen können, die rechte interagiert mit dem Puffersatz und stellt die Operanden für die Operation bereit. Ausgehend von beispielsweise einem Vektoraddierer als Funktionseinheit ist der Befehl die Angabe von Addition oder Subtraktion, und der Parameter dient dazu, die Länge der Vektoren anzugeben.

6.4.2 Zentraler Puffersatz

Der Datenfluss zwischen Quell- und Zieleinheiten über den Puffersatz ist in Abb. 6.7 dargestellt. Es gibt allgemein drei unterschiedliche Möglichkeiten, wie die Funktionseinheiten über den Puffersatz miteinander verbunden werden könnten. Die erste ist, bei jeder Einheit sowohl die Eingangsports als auch die Ausgangsports zur Laufzeit genau zu spezifizieren. Nachteilig daran ist, dass eine Vielzahl an Multiplexern und Demultiplexern dazu nötig ist, denn Multiplexer und Demultiplexer für mehr als vier Möglichkeiten sind mit den verwendeten FPGAs sehr aufwendig. Daher sollte ihre Anzahl gering gehalten werden. Somit bleibt als weitere Möglichkeit, jede Einheit fest mit einem eigenen Ausgangspuffer zu verbinden. Die Eingänge müssten dann je über Demultiplexer aus den verschiedenen Puffern auswählen. Problematisch daran sind zweierlei Dinge: Einerseits könnten die Daten aus einem Puffer von nur genau einer Einheit verbraucht werden; zur Kommunikation zwischen einem Erzeuger und mehreren Verbrauchern würden zusätzlich Duplikationseinheiten benötigt. Andererseits haben Einheiten tendenziell eher mehr Eingänge als Ausgänge. Daher sind

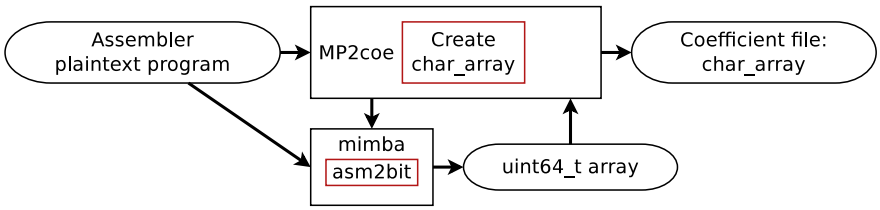


Abbildung 6.9: Werkzeugkette zum Übersetzen von Mikroprogrammen aus Assemblertexten in maschinenlesbaren Code.

als dritte und letzte Möglichkeit die Eingänge fest zu verdrahten und die Ausgänge flexibel zu halten, wodurch insgesamt weit weniger Multiplexer als Demultiplexer bei der vorher genannten Möglichkeit benötigt werden. Diese Möglichkeit wird aus den genannten Gründen gewählt.

Die Puffer sollen vor allem dem Streaming von Daten dienen, also Vektoren von großen Matrizen bei numerischen Anwendungen, kontinuierliche Datenströme in der Signalverarbeitung oder ganze Sequenzen bei bioinformatischen Anwendungen. Um die verfügbaren Speicherressourcen nicht zu stark auszulasten und dadurch das Routing unnötig zu erschweren, liegt eine sinnvolle Puffergröße bei etwa 16.384 Bytes für die Verwendung der Xilinx XC5VLX330-FPGAs auf der Convey HC-1. Dies ist mit lediglich acht BlockRAMs implementierbar und sollte die Unterbringung von etwa zehn bis zwanzig Puffern ermöglichen. Manche Einheiten benötigen an einem Eingang jedoch nur wenige Daten als Parameter, etwa bei einer Vektorskalierung. Daher lohnt es nicht, ausschließlich Puffer für Vektoren bereitzustellen, sondern insbesondere bei der gewählten dritten Möglichkeit mit fest verdrahteten Eingangspuffern lohnt es sich, diese Puffer klein zu wählen.

Über den Mikrobefehl `buf_assign` werden der Puffersatz und die darin enthaltenen Multiplexer entsprechend dem Bedarf so konfiguriert, dass die Ausgaben einer Einheit zu den davon abhängigen Verbrauchern gelangen. Indem ein Ausgangsdatenstrom auf mehrere Eingangspuffer verteilt wird, können mehrere Verbraucher zeitgleich mit unterschiedlicher Geschwindigkeit die jeweils eingehenden Daten verbrauchen und so der entstehende Pipeline-Parallelismus mehrfach ausgenutzt werden. Externe Daten werden über DMA-Einheiten eingelesen (Befehl `dmar`) und genauso in den externen Speicher zurückgeschrieben (Befehl `dmaw`). Abbildung 6.8 gibt ein Beispiel dafür, wie zwei DMA-Einheiten unterschiedliche Zielpuffer zugewiesen werden, die jeweils fest mit der Vektoradditionseinheit verbunden sind, und dann das Einlesen von Daten ab Adressen `0x1add` bzw. `0x2add` angestoßen wird. Ferner wird die Ausgabe der Vektoraddition mit dem der dritten DMA-Einheit zugehörigen Eingangspuffer verbunden, bevor die Addition für `R10=200` Elemente aufgerufen wird. Der Ergebnisvektor wird an Speicheradresse `0x3add` geschrieben. Die drei DMA-Einheiten und Vektoradditionseinheit führen zeitgleich zueinander ihre Aufgaben aus, so dass das erste Ergebnisdatum bereits im Speicher steht, während noch die letzten Daten eingelesen werden. Um dies zu erreichen, werden die Mikrobefehle asynchron aufgerufen, d.h. es wird nicht auf die Vollendung des letzten Befehls gewartet, sondern lediglich darauf, dass dieser der entsprechenden Einheit zugewiesen werden konnte (Auflösung von Ressourcenkonflikten).

6.4.3 Werkzeugkette

Eine einfache Werkzeugkette (Abb. 6.9) sorgt dafür, den menschengeschriebenen Assembler in maschinenlesbaren Binärcode zu übersetzen. Darüberhinaus kann sie Speicherkonfigurationsdateien für die statische Speicherbelegung zur Synthesezeit erzeugen, die dafür benötigt wird, dass das Steuerwerk sich in einem Wartezustand für einen Befehlsaufruf oder für die Übertragung eines neuen Mikroprogramms befindet. Ein Treibermechanismus, der sowohl in die Systemumgebung als auch in die Anwendung selbst integriert sein kann, wird benötigt, um die neuen Mikroprogramme vom Host aus zu übertragen.

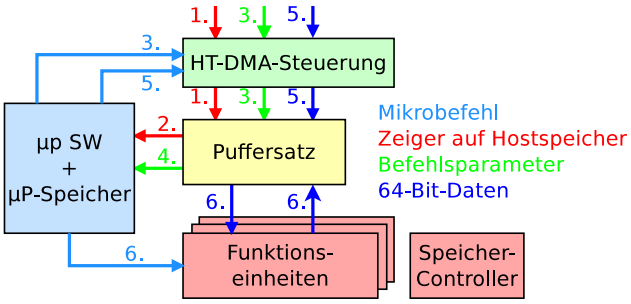


Abbildung 6.10: Implementierung des Rahmenwerks auf dem UoH HTX-Board.

6.4.4 Hardware-Software-Schnittstelle

Über die Hardware-Software-Schnittstelle wird ein Koprozessor gesteuert, werden Daten ausgetauscht und die CPU darüber informiert, dass ein Koprozessoraufruf vollendet wurde.

Die Verwendung eines Koprozessors über das mikroprogrammierbare, datengetriebene Rahmenwerk soll wie folgt ablaufen: Zunächst muss das auszuführende Mikroprogramm übertragen werden. Dazu wird diejenige Speicheradresse übermittelt, an welcher das mithilfe der Werkzeugkette übersetzte Mikroprogramm abgelegt wurde. Von dieser Adresse holt das Steuerwerk selbständig das Mikroprogramm, unter Umständen unter Zuhilfenahme eines weiteren Mikroprogramms, welches die DMA-Einheiten konfiguriert und in den Mikroprogrammspeicher schreiben lässt. Nach erfolgreicher Übertragung des Mikroprogramms kann selbiges aufgerufen werden und, wie bereits anhand des Beispiels einer Vektoraddition geschildert, die Abarbeitung durchgeführt werden, wozu Daten geladen und geschrieben sowie die Funktionseinheiten mit Befehlen versorgt werden.

Umsetzung auf dem UoH HTX-Board

Auf dem HTX-Board (vgl. Abschnitt 2.1.4) steht mit dem freien HT-Core keine Speicherkohärenz hinsichtlich des Hauptspeichers zur Verfügung. Um mit dem Koprozessor zu kommunizieren, wird daher selbiger über Speicheradressierung im virtuellen Speicher zugänglich gemacht. Die dorthin geschriebenen Daten werden dadurch unmittelbar an das HTX-Board gesendet. Neben dem Schreiben von Parametern, Mikroprogrammadressen im Hostspeicher und Befehlen ist auch eine Zustandsabfrage und in geringem Maße sogar Debugging möglich. Um wie in Abb. 6.10 die Ausführung zu starten, wird nach der Übergabe der Mikroprogrammadresse (1. und 2.) das neue Mikroprogramm mittels des Standardmikroprogramms geholt (3.) und im Mikroprogrammspeicher abgelegt (4.). Das neue Mikroprogramm beginnt mit dem Lesen der Eingabedaten (5.) und führt dann die Befehle auf den Funktionseinheiten aus (6.). Das Rahmenwerk benachrichtigt das Anwenderprogramm auf dem Host mittels eines Interrupts über die vollendete Ausführung. Die erzeugten Daten müssen vollständig ausgelesen (verbraucht) werden, bevor der Koprozessor wieder verwendet kann. Ist keine erneute Ausführung gewünscht, so muss der zur Ansteuerung allozierte Speicher freigegeben werden und die Verbindung mit dem Gerät und dessen Treiber geschlossen werden.

Das Rahmenwerk ist insbesondere dazu gedacht, die Datenwiederverwendung für mehrere exakte Skalarprodukteinheiten bei der exakten Matrixmultiplikation zu erreichen. Die benötigten Hardwareressourcen nach *Place & Route* mit lediglich drei einfachen Funktionseinheiten zur Inkrementierung von Integer-Werten um den Wert 1 sind in Tabelle 6.1 aufgeführt. Es war nicht möglich, die Zeitvorgaben von 5 ns / 200 MHz zu erreichen aufgrund der hohen Anforderungen der Steuereinheit und des Puffersatzes. Daher musste auf die vollständige Integration und Synthese/Place&Route der exakten mikroprogrammierbaren Skalarprodukteinheit für das HTX-Board verzichtet werden. Da die Takt-Ungenauigkeit mit 0,11 ns allerdings höher ausfällt als die Überschreitung mit $5,0\,\text{ns} - 5,086\,\text{ns} = -0,086\,\text{ns}$, konnte man mit dem generierte Bitstream unter Verwendung eines zusätzlichen auf dem FPGA montierten Lüfters erste Messungen durchführen.

Die durchgeführten Leistungsmessungen zeigen jedoch in Abb. 6.11, dass nur etwa 500 MB/s der maximal 1600 MB/s an möglicher Bandbreite jeweils pro Richtung erreicht werden. Der Nutzen muss also durch sehr hohe Wiederverwendungsraten der eingelesenen und berechneten Daten durch

Ressource	Anzahl	Nutzungsgrad
Slices	11531	27 %
Flip-Flops	8856	10 %
LUTs	16005	18 %
BRAMs	49	13 %
Maximale Frequenz nach		
Synthese	4,426 ns / 225,958 MHz	
Place & Route	5,086 ns / 196,618 MHz	

Tabelle 6.1: Ressourcenverwendung des Rahmenwerks auf dem UoH HTX-Board nach Place & Route beim Xilinx Virtex-4 FX100.

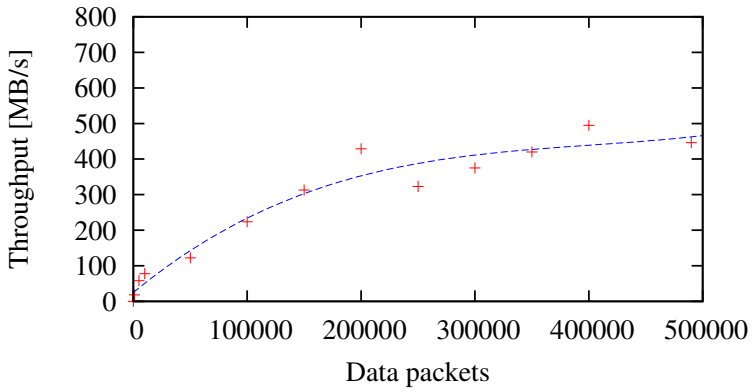


Abbildung 6.11: Durchsatz des Rahmenwerks auf dem UoH HTX-Board an 64-Bit-Wörtern mit überlappten Lese- und Schreibzugriffen von/zum Hostspeicher.

Ausnutzung von Task- und insbesondere Pipeline-Parallelismus erzielt werden, um die geringe Ausnutzung der Bandbreite zu kompensieren. Die Hinzunahme weiterer Funktionseinheiten war jedoch nicht mehr möglich, solange nicht wesentlich geringere Taktfrequenzen verwendet werden sollen. Dadurch würde jedoch auch die ausnutzbare Bandbreite weiter sinken. Demzufolge wurde auf Verbesserungen der Implementierung verzichtet und anstelle des UoH HTX-Boards mit der Convey HC-1 eine andere Plattform betrachtet, wie im Folgenden beschrieben.

Umsetzung auf der Convey HC-1

Mit einem leistungsfähigen FPGA-System wie der Convey HC-1 (vgl. Abschnitt 2.1.5 können nun Vektor- und Matrixoperationen angegangen werden. Entsprechend zielt die Umsetzung der datengetriebenen, mikroprogrammierbaren Steuerung auf der Convey HC-1 auch auf matrix-basierende Anwendungen ab. Dadurch erhalten auch Mathematiker und Numeriker, die tendenziell eher Anwender abstrakter Programmieransätze zur Nutzung rekonfigurierbarer Hardware sind, bequem Zugang zu vorgefertigten, leistungsstarken Hardwareimplementierungen. Sehr häufig werden Vektoren addiert und skaliert (vgl. axpy-Operation). Weiter wichtig sind Skalarprodukte, denn sie werden auch für Matrix-Vektor-Multiplikationen und Matrixmultiplikationen benötigt. Viele Gleichungslöser basieren auf Stencils, daher ist eine effiziente, datengetriebene Einheit zur Anwendung von Stencil-Operationen auf Matrizen implementiert. Sie basiert auf den in den Abschnitten 5.8.3 und 5.8.4 geschilderten Überlegungen und Erkenntissen. Auf dieser Basis konnte auch ein Löser für Rot-Schwarz-Stencil-Schemata als Pipeline implementiert werden. Zusätzlich sind Skalardivisionen und Vergleichseinheiten nötig, um beispielsweise das CG-Verfahren unterstützen zu können. Abgesehen von den arithmetischen Einheiten müssen die Daten natürlich auch gelesen und geschrieben werden können, was wieder mittels DMA-Einheiten erfolgt. Die Implementierung dieses Rahmenwerks mit jeweils einer der oben genannten Einheiten und drei DMA-Einheiten ist in Abb. 6.12 veranschaulicht. Mit dieser Auswahl an domänenspezifischen Komponenten und der Möglichkeit zur bequemen Ansteuerung über den Standard-Koprozessoraufruf auf der Convey HC-1 sollten sich die benötigte Datenwiederverwendung und auch der Pipeline-Parallelismus ausnutzen lassen.

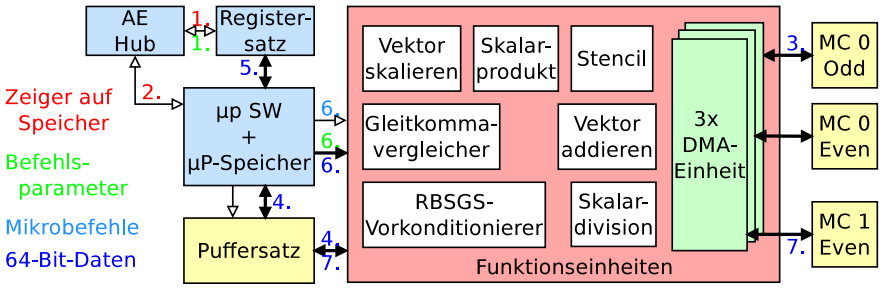


Abbildung 6.12: Implementierung des Rahmenwerks auf der Convey HC-1.

Ressource	Anzahl	Nutzungsgrad
Slices	34.520	66 %
LUTs	89.447	43 %
Register	86.379	41 %
DSPs	110	57 %
BRAMs	166	57 %
Max. Freq.	6,643 ns / 150,5 MHz	

Tabelle 6.2: Ressourcenbedarf des Rahmenwerks mit je einem einzelnen Exemplar einer Komponente auf der Convey HC-1 mit einem Xilinx Virtex-5 LX330.

Auf der Convey HC-1 wird der allgemeine Registersatz des AE Hubs genutzt, um die Adresse des Mikroprogramms und der Speicherorte der Ein- und Ausgaben zu übergeben (1.). Wenn der Koprozessor und damit auch das Rahmenwerk über den AE Hub aufgerufen wird (2.), wird mittels des Standardmikroprogramms zunächst das neue Mikroprogramm aus dem externen Speicher geladen (3.) und über den Puffersatz in den Mikroprogrammspeicher geschrieben (4.). Dann wird zum Mikroprogramm gesprungen und die Befehle aus dem Mikroprogramm werden dekodiert und die Register gelesen (5.), um schließlich die entsprechenden Funktionseinheiten mit den passenden Parametern anzusprechen (6.). Die Funktionseinheiten werden gleichzeitig zueinander auf datengetriebene Weise ausgeführt (7.).

Die Implementierungsergebnisse mit jeweils nur einem Exemplar jeder der o.g. Einheiten finden sich in Tabelle 6.2. Damit müssen Anwendungen wie das CG-Verfahren, die manche Operationen pro Iteration mehrfach verwenden, aufgeteilt werden in mehrere sequentielle Aufrufe („Blockvariante“). Da die Daten dabei aber nicht vom Host zum Koprozessor oder umgekehrt migriert werden müssen, sollte dies keinen nennenswerten Nachteil bei großen Datenmengen darstellen.

6.4.5 Ergebnis und Zusammenfassung des mikroprogrammierbaren, datengetriebenen Rahmenwerks

Es gibt bereits mehrerlei Ansätze, wie wissenschaftliche Anwendungen, die stark auf Gleitkommaberechnungen basieren, in Hardware gebracht werden können. Ein solcher Ansatz sind Coarse-Grained Reconfigurable Arrays (CGRAs). Diese sind auch in rekonfigurierbare Logik übertragbar und darüber das Konzept der Komponenten ausnutzbar. Wesentlich ist dabei die überwiegend statische Verdrahtung der Zellen entsprechend der Anwendungslogik. Mikroprogramme können ein adäquates Mittel dazu darstellen, FPGAs wie CGRAs zu verwenden und durch die nahtlose Kopplung der Funktionseinheiten, welche die Funktionalität einer Komponente umsetzen, die Datenwiederverwendung wesentlich zu erhöhen. Die Kernkomponente dabei ist ein Puffersatz, dessen Ressourcen den Funktionseinheiten zuweisbar sind, um so den Datenfluss zu modellieren.

Das implementierte und getestete Rahmenwerk ermöglicht nun Folgendes: den Nutzen, die Ausführung einer Komponente in rekonfigurierbare Hardware zu verlagern, zu bestimmen; unterschiedliche Komponenten in Hardware separat voneinander zu evaluieren hinsichtlich größtmöglichen Nutzens; mehrere Komponenten aus der verfügbaren Menge zeitlich versetzt zueinander nutzen; und schließlich die zeitgleiche, task-parallele Nutzung mehrerer Komponenten sowie obendrein die überlappte, pipeline-parallele Nutzung von Komponenten, die im Kontrolldatenflussgraphen

voneinander abhängig sind. In ein solches Rahmenwerk kann exakte Arithmetik eingebracht werden, von welcher der Anwender entweder gar nichts weiß und daher automatisch genauere Ergebnisse erhält, oder diese bewusst einsetzt, um eine Umgestaltung des implementierten Verfahrens zu vermeiden.

Das Erzeugen von rekonfigurierbaren Hardwaredesigns ist seitens des Anwenders nicht mehr nötig. Dadurch entfällt die zeitaufwendige Entwurfsraumexploration bei der Umsetzung des Algorithmus in einer Hochsprache und zusätzlich die für Synthese und Platzierung&Verdrahtung benötigte Zeit. Als Alterative zur Quelltexterzeugung aus hochsprachlichem Code sollte jedoch die (semi-)automatische Erzeugung von Mikroprogrammen für solche Steuerwerke untersucht werden.

6.5 Ergebnis und Zusammenfassung

Der Schlüssel zur Anwendungsbeschleunigung mithilfe von FPGAs liegt einerseits in der Auslastung, aber nicht Überlastung, der verfügbaren Speicher- und Schnittstellenbandbreite. Also müssen Daten so oft und viel wie möglich wiederverwendet werden, bevor sie verdrängt oder zurückgeschrieben werden. Entsprechend muss Task-Parallelismus genutzt werden, was über das Datenflussprinzip erreicht werden kann. Ein statisches Rahmenwerk wie H-MOL, das zunächst wie ein MIMD-artiger Koprozessor gedacht war, kann dies zwar durch Hinzufügen eines Verbindungs- und Konfigurationssystems leisten, ist aber nicht vom Nutzer programmierbar. Die Reihenfolge des Aufrufs der Komponenten und die Verbindung selbiger miteinander müssen vom Nutzer konfigurierbar sein. Dazu eignen sich Mikroprogramme, die von Software-Seite aus übertragen und dann auf dem Koprozessor gestartet werden. Die einzelnen Funktionsaufrufe an die Komponenten werden innerhalb weniger Takte vom Steuerwerk durchgeführt, insofern die Komponente verfügbar und ausführungsbereit ist. Die entsprechende Ausführung der Komponenten dauert aber idealerweise einige tausend Takte, während weitere Komponenten aufgerufen und ausgeführt werden. Das Mikroprogramm spiegelt insofern eher die Konfiguration des Rahmenwerks wieder, und damit nicht zuletzt auch die Abbildung des Datenflussgraphens auf den FPGA. Datenflussgraphen sind es wiederum, welche für Anwendungsentwickler intuitiv nutzbar sind, um ihre Anwendungen zu beschreiben.

Weitergehend ist zu untersuchen, wie die Entwicklung eines Anwendungsbeschleunigers unter Zuhilfenahme des Konzepts des datengetriebenen, mikroprogrammierbaren Rahmenwerks erfolgt und weiter vereinfacht werden kann. Außerdem ist eine Möglichkeit zu finden, wie die Ausführung von Anwendungsbeschleunigern optimal in Mehrkernsysteme integriert werden kann, da ein Prozessorkern meistens mit der Kommunikation mit dem Beschleuniger beschäftigt ist, die übrigen Kerne aber sinnvolle Arbeit leisten sollten.

Kapitel 7

Attributierung von Eigenschaften und Anforderungen

Bis jetzt wurden eine Reihe von entwickelten Spezialschaltungen für bioinformatische und numerische Anwendungen sowie einige Möglichkeiten und Implementierungen zur abstrakteren Ansteuerung von rekonfigurierbaren Systemen beschrieben. Die FPGA-Nutzung bei HHblits lohnt sich nur für Sequenzen ab 500 Zeichen Länge; das exakte Skalarprodukt in Hardware lohnt nicht für numerisch stabile Verfahren; der mit einer Hochsprache entwickelte Vorkonditionierer ist äußerst langsam, kann aber nebenläufig zu Berechnungen auf dem Hostprozessor eingesetzt werden; und bei Verwendung des mikroprogrammierbaren Steuerwerks hängt die erzielbare Leistung vom implementierten Mikroprogramm und den Eingabedaten ab. Somit hat jede Schaltung besondere Eigenschaften und kann dazu passende Anforderungen der Anwender erfüllen. Damit Systeme mit solchen Spezialschaltungen für den Anwender oder Entwickler attraktiv, bequem programmierbar, mit wenig Aufwand nutzbar werden und Anwendungsbeschleunigung bieten, muss ein geeignetes Mittel bereitgestellt werden. Als Mittel der Wahl wird Attributierung vorgeschlagen, mittels dessen ein Laufzeitsystem den Ausführungsort für eine Komponente bestimmen kann. Das Laufzeitsystem, welches einen Funktionsaufruf auf eine alternative Implementierung umlenken kann, wurde von Mario Kicherer direkt am Lehrstuhl entwickelt [44, 168, 169]. Um die Komponente an dem gewünschten oder hinsichtlich einer Metrik besten Ort auszuführen, sucht es die für den jeweiligen Ort passende Implementierung aus, wenn der Anwender seine Anforderungen bekanntgibt. Dieses Kapitel stellt das Konzept der Attributierung vor [NBK09], die Umsetzung der Werkzeugkette und den nötigen Aufwand zum Auslesen der Attribute [NKBK10]. Der Nutzen in heterogenen Systemen hinsichtlich der anwendungsangepassten Auswahl der passendsten Implementierung [KNBK12] wird mittels des von Mario Kicherer entwickelten Laufzeitsystems evaluiert. Dieses Kapitel basiert entsprechend auf den zugehörigen Veröffentlichungen.

7.1 Idee und Konzept

In rekonfigurierbaren Systemen ist die Hardware zu unterschiedlichen Zeitpunkten unterschiedlich konfiguriert. Für eine Anwendung bedeutet dies einerseits, dass die benötigte Hardware-Konfiguration noch gar nicht geladen sein kann und daher zu überlegen ist, ob es sich zeitlich mitsamt der Rekonfigurationsdauer überhaupt lohnt, die rekonfigurierbare Hardware zu nutzen. Andererseits stehen eventuell mehrere unterschiedliche, für verschiedene Teile der Anwendung geeignete, Konfigurationen mit verschiedenen Eigenschaften zur Auswahl, mit denen die rekonfigurierbare Hardware konfiguriert werden kann. Auch der Gesichtspunkt des Energiebedarfs einer Hardwareschaltung zur Ausführungszeit sowie für die Rekonfiguration gewinnt zunehmend an Bedeutung. Im größeren Kontext sind daher die Untersuchungen geschildert, welche Mittel dafür geeignet sind, ein Laufzeitsystem bei der Entscheidung zu unterstützen, so dass eine Anwendung maximal vom vorhandenen heterogenen System profitiert.

Als Lösungsansatz werden Attribute vorgeschlagen. Die Attribute sollen die Anforderungen von Anwendungen gleichermaßen wie den Systemzustand angeben und damit bei der Entscheidungsfindung anhand von Schwellwerten und Regeln helfen. Sie sollen im Quelltext direkt angegeben werden können und in den ausführbaren Dateien bzw. Bibliotheken gespeichert werden. Das Laufzeitsystem kann dann in Kenntnis der als Attribute gespeicherten Anforderungen wie in Abbildung 7.1

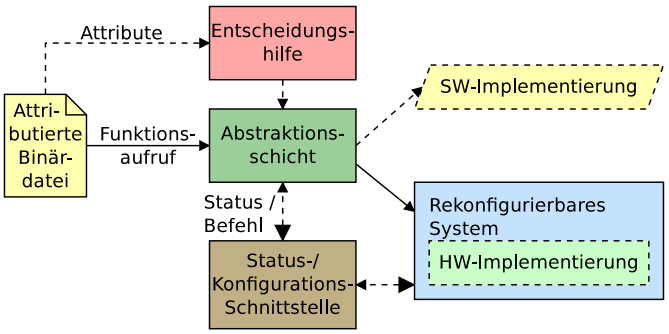


Abbildung 7.1: Attributierte Anwendungsausführung.

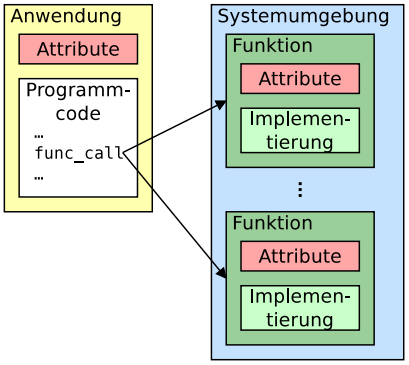


Abbildung 7.2: Zu einem Funktionsaufruf passende Implementierungen.

Name	Beschreibung	Beispiel
`precision`	Zahlenformat	single, double, quad, GMP, exact
`signature`	Funktionssignatur	`int f(char *)`
`psize`	Problemgröße	100, 5000+
`target`	Typ der Verarbeitungseinheit	FPGA, GPGPU, CMP, SMP
`pmodel`	Programmiermodell	CUDA, OpenMP, H-MOL
`cpu_features`	Anforderungen an Hardware	MMX, SSE*, AVX
`speed`	Grobe Leistungsangabe	fast, slow

Tabelle 7.1: Auswahl an Attributen.

dargestellt geeignete Hardwarekonfigurationen suchen, diese im slot-basierten oder mikroprogrammierbaren Rahmenwerk [KVB+09] bereits laden und später die Funktion zur Ansteuerung der rekonfigurierbaren Hardware beim attributierten Funktionsaufruf aus der Anwendung heraus verwenden oder eine Softwareimplementierung der benötigten Funktionalität (vgl. Abb. 7.2). Dazu ist es nötig, dass die verfügbaren Implementierungen gleichermaßen attributiert sind wie auch die Aufrufe bzw. die Anwendung, um einerseits Komponenten mit passender Funktionalität zu finden und andererseits auch die Anforderungen mit diesen Eigenschaften vorhandener Komponenten abgleichen zu können.

7.1.1 Attribute

In Tabelle 7.1 ist eine Auswahl von sinnvoll scheinenden Attributen angegeben, die gewissermaßen als Standardattributsatz verwendet werden kann. Manche Algorithmen wie beispielsweise das Lanczos-Verfahren (vgl. Abschnitt 5.1.2) benötigen hohe Auflösung bei den Gleitkommaoperationen (und ggf. auch -operanden), um bei der Berechnung im Computer numerisch stabil zu sein (s. Abschnitt 5.1.3). Dazu wird das Attribut `precision` eingeführt. Das Problem, dass die beim Funktionsaufruf übergebenen Daten breiter oder schmaler sein können und dazu einen anderen Datentyp brauchen, lässt sich einerseits über Zeiger auf die Speicherstellen lösen, und andererseits über das Attribut `signature`.

Der Nutzen von Beschleunigern ist häufig abhängig von der Problemgröße (z.B. das CG-Verfahren auf der Convey HC-1 unter Verwendung der Vector-Personality; Abb. 5.43), was häufig an der Efffizienz des Datentransfers zum Beschleuniger liegt sowie an der koprozessorseitigen Wiederverwendung der Daten. Daher kann über das Attribut `psize` eine untere Schranke angegeben werden, ab welcher die Nutzung sinnvoll scheint. Auch für die Berechnung eines lückenlosen Scores gibt es eine untere Schranke von 500 Zeichen pro Sequenz, ab welcher sich erst die Nutzung des Koprozessors lohnt [NBSK13] (Abbildung 4.10).

Implementierungen von Komponenten geben anhand des Attributs `cpu_features` die Nutzung spezifischer Hardware an. Zur Vereinfachung der Auswahl zur Laufzeit können Implementierungen für unpassende Hardware/Zielarchitekturen zu Beginn des Auswahlprozesses aussortiert werden.

Hingegen werden Wünsche nach der Zielarchitektur über das Attribut `target` oder über das Programmiermodell (`pmodel`) angegeben, etwa weil sich der Anwender besondere Qualitäten von der Ausführung auf bestimmter Hardware erhofft.

Manchmal sind vorab der Ausführung schon Leistungseigenschaften bekannt, z.B. mag die mitgelieferte Standardimplementierung bekanntermaßen sehr langsam sein, wozu das Attribut `speed` passenderweise auf `slow` gesetzt wird. Umgekehrt könnte ein Hardwareentwickler die von ihm bereitgestellte Implementierung bereits vorab evaluiert haben und weiß daher, dass sie besonders schnell ist, eventuell unter Vernachlässigung anderer Eigenschaften.

Manche Attribute können auch automatisch hergeleitet werden wie etwa die Funktionsnamen oder die Signatur; auch erste Leistungsdaten könnten bereits bei der Kompilierung automatisiert ermittelt werden. An sich soll es dem Anwender und gleichermaßen dem Hardwareentwickler überlassen sein, geeignete Attribute anzugeben. Dabei ist letzlich zu beachten, dass eine zu große Menge von Attributen dazu führt, dass der Auswahlprozess aufwendiger wird.

7.1.2 Code-Erweiterung mit Pragmas

Sowohl die Komponentenimplementierungen als auch die -aufrufe sollen mit Pragmas für das sogenannte *Dynamic Linking and Runtime System* (DLRS) erweitert werden. Die Pragmas beginnen mit `#pragma DLRS` und enthalten beliebig viele Paare aus Schlüssel und Wert: `key=value`. Idealerweise sind die Attribute orthogonal zueinander, um die Parameterexploration zur Laufzeit einfach zu gestalten. Zum besseren Verständnis sind die Implementierungen für 3DES im H-MOL-System [273] und die Aufrufe aus der Methode *main* heraus in Listing 7.1 aufgeführt. Bei eigenmächtig festgelegten Attributen muss auf numerisches Werte zurückgegriffen werden sowie der Minimal- und Maximalwert angegeben werden, damit das DLRS ohne händische Erweiterungen die Attribute auswerten und somit auch berücksichtigen kann.

```
#pragma DLRS name=calculate_3DES psize=10KB features=FPGA
double * calculate_3DES_hw (double* x) {
  return hmol_accelerate_direct (accel_slot, ...);
}

#pragma DLRS name=calculate_3DES psize=50KB features=CPU
double * calculate_3DES_sw (double* x) {
  double *encrypted;
  ...
  return encrypted;
}

int main (int argc, char **argv) {
  double *input, *encrypted;
  input = ...
  #pragma DLRS prefer_hardware=1 psize=30KB
  encrypted = calculate_3DES (input);
  ...
}
```

Listing 7.1: Attributierter Funktionsaufruf und attributierte Implementierungen.

Wie man leicht erkennen kann, ist es wichtig, mehrere Attribute oder interne Vorgaben zu haben, um die Entscheidung in solchen Situationen zu erleichtern, in denen mehrere Implementierungen passend scheinen. Beispielsweise kann das die Angabe der Leistungsaufnahme sein oder die standardmäßige Bevorzugung von FPGAs bzw. Koprozessoren (`prefer_hardware`).

7.1.3 Erweiterung der Standardwerkzeugkette und der Binärdateien

Die gewöhnliche Werkzeugkette bestehend aus Präprozessor, Compiler und Linker soll beibehalten und nur geringfügig erweitert werden wie in Abb. 7.3, so dass die vorgeschlagenen Erweiterungen

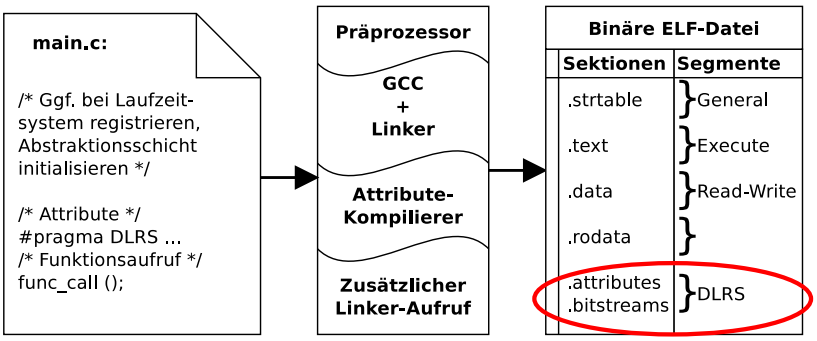


Abbildung 7.3: Übersetzen eines attributieren Programmcodes zu attributierter Binärdatei.

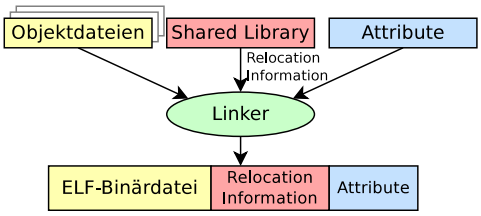


Abbildung 7.4: Speichern der Objekte, Relocation-Information und des DLRS-Segments in der ELF-Datei.

eines Tages in die Standardwerkzeugkette vollständig einfließen könnten. Die Attribute werden dazu im Quelltext als Pragmas angegeben und können daher vom Präprozessor extrahiert werden. Der Compiler und Linker werden nicht berührt. Abschließend müssen die extrahierten Pragma-Daten und ggf. weitere Informationen wie Bitstreams der Binärdatei hinzugefügt werden, so dass sie vom Laufzeitsystem oder den Anwendungen selbst bequem gelesen werden können. Dazu werden die Attribute von einem weiteren Werkzeug kompiliert und zusätzlich in die entstehende Binärdatei gelinkt.

Mittels eines weiteren Segments in der Binärdatei der Anwendung oder Bibliothek lässt sich die binäre Einbindung der Attributdaten elegant erreichen. In modernen UNIX-basierten Betriebssystemen werden solche Dateien im *Executable and Linkable Format* [253] (ELF) gespeichert. Selbiges enthält Kopfdaten mit der Angabe der Anzahl an Segmenten und ihren Positionen in der Datei; die Segmente wiederum bestehen aus ein oder mehreren Sektionen, in denen schließlich Daten wie Konstanten und der übersetzte Programmcode zu finden sind. Um in den ELF-Dateien die Attribute zu speichern, erstellt ein zusätzliches Werkzeug eine reguläre Objektdatei, welche die aus den Pragmas gewonnenen Informationen erhält. Es ist nicht sinnvoll, diese Aufgabe direkt in den Compiler zu implementieren, um auch weiterhin die saubere Trennung der Standardwerkzeugkette in ihre Bestandteile zu wahren. Ein Linker fügt diese Objektdatei als DLRS-Segment hinzu, genau so wie auch andere Objektdateien und die Relocation-Informationen der dynamisch einzubindenden Bibliotheken hinzugelinkt werden, wie es in Abb. 7.4 dargestellt ist. Die DLRS-Sektionen lassen sich auch bestehenden Bibliotheken hinzufügen, wenn dazu Bedarf sein sollte.

7.2 Implementierung

Nach dem im vorigen Abschnitt 7.1 vorgestellten Konzept und einer skizzierten möglichen Umsetzung wird mit der Beschreibung der konkreten Implementierung [Che09] der Attribute, Pragmas, der Erweiterung der Standardwerkzeugkette und einer Bibliothek zum Auslesen der Attribute aus den Binärdateien fortgefahren.

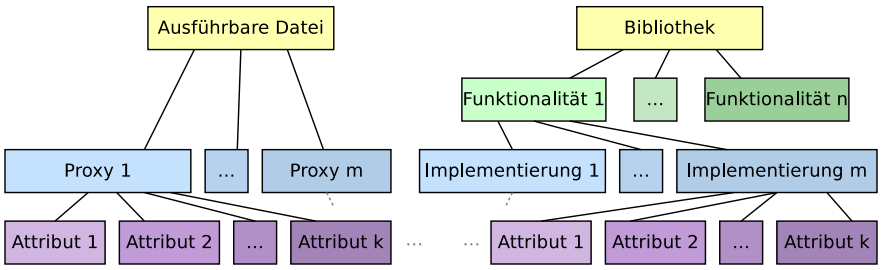


Abbildung 7.5: Baumartige Organisation der Proxies und Funktionalitäten in ausführbaren Dateien und Bibliotheken sowie deren Attribute.

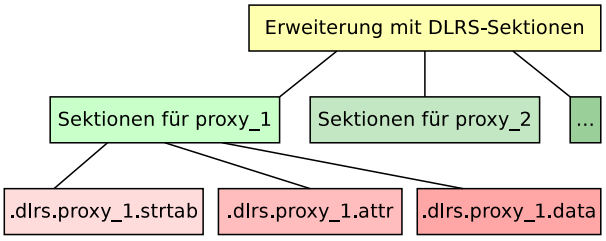


Abbildung 7.6: ELF-Sektionen für jeden Proxy bzw. Funktionalität mit Wörtertabelle, Attributen und ggf. zusätzlichen Daten.

7.2.1 DLRS-Sektionen

Der Großteil an Attributen besteht aus Zeichenketten wie den Funktionsnamen, den Schlüsseln der Attribute und häufig auch den Werten zu den Schlüsseln. Speichert man diese Wörter in Tabellen und greift mittels Indizes auf die Einträge zu, muss jedes Wort nur einmal gespeichert werden, und es kann sehr viel Platz gespart werden, da davon auszugehen ist, dass gewisse Schlüsselwörter in vielen Implementierungen vorkommen. Es würde sich anbieten, diese Informationen in den speziell für Zeichenketten vorhandenen, globalen Tabellen einer ELF-Datei zu speichern. Dies könnte jedoch die reguläre Programmausführung möglicherweise ändern, weshalb von der Speicherung in den dateiglobalen Tabellen abgesehen wird.

Aufgrund des logischen Aufbaus und Zusammenhangs von Funktionsaufrufen und Implementierungen, die als mögliches Ziel dienen, bietet es sich an, die Attribute in einem Baum wie in Abb. 7.5 zu organisieren. Bei ausführbaren Dateien (Anwendungen) hat jeder Stellvertreter (sogenannter *Proxy*, also eine bestimmte Komponente) eine Menge von Attributen, welche die Anforderungen beschreiben. Bei Bibliotheken werden die Implementierungen zusätzlich nach ihrer implementierten Funktionalität organisiert. Jede Implementierung leistet idealerweise das gleiche, bspw. die Berechnung eines Skalarprodukts[1]. Zur Unterscheidung dienen dann die Attribute.

Ein Proxy soll auf mehrere unterschiedliche Implementierungen verweisen können, welche idealerweise in unterschiedlichen Bibliotheken, etwa für jede Zielhardware, untergebracht sind. Jeder Proxy bzw. jede Funktionalität benötigt eine Tabelle (`strtab`) mit Wörtern für die Schlüssel und Werte als eigene Sektion. In einer weiteren Sektion (`attr`) sind die Tupel enthalten, welche Schlüssel und Werte miteinander verbinden, in einer letzten können zusätzliche Daten wie Bitstreams stehen. Dies ist in Abb. 7.6 illustriert. Die Wörtertabelle könnte gemeinsam für mehrere Proxies/Implementierungen genutzt werden. Jede Implementierung benötigt jedoch ihre eigene Paarungstabelle. Deshalb werden diese Proxy-Sektionen anhand ihrer Funktionsnamen benannt. Um gegebenenfalls auch einen Bitstreams zur partiellen Rekonfiguration eines FPGAs oder ein Mikroprogramm für das in Abschnitt 6.4 vorgeschlagene Rahmenwerk zur Verfügung bereitstellen zu können, existiert auch ein Eintrag für beliebige Daten (`data`).

Die entwickelte Bibliothek `libdlrs` ermöglicht es einerseits, die DLRS-Objekte mit `dlrsobj` überhaupt zu erstellen, und bietet andererseits Lesezugriff auf insbesondere die Bibliotheksseite mit

[1] Unterschiedlich genaue und dadurch unterschiedlich schnelle Implementierungen sind natürlich explizit gewünscht.

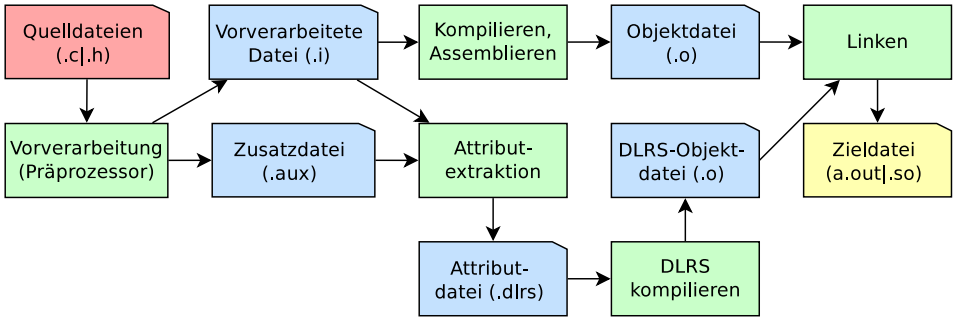


Abbildung 7.7: Ablauf der Werkzeugkette beim Erstellen attributierter ELF-Binärdateien.

Komponentenaufrufen bzw. -implementierungen und ihren Attributen. Zusätzlich profitiert das Werkzeug zur Fehlersuche, `dlrsdump`, von dieser Schnittstelle.

7.2.2 Werkzeugkette

Wie bereits erwähnt, musste eine Werkzeugkette entwickelt werden, um die Attribute aus den Quelltexten überhaupt zu erhalten und schließlich in den ELF-Dateien dem Anwender und Laufzeitsystem zugänglich zu machen. Die Werkzeugkette ist in Abb. 7.7 angedeutet und basiert auf der Standardwerkzeugkette bestehend aus GCC mit Präprozessor und abschließendem Linken der Objektdateien. Mit dem Präprozessor werden zunächst aus den Quelltexten Zwischenrepräsentationen[2] erzeugt. Daraufhin werden die Quelltexte mit GCC vollends zu Objekten kompiliert und Zusatzdateien[3] ausgegeben. Anhand der Zwischen- und Zusatzdateien lassen sich die Attribute aus den verbliebenen Pragmas leicht extrahieren, wozu ein in Python geschriebener zusätzlicher Präprozessor `dlrsextract` eingesetzt wird. Die Möglichkeit zum Einsatz von weiteren Pragmas für z.B. OpenMP wird durch ihn nicht unterbunden, da vorab seines Einsatzes bereits alle dem regulären Präprozessor bekannten Pragmas behandelt wurden. Der dedizierte Compiler `dlrsobj` erzeugt schließlich die DLRS-Objektdateien mit den Attributstrukturen wie oben beschrieben. Die Objektdateien der übersetzten Quelltexte werden mit den DLRS-Objekten vom regulären Linker zu einer ausführbaren Datei oder einer gemeinsam nutzbaren, dynamischen Bibliothek zusammengeführt.

Ferner wurde ein grafisch gestützter Prototyp zum Überprüfen der Segmente und Sektionen entwickelt. Ein Eindruck kann aus Abb. 7.8 erhalten werden.

7.2.3 Evaluation der Implementierung

Auf einem Intel Core2 Quad Q6600 mit 2,4 GHz wurde eine Reihe von Messungen durchgeführt, um den Zusatzaufwand durch größere Dateien und verlängerte Kompilierungsdauer sowie die Dauer des Auslesens zu bewerten.

Zunahme der Dateigröße. Anhand der Organisation und Struktur der Attributdaten lässt sich natürlich bereits der zusätzliche Aufwand abschätzen oder sogar errechnen. Interessant ist jedoch, die Platzersparnisse bei Wiederverwendung der Attribute zu bewerten. Die Größen der vollständig gelinkten Dateien sind für unterschiedlich viele Attribute und Implementierungen zu einem Proxy bzw. einer Funktionalität in Tabelle 7.2 zusammengestellt. Die Attributtupel aus Schlüssel und Wert haben in diesem Fall jeweils 5 Bytes Länge, daher ändert sich für weitere Attribute die Dateigröße. Die allgemeine Organisationsstruktur benötigt 96 Bytes je Proxy bzw. Funktionalität, wie sich aus der Spalte mit 0 Attributen gegenüber der regulären Dateigröße erkennen lässt. Ein Attribut einer Funktionalität benötigt 298 Bytes. Wird dieses 100 mal verwendet, so beträgt der zusätzliche Speicherbedarf nicht $100 * 298 = 29.800$ Bytes, sondern lediglich $24.155 - 18.315 = 5.840$

[2] Intermediate files, `.i`-Dateien
[3] Auxiliary files, `.aux`-Dateien

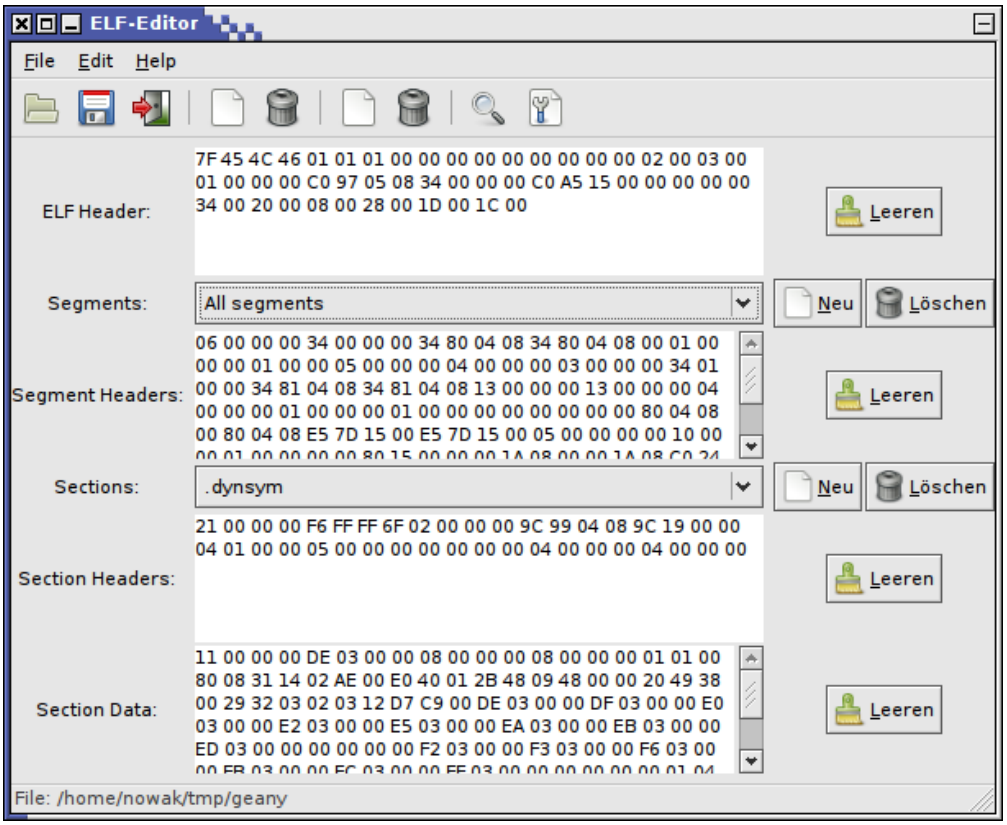


Abbildung 7.8: ELF-Editor zum Überprüfen der Segmente und Sektionen.

Anzahl Implementierungen	**Regulär**	**DLRS** (# Attribute)				
		0	1	10	50	100
1	10.952 B	11.048 B	11.336 B	11.593 B	12.713 B	14.106 B
2	10.985 B	11.081 B	11.433 B	11.818 B	13.578 B	15.787 B
10	11.242 B	11.338 B	12.138 B	13.675 B	20.571 B	29.164 B
50	16.618 B	16.714 B	19.754 B	27.051 B	59.547 B	100.140 B
100	18.219 B	18.315 B	24.155 B	38.668 B	103.148 B	183.741 B

Tabelle 7.2: Dateigröße in Bytes, wenn die gleichen Attribute für die unterschiedlichen Implementierungen wiederverwendet werden.

Bytes. Werden hingegen 100 unterschiedliche Attribute gespeichert, so benötigt ein Attribut gemittelt gegenüber der unattributierten, regulären Datei $(14.106 - 10.952)/100 = 31,54$ Bytes. Für 100 unterschiedliche Implementierungen verhundertfacht sich dieser Zusatzbedarf ebensowenig. Die Namen der Implementierungen werden jedoch auch gespeichert, was damit dennoch zu einer Erhöhung des Speicherbedarfs gegenüber dem für eine einzelne Funktionalität führt. In Tabelle 7.3 sieht man, dass ohne Wiederverwendung von Attributen, also auch ohne die intelligente Speicherung, der Speicheraufwand bis um das 1,65-fache steigt. Für jeden weiteren Proxy oder jede weitere Fuktionalität ist asymptotisch jeweils der gleiche Zusatzaufwand hinzuzurechnen, da deren Tabellen nicht mehr gemeinsam genutzt werden (vgl. Abb. 7.6).

Auslesen von Attributen. Wesentlich interessanter als die Dateigröße ist die Zeit, die benötigt wird, die Informationen zu Proxies und ihren Attributen aus den Dateien zu erhalten. Die Ergebnisse der Zeitmessungen sind in Tabelle 7.4 aufgeführt für 1, 10 und 100 Implementierungen sowie jeweils 1, 10 und 100 Attribute pro Implementierung. Die benötigte Zeit liegt jeweils im Bereich weniger Millisekunden.

Weitergehend wurde die Zeit zum selektiven Auslesen einzelner Attribute gemessen, wenn zehn Funktionalitäten mit je 100 Implementierungen und Attributen in der Binärdatei vorhanden sind. Die Implementierungen sind nicht notwendigerweise geordnet gespeichert, sondern die Reihenfolge

Anzahl Implementierungen	**Regulär**	**DLRS** (# Attribute)				
		0	1	10	50	100
2	10.987 B	11.083 B	11.435 B	11.932 B	14.172 B	16.973 B
10	11.244 B	11.340 B	12.140 B	14.653 B	25.853 B	39.854 B
50	16.620 B	16.716 B	19.756 B	33.757 B	89.757 B	159.758 B
100	18.221 B	18.317 B	24.157 B	52.158 B	152.958 B	304.159 B

Tabelle 7.3: Dateigröße in Bytes, wenn einzigartige, unterschiedliche Attribute verwendet werden.

Anzahl Attribute	**Anzahl Implementierungen**			
	1	10	50	100
1	2,7 ms	2,2 ms	2,9 ms	3,0 ms
10	2,5 ms	2,6 ms	2,9 ms	3,0 ms
50	2,3 ms	2,7 ms	3,0 ms	3,1 ms
100	2,7 ms	2,9 ms	3,0 ms	4,0 ms

Tabelle 7.4: Zeitbedarf zum Einlesen sämtlicher Attribute.

wird von den Bibliotheksroutinen bestimmt. Die sequentielle Suche nach dem letzten Eintrag in der Wörtertabelle führt zu zunehmendem Aufwand, wie aus Tabellen 7.5 und 7.6 erkenntlich wird beim Auslesen aus der ersten bzw. letzten Funktionalität.

Aus den geringen Abweichungen von maximal 1,26 %, die wesentlich auf das Betriebs- und Dateisystem zurückzuführen sind, lässt sich schließen, dass die Reihenfolge der Speicherung nicht ausschlaggebend für die erzielbare Leistung beim Auslesen der Attribute ist und dass somit der Zugriff wahlfrei erfolgen kann. Weiter bedeuten die geringen Zugriffszeiten von etwa 4,7 ms, dass kein negativer Einfluss auf die Ausführungszeit von Anwendungen zu befürchten ist und dass die Attributierung bereits dann vorteilhaft ist, wenn die Wahl einer alternativen Implementierung einen Zeitvorteil von 4,7 ms zuzüglich der für die Entscheidung benötigten Zeit verschafft. Die Verlagerung der Berechnung des lückenlosen Scores auf den Koprozessor der Convey HC-1 in Abschnitt 4.2.3 beispielsweise bringt eine Zeitersparnis von mindestens 3,2 Sekunden, so dass die 4,7 ms vernachlässigenswert sind.

Ausführungszeit der Werkzeugkette. Für unterschiedlich viele Attribute wurde die Ausführungszeit der gesamten erweiterten Werkzeugkette gemessen gegenüber der unveränderten Werkzeugkette. Zusätzlich wurde in weiteren Läufen die Dauer der zusätzlichen Werkzeuge für die Extraktion (`dlrsextract`) der Attribute und das Erstellen (`dlrsobj`) einer Objektdatei ermittelt. Die Messergebnisse sind in Tabelle 7.7 aufgeführt. Die zusätzlichen Werkzeuge benötigen zwar nahezu ein Drittel der Ausführungszeit der regulären Werkzeugkette, sind aber dennoch weitgehend unabhängig von der Anzahl an Attributen. Bei größeren Projekten ist davon auszugehen, dass die Ausführungszeit von `dlrsextract` und `dlrsobj` kaum zunehmen wird, da nur die Zwischendateien anstelle des gesamten Quelltextes untersucht werden müssen. Insgesamt ist die benötigte Zeit zum Kompilieren vertretbar gering im Bereich von 20 bis 30 Millisekunden, so dass der Nutzung der Attributierung für die Zukunft nichts im Wege steht.

Tabelle 7.8 bestätigt dies mit der nahezu konstanten Ausführungszeit von `dlrsextract` für unterschiedlich viele Implementierungen, die keine gemeinsamen Attributnamen haben. Der höhere

Attribut	**Implementierung**		
	erste	letzte	Mittelwert
erstes	4,703 ms	4,734 ms	4,718 ms
letztes	4,718 ms	4,750 ms	4,734 ms
Mittelwert	4,710 ms	4,742 ms	

Tabelle 7.5: Zeit zum Einlesen des ersten und letzten Attributs der ersten und letzten Implementierungen der ersten Funktionalität.

Attribut	**Implementierung**		
	erste	letzte	Mittelwert
erstes	4,718 ms	4,763 ms	4,740 ms
letztes	4,728 ms	4,742 ms	4,735 ms
Mittelwert	4,723 ms	4,753 ms	

Tabelle 7.6: Zeit zum Einlesen des ersten und letzten Attributs der ersten und letzten Implementierungen der letzten Funktionalität.

Anzahl Attribute	Reguläre Werkzeugkette	Erweiterte Werkzeugkette	dlrsextract	dlrsobj
0	74,3 ms	108,2 ms	23,2 ms	5,2 ms
1	74,4 ms	113,6 ms	23,3 ms	5,0 ms
10	73,9 ms	111,8 ms	23,4 ms	5,0 ms
50	74,5 ms	111,1 ms	23,2 ms	5,1 ms
100	74,7 ms	112,7 ms	23,6 ms	6,0 ms

Tabelle 7.7: Ausführungsdauer der erweiterten gegenüber der regulären Werkzeugkette für eine Implementierung und unterschiedlich viele Attribute.

Anzahl Implementierungen	Reguläre Werkzeugkette	Erweiterte Werkzeugkette	dlrsextract	dlrsobj
2	77,0 ms	113,8 ms	23,1 ms	5,1 ms
10	80,8 ms	118,7 ms	23,2 ms	5,0 ms
50	100,0 ms	144,3 ms	23,5 ms	6,2 ms
100	122,1 ms	175,8 ms	23,5 ms	7,7 ms

Tabelle 7.8: Ausführungsdauer der erweiterten gegenüber der regulären Werkzeugkette für unterschiedlich viele Implementierungen mit jeweils zehn Attributen.

Aufwand wird maßgeblich durch das Linken verursacht, während der Beitrag durch die neuen Werkzeuge weiterhin nahezu konstant und gering ist.

Zusammenfassung zur Evaluation der Implementierung. Anhand verschiedener Untersuchungen wurde gezeigt, dass die Implementierung der Attribute und die Anwendung der Werkzeugkette keine nennenswerten Nachteile hinsichtlich Speicherbedarf oder zeitlichen Aufwands mit sich bringen, sondern lediglich den gewünschten Vorteil bringen, die Programmausführung auf heterogenen Systemen zu unterstützen. Das dazu nötige Auslesen der Attribute ist sehr schnell durchführbar.

7.3 Einsatz und Verwendung von Attributen

Mit Hilfe von Attributen soll eine zusätzliche Abstraktion geschaffen werden, welche die Entkopplung von Anwendungen und hardwarespezifischen Teilen unterstützt, um so portable Anwendungen zu schaffen, welche auf dem jeweiligen System optimal zu jedem Zeitpunkt die verfügbare Hardware nutzen können. Dazu ist es nötig, dass die benötigten Teile erst zur Laufzeit bei Bedarf geladen werden. Die Entscheidung, welche Teile zu laden sind, wird durch die Angaben der Anwendungsanforderungen seitens des Anwendungsentwicklers und der (Hardware-)Implementierungseigenschaften seitens des Bibliotheks- oder Hardwarentwicklers unterstützt. Kommen mehrere Implementierungen in Frage, so wird zusätzlich ein verlaufsgestütztes Lernverfahren als Hilfe eingesetzt. In MPI-Anwendungen können aufgrund der Entkopplung in jedem Knoten sogar unterschiedliche Implementierungen verwendet werden. Der verursachte Zusatzaufwand ist als gering zu bewerten. Dieser Abschnitt basiert auf einer gemeinsamen Veröffentlichung [KNBK12].

7.3.1 Spezifikation der Systemumgebung

Normalerweise wird anwendungsspezifischer Beschleunigercode direkt mit einer Anwendung entwickelt und ausgeliefert. Dadurch ist die Anwendung auf anderen Systeme nicht mehr ausführbar, wenn der Beschleuniger nicht vorhanden ist. Der häufig gewählte Ansatz, ein sogenanntes „Fat Binary“ zu erzeugen mit Codes für alle möglichen Beschleuniger, hat das Problem, dass alle gelinkten Bibliotheken wie CUDA, OpenCL, Koprozessortreiber im Zielsystem installiert sein müssen, was aus Wartungssicht äußerst unpraktikabel und damit nicht akzeptabel ist. Daher wird die Entkopplung von Funktionsaufruf und Implementierung auf solche Art vorgeschlagen, dass die Implementierungen in separaten Bibliotheken untergebracht sind, die ausgetauscht werden können, wie es in Abbildung 7.9 dargestellt ist. Damit hängt die Anwendung von weit weniger Bibliotheken ab; einzig

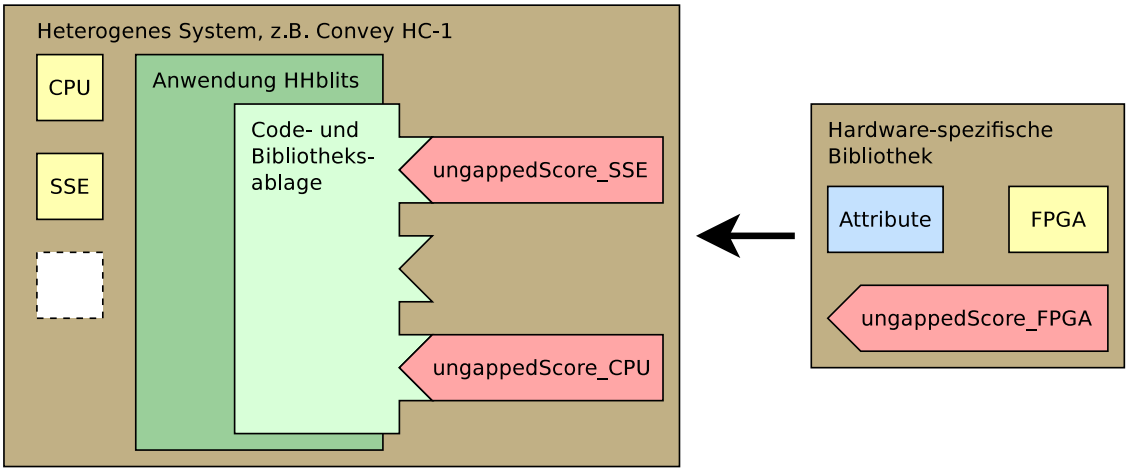


Abbildung 7.9: Transparente Erweiterung einer Anwendung mit einer hardwarespezifischen Bibliothek zur Nutzung zusätzlicher Hardware.

gegen die Schnittstelle zum Laufzeitsystem muss gelinkt werden. Umgekehrt wird damit der einmal entwickelte Beschleunigercode wiederverwendbar. Zur genauen Spezifikation der implementierten Funktionalität sind die Attribute einsetzbar. Sind Proxies bzw. Funktionalitäten bereits spezifiziert, können damit auch weitere Implementierungen nachgereicht werden und somit nahtlos ein System erweitern. Beispielsweise ist auch vorstellbar, dass FPGA-Ressourcen über Ethernet zur Laufzeit an das Hostsystem angebunden werden. Ausgeschlossen sind natürlich unterschiedliche Instruktionssätze für unterschiedliche Host-CPUs. Um die Suche nach passenden Implementierungen zu erleichtern, sollten nicht beliebige Bibliotheken attributiert werden. Die attributierten Bibliotheken werden in einer zentralen Codeablage registriert, und ein Laufzeitsystem wie in Abb. 7.1 durchsucht nur die darin eingetragenen Bibliotheken nach passenden Implementierungen.

7.4 Evaluation

Der Nutzen und die Kosten der attributgesteuerten, auf Lernverfahren basierenden Ausführung von Anwendungen auf heterogenen Systemen wurden genauer bestimmt. Zur Evaluation wurden zwei GPU-Systeme (AMD Opteron Quad-Core 2378 mit NVIDIA GeForce GTX 275; Intel Core2 Quad mit NVIDIA GeForce 8400 GS), ein Zwölfkernsystem (Intel Xeon X5670 Six-Core) und ein FPGA-erweitertes System (AMD Opteron Dual-Core Processor 870 mit Xilinx Virtex-4 FX100 auf UoH HTX-Board) mit H-MOL [KVB+09] verwendet.

7.4.1 Mehraufwand

Zur Bewertung des statischen Anteils bei der attributgestützen und auf Lernverfahren basierenden Anwendungsausführung wurde eine einfache Anwendung erstellt, die nur eine einzelne Funktion hat. Ihre Ausführungszeit beträgt 3,9 ms. Verbunden mit dem Laufzeitsystem erhöht sich diese auf 7,5 ms, weshalb die Einmalkosten zur Initialisierung des gesamten Systems bei nur 3,6 ms liegen. Der Aufwand pro Funktionsaufruf lässt sich aus der Differenz bei 10^7-fachem Aufruf der Funktion berechnen. Die reguläre Ausführung benötigt dazu nur 50 ms. Mit dem Laufzeitsystem sind 7,8 s nötig, so dass der Aufwand $\frac{(7{,}8\,\mathrm{s}-7{,}5\,\mathrm{ms})-(50\,\mathrm{ms}-3{,}9\,\mathrm{ms})}{10.000.000-1} \approx 775\,\mathrm{ns}$ beträgt.

7.4.2 Erzeugung von Zufallszahlen

Die Zufallszahlenerzeugung ist ein äußerst interessanter Anwendungsfall, weil es mehrere Eigenschaften und Anforderungen bei Zufallszahlen gibt. In Tabelle 7.9 sind die Attribute zusammengetragen. Über `periodicity` kann sichergestellt werden, dass sehr viele unterschiedliche Zahlen

Attribut	Implementierung	Wert
`quality`	/dev/random	3 (hoch)
	MT *	2 (mittel)
	random()	1 (niedrig)
`periodicity`	/dev/random	-
	MT *	2^{19937}
	random()	2^{35}
`random_type`	/dev/random	real
	MT *	pseudo
	random()	pseudo
`psize`	/dev/random	0-10
	MT H-MOL	0-1.000

Tabelle 7.9: Attributierung der Zufallszahlenerzeuger.

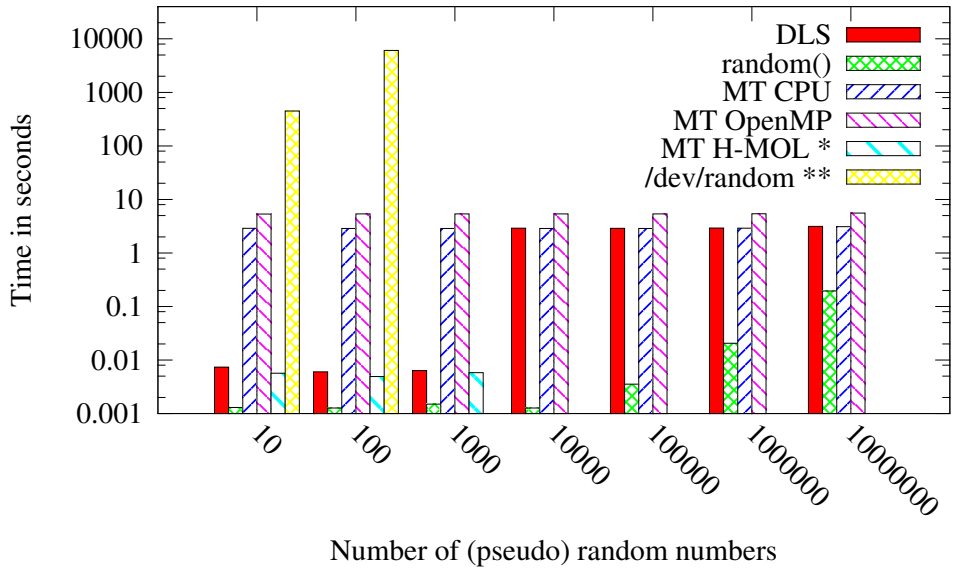


Abbildung 7.10: Durchschnittliche Laufzeit der MT-Bibliothek, Systemfunktionen und Hardwarebeschleuniger.
(* = Einschränkungen der Hardware; ** = 8 Läufe für 100.)

erzeugt werden, mit `random_type` können echte Zufallszahlen angefordert werden, mit `psize` sinnvolle Problemgrößen angegeben werden und schlussendlich mit `quality` grob hinsichtlich der Qualität klassifiziert werden. Zufallszahlen werden in sehr großer Menge bei Monte-Carlo-Simulation benötigt, beispielsweise bei der Simulation oder Vorhersage von Aktienmärkten. Eine Implementierung des *Mersenne Twisters* (MT) basierend auf der *Dynamic Creator Library*[4] wurde für die CPU gestellt und in der OpenMP-Version wurden mehrere dieser MTs parallel zueinander ausgeführt. Die frei erhältliche FPGA-Implementierung[5] ist in das erste Steuerwerk H-MOL (Abschnitt 6.3, [KVB+09]) integriert. Linux selbst bietet die Funktion `random()` der GNU C-Bibliothek, die nur Pseudozufallszahlen erzeugt, und die virtuelle Datei `/dev/random`, die echte Zufallszahlen von besonders hoher Qualität liefert, solange der Entropie-Pool des Linux-Kernels ausreichend gefüllt ist, die sonst aber sehr langsam ist.

Die Messungen über 30 Läufe sind in Abb. 7.10 dargestellt. OpenMP lohnt sich aufgrund des Aufwands zur Initialisierung unterschiedlicher Mersenne Twister-Generatoren erst ab 1 Millarde Zufallszahlen. Bei sehr kleinen Mengen ist die H-MOL-Version sehr gut. `/dev/random` kann nur für sehr wenige Zufallszahlen verwendet werden. Bei der attributierten Ausführung wurde ein

[4] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html`
[5] URL: `http://www.ht-lab.com/freecores/mt32/mersenne.html`

hohes Maß an Qualität verlangt (`quality=2+`). Das Laufzeitsystem wählt erkennbarerweise die richtige Implementierung, welche für 10 bis 1000 Zahlen aus MT H-MOL besteht und später aus der sequentiellen CPU-Version, da die Qualität der übrigen Implementierungen nicht ausreichend ist und `/dev/random` wesentlich langsamer, wie das Laufzeitsystem durch eine Projektion der wenigen ersten Messwerte selbständig erkennt.

7.4.3 Anwendbarkeit bei HHblits

Nachdem gezeigt wurde, dass es möglicht ist, mit Attributen und Lernverfahren die Programmausführung in heterogenen Systemen zu regeln, können die hier erzielten Ergebnisse auch auf weitere Anwendungen appliziert werden. Von HHBlits ist die Ausführungszeit für die Suche durch die gesamte Datenbank bekannt. Sie beträgt 44,5 s (vgl. Tabelle 4.7). Weiterhin bietet die Ausführung mit dem Koprozessor nur eine Beschleunigung von $1,08\times$ und eine Verkürzung der Ausführungszeit auf 41,3 s. Durch die Verlagerung des Vorfilterungsschrittes können jedoch 25,1 s - 13,1 s = 12,0 s eingespart werden. Dazu muss für die Berechnung des lückenlosen Scores lediglich eine andere Bibliothek/Implementierung aufgerufen werden. Den Nutzen der Verlagerung kann das Laufzeitsystem noch feiner im Intervall [0; 500] der Eingabesequenzlängen bestimmem. Damit kann sich eine Verkürzung auf 44,5 s - 12,0 s = 32,5 s erzielen lassen (die Millisekunden zur Entscheidungsfällung sind vernachlässigbar). Somit ist theoretisch eine Beschleunigung der Gesamtanwendung von $\frac{44{,}5\,s}{32{,}5\,s} = 1,37$ erreichbar.

7.5 Ergebnis und Zusammenfassung

Gewisse numerische Verfahren können von erhöhter Genauigkeit bei einzelnen Operationen wie dem Skalarprodukt profitieren. Der Anwendungsentwickler möchte solche Anforderungen an beispielsweise die Skalarproduktkomponente bekanntgeben und bei Nichtvorhandensein einer exakteren Implementierung auf alternative mathematische Verfahren wie Reorthogonalisierung zurückgreifen. In diesem Kapitel wurde dazu der folgend zusammengefasste Lösungsansatz vorgestellt.

Zur Auszeichnung von Eigenschaften vorhandener Funktionsimplementierungen sowohl in Software als auch in Hardware wird das Konzept der Attributierung eingesetzt. Die Attribute werden zunächst als Pragmas angegeben, werden nach der Kompilierung in den ausführbaren Dateien festgehalten und zur Laufzeit in einer Datenbank gesammelt und erweitert. Das Konzept erfordert für den Entwickler der Bibliothekskomponenten nur wenig Zusatzaufwand bei der Werkzeugkette.

Gleichermaßen können Anwender die Funktionsaufrufe in ihren Quelltexten hinsichtlich möglicher Anforderungen attributieren. Die Attribute lassen sich ferner in grafischen Programmierumgebungen direkt beim Funktionsaufruf angeben. Ein Laufzeitsystem gleicht die Anforderungen mit den Eigenschaften der verfügbaren Implementierungen ab. Da der Nutzen von Spezialschaltungen häufig auch von der verarbeiteten Datenmenge abhängt, werden zusätzlich Profile über die benötigte Ausführungszeit erstellt. Dadurch stehen für spätere Aufrufe zusätzliche Angaben zur Verfügung, welche die Wahl einer passenden Implementierung zusätzlich beeinflussen können, so dass bessere Entscheidungen für die nächsten Läufe getroffen werden können. Es konnte gezeigt werden, dass die richtigen Implementierungen ausgewählt werden bei sehr geringem Zusatzaufwand zur Laufzeit. Da das Laufzeitsytem auf jedem Knoten in einem Cluster diese Entscheidungen separat vornimmt, kann eine MPI-Anwendung von den jeweils verfügbaren und schnellsten Varianten profitieren. Somit hat sich das Konzept der Attributierung als vorteilhaft erwiesen. Das nächste Kapitel beschäftigt sich mit der möglichst engen Verzahnung der in Software und Hardware ausgeführten Funktionalitäten.

Kapitel 8

Programmierung und Ausführung numerischer Anwendungen in heterogenen Systemen

Bislang wurde gezeigt, dass rekonfigurierbare Hardware für Anwender Nutzen versprechen kann, sogar wenn Mehrkernsysteme zur Verfügung stehen. Dazu müssen die Hardwareimplementierungen sehr sorgfältig implementiert sein (siehe Kapitel 4) und durch geringen Ressourcenbedarf die Ausnutzung von Datenparallelismus qua mehrfacher Einbindung der Implementierung maximieren sowie die verfügbare Speicherbandbreite ausnutzen. Häufig handelt es sich dabei um Spezialfunktionen, welche auf besonderen Datentypen arbeiten oder besondere Funktionalität bereit stellen (Abschnitt 5.6), die in Software nur sehr mühselig und ineffizient umzusetzen ist. Die besonderen Fähigkeiten dieser Implementierungen lassen sich quantitativ sowie qualitativ mittels Attributen angeben, wie im vorangegangen Kapitel beschrieben. Ferner muss eine geeignete Granularität gewählt werden, welche die Ausnutzung von Datenlokalität ermöglicht und die Speicherschnittstelle zu entlasten bzw. besser auszunutzen vermag. Um dem Anwendungsentwickler die Möglichkeit zu geben, die grobgranulare Funktionalität selbst unter Verwendung der vom Hardwareexperten entwickelten und bereitgestellten Hardwareimplementierungen nahezu beliebig zusammenzustellen, wurde ein mikroprogrammierbares Steuerwerk entwickelt (vgl. Abschnitt 6.4). Es ermöglicht die Datenwiederverwendung durch einen zentralen Puffersatz und die datengetriebene Ausführung der aufgerufenen Komponenten. Es verbleibt noch, den Koprozessor auch datengetrieben in die Anwendung einbinden und mit den in Software ausgeführten Komponenten zeitgleich ausführen zu können, um folgende Ziele zu erreichen:

- die Umsetzung des Kontrolldatenflussgraphens (CDFG) in reiner Software zum Zwecke des Tests und der bestmöglichen Validierung vorab der Verlagerung in die Hardware,
- die Versorgung des Koprozessors mit Daten während der Ausführungszeit anstelle des weitverbreiteten „Offloading“-Prinzips, bei welchem Eingabedaten auf die Koprozessorressource vollständig kopiert werden, bevor auf ihnen gearbeitet wird, und nach erfolgter Berechnung die Ergebnisse zurück transportiert werden,
- die iterative Verlagerung von zunehmend größeren Teilen des CDFGs auf die Beschleunigerhardware, bis kein weiterer Nutzen mehr durch die Verlagerung erzielbar ist.

8.1 Ein datengetriebenes, taskparalleles Programmier- und Ausführungsmodell für Mehrkernsysteme

Ein Programmier- und Ausführungsmodell wird benötigt, das nicht nur für numerische Anwendungen anwendbar ist, sondern auch auf weitere Domänen übertragbar ist. Zumeist wird auf Systemen mit gemeinsamem Speicher über die datenparallele Ausführung Nutzen aus mehreren Prozessorkernen erzielt. Dieser Ansatz ist nicht immer ausreichend zum Verarbeiten großer Datenmengen, insbesondere dann, wenn es sich um asymmetrische, NUMA-Systeme handelt, bei denen die Latenzen für Speicherzugriffe unterschiedlich ausfallen. Da die Caches keine großen Datenmengen bereitstellen können, werden die Daten aus dem Hauptspeicher gelesen und die Ergebnisse in selbigen zurückgeschrieben, ohne Nutzen aus den Caches zu ziehen. Dies geschieht bei jedem neuen Funktionsaufruf, der zwar intern parallelisiert sein mag, ist aber ineffizient. Stattdessen sollten mehrere Funktionen so aufgerufen werden, dass sie zueinander parallel ausgeführt werden, so dass

die Daten noch aus dem Cache heraus wiederverwendet werden können, bevor sie verdrängt oder zurückgeschrieben werden.

8.1.1 Das datenflussbasierte, taskparallele Konzept

Um mehrere Funktionen gleichzeitig, taskparallel, auszuführen, sollten die Funktionen datengetrieben ausgeführt werden. Sobald gültige Eingabedaten vorliegen, werden die Funktionen ausgeführt, und zwar parallel zu den erzeugenden Funktionen sowie zu eventuell von den aktuell berechneten Werten abhängenden, weiteren Funktionen. Vergleiche hierzu auch Abschnitt 6.1.

Da mit Task-Parallelismus jedoch nicht zwangsläufig jeglicher zur Verfügung stehender Parallelismus erfolgreich ausgenutzt werden kann, muss zur vollständigen Ausnutzung des verfügbaren Parallelismus auch Datenparallelismus ausgenutzt werden. Es gibt also eine Menge von Arbeitsthreads, die selbst wieder datenparallel arbeiten lassen und dazu weitere Threads einsetzen. Der Entwickler sieht jedoch nur die von ihm aufgerufenen Tasks bzw. Arbeiterthreads im Rahmen dieses Programmiermodells. Diese ruft er zudem sequentiell auf, was der natürlichen Denkweise entgegenkommt und das Programmieren vereinfacht.

Die datengetriebene Ausführung folgt genau dem Erzeuger-Verbraucher-Muster. Jede Task erzeugt neue, zusätzliche Daten an einer anderen Stelle im Speicher und überschreibt die Eingangsdaten nicht, so dass sie von weiteren Tasks ebenso verwendet werden können[1]. Selbstverständlich müssen geeignete Maßnahme ergriffen werden, den unbenötigten, aber noch allozierten Speicher auch wieder freizugeben.

Idealerweise werden mehrere Iterationen bei iterativen Anwendungen zeitgleich bearbeitet, insofern keine Barrieren durch Skalaroperationen wie die Division eines Skalarproduktergebnisses durch ein weiteres Skalar starke Datenabhängigkeiten einführen oder Kontrollflussoperationen die zeitgleiche Ausführung verhindern. Für den letzteren Fall ist die Technik der spekulativen Ausführung ein geeigneter Lösungsansatz. Die Domäne numerischer Anwendungen beschränkt die Einheiten auf Matrixoperationen. Daher entfallen die Betrachtungen von Barrieren und Kontrollfluss weitestgehend.

Durch die datengetriebene Ausführung kann bereits früher mit einer Ausgabe begonnen werden, eben zeitlich überlappt zu den Berechnungen, und gleichermaßen können auch die Ergebnisse der Gesamtberechnung früher erhalten werden. Die zeitgleiche, überlappte Ausführung von Tasks kann vor allem auch dazu verwendet werden, das Laden von Daten mit der Weiterverarbeitung zu überlagern und dadurch den Datentransfer teilweise oder gar vollständig zu verbergen. Die überlappte Ausführung erreicht der Anwendungsentwickler, indem die hintereinander aufgerufenen Funktionen mittels POSIX-Threads aus einem Thread-Pool ausgeführt werden. Für die Domäne der numerischen Anwendungen wird eine zusätzliche Funktion zur Initialisierung der verwendeten Strukturen nötig. Neben der Initialisierung ruft die Haupt-Task nur noch die Komponenten auf, welche neben Berechnungen auch Festplattenzugriffe zum Lesen oder Schreiben von Daten selbst durchführen.

Die zur Steuerung der Komponenten benötigten Daten wie z.B. Zeiger auf Eingangsspeicher müssen bereits vorhanden sein und dürfen daher nur von der Haupt-Task erzeugt werden, wohingegen die zur Ausführung der Komponenten benötigten Daten noch von einem Produzenten erzeugt werden dürfen. Die Verbraucher warten auf die Menge benötigter Eingabedaten, aber es wird nur auf einzelne Daten gewartet und bei Gültigkeit der Daten werden komplexere Operationen in Streaming-Manier auf diesen durchgeführt – im Gegensatz zu feingranularen Operationen auf Einzeldaten beim ursprünglichen Datenflussmodell. Sind keine Daten verfügbar, so kann die verwendete Rechenressource einer anderen, derzeit nicht ausgeführten, aber bereits lauffähigen Komponente überlassen werden. Eingabedaten dürfen nicht mehrfach modifiziert werden, solange nicht klar ist, dass keine weitere Komponente von den alten Werten abhängig ist. Erst wenn all diese Aspekte berücksichtigt sind, kann eine Anwendung auch alle Komponenten über die zugehörigen Funktionsaufrufe der Reihe nach aufrufen, um sie dann allein unter Berücksichtigung der Datenabhängigkeit in beliebiger Reihenfolge auszuführen, wodurch der maximal vorhandene Parallelismus ausnützbar wird.

[1] Single Static Assignment (SSA)

Abstraktionsebene	Verantwortlichkeit	Parallelisierungsansatz	Synchronisierung
Anwendung	Anwendungsentwickler unter Verwendung von Hochsprachen	Taskparallelismus	Rangfolge der Funktionsaufrufe
Task (Funktionsimplementierung, Komponente)	Hardware- oder Bibliotheksentwickler	Datenparallelismus, z.B. mittels OpenMP	Gültigkeit von Teilbereichen oder Unterstrukturen

Tabelle 8.1: Verantwortlichkeit, Parallelisierungsansatz und Synchronisierung auf unterschiedlichen Abstraktionsebenen.

8.1.2 Umsetzungsaspekte

Damit die datengetriebene Abarbeitung funktioniert, müssen die Komponenten, also die erzeugenden Tasks, angeben, welche Teile wieweit bereits gültig und weiterverwendbar sind. Eine Bitmaske für jedes einzelne Datum stellt sicherlich die akurateste Variante dar, hat aber zu hohen Zusatzaufwand. Diese Angaben könnten daher auf eine höhere Ebene gezogen werden entgegen der gewöhnlichen datumsbasierten Gültigkeit und damit einhergehenden Synchronisation der Tasks im regulären Datenflussmodell.

Gemeinsam genutzte Datenstrukturen dürfen erst dann freigegeben werden, wenn sie vollständig verwendet sind und keine weitere Task mehr von ihnen abhängt. Um dies zu erreichen, werden POSIX-Locks eingesetzt. Sie sind nicht an einen spezifischen Thread gebunden, wodurch die Haupt-Task eine Datenstruktur bereits sperren kann, bevor überhaupt eine Task zum Freigeben der Daten diesen Lock erhält. Eine entsprechend erweiterte Implementierung einer Komponente zum Freigeben der Daten steht zur Verfügung, welche überprüft, dass abhängige Operationen vollständig abgeschlossen sind, indem auf die vollständige Gültigkeit der von ihr erzeugten Daten gewartet wird. Sind mehrere Operationen abhängig, so müssen die Daten als neue Eingangsdaten für diese Operationen repliziert werden, damit die Überprüfung für jede abhängige Operation stattfinden kann. Diese Replikation der Daten ist der einzige kritische Punkt, den Anwendungsentwickler sorgfältig beachten müssen. Grafisch basierte Werkzeuge zur automatischen Programmerstellung nach dem Datenflussmodell sollten in der Lage sein, diese Aufgabe selbst automatisiert zu übernehmen, wenn der Verzweigungsgrad eines Ausgangs einer Komponente größer als eins ist.

Am meisten Nutzen wurde bislang durch datenparallele Berechnungen auf den gleichen Daten mit mehreren Threads von den Mehrkernarchitekturen erlangt. Der Speicherzugriff stellt jedoch zunehmend den eigentlichen Engpass dar. Daher wird zwar auch die Technik des Multithreadings eingesetzt, jedoch nur in geringerem Umfang, wenn sich die datenparallele Verarbeitung der nächsten gültigen Daten überhaupt lohnt.

Globale Locks, also anwendungsweit, werden verwendet, um beispielsweise lesbare Ausgaben zu erhalten. Die Haupt-Task sichert sich diesen Lock, bevor eine Ausgaberoutine als Task aufgerufen wird. Die aufgerufene Task gibt diesen Lock nach Vollendung wieder frei. Der Vorteil liegt darin, dass Ausgaberoutinen vollständig in die datengetriebene Ausführung integriert werden.

Tabelle 8.1 gibt abschließend eine Zusammenfassung der Abstraktionsebenen hinsichtlich der diskutierten Aspekte.

Operationen auf Matrizen

Bei Matrizen kann die Gültigkeit auf Zeilen- bzw. Spaltenbasis angegeben werden. Dies stellt einen guten Mittelweg zwischen zusätzlichem Speicherbedarf und noch möglichem Ausnutzen der Datenlokalität in den Caches dar.

Die Komponentenimplementierungen sind intern mit OpenMP parallelisiert. Da bei OpenMP die Reihenfolge der generierten Daten jedoch nicht eindeutig ist, müsste eigentlich eine Bitmaske wie bereits angesprochen zur Kennzeichnung gültiger Daten verwendet werden. Daher sind nur die inneren Schleifen parallelisiert, welche jeweils eine vollständige Zeile oder Spalte berechnen. Die

äußere Schleife wartet ggf. auf gültige Eingangsdaten und gibt dann an, wieviele aufeinanderfolgende, zusammenhängende Zeilen bzw. Spalten bereits berechnet sind. Da unter OpenMP die erstellten Threads die Prozessorressource nicht mittels `yield` freigeben können, muss geschäftiges Warten[2] eingesetzt werden, um auf Verfügbarkeit noch fehlender Arbeitsdaten zu warten.

8.1.3 Behandlung des Kontrollflusses

Auch bei der datenflussorientierten Entwicklung spielt Kontrollfluss eine wichtige Rolle. Verzweigungen werden folgendermaßen behandelt: Alle zu evaluierenden Daten müssen bereits vollständig gültig sein, bevor sie in der Abfrage evaluiert werden, was sich durch geschäftiges Warten auf Datenverfügbarkeit erreichen lässt, also durch eine Barriere implementiert ist. Innerhalb der beiden möglichen Pfade können beliebige Komponenten aufgerufen werden und sogar die umgebende Schleife verlassen werden, z.B. wenn ein Abbruchkriterium erfüllt ist. Vorab der erfolgreichen Evaluation wird jedoch kein Pfad ausgeführt werden. Die spekulative Ausführung eines oder beider Pfade stellt eine mögliche Erweiterung dar. Dazu sind beispielsweise nicht-exponierte Hilfsvariablen einsetzbar, die bei Gültigkeit der Bedingung auf die Gültigkeitsanzeige der betreffenden Variablen des genommenen Pfads kopiert werden; die Operationen und Variablen des nicht genommenen Pfads müssten abschließend abgeräumt werden.

Es besteht keine Möglichkeit zu einer Wettlaufsituation[3] zwischen der Barriere und der Auswertung, da die Matrixdaten nicht verändert werden, wenn oben formulierte Regeln beachtet werden, und da dieser Teil nun sequentiell zu den erzeugenden Komponenten ausgeführt wird aufgrund der Barriere (SSA-Form). Die Daten werden also auch nicht zwischendurch freigegeben.

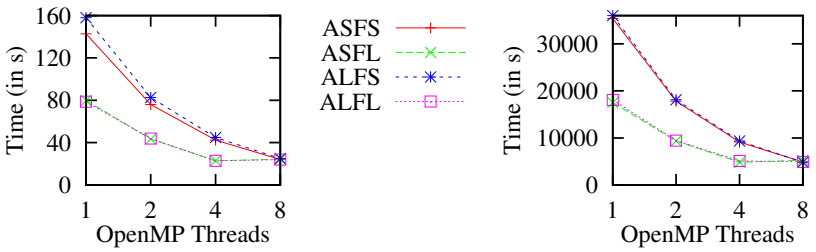
Schleifen sind sehr interessant bei der datengetriebenen Programmausführung. Natürlich kann vor dem Ende einer Iteration auf die Vollendung der in der aktuellen Iteration aufgerufenen Komponenten gewartet werden. Nach Möglichkeit sollen jedoch auch mehrere Iterationen überlappt zueinander ausgeführt werden, ähnlich dem Ausrollen von Schleifen. Dazu wartet jede Komponente lediglich auf das Vorhandensein der Arbeitsdaten, wie bereits ausführlich geschildert. Der Nachteil besteht offensichtlich darin, dass weit mehr Datenstrukturen angelegt werden müssen, nämlich für jede Iteration und jedes Objekt/Variable eine eigene, solange nicht dafür Sorge getragen wird, dass die Daten vor dem nächsten Beschreiben in einer neuen Iteration vollständig verbraucht wären. Es muss jedoch betont werden, dass das heutige Problem nicht die Größe des Speichers ist, sondern die Anbindung des Hauptspeichers und die Größe der Caches. Somit ist der vorläufige Ansatz, für jede Iteration bereits im Voraus eigene Variablen in Form eines Feldes anzulegen. Das Freigeben von Daten muss aufgrund der großen Speicher auch nicht gesondert betrachtet werden, sondern kann datengetrieben in einer der folgenden Iterationen angestoßen werden.

Der Ausgang der Anwendungsausführung muss genau dem der sequentiellen Ausführung entsprechen, wo vorab aufgerufene Funktionen immer vollständig ausgeführt sind. Daher werden vorab aufgerufene Komponenten selbstverständlich nicht abgebrochen durch Kontrollflussoperationen. Einzig die spekulative Ausführung dürfte aufgrund der Verwendung von für den Anwendungsentwickler nicht sichtbaren Hilfsobjekten abgebrochen werden.

8.1.4 Implementierungsalternativen

Bei der Implementierung des vorgeschlagenen Konzepts gibt es eine Reihe von Entwurfs- und Implementierungsalternativen, die untersucht werden müssen. Diese Untersuchungen wurden auf einem Intel Xeon X5670 Westmere-System mit zwei mal sechs Prozessorkernen und HyperThreading, also insgesamt 24 Hardwarethreads, durchgeführt.

[2] engl. busy waiting
[3] engl. race condition



(a) 2.000×2.000 (b) 10.000×10.000

Abbildung 8.1: Auswirkung der Reihenfolge der Operationen für einen großen Pool mit 20 POSIX-Threads (Allocate {Soon|Late} – Free {Soon|Late}).

Reihenfolge und Zeitpunkte der Aufrufe. Zur Multiplikation dreier Matrizen $D = (A \cdot B) \cdot C$. müssen drei Matrizen eingelesen werden und zwei Matrixmultiplikationen durchgeführt werden. Um die Speichernutzung gering zu halten, müssen die Daten auch noch freigegeben werden. Es gibt also mindestens drei Arten von Operationen: Datenzugriffe, Berechnungen und das Freigeben des verwendeten Speichers.

Sequentiell implementiert würde man erst die Daten einlesen und dann die Berechnungen der Produkte durchführen, zwischendrin die nicht mehr benötigten Eingabematrizen A und B freigeben. Da das Einlesen der Daten jedoch eine DMA-Operation ist, kann das Einlesen von C gleichzeitig durchgeführt werden. Ähnlich können A und B freigegeben werden, wenn $A \cdot B$ berechnet ist und nur noch D berechnet wird. Die Strategie, Daten so früh wie möglich zu allozieren und einzulesen (allocate soon – AS), hat den Nachteil, dass der Speicherverbrauch sehr schnell stark ansteigt und man zusätzlich die Speicherbandbreite unnötig belastet. Ferner werden mit hoher Sicherheit die später benötigten Daten nicht mehr im Cache liegen, sondern von anderen, später eingelesenen Daten verdrängt sein. Diese Effekte können vermieden werden, wenn Daten spätestmöglich alloziert und gelesen werden (allocate late – AL). Gleichermaßen kann die Speichernutzung verringert werden, wenn Datenstrukturen möglichst früh freigegeben werden (free soon – FS), wobei es wiederum zu langem Warten auf den Abschluss der von den Daten abhängigen Operationen kommen kann, weshalb es gleichermaßen vorteilhaft sein könnte, die Daten spät freizugeben (free late – FL). Für zwei Multiplikationen großer Matrizen wurden daher die möglichen Kombinationen der Allokations- und Freigabestrategien gemessen. In Abb. 8.1 sind die Ergebnisse zusammengefasst. Wie man leicht erkennen kann, sollten Daten möglichst spät freigegeben werden, da dann die Ausführungzeit wesentlich geringer ist. Tendenziell sollten den Messergebnissen zufolge Allokationen eher frühzeitig erfolgen.

Anordnung der Daten. Matrizen werden in der Programmiersprache C meist reihenbasiert gespeichert, d.h. zuerst alle Spalten der ersten Reihe, dann der nächsten Reihe usw. Demgegenüber werden sie bei Fortran eher spaltenorientiert abgelegt. Verbindet man diese zwei Anordnungen geschickt und greift in den Implementierungen entsprechend zeilen- oder spaltenbasiert auf die Einträge zu, so können weitere Cacheeffekte ausgenutzt werden – sogar von der Position des Anwendungsentwicklers aus. Insbesondere bei der Multiplikation zweier Matrizen A und B kann dadurch das datengetriebene Modell cacheffizient arbeiten, wenn A zeilenorientiert und B spaltenorientiert abgelegt ist und idealerweise auch die Gültigkeit von B spaltenorientiert angegeben wird.

Bei einer naiven Implementierung $AB := A \cdot B$ wären alle drei Matrizen zeilenorientiert gespeichert. Die zugehörigen Messungen sind mit „Row-major“ bezeichnet. Die Matrix B sollte hingegen eher spaltenbasiert gespeichert sein, so dass bereits mit den ersten Elementen $a_{1,k}$ und $b_{k,1}$ das Element $ab_{1,1}$ berechnet werden kann. Dies kann noch einmal ausgenutzt werden, wenn $A \cdot B$ zeilenorientiert abgelegt wird und C spaltenbasiert abgelegt ist(„Row-col-col“). Weiter kann optimiert werden durch vorgezogenes Lesen der spaltenbasierten Matrix B vor A, wodurch die Berechnung des Produkts $A \cdot B$ zeilenorientiert etwas schneller erfolgen kann, da die Daten schon bzw. idealerweise noch im Cache vorhanden sind. Für die folgende, in Abb. 8.2 dargestellte Untersuchung wurden nur vier OpenMP-Threads gegenüber den 24 verfügbaren Hardware-Threads eingesetzt, um nicht durch eventuell negative Entscheidungen beim Scheduling unterschiedlich beeinflusst zu werden.

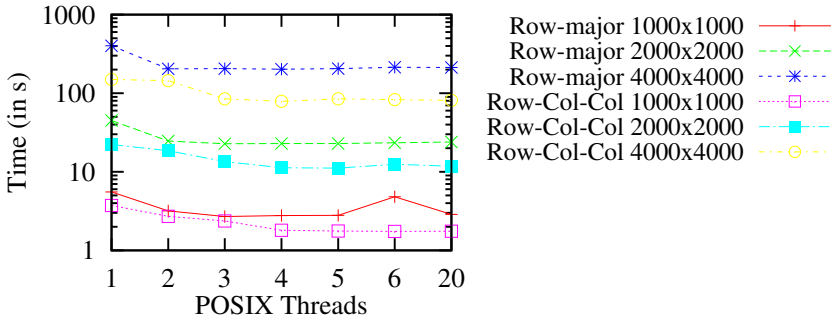


Abbildung 8.2: Einfluss der Datenanordnung für unterschiedliche Größen des Thread-Pools.

Entlang der horizontalen Achse wird erkennbar, dass mit zunehmender Anzahl an POSIX-Threads, also gleichzeitig ausgeführten Komponenten, die Laufzeit abnimmt. Die maximale Beschleunigung durch mehr POSIX-Threads schwankt zwischen 1,484 und 5,101, beträgt durchschnittlich 1,7. In den meisten Fällen genügen vier POSIX-Threads, wodurch nur maximal elf Threads insgesamt ausgeführt werden: je vier Threads für die zwei Matrixmultiplikationen, zwei zum Einlesen der Daten und schließlich der Master-Thread.

Durch das Umordnen der Operationen und Datenanordnung ergibt sich von „Row-major“ zu „Row-col-col“ eine Beschleunigung zwischen 1,484 und 2,688 bei einem einzigen POSIX-Thread. Damit ist gezeigt, dass die Datenanordnung beachtet und sorgfältig gewählt werden muss. Dieser Schritt kann nicht von der Implementierung abgenommen werden, sondern muss vom Anwendungsentwickler beachtet werden. Es ist denkbar, eine Metakomponente bestehend aus Transposition der rechtsseitigen Matrix und anschließender Multiplikation bereitzustellen, welche automatisch vom in Abschnitt 7.4 beschriebenen Laufzeitsystem zur Verwendung in Erwägung gezogen wird.

Freiwilliges Überlassen der Prozessorressource. Bei der datengetriebenen Ausführung sollen die aufgerufenen Komponenten ihre Berechnungen alsbald möglich beginnen. Daher ist einerseits zu betrachten, wie die Komponenten möglichst frühzeitig ihre Berechnungen fortsetzen können, wenn zwischendurch die aktuell benötigten Daten noch nicht verfügbar sind, beispielsweise die nächste Zeile einer Matrix. Andererseits muss überlegt und untersucht werden, wie andere Komponenten von der Wartezeit profitieren können. Es gibt insgesamt vier Varianten, wie auf die Verfügbarkeit der nächsten Daten gewartet werden kann.

1. Geschäftiges Warten auf die Eingabedaten, wodurch die Prozessorressource aber unnötig blockiert wird und davon abgehalten wird, sinnvollere Berechnungen von anderen Komponenten durchzuführen.

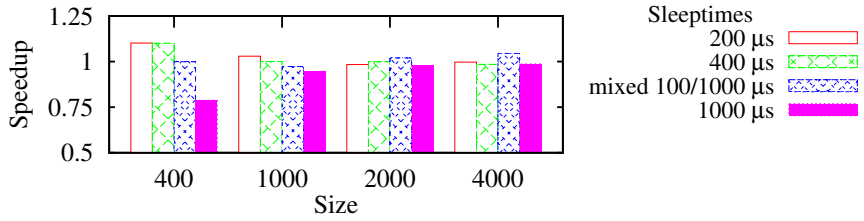
2. Überlassen der Prozessorressource an eine andere Komponente mittels `yield` ist die beste bekannte Lösung. Bei der Implementierung und ersten Evaluationen hat sich jedoch herausgestellt, dass nicht zwangsweise eine andere Komponente die Prozessorressource auch tatsächlich erhält. Ferner ist `yield` nicht möglich mit OpenMP-Threads.

3. Synchronisation mittels Bedingungsvariablen[4] erlaubt, die Ausführung erst dann fortzusetzen, wenn die Daten auch verfügbar sind, sonst aber den Thread einfach zu blockieren. Das Problem bei Bedingungsvariablen ist allerdings, dass die empfohlene Granularität wesentlich höher ist, für ganze Strukturen, also beispielsweise Matrizen anstelle von einzelnen Daten oder Zeilen. Ferner ist auch dieser Ansatz mit OpenMP-Threads nicht möglich.

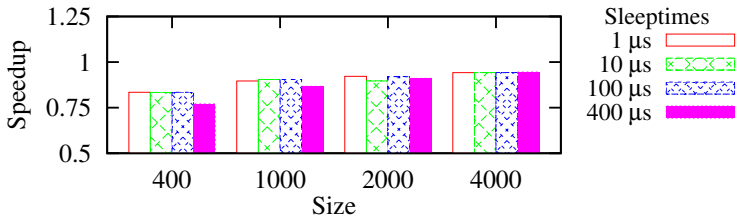
4. Ein Thread kann schlafen gelegt werden für eine bestimmte Zeitspanne.

Einen Thread nur eine Mikrosekunde schlafenzulegen, entspricht nahezu einem `yield`. Auch 100 Mikrosekunden reichen noch nicht aus, dass ein anderer Prozess eine ausreichende Menge an Daten verarbeiten würde und dadurch erkennbaren Fortschritt bei der Programmverarbeitung erzielen würde. In Abbildung 8.3(a) zeigen sich die Unterschiede im maximalen Speedup gegenüber sequentieller Ausführung für unterschiedliche Matrixdimensionen und unterschiedlich lange Zeitspannen

[4] engl. condition variables



(a) Schlafenlegen bei Tasks (POSIX-Threads).



(b) Schlafenlegen bei komponenteninternen Threads (OpenMP-Threads).

Abbildung 8.3: Einfluss unterschiedlich langer Zeitspannen für das Schlafenlegen von Threads, wenn auf die nächsten Daten gewartet werden muss.

beim Schlafenlegen. Bei „mixed 100/1000 μs“ wartet die Haupt-Task mindestens 1000 μs, z.B. bei Barrieren, während die Komponenten intern nur 100 μs warten müssen.

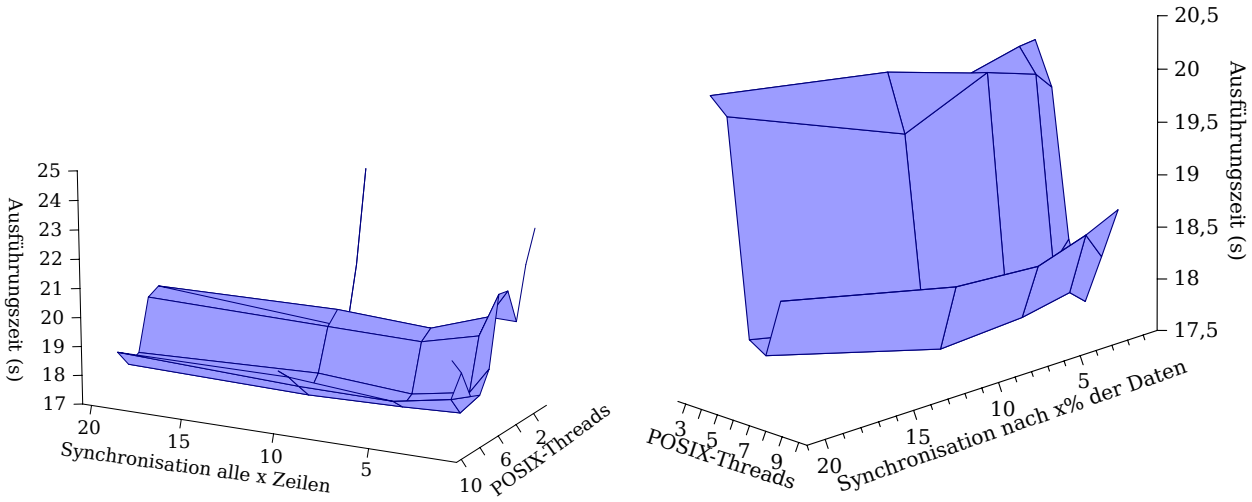
Eine Schlafdauer von 1000 Mikrosekunden ist erkennbarerweise bereits zu viel, um noch kontinuierliche Datenverarbeitung zu gewährleisten. Eine sinnvolle Eingrenzung der nötigen Schlafdauer kann in Abhängigkeit der Matrixdimension (dim) für das verwendete System angegeben werden mit

$$\text{dim}/5 <= \text{Schlafensdauer in Mikrosekunden} <= \text{dim}/2.$$

Bei dieser groben heuristischen Einschätzung geht man davon aus, dass innerhalb einer fünftel bis halben Mikrosekunde ein Datum berechnet werde und innerhalb der Schlafdauer eine ganze Zeile im Falle von Matrizen. Da die Ausnutzung von Cacheeffekten jedoch wesentlich ist und bei zu großen Zeitspannen nicht mehr erreicht wird, haben auch die Prozessortaktrate und die Cachegröße eine wesentliche Rolle; sie sind bei der obigen einfachen Heuristik allerdings nicht berücksichtigt worden. Bei sehr großen Datenmengen hat es sich als vorteilhaft herausgestellt, unterschiedliche Zeitspannen zu verwenden, je nachdem, ob auf Anwendungsebene oder Funktionsebene gewartet werden soll. Da dieser Ansatz allerdings zu aufwendig und problemabhängig scheint, ist die Entscheidung zunächst auf eine Schlafdauer von 400 Mikrosekunden gefällt worden, wenn die Haupt-Task oder Komponenten auf Daten warten müssen.

Während es also geringfügig Nutzen bringt, die POSIX-Threads der Komponenten schlafenzulegen, zeigt Abb. 8.3(b) klar, dass sich dies für die komponenteninterne Parallelisierung mit OpenMP-Threads überhaupt nicht lohnt. Für OpenMP-Threads sollte daher die erste o.g. Option verwendet werden, dass geschäftig auf neue Daten gewartet wird. Dabei ist die Idee allerdings auch, dass die gesamte Anzahl ausgeführter Threads durch die Komponenten und deren OpenMP-Threads zu einem Zeitpunkt die insgesamt verfügbare Anzahl an Hardware-Threads nicht übersteigt.

Wie für POSIX-Threads festgestellt wurde, sind die Schlafdauern problemspezifisch zu setzen, hängen von den Cache-Größen und der Prozessorfrequenz ab und sind daher nicht portabel. Die Alternative, Bedingungsvariablen einzusetzen, hat zwar bekanntermaßen hohen Zusatzaufwand und benötigt zudem etwas Platz in den Caches, ist aber portabel. Der Zusatzaufwand lässt sich reduzieren, indem gerade bei sehr großen Datenmengen weder auf Datums- noch auf Zeilengranularität kommuniziert wird, sondern auf Blockgröße. Daher ist zu untersuchen, was eine geeignete Blockgröße ist. Bei matrixbasierten Operationen ist die kleinste Blockgröße nach dem hier vorgeschlagenen Ansatz mit bereits erhöhter Granularität die Zeilen- bzw. Spaltenebene. Demgegenüber wurde die Synchronisation nur alle 2, 5, 10 und 20 Zeilen ausgeführt. Die Ergebnisse der Evaluation für das Lösen von 1000×1000-Problemen mittels des CG-Verfahrens sind in Abb. 8.4(a) für unterschiedlich viele POSIX-Threads dargestellt, wobei jeweils die beste Anzahl OpenMP-Threads verwendet wurde. Da die Ergebnisse kaum Unterschied bei der Synchronisationshäufigkeit zeigen, wurde die



(a) Synchronisation alle x Zeilen.

(b) Synchronisation nach x Prozent der Daten.

Abbildung 8.4: Unterschiedliche Blockgrößen bei der Synchronisation der Daten zwischen unterschiedlich vielen POSIX-Threads, jeweils optimale Anzahl OpenMP-Threads gewählt.

Synchronisation noch gröber gehalten auf 1 %, 2 %, 5 %, 10 % und 20 %. Wie Abbildung 8.4(b) zeigt, lässt sich bei 10-prozentiger Synchronisation am meisten Leistung erzielen.

Anzahl Threads. Der interne Task-Parallelismus wird über einen Satz an POSIX-Threads umgesetzt, wobei jeder Pthread intern weitere Threads zur gemeinsamen datenparallelen Bearbeitung seines Teilproblems verwenden darf. In Abhängigkeit der Anzahl verfügbarer Kerne und der Arbeitsmenge können unterschiedliche viele Tasks bzw. Threads sinnvoll sein.

Weitergehend wurde untersucht, welche Kombination aus POSIX-Threads und OpenMP-Threads am Vielversprechendsten ist. Wie sich Abb. 8.5 entnehmen lässt, ist die Kombination von sieben POSIX-Threads bei Verwendung von nur drei OpenMP-Threads am Besten auf dem verwendeten Zwei-Prozessorsystem mit 24 Hardwarethreads. Aufgrund der geringen Anzahl an Basisblöcken, bis beim CG-Verfahren aufgrund einer Datenabhängigkeit alle Komponenten synchronisiert werden, nutzen mehr POSIX-Threads nicht. Mit mehr OpenMP-Threads tritt bereits Sättigung der Speicherschnittstelle ein. Wie bereits angedeutet, ist es also nicht sinnvoll, die Anzahl an verfügbaren Hardwarethreads zu überschreiten. Aufgrund dieser Erkenntnis ist der Suchraum für das Laufzeitsystem, die beste Kombination aus POSIX- und OpenMP-Threads zur Laufzeit zu ermitteln, eingeschränkt auf möglichst viele Pthreads p und OpenMP-Threads o, so dass gilt $p \cdot o \leq \#$ Hardwarethreads. Für das verwendete Zweiprozessorsystem kommen somit nur die Kombinationen {6,3}, {6,4}, {7,3}, {8, 2} und {8,3} in Frage, wie Abb. 8.5 auch bestätigt.

8.1.5 Datengetriebene Integration von Koprozessoren

Das datengetriebene Programmiermodell ist insbesondere dazu gedacht, das datengetriebene Rahmenwerk auch nahtlos in Software fortzusetzen. Dies aus zweierlei Gründen: Die Ausführung in Software bietet die Möglichkeit, als Anwender die korrekte Umsetzung der Anwendung oder des Algorithmus sicherzustellen. Somit dient das Programmiermodell der Validierung und ggf. der Fehlersuche. Andererseits sollte der Koprozessor die Daten in Streaming-Manier aufnehmen und verarbeiten. Insbesondere beim Konzept zur exakten Matrixmultiplikation mittels eines Koprozessors in rekonfigurierbarer Hardware sind die Datenübertragung und die Anordnung der Daten im Speicher von immanenter Wichtigkeit.

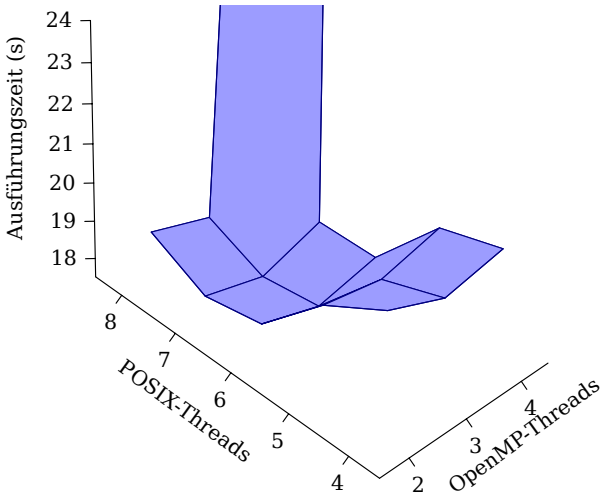


Abbildung 8.5: Ermittlung der geeignetesten Kombination von OpenMP-Threads und POSIX-Threads bei Synchronisation in Blöcken von je 10 % der Daten.

Anbindung des Koprozessors der Convey HC-1. Der Koprozessorspeicher der Convey HC-1 ist mit dem Hauptspeicher kohärent gehalten durch Ausnutzung des Cache-Kohärenzprotokolls. Dabei hat der Koprozessorspeicher nur dann den aktuellen Wert einer Speicheradresse, wenn wie bei einem Cache auch Speicherzugriffsanfragen an die betreffende Adresse bereits gestellt wurden. Um die Daten direkt aus der Anwendung heraus in den Koprozessorspeicher zu bringen, stellt Convey eigene Systemfunktionen bereit. Diese sind erst ab großen Datenmengen sinnvoll einzusetzen, während bei kleinen Datenmengen der Koprozessor das Einladen der neuen Daten besser selbst vornimmt. Insgesamt ist die Übertragsungsrate am höchsten bei Verwendung der speziellen Systemfunktionen. Beim Koprozessor der Convey HC-1 ist die Granularität also weiter zu erhöhen auf das Kopieren einer gesamten Matrix als Block. Demzufolge muss eventuell an der Stelle des Koprozessoraufrufs die datengetriebene Verarbeitung insofern unterbrochen werden, als dass unter Verwendung der bereitgestellten Mittel auf die vollständige Berechnung der Eingabedaten gewartet wird.

Die Bekanntgabe der Gültigkeit der Eingangsdaten erfolgt bei der Implementierung in Software über atomare Speicherzugriffe. Speicherzugriffe auf den Koprozessor hätten unnötiges Aus- und Einladen der betreffenden virtuellen Speicherseite zur Folge. Daher wird die Verfügbarkeit der Eingangsdaten auch nicht über den Speicher bekanntgegeben. Eine andere Möglichkeit stellt allerdings die Registerschnittstelle des Koprozessors bzw. des AE Hubs dar. Diese ist allerdings nur vor dem Aufruf des mikroprogrammierbaren Rahmenwerks sowie nach Vollendung der Ausführung verwendbar. Während der Ausführung des Rahmenwerks kann die aufrufende Funktion zwar fortgesetzt werden durch einen asynchronen anstelle eines synchronen Aufruf des Koprozessors, muss allerdings auf die vollständige Ausführung warten, bevor wieder mit dem AE Hub kommuniziert werden kann. Aus diesen Gründen wurde das datengetriebene Programmiermodell nicht weiter gezielt für numerische Anwendungen an die Convey HC-1 angeschlossen. Es bleibt jedoch, dass die auf dem Koprozessor berechneten Teilergebnisse als Eingaben für die weitere datengetriebene Verarbeitung davon abhängiger Funktionalitäten auch datengetrieben genutzt werden können. Beispielsweise können bei Absenkung der Granularität auf einzelne Elemente im Rahmen von Matrix-Vektor-Multiplikationen die Ergebnisse von exakten Skalarproduktberechnungen bereits weitergenutzt werden.

Die Convey HC-1 kann also aufgrund ihres Speichermodells zwar nicht nahtlos datengetrieben programmiert und genutzt werden, es bestehen aber durchaus einige Möglichkeiten, auch den Koprozessor der Convey HC-1 in die datengetriebene Programmausführung einzubinden.

8.1.6 Zusammenfassung der datengetriebenen, taskparallelen Programmierung

Es kann vorteilhaft sein, insbesondere bei numerischen Anwendungen, die Funktionsimplementierungen als zueinander nebenläufige Tasks umzusetzen und auszuführen. Aufgrund der Datenwiederverwendung und damit einhergehenden Entlastung der Speicherschnittstelle wird dadurch die

Ausführungszeit der Anwendung reduziert. Dazu müssen die Bibliotheksentwickler sich um die Einhaltung der Aufrufreihenfolge, geeignete Datenstrukturen zum Datenaustausch und zur Synchronisation sowie um die beste Parametrierung der Anzahl an Tasks und task-interner Threads bemühen. Dass die Reihenfolge der Aufrufe beibehalten wird, wird durch das Eintragen der aufgerufenen Komponenten in eine Warteschlange sichergestellt, die im Gegensatz zu StarSS [218] nicht bei Verfügbarkeit der Operanden einem freien POSIX-Worker-Thread zugewiesen werden, sondern allein aufgrund der Position in der Schlange. Die Task- und Threadzahl kann von Laufzeitsystemen, wie sie unter anderem in Kapitel 7 verwendet wurden, automatisiert ermittelt und für spätere Aufrufe gespeichert werden. Anwendungsentwickler hingegen müssen lediglich für die korrekte Reihenfolge der Komponentenaufrufe auf Anwendungsebene sorgen, wie sie es vom sequentiellen Programmiermodell gewohnt sind, ohne weitere Besonderheiten zu beachten. Wichtige Aspekte bei der Implementierung einer datengetriebenen Bibliothek und Laufzeitumgebung mit Fokus auf numerische Anwendungen wurden beschrieben, untersucht und für das verwendete Zweisockelsystem mit einem Intel Xeon X5670 bestimmt mit lediglich 10 % Synchronisation mittels Bedingungsvariablen, mit geschäftigem Warten bei der internen datenparallelen Verarbeitung und mit einer Kombination aus sieben POSIX-Threads für sieben zeitgleiche Tasks, welche jeweils zwei weitere OpenMP-Threads starten dürfen. Für andere Systeme müssen diese Parameter gegebenenfalls neu ermittelt werden, wobei besagtes Laufzeitsystem zu unterstützen vermag. Eine geeignete Vorschrift, nach der Operationen wie Allokationen, Datenfreigaben und Initialisierungen von Programmierwerkzeugen automatisch oder auch manuell an den bestgeeigneten Stellen im Programmtext hinsichtlich der Aufrufreihenfolge platziert werden, konnte ebenso gefunden werden: frühzeitige Allokation, zu der nebenläufig bereits die Initialisierung auf freien Rechenressourcen erfolgen kann, und späte Datenfreigabe, damit unnötiges Warten und Synchronisation vermieden werden.

Die Verwendung der datengetriebenen Programmierung und Programmausführung wird für Systeme mit FPGAs als Akzeleratoren vorgeschlagen. Indem die Ausführung auf dem Host ebenso wie in Hardware datengetrieben erfolgt und dadurch der Koprozessor nahtlos eingebunden wird, können die zur Ansteuerung des mikroprogrammierbaren Rahmenwerks benötigten Mikroprogramme bereits vorab emuliert und validiert werden und sukzessive größere Anwendungsteile in Hardware verlagert werden.

8.2 Eine grafische Schnittstelle zur datengetriebenen, attributgestützten Programmierung heterogener Systeme

Grafische Schnittstellen, die Kontrollflussgraphen ausgeben, welche mittels Laufzeitsystemen oder Bibliotheken ausgeführt werden können, oder ausführbereiten Programmcode erzeugen, gibt es bereits seit Beginn der Forschung zu Datenflusssystemen.

Davis hat 1974 mit Data-Driven Nets (DDNs) [76] die erste grafische Datenflussprogrammierung entwickelt und im Jahre 1981 mit der Graphical Programming Language (GPL) auch eine programmiererfreundliche Version nach dem Token-Modell herausgebracht. Im Gegensatz dazu ist die ebenso 1981 veröffentlichte Function Graph Language (FGL) strukturorientiert. Als grafische Eingabemöglichkeit für die textuelle Sprache Cajole wurde 1982 Grunch veröffentlicht.

Grafische Programmierung hat sich seitdem in der Industrie dort weit verbreitet [160, 70], wo die Domäne dies besonders gut ermöglicht, wie etwa in der Signalverarbeitung, und wo die Anwender und Entwickler keine ausgebildeten Programmierer sind oder interdisziplinär zusammenarbeiten müssen. Ein bekanntes Beispiel dafür ist National Instruments LabView[5]. Mit dem grobgranularen Modell und z.B. der Sprache Granular Lucid (GLU) [154] lässt sich grafische Datenflussprogrammierung auch als Mittel zur Koordination von nebenläufig ausführbaren Funktionen einsetzen. AMD hat 2010 in diesem Kontext den AMD PipelineKIT geschaffen, der später zum OpenSource-Projekt wurde[6]. Sehr ähnlich ist Apple Quartz Composer[7][8], welches wiederum auf PixelShox[9] basiert. Gra-

[5] URL: http://www.ni.com/labview/
[6] URL: http://code.google.com/p/pipelinekit/
[7] URL: https://developer.apple.com/technologies/mac/graphics-and-animation.html
[8] URL: https://developer.apple.com/library/mac/documentation/GraphicsImaging/Conceptual/QuartzComposerUserGuide/qc_intro/qc_intro.html
[9] URL: http://www.polhosting.info/web-archives/pixelshox_technology/

fische Programmierung von Kontroll-Datenflussgraphen ist derart eingängig, dass LEGO mit seinem Werkzeug MINDSTORMS NXT die Programmierung von Robotern auch Kindern ermöglicht. Wichtig ist dabei stets die leistungsfähige Implementierung der vorliegenden Komponenten oder Funktionen.

Um die Komponenten für einen vorkonfigurierten CGRA zu ändern, bietet $(\text{GECO})^2$ eine grafische Schnittstelle an [217]. Die Komponenten werden bei $(\text{GECO})^2$ mittels Mikroprogrammen ausgetauscht, aber die Verdrahtung bleibt zur Laufzeit erhalten. Die Frage ist daher nicht, ob es möglich ist, Programmcode aus einer grafischen Spezifikation zu erzeugen, sondern vielmehr, wie domänenspezifische Konfigurationen gehandhabt werden, wie Attribute eingebracht werden können und wie heterogene Systeme gezielt unterstützt werden können. Der Entwurf ist in Abbildung 8.6 skizziert. Die angedachte Vorgehensweise ist, dass über Konfigurationsdateien die domänenspezifischen Komponenten bereitgestellt werden. Ein codeerzeugendes Backend kann dann Mikroprogramme für das mikroprogrammierbare Koprozessorrahmenwerk erzeugen oder die Softwareimplementierungen der Komponenten aus einem gewöhnlichen C-Programm heraus aufrufen. Die Attribute für Anforderungen kann der Anwender auf Knotenebene angeben; auf Implementierungsseite sind sie bereits in der Konfigurationsdatei bzw. auch in der Bibliothek selbst eingetragen. Um aus der Softwarevariante heraus ein Mikroprogramm zur Ausführung eines Subgraphen auf dem Koprozessor aufzurufen, kann eine Methode zur Kommunikation mit dem Koprozessor in einer Konfigurationsdatei eingetragen werden und als Bibliotheksbestandteil mitgeliefert werden. Die Entscheidung, ob eine echte Implementierung in Software aufgerufen wird oder anstelledessen indirekt das Mikroprogramm, kann wie in Abschnitt 7.4 ein Laufzeitsystem vornehmen und anhand von Attributen wie `prefer_hardware` zusätzlich vom Anwender gelenkt werden.

8.3 Ergebnis und Zusammenfassung

In diesem Kapitel wurde anhand eines datengetriebenen Programmiermodells vorgestellt, wie die nahtlose Integration von Beschleunigern in Mehrkernsysteme erreichbar ist, indem die Kommunikation zwischen Komponenten überlappt mit Berechnungen erfolgt und die vorhandenen Rechenkerne allesamt in die Programmausführung eingebunden werden. Koprozessoren werden dazu schlichtweg als Komponenten in die Programmierung und Ausführung eingebunden. Um unterschiedliche Implementierungen auszuzeichnen und auch um bestimmte Anforderungen an die Eigenschaften einer Komponente, die ja stets zunächst als Funktion aufgerufen wird, anzugeben, wurde das Konzept der Attributierung aus Kapitel 7 eingebracht. Grafische Schnittstellen ermöglichen schließlich die bequeme Verbindung von einzelnen Komponenten zu einem größeren Ganzen und das Versehen der Komponentenknoten mit Anforderungen. Sowohl die grafische Programmierung als auch die Ausführung der Anwendung zunächst auf Basis von Softwareimplementierungen der Komponenten helfen, zügig eine korrekte Implementierung der Anwendung zu erhalten. Dabei kann über die task-parallele Ausführung der Komponenten in Software bereits Leistungssteigerung gegenüber sequentieller Ausführung der Komponenten erhalten werden, die zudem vergleichbar der mit datenparalleler Ausführung erzielten Steigerung sein kann. Beschleunigenswerte Teile sollen dann in Hardware verlagert werden. Dem Anwendungsentwickler kann die Partitionierung einer Anwendung in Software- und Hardware-Teile selbst lenken qua Spezifikation bestimmter Anforderungen mittels Attributen. Für einen in Hardware auszuführenden Teil entwickelt er ein Mikroprogramm, das von einer Software-Funktion übertragen und aufgerufen wird. Dieses Mikroprogramm implementiert einen Subgraphen des Anwendungsgraphen und wird mit der Software-Funktion zusammen als datengetrieben ausgeführte Komponente in die Gesamtanwendung eingebunden. Ein immer größerer Teil der Anwendung kann dann sukzessive in die Koprozessorhardware verlagert werden, bis keine Leistungssteigerung mehr erzielbar ist. Dabei kann das beschriebene Laufzeitsystem helfen, wenn es die Auswahl zwischen unterschiedlichen Funktionsimplementierungen in Hardware oder Software selbst treffen kann und dann die Laufzeiten vermisst, um spätere Fehlentscheidungen zu vermeiden. So lässt sich auf bequeme und zudem automatisierbare Weise eine geeignete Hardware-Software-Partitionierung finden.

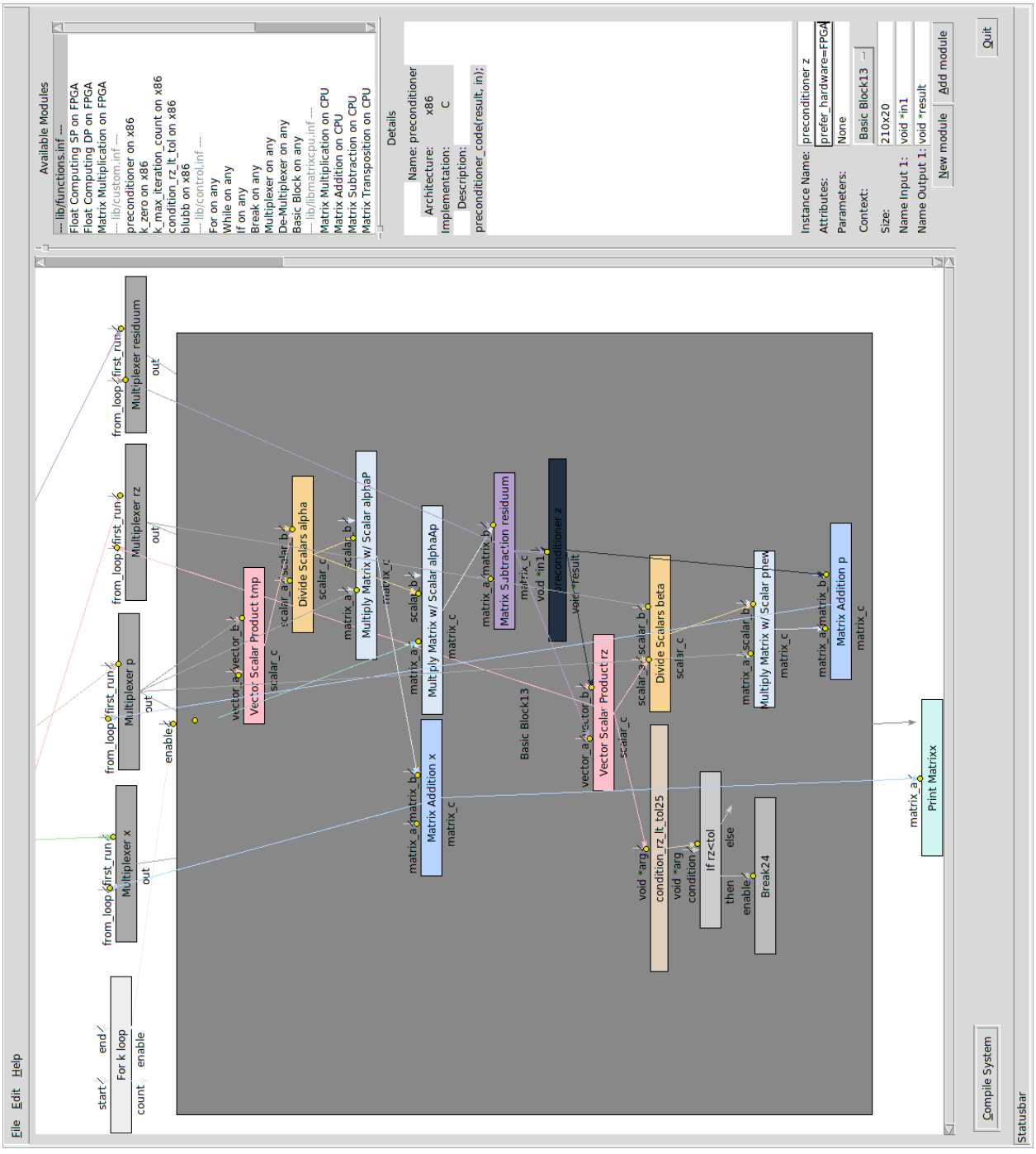


Abbildung 8.6: CG-Verfahren in grafischer Schnittstelle zur datengetriebenen Programmierung heterogener Systeme unter Verwendung von Attributen.

Kapitel 9

Evaluation

Die Evaluation der vorgestellten und der bislang nur im Rahmen der Implementierung teilevaluierten Konzepte der komponentenbasierten und datengetriebenen Ausführung von Anwendungen in Mehrkernsystemen mit rekonfigurierbarer Hardware wird in diesem Kapitel vorgestellt. Sie beginnt in Abschnitt 9.1 mit dem Vergleich einer manuellen Implementierung eines Hardwaredesigns für einen Vorkonditionierer, die durch einen Anwendungsentwickler mit einer Hochsprache erfolgt ist, mit einer Komponente im mikroprogrammierbaren Rahmenwerk. Aufgrund des in der vorliegenden Dissertation vorherrschenden Hintergrunds der Verlagerung numerischer Anwendungen in rekonfigurierbare Hardware wird das Rahmenwerk anhand des bereits mehrfach verwendeten Verfahrens der konjugierten Gradienten zum Lösen eines Gleichungssystems evaluiert in Abschnitt 9.2. Dabei kommt der bereits erwähnte Vorkonditionierer zum Einsatz. Die datengetriebene Ausführung von Anwendungen in Software in Mehrkernsystemen wird in Abschnitt 9.3 ebenso anhand des CG-Verfahrens evaluiert. Damit schließt sich auch der Kreis der datengetriebenen Ausführung und Ansteuerung von Komponenten sowohl in Software als auch in Hardware und der Verlagerung von beliebig grobgranularen Teilgraphen der Anwendung in die rekonfigurierbare Hardware – ein einzelner Vorkonditionierer bis hin zu einem gesamten Verfahren. Betrachtet man die erste Vorfilterungsstufe bei HHblits als Komponente, so lässt sich das gesamte Konzept auch auf den Bereich bioinformatischer Anwendungen übertragen. Diese Komponente wird zwar auf der Convey HC-1 bei Ausführung in rekonfigurierbarer Hardware nicht datengetrieben angesteuert bzw. ausgeführt, da die Datenübertragung so wesentlich schneller durchgeführt werden kann und zur Laufzeit keine Kommunikation mit dem Host unterhalb der Granularität einer Speicherseite möglich ist, wohl aber dienen ihre Berechnungen als Eingaben für die datengetriebene Ausführung der zweiten Vorfilterungsstufe als weiterer Komponente. Wie die Evaluation in Abschnitt 9.4 zeigt, lässt sich dadurch ebenfalls ein nicht unerheblicher Teil an Ausführungszeit einsparen.

9.1 Vergleich der mikroprogrammierbaren Steuerung gegen Impulse C auf der DRC AC2030

In Abschnitt 5.8.4 wurde bereits die Untersuchung vorgestellt, wie mit Impulse C für rekonfigurierbare Hardware SSOR- bzw. SGS-Vorkonditionierer erstellt weden können. Dabei wurde nach eingehender Analyse Folgendes festgestellt: Domänenexperten können ohne tiefgehende Schulung in der Hardwareentwicklung zwar ein funktionsfähiges Design erstellen; das entwickelte Design wird mindestens 20 mal langsamer sein; das Design ist tatsächlich 13 mal langsamer als erwartet und damit insgesamt 260 mal langsamer; mit verbesserter Implementierung unter Zuhilfenahme von Hardwarekenntnissen könnte man eine Verbesserung von Faktor 118 erzielen, damit aber noch immer eine Verlangsamung um mehr als Faktor 2.

Die erzielten Ergebnisse sollen verglichen werden mit der Ausführung eines Mikroprogramms, das eine von einem Hardwarespezialisten bereitgestellte Komponente aufruft. Der Hardwarespezialist hat auf Basis der Erläuterungen in Abschnitt 5.8.3 einen Vorkonditionierer implementiert. Die Implementierung wird im Folgenden genauer beschrieben.

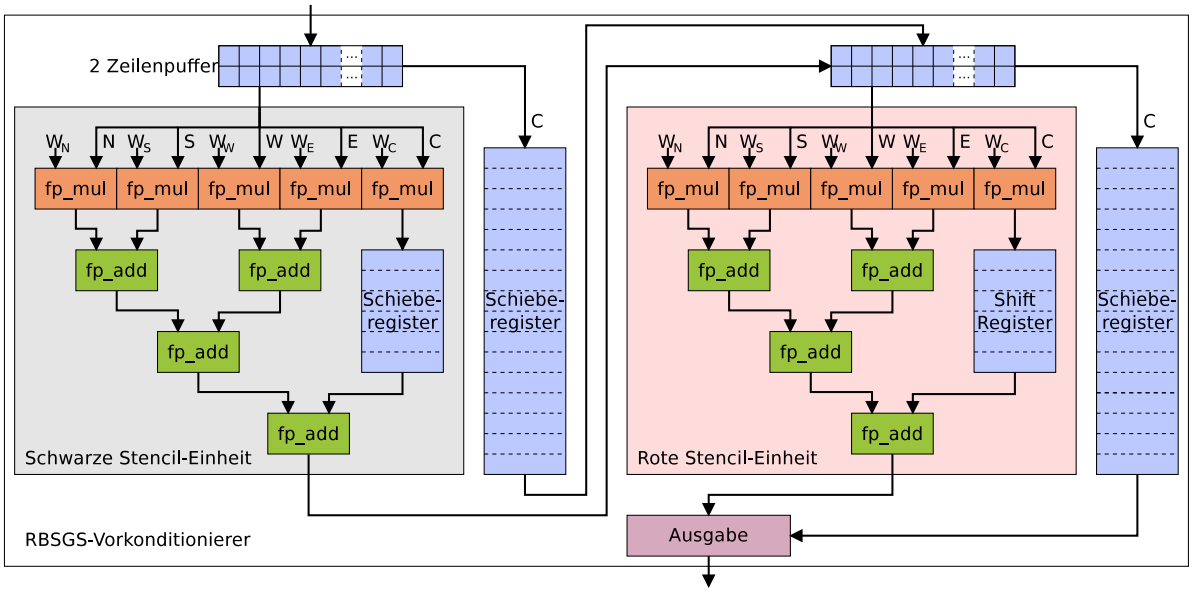


Abbildung 9.1: RBSGS-Vorkonditionierer als Komponente im datengetriebenen, mikroprogrammierbaren Rahmenwerk.

9.1.1 Implementierung eines Rot-Schwarz-Vorkonditionierers für die Poisson-Gleichung

Die in Abb. 9.1 dargestellten Stencil-Einheiten des Vorkonditionierers verarbeiten die eingehenden Daten auf datengetriebene Weise. Beide sind intern vollständig gepipelinet. Durch vorab erfolgte Berechnung der Koeffizienten ist keine Division mehr nötig. Somit können in jedem Takt zwei neue Stencil-Werte berechnet werden. Die Länge einer Stencil-Pipeline beträgt zwei mal die halbe Zeilenbreite n (die Schnittstelle hat 64 Bits, die Ausführung erfolgt jedoch auf 32-Bit-Daten) plus die Pipelinelänge der Multiplizierer von drei Takten plus drei mal die Pipelinelänge der Gleitkommaaddierer von je vier Takten, also insgesamt $n + 15$ Takte. Die Pipeline der gesamten Vorkonditioniereinheit hat somit eine Pipelinelänge von $p(n) = 2n + 30$ Takten. Für m Zeilen einer Matrix ist die Ausführungszeit (in Takten) $t(m, n) = m \cdot n/2 + n + 30 - 1$, die durchgängige Ankunft gültiger Daten aus dem Puffersatz vorausgesetzt. Also ergibt sich für $m = n = 1000$ eine Ausführungszeit von $t(m, n) = 1000 * 500 + 1000 + 29 = 501.029$ Takten. Die Implementierung mit Impulse C auf der DRC AC2030 ergab eine erwartete Ausführungszeit von $0,292\,s$ für eine Iteration bei 100 MHz, also 29.200.000 Takte. Gemessen wurden allerdings sogar $3,88\,s$ (etwa 388.056.600 Takte) für die Impulse-C-Implementierung.

9.1.2 Verwendung des Koprozessors

Die Vorkonditionierung wurde mittels des datengetriebenen, mikroprogrammierbaren Rahmenwerks auf den Koprozessor verlagert. Dazu steht das Mikroprogramm aus Listing 9.1 bereit, welches Daten aus dem Koprozessorspeicher mittels einer DMA-Einheit einliest, die passenden Zielpuffer für die Vorkonditioniereinheit zuweist, selbige Einheit startet und deren Ergebnisse in den Koprozessorspeicher zurückschreibt. Ferner erhält der Koprozessor die Adressen der Ein- und Ausgabematrizen sowie weitere Parameter über die zu verarbeitenden Daten mittels des Application Engine Hubs der Convey HC-1. Diese Parameterübergabe benötigt bei Convey ein weiteres Assemblerprogramm (Listing 9.2). Mit dem Ausführen jenes zweiten Assemblerprogramms ruft der Anwender den Koprozessor und damit das Rahmenwerk mit dem ersten Mikroprogramm auf. Beide Programme sind weitestgehend wiederverwendbar und einfach zu erweitern.

Iterationen	Auflösung		
	10×10	100×100	1000×1000
10	0,438	0,419	1,24
100	–	0,463	7,95
1000	–	–	74,180

Tabelle 9.1: Durchführungszeit (in Sekunden) unterschiedlich vieler CG-Iterationen für unterschiedliche Problemgrößen auf der Convey HC-1 mit auf den Koprozessor ausgelagertem rot-schwarzen Gauß-Seidel-Vorkonditionierer.

```
buf_assign U1 B24
dma1r R2 R10 0
buf_assign U14 B2
rbsgs1 R9 R9
dma2w R3 R10
j 0
```

Listing 9.1: Mikroprogramm für den rot-schwarzen Gauß-Seidel-Vorkonditionierer.

```
rbsgs:
mov %a8, $0, %aeg # move a8 to R0 (&program)
mov %a9, $1, %aeg # move a9 to R1 (programLength)
mov %a10, $7, %aeg # move a10 to R7 (&null)
mov %a11, $8, %aeg # move a10 to R8 (&one)
mov %a12, $31, %aeg # move a10 to R31 (&default_return_val)
mov %a13, $9, %aeg # move a10 to R9 (&cols)
mov %a14, $10, %aeg # move a10 to R10 (&vec_size)
mov %a15, $2, %aeg # move a10 to R2 (&in_vec)
mov %a16, $3, %aeg # move a11 to R3 (&out_vec)
mov $1, %aeem
caep00 $0
mov.ae0 %aeg, $31, %a8
rtn
```

Listing 9.2: Assemblerprogramm zur Ansteuerung des Koprozessors.

9.1.3 Laufzeitmessung

Die Ergebnisse der Laufzeitmessung des gesamten vorkonditionierten CG-Verfahrens auf der Convey HC-1 mit ausgelagertem Vorkonditionierer sind in Tabelle 9.1 zusammengestellt. Sie sind zu vergleichen gegenüber den Ausführungszeiten bei einem mit Impulse C formulierten Vorkonditionierer auf der DRC AC2030 (Tabelle 5.24). Da die Prozessoren der verwendeten Systeme einigermaßen vergleichbare Leistung liefern, stellt der vorgeschlagene Ansatz der komponentenbasierten, mikoprogrammierten Koprozessorverwendung eine wesentliche Effizienzsteigerung dar. Denn anstelle 260-facher Verlangsamung des Vorkonditionierers tritt nur eine Verlangsamung von etwa 4-5 bei der Verlagerung ein, wie die Untersuchung der Ausführungsgeschwindigkeit des Vorkonditionierers in Tabelle 9.2 im Vergleich zu Tabelle 5.22 zeigt, und der Aufwand für den Anwender beschränkt sich auf das Erstellen eines Mikroprogramms und Übergeben als Parameter beim Koprozessoraufruf. Für einzelne Komponentenaufrufe kann die Mikroprogrammerstellung sogar im Vorfeld vom Hardwareentwickler abgenommen werden und ausschließlich eine bibliotheksbasierte Schnittstelle zur Verfügung gestellt werden. Dabei kommt allerdings keine weitere Komponente zeitgleich zum Einsatz, welche die Datenlokalität ausnutzen würde. Das Rahmenwerk ist mit nur einer Vorkonditioniereinheit konfiguriert, könnte aber mit den vier vorhandenen DMA-Einheiten mindestens zwei unterschiedliche Datensätze gleichzeitig verarbeiten, wenn ein zweiter Vorkonditionierer geladen würde. Ebenso wird nur eine der vier Application Engines verwendet. Unter idealen Bedingungen sollte sich daher die Laufzeit des Vorkonditionieres vierteln lassen, wenn alle vier AEs eingesetzt werden, und damit eine messbare Beschleunigung einstellen anstelle der Verlangsamung beim Ansatz mit Impulse C.

Gegenüber der Nutzung einer Hochsprache ist der hier vorgeschlagene Ansatz mit domänenspezifischen, von Hardwarespezialisten entwickelten Komponenten und der Ansteuerung über Mikroprogramme bereits jetzt mehr als $\frac{260}{4,5} \approx 57,8$ mal effizienter in Relation zur eingetretenen Verlangsamung auf dem jeweiligen Host (der zeitliche Aufwand ist für den Anwendungsentwickler aufgrund der Bibliotheksaufrufe einer vorliegenden Implementierung geringer als die zeitaufwendige Synthese eines vorab selbst zu implementierenden und anzusteuernden Algorithmus). Zudem ist der Koprozessor $\frac{61,92\,s}{0,49\,s} \approx 125,5$ mal schneller.

Auflösung	Zeit auf CPU	Zeit auf FPGA	Verhältnis FPGA zu CPU
500×500	0,00137600	0,00737300	5,35828488
1000×1000	0,00687400	0,03055400	4,44486471
2000×2000	0,02932000	0,12359500	4,21538199
4000×4000	0,11972300	0,49347800	4,12183123

Tabelle 9.2: Laufzeit eines rot-schwarzen Gauß-Seidel-Vorkonditionierers in Software auf der Intel Xeon CPU der Convey HC-1 gegenüber Ausführung auf FPGA mit 150 MHz, gemittelt über zehn Aufrufe (jeweils in Sekunden).

9.1.4 Zusammenfassung der Verlagerung des Vorkonditionierers auf den Koprozessor

Einerseits konnte gezeigt werden, dass von Hardwarespezialisten entwickelte Komponenten wesentlich schneller sind, und andererseits, dass zwar etwas Effizienzverlust durch die Ansteuerung des Koprozessorrahmenwerks und die Datenübertragung gegenüber einem direkten Aufruf eintritt, aber dennoch gegenüber der vormals 260-fachen Verlangsamung nur noch vier- bis fünffache Verlangsamung eintritt bei der Nutzung einer einzelnen Stencileinheit. Auf jeder der vier Application Engines der Convey HC-1 kann ein eigenes Mikroprogramm ausgeführt werden, außerdem können mehrere Stencileinheiten untergebracht werden. Ausgehend von unbeschränktem Platz und Erreichen der Taktrate könnten mit den 16 Speicherports des Koprozessors acht Stencileinheiten mit Daten versorgt werden, so dass insgesamt knapp achtfache Beschleunigung auch ohne die weitere Ausnutzung von Pipeline-Parallelismus zwischen den Komponenten theoretisch möglich ist.

9.2 Ein vorkonditioniertes Konjugierte-Gradienten-Verfahren in datengetriebenem FPGA-Rahmenwerk

Problematisch ist bei der Verlagerung des Vorkonditionierers nach Abschnitt 9.1 allerdings noch, dass die Matrizen in die FPGAs geladen, dort verarbeitet, und anschließend sofort wieder zurückgeschrieben werden. Das mikroprogrammierbare, datengetriebene Koprozessorrahmenwerk aus Abschnitt 6.4 hat zum Ziel, die sofortige Weiterverarbeitung der ausgehenden Daten sowie die zeitgleich mehrfache Verarbeitung durch unterschiedliche Einheiten zu erreichen, und dadurch dieses Datentransportproblem zu beheben.

9.2.1 Konzept und Implementierung

Daher stellt das Rahmenwerk nun alle für das CG-Verfahren benötigten Komponenten zur Verfügung, wie sie bereits in Abb. 6.12 enthalten sind. Das CG-Verfahren muss aufgrund der zahlreichen Datenabhängigkeiten und ohnehin notwendigen Rückschreiboperationen in vier Teile wie in Abb. 9.2 aufgeteilt werden, wobei der erste nur der Initialisierung dient. Die weiteren drei Teile werden in jeder Iteration nacheinander mit jeweils einem eigenen Mikroprogramm aufgerufen [NBK+13]. Diese als *Blockvariante* bezeichnete Variante benötigt jede Recheneinheit nur einmal und vereinfacht damit das Hardwaredesign. Zukünftige konfigurations- und domänenspezifische Compiler können dadurch ebenso einfacher gehalten werden.

9.2.2 Laufzeitmessung und Vergleich

Unter Verwendung der Matrixbibliothek aus Abschnitt 5.1.1 konnte das CG-Verfahren sequentiell und parallel auf unterschiedlichen Architekturen zum Vergleich mit dem koprozessorgestützten Ansatz durchgeführt werden. Letztgenannter liefert gegenüber der sequentiellen Ausführung bereits ab Eingabedaten der Größe 32×32 Leistungssteigerung und ist bei sehr großen Datenmengen auch mit einer mit `-O2` kompilierten Version auf einem schnellen 24-Kern-System vergleichbar (sogar etwa

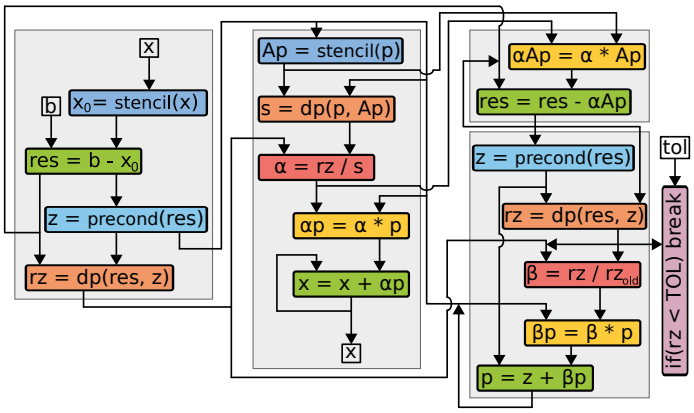


Abbildung 9.2: Aufteilung des CG-Verfahrens zur Nutzung des datengetriebenen, mikroprogrammierbaren Rahmenwerks mit Vorkonditionierer.

19 % schneller, Abbildung 9.3). Betrachtet man die Menge an intern verarbeiteten Daten gegenüber der Bandbreite und skaliert diese auf die verfügbaren vier Application Engines, so ist sowohl eine wesentlich höhere Datenverarbeitungrate erwartbar als auch bessere Ausnutzung der zur Verfügung stehenden externen Bandbreite. Die koprozessorgestützte Variante benötigt allerdings auch mehr Iterationen als die Softwarevariante, bis das Konvergenzkriterium erreicht ist.

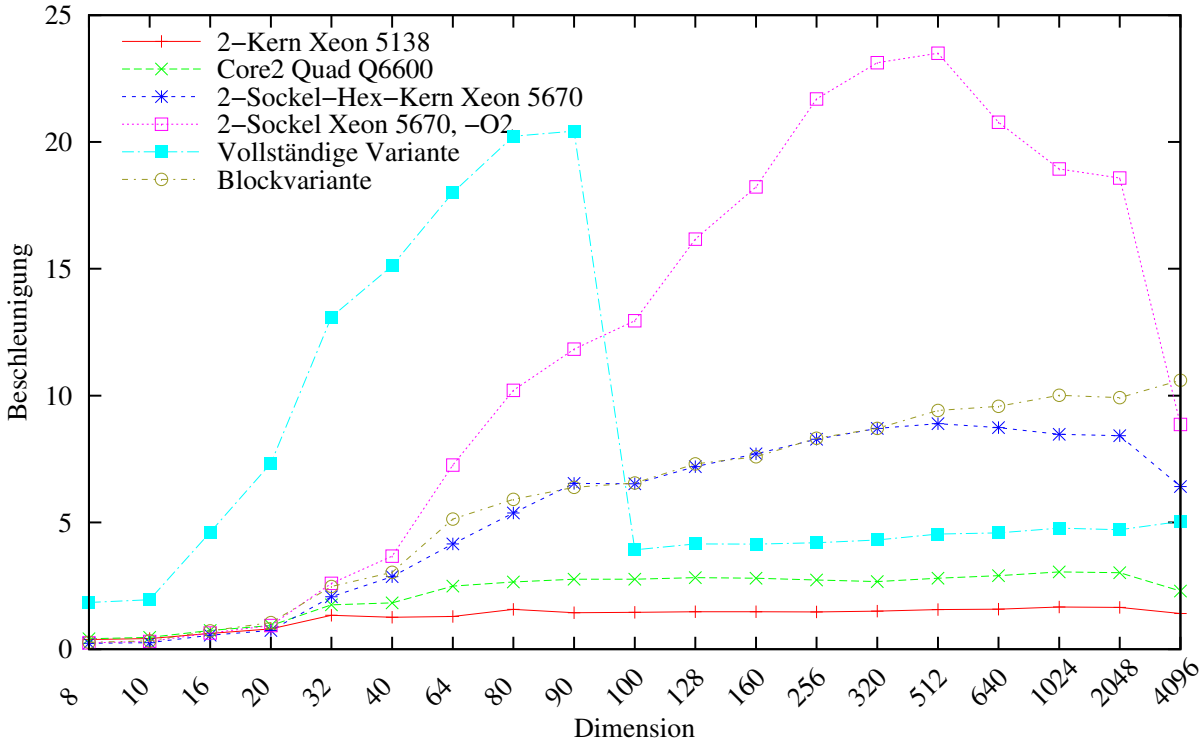


Abbildung 9.3: Leistung des CG-Verfahrens auf der Convey HC-1 mit Intel Xeon 5138 und mit einer AE gegenüber reiner Softwareausführung auf unterschiedlichen Systemen.

Besonders großer Nutzen ist von größeren FPGAs erwartbar, wo alle für die gleichzeitige, überlappte, ressourcenkonfliktlose Ausführung des CG-Verfahrens benötigten Recheneinheiten vorhanden sind (als „vollständige Variante“ für kleine Datenmengen umgesetzt; durch die Anzahl an Speicherressourcen auf Daten kleiner als 100×100 Elemente eingeschränkt). Dennoch ist ohne Weiteres das Design mit allen Komponenten weder für die Convey HC-1 mit Xilinx Virtex-5 LX330-FPGAs noch für die Convey HC-1^{ex} mit Xilinx Virtex-6 LX760-FPGAs unter Einhaltung der Zeitschranken korrekt synthetisierbar, wie Tabelle 9.3 belegt.

Ressource	Vollständige Variante HC-1	Vollständige Variante HC-1^{ex}
Slices	45,210 (87 %)	58,039 (48 %)
Slice-Register	112,603 (54 %)	110,593 (11 %)
Slice-LUTs	134,381 (64 %)	172,114 (36 %)
Slice-LUT-Flip-Flop Pairs	159,820 (77 %)	190,879 (40 %)
RAMB36	264 (92 %)	274 (38 %)
DSP48	138 (72 %)	138 (15 %)
Takt	14,164 ns	14,889 ns
(Ziel: 6,667 ns)	(212 %)	(223 %)

Tabelle 9.3: Ressourcenbedarf bei der Integrierung aller benötigten Einheiten für das CG-Verfahren.

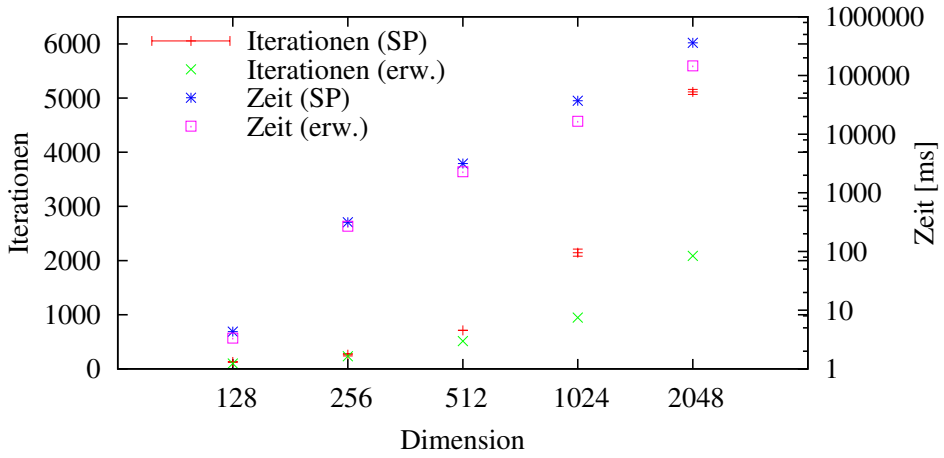


Abbildung 9.4: Reduzierung der Anzahl benötigter Iterationen und damit einhergehend der Ausführungszeit bei Erhöhung der Auflösung von einfacher Genauigkeit (SP) um 23 bzw. 32 Bits (erw.).

9.2.3 Verbesserung der Konvergenz

Mittels des Mikroprogrammansatzes lassen sich die errechneten Werte bequem in den Koprozessorspeicher zurückschreiben, um sie im Rahmen der Fehlersuche und -behebung zu verwenden. Anhand von Abgleichen der errechneten Matrizen gegenüber der Ausführung in Software ließen sich die höheren Anzahlen an benötigten Iterationen gegenüber der sequentiellen Ausführung vor allem auf das Skalarprodukt zurückführen. Selbiges ist derart implementiert, dass zum aktuellen Zeitpunkt nicht reduzierbare Daten intern zwischengespeichert werden, was von der Rate der eingehenden Daten abhängt. Die zwischengespeicherten Daten werden später akkumuliert. Der dabei vorhandene Indeterminismus beeinflusst ähnlich der Ausführung mit OpenMP und dynamischer Ablaufplanung das Ergebnis beim Skalarprodukt und führt letztendlich sogar zu variierender Anzahl an Iterationen für unterschiedliche Läufe. Der Hardwarespezialist kann im Rahmen der verfügbaren FPGA-Ressourcen eine genauere Implementierung zur Verfügung stellen. Das Skalarprodukt wurde intern bei der Multiplikation um 23 Bits erweitert, um keine Informationen zu verlieren, und um 32 Bits bei der Addition, um ein breiteres Fenster zu haben, in welchem genau akkumuliert wird. Die exakte Skalarprodukteinheit wurde aufgrund ihrer enormen Platzanforderungen nicht eingebracht, aber gewissermaßen approximiert. Dadurch ließen sich die Iterationszahlen ohne weitere softwareseitige Veränderungen stark reduzieren auf nur noch 40 % der vorherigen, wie in Abb. 9.4 gut deutlich wird, wodurch sich auch die Gesamtausführungszeit entsprechend senkt. Damit ist der vorgeschlagene Ansatz bereits ab Problemgrößen von 1024×1024 vorteilhaft gegenüber einem mit OpenMP programmierten 24-Kern-System.

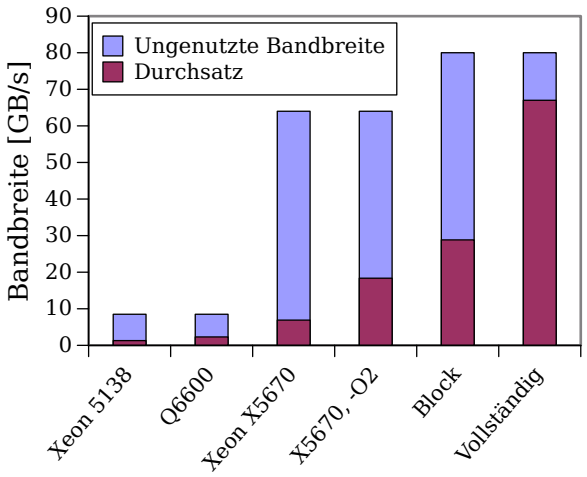


Abbildung 9.5: Nutzung der verfügbaren Bandbreite (skaliert auf vier Application Engines bei der HC-1).

9.2.4 Ergebnis der mikroprogrammgestützten Verlagerung des Konjugierte-Gradienten-Verfahrens

Der vorgeschlagene Ansatz, dem Anwendungsentwickler vorgefertigte, domänenspezifische Komponenten auf einem Koprozessor anzubieten, der über Mikroprogramme konfiguriert und gesteuert wird, ermöglicht die Verlagerung von ganzen Verfahren in rekonfigurierbare Hardware. Unnötiger Datentransport wird durch Vorhalten der Zwischendaten in einem zentralen Puffersatz vermieden; das Rückschreiben von Daten erfolgt nebenläufig zu weiteren Berechnungen. Obwohl nur maximal vier DMA-Einheiten vorhanden sind, und damit nur ein Viertel der eigentlich verfügbaren externen Bandbreite nutzbar ist, ist die Anzahl durchgeführter Gleitkommaoperationen (MFLOPS bzw. MFLOPS * 4 Byte/Operation) auf den gelesenen Daten sehr hoch gegenüber der Verwendung von Standardprozessoren (Abb. 9.5). Der vorgeschlagene Ansatz ermöglicht also, die Belastung der Speicherschnittstellen zu verringern, und kann die Speicherbeschränktheit von Anwendungen teilweise umgehen durch die hohe interne Datenwiederverwendung. Dadurch lässt sich bereits vergleichbare Leistung zu aktuellen Mehrkernsystemen mit vielen Hardware-Threads erzielen. Das vorherrschende Datentransferproblem ist somit gelöst. Diese Leistung lässt sich durch die Integration von Spezialeinheiten noch weiter verbessern. Durch ein exakteres Skalarprodukt reduziert sich die Anzahl benötigter Iterationen auf nur 40 %. Die Spezialimplementierung der Skalarprodukteinheit ist für den Anwender transparent und erfordert auch keinen Zusatzaufwand seitens des Anwenders.

9.2.5 Abschließende Energiebetrachtungen

Bei der Koprozessorausführung wurde lediglich eine Application Engine eingesetzt. Zusätzlich verfügt der Koprozessor über drei weitere AEs, die je aus einem FPGA bestehen, und über acht Speichercontroller, die aus einem weiteren FPGA bestehen, sowie schlussendlich auch über die Speicher an sich. Da ein homogenes Mehrkernsystem ebenso über gewisse Speichercontroller und Speicherbausteine verfügen muss, soll es genügen, die Energiebetrachtungen auf die durchschnittliche Leistungsaufnahme der Ausführungseinheiten und auf ihre jeweilige Ausführungsdauer zu reduzieren. In den AEs der Convey HC-1 kommen Xilinx Virtex-5 LX330 mit einer maximalen Leistungsaufnahme von 35,8 W zum Einsatz. Hingegen ist beim Prozessor Intel Xeon X5670 auf dem zum Vergleich herangezogenen Zwei-Sockel-System mit insgesamt 24 Hardware-Threads die maximal abzuführende Wärme mit 95 W spezifiziert[1]. Die Ausführungszeiten und der zugehörige Energiebedarf für unterschiedliche Problemgrößen sind in Tabelle 9.4 aufgeführt und zusammengefasst. Der Energiebedarf wurde dazu über die als durchschnittlich angenommene Leistungsaufnahme einer Einheit und die jeweilige Ausführungszeit ermittelt, d.h. für eine Auflösung von 512×512 ergibt

[1] engl. Thermal Design Power (TDP): Maximale erwartbare, dauerhafte Wärmeabgabe. Wird in Ermangelung besserer Angaben zur modellhaften Rechnung benutzt.

	512×512	1024×1024	2048×2048	4096×4096
Ausführungszeit CPU	2,405 s	21,242 s	168,618 s	1896,006 s
Ausführungszeit AE	2,286 s	16,607 s	145,628 s	1196,383 s
Max. Energiebedarf CPU (a)	126,938 mWh	1121,100 mWh	8899,266 mWh	100066,993 mWh
Max. Energiebedarf FPGA	22,729 mWh	165,144 mWh	1448,186 mWh	11897,358 mWh
Errechneter Energiebedarf FPGA (b)	11,072 mWh	80,446 mWh	705,444 mWh	5795,476 mWh
Verhältnis ($\frac{a}{b}$)	11,5	13,9	12,6	17,3

Tabelle 9.4: Ausführungszeit, mittels durchschnittlicher Leistungsaufnahme und benötigter Ausführungszeit bestimmter Energiebedarf und Verhältnis des Energiebedarfs zwischen der Ausführung auf einer AE der Convey HC-1 und mehrfädiger Ausführung auf zwei Intel Xeon X5670.

sich $E_{\text{CPU}} = 2 * 95\,\text{W} * \frac{2{,}405}{3600}\,\text{h} = 0,127\,\text{Wh}$. Es ist anzumerken, dass das verwendete Design aufgrund der nicht vollständig verwendeten FPGA-Ressourcen lediglich eine Leistungsaufnahme von 17,439 W haben soll laut Bericht des *Xilinx XPower Analyzer* (XPA). Für die FPGA-Ausführung halbiert sich daher die aufgenommene Leistung im Endeffekt, während die Ressourcen der zwei CPUs nahezu vollständig genutzt sind. Die maximale Leistungsaufnahme eines Prozessors liegt etwa bei dem Anderthalbfachen der TDP (oder noch höher); die durchschnittliche[2] liegt bei etwa 2/3 der TDP. Entsprechend ist davon auszugehen, dass die Verbesserung des Energiebedarfs nicht zwischen 11,5× und 17,3× liegt, sondern zwischen 7,7× und 25,9×; also noch immer sehr viel weniger Energie benötigt wird. Insofern stellt die Wahl der TDP zur Bestimmung des Energiebedarfs ein simples Mittel dar zur ersten Abschätzung der Energieeffizienz.

9.3 Datengetriebene Ausführung des CG-Verfahrens in Software

Nachdem es sich als vorteilhaft erwiesen hat, in Koprozessoren die Daten datengetrieben und möglichst überlagert zu verarbeiten (Abschnitt 9.2), idealerweise unter Einsatz von Spezialeinheiten wie Reduktionseinheiten für die Maximumsberechung (Abschnitt 4.2.3) oder Skalarprodukten (Abschnitte 5.4, 9.2.3), wird dieser Ansatz auch auf die Ausführung in Software übertragen. Mit der datengetriebenen Umsetzung (Abschnitt 8.1) der Matrixbibliothek (Abschnitt 5.1.1) wurde daher das CG-Verfahren implementiert, auf dem 24-Hardwarethreads-System mit zwei Intel Xeon X5670 ausgeführt und verglichen gegenüber OpenMP.

Dabei wurden die vorab erlangten Erkenntnisse beachtet und wie folgt die Implementierung angepasst. Die Operationen sind derart angeordnet, dass Allokationen so früh wie möglich erfolgen, abgesehen von der Initialisierung des Vektors b für die rechte Seite, die bereits zeitgleich mit weiteren Allokationen erfolgen kann. Sie weiter nach hinten zu verschieben, hat in der Implementierungsphase bereits negative Auswirkungen offenbart. Auch für unterschiedliche andere Anordnungsmöglichkeiten der unterschiedlichen Typen von Operationen (Allokationen, arithmetische Berechnungen, Freigeben) wurden Messungen vorgenommen. Die beste Alternative wurde implementiert – Allokationen frühzeitig durchführen. Matrizen werden allesamt den Zeilen nach gespeichert („row-major“), da keine Operationen wie Matrixmultiplikationen durchgeführt werden, die von andersartiger Speicherung profitieren würden. `Gcc -O3` wurde zum Kompilieren verwendet in Version 4.5 und 4.6, da sich mit ICC keine nennenswerten Unterschiede erkennen ließen abgesehen von schnellerer Ausführung. Abbildung 9.6 zeigt deutlich, dass der Nutzen datengetriebener Ausführung wie auch schon zuvor bei der Verlagerung auf einen Koprozessor erst ab größeren Datenmengen oberhalb von Auflösungen von 2000×2000 auftritt, wenn die im System vorhandenen Caches zur Vorhaltung der Daten bei der Ausführung mit OpenMP nicht mehr ausreichen. Der datengetriebene Ansatz hingegen verwendet die Daten nach Möglichkeit wieder, solange sie noch im Cache vorhanden sind. Dadurch verringert sich die Ausführungszeit um 10 % bis 25 %.

[2] engl. Average CPU Power (ACP)

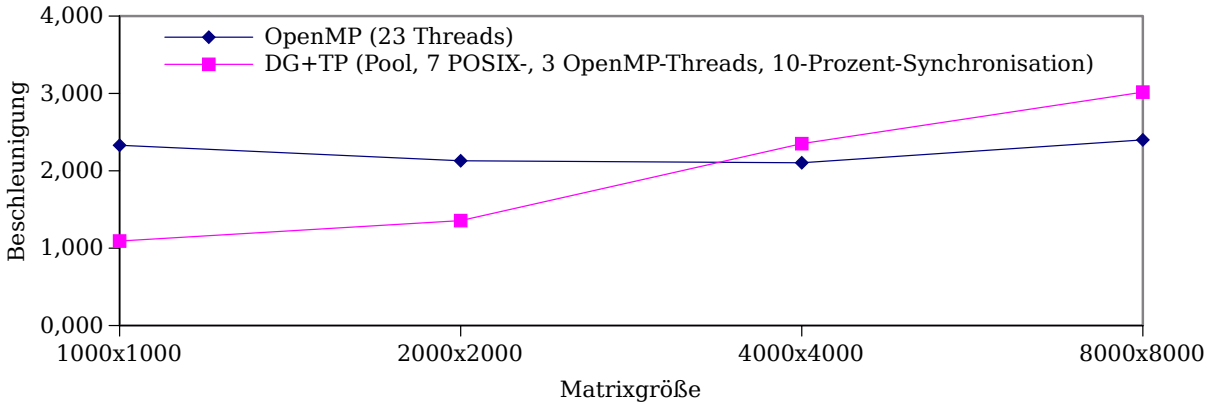


Abbildung 9.6: Speedup von bibliotheksbasierter OpenMP-Ausführung des CG-Verfahrens gegenüber datengetriebener, taskparalleler (DG+TP) Ausführung.

9.4 Überlagerte Ausführung von HHblits in Software und rekonfigurierbarer Hardware

Die Ziele des oben beschriebenen und evaluierten datengetriebenen Programmiermodells sind einerseits, die Koprozessorausführung eng in die Ausführung der nicht verlagerten Programmteile einzubinden, sowie andererseits, die Programmierung für das datengetriebene, mikroprogrammierbare Rahmenwerk dadurch zu erleichtern, dass sehr ähnlicher Programmcode in Hochsprache entwickelt und in Software vorab ausgeführt werden kann. Letztendlich wird so erreicht, dass leicht das Mikroprogramm für die Koprozessorausführung verändert werden kann, um kleinere oder größere Teile der Anwendung in die Koprozessorhardware zu verlagern und so den größtmöglichen Nutzen des Koprozessors zu bestimmen.

Der erzielte Nutzen des datengetriebenen, taskparallelen Programmiermodells ist noch zu gering für den betriebenen Aufwand mit Warteschlangen und Evaluation der besten Parameter bestehend aus Anzahl POSIX- und OpenMP-Threads sowie der bestgeeigneten Möglichkeit zur Synchronisation. Daher ist nun entsprechend oben zusammengefasster Ziele zu untersuchen, ob die datengetriebene Programmausführung über die Grenze zwischen Prozessor und Koprozessor hinweg mehr Verbesserung erzielen lässt als die Verlagerung von Anwendungsteilen in den Koprozessor. Da der Koprozessor der Convey HC-1 einen eigenen Speicher hat, zu welchem die schnellste Möglichkeit zur Übertragung großer Daten aus Blocktransfers besteht, bietet es sich nicht an, das CG-Verfahren mit seinen Eingabe- und Ausgabematrizen dafür zu untersuchen. Stattdessen braucht es eine Anwendung, bei welcher die Ein- oder Ausgabe nur geringe Datenmenge hat. Eine solche findet sich mit der Berechnung des lückenlosen Scores in HHblits. Dabei wird für jede verglichene Sequenz nur ein 1 Byte großer Maximalwert zurückgegeben Es bietet sich daher an, die Ausführung des Koprozessors mit nachfolgenden Operationen zu überlagern.

Dies konnte wie folgt umgesetzt werden: Der erste Vorfilterungsschritt, die Berechnung des lückenlosen Scores, vergleicht mittels einer OpenMP-SSE-Implementierung oder auf dem Koprozessor (Abschnitt 4.2.3) die Sequenzen der Datenbankeinträge mit dem Profil der Anfrage. Sobald ein oder mehrere Sequenzvergleiche erfolgt sind und die berechneten Maxima über einem gewissen Schwellwert liegen, startet bereits der zweite Schritt der Vorfilterung seine Arbeit, der aus einer streifenweise arbeitenden Smith-Waterman-Implementierung besteht. Die Komponente der ersten Stufe für den lückenlosen Score wird dazu als eigener POSIX-Thread gestartet und synchronisiert sich mit der zweiten Komponente (Berechnung des lückenbehafteten Scores) mittels Semaphoren. Die Smith-Waterman-Implementierung hat einen Anteil von etwa 4 % an der sequentiellen, unbeschleunigten Ausführungszeit und sollte sich daher perfekt mit dem ersten Schritt überlappt ausführen lassen.

Wie die Messung der Software-Implementierung auf einem 24-Kern-System mit zwei Intel Xeon X5670 in Abb. 9.7 zeigt, ist es wirklich möglich, die zweite Stufe nahezu vollständig hinter der Ausführung der ersten Stufe zu verstecken.

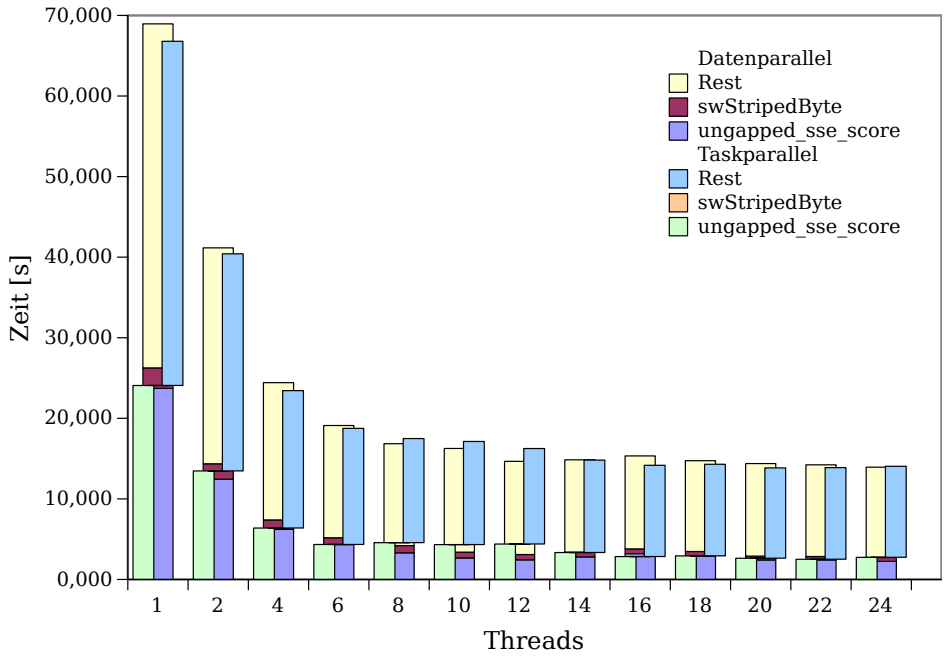


Abbildung 9.7: Überlappung der Smith-Waterman-Implementierung mit der Berechnung des lückenlosen Scores auf dem Intel Xeon X5670-System mit 24 Hardwarethreads.

Aufgrund dieser positiven Ergebnisse wurde die Implementierung auch auf den in Abschnitt 4.2.3 entwickelten Koprozessor übertragen. Es stellt sich in den über fünf Läufe gemittelten Messungen eine Verbesserung der Gesamtausführungszeit von 59,6 Sekunden auf 51,6 Sekunden ein bei Verwendung nur eines einzigen weiteren Threads/Kerns für die o.g. zweite Komponente, also mehr als 13 % Verbesserung. Wird HHblits zusätzlich mit zwei Worker-Threads anstelle nur eines Workers zur parallelen Ausführung des Viterbi-Algorithmus auf der Convey HC-1 betrieben, so ändert sich dieses Verhältnis geringfügig auf 49,8 Sekunden zu 45,8 Sekunden durch die Ausnutzung von Task-Parallelismus zur Ausführung der zweiten Komponente, also auch eine Verbesserung um 8 %.

9.4.1 Abschließende Energiebetrachtungen

Bei der Verlagerung der ersten Vorfilterungsstufe auf den Koprozessor werden alle vier Application Engines eingesetzt. Jede einzelne hat eine maximale Leistungsaufnahme von 35,8 W sowie eine von XPA errechnete Leistungsaufnahme von 18,561 W. Der auf der Convey HC-1 eingesetzte Host-Prozessor ist unter Berücksichtigung der Energieeffizienz entwickelt worden und verfügt über eine TDP von lediglich 35 W. Tabelle 9.5 ist entnehmbar, dass die Verwendung eines heterogenen Systems insbesondere dann unter dem Gesichtspunkt des Energiebedarfs sinnvoller ist als die alleinige Ausführung im Prozessor, wenn der Prozessor ebenso in die Ausführung der Anwendung qua Ausnutzung von Task- bzw. Pipeline-Parallelismus eingebunden ist.

9.5 Ergebnis und Zusammenfassung

Um Koprozessoren erfolgreich einzusetzen, müssen die Schaltungen von Hardwarespezialisten entwickelt werden. Problematisch bleibt dabei das Verhältnis von Datentransfer zu Berechnung. Um dieses zu erhöhen, wurden mehrere domänenspezifische Komponenten auf einem FPGA mittels eines zentralen Puffersatzes verbunden und über eine mikroprogrammierbare Steuereinheit angesteuert. Damit lässt sich das wiederholte Einlesen von Daten vermeiden und die Granularität der Verlagerung erhöhen von Funktions- auf Algorithmen- oder Anwendungsebene. Durch die dabei entstehende Wiederverwendung der Daten wird obiges Problem gelöst. Aufgrund der geringen

	CPU (1 Thread)	**Koprozessor-gestützt**	**Überlappt**
Ausführungszeit [s]	74,585	37,465	34,147
Leistungsaufnahme (TDP-Wert / XPA-Ausgabe) [W]	35	CPU: 35; FPGA: 18,561	
Energiebedarf [mWh]	362,566	507,048	507,048
Beschleunigung gegenüber CPU	–	1,991	2,184
Energiemehrbedarf (rel.) gegenüber CPU	–	1,398	1,398
Effizienzsteigerung (1/EDP)	–	1,424	1,562

Tabelle 9.5: Ausführungszeit, über benötigte Ausführungszeit und als durchschnittlich angenommene Leistungsaufnahme ermittelter Energiebedarf und Effizienzsteierung zwischen der Ausführung von HHblits auf der Convey HC-1 ohne und mit überlappter Ausführung des Koprozessors sowie die sich dadurch errechnende Erhöhung der Energieeffizienz.

Taktfrequenz von FPGA-Schaltungen sowie aufgrund der Verwendung von Gleitkommaoperationen auf FPGAs ist die Leistung rekonfigurierbarer Hardware der Leistung eines aktuellen SMP-Systems mit Mehrkernprozessoren noch nicht überlegen. Die Verwendung mehrerer FPGAs mit unterschiedlichen Mikroprogrammen oder mehrerer FPGAs zur Berechnung von Teilproblemen nach Gebietszerlegung wird dann aber auch gegenüber solchen Systemen Geschwindigkeitssteigerung erzielen. Besonders am komponentenbasierten Ansatz ist aber, dass transparent für den Anwendungsentwickler Spezialimplementierungen wie etwa exaktere Skalarprodukte integriert sein können. Deren zusätzliche Genauigkeit reduziert die Anzahl benötigter Iterationen wesentlich um 60 % und damit auch die Ausführungszeit der koprozessorgestützten Anwendung. Die Komponenten werden auf dem Koprozessor nach Möglichkeit zeitgleich ausgeführt, und bei Vorhandensein von Datenabhängigkeiten werden sie datengetrieben ausgeführt, ohne auf die vollständige Bearbeitung des Vorgängerknotens im Anwendungsgraphen warten zu müssen. Dieses Modell ist auch auf die Ausführung in homogenen Mehrkernprozessoren übertragbar und bringt geringfügige Verbesserung gegenüber mit OpenMP parallelisierten Varianten. Bindet man jedoch den Koprozessor ebenso datengetrieben in die Anwendung ein, so können die Kerne des Hostprozessors zeitgleich weitere Komponenten ausführen und Anwendungen sowohl von der Nutzung des Koprozessors als auch vom Einsatz mehrere Prozessorkerne profitieren. Ein Mikroprogramm für den Koprozessor kann dann als Implementierung einer mittel- bis grobgranularen Komponente in ein heterogenes System eingebunden werden. Die (zusammengesetzte) Komponente wird anstelle auf Prozessorkernen dann auf dem Koprozessor ausgeführt, wo sie Leistung vergleichbar zu etwa 24 Hardwarethreads liefern kann. Die übrigen Rechenkerne im heterogenen Mehrkernsystem führen weitere Komponenten als Tasks aus, und können dadurch sogar geringfügig Leistungssteigerung gegenüber der datenparallelen Ausführung mit OpenMP erzielen. Vorliegende Datenkonflikte werden durch Angabe der Gültigkeit bereits berechneter Daten behoben und die Komponenten können sowohl in Hardware als auch in Software datengetrieben arbeiten. Liegen hingegen keine Datenabhängigkeiten vor, so kann task-parallel gearbeitet werden und die Speicherschnittstelle durch gleichzeitig mehrfache Nutzung der Daten maximal ausgenutzt sowie durch mehrere gleichzeitige Transfers maximal ausgelastet werden.

Kapitel 10

Zusammenfassung

Der Beitrag dieser Dissertation soll abschließend zusammengefasst werden. Neben dem bloßen Herausstellen des Beitrags und Inhalts sollen auch die durch die vorliegende Arbeit gelieferten Perspektiven aufgezeigt werden und mit den gewonnenen Erkenntnissen hinsichtlich der Anforderungen an zukünftige Systeme mit FPGAs als Koprozessoren abgeschlossen werden.

10.1 Inhalt und Beitrag dieser Arbeit

Aufgrund der Ineffizienz – hinsichtlich nötigem Rechenaufwand und erzielbarem Nutzen – von Standardprozessoren bei besonderen Operationen oder Anwendungsgebieten und umgekehrt der Ineffizienz – hinsichtlich Ressourcenbedarf, Ausführungsdauer und erzieltem Nutzen – von Spezialhardware beim Allzweckrechnen ist davon auszugehen, dass moderne Rechner und Chips zukünftig heterogen sein werden [179]. Eventuell werden sie sogar aus mehreren Komponenten mit unterschiedlichen Befehlssätzen und Einsatzzwecken bestehen. In dieser Arbeit wurden verschiedene Methoden und Konzepte untersucht, wie rekonfigurierbare Hardware in aktuellen Mehrkernsystemen und zukünftigen Vielkernsystemen gewinnbringend eingesetzt werden kann.

Der in rekonfigurierbarer Hardware massiv ausnutzbare Parallelismus kann sehr gut für bioinformatische Anwendungen eingesetzt werden. Die Bioinformatik hat sich bereits auf algorithmischer Ebene erfolgreich mit dem Problem beschäftigt, Datenabhängigkeiten zu vermeiden bzw. bestehende Datenkonflikte zu beheben und so stark parallele Verarbeitung zu erlauben, etwa die Verwendung von SSE-Instruktionen zur 16-fach parallelen Berechnung eines lückenlosen Scores bei der Vorfilterung im Werkzeug HHblits. Mittels der Convey HC-1 lässt sich gegenüber der SSE-Implementierung eine Beschleunigung von 4,46 für die Berechnung aller lückenlosen Scores dadurch erreichen, dass die gesamte Filterfunktion ausgelagert wird, wie in Abschnitt 4.2 beschrieben. Die Filterfunktion führt über alle in der Datenbank eingetragenen Sequenzen einen Kernel aus, welcher aus der Berechnung des Maximums über 16 Werten besteht, die parallel zueinander ausgeführt werden. Der Kernel allein hätte entsprechend der bereits erhaltenen Erkenntnisse keinen Nutzen durch eine Portierung auf einen FPGA erhalten können. Die Funktionsausführung hat jedoch nur einen Anteil von etwa 63 % an der Gesamtausführungszeit, so dass mit dem zusätzlichen Aufwand lediglich geringfügige Beschleunigung von Faktor $\frac{44{,}524\text{ s}}{39{,}456\text{ s}} = 1,13$ erreichbar ist. Versteckt man jedoch die zweite Stufe der Vorfilterung, und zwar die byteweise Berechnung des Smith-Waterman-Algorithmus mit ihrer Laufzeit von 2,96 s, „hinter“ der Koprozessorausführung mittels eines weiteren Threads nach dem Erzeuger-Verbraucher-Prinzip, so erhöht sich der Speedup weiter auf 1,22. **Durch diese als Pipeline-Parallelismus bezeichnete Unterform des Task-Parallelismus kann Leistungssteigerung erhalten werden und zugleich die hohe Leistungsaufnahme von rekonfigurierbarer Hardware kompensiert werden.**

Als Beispiel zur Nutzung rekonfigurierbarer Hardware für numerische Anwendungen wurden verschiedene exaktere Arithmetiken [177] betrachtet. Wie in Abschnitt 2.2.3 gezeigt wurde, sind sie schlecht mit Mikroprozessoren und Software umzusetzen, versprechen aber hohen Nutzen durch Verringerung der Anzahl benötigter Iterationen und damit der Gesamtlaufzeit beispielsweise des Konjugierte-Gradienten-Verfahrens (vgl. Abschnitt 5.3). Die Implementierung [31] exakter Arithmetik muss mit hardwarenahen Entwicklungssprachen erfolgen, um die Parallelität auf Bitebene, spezielle Formate und Vektorlängen erfolgreich auszunutzen, und kann daher keinesfalls abstrahiert mit C-zu-VHDL-Sprachen erfolgen. Eine hardwarenahe Implementierung allein reicht jedoch nicht

aus, die bereits vergleichsweise sehr langsamen Softwareimplementierungen zu ersetzen, wie in der Evaluation der ersten Implementierung in Abschnitt 5.4 ersichtlich wird [NBKK09]. Grund dafür ist im Wesentlichen der für das Starten des Koprozessors und Übertragen der Instruktion und Daten nötige Zusatzaufwand, im Weiteren aber auch das schlechte Verhältnis von Berechnungen zu Datentransfer[1]. Dieses Verhältnis lässt sich über Datenwiederverwendung mittels Vorabladen eines Vektors und Ablegen selbigens in einem dedizierten Speicher erreichen [NB10]. Dadurch steigt die Granularität vom Skalarprodukt zweier Vektoren hin zu einem Matrix-Vektor-Produkt. **Demzufolge muss die Granularität der FPGA-Nutzung oberhalb der Instruktionsebene liegen; auf dem FPGA muss ein komplexerer Algorithmus oder zumindest ein Teil dessen ausgeführt werden.**

Die Verwendung von Hochsprachen wie C zur Synthese von Hardwareschaltungen für FPGAs wurde anhand von Impulse C in Abschnitt 5.8.4 und dem Compiler ROCCC in Abschnitt 5.8.5 untersucht. Die Ausführungszeiten [HKN+11] eines mit Impulse C beschriebenen Vorkonditionierers [233] bei **Implementierung seitens eines Domänenexpertens sind zwei Größenordnungen langsamer als die CPU-Ausführung**.[2] Ursächlich dafür ist, dass neben der Einarbeitung in das Werkzeug und die Sprache auch die Konzepte zur Datenübertragung, Datenwiederverwendung und Überlappung von Datentransfer, hostseitiger Berechnung und FPGA-seitiger Berechnung bekannt sein müssen. Darüberhinaus muss die eingesetzte digitale Arithmetik effizient hinsichtlich Anzahl Verarbeitungsschritten und Ressourcenbedarf sein sowie passende Operationen nutzen (vgl. Abschnitt 2.2) und darf zu keinen Genauigkeitsproblemen führen. All dies muss schließlich auch mit dem hochsprachlichen Werkzeug implementierbar sein. Wie in Abschnitt 5.8.4 beschrieben, muss Pipelining eingeführt werden durch Ersetzen der Division durch Multiplikation mit dem Kehrwert, die geeigneteste Methode zur Datenübertragung ermittelt werden, müssen langsame Berechnungen wie die Modulo-Berechnung hardwareorientiert durch einen Zustand modelliert werden oder der entsprechende Code ausgerollt, Speicherzugriffe serialisiert und gestreamt werden durch Bereithaltung der zuletzt verwendeten und noch wiederzuverwendenden Daten, mehrere Stencil-Einheiten parallel zueinander ausgeführt werden. Allein aufgrund der mit der Division wegfallenden zwangsweisen Serialisierung bei gleichzeitiger Entfernung der verbleibenden möglichen Engstellen ist 12- bis 13-fache Beschleunigung zu erwarten, die jedoch nicht mehr gemessen werden konnte aufgrund von Hardwareversagen.

Auf dem Multi-FPGA-System Convey HC-1 kann eine Anwendung durch Einsatz eines angepassten Compilers automatisch die „Single-Precision Vector Personality“ nutzen, um Vektoroperationen auf einfach genauen Daten auszuführen. Bei Auslagerung allein des Vorkonditionierers auf den Koprozessor unter Zuhilfenahme des Compilers wird die Anwendung verlangsamt trotz Beachtens oben formulierter Prinzipien. Demgegenüber zeigt Abschnitt 5.8.7, dass die Ausführung sämtlicher für das **Konjugierte-Gradienten-Verfahren benötigter Operationen auf den FPGAs mittels der Vektor-Personalitys** zu einem Speedup bis zu etwa 2,2 führen kann. Zwar müssen die Daten vom Host zum Koprozessor und zurück übertragen werden, die von Convey bereitgestellte systemangepasste Kopierroutine vermag diese Operation jedoch sehr schnell durchzuführen, so dass insgesamt gegenüber Softwareausführung Beschleunigung möglich ist. **Bei zunehmender Abstraktion der Werkzeuge von der Hardware muss auch die Granularität und damit einhergehend die Datenmenge und der Rechenaufwand auf Akzeleratoren höher** sein, um hinreichend Beschleunigung erhalten zu können.

Das Übertragen der Daten zwischen hostseitigem Speicher und beschleunigerseitigem Speicher stellt stets ungewünschten zeitlichen Zusatzaufwand dar. Zwar lässt sich die Übertragung in Abhängigkeit des Systems und des auf den Beschleuniger zu portierenden Verfahrens manchmal verbergen, viel wünschenswerter ist es jedoch, die Daten im Rahmen der verfügbaren Chipfläche in weiteren Schaltungen (Komponenten) sofort weiterzuverwenden. Beispielsweise kann die (Quadrat-)Norm eines mittels einer `axpy`-Operation aktualisierten Vektors zeitgleich bei jedem bereits vollständig aktualisierten Element weiterberechnet werden. Selbstverständlich lässt sich eine dedizierte Hardwareschaltung dafür mit einem hochsprachlichen Werkzeug wie FloPoCo [82] und ähnlichen direkt synthetisieren. Sie benötigt aber mehr Platz als eine einfache, skalare `axpy`-Einheit und wäre somit

[1] engl. compute to memory access ratio

[2] Es ist nicht zielführend, die Fähigkeit von Domänenexperten zur korrekten und effizienten Verwendung zu diskutieren, sondern vielmehr den Nutzen bzw. die Aufgabenverteilung, ob der Domänen- oder der Hardwareexperte die Portierung vornimmt.

für die Durchführung von regulären `axpy`-Operationen unnötig ressourcenaufwendig. Gleichermaßen könnte eine Skalarproduktеinheit auch ohne `axpy`-Einheit benötigt werden, etwa zum Summieren von Elementen. Es wird also ein Rahmenwerk benötigt, das die zeitgleiche, unabhängige sowie datenabhängige Ausführung mehrerer Hardwareimplementierungen von verschiedenen oder gleichen Einheiten ermöglicht. Die Verwendung einer ringbusähnlichen Verbindung (Abschnitt 6.3) zum Datenaustausch zwischen verschiedenen Funktionseinheiten in FPGA-Hardware hat sich aufgrund der hohen Latenz und der schlechten Konfigurierbarkeit seitens eines Anwendungsentwicklers bezüglich der Verbindung zwischen den Funktionseinheiten als ungeeignet herausgestellt. Bieten FPGA-Hersteller Schnittstellen für ihre Hardware in RTL (meist Verilog oder VHDL) an, so ist diese auf den Anschluss genau einer Funktionseinheit (oder ggf. eines selbst zu entwerfenden Rahmenwerks) zugeschnitten passend zu den von Hochsprachen wie Impulse C benötigten Schnittstellen. Entsprechend einfach lässt sich z.B. die Schnittstelle der DRC AC2030 für ROCCC anstelle von Impulse C umgestalten und anpassen; unabhängig von der verwendeten Sprache ist die Wiederverwendbarkeit jedoch nicht auf Hardwareebene zur Laufzeit gegeben, sondern alleine auf Quelltextebene. Somit wird für jedes zu nutzende Design bei Verwendung von **Hochsprachen eine erneute Spezifikation, Testen und Verifizieren auf Hochebene nötig, anschließend die zeitaufwendige Synthese** der Hardwareschaltung und abschließend der Test der Hardware. Benötigt eine Anwendung dann auch noch zwei oder mehr unterschiedliche mit einer Hochsprache erstellte FPGA-Designs, so fällt zur Laufzeit zusätzlich eine Rekonfigurationsdauer von etwa 100 ms an.

Aus diesen Gründen wurde ein mikroprogrammierbares, pufferzentriertes, datengetriebenes, taskparalleles Rahmenwerk entwickelt (Abschnitt 6.4). Die Programmierung und Verwendung seitens eines Anwendungsentwicklers erfolgen ausschließlich über das Laden und Ausführen von Mikroprogrammen. In den Mikroprogrammen wird angegeben, was welche Einheit ausführen soll und an welche weiteren Einheiten die Daten geschrieben werden sollen. Das Lesen und Schreiben ist über ein zentrales Pufferset voneinander entkoppelt. Jede Einheit hat fest zugewiesene Eingänge. Für das Schreiben in die Eingangspuffer anderer Einheiten ist eine Crossbar implementiert und dadurch sehr gute Konfigurierbarkeit gegeben im Gegensatz zu einer busbasierten Schnittstelle. Funktionseinheiten sind lauffähig, sobald ein gültiger Befehl vorliegt und mindestens ein gültiges Datum in jedem Eingangspuffer vorhanden ist (datengetriebene Ausführung). Auf diese Art können mehrere Einheiten voneinander unabhängig laufen (Task-Parallelismus). Datenabhängigkeiten führen durch die Puffer zu keinem Konflikt und somit wird Pipeline-Parallelismus ausnutzbar. Für größere Datenmengen, aber bereits weit bevor die Kapazitäten der Prozessorcaches erschöpft sind, kann die Convey HC-1 mittels **dieses Rahmenwerks** und effizienter Stencilimplementierungen, Vorkonditionierer und genauerer Skalarproduktimplementierungen **ein System mit 24 Hardwarethreads und großer Speicherbandbreite übertreffen**, wie in Abschnitt 9.2 evaluiert [NBK+13].

Es ist jedoch nicht unbedingt notwendig, stets ein exaktes Skalarprodukt zu verwenden. Gleichermaßen müssen Zufallszahlen nicht immer echt zufällig sein, sondern es können Pseudozufallszahlen mit besserer oder gar schlechterer Verteilung ausreichend sein. Insgesamt kann durch die geschickte Auswahl der Implementierungen bzw. Hardware die Laufzeit mitunter wesentlich verringert werden. Um die Auszeichnung von Funktionen und Implementierungen sowie die Angabe der Anforderungen seitens des Anwendungsentwicklers zu unterstützen, wurde das Konzept der Attributierung entwickelt [NBK09], wie es in Abschnitt 7.1 vorgestellt wird. Die **Attribute der Anforderungen bei Funktionsaufrufen und der Eigenschaften der Implementierungen** werden dabei in den ausführbaren ELF[3]-Dateien gespeichert und sind dadurch portabel zwischen unterschiedlichen Linux-basierten Systemen [NKBK10] (Abschnitt 7.2). Ein Laufzeitsystem ermöglicht dann, die Attribute der Anforderungen und Implementierungen zur Laufzeit abzugleichen und dadurch die jeweils passendste Implementierung und Hardware auszuwählen [KNBK12]. Es misst darüberhinaus die einzelnen Läufe und speichert die Ausführungszeiten sowie die dabei verwendeten Funktionsparameter in einer systemzentralen Datenbank ab.

In Abhängigkeit davon, wie die Hardware-Software-Schnittstelle vom FPGA-Hersteller implementiert wurde, erfolgt die Synchronisation und Kommunikation des Hostprozessors mit dem Koprozessor über geschäftiges Warten[4], so dass ein Rechenkern belegt ist. Bei aktuellen Mehrkernprozessoren lohnt es sich entsprechend, für die weiteren, unbenutzten Kerne Arbeit zu suchen. Im Falle von HHblits ist dies die zweite Stufe der Vorfilterung, die mit der Koprozessorausführung der ersten

[3] Executable and Linkable Format

[4] engl. busy waiting

Vorfilterungsstufe überlappt abgearbeitet werden kann. Für numerische Anwendungen kann Nutzen durch datengetriebene Ausführung entstehen, da gerade bei Matrizen solcher Dimensionen, wo die Daten nicht mehr vollständig in den Cache passen, in den Cache geschriebene Daten vorab der Verdrängung wiederverwendet werden können. Problematisch bei der Umsetzung in Abschnitt 8.1 eines derart feingranularen datenflussorientierten Programmiermodells in Software ist der Aufwand für die Kommunikation und Synchronisation der Arbeiterthreads. Einfache Ansätze wie das Verringern der Anzahl an Nachrichten führen bereits zu Nutzen dieses Programmiermodells gegenüber datenparalleler Programmierung. Insgesamt ist die **datengetriebene Programmierung mit Komponenten**, evaluiert am Beispiel numerischer und bioinformatischer Anwendungen, der vorgeschlagene Ansatz, um **cacheeffizient, abstrakt, konstruktiv und effektiv** Anwendungen zu entwickeln, die, in Teilen oder ganz, **in Software oder (rekonfigurierbarer) Hardware** ausgeführt werden können entsprechend der Eignung der vorliegenden Implementierungen und der vom Domänenenexperten angegebenen Anforderungen der Anwendung.

Komponentenbasierte Entwicklung ist bekannt dafür, **den Entwicklungsaufwand zu senken**, da die Implementierungen bereits vorhanden sind und da das Verhalten für unterschiedliche Implementierungen gleich zu sein hat. Im Falle des Rahmenwerks in rekonfigurierbarer Hardware wird der Entwicklungsaufwand also einerseits dadurch gesenkt, dass anstelle einer vollständigen Ausformulierung jedes Details nur noch Komponenten aufgerufen werden müssen (vgl. Listing 9.1 in Abschnitt 9.1.1), wie es vorab auch in Software entwickelt und getestet werden kann. Andererseits wird er dadurch reduziert, dass die Hardwarebeschreibung nicht für jede neue Anwendung zeitaufwendig neu synthetisiert werden muss, da **Komponenten wiederverwendbar** sind. Beispielsweise sind die Komponenten, die in Abschnitt 9.2 verwendet wurden, für zahlreiche andere Algorithmen einsetzbar. Ebenso sind Komponenten und die Mikroprogramme zur Ansteuerung wiederverwendbar in anderen Instanzen des Rahmenwerks auf anderen Systemen; der Ansatz ist also auch wie gewünscht **portierbar**, was anhand der Implementierungen auf dem UoH HTX-Board und auf der Convey HC-1 in Abschnitt 6.4.4 ersichtlich wird. Wie Abschnitt 9.2.3 insbesondere zeigt, sind die Komponenten über die Parametrierung des Aufrufs hinaus offline, statisch **adaptierbar**, wobei die Adaption wiederum vom Hardwarespezialisten vorgenommen wird und neue Synthesen bedingt.

10.2 Ausblick und Perspektive

Zukünftige Mehr- oder Vielkernprozessoren werden mit großer Wahrscheinlichkeit netzwerkartig mit Gitterstruktur aufgebaut und organisiert sein, jeweils mit Routern an den Funktions- und Speichereinheiten, um Datenpakete zu empfangen, weiterzusenden oder selbst zu verschicken. Die Funktionseinheiten an den Gitterknoten werden untereinander heterogen sein. Die Verwendung und Kombination mehrerer Funktionseinheiten zu einer größeren Gesamteinheit wird nötig sein, um eine Aufgabe schnell und korrekt bearbeiten oder eine gewisse Funktionalität stellen zu können. Beispiele dafür sind die Kombination von Allzweckrechenkernen zu SIMD-Einheiten oder die Implementierung von 3DES als drei DES-Einheiten auf drei FPGA-Ressourcen.

Es ist davon auszugehen, dass Prozessorkerne zunehmend die Aufgabe einer Kontrolleinheit für weitere Ausführungseinheiten übernehmen werden, da zum einen mehr effiziente[5] Kommunikation und Synchronisation nötig sein wird, und zum anderen weniger Parallelismus auf Befehlsebene ausnutzbar sein wird [14]. Mit der vorliegenden Arbeit wurde ein wichtiger weiterer Grundbaustein dazu gelegt, zukünftig wissenschaftliche, numerische Anwendungen grafisch gestützt für heterogene Systeme entwickeln zu können. Domänenexperten verfügen über hohe Fähigkeiten, mit zur Verfügung gestellten Programmierbibliotheken Anwendungen umzusetzen. Es wurde eine für das CG-Verfahren und das Lanczos-Verfahren ausreichende datengetriebene Programmierbibliothek entwickelt, deren Verwendung direkt der Verbindung von Funktionsblöcken entspricht, die grafisch darstellbar sind. Die Funktionsblöcke können als Komponenten in Hardware oder in Software ausgeführt werden. Zur Spezifikation besonderer Anforderungen können Attribute in einer solchen grafisch orientierten Programmierumgebung angegeben werden.

[5] aufwandsarm, durchsatzstark, mit geringen Verzögerungen

Integriert man im datengetriebenen, mikroprogrammierbaren, task-parallelen FPGA-Rahmenwerk, das die Ausführung der Funktionsblöcke in Hardware umsetzt, Schnittstellen zu schnellen Netzwerken oder optischen Verbindungen beispielsweise zu weiteren FPGA-Karten hin, so lässt sich eine „Remote Accelerator Architecture“[6] aufbauen, bei der die Latenz der Verbindungen hervorragend mittels der puffergestützten Transfers versteckt werden kann. Dadurch können auf den entfernten FPGA-Karten größere, komplexere Funktionseinheiten genutzt werden, die nicht mehr auf den FPGA mit dem Rahmenwerk passen.

Der Nutzen der Attributierung wurde bereits innerhalb eines neuartigen Laufzeitsystems für heterogene Mehr- und Vielkernsysteme evaluiert. Die entwicklergestützte Auswahl passender Implementierungen kann die Ausführungszeit wesentlich verbessern. Das Laufzeitsystem misst dazu die Ausführungszeit der vom Entwickler ausgewählten Implementierungen unter Berücksichtigung der Parameter wie z.B. der Datenmenge und wählt später die anzunehmenderweise beste Implementierung. Es ist denkbar, dass anhand der grafisch entwickelten datengetriebenen Programme automatisiert unterschiedliche Teile als Mikroprogramm synthetisiert und auf FPGA-Koprozessoren ausgeführt werden. Dies ist vor dem Hintergrund zu sehen, dass wissenschaftliche Anwendungen oftmals viele Stunden Laufzeit haben und daher einige Millionen Funktionsaufrufe durchführen, so dass die Wahl einer auch nur geringfügig besseren Aufteilung in Hardware- und Software-Ausführung großen Nutzen liefern kann.

Das entwickelte datengetriebene Rahmenwerk ist nicht ausschließlich für FPGAs geeignet oder gar bestimmt. Zum Zeitpunkt der Untersuchungen war der Bibliotheks-Code von Nvidia CUDA noch nicht zugänglich und mit dem damals vorliegenden CUDA das Streamen und Verarbeiten von Daten auf feingranular datengetriebene Weise nicht möglich, sondern die Eingangsdaten mussten vollständig vorliegen, bevor ein Stream gestartet werden konnte. Die einzige Alternative ist das feingranulare Kopieren einzelner Daten, Aufrufen der GPU zum Durchführen der Operation auf diesen Daten bei zeitgleicher Übertragung der nächsten Arbeitsdaten, und dann das erneute Aufrufen usw., wodurch kein Nutzen erzielbar ist. Zukünftige Arbeiten sollten unbedingt GPUs betrachten, da diese mitunter energieeffizienter als FPGAs arbeiten und bei Anwendungsteilen mit wenig Kontrollfluss ihre hohe Anzahl an Threads und Verarbeitungseinheiten ausspielen können bei 10-20 mal höherer Taktrate als FPGAs.

Application-Specific Integrated FPGAs (ASIFs) [214] bilden eine interessante Alternative zu FPGAs und GPUs vor dem Hintergrund der Nutzung von leistungsfähigen, energieeffizienten Koprozessoren, die – wie sich in der vorliegenden Dissertation gezeigt hat – ohnehin vom Hardwareexperten entwickelt werden müssen für eine bestimmte Anwendungsdomäne. Das Ziel der ASIFs ist es nämlich, möglichst viel Logik auf möglichst wenige, zeitlich getrennt genutzte Hardwareblöcke zu bringen, indem die Blöcke zur Laufzeit parametriert werden und die Verbindungen zwischen Blöcken mittels Multiplexern unterschiedlich geroutet werden, so dass eine Vielzahl von Schaltungen mit der vorliegenden Hardware umgesetzt werden kann entsprechend der Parametrierung. Damit sind keine teuren, langsamen FPGAs als Koprozessoren mehr nötig, sondern können durch ASIC-ähnliche Hardware ersetzt werden. Das entwickelte Programmier- und Ausführungsmodell lässt sich hervorragend mit dem Ansatz der ASIFs kombinieren, indem die Funktionsblöcke selbst mit Parametern versehen werden und somit zur Ausführungszeit eine heterogene MIMD-Einheit entsteht, die sowohl in SIMD-Manier datenparallel als auch task-parallel und pipeline-parallel arbeiten kann.

10.3 Erkenntnisse bezüglich der Anforderungen an zukünftige heterogene Systeme

In dieser Arbeit wurde gezeigt, dass die datengetriebene Ausführung einen erfolgsversprechenden Weg darstellt, wie Koprozessoren eingebunden, Mehrkernprozessoren ausgenutzt und heterogene Systeme effizient verwendet werden können. Problematisch ist stets der Datentransfer zwischen Recheneinheiten. Die Lösung dazu ist gemeinsamer physischer Speicher, so dass zusätzliche Kopieroperationen, wie sie bei dem FPGA-basierten Koprozessor auf der Convey HC-1 implizit oder explizit gemacht werden müssen zwischen den unterschiedlichen physischen Speichern, entfallen.

[6] engl., Architektur mit entfernten Akzeleratoren

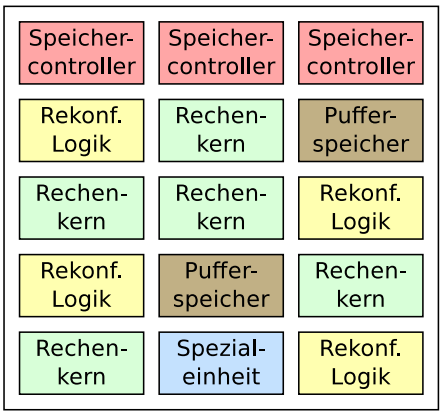


Abbildung 10.1: Skizze eines zukünftigen heterogenen Manycore-Prozessors.

Aufgrund von Kohärenzproblemen zwischen mehreren physischen Speichern sind dazu alternative, neuartige Speicherkonstrukte einzuführen. Ein Ansatz ist, zwischen Recheneinheiten über FIFO-Speicher zu kommunizieren. Die Alternative dazu ist, die verwendeten Synchronisationskonstrukte mit Gültigkeit und Anzahl berechneter Daten aus der Softwarevariante des datengetriebenen Programmiermodells in Hardwareschaltungen zu übertragen, die atomar gelesen und beschrieben werden. Transactional Memory bietet dahingehend bereits einige Lösungen und weitere Fortschritte an.

Zukünftige heterogene Manycore-Prozessoren sollten aufgrund der insgesamt möglichen energieeffizienteren Ausführung unbedingt rekonfigurierbare Logik integrieren. Dies verspricht beispielsweise 27-fach energieeffizientere Ausführung des CG-Verfahrens. Um rekonfigurierbare Logik bequem programmierbar zu halten, bietet sich das entwickelte mikroprogrammierbare, datengetriebene Rahmenwerk an.

Teilweise fällt der Nutzen der Verwendung rekonfigurierbarer Hardware jedoch geringer aus wie im Fall von HHblits. Der Nutzen lässt sich jedoch erhöhen, indem task- bzw. pipeline-parallel gearbeitet wird, beispielsweise die zwei Vorfilterungsstufen überlappt zueinander in rekonfigurierbarer Hardware und in Software auf dem Prozessor. Beim CG-Verfahren ermöglicht die pipeline-parallele Ausführung überhaupt erst die energieeffiziente Ausführung. Dazu sind allerdings viele und besondere Speicherstrukturen nötig. Diese müssen daher ebenso in kommenden Manycore-Prozessoren angeboten werden, um den Pipeline-Parallelismus zwischen Funktionseinheiten, welche Komponenten darstellen oder solche ausführen, anbieten und ausnutzen zu können.

Abbildung 10.1 skizziert abschließend einen möglichen, anhand der in der vorliegenden Dissertation gewonnenen Kenntnisse entworfenen heterogenen Manycore-Prozessor. Eingezeichnet sind dem Stand der Technik entsprechend mehrere Speichercontroller, mehrere Rechenkerne und vertretungsweise eine Spezialeinheit wie etwa eine SSE-/AVX-Einheit. Ferner sind mehrere Flächen mit rekonfigurierbarer Logik in der Skizze enthalten, welche datengetrieben Komponenten ausführen könnten. Die datengetriebene Ausführung wird über mehrere Pufferspeicher unterstützt. Sie sollen sowohl von Komponenten in rekonfigurierbarer Logik als auch von auf den Rechenkernen ausgeführten Komponenten zur Kommunikation genutzt werden.

Glossar

AE Application Engine. 53

AEH Application Engine Hub. 18, 19

API Application Programming Interface. 17

ASIC Application-Specific Integrated Circuit. 9

BCD Binary Coded Decimal. 22

BRAM Block RAM. 8

CDFG Kontrolldatenflussgraph. 13

CGRA Coarse-Grained Reconfigurable Array. 34, 35, 123, 124, 148

CLB Configurable Logic Block. 8

CPI Cycles per Instruction. 82

CSP Communicating Sequential Processes. 17, 43, 101, 102

CUDA Compute Unified Device Architecture. 7

DSP Digital Signal Processing. 8, 9

FDM Finite-Differenzen-Methode. 96

FPGA Field-Programmable Gate Array. 1–3, 8–11

FSB Front-Side Bus. 15, 18, 19

GPGPU General-Purpose Graphics-Processing Unit. 5, 8

GPU Graphics-Processing Unit. 2

ICAP Internal Configuration Access Port. 9

IOB IO Block. 9

ISA Instruction Set Architecture. 6

JVM Java Virtual Machine. 37

LUT Look-Up Table. 8, 13

MAC Multiply-Accumulate. 59, 116

MC Memory Controller. 50

MMU Memory Management Unit. 19

MPPA Multi-Purpose Processor Array. 35

NoC Network on Chip. 14

PGAS Partitioned Global Address Space. 15, 16

PSP Platform Support Package. 17, 107

QPI QuickPath Interconnect. 15

ROCCC Riverside Optimizing Compiler for Configurable Computing. 106, 107

ROQ Read-Order Queue. 51, 55

SERDES Serializer-Deserializer. 9

SMT Simultaneous Multi-Threading. 5

SoC System on Chip. 6, 14

SSE SIMD Streaming Extensions. 5

TLB Translation-Lookaside Buffer. 19

Literaturverzeichnis

[1] Advanced Micro Devices, Inc.: *AMD Stream Computing Software Stack*. Online, Nov. 2007. – `http://www.cct.lsu.edu/~scheinin/Parallel/firestream-sdk-whitepaper.pdf`

[2] Advanced Micro Devices, Inc.: *Brook+ SC07 BOF Session*. Online, Nov. 2007. – `http://developer.amd.com/gpu_assets/AMD-Brookplus.pdf`

[3] Ahmadinia, Ali ; Fernandez-Canque, Hernando ; Ramirez-Iniguez, Roberto: Dynamic Reconfiguration in JPEG2000 Hardware Architecture. In: *Knowledge-Based and Intelligent Information and Engineering Systems* Bd. 6883, Springer-Verlag, 2011 (LNCS). – ISBN 978–3–642–23853–6, S. 453–461

[4] Ahmed, W. ; Shafique, M. ; Bauer, L. ; Henkel, J.: mRTS: Run-time system for reconfigurable processors with multi-grained instruction-set extensions. In: *Design, Automation & Test in Europe Conference & Exhibition*, 2011 (DATE '11). – ISSN 1530–1591, S. 1–6

[5] Alachiotis, Nikolaos ; Stamatakis, Alexandros: FPGA optimizations for a pipelined floating-point exponential unit. In: *7th conference on Reconfigurable Computing: Architectures, Tools and Applications*, Springer-Verlag, 2011 (ARC '11). – ISBN 978–3–642–19474–0, S. 316–327

[6] Allan, Benjamin A. ; Armstrong, Robert ; Bernholdt, David E. ; Bertrand, Felipe ; Chiu, Kenneth ; Dahlgren, Tamara L. ; Damevski, Kostadin ; Elwasif, Wael R. ; Epperly, Thomas G. W. ; Govindaraju, Madhusudhan ; Katz, Daniel S. ; Kohl, James A. ; Krishnan, Manoj ; Kumfert, Gary ; Larson, J. W. ; Lefantzi, Sophia ; Lewis, Michael J. ; Malony, Allen D. ; Mclnnes, Lois C. ; Nieplocha, Jarek ; Norris, Boyana ; Parker, Steven G. ; Ray, Jaideep ; Shende, Sameer ; Windus, Theresa L. ; Zhou, Shujia: A Component Architecture for High-Performance Scientific Computing. In: *International Journal of High Performance Computing Applications, Sage Publications* 20 (2006), Nr. 2, S. 163–202. – ISSN 1094–3420

[7] Alt, T. ; Lindenstruth, V. ; Steinbeck, T. ; Tilsner, H. ; Wiebalck, A. ; Appelshäuser, H. ; Loizides, C. ; Becker, B. ; Cleyrnans, J. ; De Vaux, G. ; Fearick, R. W. ; Szostak, A. ; Vilakazi, Z. Z. ; Chattopadhyay, S. ; Cheshkov, C. ; Cicaló, C. ; Helstrup, H. ; Richter, M. ; Röhrich, D. ; Ullaland, K. ; Vestbø, A. ; Skaali, B. ; Vik, T. ; Staley, F.: Benchmarks and implementation of the ALICE high level trigger. In: *14th IEEE-NPSS Real Time Conference*, IEEE Computer Society, 2005 (RT '05). – ISBN 0–7803–9183–7, S. 320–324

[8] Altschul, Stephen F. ; Gish, Warren ; Miller, Webb ; Myers, Eugene W. ; Lipman, David J.: Basic local alignment search tool. In: *Journal of Molecular Biology, Elsevier* 215 (1990), Nr. 3, S. 403–410. – ISSN 0022–2836

[9] Amaricai, A. ; Vladutiu, M. ; Prodan, L. ; Udrescu, M. ; Boncalo, O.: Design of Addition and Multiplication Units for High Performance Interval Arithmetic Processor. In: *Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 13. Nov. 2007 (DDECS '07). – ISBN 1–4244–1161–0, S. 1 –4

[10] Ament, Marco ; Knittel, Gunter ; Weiskopf, Daniel ; Strasser, Wolfgang: A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In: *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, IEEE Computer Society, Feb. 2010 (Euromicro 2010). – ISBN 978–0–7695–3939–3, S. 583–592

[11] Andersch, M. ; Juurlink, B. ; Chi, C. C.: A Benchmark Suite for Evaluating Parallel Programming Models. In: *Mitteilungen* Bd. 28, Gesellschaft für Informatik, Parallel-Algorithmen und Rechnerstrukturen, Oktober 2011 (PARS '11). – ISSN 0177–0454, S. 7–17

[12] Andrews, David ; Niehaus, Douglas ; Jidin, Razali ; Finley, Michael ; Peck, Wesley ; Frisbie, Michael ; Ortiz, Jorge ; Komp, Ed ; Ashenden, Peter: Programming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. In: *IEEE Micro* 24 (2004), Juli, Nr. 4, S. 42–53. – ISSN 0272–1732

[13] Araya-Polo, M. ; Cabezas, J. ; Hanzich, M. ; Pericas, M. ; Rubio, F. ; Gelado, I. ; Shafiq, M. ; Morancho, E. ; Navarro, N. ; Ayguade, E. ; Cela, J.M. ; Valero, M.: Assessing Accelerator-Based HPC Reverse Time Migration. In: *IEEE Transactions on Parallel and Distributed Systems* 22 (2011), Jan., Nr. 1, S. 147–162. – ISSN 1045–9219

[14] Arora, Manish ; Nath, Siddhartha ; Mazumdar, Subhra ; Baden, Scott B. ; Tullsen, Dean M.: Redefining the Role of the CPU in the Era of CPU-GPU Integration. In: *IEEE Micro* 32 (2012), Nov./Dez., Nr. 6, S. 4–16. – ISSN 0272–1732

[15] Augustin, Werner ; Weiss, Jan-Philipp ; Heuveline, Vincent: Convey HC-1 Hybrid Core Computer – The Potential of FPGAs in Numerical Simulation. In: *2nd International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, KIT Scientific Publishing, Karlsruhe, Feb. 2011 (HipHaC '11). – ISBN 978–3–86644–626–7, S. 1–8

[16] Ayguadé, Eduard ; Badia, Rosa M. ; Cabrera, Daniel ; Duran, Alejandro ; González, Marc ; Igual, Francisco D. ; Jimenez, Daniel ; Labarta, Jesús ; Martorell, Xavier ; Maio, Rafael ; Pérez, Josep M. ; Quintana-Ortí, Enrique S.: A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In: *Evolving OpenMP in an Age of Extreme Parallelism, 5th International Workshop on OpenMP, (IWOMP 2009)* Bd. 5568, Springer-Verlag, Juni 2009 (LNCS). – ISBN 978–3–642–02284–5, S. 154–167

[17] Azevedo, Rodolfo ; Rigo, Sandro ; Bartholomeu, Marcus ; Araujo, Guido ; Araujo, Cristiano ; Barros, Edna: The ArchC architecture description language and tools. In: *International Journal on Parallel Programming, Kluwer Academic Publishers* 33 (2005), Oktober, Nr. 5, S. 453–484. – ISSN 0885–7458

[18] Bailey, David H.: High-Precision Floating-Point Arithmetic in Scientific Computation. In: *IEEE Computing in Science and Engineering* 7 (2005), Mai, S. 54–61. – ISSN 1521–9615

[19] Bailey, David H. ; Hida, Yozo ; Jeyabalan, Karthik ; Li, Xiaoye S. ; Thompson, Brandon: *QD*. Okt. 2012. – Webseite: `http://crd.lbl.gov/~dhbailey/mpdist/`

[20] Bakos, Jason D. ; Nagar, Krishna K.: Exploiting Matrix Symmetry to Improve FPGA-Accelerated Conjugate Gradient. In: *17th Symposium on Field Programmable Custom Computing Machines*, IEEE Computer Society, 2009 (FCCM '09). – ISBN 978–0–7695–3716–0, S. 223–226

[21] Batory, D.: Multilevel models in model-driven engineering, product lines, and metaprogramming. In: *IBM Systems Journal – Model-driven software development* 45 (2006), Juli, Nr. 3, S. 527–539. – ISSN 0018–8670

[22] Bauer, Lars: *RISPP: A Run-time Adaptive Reconfigurable Embedded Processor*, Universität Karlsruhe (TH), Fakultät für Informatik (INFORMATIK), Institut für Technische Informatik (ITEC), Dissertation, 2009

[23] Bauer, Lars ; Shafique, Muhammad ; Henkel, Jörg: Run-time instruction set selection in a transmutable embedded processor. In: *45th Design Automation Conference*, ACM, 2008 (DAC '08). – ISBN 978–1–60558–115–6, S. 56–61

[24] Bauer, Lars ; Shafique, Muhammad ; Kramer, Simon ; Henkel, Jörg: RISPP: rotating instruction set processing platform. In: *Proceedings of the 44th annual Design Automation Conference*. New York, NY, USA : ACM, 2007 (DAC '07). – ISBN 978–1–59593–627–1, S. 791–796

[25] Baumgarte, V. ; Ehlers, G. ; May, F. ; Nückel, A. ; Vorbach, M. ; Weinhardt, M.: PACT XPP–A Self-Reconfigurable Data Processing Architecture. In: *The Journal of Supercomputing, Kluwer Academic* 26 (2003), September, Nr. 2, S. 167–184. – ISSN 0920–8542

[26] Becker, Jürgen ; Brändle, Kurt ; Brinkschulte, Uwe ; Henkel, Jörg ; Karl, Wolfgang ; Köster, Thorsten ; Wenz, Michael ; Wörn, Heinz: Digital On-Demand Computing Organism for Real-Time Systems. In: *Workshops Proceedings of the 19th International Conference on Architecture of Computing Systems (ARCS' 06)* Bd. P81, Gesellschaft für Informatik, März 2006 (Lecture Notes in Informatics). – ISBN 3–88579–175–7, S. 230–245

[27] Bellens, Pieter ; Perez, Josep M. ; Badia, Rosa M. ; Labarta, Jesus: CellSs: a Programming Model for the Cell BE Architecture. In: *Conference on Supercomputing*, ACM, Nov. 2006 (SC '06). – ISBN 0–7695–2700–0, S. 5

[28] Benabderrahmane, M.-W. ; Pouchet, L.-N. ; Cohen, A. ; Bastoul, C.: The Polyhedral Model Is More Widely Applicable Than You Think. In: *Proceedings of the International Conference on Compiler Construction (ETAPS CC '10)* Bd. 6011/2010, Springer-Verlag, März 2010 (LNCS). – ISBN 978–3–642–11969–9, S. 283–303

[29] Berkeley Design Technology, Inc.: *An Independent Analysis of Altera's FPGA Floating-point DSP Design Flow.* online. `http://eejournal.com/archives/on-demand/2011091201_altera/`. Version: Sep. 2011

[30] Beuchat, Jean-Luc ; Miyoshi, Takanori ; Oyama, Yoshihito ; Okamoto, Eiji: Multiplication over $\mathbb{F}_{p^m}$ on FPGA: A Survey. In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC '07)* Bd. 4419, Springer-Verlag, 2007 (LNCS). – ISBN 978–3–540–71430–9, S. 214–225

[31] Bierlox, Norbert: *Ein VHDL Koprozessorkern für das exakte Skalarprodukt*, Universität Karlsruhe, Diss., Nov. 2002

[32] Birk, Matthias ; Hagner, Clemens ; Balzer, Matthias ; Ruiter, Nicole ; Hübner, Michael ; Becker, Jürgen: First Evaluation of FPGA Reconfiguration for 3D Ultrasound Computer Tomography. In: *5th Workshop on Reconfigurable Communication-centric Systems on Chip 2010 (ReCoSoC '10)* Bd. 7551, KIT Scientific Publishing, Mai 2010 (KIT scientific reports). – ISBN 978–3–86644–515–4, 109–114

[33] Blomquist, Frithjof ; Hofschuster, Werner ; Krämer, Walter: A Modified Staggered Correction Arithmetic with Enhanced Accuracy and Very Wide Exponent Range. In: Cuyt, Annie (Hrsg.) ; Krämer, Walter (Hrsg.) ; Luther, Wolfram (Hrsg.) ; Markstein, Peter (Hrsg.): *Numerical Validation in Current Hardware Architectures.* Springer-Verlag, 2009. – ISBN 978–3–642–01590–8, S. 41–67

[34] Blumofe, Robert D. ; Joerg, Christopher F. ; Kuszmaul, Bradley C. ; Leiserson, Charles E. ; Randall, Keith H. ; Zhou, Yuli: Cilk: An Efficient Multithreaded Runtime System. In: *Journal of Parallel and Distributed Computing, Elsevier* 37 (1996), Nr. 1, S. 55–69. – ISSN 0743–7315

[35] Bode, Arndt: *Informatik-Fachberichte.* Bd. 82: *Mikroarchitekturen und Mikroprogrammierung: Formale Beschreibung und Optimierung: Formale Beschreibung und Optimierung*. Berlin Heidelberg : Springer-Verlag, 1984. – ISBN 978–3540133803

[36] Bodin, François: *An Evolutionary Path for High Performance Heterogeneous Multicore Programming.* Online. `http://www.caps-entreprise.com/resources.html`. Version: Juni 2008. – Whitepaper

[37] Boehm, B.W.: Software Engineering. In: *IEEE Transactions on Computers* C-25 (1976), Nr. 12, S. 1226–1241. – ISSN 0018–9340

[38] Bonorden, Olaf ; Heide, Friedhelm Meyer auf d. ; Wanka, Rolf: Composition of Efficient Nested BSP Algorithms: Minimum Spanning Tree Computation as an Instructive Example. In: *Conference on Parallel and Distributed Processing Techniques and Applications*, CSREA Press, Juni 2002 (PDPTA '02). – ISBN 1–892512–90–4, S. 2202–2208

[39] Boukerche, Azzedine ; Correa, Jan M. ; Melo, Alba Cristina Magalhaes A. ; Jacobi, Ricardo P. ; Rocha, Adson F.: An FPGA-Based Accelerator for Multiple Biological Sequence Alignment with DIALIGN. In: *14th International Conference on High Performance Computing (HiPC '07)* Bd. 4873, Springer-Verlag, Dezember 2007 (LNCS). – ISBN 978–3–540–77219–4, S. 71–82

[40] Bouwens, Frank ; Berekovic, Mladen ; Kanstein, Andreas ; Gaydadjiev, Georgi: Architectural exploration of the ADRES coarse-grained reconfigurable array. In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC '07)* Bd. 4419, Springer-Verlag, 2007 (LNCS). – ISBN 978–3–540–71430–9, S. 1–13

[41] Bridges, Matthew ; Vachharajani, Neil ; Zhang, Yun ; Jablin, Thomas ; Aug., David: Revisiting the Sequential Programming Model for Multi-Core. In: *40th International Symposium on Microarchitecture*, IEEE Computer Society, 2007 (MICRO-40 2007). – ISBN 0–7695–3047–8, S. 69–84

[42] Brüning, Ulrich: *The HTX board – a universal HTX test platform*. Online. `http://www.hypertransport.org/members/u_of_man/htx_board_data_sheet_UoH.pdf`. – Data Sheet

[43] Brüning, Ulrich et a.: *The UoM non-coherent HT Cave Core*. 2005. – `http://www.hypertransport.org/products/productdetail.cfm?RecordID=83`

[44] Buchty, Rainer ; Kramer, David ; Kicherer, Mario ; Karl, Wolfgang: A Light-weight Approach to Dynamical Runtime Linking Supporting Heterogenous, Parallel, and Reconfigurable Architectures. In: Berekovic, Mladen (Hrsg.) ; Müller-Schloer, Christian (Hrsg.) ; Hochberger, Christian (Hrsg.) ; Wong, Stephan (Hrsg.): *22nd Conference on Architecture of Computing Systems (ARCS '09)* Bd. 5455, Springer-Verlag, März 2009 (LNCS). – ISBN 978–3–642–00453–7, S. 60–71

[45] Buchty, Rainer ; Mattes, Oliver ; Karl, Wolfgang: Self-aware Memory: Managing Distributed Memory in an Autonomous Multi-Master Environment. In: *21st Conference on Architecture of Computing Systems (ARCS '08)* Bd. 4934, 2008 (LNCS). – ISBN 978–3–540–78152–3, S. 98–113

[46] Burke, D. ; Wawrzynek, J. ; Asanovic, K. ; Krasnov, A. ; Schultz, A. ; Gibeling, G. ; Droz, P.-Y.: RAMP Blue: Implementation of a Multicore 1008 Processor FPGA System. In: *Proceedings of the Fourth Annual Reconfigurable Systems Summer Institute*, 2008 (RSSI '08). – Online: `http://rssi.ncsa.illinois.edu/proceedings/papers/presentations/19_Burke.pdf`

[47] Buyukkurt, Betul ; Cortes, John ; Villarreal, Jason ; Najjar, Walid A.: Impact of high-level transformations within the ROCCC framework. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 7 (2010), Dez., S. 17:1–17:36. – ISSN 1544–3566

[48] Buyukkurt, Betul ; Guo, Zhi ; Najjar, Walid A.: Impact of Loop Unrolling on Area, Throughput and Clock Frequency in ROCCC: C to VHDL Compiler for FPGAs. In: *Second Workshop on Reconfigurable Computing: Architectures and Applications (ARC '06)* Bd. 3985, Springer-Verlag, März 2006 (LNCS). – ISBN 3–540–36708–X, S. 401–412

[49] Cabrera, Daniel ; Martorell, Xavier ; Gaydadjiev, Georgi ; Ayguade, Eduard ; Jiménez-González, Daniel: OpenMP extensions for FPGA accelerators. In: *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, IEEE, 2009 (IC-SAMOS '09). – ISBN 978–1–4244–4502–8, S. 17–24

[50] Canny, John: A Computational Approach to Edge Detection. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8 (1986), Nr. 6, S. 679–698. – ISSN 0162–8828

[51] Chai, Sek ; Bellas, Nikolaos ; Lopez-Lagunas, Abelardo: Extending a Stream Programming Paradigm to Hardware Accelerator Platforms. In: *Symposium on Application Accelerators in High Performance Computing*, 2009 (SAAHPC '09). – `http://saahpc.ncsa.illinois.edu/papers/Chai_paper.pdf`

[52] Chamizo, J.M.G. ; Pascual, J.M. ; Mora, H.G.: Exact numerical processing. In: *Euromicro Symposium on Digital System Design*, IEEE Computer Society, Sep. 2003 (DSD '03). – ISBN 0–7695–2003–0, S. 434–437

[53] Chandramowlishwaran, Aparna ; Knobe, Kathleen ; Vuduc, Richard: Applying the Concurrent Collections Programming Model to Asynchronous Parallel Dense Linear Algebra. In: *15th Symposium on Principles and Practice of Parallel Programming*, ACM, 2010 (PPoPP '10). – ISBN 978–1–60558–877–3, S. 345–346

[54] Chavarría-Miranda, Daniel ; Nieplocha, Jarek ; Gorton, Ian: Hardware-accelerated Components for Hybrid Computing Systems. In: *Proceedings of the 2008 compFrame/HPC-GECO workshop on Component-Based High Performance Computing*, ACM, 2008 (CBHPC '08). – ISBN 978–1–60558–311–2, S. 1–8

[55] Chellappa, Srinivas ; Franchetti, Franz ; Püschel, Markus: Computer generation of fast fourier transforms for the cell broadband engine. In: *Proceedings of the 23rd international conference on Supercomputing*, ACM, 2009 (ICS '09). – ISBN 978–1–60558–498–0, S. 26–35

[56] Chisnall, David: *Introducing OpenCL*. Online, Juli 2010. – `http://www.informit.com/articles/article.aspx?p=1606232&WT.rss_f=Article&WT.rss_a=Introducing%20OpenCL&WT.rss_ev=a`

[57] Computer Architecture and Parallel Systems Laboratory (CAPSL): *Open64*. 2006–2012. – `http://www.open64.net/` (Webseite)

[58] Cong, J. ; Liu, Bin ; Neuendorffer, S. ; Noguera, J. ; Vissers, K. ; Zhang, Zhiru: High-Level Synthesis for FPGAs: From Prototyping to Deployment. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 30 (2011), April, Nr. 4, S. 473–491. – ISSN 0278–0070

[59] Cong, Jason ; Huang, Muhuan ; Zou, Yi: Accelerating Fluid Registration Algorithm on Multi-FPGA Platforms. In: *21st Conference on Field Programmable Logic and Applications*, IEEE Computer Society, Sep. 2011 (FPL '11). – ISBN 978–0–7695–4529–5, S. 50–57

[60] Convey Computer Corporation: *Convey Computer Proteomics Search Personality*. 2009. – `http://conveycomputer.com/Resources/Proteomics%20Application%20Brief.pdf` (Application Brief)

[61] Convey Computer Corporation: *Convey's hybrid-core technology: the HC-1 and the HC-1^{ex}*. Online, 2010. – Product Overview: `http://www.conveycomputer.com/Resources/Convey_HC1_Family.pdf`

[62] Convey Computer Corporation: *Hybrid-Core Computing for High Throughput Bioinformatics*. 2010. – `http://conveycomputer.com/Resources/ConveyBioinformatics_web.pdf` (White Paper)

[63] Convey Computer Corporation: *Convey BWA Data Sheet*. 2011. – `http://conveycomputer.com/Resources/BWADatasheet.pdf`

[64] Convey Computer Corporation: *Convey Computer GraphConstructor*. 2011. – `http://conveycomputer.com/Resources/ConveyGraphConstructor_datasheet_V_11_019.1CGCe.pdf` (Datasheet)

[65] Convey Computer Corporation: *Convey Computer Smith-Waterman Personality*. 2011. – `http://conveycomputer.com/Resources/ConveySmithWaterman6202011.pdf` (Datasheet)

[66] Convey Computer Corporation: *Convey's New Burrows-Wheeler Alignment Delivers 15x Increase in Research Efficiency*. 2011. – `http://conveycomputer.com/Resources/BWA.Final.NR.10.11.11-1.pdf` (Releae Notes)

[67] *Convey BLAST-N/P/X Personalities*. 2011

[68] Cook, Stephen A.: *On the Minimum Computation Time of Functions*, Diss., 1966

[69] Cornelius, Herbert: Accelerating HPC. In: *HPC Advisory Council Switzerland Conference 2012*, 2012. – http://hpcadvisorycouncil.com/events/2012/Switzerland-Workshop/Presentations/Day_3/2_INTEL.pdf

[70] Cox, Philip T. ; Gauvin, Simon: Controlled Dataflow Visual Programming Languages. In: *Proceedings of the 2011 Visual Information Communication - International Symposium*. New York, NY, USA : ACM, 2011 (VINCI '11). – ISBN 978–1–4503–0786–4, S. 9:1–9:10

[71] Czajkowski, T.S. ; Aydonat, U. ; Denisenko, D. ; Freeman, J. ; Kinsner, M. ; Neto, D. ; Wong, J. ; Yiannacouras, P. ; Singh, D.P.: From opencl to high-performance hardware on FPGAS. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 531–534

[72] Daigneault, M.-A. ; David, J.-P.: Raising the abstraction level of HDL for control-dominant applications. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 515–518

[73] Dandekar, Omkar ; Castro-Pareja, Carlos ; Shekhar, Raj: FPGA-based real-time 3D image preprocessing for image-guided medical interventions. In: *Journal of Real-Time Image Processing, Springer* 1 (2007), S. 285–301. – ISSN 1861–8200

[74] Danese, Giovanni ; Leporati, Francesco ; Bera, Marco ; Giachero, Mauro ; Nazzicari, Nelson ; Spelgatti, Alvaro: An Accelerator for Physics Simulations. In: *IEEE Computing in Science and Engineering* 9 (2007), Nr. 5, S. 16–25. – ISSN 1521–9615

[75] Dave, Nirav ; Fleming, Kermin ; King, Myron ; Pellauer, Michael ; Vijayaraghavan, Muralidaran: Hardware Acceleration of Matrix Multiplication on a Xilinx FPGA. In: *Proc. of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, IEEE Computer Society, 2007 (MEMOCODE '07). – ISBN 1–4244–1050–9, S. 97–100

[76] Davis, A. L.: Data Driven Nets - A class of maximally parallel, output-functional program schemata. 1974. – Forschungsbericht. – Burroughs IRC Report

[77] Davis, John D. ; Thacker, Charles P. ; Chang, Chen: BEE3: Revitalizing Computer Architecture Research / Microsoft Research. 2009 (MSR-TR-2009-45). – Forschungsbericht. – http://research.microsoft.com/pubs/80369/BEE3_TechReport.pdf

[78] Demmel, James W.: *Applied numerical linear algebra*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1997. – ISBN 0–89871–389–7

[79] Dennis, Jack B.: Retrospective: A Preliminary Architecture for a Basic Data Flow Processor. In: *25 years of the international symposia on Computer architecture (selected papers)*, ACM, 1998 (ISCA '98). – ISBN 1–58113–058–9, S. 2–4

[80] Dinechin, F. de ; Pasca, B. ; Cret, O. ; Tudoran, R.: An FPGA-specific approach to floating-point accumulation and sum-of-products. In: *Conference on Field-Programmable Technology*, IEEE Computer Society, Dez. 2008 (FPT '08). – ISBN 978–1–4244–2796–3, S. 33–40

[81] Dinechin, Florent de ; Klein, Cristian ; Pasca, Bogdan: Generating high-performance custom floating-point pipelines. In: *19th Conference on Field Programmable Logic and Applications*, IEEE, Aug. 2009 (FPL '09). – ISBN 978–1–4244–3892–1, S. 59–64

[82] Dinechin, Florent de ; Pasca, Bogdan: Designing Custom Arithmetic Data Paths with FloPoCo. In: *IEEE Design & Test of Computers* 28 (2011), Juli/Aug., Nr. 4, S. 18–27. – ISSN 0740–7475

[83] Doeffinger, Reimar: *Implementierung einer Recheneinheit für schnelle und genaue Vektoroperationen*. Karlsruhe, Deutschland, Universität Karlsruhe (TH), Lehrstuhl für Rechnerarchitektur und Parallelverarbeitung, Studienarbeit, 9. Mai 2008

[84] Dolbeau, Romain ; Bihan, Stéphane ; Bodin, François: HMPP: A Hybrid Multi-core Parallel Programming Environment. In: *Proceedings of First Workshop on General Purpose Processing on Graphics Processing Units*, 2007 (GPGPU1)

[85] Donaldson, Alastair F. ; Keir, Paul ; Lokhmotov, Anton: Compile-Time and Run-Time Issues in an Auto-Parallelisation System for the Cell BE Processor. In: *Euro-Par 2008 Workshops - Parallel Processing: VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, Aug. 25-26, 2008, Revised Selected Papers*, Springer-Verlag, 2009. – ISBN 978–3–642–00954–9, S. 163–173

[86] Dongarra, Jack ; Beckman, Pete ; Moore, Terry ; Aerts, Patrick ; Aloisio, Giovanni ; Andre, Jean-Claude ; Barkai, David ; Berthou, Jean-Yves ; Boku, Taisuke ; Braunschweig, Bertrand ; Cappello, Franck ; Chapman, Barbara ; Chi, Xuebin ; Choudhary, Alok ; Dosanjh, Sudip ; Dunning, Thom ; Fiore, Sandro ; Geist, Al ; Gropp, Bill ; Harrison, Robert ; Hereld, Mark ; Heroux, Michael ; Hoisie, Adolfy ; Hotta, Koh ; Jin, Zhong ; Ishikawa, Yutaka ; Johnson, Fred ; Kale, Sanjay ; Kenway, Richard ; Keyes, David ; Kramer, Bill ; Labarta, Jesus ; Lichnewsky, Alain ; Lippert, Thomas ; Lucas, Bob ; Maccabe, Barney ; Matsuoka, Satoshi ; Messina, Paul ; Michielse, Peter ; Mohr, Bernd ; Mueller, Matthias S. ; Nagel, Wolfgang E. ; Nakashima, Hiroshi ; Papka, Michael E. ; Reed, Dan ; Sato, Mitsuhisa ; Seidel, Ed ; Shalf, John ; Skinner, David ; Snir, Marc ; Sterling, Thomas ; Stevens, Rick ; Streitz, Fred ; Sugar, Bob ; Sumimoto, Shinji ; Tang, William ; Taylor, John ; Thakur, Rajeev ; Trefethen, Anne ; Valero, Mateo ; Van Der Steen, Aad ; Vetter, Jeffrey ; Williams, Peg ; Wisniewski, Robert ; Yelick, Kathy: The International Exascale Software Project roadmap. In: *International Journal of High Performance Computing Applications, Sage Publications* 25 (2011), Feb., Nr. 1, S. 3–60. – ISSN 1094–3420

[87] Dou, Yong ; Vassiliadis, S. ; Kuzmanov, G. K. ; Gaydadjiev, G. N.: 64-bit floating-point FPGA matrix multiplication. In: *13th Symposium on Field Programmable Gate Arrays*, ACM, 2005 (FPGA '05). – ISBN 1–59593–029–9, S. 86–95

[88] DRC Computer Corporation: *DRC Accelium Coprocessors – Ultra-High Performance Reconfigurable Processors*. Sunnydale, CA, USA : Online, 2008. – Product Datasheet: `http://drccomputer.com/pdfs/DRC_Accelium_Coprocessors.pdf`

[89] DuBois, David ; DuBois, Andrew ; Boorman, Thomas ; Connor, Carolyn: Non-Preconditioned Conjugate Gradient on Cell and FPGA Based Hybrid Supercomputer Nodes. In: *17th Symposium on Field Programmable Custom Computing Machines*, IEEE Computer Society, 2009 (FCCM '09). – ISBN 978–0–7695–3716–0, S. 201–208

[90] DuBois, David ; DuBois, Andrew ; Boorman, Thomas ; Connor, Carolyn ; Poole, Steve: An Implementation of the Conjugate Gradient Algorithm on FPGAs. In: *16th International Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, April 2008 (FCCM '08). – ISBN 978–0–7695–3307–0, S. 296–297

[91] Durbano, J.P. ; Ortiz, F.E.: FPGA-based acceleration of the 3D finite-difference time-domain method. In: *12th Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, April 2004 (FCCM '04), S. 156–163

[92] Döring, Andreas: Fast wie das menschliche Auge. In: *Mechatronik* 2010, IGT-Verlag, Nr. 10-11, S. 48–50. – ISSN 1867–2590

[93] Ej-Araby, E. ; Gonzalez, I. ; El-Ghazawi, T.: Bringing High-Performance Reconfigurable Computing to Exact Computations. In: *17th Conference on Field Programmable Logic and Applications*, IEEE Computer Society, Aug. 2007 (FPL '07). – ISBN 1–4244–1060–6, S. 79–85

[94] Fagin, B. ; Skrien, D.: Debugging on the Shoulders of Giants: Von Neumann's Programs 65 Years Later. In: *IEEE Computer* 45 (2012), Nov., Nr. 11, S. 59–68. – ISSN 0018–9162

[95] Farhad, Sardar M. ; Ko, Yousun ; Burgstaller, Bernd ; Scholz, Bernhard: Orchestration by Approximation: Mapping Stream Programs onto Multicore Architectures. In: *16th conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2011 (ASPLOS-XVI). – ISBN 978–1–4503–0266–1, S. 357–368

[96] Farrar, M.: Striped Smith-Waterman speeds database searches six times over other SIMD implementations. In: *Journal of Bioinformatics, Oxford University Press* 23 (2007), Januar, Nr. 2, S. 156–161. – ISSN 1367–4803

[97] Fatahalian, Kayvon ; Horn, Daniel R. ; Knight, Timothy J. ; Leem, Larkhoon ; Houston, Mike ; Park, Ji Y. ; Erez, Mattan ; Ren, Manman ; Aiken, Alex ; Dally, William J. ; Hanrahan, Pat: Sequoia: programming the memory hierarchy. In: *ACM/IEEE conference on Supercomputing*, ACM, 2006 (SC '06). – ISBN 0–7695–2700–0

[98] Ferrer, Edgar ; Bollman, Dorothy ; Moreno, Oscar: A fast finite field multiplier. In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC '07)* Bd. 4419, Springer-Verlag, 2007 (LNCS). – ISBN 978–3–540–71430–9, S. 238–246

[99] Fons, Francisco ; Fons, Mariano ; Cantó, Enrique ; López, Mariano: Deployment of Run-Time Reconfigurable Hardware Coprocessors Into Compute-Intensive Embedded Applications. In: *Journal of Signal Processing Systems, Springer* 66 (2012), S. 191–221. – ISSN 1939–8018

[100] Fontanelli, Flavio ; Mini, Giuseppe ; Sannino, Mario ; Guzik, Zbigniew ; Jacobsson, Richard ; Jost, Beat ; Neufeld, Niko: Embedded controllers for local board-control. In: *Proceedings of the 14th IEEE-NPSS conference on Real time*, IEEE Computer Society, 2005 (RTC '05). – ISBN 0–7803–9183–7, S. 64–68

[101] Fousse, Laurent ; Hanrot, Guillaume ; Lefèvre, Vincent ; Pélissier, Patrick ; Zimmermann, Paul: MPFR: A multiple-precision binary floating-point library with correct rounding. In: *ACM Transactions on Mathematical Software (TOMS)* 33 (2007), Nr. 2, S. 13. – ISSN 0098–3500

[102] Francis, Robert J.: A tutorial on logic synthesis for lookup-table based FPGAs. In: *Proceedings of the international conference on Computer-aided design*, IEEE Computer Society, 1992 (ICCAD '92). – ISBN 0–89791–540–2, S. 40–47

[103] Freeman, Ross H.: XC3000 family of user-programmable gate arrays. In: *Microprocessors and Microsystems, Elsevier* 13 (1989), Nr. 5, S. 313–320. – ISSN 0141–9331

[104] Fröning, Holger ; Nüssle, Mondrian ; Slogsnat, David ; Litz, Heiner ; Brüning, Ulrich: The HTX-Board: A Rapid Prototyping Station. In: *Proceedings of the 3rd Annual FPGAworld Conference*, 2006

[105] Gajski, Daniel D.: *Silicon Compilation*. Addison Wesley Publishing Company, 1988. – ISBN 978–0201099157

[106] Galuzzi, Carlo ; Bertels, Koen: The Instruction-Set Extension Problem: A Survey. In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC '08)* Bd. 4943, Springer-Verlag, 2008 (LNCS). – ISBN 978–3–540–78609–2, S. 209–220

[107] Gamm Fachausschuss On Computer Arithmetic and Scientific Computing: *Basic Requirements for a Future Floating-Point Arithmetic Standard*. Okt. 2006. – Draft, `http://www.math.uni-wuppertal.de/~xsc/gamm-fa/BasicRequ.pdf`

[108] Gaster, Benedict R. ; Howes, Lee: Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? In: *IEEE Computer* 45 (2012), S. 42–52. – ISSN 0018–9162

[109] Geilen, Marc: Synchronous Dataflow Scenarios. In: *ACM Transactions in Embedded Computing Systems (TECS)* 10 (2011), Jan., S. 16:1–16:31. – ISSN 1539–9087

[110] Gerwig, Guenter ; Wetter, Holger ; Schwarz, Eric M. ; Haess, Juergen: High Performance Floating-Point Unit with 116 Bit Wide Divider. In: *16th Symposium on Computer Arithmetic* Bd. 16, IEEE Computer Society, 2003 (ARITH '03). – ISBN 0–7695–1894–X, S. 87

[111] Ghazanfari, L.S. ; Airoldi, R. ; Nurmi, J. ; Ahonen, T.: Reconfigurable multi-processor architecture for streaming applications. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 477–478

[112] GiDEL Ltd.: *Application Acceleration on CHREC Novo-G: Breaking New Ground for Bioinformatics*. Sep. 2009. – `http://www.gidel.com/pdf/NovoG_SWappV1.pdf`

[113] Goldberg, David: What Every Computer Scientist Should Know About Floating-Point Arithmetic. In: *ACM Computing Surveys (CSUR)* 23 (1991), März, Nr. 1, S. 5–48. – ISSN 0360–0300

[114] Gonzalez, R. ; Horowitz, M.: Energy Dissipation InGgeneral Purpose Microprocessors. In: *IEEE Journal of Solid-State Circuits* 31 (1996), Sep, Nr. 9, S. 1277–1284. – ISSN 0018–9200

[115] Gonzalez, R.E.: A Software-Configurable Processor Architecture. In: *IEEE Micro* 26 (2006), Sept.-Okt., Nr. 5, S. 42–51. – ISSN 0272–1732

[116] Gowland, Paul ; Lester, David: A Survey of Exact Arithmetic Implementations. In: *Computability and Complexity in Analysis* Bd. 2064/2001, Springer-Verlag, Jan. 2001 (LNCS). – ISBN 978–3–540–42197–9, S. 30–47

[117] Granlund, T.: *The GNU MP Bignum Library.* 2008. – Web site: `http://gmplib.org`

[118] Greiner, Alain ; Faure, Etienne ; Pouillon, Nicolas ; Genius, Daniela: A Generic Hardware / Software Communication Middleware for Streaming Applications on Shared Memory Multi Processor Systems-on-Chip. In: *Forum on Specification, Verification and Design Languages*, IEEE, Sep. 2009 (FDL '09). – ISBN 978–2–9530504–1–7, S. 1–4

[119] Gschwind, Michael ; Hofstee, H. P. ; Flachs, Brian ; Hopkins, Martin ; Watanabe, Yukio ; Yamazaki, Takeshi: Synergistic Processing in Cell's Multicore Architecture. In: *IEEE Micro* 26 (2006), März-April, Nr. 2, S. 10–24. – ISSN 0272–1732

[120] Gu, Yongfeng: *FPGA Acceleration of Molecular Dynamics Simulations*, Boston University, Diss., 2008

[121] Gu, Yongfeng ; VanCourt, Tom ; Herbordt, Martin C.: Explicit design of FPGA-based coprocessors for short-range force computations in molecular dynamics simulations. In: *Elsevier Journal on Parallel Computing* 34 (2008), Mai, Nr. 4-5, S. 261–277. – ISSN 0167–8191

[122] Guo, Zhi ; Buyukkurt, Betul ; Najjar, Walid: Input data reuse in compiling window operations onto reconfigurable hardware. In: *ACM SIGPLAN Notices* 39 (2004), Juni, S. 249–256. – ISSN 0362–1340

[123] Guo, Zhi ; Najjar, Walid ; Buyukkurt, Betul: Efficient hardware code generation for FPGAs. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 5 (2008), Nr. 1, S. 1–26. – ISSN 1544–3566

[124] Guo, Zhi ; Najjar, Walid ; Vahid, Frank ; Vissers, Kees: A Quantitative Analysis of the Speedup Factors of FPGAs over Processors. In: *12th Symposium on Field Programmable Gate Arrays*, ACM, 2004 (FPGA '04). – ISBN 1–58113–829–6, S. 162–170

[125] Göhringer, D. ; Becker, J.: FPGA-Based Runtime Adaptive Multiprocessor Approach for Embedded High Performance Computing Applications. In: *IEEE Computer Society Annual Symposium on VLSI*, IEEE, Juli 2010 (ISVLSI '10'), S. 477–478

[126] Göhringer, D. ; Hübner, M. ; Schatz, V. ; Becker, J.: Runtime adaptive multi-processor system-on-chip: RAMPSoC. In: *22nd IEEE International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, April 2008 (IPDPS '08). – ISSN 1530–2075, S. 1–7

[127] Göhringer, D. ; Liu, Bin ; Hübner, M. ; Becker, J.: Star-Wheels Network-on-Chip featuring a self-adaptive mixed topology and a synergy of a circuit- and a packet-switching communication protocol. In: *19th Conference on Field Programmable Logic and Applications*, IEEE, Aug. 2009 (FPL '09). – ISBN 978–1–4244–3892–1, S. 320–325

[128] Göhringer, Diana ; Werner, Stephan ; Hübner, Michael ; Becker, Jürgen: RAMPSoCVM: Runtime Support and Hardware Virtualization for a Runtime Adaptive MPSoC. In: *21st Conference on Field Programmable Logic and Applications*, IEEE Computer Society, Sep. 5-7 2011 (FPL '11). – ISSN 978–0–7695–4529–5/11, S. 181–184

[129] Haase, Jan ; Eschmann, Frank ; Klauer, Bernd ; Waldschmidt, Klaus: The SDVM: A Self Distributing Virtual Machine for computer clusters. In: *Conference on Architecture of Computing Systems: Organic and Pervasive Computing (ARCS 2004),* Bd. 2981, Springer-Verlag, 2004 (LNCS). – ISBN 3–540–21238–8, S. 9–19

[130] Haase, Jan ; Eschmann, Frank ; Waldschmidt, Klaus: The SDVM – an Approach for Future Adaptive Computer Clusters. In: *10th Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, IEEE, April 2005 (DPDNS '05). – ISBN 0–7695–2312–9

[131] Halfhill, Tom R.: Parallel Processing with CUDA. In: *Microprocessor Report* Bd. Jan 28, 2008. – `http://www.mpronline.com/`

[132] Hannig, F. ; Schmid, M. ; Teich, J. ; Hornegger, H.: A Deeply Pipelined and Parallel Architecture for Denoising Medical Images. In: *Conference on Field-Programmable Technology*, IEEE Computer Society, Dez. 2010 (FPT '10). – ISBN 978–1–4244–8981–7, S. 485–490

[133] Hauck, Scott A.: *Multi-FPGA systems.* Seattle, WA, USA, University of Washington, Diss., 1995

[134] Hayes, Brian: A Lucid Interval. In: *American Scientist* 91 (2003), Nov./Dez., Nr. 6, S. 484–488

[135] He, Chuan ; Qin, Guan ; Lu, Mi ; Zhao, Wei: An Efficient Implementation of High-Accuracy Finite Difference Computing Engine on FPGAs. In: *International Conference on Application-specific Systems, Architectures and Processors*, 2006 (ASAP '06). – ISSN 2160–0511, S. 95–98

[136] He, Chuan ; Qin, Guan ; Lu, Mi ; Zhao, Wei: Optimized high-order finite difference wave equations modeling on reconfigurable computing platform. In: *Microprocessors and Microsystems, Elsevier* 31 (2007), März, S. 103–115. – ISSN 0141–9331

[137] He, Chuan ; Qin, Guan ; Zhao, Wei: High-order Finite Difference Modeling on Reconfigurable Computing Platform. In: *SM 1 GENERAL MODELING* Bd. 24, Society of Exploration Geophysicists, 2005 (SEG Expanded Abstracts). – ISSN 1052–3812

[138] He, Wenhao ; Yuan, Kui: An improved Canny edge detector and its realization on FPGA. In: *7th World Congress on Intelligent Control and Automation*, 2008 (WCICA '08), S. 6561–6564

[139] Helstrup, H. ; Lien, J. ; Lindenstruth, Volker ; Röhrich, Dieter ; Skaali, B. ; Steinbeck, Timm M. ; Ullaland, K. ; Vestbø, Anders S. ; Wiebalck, Arne: High Level Trigger System for the LHC ALICE Experiment. In: *International Conference on Computational Science – Part I*, Springer London, Vereinigtes Königreich, 2002 (ICCS '02). – ISBN 3–540–43591–3, S. 494–502

[140] Henkel, J. ; Herkersdorf, A. ; Bauer, L. ; Wild, T. ; Hubner, M. ; Pujari, R.K. ; Grudnitsky, A. ; Heisswolf, J. ; Zaib, A. ; Vogel, B. ; Lari, V. ; Kobbe, S.: Invasive manycore architectures. In: *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC),*, 2012. – ISSN 2153–6961, S. 193–200

[141] Herbordt, Martin C. ; Sukhwani, Bharat ; Chiu, Matt ; Khan, Md. A.: Production Floating Point Applications on FPGAs. In: *Symposium on Application Accelerators in High Performance Computing*, 2009 (SAAHPC '09). – `http://saahpc.ncsa.illinois.edu/09/papers/Herbordt_paper.pdf`

[142] Herbordt, M.C. ; Gu, Y. ; VanCourt, T. ; Model, J. ; Sukhwani, B. ; Chiu, M.: Computing Models for FPGA-Based Accelerators. In: *IEEE Computing in Science & Engineering* 10 (2008), Nov./Dez., Nr. 6, S. 35–45. – ISSN 1521–9615

[143] Hestenes, M.R. ; Stiefel, E.: Methods of Conjugate Gradients for Solving Linear Systems. In: *Journal of Research of the National Bureau of Standards* 49 (1952), Dez., Nr. 6, S. 409–436

[144] Hoare, C. A. R.: Communicating sequential processes. In: *Communications of the ACM* 21 (1978), Aug., Nr. 8, S. 666–677. – ISSN 0001–0782

[145] Hofmann, Andreas ; Waldschmidt, Klaus: SDVM^R: A Scalable Firmware for FPGA-based Multi-Core Systems-on-Chip. In: *9th Workshop on Parallel Systems and Algorithms (PASA 2008)* Bd. P-124, GI e.V., Jan. 2008 (Lecture Notes in Informatics), S. 59–68

[146] Holzmann, O. ; Lang, B. ; Schütt, H.: Newton's constant of gravitation and verified numerical quadrature. In: *Reliable Computing, Kluwer Academic Publishers* 2 (1996), Nr. 3, S. 229–239. – ISSN 1385–3139

[147] Hormati, Amir ; Kudlur, Manjunath ; Mahlke, Scott ; Bacon, David ; Rabbah, Rodric: Optimus: efficient realization of streaming applications on FPGAs. In: *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, ACM, Okt. 2008 (CASES '08). – ISBN 978–1–60558–469–0, S. 41–50

[148] Horrein, Pierre-Henri ; Hennebert, Christine ; Pétrot, Frédéric: An environment for (re)configuration and execution management of heterogeneous flexible radio platforms. In: *Elsevier Microprocessors and Microsystems* 37 (2013), Aug.–Okt., Nr. 6–7, S. 701–712. – ISSN 0141–9331

[149] Huang, Miaoqing ; Wang, Lingyuan ; El-Ghazawi, Tarek: Accelerating Double Precision Floating-Point Hessenberg Reduction on FPGA and Multicore Architectures. In: *Symposium on Application Accelerators in High Performance Computing*, 2010 (SAAHPC '10). – `http://saahpc.ncsa.illinois.edu/10/papers/paper_48.pdf`

[150] Huang, Shan ; Hormati, Amir ; Bacon, David ; Rabbah, Rodric: Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In: *Proceedings of the 2008 European Conference on Object-Oriented Programming (ECOOP '08)* Bd. 5142. Berlin / Heidelberg : Springer-Verlag, Juli 2008 (LNCS). – ISBN 978–3–540–70591–8, S. 76–103

[151] IEEE Standard for Floating-Point Arithmetic. In: *IEEE Std 754-2008* (2008), 29, S. 1–58

[152] Impulse Accelerated Technologies, Inc.: *Accelerate C in FPGA – Fast Prototyping, Fast Optimization, Fast Applications.* `http://www.impulseaccelerated.com/ImpulseFlyerV1.pdf`. Version: 2007. – White Paper

[153] Isard, Michael ; Budiu, Mihai ; Yu, Yuan ; Birrell, Andrew ; Fetterly, Dennis: Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: *2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ACM, 2007 (EuroSys '07). – ISBN 978–1–59593–636–3, S. 59–72

[154] Jagannathan, R.: Coarse-Grain Dataflow Programming of Conventional Parallel Computers. In: *Advanced Topics in Dataflow Computing and Multithreading*, IEEE Computer Society Press, 1995, S. 113–129

[155] Jain, Advait ; Gambhir, Pulkit ; Jindal, Priyanka ; Balakrishnan, M. ; Paul, Kolin: FPGA Accelerator for Protein Structure Prediction Algorithms. In: *5th Southern Programmable Logic Conference*, IEEE Circuits and Systems Society, April 2009 (SPL V). – ISBN 9781–4244–3846–4, S. 123–128

[156] Jaiswal, M.K. ; Cheung, R.C.C.: Area-Efficient Architectures for Large Integer and Quadruple Precision Floating Point Multipliers. In: *20th Symposium on Field-Programmable Custom Computing Machines*, 2012 (FCCM '12). – ISBN 978–0–7695–4699–5, S. 25–28

[157] Jerez, Juan L. ; Constantinides, George A. ; Kerrigan, Eric C.: Fixed Point Lanczos: Sustaining TFLOP-equivalent Performance in FPGAs for Scientific Computing. In: *20th Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, 2012 (FCCM '12). – ISBN 978–0–7695–4699–5, S. 53–60

[158] Jiang, Jiang ; Mirian, Vincent ; Tang, Kam P. ; Chow, Paul ; Xing, Zuocheng: Matrix Multiplication Based on Scalable Macro-Pipelined FPGA Accelerator Architecture. In: *Conference on Reconfigurable Computing and FPGAs*, IEEE Computer Society, 2009 (ReConFig '09). – ISBN 978–0–7695–3917–1, S. 48–53

[159] Jiang, Jun ; Luk, W. ; Rueckert, D.: FPGA-based computation of free-form deformations in medical image registration. In: *Conference on Field Programmable Technology*, IEEE Computer Society, Dez. 2003 (FPT '03). – ISBN 0–7803–8320–6, S. 234–241

[160] Johnston, Wesley M. ; Hanna, J. R. P. ; Millar, Richard J.: Advances in dataflow programming languages. In: *ACM Computing Surveys (CSUR)* 36 (2004), März, Nr. 1, S. 1–34. – ISSN 0360–0300

[161] Jóźwiak, Lech ; Nedjah, Nadia ; Figueroa, Miguel: Modern development methods and tools for embedded reconfigurable systems: A survey. In: *Integration VLSI Journal, Elsevier* 43 (2010), Januar, Nr. 1, S. 1–33. – ISSN 0167–9260

[162] Kahan, W.: Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic. Version: Mai 1996. `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`. Elect. Engineering & Computer Science, University of California, Berkeley CA 94720-1776, USA, Mai 1996. – Work in Progress

[163] Kahan, William: *How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?* Online. `http://www.cs.berkeley.edu/~wkahan/Mindless.pdf`. Version: 2006

[164] Kernhof, J. ; Baumhof, C. ; Höfflinger, B. ; Kulisch, U. ; Kwee, S. ; Schramm, P. ; Selzer, M. ; Teufel, T.: A CMOS Floating-Point Processing Chip for Verified Exact Vector Arithmetic. In: *20th European Solid-State Circuits Conference*, Editions Frontières, Sep. 1994 (ESSCIRC '94). – ISBN 2–86332–160–9, S. 196–199

[165] Kestur, S. ; Davis, J.D. ; Chung, E.S.: Towards a Universal FPGA Matrix-Vector Multiplication Architecture. In: *20th Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, April/Mai 2012 (FCCM '12). – ISBN 978–0–7695–4699–5, S. 9–16

[166] Khronos Group: *Khronos OpenCL API Registry*. Dez. 2008. – `http://www.khronos.org/registry/cl/`

[167] Khronos OpenCL Working Group ; Munshi, Aaftab (Hrsg.): *The OpenCL Specification*. Juni 2010

[168] Kicherer, Mario: *Design and Implementation of a Low-overhead Run-time System for Self-X Architectures*. Karlsruhe, Universität Karlsruhe (TH), Institut für Technische Informatik, Diplomarbeit, Dez. 2008. – `http://capp.itec.kit.edu/diploma/da/kicherer-2008.pdf`

[169] Kicherer, Mario: *Reducing the complexity of heterogeneous computing : a unified approach for application development and runtime optimization*, Karlsruher Institut für Technologie (KIT), Deutschland, Dissertation, 2014

[170] Kicherer, Mario ; Buchty, Rainer ; Karl, Wolfgang: Cost-Aware Function Migration in Heterogeneous Systems. In: *6th Conference on High Performance and Embedded Architectures and Compilers*, ACM, Jan. 2011 (HiPEAC '11). – ISBN 978–1–4503–0241–8, S. 137–145

[171] Kim, Jung S. ; Deng, Lanping ; Mangalagiri, Prasanth ; Irick, Kevin ; Sobti, Kanwaldeep ; Kandemir, Mahmut ; Narayanan, Vijaykrishnan ; Chakrabarti, Chaitali ; Pitsianis, Nikos ; Sun, Xiaobai: An Automated Framework for Accelerating Numerical Algorithms on Reconfigurable Platforms Using Algorithmic/Architectural Optimization. In: *IEEE Transactions on Computers* 58 (2009), Nr. 12, S. 1654–1667. – ISSN 0018–9340

[172] Kirchner, Reinhard ; Kulisch, Ulrich: Accurate arithmetic for vector processors. In: *Journal of Parallel and Distributed Computing, Academic Press* 5 (1988), Nr. 3, S. 250–270. – ISSN 0743–7315

[173] Klatte, Rudi: *C-XSC: A C++ Class Library for Extended Scientific Computing*. Springer New York, 1993. – ISBN 0–387–56328–8. – Translator: G. F. Corliss

[174] Klatte, Rudi ; Kulisch, Ulrich W. ; Neaga, M. ; Ratz, D. ; Ullrich, C. P.: *Pascal-XSC: Language Reference with Examples*. Springer New York, 1992. – ISBN 0–387–55137–9

[175] Kulisch, Ulrich: The XSC tools for extended scientific computing. In: *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical software*, Chapman & Hall, Ltd., 1997. – ISBN 0–412–80530–8, S. 280–284

[176] Kulisch, Ulrich: *XSC Languages (C-XSC, PASCAL-XSC)*. 2008. – Web site: `http://www.xsc.de/`

[177] Kulisch, Ulrich W.: *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer New York, 2002. – ISBN 3–211–83870–8

[178] Kulisch, Ulrich W.: Complete Interval Arithmetic and Its Implementation on the Computer. In: *Numerical Validation in Current Hardware Architectures* Bd. 5492/2009, Springer-Verlag, April 2009 (LNCS). – ISBN 978–3–642–01590–8, S. 7–26

[179] Kumar, R. ; Tullsen, D.M. ; Jouppi, N.P. ; Ranganathan, P.: Heterogeneous Chip Multiprocessors. In: *IEEE Computer* 38 (2005), Nr. 11, S. 32–38. – ISSN 0018–9162

[180] Kumar, S. ; Forward, K. ; Palaniswami, M.: A Fast-Multiplier Generator for FPGAs. In: *8th Conference on VLSI Design*, IEEE Computer Society, Jan. 1995 (VLSID '95). – ISBN 0–8186–6905–5, S. 53

[181] Kumar, Vinay B. Y. ; Joshi, Siddharth ; Patkar, Sachin B. ; Narayanan, H.: FPGA Based High Performance Double-Precision Matrix Multiplication. In: *22nd Conference on VLSI Design*, IEEE Computer Society, 2009 (VLSID '09). – ISBN 978–0–7695–3506–7, S. 341–346

[182] Lanczos, Cornelius: Solution of systems of linear equations by minimized-iterations. In: *Journal of Research of the National Bureau of Standards* 49 (1952), S. 33–53. – ISSN 0160–1741

[183] Leiserson, Charles E.: The Cilk++ concurrency platform. In: *46th Design Automation Conference*, ACM, 2009 (DAC '09). – ISBN 978–1–60558–497–3, S. 522–527

[184] Li, Baofeng ; Dou, Yong ; Zhou, Haifang ; Zhou, Xingming: FPGA Accelerator for Wavelet-based Automated Global Image Registration. In: *EURASIP Journal on Embedded Systems, Hindawi* (2009), Jan., S. 1:1–1:10. – ISSN 1687–3955

[185] Li, Heng ; Durbin, Richard: Fast and accurate short read alignment with Burrows-Wheeler transform. In: *Bioinformatics, Oxford University Press* 25 (2009), Juli, Nr. 14, S. 1754–1760. – ISSN 1367–4803

[186] Liu, Ming ; Lu, Zhonghai ; Kuehn, Wolfgang ; Jantsch, Axel: FPGA-Based Particle Recognition in the HADES Experiment. In: *IEEE Design & Test of Computers* 28 (2011), Nr. 4, S. 48–57. – ISSN 0740–7475

[187] Lopes, Antonio R. ; Constantinides, George A.: A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation. In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC '08)* Bd. 4943, Springer-Verlag, 2008 (LNCS). – ISBN 978–3–540–78609–2, S. 75–86

[188] Lopes, Antonio R. ; Constantinides, George A.: A Fused Hybrid Floating-Point and Fixed-Point Dot-Product for FPGAs. In: *Reconfigurable Computing: Architectures, Tools and Applications (ARC '10)* Bd. 5992, Springer-Verlag, 2010 (LNCS). – ISBN 978–3–642–12132–6, S. 157–168

[189] Lopes, Antonio R. ; Constantinides, George A. ; Kerrigan, Eric C.: A floating-point solver for band structured linear equations. In: *Conference on Field Programmable Technology*, IEEE Computer Society, Okt. 2008 (FPT '08). – ISBN 978–1–4244–2796–3, S. 353–356

[190] Louie, M.E. ; Ercegovac, M.D.: On Digit-Recurrence Division Implementations for Field Programmable Gate Arrays. In: *11th Symposium on Computer Arithmetic*, IEEE Computer Society, Juni/Juli 1993 (ARITH '93), S. 202–209

[191] Lysecky, Roman ; Stitt, Greg ; Vahid, Frank: Warp Processors. In: *41st Design Automation Conference*, ACM, 2004 (DAC '04). – ISBN 1–58113–828–8, S. 659–681

[192] Mackay, David R.: *A Library Based Approach to Threading for Performance.* 02 2004. – Whitepaper, `http://software.intel.com/en-us/articles/a-library-based-approach-to-threading-for-performance/`

[193] Mei, Bingfeng ; Vernalde, Serge ; Verkest, Diederik ; Man, Hugo ; Lauwereins, Rudy: ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In: *13th Conference on Field Programmable Logic and Applications (FPL '03)* Bd. 2778. Berlin Heidelberg : Springer-Verlag, September 2003 (LNCS). – ISBN 978–3–540–40822–2, S. 61–70

[194] Membarth, Richard ; Hannig, Frank ; Teich, Jürgen ; Körner, Mario ; Eckert, Wieland: Frameworks for Multi-core Architectures: A Comprehensive Evaluation Using 2D/3D Image Registration. In: *24th Conference on Architecture of Computing Systems (ARCS '11)* Bd. 6566, Springer-Verlag, 2011 (LNCS). – ISBN 978–3–642–19136–7, S. 62–73

[195] Menotti, Ricardo ; Cardoso, Joao M. ; Fernandes, Marcio M. ; Marques, Eduardo: LALP: A Novel Language to Program Custom FPGA-Based Architectures. In: *Symposium on Computer Architecture and High Performance Computing*, IEEE Computer Society, Okt. 2009 (SBAC-PAD '09). – ISSN 1550–6533, S. 3–10

[196] Meswani, M.R. ; Carrington, L. ; Unat, D. ; Snavely, A. ; Baden, S. ; Poole, S.: Modeling and predicting application performance on hardware accelerators. In: *International Symposium on Workload Characterization (IISWC)*, IEEE Computer Society, Nov. 2011. – ISBN 978–1–4577–2064–2, S. 73

[197] Metri, Grace ; Sabharwal, Manuj ; Iyer, Sundar ; Agrawal, Abhishek: Hardware/Software Codesign to Optimize SoC Device Battery Life. In: *IEEE Computer* 46 (2013), Oktober, Nr. 10, S. 89–92. – ISSN 0018–9162

[198] Meyer, B. ; Schumacher, J. ; Plessl, C. ; Forstner, J.: Convey vector personalities - FPGA acceleration with an openmp-like programming effort? In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 189–196

[199] Moore, Ramon E.: *Interval arithmetic and automatic error analysis in digital computing.* Stanford, CA, USA, Stanford University, Diss., 1963

[200] Morris, Gerald R. ; Prasanna, Viktor K. ; Anderson, Richard D.: A Hybrid Approach for Mapping Conjugate Gradient onto an FPGA-Augmented Reconfigurable Supercomputer. In: *14th Sympsium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, 2006 (FCCM '06). – ISBN 0–7695–2661–6, S. 3–12

[201] Nagar, K.K. ; Bakos, J.D.: A Sparse Matrix Personality for the Convey HC-1. In: *19th Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, Mai 2011 (FCCM '11). – ISBN 978–0–7695–4301–7, S. 1–8

[202] Nane, R. ; Sima, V. ; Olivier, B. ; Meeuws, R. ; Yankova, Y. ; Bertels, K.: DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 619–622

[203] Natoli, Vincent ; Allred, Jeff ; Coyne, Jack ; Lynch, William: In-Socket FPGA Implementation of Bioinformatic Algorithms Using the Intel AAL. In: *Symposium on Application Accelerators in High Performance Computing*, 2009 (SAAHPC '09). – `http://saahpc.ncsa.illinois.edu/09/papers/Natoli_paper.pdf`

[204] Nelson, Chad ; Townsend, Kevin ; Rao, Bhavani S. ; Jones, Phillip ; Zambreno, Joseph: Shepard: A Fast Exact Match Short Read Aligner. In: *Conference on Formal Methods and Models for Codesign*, IEEE, Juli 2012 (MEMOCODE '12). – ISBN 978–1–4673–1314–8

[205] Neumaier, Arnold: Grand Challenges and Scientific Standards in Interval Analysis. In: *Reliable Computing, Springer* 8 (2002), Aug., Nr. 4, S. 313–320. – ISSN 1385–3139

[206] Nickolls, John ; Buck, Ian ; Garland, Michael ; Skadron, Kevin: Scalable parallel programming with CUDA. In: *ACM Queue* 6 (2008), Nr. 2, S. 40–53. – ISSN 1542–7730

[207] Njoroge, Njuguna ; Casper, Jared ; Wee, Sewook ; Teslyar, Yuriy ; Ge, Daxia ; Kozyrakis, Christos ; Olukotun, Kunle: ATLAS: a chip-multiprocessor with transactional memory support. In: *Design, Automation & Test in Europe Conference & Exhibition*, EDA Consortium, 2007 (DATE '07). – ISBN 978–3–9810801–2–4, S. 3–8

[208] OpenACC Group: *NVIDIA, Cray, PGI, CAPS Unveil 'OpenACC' Programming Standard for Parallel Computing.* Online, Nov. 2011. – `http://www.openacc-standard.org/announcements-1/nvidiacraypgicapsunveil'openacc'programmingstandardforparallelcomputing`

[209] Opfer, Gerhard: *Numerische Mathematik für Anfänger*. Friedrich Vieweg & Sohn Verlagsgesellschaft, 2001. – ISBN 3–528–27265–1

[210] Palatin, P. ; Lhuillier, Y. ; Temam, O.: CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs. In: *39th International Symposium on Microarchitecture*, 2006 (MICRO-39 2006), S. 247–258

[211] Panda, R. ; Ebeling, C. ; Hauck, S.: Adding dataflow-driven execution control to a Coarse-Grained Reconfigurable Array. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 353–360

[212] Park, In K. ; Singhal, Nitin ; Lee, Man H. ; Cho, Sungdae ; Kim, Chris W.: Design and Performance Evaluation of Image Processing Algorithms on GPUs. In: *IEEE Transactions on Parallel and Distributed Systems* 22 (2011), Nr. 1, S. 91–104. – ISSN 1045–9219

[213] Parker, Michael: Floating-Point Matrix Processing using FPGAs. In: *Hot Chips 24 – A Symposium on High Performance Chips*, 2012. – `http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24-6-Tech-Scalability/HC24.29.610-FPP-FPGAs-Parker-Altera.pdf`

[214] Parvez, Husain ; Marrakchi, Zied ; Mehrez, Habib: Application Specific FPGA Using Heterogeneous Logic Blocks. In: Sirisuk, Phaophak (Hrsg.) ; Morgan, Fearghal (Hrsg.) ; El-Ghazawi, Tarek (Hrsg.) ; Amano, Hideharu (Hrsg.): *Reconfigurable Computing: Architectures, Tools and Applications (ARC '10)* Bd. 5992. Berlin Heidelberg : Springer-Verlag, 2010. – ISBN 978–3–642–12132–6, S. 92–109

[215] Pasca, Bogdan: Correctly rounded floating-point division for DSP-enabled FPGAs. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 249–254

[216] Perez-Vidal, Carlos ; Gracia, Luis: High speed filtering using reconfigurable hardware. In: *Journal of Parallel and Distributed Computing, Elsevier* 69 (2009), Nr. 11, S. 896–904. – ISSN 0743–7315

[217] Philipp, F. ; Glesner, M.: $(GECO)^2$: A graphical tool for the generation of configuration bitstreams for a smart sensor interface based on a Coarse-Grained Dynamically Reconfigurable Architecture. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 679–682

[218] Planas, Judit ; Badia, Rosa M. ; Ayguadé, Eduard ; Labarta, Jesus: Hierarchical Task-Based Programming With StarSs. In: *International Journal of High Performance Computing Applications, Sage Publications* 23 (2009), Aug., S. 284–299. – ISSN 1094–3420

[219] Quilleré, Fabien ; Rajopadhye, Sanjay: Optimizing memory usage in the polyhedral model. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22 (2000), September, Nr. 5, S. 773–815. – ISSN 0164–0925

[220] Radakovits, Randor ; Jinkerson, Robert E. ; Fuerstenberg, Susan I. ; Tae, Hongseok ; Settlage, Robert E. ; Boore, Jeffrey L. ; Posewitz, Matthew C.: Draft genome sequence and genetic transformation of the oleaginous alga Nannochloropis gaditana. In: *Nature Communications, Nature Publishing Group* 3 (2012), Feb., Nr. 686

[221] Rafique, Abid ; Kapre, Nachiket ; Constantinides, George A.: A High Throughput FPGA-Based Implementation of the Lanczos Method for the Symmetric Extremal Eigenvalue Problem. In: *8th conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC '12)* Bd. 7199, Springer-Verlag, 2012 (LNCS). – ISBN 978–3–642–28364–2, S. 239–250

[222] Remmert, M. ; Biegert, A. ; Hauser, A. ; Söding, J.: HHblits: lightning-fast iterative protein sequence searching by HMM-HMM alignment. In: *Journal of Nature methods (Nature Publishing Group)* 9 (2012), Feb., Nr. 2, S. 173–175. – ISSN 1548–7105

[223] Robison, Arch D.: Composable Parallel Patterns with Intel Cilk Plus. In: *IEEE Computing in Science and Engineering* 15 (2013), Nr. 2, S. 66–71. – ISSN 1521–9615

[224] Rodriguez, S.M.H. ; Rodriguez-Henriquez, F.: An FPGA arithmetic logic unit for computing scalar multiplication using the half-and-add method. In: *Conference on Reconfigurable Computing and FPGAs*, IEEE Computer Society, Sep. 2005 (ReConFig '05). – ISBN 0–7695–2456–7, S. 7 ff.

[225] Rokne, Jon G.: Interval Arithmetic and Interval Analysis: An Introduction. In: *Granular computing: an emerging paradigm, Physica-Verlag* (2001), S. 1–22. ISBN 3–7908–1387–7

[226] Roldao, Antonio ; Constantinides, George A.: A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation for Dense Matrices. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3 (2010), Jan., S. 1:1–1:19. – ISSN 1936–7406

[227] Rumpf, Martin ; Strzodka, Robert: Using Graphics Cards for Quantized FEM Computations. In: *IASTED International Conference on Visualization, Imaging and Image Processing*, ACTA Press, Sep. 2001 (VIIP '01). – ISBN 0–88986–309–1, S. 193–202

[228] Russell, John: *Hybrid Core Computing – Convey Computer Tames Data Deluge*. `http://conveycomputer.com/Resources/Convey_Computer_BioIT_White_Paper.pdf`. Version: Nov./Dez. 2010

[229] Saha, Bratin ; Zhou, Xiaocheng ; Chen, Hu ; Gao, Ying ; Yan, Shoumeng ; Rajagopalan, Mohan ; Fang, Jesse ; Zhang, Peinan ; Ronen, Ronny ; Mendelson, Avi: Programming model for a heterogeneous x86 platform. In: *Conference on Programming Language Design and Implementation*, ACM, 2009 (PLDI '09). – ISBN 978–1–60558–392–1, S. 431–440

[230] Saldaña, Manuel ; Nunes, Daniel ; Ramalho, Emanuel ; Chow, Paul: Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI. In: *Conference on Reconfigurable Computing and FPGAs*, IEEE Computer Society, Sep. 2006 (ReConFig '06). – ISBN 1–4244–0690–0, S. 1–10

[231] Saldaña, Manuel ; Ramalho, Emanuel ; Chow, Paul: A message-passing hardware/software cosimulation environment for reconfigurable computing systems. In: *International Journal on Reconfigurable Computing, Hindawi* 2009 (2009), S. 1–9. – ISSN 1687–7195

[232] Schaumont, P. ; Verbauwhede, I.: A Component-Based Design Environment for ESL Design. In: *IEEE Design Test of Computers* 23 (2006), Nr. 5, S. 338–347. – ISSN 0740–7475

[233] Schmidtobreick, Anke M.: *Numerical Methods on Reconfigurable Hardware using High Level Programming Paradigms*. Karlsruhe, Deutschland, Karlsruher Institut für Technologie, Engineering Mathematics and Computing Lab, Diplomarbeit, 2010

[234] Schulte, M.J. ; Swartzlander, Jr. E.E.: A Family of Variable-Precision Interval Arithmetic Processors. In: *IEEE Transactions on Computers* 49 (2000), Mai, Nr. 5, S. 387–397. – ISSN 0018–9340

[235] Sczyrba, A. ; Pratap, A. ; Canon, S. ; Han, J. ; Copeland, A. ; Wang, Z. ; Brewer, T. ; Soper, D. ; D'Jamoos, M. ; Collins, K. ; Vacek, G.: *Efficient Graph Based Assembly of Short-Read Sequences on Hybrid Core Architecture*. März 2011. – `http://184.73.149.1/fedora/repository/ir%3A155198` (DOE Joint Genome Institute User Meeting) (Poster Abstract)

[236] Sergyienko, Anatoli ; Maslennikov, Oleg: Implementation of Givens QR-Decomposition in FPGA. In: *4th Conference on Parallel Processing and Applied Mathematics-Revised Papers*, Springer London, Vereinigtes Königreich, 2002 (PPAM '01). – ISBN 3–540–43792–4, S. 458–465

[237] Shafique, Muhammad ; Bauer, Lars ; Henkel, Jörg: Adaptive Energy Management for Dynamically Reconfigurable Processors. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 33 (2014), Januar, Nr. 1, S. 50–63

[238] Silc, Jurij ; Robic, Borut ; Ungerer, Theo: *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer, 1999. – ISBN 978–3–540–64798–0

[239] Sinha, S. ; Srikanthan, T.: Dataflow graph partitioning for high level synthesis. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 503–506

[240] Slogsnat, David ; Giese, Alexander ; Brüning, Ulrich: A versatile, low latency HyperTransport core. In: *15th Symposium on Field Programmable Gate Arrays*, ACM, 2007 (FPGA '07). – ISBN 978–1–59593–600–4, S. 45–52

[241] Slogsnat, David ; Giese, Alexander ; Nüssle, Mondrian ; Brüning, Ulrich: An open-source HyperTransport core. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 1 (2008), Nr. 3, S. 1–21. – ISSN 1936–7406

[242] Smith, T. ; Waterman, M.: Identification of Common Molecular Subsequences. In: *Journal of Molecular Biology, Elsevier* 147 (1981), Nr. 1, S. 195–197. – ISSN 0022–2836

[243] Smith, T.F. ; Waterman, M.S.: Identification of common molecular subsequences. In: *Journal of Molecular Biology (Elsevier)* 147 (1981), Nr. 1, S. 195 – 197. – ISSN 0022–2836

[244] So, Hayden Kwok-Hay: *BORPH: An Operating System for FPGA-based Reconfigurable Computers*. Berkeley, Kalifornien, USA, University of California at Berkeley, Diss., 2007

[245] So, Hayden Kwok-Hay ; Brodersen, Robert: A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. In: *ACM Transactions on Embedded Computing Systems* 7 (2008), Feb., Nr. 2, S. 14:1–14:28

[246] Soliman, Mostafa I. ; Abozaid, Ghada Y.: FPGA implementation and performance evaluation of a high throughput crypto coprocessor. In: *Journal of Parallel and Distributed Computing, Elsevier* 71 (2011), Nr. 8, S. 1075–1084. – ISSN 0743–7315

[247] Storaasli, Olaf ; Ornl, Weikuan Y. ; Strenski, Dave ; Maltby, Jim: *Performance Evaluation of FPGA-Based Biological Applications Olaf*. Mai 2007. – Proceedings of the Cray Users Group'07

[248] Stratton, John A. ; Rodrigues, Christopher ; Sung, I-Jui (. ; Chang, Li-Wen ; Anssari, Nasser ; Liu, Geng (. ; W. Hwu, Wen mei ; Obeid, Nady: Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems. In: *IEEE Computer* 45 (2012), S. 26–32. – ISSN 0018–9162

[249] Strzodka, Robert: *Hardware Efficient PDE Solvers in Quantized Image Processing*, Universität Duisburg, Dissertation, 2004

[250] Söding, Johannes ; Remmert, Michael ; Hauser, Andreas: *HH-suite for sensitive sequence searching based on HMM-HMM alignment*. März 2012. – `ftp://toolkit.genzentrum.lmu.de/pub/HH-suite/hhsuite-userguide.pdf` (Userguide)

[251] Tanner, Stephen ; Shu, Hongjun ; Frank, Ari ; Wang, Ling-Chi ; Zandi, Ebrahim ; Mumby, Marc ; Pevzner, Pavel A. ; Bafna., Vineet: InsPecT: Fast and accurate identification of post-translationally modified peptides from tandem mass spectra. In: *Analytical Chemistry (American Chemistry Society)* 77 (2005), Juli, Nr. 14

[252] Taylor, M.B. ; Kim, J. ; Miller, J. ; Wentzlaff, D. ; Ghodrat, F. ; Greenwald, B. ; Hoffman, H. ; Johnson, P. ; Lee, Jae-Wook ; Lee, W. ; Ma, A. ; Saraf, A. ; Seneski, M. ; Shnidman, N. ; Strumpen, V. ; Frank, M. ; Amarasinghe, S. ; Agarwal, A.: The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. In: *IEEE Micro* 22 (2002), März/April, Nr. 2, S. 25–35. – ISSN 0272–1732

[253] The Santa Cruz Operation, Inc.: *System V Application Binary Interface (Edition 4.1)*. 1997. – `http://www.caldera.com/developers/devspecs/gabi41.pdf`

[254] Thies, William ; Chandrasekhar, Vikram ; Amarasinghe, Saman: A Practical Approach to Exploiting Coarse-Grained Pipeline Parallelism in C Programs. In: *40th International Symposium on Microarchitecture*, IEEE Computer Society, 2007 (MICRO-40 2007). – ISBN 0–7695–3047–8, S. 356–369

[255] Thies, William ; Karczmarek, Michal ; Amarasinghe, Saman P.: StreamIt: A Language for Streaming Applications. In: *Proceedings of the 11th International Conference on Compiler Construction*, Springer London, Vereinigtes Königreich, 2002 (CC '02). – ISBN 3–540–43369–4, S. 179–196

[256] Thoma, F. ; Kühnle, M. ; Bonnot, P. ; Panainte, E.M. ; Bertels, K. ; Goller, S. ; Schneider, A. ; Guyetant, S. ; Schuler, E. ; Müller-Glaser, K.D. ; Becker, J.: MORPHEUS: Heterogeneous Reconfigurable Computing. In: *17th Conference on Field Programmable Logic and Applications*, IEEE Computer Society, Aug. 2007 (FPL '07). – ISBN 1–4244–1060–6, S. 409–414

[257] Thoma, Florian ; Becker, Jürgen: ISRC: a runtime system for heterogeneous reconfigurable architecture. In: *5th Workshop on Reconfigurable Communication-centric Systems on Chip 2010 (ReCoSoC '10)* Bd. 7551, KIT Scientific Publishing, Mai 2010 (KIT scientific reports). – ISBN 978–3–86644–515–4, 59–65

[258] Thoma, Florian ; Kühnle, Matthias ; Grasset, Arnaud ; Brelet, Paul ; Millet, Philippe ; Bonnot, Philippe ; Campi, Fabio ; Voros, Nikolaos S. ; Putzke-Roeming, Wolfram ; Schneider, Axel ; Hübner, Michael ; Müller-Glaser, Klaus D. ; Becker, Jürgen: A heterogeneous reconfigurable System-on-Chip: MORPHEUS. In: Plaks, Toomas P. (Hrsg.): *The 2011 International Conference on Engineering of Reconfigurable Systems and Algorithms*, 2011 (ERSA '11). – ISBN 1–60132–177–5, S. 243–249

[259] Thomas, Alexander ; Becker, Jürgen: New Adaptive Multi-grained Hardware Architecture for Processing of Dynamic Function Patterns (Neue adaptive multi-granulare Hardwarearchitektur). In: *it – Information Technology, Oldenbourg* 49 (2007), Nr. 3, S. 165–173. – ISSN 1611–2776

[260] Tomasulo, R. M.: An Efficient Algorithm for Exploiting Multiple Arithmetic Units. In: *IBM Journal of Research and Development* 11 (1967), Jan., Nr. 1, S. 25–33. – ISSN 0018–8646

[261] Treleaven, Philip C. ; Brownbridge, David R. ; Hopkins, Richard P.: Data-Driven and Demand-Driven Computer Architecture. In: *ACM Computing Surveys (CSUR)* 14 (1982), März, Nr. 1, S. 93–143. – ISSN 0360–0300

[262] Tripp, J.L. ; Peterson, K.D. ; Ahrens, C. ; Poznanovic, J.D. ; Gokhale, M.B.: Trident: an FPGA compiler framework for floating-point algorithms. In: *15th Conference on Field Programmable Logic and Applications*, IEEE Computer Society, Aug. 2005 (FPL '05). – ISBN 0–7803–9362–7, S. 317–322

[263] Tripp, Justin L. ; Gokhale, Maia B. ; Peterson, Kristopher D.: Trident: From High-Level Language to Hardware Circuitry. In: *IEEE Computer* 40 (2007), Nr. 3, S. 28–37. – ISSN 0018–9162

[264] Trujillo, Salvador ; Azanza, Maider ; Diaz, Oscar: Generative Metaprogramming. In: *6th conference on Generative Programming and Component Engineering*, ACM, 2007 (GPCE '07). – ISBN 978–1–59593–855–8, S. 105–114

[265] Udupa, Abhishek ; Govindarajan, R. ; Thazhuthaveetil, Matthew J.: Synergistic Execution of Stream Programs on Multicores with Accelerators. In: *Conference on Languages, Compilers, and Tools for Embedded Systems*, ACM, 2009 (LCTES '09). – ISBN 978–1–60558–356–3, S. 99–108

[266] Ullmann, Michael ; Hübner, Michael ; Grimm, Björn ; Becker, Jürgen: An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. In: *18th International Parallel and Distributed Processing Symposium*, IEEE Computer Society, April 2004 (IPDPS '04). – ISBN 0–7695–2132–0, S. 135a

[267] Ullmann, Michael ; Hübner, Michael ; Grimm, Björn ; Becker, Jürgen: On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities. In: *14th Conference on Field Programmable Logic and Application (FPL '04)* Bd. 3203, Springer-Verlag, 30. Aug. - 1. Sep. 2004 (LNCS). – ISBN 3–540–22989–2, S. 454–463

[268] Vacek, G.: Hybrid-Core Computing for High Throughput Bioinformatics. In: *Journal of Biomolecular Techniques* 22 (Supplement) (2011), Okt. – `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3186629/pdf/jbtS37b.pdf` (Poster Abstract)

[269] Vacek, George ; Collins, Kirby: *Efficient Short-Read Sequence Mapping and Assembly on a Hybrid-Core Architecture.* – `http://conveycomputer.com/Resources/SequenceMappingAssemblyHybrid-CoreArchitecture.pdf` (White Paper)

[270] Vacek, George ; Kelly, Mark ; Collins, Kirby: *Screening DNA Sequences Against a Protein Database Using a Hybrid-Core Implementation of Smith-Waterman.* 2012. – `http://conveycomputer.com/Resources/Screening-DNA-Sequences-Against-a-Protein-Database-Using-a-Hybrid-Core-\Implementation-of-Smith-Waterman-by-Convey.pdf` (White Paper)

[271] Vassiliadis, S. ; Wong, S. ; Cotofana, S. D.: The MOLEN μ-coded Processor. In: *11th Conference on Field-Programmable Logic and Applications (FPL '01)* Bd. 2147, Springer-Verlag, Aug. 2001 (LNCS), S. 275–285

[272] Vassiliadis, Stamatis ; Wong, Stephan ; Gaydadjiev, Georgi ; Bertels, Koen ; Kuzmanov, Georgi ; Panainte, Elena M.: The MOLEN Polymorphic Processor. In: *IEEE Transactions on Computers* 53 (2004), Nr. 11, S. 1363–1375. – ISSN 0018–9340

[273] Vogel, Thorsten: *An FPGA-based Reconfigurable Accelerator Framework using HyperTransport.* Karlsruhe, Universität Karlsruhe (TH), Institut für Technische Informatik, Diplomarbeit, Dez. 2008

[274] Voigt, S.-O. ; Teufel, T.: Dynamically Reconfigurable Dataflow Architecture for High-Performance Digital Signal Processing on Multi-FPGA Platforms. In: *17th Conference on Field Programmable Logic and Applications*, IEEE Computer Society, Aug. 2007 (FPL '07), S. 633–637

[275] Waingold, E. ; Taylor, M. ; Srikrishna, D. ; Sarkar, V. ; Lee, W. ; Lee, V. ; Kim, J. ; Frank, M. ; Finch, P. ; Barua, R. ; Babb, J. ; Amarasinghe, S. ; Agarwal, A.: Baring it all to software: Raw machines. In: *IEEE Computer* 30 (1997), Sep., Nr. 9, S. 86–93. – ISSN 0018–9162

[276] Wang, Perry H. ; Collins, Jamison D. ; Chinya, Gautham N. ; Jiang, Hong ; Tian, Xinmin ; Girkar, Milind ; Yang, Nick Y. ; Lueh, Guei-Yuan ; Wang, Hong: EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In: *ACM SIGPLAN Notices* 42 (2007), Nr. 6, S. 156–166. – ISSN 0362–1340

[277] Wang, Xiaojun ; Leeser, Miriam: VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 3 (2010), Nr. 3, S. 1–34. – ISSN 1936–7406

[278] Wang, Xiaojun ; Nelson, Brent E.: Tradeoffs of Designing Floating-Point Division and Square Root on Virtex FPGAs. In: *11th Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, 2003 (FCCM '03). – ISBN 0–7695–1979–2, S. 195–203

[279] Wei, Haitao ; Tan, Hong ; Liu, Xiaoxian ; Yu, Junqing: StreamX10: a stream programming framework on X10. In: *Proceedings of the 2012 ACM SIGPLAN X10 Workshop*, ACM, 2012 (X10 '12). – ISBN 978–1–4503–1491–6, S. 1:1–1:6

[280] Wilkes, M. V.: The early British computer conferences. Cambridge, MA, USA : MIT Press, 1989 (1951). – ISBN 0–262–23136–0, Kapitel The Best Way to Design an Automatic Calculating Machine, S. 182–184

[281] Woh, Mark ; Lin, Yuan ; Seo, Sangwon ; Mahlke, Scott ; Mudge, Trevor ; Chakrabarti, Chaitali ; Bruce, Richard ; Kershaw, Danny ; Reid, Alastair ; Wilder, Mladen ; Flautner, Krisztian: From SODA to scotch: The evolution of a wireless baseband processor. In: *41st International Symposium on Microarchitecture*, IEEE Computer Society, 2008 (MICRO-41 2008). – ISBN 978–1–4244–2836–6, S. 152–163

[282] Woods, Andrew ; Inggs, Michael ; Langman, Alan: Accelerating a Software Radio Astronomy Correlator using FPGA co-processors. In: *Symposium on Application Accelerators in High Performance Computing*, 2009 (SAAHPC '09). – `http://saahpc.ncsa.illinois.edu/09/papers/Woods_paper.pdf`

[283] Xia, Yinglong ; Prasanna, Viktor K. ; Li, James: Hierarchical Scheduling of DAG Structured Computations on Manycore Processors with Dynamic Thread Grouping. In: *Proceedings of the 15th international conference on Job scheduling strategies for parallel processing (JSSPP '10)*, Springer-Verlag, 2010 (LNCS 6253). – ISBN 3–642–16504–4, 978–3–642–16504–7, S. 154–174

[284] Xu, Yan ; Muller, O. ; Horrein, P. ; Petrot, F.: HCM: An abstraction layer for seamless programming of DPR FPGA. In: *22nd Conference on Field Programmable Logic and Applications*, 2012 (FPL '12). – ISBN 978–1–4673–2257–7, S. 583–586

[285] Yang, Depeng ; Peterson, Gregory: Performance Comparison of Cholesky Decomposition on GPUs and FPGAs. In: *Symposium on Application Accelerators in High Performance Computing*, 2010 (SAAHPC '10). – `http://saahpc.ncsa.illinois.edu/10/papers/paper_45.pdf`

[286] Yang, Depeng ; Peterson, Gregory ; Li, Husheng: High Performance Reconfigurable Computing for Cholesky Decomposition. In: *Symposium on Application Accelerators in High Performance Computing*, 2009 (SAAHPC '09). – `http://saahpc.ncsa.illinois.edu/papers/Yang_paper.pdf`

[287] Yelick, Katherine ; Bonachea, Dan ; Chen, Wei-Yu ; Colella, Phillip ; Datta, Kaushik ; Duell, Jason ; Graham, Susan L. ; Hargrove, Paul ; Hilfinger, Paul ; Husbands, Parry ; Iancu, Costin ; Kamil, Amir ; Nishtala, Rajesh ; Su, Jimmy ; Welcome, Michael ; Wen, Tong: Productivity and performance using partitioned global address space languages. In: *Proceedings of the international workshop on Parallel symbolic computation*, ACM, 2007 (PASCO '07). – ISBN 978–1–59593–741–4, S. 24–32

[288] Yu, Haiqian ; Leeser, M.: Automatic Sliding Window Operation Optimization for FPGA-Based Computing Boards. In: *14th Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society, April 2006 (FCCM '06). – ISBN 0–7695–2661–6, S. 76–88

[289] Zerbino, Daniel R. ; Birney, Ewan: Velvet: algorithms for de novo short read assembly using de Bruijn graphs. In: *Genome Research (CHS Press)* Advance March 18 (2008), S. 821–829

[290] Zhang, David ; Li, Qiuyuan J. ; Rabbah, Rodric ; Amarasinghe, Saman: A lightweight streaming layer for multicore execution. In: *ACM SIGARCH Computer Architecture News (CAN)* 36 (2008), Nr. 2, S. 18–27. – ISSN 0163–5964

[291] Zhang, Yan ; Shalabi, Y.H. ; Jain, R. ; Nagar, K.K. ; Bakos, J.D.: FPGA vs. GPU for sparse matrix vector multiply. In: *Conference on Field-Programmable Technology*, IEEE Computer Society, 9-11 2009 (FPT '09). – ISBN 978–1–4244–4377–2, S. 255–262

[292] Zhou, Bo ; Hu, X. S. ; Chen, Danny Z. ; Yu, Cedric X.: A multi-FPGA Accelerator for Radiation Dose Calculation in Cancer Treatment. In: *Symposium on Application Specific Processors*. Los Alamitos, CA, USA : IEEE Computer Society, 2009 (SASP '09). – ISBN 978–1–4244–4939–2, S. 70–79

[293] Zhuo, Ling ; Prasanna, Viktor K.: Design Tradeoffs for BLAS Operations on Reconfigurable Hardware. In: *International Conference on Parallel Processing*, IEEE Computer Society, 2005 (ICPP '05). – ISBN 0–7695–2380–3, S. 78–86

[294] Zhuo, Ling ; Prasanna, Viktor K.: High-Performance Designs for Linear Algebra Operations on Reconfigurable Hardware. In: *IEEE Transactions on Computers* 57 (2008), Nr. 8, S. 1057–1071. – ISSN 0018–9340

[295] Zierke, Stephanie ; Bakos, Jason D.: FPGA Acceleration of the phylogenetic likelihood function for Bayesian MCMC inference methods. In: *BMC Bioinformatics* 11 (2010), April, Nr. 184