Karlsruher Institut für Technologie

Reducing the Complexity of Heterogeneous Computing: A Unified Approach for Application Development and Runtime Optimization

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

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

genehmigte

Dissertation

von

Mario Kicherer

aus Schramberg

Tag der mündlichen Prüfung: 18. Dezember 2014

Erster Gutachter: Prof. Dr. Wolfgang Karl

Zweiter Gutachter: Prof. Dr. Frank Bellosa

Zusammenfassung

Mit zunehmender Verbreitung heterogener Parallelsysteme müssen sich Programmierer von rechenintensiven Anwendungen neuen Herausforderungen stellen, um selbst das Potential von Allzweckrechnern überhaupt ansatzweise zu nutzen. Ein grundlegendes Problem hierbei ist, dass die weitverbreiteten Programmiersprachen nicht für die Programmierung sogenannter Beschleuniger wie Grafikprozessoren vorgesehen sind und für die Code-Generierung neue, zum Teil architektur- und herstellerspezifische Sprachen genutzt werden müssen. Erste Ansätze, die dieses Problem lösen, sind einheitliche Programmiersprachen, wie OpenCL oder OpenACC, die mittels dynamischer Code-Erzeugung für unterschiedliche CPU- und Beschleunigerarchitekturen übersetzt werden können. Mit diesen Ansätzen entfällt allerdings nur die Notwendigkeit sich mit unterschiedlichen Sprachen beschäftigen zu müssen. Oft ist anstelle einer Portierung sogar ein anderer Algorithmus notwendig, der den grundlegenden Architekturmerkmalen besser entspricht. Um die Rechensysteme dann auch effizient zu nutzen, muss neben architekturspezifischen Eigenheiten auch das Zusammenspiel der unterschiedlichen Recheneinheiten in einem System beachtet werden. Soll eine Anwendung daher nicht nur auf bestimmten bekannten Systemen ausgeführt werden, wird durch die Vielfalt an möglichen Systemkonfigurationen und Optimierungsmöglichkeiten eine automatische Anpassung der Anwendung notwendig. Eine solche automatische Anpassung hat unter anderem auch für die Ausführung auf bekannten Systemen den Vorteil, dass – mit entsprechendem Funktionsumfang – auf unvorteilhafte Ereignisse wie Ressourcenkonflikte in einem Mehrbenutzersystem oder Fehler während der Ausführung, z. B. ausgelöst durch die zu erwartetende steigende Anfälligkeit der Schaltungen durch die fortlaufende Miniaturisierung, reagiert werden kann ohne eventuell vermeidbare Leistungseinbußen oder Datenverlust hinnehmen zu müssen.

Neben den Vorteilen während des Einsatzes auf Endnutzersystemen, bietet ein solcher Automatismus auch Vorteile für den Entwicklungsprozess selbst. Zum einen können verschiedene Optimierungsmöglichkeiten automatisch durchgeführt und evaluiert werden, was die Komplexität des Quellcodes verringert und die Produktivität erhöht. Zum anderen können Techniken zur Fehlererkennunung zum Beispiel auch genutzt werden, um die Suche nach Fehlern im Quellcode zu unterstützen.

Für einige der erwähnten Vorteile – insbesondere für verschiedene Optimierungsmöglichkeiten – wurden bereits eine Vielzahl an Implementierungen auf Basis von Laufzeitsystemen vorgeschlagen, die eine entsprechende automatische Anpassung vornehmen. Diese Arbeiten beschränken sich bis auf wenige Ausnahmen allerdings nur auf bestimmte Anpassungen beziehungsweise Optimierungen. In dieser Arbeit wird stattdessen ein grundlegendes Rahmenwerk vorgeschlagen, mit dem Mechanismen für unterschiedliche Anpassungen in einem Ansatz zusammengeführt werden können um nicht nur entstehende Synergieeffekte zur Anwendungslaufzeit zu nutzen sondern auch um den Entwicklungsprozess einer Anwendung zu unterstützen. Dieses Rahmenwerk basiert auf einem neuen Laufzeitsystem, das während der Anwendungslaufzeit den Systemzustand analysiert und den implementierten Mechanismen damit erlaubt ihre Entscheidungen entsprechend dynamisch anzupassen.

Ein Problem für Anwendungen sind zum Beispiel die einzelnen Hardware-spezifischen Implementierungen der gewählten Algorithmen. Die hierfür verwendeten Programmiermodelle benötigen in der Regel eigene Laufzeitbibliotheken oder Compiler, die zur Laufzeit den Code für die vorhandene Hardware übersetzen. Sind diese auf dem aktuellen System nicht installiert, da z. B. die entsprechende Hardware nicht vorhanden ist, bricht die Anwendungsausführung ab ohne die Möglichkeit die Ausführung auf die CPU zu beschränken. Verwandte Arbeiten setzen hier auf vereinheitlichte Programmiermodelle, die den Quellcode zur Laufzeit für vorhandene Recheneinheiten kompilieren. Durch ihren einheitlichen Ansatz wird es allerdings schwieriger oder teilweise auch unmöglich architekturspezifische Merkmale auszunutzen. Um dieses Problem zu lösen, wird in dieser Arbeit ein entkoppelnder Ansatz für die Anwendungsentwicklung vorgeschlagen, der Hardware-spezifische Implementierungen in Bibliotheken auslagert. Das Laufzeitsystem wird dann genutzt um zur Anwendungslaufzeit nach geeigneten Implementierungen zu suchen und diese zu laden, falls sie den gegebenen Anforderungen der Anwendung entsprechen und ausgeführt werden können. Mit diesem Ansatz werden nicht nur die strikten Abhängigkeiten reduziert und damit die Portabilität der Anwendung erhöht. Er ermöglicht es außerdem bereits fertiggestellten Anwendungen nachträglich noch von neuen Implementierungen zu profitieren, die zum Beispiel von Geräteherstellern für neue Recheneinheiten optimiert wurden.

Insbesondere in wissenschaftlichen Bereichen kommt es außerdem vor, dass nicht nur unterschiedliche Implementierungen von Algorithmen entwickelt werden sondern auch neue Algorithmen und Verfahren. Dabei kann sich die Folge von abzuarbeitenden Rechenaufgaben unterscheiden und der Entwickler wäre ähnlich wie bei verschiedenen Implementierungen zusätzlich noch dafür verantwortlich, die jeweiligen Verfahren und die zugehörigen Implementierungen zu untersuchen und die beste auszuwählen. Um diesen Schritt ebenfalls zu automatisieren, kann ein Entwickler dem Laufzeitsystem mehrere alternative Folgen von Rechenaufgaben übermitteln um an ein Ergebnis zu gelangen und das Laufzeitsystem nutzt dann seine Mechanismen um den resultierenden bedingten Graph bestmöglich auf das gegebene heterogene System abzubilden.

Um aus der Menge der geeigneten Implementierungen und Recheneinheiten für eine Rechenaufgabe die beste auszuwählen, nutzt das Laufzeitsystem ähnlich wie verwandte Arbeiten eine Datenbank, in der Messwerte vergangener Ausführungen dieser Implementierungen gespeichert sind. Basierend auf diesen Daten wird die Implementierung und die Recheneinheit mit den besten Leistungswerten gemäß dem gewählten Optimierungsziel, z. B. die Reduzierung der Laufzeit, bestimmt und ausgeführt. Während der Ausführung können allerdings Ereignisse auftreten, die diese Messwerte ungültig machen. Konkurrierende Anwendungen und fehleranfällige Recheneinheiten können die Laufzeit erhöhen, da Berechnungen verzögert werden oder eine Wiederholung der Berechnungen notwendig wird. In homogenen Systemen kann im besten Fall einfach auf eine andere freie Recheneinheit gewechselt werden. In heterogenen Systemen weisen die Recheneinheiten allerdings unterschiedliche Leistungswerte auf und ein Wechsel erfordert zusätzliche Operationen, wie z. B. Datentransfers. Daher kann es unter Umständen vorkommen, dass ein Wechsel die Ausführungszeit mehr verlängert als eine konkurrierende Anwendung oder notwendige Wiederholungen. Während wenige verwandte Arbeiten grundsätzlich konkurrierende Anwendungen in Betracht ziehen, werden die Auswirkungen von Fehler von ähnlichen Arbeiten in diesem Bereich nicht berücksichtigt. Diese Arbeit hingegen verfügt über verschiedene Mechanismen, die die Erkennung von konkurrierenden Anwendungen als auch von auftretenden Fehlern während der Ausführung bei Bedarf ermöglichen und weitere Entscheidungen entsprechend anpassen.

Zur grundsätzlichen Entscheidungsfindung setzen verwandte Arbeiten auf den „Heterogeneous Earliest Finish Time”-Algorithmus (HEFT). Der Vorteil dieses Algorithmus ist, dass er verhältnismässig einfach zu implementieren ist, gute Ergebnisse liefert und durch die beschränkte lokale Sicht nach einer Entscheidung für eine Rechenaufgabe sofort mit der

Ausführung begonnen werden kann und nicht auf folgende Entscheidungen gewartet werden muss. Die Nachteile sind, dass zu Anfang getroffene Entscheidungen später Verbesserungen verhindern können und unter Umständen nicht das globale Optimum gefunden wird oder dass im Falle gesetzter Beschränkungen und Nebenbedingungen sogar eine korrekte Ausführung unmöglich wird. Um solche Fälle zu vermeiden, bietet das in dieser Arbeit vorgestellte Laufzeitsystem die Möglichkeit im Voraus mehrere hypothetische Ablaufpläne mit gewählten Eigenschaften und den daraus resultierenden Operationen zu erstellen und zu evaluieren. Mit dieser Technik werden zusätzlich auch aufwändigere Algorithmen ermöglicht, wie zum Beispiel Simulated Annealing, für das eine Vielzahl an Ablaufplänen evaluiert werden müssen, bis eine finale sinnvolle Entscheidung getroffen wird. Um Leistungseinbußen durch die zum Teil aufwändigen Verfahren zu vermeiden, können vorteilhafte Ablaufpläne gespeichert und für weitere Anwendungsläufe wiederverwendet werden.

Mit dem in dieser Arbeit vorgestellten Rahmenwerk wird somit ein umfassender Ansatz mit dem Ziel, die Komplexität von Anwendungen für heterogene Parallelsysteme zu reduzieren, vorgestellt, der den Entwicklungsprozess vereinfacht, sowie automatisch die Anwendungsausführung auf unterschiedlichen Rechensystemen verbessert.

Zusammengefasst trägt diese Arbeit die folgenden einzelnen Neuerungen bei:

- einen entkoppelten Ansatz für die Anwendungsentwicklung, der die Portabilität der Anwendungen erhöht und es ermöglicht nachträglich optimierte Implementierungen zu nutzen
- eine dynamisch wählbare Kombination verschiedener Mechanismen zur Erkennung und effizienten Behandlung von konkurrierenden Anwendungen und Fehlern während der Ausführung
- einen Mechanismus zur Vorabanalyse unterschiedlicher Scheduling-Algorithmen und Optimierungsansätze, der es außerdem erlaubt aufwändigere Algorithmen, wie z.B. Simulated Annealing, zur Suche nach dem globalen Optimum zu nutzen
- leichtgewichtige Methoden zur Integration des Laufzeitsystems in C- und Matlab-Anwendungen um möglichst wenig Komplexität im Anwendungsquellcode selbst einzuführen

Im Fall einer bildgebenden Anwendung aus der Medizintechnik konnte das Laufzeitsystem mittels der Vorabanalyse mehrerer Ablaufpläne eine Verkürzung der Laufzeit um knapp 20 % gegenüber des HEFT-Algorithmus erreichen. In einem Experiment in dem Fehler unter gegebenen Umständen in den Quellcode injiziert wurden, erzielte das Laufzeitsystem durch die geschickte Wahl der Recheneinheit gegenüber naiven Ansätzen eine Reduktion der Laufzeit jeweils um 50 % und 30 %. Ebenfalls fand das Laufzeitsystem bei Anwendungen einer Benchmarksammlung auf einem System mit mehreren unterschiedlichen Recheneinheiten erfolgreich die günstigste Wahl.

Abstract

Heterogeneous systems with accelerators promise considerable performance improvements and energy savings at a lower cost than computing on a homogeneous CPU-only system. However, to benefit from this potential, considerable work is required from developers to employ accelerators and to integrate them efficiently in an application.

In scientific publications, the benefit of accelerators for certain algorithms is usually quantified on specific systems with optimized implementations. On differently configured systems or compared with equally optimized implementations for other processing units, however, the performance of accelerators can be much closer to or even worse than executing on a CPU, for example. Therefore, runtime systems are used to dynamically determine and select the best implementation and processing unit for a task in the current system. Similar to the runtime system in this work, most of them use an empirical approach to determine the best combination. First, the execution time of all possible combinations is measured and stored. Afterwards, the combination with the lowest execution time is chosen for further task executions. As the size of the input data also has a significant impact on the execution time in many cases, it is common to store the execution times as function of the input size.

However, besides the input size, other circumstances can have a considerable impact on the benefit and may suddenly change the previous best choice for a task. Most related works expect exclusive usage of a flawless system. During application runtime, competing applications may start to use the same processing unit or faults may abort execution or falsify the results. In both cases, a considerable overhead may occur due to shared usage and necessary restarts of computations. In homogeneous systems, any other free processing unit can be used to avoid a unit causing such an overhead. In heterogeneous systems, however, switching the processing unit may result in a higher overhead than sharing a processing unit or repeating a calculation. Hence, this work considers both events during task execution and provides different mechanisms to detect and react on such events appropriately.

Besides detecting faults at application runtime, automatic fault detection is also a benefit for application development. Usually, development starts with a sequential reference implementation of a task written by problem-oriented developers, e.g., engineers. Afterwards, hardware-oriented developers start to write implementations optimized for specific types of processing units. To determine the correctness of their code, they usually manually compare the results of their implementation with those of the reference implementation. Instead, with this runtime system, this comparison is done automatically and the runtime system can also give valuable additional information for debugging, e.g., if the results differ completely or only in certain parts.

If the hardware-specific implementations work as expected, they are usually statically included in the application. However, every programming model uses an own software stack, e.g., runtime libraries or just-in-time compilers, that imply additional dependencies for the resulting application. If one of these is not available on another system, e.g., because

the respective hardware is not present, application startup will fail without the opportunity for a fallback to CPU-only computation. For this issue, related work proposes unified programming models that use just-in-time compilation to generate code for the actually available hardware. However, using a one-for-all approach also makes it difficult or even impossible to exploit hardware-specific features.

Therefore, this work proposes a decoupled development concept. Following this concept, general application code and hardware-specific task implementations are not only developed by different persons but are also packaged in independent binaries. During application runtime, the runtime system looks for available implementations and hardware and only loads the implementations which fulfill the requirements of the application and can be executed on the system. Besides increasing the portability of applications while keeping the benefits of hardware-specific optimizations, this approach also enables older applications to automatically benefit from new implementations, e.g., BLAS libraries optimized for new accelerators, if they match the requirements of the application.

Especially in scientific areas, developers not only work on hardware-optimized implementations for specific tasks but also on alternative algorithms that will calculate the same result but require a different sequence of tasks. Manually managing and evaluating the benefit of the different sequences further increases the costs in terms of code complexity and development time. Therefore, with this runtime system, developers can submit tasks in conditional sequences and its task scheduler dynamically determines the best sequence for the current system.

Besides hardware-specific implementations and alternative algorithms, heterogeneous systems offer many possibilities for generic optimizations, e.g., memory-mapping data instead of transfers or different algorithms for the scheduling of task execution and data transfers. However, the benefit of most optimizations depends again on multiple circumstances. For example, the commonly used heterogeneous earliest finish time scheduler is a greedy algorithm and may not find the global optimum if the fastest processing unit has high initialization costs for just-in-time compilation of the code or for transfers of the input data into the device's memory. For most opportunities, related work proposed mechanisms that automatically optimize certain aspects of application execution. However, to gain the best performance, multiple optimizations have to be evaluated and applied if beneficial. While evaluating each optimization individually has been proposed before, determining their impact on each other and the best set of optimizations for application performance becomes increasingly complex. This becomes even worse, if additional objectives come into play like dependability. In order to simplify this problem, this work enables the simulation of application execution on task level during runtime which permits an evaluation of different optimizations and schedulers before choosing the best combination for execution. With such a simulation, it is also possible to employ more demanding scheduling algorithms like simulated annealing which can find a global optimum but require an evaluation of many different schedules before a good decision can be made.

While the proposed runtime system provides several mechanisms that simplify development and improve the performance, an important aspect is also the required effort for integrating the runtime system itself. While the runtime system provides wrapper functions for programming models and script languages like OpenCL and Matlab, this work also introduces a non-intrusive approach based on light-weight instrumentation to include the runtime system with marginal additional effort in C source code.

To summarize, related work proposed solutions for many different problems of heterogeneous computing. However, all have a narrow focus on specific aspects, e.g., assume exclusive usage of a flawless system or perform only specific optimizations. Instead, this work contributes a fundamental framework implemented with an online-learning runtime system

that provides a common basis for different mechanisms to simplify development and make applications more portable, efficient and reliable across different systems. The single contributions of this work are:

- a decoupled development concept to increase portability of applications without loosing performance benefits of hardware-optimized task implementations that also enables already compiled applications to benefit from new task implementations,
- a dynamically selectable combination of mechanisms to detect and efficiently resolve concurrent use of resources and faults during task execution, including a new metric to balance performance benefits with fault susceptibility of implementations and processing units.
- an online simulation of different optimizations and schedulers in advance to execution for case-by-case performance evaluation which also enables demanding and non-greedy scheduling algorithms like simulated annealing
- a non-intrusive integration of the runtime system in script-based Matlab and C applications using a novel light-weight instrumentation technique

In case of an application for preprocessing in medical imaging, this approach was able to reduce the runtime by 20 % compared to the common HEFT algorithm using its simulation of multiple schedules. In an experiment where faults are injected under given conditions in a task implementation, this approach achieved a reduction of the runtime by 50 % and 30 % compared to two naive approaches using a smart task mapping. Furthermore, the runtime system successfully determined the best mapping for the applications of a benchmark suite on a system with multiple processing units.

Acknowledgements

First, I want to thank my advisor Prof. Dr. Wolfgang Karl for giving me the opportunity to conduct this research and for the countless hours he invested to guide me and discuss my work. This greatly influenced and shaped my thinking which was vital for the success of this work. I also thank my co-advisor Prof. Dr. Frank Bellosa for broadening my view and giving me valuable advices during the writing of this thesis.

Furthermore, I want to express my gratitude to my former and current colleagues at the Chair for Computer Architecture and Parallel Processing: Thomas Becker, Michael Bromberger, Rainer Buchty, David Kramer, Oliver Mattes, Fabian Nowak and Martin Schindewolf as well as the colleagues of the CES and CDND for their support, advice and friendship.

Finally, I want to thank my parents Bernadette and Werner as well as my sisters Verena and Lea for their steady support throughout my life. I am also very thankful to my wife Esther and my parents-in-law Gudrun and Günther who made it possible to write this thesis during the first months of our son Ian-Mika. I would also like to use this opportunity to particularly thank my aunt Luzia and uncle Rolf who early encouraged my curiosity and recovered my computer many times after my first, only partially successful experiments.

Publications

[1] Rainer Buchty, David Kramer, Mario Kicherer, and Wolfgang Karl. A Light-Weight Approach to Dynamical Runtime Linking Supporting Heterogenous, Parallel, and Reconfigurable Architectures. In *Architecture of Computing Systems ,– ARCS 2009*, volume 5455/2009 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin / Heidelberg, February 2009.

[2] Mario Kicherer. Design and Implementation of a Low-overhead Run-time System for Self-X Architectures. Diplomarbeit, Universität Karlsruhe (TH), 2008.

[3] Mario Kicherer, Rainer Buchty, and Wolfgang Karl. Cost-aware function migration in heterogeneous systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 137–145, New York, NY, USA, 2011. ACM.

[4] Mario Kicherer and Wolfgang Karl. Heterogeneity-aware Fault Tolerance using a Self-Organizing Runtime System. *ArXiv e-prints*, First Workshop on Resource awareness and adaptivity in multi-core computing, May 2014.

[5] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. Extending a Light-weight Runtime System by Dynamic Instrumentation For Performance Evaluation. In Michael Beigl and Franciso J. Cyzorla-Almeida, editors, *ARCS 2010 Workshop Proceedings*, pages 279–284. VDE, February 2010.

[6] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. Seamlessly portable applications: Managing the diversity of modern heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 8(4):42:1–42:20, January 2012.

Contents

1. Introduction **1**

2. Background and related work **5**
2.1. Heterogeneous systems . . . 5
2.2. Programming models and code generation for heterogeneous systems . . . 8
2.3. Automatic optimization of hardware-specific implementations . . . 12
2.4. Data and task mapping . . . 14
2.5. Enhancing dependability in modern systems . . . 17

3. A unified approach **21**

4. Light-weight integration and transparent task migration **25**
4.1. Introduction . . . 25
4.2. Native C interface . . . 26
4.2.1. Pointer-based function migration . . . 26
4.2.2. Explicit integration using the DLS programming interface . . . 30
4.2.3. Discussion . . . 32
4.3. Integration in Matlab for rapid prototyping . . . 33
4.4. Transparent OpenCL wrapper . . . 34
4.5. Call stack infrastructure . . . 37
4.6. Extensible hardware interface . . . 38
4.7. Ad-hoc work offloading in local networks . . . 39

5. Implementation management and application portability **43**
5.1. Introduction . . . 43
5.2. Related work . . . 44
5.3. Decoupled application development . . . 46
5.3.1. Implications on source code and build systems . . . 46
5.3.2. Assembling applications and implementations . . . 47
5.3.3. Supporting fault diagnostics during development . . . 48
5.4. Balancing requirements and abilities . . . 50
5.5. Evaluation . . . 52
5.5.1. Overhead for matching requirements and abilities . . . 52
5.5.2. Performance with Rodinia benchmarks . . . 53
5.5.3. Use case: random number generation . . . 54
5.5.4. Portable MPI application . . . 55

6. Establishing cost awareness **59**
6.1. Introduction . . . 59
6.2. Learning costs during application execution . . . 61
6.3. Memory management and the impact of data locality . . . 63

6.4. Reacting on competition for resources . . . 66
6.4.1. Passive checks . . . 66
6.4.2. Shared-memory waiting queues . . . 68
6.5. Impact of faults on costs . . . 70
6.5.1. Symptom-based fault detection . . . 72
6.5.2. Fault-aware runtime estimation . . . 77

7. Anticipatory scheduling in heterogeneous systems **79**
7.1. Introduction . . . 79
7.2. Online simulation of task execution . . . 82
7.3. Building blocks for scheduling decisions . . . 83
7.3.1. Task mapping . . . 84
7.3.2. Conditional task graphs . . . 85
7.3.3. Task splitting . . . 88
7.3.4. Taking precautions against faults . . . 93
7.4. Decision making . . . 95
7.5. Evaluation . . . 102
7.5.1. Case study: Preprocessing for medical imaging . . . 103
7.5.2. Fault-tolerant task execution . . . 104

8. Conclusion **111**
8.1. Summary . . . 111
8.2. Outlook . . . 112

Bibliography **115**

Appendix **129**
A. Case study: Preprocessing for medical imaging . . . 129

1. Introduction

Heterogeneous computing has become an acknowledged method to raise the computing power of modern systems for specific application areas while preserving a high performance to cost ratio. In most such systems, one or more general-purpose CPUs are accompanied by specialized logic, usually referred to as "accelerator", like a graphics processor (GPU), digital signal processors (DSP) or field-programmable gate arrays (FPGAs). While GPUs can be found in almost any class of computing device today, from embedded systems to supercomputers, DSPs and FPGAs are more common in specific areas, e.g., DSPs for communications and multimedia in mobile phones or FPGAs in bioinformatics, computer vision and hardware prototyping. Accelerators can be integrated in a system in different ways, e.g., as dedicated chip on a PCI Express card, as additional die in the package like Intel's Atom E6x5C series or as part of the CPU die with AMD's Heterogeneous System Architecture (HSA).

In the past, accelerators were either expensive, highly specialized devices for only one or a few specific tasks, e.g., DSPs, or difficult to program, e.g., FPGAs. Similar, GPUs were highly specialized architectures with a fixed pipeline for graphic-related calculations. With the advent of so-called shader cores, GPUs became programmable but their abilities were still limited and focused on graphics calculations. With the ongoing development and improving functionality, several projects started to provide high-level programming models in order to use these shaders for general-purpose programming. The first approach that gained broad attention was the CUDA project from Nvidia, introduced in 2007. Due to their C-based programming model and powerful GPUs becoming commodity hardware, CUDA significantly lowered the entry threshold for using an accelerator.

The advantage of accelerators is their specialized architecture that enables faster calculations for certain algorithms than the CPU that is designed to provide a high average performance for a wide range of computations. For an application, however, this benefit comes not for free. The algorithms as well as their actual implementations with accelerator-specific programming models mostly differ from their CPU counterpart. Thus, employing an accelerator requires an adaptation or even a complete rewrite of parts of the applications. At best, there already exist libraries that implement certain algorithms on an accelerator, e.g., developed by the manufacturer, and the application only has to call the library without bothering with the details of the accelerator. However, such libraries are still scarce and in most cases, considerable manual work is necessary. Furthermore, even if an implementation is available for a processing unit, the benefit of the processing unit compared to other units in the current system depends on multiple circumstances.

In scientific publications, the benefit of accelerators for certain algorithms is usually quantified on specific systems with optimized implementations for the accelerator. On differently configured systems or compared with equally optimized implementations for other processing units, however, the performance of accelerators can be much closer to or even worse than executing on a CPU [95], for example. Therefore, mechanisms, e.g., based on runtime systems or compilers, were introduced to automatically determine and select the best implementation and processing unit for a task in the current system. While some solutions use static code analysis, most of them use an empirical approach to determine the best combination. First, the execution time of all possible combinations is measured and stored. Afterwards, the combination with the lowest execution time is chosen for further task executions. As the execution times may also vary due to changing input data, e.g., matrices of different sizes, it is common to store the measured values as function of the problem size.

However, besides the input data, other circumstances can have a considerable impact on the benefit and suddenly change the previous best choice for a task. Resource conflicts usually result in an increased runtime due to postponed or shared execution. As today's major operating systems are not involved in the scheduling on accelerators, it is usually not possible for applications to detect competition in advance. Similarly, faults may occur during task execution that falsify the results and require expensive restarts of calculations – if faults during execution are even considered and detection mechanisms are installed. Due to the redundancy in terms of task implementations and processing units, heterogeneous systems cause additional complexity but also offer additional opportunities to, e.g., avoid faulty or overloaded processing units. While on homogeneous systems, any free processing unit is a suitable alternative, switching the processing unit in a heterogeneous system may cause a higher overhead than sharing a unit or repeating a calculation. Hence, special care is required during such decisions to avoid additional performance loss.

Especially in scientific areas, developers not only work on hardware-optimized implementations for specific tasks but also on alternative algorithms that may fit better to certain architectures. If the alternative implementations require additional preparations, e.g., a transformation of the data structures, additional tasks might be necessary depending on the chosen algorithm. With the common approaches that automatically select the best processing unit for a task, a developer is responsible for selecting the best algorithm and correctly submitting all required tasks for the chosen algorithm. However, manually managing and evaluating the benefit of different task sequences for the current system further increases the costs in terms of code complexity and development time.

The hardware-specific implementations for each task are usually statically included in the application. However, every programming model uses an own software stack, e.g., runtime libraries or just-in-time compilers, that imply additional dependencies for the resulting application. If one of these is not available on another system, e.g., because the respective hardware is not present, application startup will fail without the opportunity for a fallback to CPU-only computation. For this issue, related work proposes unified programming models that use just-in-time compilation to generate code for the currently available hardware. However, using a one-for-all approach also makes it difficult or even impossible to exploit hardware-specific features.

Besides hardware-specific implementations and alternative algorithms, heterogeneous systems offer many possibilities for generic optimizations, e.g., memory-mapping data instead of transfers or different algorithms for the scheduling of task execution and data transfers. However, the benefit of most optimizations depends again on multiple circumstances. For example, the commonly used heterogeneous earliest finish time (HEFT) scheduler is a greedy algorithm and may not find the global optimum if the fastest processing unit has

high initialization costs for just-in-time compilation of the code or for transfers of the input data into the device's memory. For most opportunities, related work proposed mechanism that automatically optimize certain aspects of application execution. However, to gain the best performance, multiple optimizations should be evaluated and applied if beneficial. While evaluating each optimization individually has been proposed before, determining their impact on each other and the best set of optimizations for application performance becomes increasingly complex. This becomes even worse, if additional objectives come into play like dependability.

To conclude, a multitude of challenges await developers while adding support for heterogeneous systems to their applications and optimizing the performance. For example, after writing implementations for the different hardware, including them into the application decreases its portability. In addition, their benefit depends on multiple circumstances that may even vary during application runtime due to competition for resources or occurring faults. Considering and handling all these challenges in the source code requires considerable efforts from a developer and makes the development and the resulting application more complex. However, this complexity can be reduced for a developer if the application is decoupled from the underlying system and the handling of the recurring problems is left to a dedicated mechanism. Such an abstraction not only simplifies application development and maintains performance in different situations but can also automatically optimize different aspects of application execution and promote new research opportunities.

In this work, a new framework is introduced that provides such an abstraction in order to reduce the complexity of heterogeneous systems for application developers. This framework constitutes a unified approach that simplifies application development and automatically adjusts application execution during the runtime. With the initial mechanisms implemented on top of the framework, this work demonstrates how mechanisms for different objectives can be integrated in one approach and how they can benefit from each other. In Figure 1.1, an overview of the approach and the implemented objectives in this work is shown. An application can submit tasks and the mechanisms of this work map these tasks to the available heterogeneous hardware before returning the results of the calculations back to the application. Hence, these mechanisms abstract the details of task processing in order to simplify the application code and to make application execution more portable, efficient and reliable across different heterogeneous systems. This abstraction of the heterogeneous hardware is based on an online-learning runtime system that dynamically evaluates the system state and chooses the best mapping of the application's tasks to the hardware under the current conditions. In addition, the mechanisms not only reduce the complexity for applications but the knowledge they gather during execution is also valuable for developers to improve the application, e.g., for performance analysis or debugging.

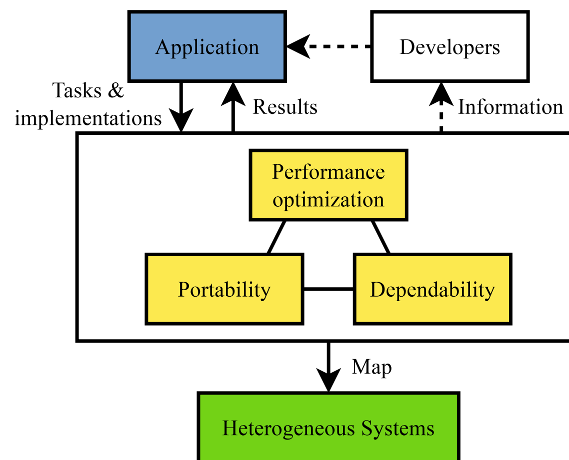


Figure 1.1.: Basic principle of this work

2. Background and related work

In this chapter, the necessary background for this work is introduced and related works are discussed. First, the different types of processor and system architectures considered in this work are described. The second section presents available methods to develop implementations of algorithms for different types of processor architectures. In order to maximize performance of such implementations on differently configured systems, approaches to automatically tune an implementation to specific hardware are described in Section 3. Afterwards, existing mechanisms to optimize the data and task mapping in a heterogeneous system are introduced and compared with this work. This chapter is concluded with a selection of different approaches that increase the dependability of modern systems. Parts of this chapter are based on prior publications [76, 79, 77].

2.1. Heterogeneous systems

In computer architecture, the term "heterogeneous system" usually refers to a system that contains different processing units. These processing units can, for example, differ in the used architecture but also in more detailed aspects like different variants of the same architecture. Besides the differences between processing units, heterogeneous systems also differ in how these processing units are connected. In distributed computing, the single nodes may contain different hardware, e.g., due to different stages of extension. Inside a node, the system may contain one or more different CPUs or additional accelerators like GPUs. In such a case, the motherboard usually contains multiple sockets for the CPUs and the accelerators are integrated using a dedicated board, e.g., connected over PCI Express. Furthermore, single chips can be heterogeneous in the inside. Examples for on-chip heterogeneity are different CPU cores, e.g., ARM's BigLittle architecture [100], integrated GPUs, e.g., architectures from Intel [168] and AMD's Fusion or Heterogeneous System Architecture (HSA) concept [34], or other specialized cores, e.g., for SIMD execution on the Cell BE processor [30].

Although this work includes a mechanism for migrating a task to a remote system, the runtime system presented in this work mainly focuses on heterogeneity below the node level. In the following, common examples for such systems and processor architectures in general-purpose computers are described with a focus on the important properties for application developers.

The CPU parts in today's general-purpose computers are usually multicore architectures similar to the example shown in Figure 2.1. In this example, the CPU contains four cores

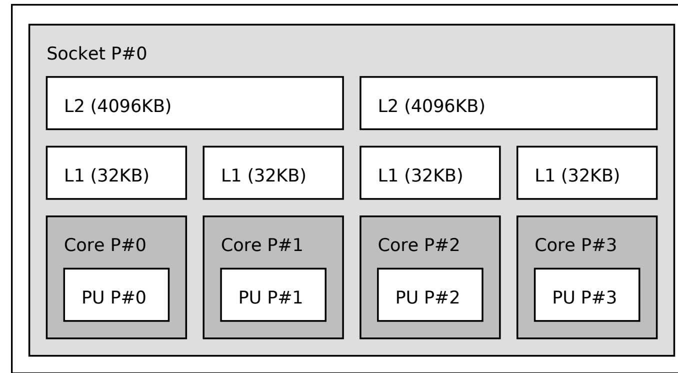


Figure 2.1.: Intel Core2 CPU Q6600 with four cores

and each core has a private L1 cache and two cores share an L2 cache. Hence, besides cache optimizations, developers also have to parallelize their code in order to achieve good performance. Similarly, additional efforts from developers are required to benefit from extensions of common instruction set architectures. One well-known type of extensions are instructions for single instruction, multiple data (SIMD) processing. Modern examples of SIMD extensions on x86 architectures are the different versions of SSE and AVX that each introduced new operations and larger registers to increase the amount of data processed in parallel. Other recent examples for extensions of the x86 instruction set architecture were, for example, AES and TSX that introduced support for AES encryption and transactional memory in hardware.

In desktop and server systems, the most common type of on-chip heterogeneity found today are CPUs with an integrated GPU – commonly referred to as accelerated processing unit (APU). Given the difference in number of transistors – 7.1 billion for the Nvidia Kepler architecture [114] and 2.41 billion combined for CPU and GPU on AMD's Kaveri chips – or memory bandwidth – 144 GB/s on a Nvidia Tesla C2050 [18] and 29 GB/s on an AMD Llano APU [25] – it is not surprising that an integrated GPU provides lower performance than a dedicated GPU. However, one of the considerable benefits of the integrated approach is the short distance between the CPU and GPU and the resulting low latency compared to the rather slow communications over interconnect hardware like PCI express.

In the first generation of such chips for desktop and server systems, the CPU and GPU cores already shared the same physical memory of the system [25, 142]. However, each of them is assigned a separate part of the memory with an own address space. Therefore, data has to be transferred from the CPU to the GPU part before starting execution on the GPU as shown in Figure 2.2. Daga et al. give an overview of the AMD Fusion architecture Zacate and present performance results using selected benchmarks [34]. Their results show that this initial version of the Fusion architecture profits from the physically shared memory but not fully leverages its potential as memory transfers are still required.

To make things worse, actually transferring data to the GPU memory requires two copies without further preparations. As shown in the top left of Figure 2.3, the data is first copied into a pinned memory region. Pages in pinned regions must not be evicted into background memory and thus a concurrent DMA engine can safely copy the data into GPU memory. One extra copy operation can be avoided if the application explicitly requests pinned memory as storage for its data as shown in the bottom left of the Figure 2.3. Some accelerators also offer the opportunity to pass addresses of the host RAM to the accelerators. In such a case, each single access of the accelerator to this area is separately rerouted over the interconnect as shown in the top right of Figure 2.3. While this simplifies

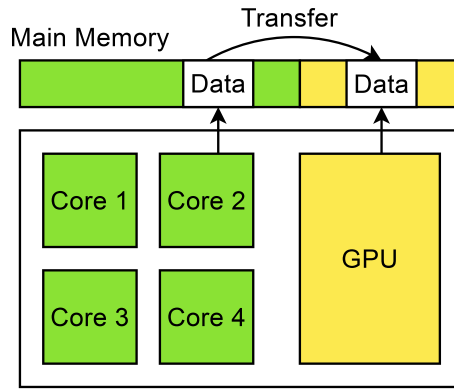


Figure 2.2.: Dedicated memory areas for CPU and GPU require additional data transfers

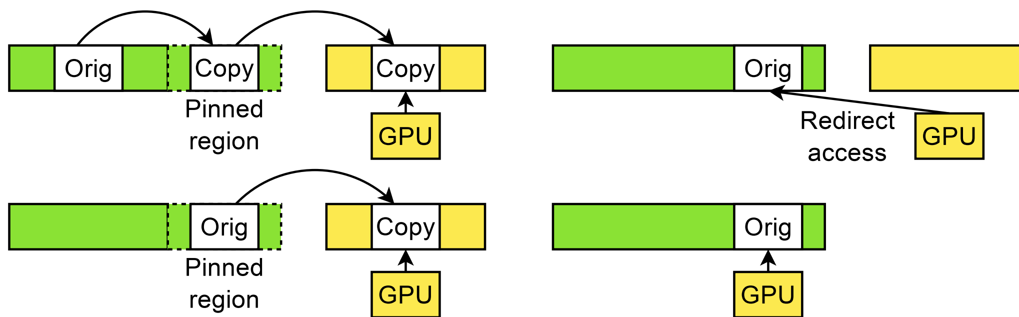


Figure 2.3.: Different possible access methods with modern accelerators

application development, it can significantly decrease the performance.

In order to avoid copies at all, newer CPU architectures like AMD's Kaveri or Intel's Haswell remove this limitation and enable unrestricted access from both parts to the whole memory as illustrated in the bottom right of Figure 2.3. Similar to the previous method this simplifies development as pointers stay valid but this approach also avoids expensive data transfers.

On system level, a common form of heterogeneous systems are combinations of CPUs with one or more dedicated accelerators like a GPU. Typically, these accelerators reside on a separate board connected over PCI Express and possess an own memory as illustrated in Figure 2.4. Contrary to an APU, these accelerators have an own physical memory in order to avoid the high latency to the host RAM for each access and to adapt the memory interface to their needs, e.g., a high-bandwidth interface on a GPU to fetch multiple values in parallel. Other examples for accelerators besides GPUs are field-programmable gate arrays (FPGAs), Intel's new Many Integrated Core architecture (MIC) [66, 84] or Clearspeed accelerators [82].

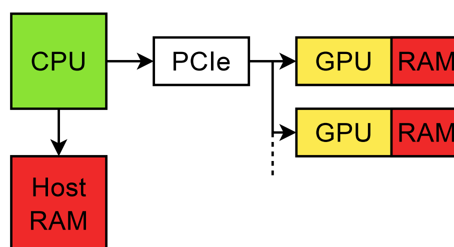


Figure 2.4.: Example of a typical heterogeneous system with a CPU and GPUs

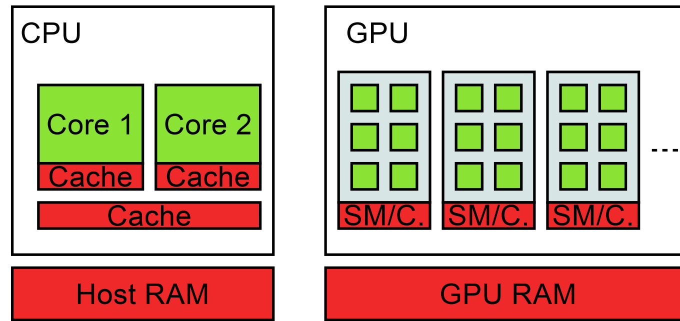


Figure 2.5.: Comparison of the simplified architectures of CPUs and GPUs

Due to their comparatively low price and familiar programming models, GPU-based calculations gained much attention in the recent years. To take a closer look a their benefit for general-purpose computations, a comparison of a simplified version of a CPU with four large general-purpose cores and a GPU with many small cores is shown in Figure 2.5. Each core of the CPU has an own cache in addition to a large cache that is shared among the cores. Instead, the GPU cores are organized in blocks and each block has a shared memory that can be either used explicitly or configured as a cache in newer GPUs. This kind of architecture is well suited for computations with a high degree of calculations per memory access and a large amount of mostly independent, parallel calculations.

2.2. Programming models and code generation for heterogeneous systems

To benefit from heterogeneous systems, developers have to consider the architecture-specific properties of a processing unit. On a CPU, a key to good performance is usually the parallel execution of an application. However, established languages were not designed with parallelism in mind and developers have to use special libraries or specific extensions to those languages to create and synchronize multiple threads that will perform the calculations.

One of the basic methods to create parallel applications for multicore CPUs is the POSIX threads API (pthreads). With simple functions, this API enables a developer to spawn threads and synchronize them, e.g., using mutexes or barriers. Contrary to forking processes, this enables simple execution of parallel calculations in shared memory. Due to its low-level approach, pthreads is mostly used as basis for runtime systems and programming models on a higher abstraction level.

One of these programming models is OpenMP (Open Multi-Processing). With OpenMP, developers can annotate their code with preprocessor pragmas and the compiler will transform these code blocks for parallel execution. One of the simplest methods to parallelize code is annotating `for` loops. In the following example, a `for` loop that increments an array a and stores the result in array b is shown:

```
#pragma omp parallel for
for (i=0; i<count; i++)
        b[i] = a[i] + 1;
```

With the pragma, OpenMP will automatically distribute the loop iterations among the available CPU cores. For more complex loops, OpenMP also offers the possibility to control the parallelization like marking variables as shared or private to the single threads. The benefit of this pragma-based approach is that, if a compiler does not support OpenMP, these pragmas are ignored by default and the application will be executed sequentially. In

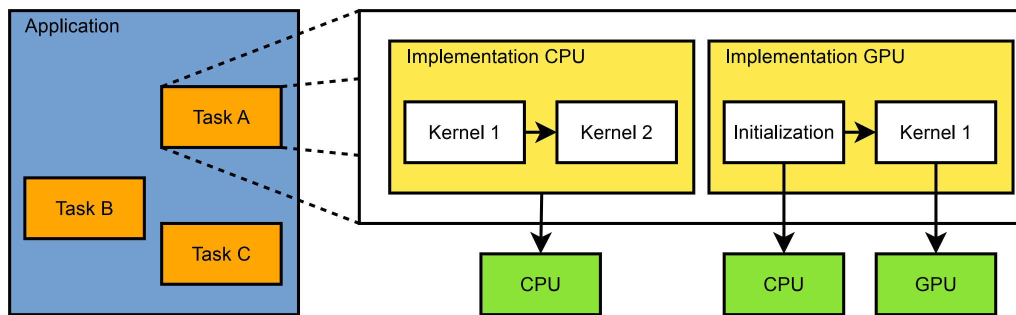


Figure 2.6.: Correlation of application, task, implementation, kernel and processing unit

addition to the `parallel for` pragma, OpenMP also provides other types of parallel regions as well as synchronization constructs. Other common approaches to exploit parallelism are, for example, Intel's Thread Building Blocks (TBB) and Cilk Plus for shared memory systems and MPI libraries for distributed systems [134, 7].

As accelerators usually have different ISAs or a considerably different execution model than CPUs, most of them require own languages and programming models that can be similar to familiar C-based languages, e.g., for GPUs, or vastly different, e.g., VHDL for FPGAs. An exception are accelerators based on Intel's MIC architecture. As these accelerators use a large number of x86 processors with SIMD extensions, code written with familiar programming models like OpenMP can be reused.

For GPUs, a programming model that gained wide attention is Nvidia CUDA. In CUDA, implementations are divided in a host and a device part. The host part is written in common CPU languages like C or Fortran and responsible for the set-up of the device, e.g., to initiate necessary data transfers. The device part is written in a C/C++ dialect and usually processes only one work item. For each work item, a new thread on the GPU is created that executes this compute kernel with the associated item. A crucial point that is even more important on GPUs is to adapt the code to the characteristics of the GPU architecture in order to come even close to peak performance. For example, as previously indicated in Figure 2.5, the GPU cores are grouped in blocks that share a local memory and enable synchronization between the threads executing in this block. If different threads will access the same data or if the algorithm benefits from sharing intermediate results, efficiently using this shared memory with block synchronization can significantly improve the performance.

In the following, the term "implementation" refers to an algorithm written with a certain programming model or language. This includes the host and device part for accelerator implementations. If not stated otherwise, the term "compute kernel" or "kernel" refers to code that executes one of the compute-intensive parts of an application and not to the kernel of the operating system. Hence, an implementation contains code for initialization and one or more associated compute kernels. On the other hand, the term "task" refers to a specific compute-intensive part of the application in this work – not to a process in the operating system – and there may be multiple implementations that execute this task. To summarize, an application for heterogeneous systems may contain multiple tasks where each task can be executed by one of multiple implementations and each implementation may contain one or more compute kernels as shown in Figure 2.6. In this example, an application contains three tasks `A`, `B` and `C`. Task `A` can be either executed on a CPU or on a GPU. The CPU implementation contains two compute kernels, e.g., two parallel regions, and the GPU implementation contains an initialization routine that executes on the CPU and a kernel that executes on the device.

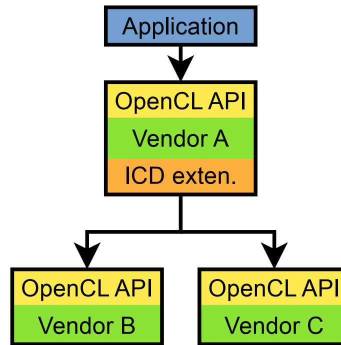


Figure 2.7.: The ICD extension enables multiple vendor-specific OpenCL libraries in one process

As CUDA targets only Nvidia GPUs, a competing standard called OpenCL (Open Computing Language) has been introduced that provides a common API to program different kinds of processing units. Compared with CUDA, OpenCL follows a similar design but differs in several points [86]. For example, as OpenCL tries to be a common denominator for different types of processing units and vendors, usually not all features of new architectures are exploitable with OpenCL contrary to the respective native programming model. Furthermore, as OpenCL is a library-based approach, it does not require a special compiler for the host code. However, a considerable amount of code changes are necessary to set-up the execution on a OpenCL device. To support different types of processing units, the source code of the OpenCL kernel is usually shipped with the application and dynamically compiled for the available hardware on the respective system. As OpenCL only defines an API and the C dialect for the kernel code, each vendor provides an own OpenCL library with just-in-time compiler for his devices. To use different devices and OpenCL libraries with the same API in the same application nonetheless, the installable client driver extension (ICD) has been introduced. This extension enables one OpenCL library to load other OpenCL libraries and to pass through the API calls to them as shown in Figure 2.7.

The logical organization of the processing units available through the different OpenCL libraries is shown in Figure 2.8. Using the OpenCL API, the application receives a list of so-called platforms – usually one platform for each vendor and OpenCL library. Each platform may provide multiple types of processing units and contain a variable number of processing units, called compute devices. Furthermore, each device consists of one more processing elements that each usually resembles one core.

To start a calculation with OpenCL, the developer first has to choose a platform and a device. He can either choose a specific device, request a specific device type or use the default device of the library. Besides further generic set-up procedures, the developer is also responsible for making the necessary data available to the OpenCL device. For this task, OpenCL provides two methods similar to other approaches: data transfer and data mapping. During a transfer, data is simply copied from one memory to another. With data mapping, two regions in the respective memories are coupled. At any time, only one of them may be accessed and control over the active region can be passed to or from the device using special functions of the OpenCL API. For example, in case the OpenCL device is the CPU, using this method enables the OpenCL library to safely use the original data without copying it to a separate region. If the application would issue a transfer instead, the data has to be copied to a separate region as the application is allowed to modify the original data during execution of the OpenCL kernel.

If the necessary data is accessible from the device, the developer has to initiate the actual

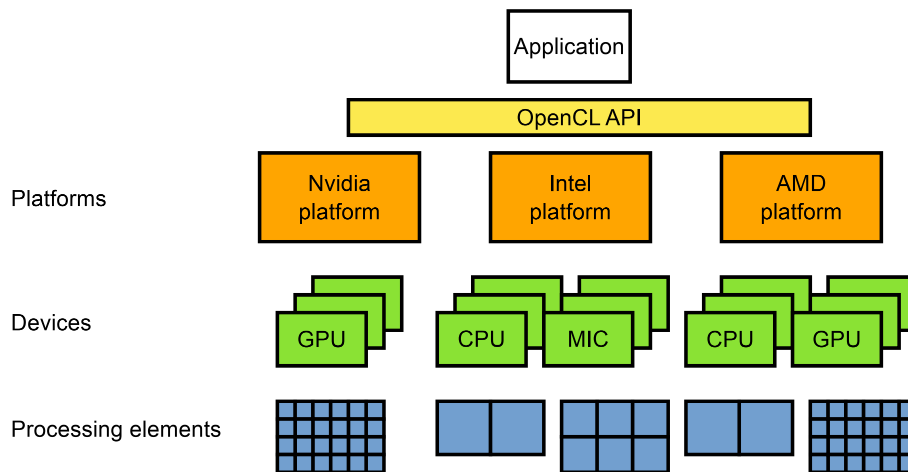


Figure 2.8.: Logical organization of processing units in OpenCL

kernel execution. Basically, he has to pass the kernel source code to the library, that will compile the code for the chosen device, and submit a new task based on this kernel into the device command queue.

As OpenCL requires considerable efforts to set-up kernel execution, synchronize the data and writing the OpenCL kernel itself, a competing approach called OpenACC (Open Accelerators) was introduced. Similar to OpenMP, developers can use pragmas to annotate code that shall be executed on an accelerator as well as the required data [128]. The compiler will then create the necessary kernel code and synchronize the data.

Besides the approaches backed by industry, there are also several research projects that simplify the programming of heterogeneous systems. Cooper et al. introduced an extension of C++ to simplify code offloading to an accelerator [32]. In their work, code and variables can be marked and afterwards, the code is automatically duplicated to create multiple versions for execution on a Cell BE processor. A work with similar motivation was introduced by Gaster and Howes [53]. In their work, they show how to employ the features of the C++ language to simplify the programming of OpenCL applications. Reyes et al. introduced their approach of an OpenACC implementation based on source-to-source compiler combined with a runtime system [127]. In their evaluation, they show their results with several benchmarks and systems using CUDA and OpenCL as backends. In a following work, they also provide a detailed comparison of their work with competing implementations [128].

Besides introducing new programming models, there are also efforts to overcome limitations of some models by source-to-source or cross-compilers that transform implementations written with one programming model, e.g., CUDA, into implementations with other models, e.g., OpenCL to benefit from different accelerators. A CUDA-to-OpenCL compiler based on Clang was introduced by Martinez et al. that achieves equal execution times as manual translations [104]. Diamos et al. introduced a dynamic compiler called Ocelot that enables the execution of CUDA code on CPUs [40]. In their work, they describe their techniques to transform CUDA's intermediate PTX code into efficient code for the CPU. Another approach presented by Gummaraju et al. is Twin Peaks [59]. In their work, they describe how OpenCL kernels that exploit GPU-specific features can be efficiently executed on a CPU as well.

Grewe et al. introduced their OpenMP to OpenCL compiler that applies different code optimizations to improve the data layout for GPU execution [115]. Using machine learning on code metrics, their approach also decides automatically if the execution on the GPU

with OpenCL is superior to the execution on the CPU with OpenMP. In the evaluation, they show their results with the NAS benchmark suite and also compare with hand-written OpenCL code.

While each programming model has its strengths and weaknesses, an important question that is often left open is the resulting performance in comparison with other models. Therefore, Fang et al. present a detailed comparison of OpenCL and CUDA and a comparison of OpenCL performance on different architectures using selected benchmarks from different suites [45]. First results showed that CUDA implementations are faster in most cases. During further analysis they discovered that most differences are caused by uneven optimizations and the higher development stage of the CUDA tool chain. Regarding performance portability, they note that, with OpenCL, the programmers have to consider architectural details as well as compiler options and execution configuration in order to gain high performance and, in some cases, to even successfully execute a kernel. In their opinion, OpenCL is “very useful as a prototyping tool, enabling portability while still achieving good performance”. Thus, it only provides a common programming model for different architectures but still requires specific hardware optimizations. A similar analysis has been conducted by Komatsu et al. [86]. They analyzed the generated intermediate code by CUDA and OpenCL and discovered that their tested OpenCL compiler lacks significant optimizations like loop unrolling. After manually applying different optimizations the code achieved a similar performance as the CUDA code. Besides CUDA and OpenCL, further works also include other architectures in their evaluation like FPGAs [163]. As in today’s system, energy efficiency is an important topic, Kang et al. presented a comparative study of selected benchmarks where they also evaluated the impact on power consumption and temperature of CPUs and GPUs [71].

To quantify and compare the performance benefits of their approaches, related work either relies on code samples of different software development kits (SDKs), e.g., the Nvidia CUDA SDK, or on one of the new benchmark suites for heterogeneous systems [29, 35, 143, 108]. In addition, Feng et al. propose so-called 13 dwarfs that constitute 13 OpenCL applications with a unique and pure pattern of computation and communication [47]. In their preliminary evaluation, they also show how each application performs when executing with different OpenCL libraries and different GPUs.

To enable a closer analysis how an application and its implementations perform, heterogeneous system simulators have been introduced as well. For example, Ubal et al. introduced Multi2sim the first simulation framework that combines the simulation of a x86 CPU and an AMD Evergreen GPU in one tool [155].

2.3. Automatic optimization of hardware-specific implementations

To gain maximum performance on a system, hardware-specific optimizations are necessary in most cases. However, as such optimizations may cause degraded performance on differently configured systems, such optimizations are only applied statically if the application is only executed on a certain type of systems. In order to benefit from hardware-specific optimizations on differently configured systems, mechanisms have been introduced that perform such optimizations for specific systems dynamically. Such mechanisms are also referred to as auto-tuners. Similar to this work, auto-tuning is often an empiric approach that actively evaluates different execution parameters and chooses the best parameters for further executions [165]. However, as illustrated in Figure 2.9, auto-tuners usually optimize the performance of an implementation for specific hardware while the main purpose of this work is to optimize the choice between multiple – possibly auto-tuned – implementations

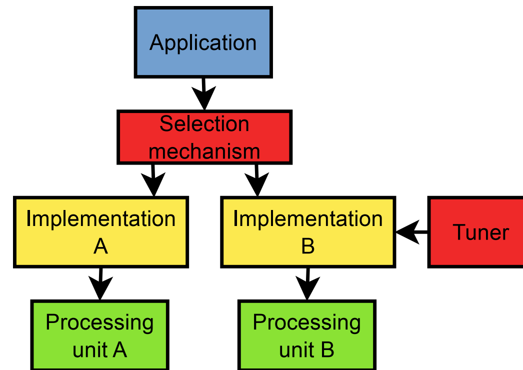


Figure 2.9.: Different positions of a selection mechanism and an auto-tuner

and processing units for a set of task. Hence, both share the same principle and adapt application execution dynamically to a specific system but each focuses on optimizations on a different level. In the following, selected examples of auto-tuning projects are presented.

A generic auto-tuning framework was presented by Karcher and Pankratius [72]. Their framework consists of a part in the kernel of the operating system and in the userspace. In the userspace, the modified application registers the to-be-tuned variables at the kernel module via a syscall and the code blocks that depend on these variables are instrumented in order to measure their time consumption. At runtime, the kernel part tries to find the best values for these variables for a single process or for multiple competing processes.

Hoffmann et al. introduced their framework called Application Heartbeats, that allows generic monitoring and adjustment of application performance [64]. Using their framework, application developers can insert function calls at appropriate places that simulate a heartbeat and later they can analyze different aspects like beat rate or latency. With this information, an internal or external mechanism can adjust the application behavior, e.g., number of enabled CPU cores, to achieve a certain goal, e.g., a minimum frame rate for video encoders. In their evaluation, they also show an example where two applications compete for CPU cores. Their framework dynamically adapts the applications to meet their individual goals, if possible, or, if not, only the priorized application while the other, a video encoder, has to reduce the image quality in order to achieve its minimal frame rate.

Ţăpuş et al. introduced their “Active Harmony” project that enables automatic performance tuning of applications [33]. Their solution operates on two levels: first, it allows to dynamically exchange libraries that provide the same functionality but use different implementations, and second, provide an API to register tunable parameters at their so-called adaption controller. To guide the selection of libraries and parameters, they also introduce a special language that enables developers and administrators to describe requirements and characteristics of the application and the system, e.g., valid ranges of the tunable parameters. This approach is similar to this work as they also propose to change whole implementations – although they don’t consider heterogeneous systems – and they also use a domain-specific language to pass additional information.

The presented projects so far have a generic design but they do not consider heterogeneous systems. Programming models for heterogeneous computing, e.g., OpenCL, have different generic and tunable parameters like the thread block size. Different projects exist that explicitly target such programming models and automatically tune their parameters [141, 103, 31, 14, 56].

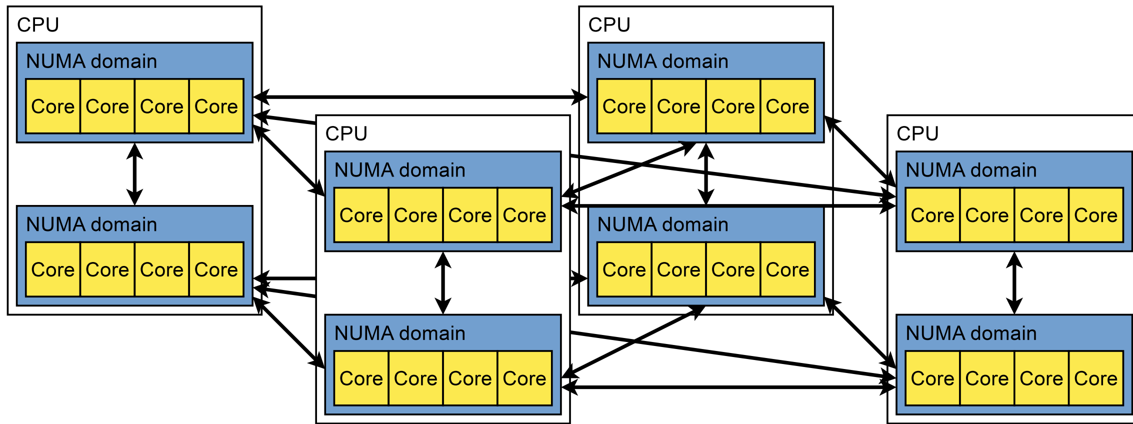


Figure 2.10.: NUMA topology of a 32 core AMD Opteron system

2.4. Data and task mapping

In a system with non-uniform memory access (NUMA), all CPUs share the same address space that spans multiple memories but for each CPU core these memories may have a different access latency. One or more CPU cores are connected to an own memory controller and form a so-called NUMA node or NUMA domain. Within such a domain, the latency of each memory is the same for all cores. As the other memories are only reachable over one or more hops on the interconnect, each required hop increases the latency for these memories. In Figure 2.10, an example of a AMD Opteron system with 32 cores is shown that has four CPU sockets and each CPU has two NUMA domains with four cores. Each NUMA domain is connected with four other domains. This results in three latency classes: the fast local memory, the memory of the neighboring domains and the slow memory of the non-neighboring domains.

Due to the potential penalty for accessing data in a remote memory, keeping threads and their data in close proximity is an important task in such systems. As such a topology can change from system to system, different approaches exist that automatically improve the thread and data mapping in homogeneous systems to minimize remote accesses [26, 150, 137].

If we look at heterogeneous systems, we have a similar problem: multiple processing units with their own memory. However, as described in Chapter 2.1, these processing units may not share the same address space, the latency over PCI express is considerably higher and a thread cannot be simply migrated to another processing unit with a different architecture in every case. Therefore, depending on the actual system, data placement not only becomes more important for good performance but it may also be critical for correct execution and a different code path may be necessary for execution on a specific processing unit.

If the original implementation that a thread would execute is not suitable for a desired processing unit, a different matching implementation has to be used. As a solution, a developer could add corresponding `if` statements to the application's source code to select a fitting implementation for the system. However, such an approach increases code complexity and is limited to known implementations and hardware at compile time.

Furthermore, not every type of calculation is suitable for migration to an accelerator and, as we will see in the following chapters, even for suitable algorithms an acceleration is only achieved under certain conditions. Evaluating and keeping track of these conditions for one or multiple accelerators can be time-consuming and error-prone while implementing these conditions considerably increase source code complexity further. This becomes significantly

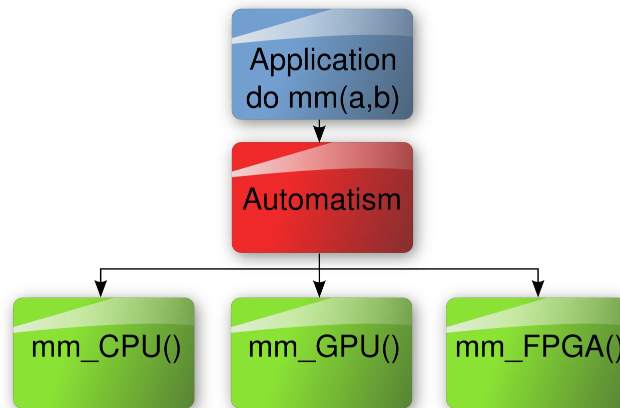


Figure 2.11.: Principle of system abstraction for applications

worse, if the application shall run efficiently on systems with different accelerators. In case an application will be distributed to a wide range of customers, evaluating any system configuration becomes infeasible.

As we will see in this work, leaving the decision to a separate mechanism not only avoids additional complexity in the source code but also provides further benefits that simplify development and improve the performance. Hence, the central goal of this work is to abstract the underlying heterogeneous system for an application in order to simplify the source code and to make the application portable across different systems. The general principle of this goal is depicted in Figure 2.11. In this example, an application requires the result of a matrix multiplication with matrix `a` and `b`. Instead of calling one of the three implementations for the CPU, GPU or FPGA, the application hands control over to an automatism that chooses the best one for execution and returns the result to the application. This way, only the automatism has to know about the available implementations and hardware, while the developer can concentrate on the general logic of the application.

An initial question – that will be discussed further in the following Chapter 4 – is how such a switch mechanism can be integrated in the application without causing too much trouble itself. Some related works require that the developer explicitly integrates and initializes such a mechanism in the application source code [13, 67, 164], some leverage the features of the programming language [32, 145, 105, 57, 62, 158, 144] or integrate their mechanism in the compiler [48, 68, 147, 85, 101, 55, 16, 68] to hide the additional complexity from the developer. In Chapter 4, this work will introduce new and previously proposed methods to integrate such a mechanism in different types of applications. Besides switching implementations and processing units on one host system, different projects were introduced that also enable a switch to remote accelerators [42, 10] and improve the different aspects like data transfers during execution [149, 55, 81, 16, 110, 116, 167]. In contrast to most of these projects, this work includes an own simple protocol as proof of concept that does not depend on specific programming models.

With a switching mechanism in place, an initial idea to enable good performance on a random heterogeneous system could be to include as many different implementations as possible in an application. But, each programming model and language usually requires an own software stack with runtime libraries or just-in-time compilers which limit the portability of applications. If the libraries are not available, the startup of the application will be aborted without the opportunity of a fallback to CPU-only execution. Regarding this topic, related work rely on programming models like OpenCL that alleviate the problem with just-in-time compilation but they also lack the expressiveness and optimization of

native programming models, e.g., provided by the hardware manufacturer [45, 86]. Instead, this work proposes a decoupled concept in Chapter 5 that reduces the strict dependencies of the application. With this concept, hardware-specific implementations are only loaded if they can be executed and if they fulfill the requirements of the application.

If a set of usable implementations has been determined, the remaining question is which of them should be chosen for execution on the available processing units. The entity that is responsible for such decisions is commonly referred to as “scheduler”. As the scheduler of current operating systems is not involved in scheduling decisions on accelerators, abstraction mechanisms as proposed in this work contain an own scheduler that decides which implementation is executed on which processing unit. However, there are also attempts to include scheduling decisions into the kernel of the operating system to guarantee fairness and isolation also on accelerators [129]. To gain a complete view of the options for this decision, a scheduler usually creates a list of so-called mappings. In the simplest case, a mapping determines which implementation is executed on which processing unit. As many applications for heterogeneous systems either contain only tasks in sequential order or repeating tasks where the actual temporal ordering is predetermined or insignificant, this work also refers to the scheduler as “mapper” sometimes to emphasize that finding the best mapping for each task is the actual challenge in a certain case.

As indicated, a mapping may contain additional parameters besides the implementation and processing unit that impact the way a task is executed in this work – like the location of the data, e.g., if the data is only mapped into the device’s address space or transferred into the device’s memory. Please note that the meaning of “to map sth.” is used twofold in this work. In the context of a scheduler it determines the way a task is executed on a system and in the context of memory management it denotes that data is made available in the address space of a processor although the data is not present in the processor’s own memory.

Depending on the considered topic, related work also evaluate specific execution parameters. For example, in distributed systems that include nodes with accelerators, if a task shall be mapped to remote accelerators and how the data transfers can be improved [149, 55, 81, 16, 110, 116, 167]. Or, if not all processing units are in use and the local system is underutilized, e.g., due to lacking task parallelism, several projects also offer automatic task splitting to map a task to multiple processing units in a system in parallel [85, 80, 133, 92, 39, 102].

The best mapping depends on the chosen optimization goal. Related projects to this work focus either on minimizing the execution time or power consumption as optimization goal. Most of the related projects focus on minimizing the execution time [67, 121, 138, 60, 105, 13, 62, 60, 21, 8]. As power efficiency becomes an increasingly important topic, there are also other approaches that select implementations and processing units with regard to their energy consumption [147, 140, 49] and apply dynamic voltage and frequency scaling [102, 98, 158, 167]. In this initial work, the primary goal of the experiments during evaluation is also the lowering of the execution time. However, due to its modular design that is described in Chapter 6, other metrics besides the execution time can be set as optimization goal as well. To account for the exchangeable optimization goal, this work commonly refers to the time or power consumption of a mapping as “costs”.

To estimate the single costs of a mapping, most related projects either use an empirical approach and measure the execution time [67, 13, 164, 24, 21, 101, 138, 21] or use code analysis combined with machine learning [58, 85]. In this work, a sufficiently good estimation of the actual costs is necessary for various purposes. For example, this work also considers changing costs during application runtime due to competing applications or faults, as described in Chapter 6.4. Hence, also an empirical approach has been chosen for this work.

If the suitable mappings and their costs have been determined, the actual work of the scheduler begins. Although graph scheduling is a topic with a long history in research, related work rely on the heterogeneous earliest finish time (HEFT) algorithm or a variation of it [154, 153, 13, 24]. HEFT is a popular algorithm as it finds a good schedule in many cases and is comparatively easy to implement once the costs of each mapping are available. Furthermore, it causes only a low overhead for the scheduling decision and, after the first tasks have been scheduled, the decisions for further tasks can be made while already executing the first tasks. However, the downside of HEFT is its greedy nature. If, for example, the best processing unit for all tasks of an application has high initialization costs, HEFT would choose a worse mapping for the first task and also stick to the corresponding processing unit if switching the unit would not be immediately profitable due to necessary data transfers. For such a case, a scheduler could always choose the same mapping for one application run and store the best mapping after all mappings have been tried. However, as the problem sizes could change, for example, the result of such an evaluation could be incorrect. Hence, this work proposes a different approach. As a mapping decision modifies the global state, e.g., the most recent data may be located in a different memory afterwards, tracking these changes for preceding tasks is necessary to get a realistic impression of the hypothetical global state for following tasks. Therefore, this work introduces a mechanism to simulate the execution of task graphs in Chapter 7 which enables evaluation of different schedulings and optimizations in advance. This is especially important for evolutionary algorithms like simulated annealing which require an extensive amount of scheduling iterations before the global optimum may be found and actually executing each scheduling would cause an unacceptable overhead.

2.5. Enhancing dependability in modern systems

As we saw in the previous sections, today's systems and applications become increasingly complex. To maximize the performance, applications usually have to use all available processing units. Consequently, application execution depends on multiple processing units that require individual implementations that in turn depend on own software stacks with runtime libraries and just-in-time compilers. Due to the increasing amount of hardware and software required for correct execution, also the probability for faults or unforeseen incompatibilities rises. To make things worse, the susceptibility of the hardware to faults is expected to increase as well: aging effects and charged particles hitting conductor paths could become a considerable threat for calculations due to shrinking feature sizes [139, 69]. For example, Haque and Pande created a test application for GPUs and tested over 50,000 systems [61]. They discovered that *"two-thirds of tested GPUs exhibit a detectable, pattern-sensitive rate of memory soft errors"*.

This work describes how selected methods for fault detection can be used in combination with the own mechanisms for efficient task mapping in order to make task execution more reliable without requiring additional efforts from application developers and without depending on special compilers and hardware. Similar to different optimizations, the methods for fault detection can each be enabled dynamically, e.g., for specific tasks, to enable a scheduler to trade performance off against dependability.

A common strategy to detect faults is redundant execution with subsequent comparison of the results. However, sequentially executing a calculation twice considerably increases the execution time. Therefore, many research projects propose to use the existing on-chip hardware redundancy to benefit from underutilized resources. Targeting general-purpose CPUs, several projects utilize the features of modern processors, e.g., multiple cores and superscalar out-of-order pipelines [54, 124, 126, 107, 157].

Vera et al. [156] also propose a fine-grained redundancy approach for CPUs but they argue that only 20% of the instructions of a modern architecture are responsible for more than 60% of the total vulnerability. They introduce so-called selective replication of only certain instructions and achieve a considerable fault coverage while introducing only minor overhead. A similar approach based on VLIW architectures is introduced by Lee et al. [93] that exploits empty slots for dynamic duplication.

As a software-based solution, Rebaudengo et al. present a source-to-source compiler creating redundancy on the source-code level [125]. Their efforts aim to detect transient faults causing data and program-flow corruption. Tahan et al. presented a dependability-focused extension of OpenMP [146]. Using their solution, developers can mark parallel regions that are critical for the application execution with an additional `reliable` keyword. Regions marked with reliable are executed using Triple Modular Redundancy (TMR) and a voter is comparing the results afterwards.

Similar to these works, one method of the framework in this work is redundant execution. However, as it operates on task level, it does not depend on special hardware or compilers. The downside of this approach is, though, that the overhead of redundant execution can only be decreased if enough processing units are idle to execute the redundant tasks.

Besides reducing the overhead of redundant execution, other approaches try to avoid redundancy at all by detecting faults by other light-weight indicators, such as symptoms like anomalous application behavior detected by segmentation faults or an unusual rate of branch mispredicts or cache misses [46, 159, 111]. Such detection mechanisms save time, but come at the price of mispredictions or lower fault coverage. Similar to these approaches, this work offers symptom-based fault detection but also considers processing units besides the CPU that provide necessary information, e.g., using hardware performance counters.

Besides symptom-based fault detection, arithmetic codes can be used for detection [161, 136]. Here, input values for calculations are modified in a way that the results can be validated using a checksum-like mechanism.

In heterogeneous systems, important tasks of the application are migrated to accelerators and only protecting the computations on the CPU is not sufficient. Therefore, other projects present their efforts to increase reliability of heterogeneous computing.

To analyze the behavior of selected GPU benchmarks in case of transient faults, Fang et al. introduce their debugger-based fault injector GPU-Qin [44]. This approach enables injection of faults at instruction level while avoiding the overhead of hardware simulation. After introducing the design of their injector, they analyze the outcome of the injection in selected GPU benchmarks. In their evaluation, they observed that the applications have a different susceptibility to corrupted results and abortions and classify them into different categories.

For redundancy-based fault detection on GPUs, Dimitrov et al. [41] introduce and evaluate three possible methods to efficiently execute kernel code multiple times: simple duplication of kernel computations, interleaved kernel instructions, and exploiting unused thread-level parallelism. A similar approach has been presented by Sabena et al. where they compare different methods for redundant execution on GPUs [132]. However, like the mechanisms described before, these efforts concentrate on a single type of accelerator while the mechanisms in this work do not depend on specific hardware.

Takizawa et al. introduce CheCUDA that enables a checkpoint and restart mechanism for applications that use CUDA GPUs [148]. In combination with a tool for CPU-bound application checkpointing, applications with CUDA kernels can be restarted after a fault or even be migrated to another host. To achieve this, CheCUDA logs the CUDA API calls

and before creating a checkpoint, transfers all data from device to host memory. After a checkpoint is restored, it moves the data back to the device before giving back control to the application.

Kawai et al. introduced DS-CUDA that enables a normal CUDA application to exploit accelerators on different nodes [74]. For critical applications, redundant execution can be activated as well to detect a fault during execution on a remote GPU.

As DS-CUDA, the redundant execution works on task level and as CheCUDA the runtime system creates checkpoints to enable a rollback in case of faults. In contrast to both, the mechanism in this work does not depend on a special programming model like CUDA and it can be enabled dynamically even only for specific tasks.

Lee et al. present their extension of an OpenACC-compatible compiler that enables automatic comparison of results from CPU and GPU calculations [94] and determines redundant or missing data transfers for GPU calculations. Similar to this approach, they try to simplify the development of applications for heterogeneous systems with automatic fault detection. In contrast to them, this work introduces generic mechanisms that are not bound to specific hardware, programming models or compilers and an additional tool which helps a developer to identify bugs in the calculations.

Boyer et al. presented a dynamic load-balancing mechanism for systems with non-uniform processing units [24]. In their work, they profile the processing units with small chunks of the total work load during application execution and thereby also detect unresponsive units which are avoided in further runs. Like in their work, the mechanisms in this work can also detect unresponsive units. However, as the runtime system stores the average execution time, also aborted executions, i.e. too short executions, can be detected.

Generic approaches not targeting a certain type of processing unit are proposed as well. Zhang et al. presented a mechanism for efficiently hiding faulty cores in a manycore processor [169]. Their solution maintains a sane view of a logical topology that does not only hide faulty cores but also improves the alignment of the cores for minimal communication costs. In contrast to their work, the methods in this work enable a fine-grained trade-off between the benefit of a processing unit and their susceptibility to faults in case the unit only suffers from transient or intermittent faults. If a unit suffers from a permanent fault, it is ignored during further decisions as well.

Another approach for increased reliability is the reduction of hardware susceptibility itself. Mitra [109] proposes hardware-level techniques for reducing the susceptibility of circuits and predicting faults induced by infant mortality or aging. Also, he introduces special test patterns for online self-tests.

To summarize, most of the presented works depend on additional work by a developer, specific programming models, compilers or hardware. In this work, the presented concepts are applied on task level and, with the already existing mechanisms for efficient task mapping, enable dynamic activation of the mechanisms for all or specific tasks with any implementation and hardware that is usable by the runtime system. Furthermore, a new metric is introduced that enables a dynamic trade-off between performance benefits and the fault susceptibility of a processing unit in contrast to keep using a faulty unit or ignoring a beneficial unit with a negligible fault rate.

However, although the framework contains methods to increase the reliability of task execution, additional techniques like the ones described in this section are required to protect the execution of the other parts of the application, the runtime system itself and the operating system on the CPU.

3. A unified approach

As we saw in Chapter 2, a multitude of challenges await developers while adding support for heterogeneous systems to their applications and optimizing the performance. Considering and handling all these challenges in the source code of the application requires considerable efforts from a developer and makes the development and the resulting application more complex.

While related works only automate specific aspects to lower the complexity, this work proposes a new framework implemented with an online-learning runtime system, called Dynamic Linking System (DLS), that provides a common basis for different mechanisms to simplify development and make applications more portable, efficient and reliable across different systems.

A basic mechanism of this runtime system is the possibility to dynamically choose the implementation that will execute a task. Instead of statically executing the same implementation chosen by the developer, as shown in the top of Figure 3.1, the runtime system is placed between the application and the implementations, as shown in the bottom. As the runtime system can be integrated in multiple types of applications, a wrapper is necessary first that translates and passes necessary information to the core of the runtime system, e.g., which task shall be executed and the available implementations. The core will make a decision which implementation shall execute a task and then initiate the execution of this implementation. In Chapter 4, the necessary parts to enable such a basic adaption will be introduced. This includes the basic architecture of the DLS core, the setup of calls to local and remote processing units and the wrappers for a non-intrusive integration of the runtime system in different types of applications, e.g., a low-level C interface, an interface to use the runtime system for rapid prototyping in Matlab and an OpenCL wrapper that enables transparent integration of the runtime system without modification of applications or knowledge of their source code.

As mentioned, in this basic design, the developer is responsible for registering the implementations included in an application. However, each implementation included in the application usually increases the list of dependencies due to individual runtime libraries and just-in-time compiler which limits the portability of the application. Hence, in Chapter 5, this work proposes a decoupled development concept that outsources hardware-specific implementations and the design of an implementation repository that maintains a list of known implementations on a system and enables the runtime system to dynamically load only executable implementations with matching requirements for the applications and

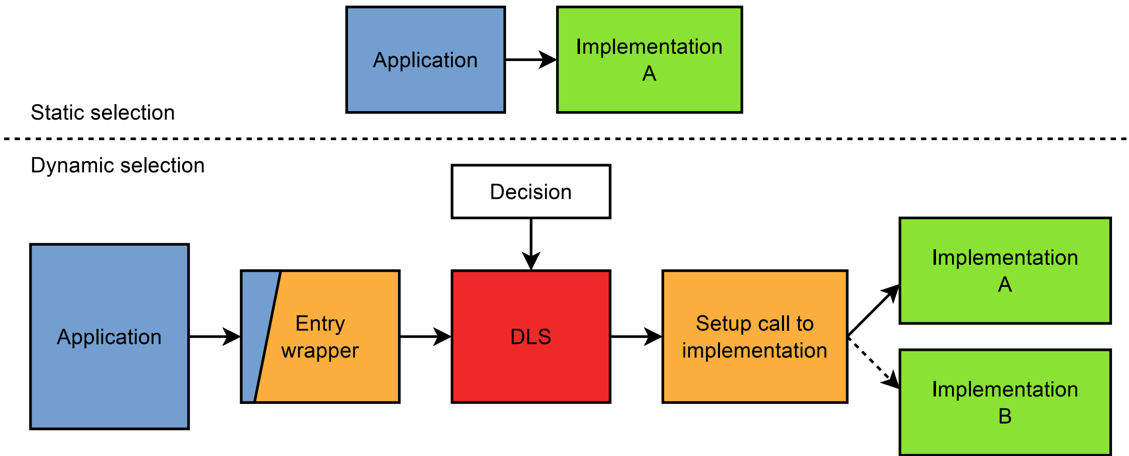


Figure 3.1.: Static and dynamic selection of implementations

satisfied dependencies. Besides improving portability, the repository also enables already compiled applications to automatically benefit from new implementations sharing the same interface, e.g., optimized BLAS libraries bundled with new accelerators.

Besides identifying suitable implementations, another requirement for the work of a scheduler are cost assessments in order to compare the possible mappings later. These costs not only consist of, for example, the actual execution time of a mapping but also of the costs for supplementary tasks that might be necessary, e.g., data transfers. As supplementary tasks like data transfers depend on the order of used processing units, the runtime system also includes a mechanism for automatic data management that is used to determine necessary transfers in advance and initiating transfers during execution. Other sources for additional costs that are considered by the runtime system are resource competition and faults during compute kernel execution. Besides the infrastructure of the runtime system for empiric identification of costs and the data management mechanism, Chapter 6 will introduce the methods of the runtime system to account for costs caused by competing cooperative and uncooperative applications and by faults that might occur during task execution. For the latter, a new metric is proposed that enables a scheduler to trade off performance benefits with the expected costs of fault recovery.

A declared goal of the framework is a wide support for different optimizations and execution variants. For example, besides task mapping and task splitting that are described in Chapter 7, this work also supports alternative code paths, e.g., solving a problem with a different algorithm, which results in mutual-exclusive branches in a task graph. During the scheduling of such a graph, the runtime system evaluates the alternative branches and chooses the better one for execution. Furthermore, as this work also considers faults that may occur during kernel execution, all or specific tasks can be duplicated and their results are automatically compared by the runtime system which also checkpoints the data to enable a rollback.

In order to keep the scheduler code simple despite the different possible execution variants, the runtime system encapsulates operations that may take a considerable amount of time into own special task types. Besides the usual type of tasks submitted by a developer – in the following called *compute task* – the runtime system knows other types of tasks like data mapping tasks or data transfer tasks. Therefore, the only things a scheduler has to know are the time a certain task will occupy a certain processing unit or memory and the other costs of the task that may be considered by the runtime system, e.g., power consumption. In order to compare the different mappings of a task, the scheduler could create and enqueue the necessary tasks for a mapping, store the total costs, remove the

tasks from the queue and start again with the next mapping. However, with this approach, an unnecessary overhead will occur as the tasks for the best mapping have to be calculated twice and it inhibits parallel evaluation of different mappings. Instead, this work proposes an approach based on so-called containers in Chapter 7 that encapsulate hypothetical changes of objects like queues, data or tasks without modifying the globally visible state of these objects. Hence, the necessary tasks for each mapping are created and enqueued in an individual container and do not interfere with the other mappings. If a mapping is chosen, the container can simply be merged into the global state.

Besides evaluating different mappings of one task, containers also enable an evaluation of complete graph schedules in advance. Instead of merging mapping containers into the global state, they are merged into an intermediate container. Hence, multiple schedules can be calculated and the best schedule is later executed in the context of the global state. This is especially useful for applications where greedy algorithms like HEFT will not find the global optimum. Therefore, Chapter 7 will also present other non-greedy scheduling algorithms like simulated annealing.

To summarize, besides covering multiple challenges of heterogeneous computing in one unified approach, this work provides the following single contributions:

- a decoupled development concept to increase portability of applications without loosing performance benefits of hardware-optimized implementations that also enables already compiled applications to benefit from new implementations,
- a dynamically selectable combination of mechanisms to detect and efficiently resolve concurrent use of resources and faults during task execution, including a new metric to balance performance benefits with fault susceptibility of implementations and processing units.
- an online simulation of different optimizations and schedulers in advance to execution for case-by-case performance evaluation which also enables demanding and non-greedy scheduling algorithms like simulated annealing
- a non-intrusive integration of the runtime system in C applications using a novel light-weight instrumentation technique and in script-based Matlab applications

For developers, the DLS framework serves multiple purposes as shown in Figure 3.2. For an application developer, e.g., an engineer or natural scientist, it simplifies development as it reduces code complexity and automatically adapts application execution for a chosen optimization goal. For a computer scientist, it provides the fundamental mechanisms to easily develop and evaluate new schedulers as well as generic optimizations of application execution in a heterogeneous system. For a hardware expert, it provides a simple method to evaluate the performance and correctness of his implementations and new hardware with different applications without the need to modify or recompile them.

In Figure 3.3, the outline of this work is also figuratively summed up.

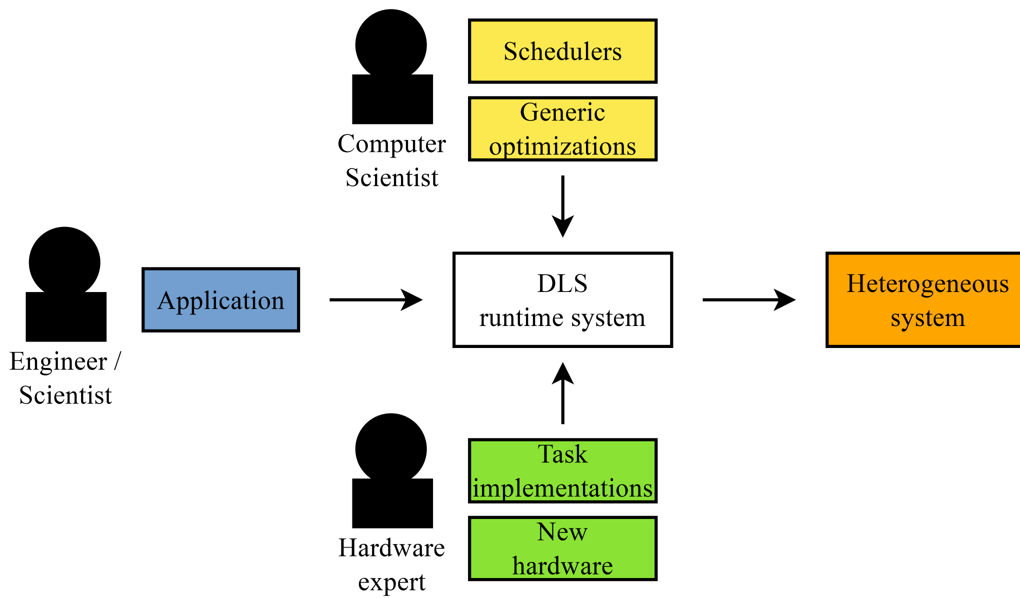


Figure 3.2.: Different types of users

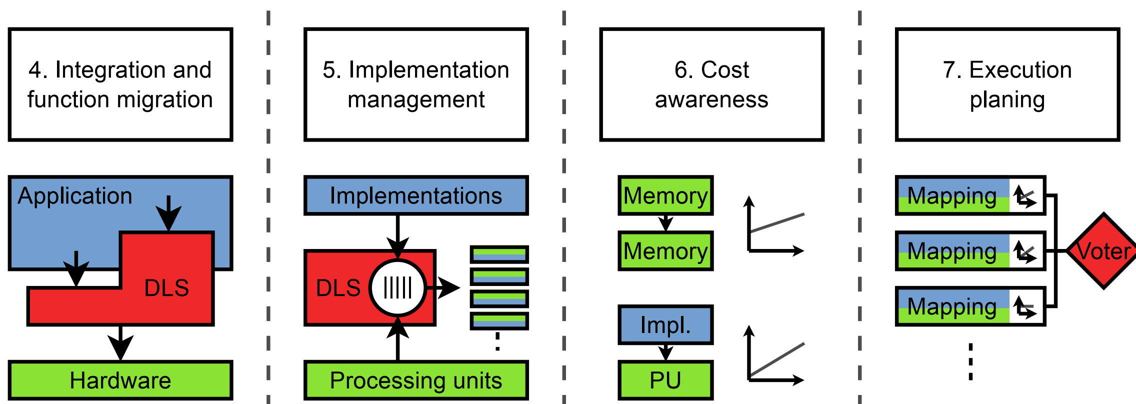


Figure 3.3.: Outline of this work

4. Light-weight integration and transparent task migration

This chapter introduces the available methods to integrate the runtime system in different types of applications and the fundamental mechanisms for dynamic selection of implementations and processing units as illustrated in Figure 4.1. In the first section, the native interfaces of the runtime system for applications written in C are introduced. Afterwards, a wrapper for Matlab applications is presented that enables improved rapid prototyping in heterogeneous systems. Due to the comprehensive API of OpenCL, it is also possible to benefit from the runtime system without the need to modify applications using a wrapper library that is introduced in Chapter 4.4. In the following two sections, parts of the runtime system's core infrastructure are described that enable dynamic activation of functionality and an extendable plugin interface to support different types of processing units. In the final chapter, a special plugin, based on the previously introduced interface, is introduced that also enables the migration of execution to remote systems and accelerators.

4.1. Introduction

In order to achieve good performance on differently configured systems, an application has to adapt to the available hardware in the current system. A simple method used to adjust

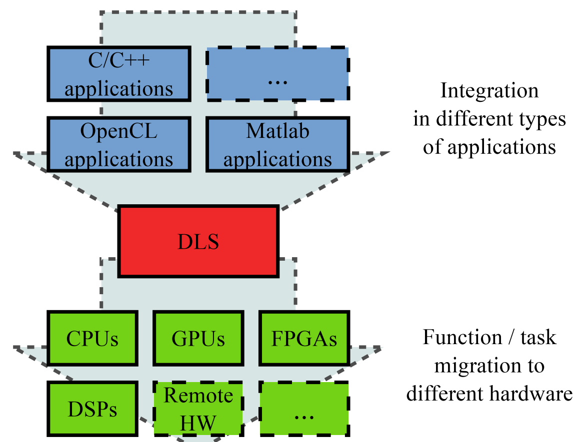


Figure 4.1.: Light-weight integration and transparent function migration

applications for specific systems are compiler flags that select a specific implementation for inclusion during compile time. While this approach causes no additional performance overhead, the resulting binary is not portable, the compilation has to be repeated for every type of supported processing unit and the selection may be only beneficial for some of the executed tasks. Similarly, including multiple implementations and choosing the best implementation or processing unit using commandline flags would enable a user to change the mapping between application runs but still, one mapping might not be the best choice for all tasks executed in the application. One approach proposed by related work is to leave the decision to the compiler. A compiler can, for example, use code analysis [58] or profiling [101] to determine the best mapping for a task. However, as the decision is fixed after compiling the application, this process has to be repeated on every system with a new set of hardware or if new hardware is added to the system. Furthermore, the application is unable to react to unpredictable events like resource competition or faults that occur during execution.

Hence, this and most of the related works favor a fully dynamic approach based on runtime systems that are able to account for the available hardware and the current system state. For such dynamic approaches, the following design decision is how they are integrated in the application. Some related projects require that the developer explicitly integrates and initializes such a mechanism in the application source code [13, 67, 164], some leverage the features of the programming language [32, 145, 105, 57, 62, 158, 144] or integrate their mechanism in the compiler [48, 68, 147, 85, 101, 55, 16, 68] to hide the additional complexity from the developer.

In the following, the basic design of the DLS runtime system for dynamic selection of implementations and processing units is introduced including several approaches for the integration in different types of applications.

4.2. Native C interface

The native C interface of the runtime system actually provides three different ways to integrate the runtime system. The first two are based on function pointers that can be dynamically set to point to specific implementations. While the first approach requires small changes in the source code, the second exploits existing structures in the ELF binary format to adapt the execution of unmodified applications. The third approach uses the regular API of the runtime system which gives precise control over the runtime system but also requires additional source code modifications.

As we will see later in Chapter 4.3 and 4.4, providing an interface based on C also enables an integration of the runtime system in other languages as many languages offer a C interface themselves for custom extensions.

4.2.1. Pointer-based function migration

The initial version of this runtime system was introduced in a diploma thesis [75] that proposed two mechanisms for light-weight switching between different functions. The basic principle of these mechanisms is the usage of function pointers that enable indirection of function calls and introduce no perceptible overhead for switching to and calling a function. To determine the destination of a pointer, an external control entity was proposed that can issue a switch through a custom kernel module. In Figure 4.2, an overview of this design derived from Figure 2.11 is shown.

Following the first pointer-based approach, the function pointers are defined in the source code. This method has the advantage that the function pointer can be used as if the

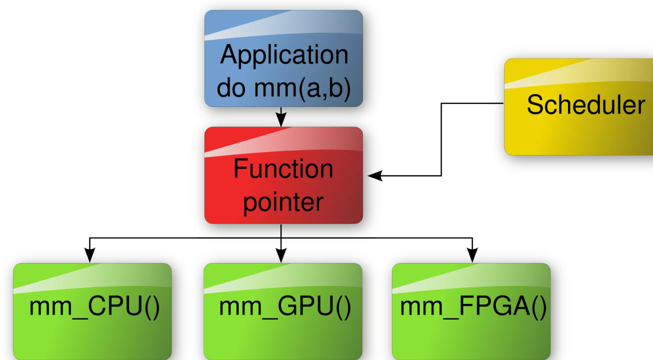


Figure 4.2.: Overview with external scheduler

developer would call one of the real functions, thus causing neither additional complexity in the code nor time overhead. In the following, an example for a matrix multiplication with a CPU and GPU implementation is shown:

```
void (*matmul)(matrix A, matrix B, matrix C);

void matmul_CPU(matrix A, matrix B, matrix C) {
        /* ... */
}
void matmul_GPU(matrix A, matrix B, matrix C) {
        /* ... */
}

void calculate() {
        /* ... */
        matmul(A, B, C);
}
```

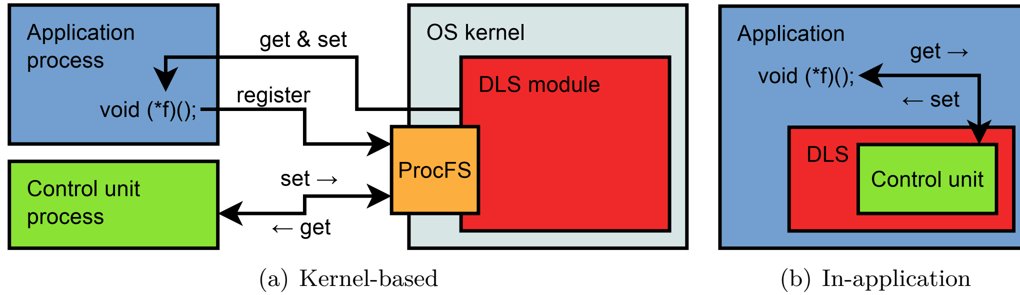
Depending on the address in the pointer `matmul` either the CPU or the GPU version is executed.

To change the destination of a pointer, the external control entity uses a custom kernel module to initiate a switch. However, the kernel does not know about available pointers in each process. Therefore, the kernel module also provides a userspace interface through the procfs filesystem that can be used by the application to register the function pointers. The registration routine inside the application is bundled with the function pointer definition and additional management information inside a macro. With this macro, the first line of the previous listing is replaced by this statement:

```
DLS_DEFINE(matmul, void, matrix A, matrix B, matrix C);
```

This macro defines a function pointer called `matmul` with no return value and three matrices as parameter. Hence, only this single line and the inclusion of the DLS header file is required in the source code to integrate the runtime system.

Another part of the thesis was a similar approach that requires no source code changes. It is also based on function pointers but the pointers are not defined in the source code. Instead, the implicit function pointers of ELF binaries for dynamic symbol resolution are diverted from their intended use. Symbols are used to reference objects like global variables or functions in memory that change their location between application executions, e.g.,



(a) Kernel-based (b) In-application

Figure 4.3.: Different types of control infrastructures

libraries are loaded to different addresses for security purposes. For such symbols, the ELF structures contain allocated space for a string and for an address in memory. The string equals the name of the object, e.g., `matmul_CPU`, and the address field contains the address of the object in memory. If the object is a function that is not defined in the same binary but in a separate library, the address of the object is unknown and the address field contains the address of a lookup function. If the function is called in the original code, the lookup function will be executed first and it will update the address field with the correct address that is later on called directly. Hence, if the application should call `matmul_GPU` instead of `matmul_CPU`, the kernel module modifies the symbol name and resets the address field to the lookup function. Consequently, if the function is called the next time, the lookup function will resolve the address of `matmul_GPU` and write its address into the address field.

This way a function call can be redirected to another function during runtime without source or binary code modification and therefore also works with proprietary applications, for example. However, this approach also has considerable limitations. For example, to automatically determine necessary data transfers, the runtime system needs additional information like the location of the input data and the data each implementation will access. This information is difficult to obtain without changing the source code. Hence, as source code modifications are necessary to benefit from all features of the runtime system, the approach based on ELF function pointers is not used in the rest of this work.

As mentioned, the initial design of this approach relied on a module in the kernel of the operating system to select the target of a function pointer. This module provides an interface for applications to register the address of their pointers and an interface for a control unit in a separate process as shown in Figure 4.3(a). As target for the pointer, the control unit cannot only choose between present task implementations in the application but also instruct the application to load new libraries with further task implementations.

However, due to the indirection through the kernel, only coarse-grained decisions can be made as the control unit only sets the default target and has no information if and how many threads are executing a task implementation. With additional measures the application could signal the control unit that it is about to call a specific function pointer and the control unit could set a target per call but, besides other disadvantages, this would add an additional time overhead.

Instead, the mechanisms inside the application itself were extended to make the application self-directed and independent from special kernel modules and other processes as shown in Figure 4.3(b). The challenge of such an approach is to hide this control mechanism as much as possible from the developer in order to keep the source code simple.

To hide the functions of the control mechanism, this approach uses instrumentation – that is based on previously presented work [78] – to enable the execution of additional functions between the call through a pointer and the actual target function. Most calling conventions,

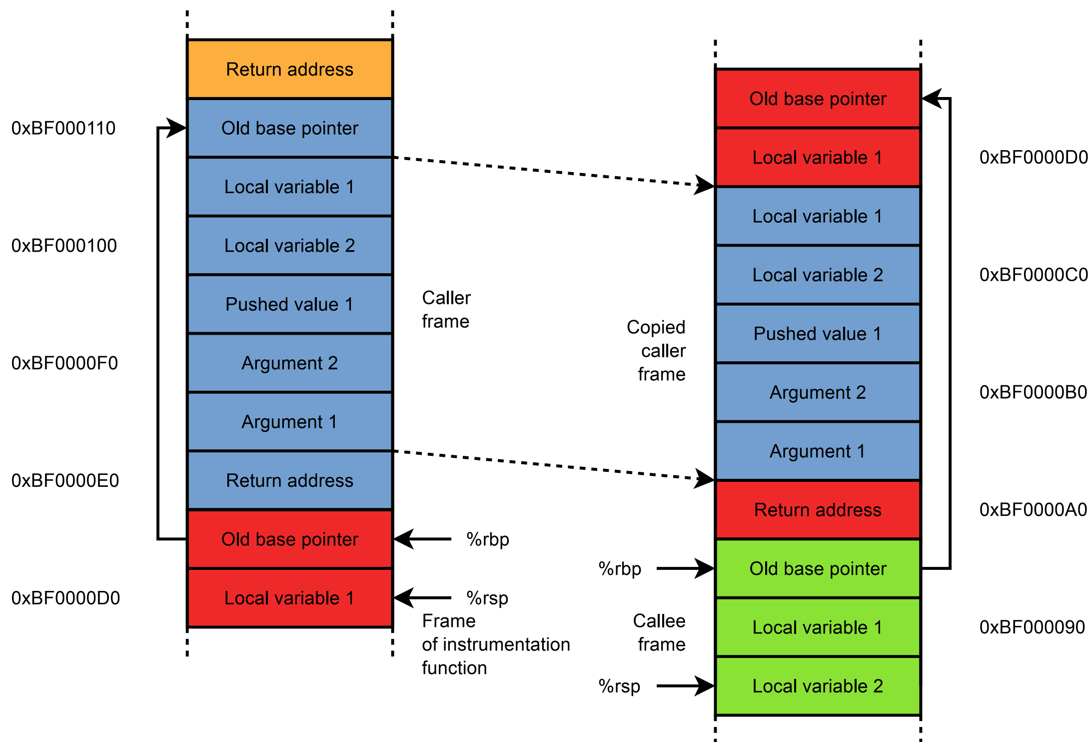


Figure 4.4.: Duplication of caller stack frame for function call instrumentation

e.g., on x86-64 architectures [106], specify that function arguments are passed in registers and on the stack – depending on the number, size and type of the arguments – and the return value is passed back in registers. As registers and the stack are modified by the additional functions, these values have to be preserved before executing additional code and reinstantiated before calling the actual target function.

Hence, instead of referencing an implementation directly, the function pointer contains the address of a special function written in assembler. At the beginning, this function preserves to contents of the registers containing the arguments. First the stack is increased to allocate space for the registers. Afterwards, the data in the registers is stored in this area. Then, the code to retrieve the arguments on the stack and to jump inside the runtime system is executed. After the runtime system executed the target function and before returning to the application the allocated stack space is freed by restoring the original stack address.

A detailed visualization of the stack operations is shown in Figure 4.4. The first task is to calculate the boundaries of the original stack frame. The so-called base pointer is a register that points to a location in the current stack frame where the address of the previous stack frame is stored. This results in a linked list of stack frames that can be used to determine the frame of the caller function.

After preserving the arguments in the registers and on the stack, arbitrary functions can be executed by the core of the runtime system. Before calling the main function of an implementation, the runtime system copies the original stack frame on top of the stack and the registers are filled with their original values. Hence, the implementation executes as if it was called directly. After its execution, the return value is also stored in a temporary location and further internal functions of the runtime system can be executed, e.g., to measure and store the actual time consumption. Finally, the return value is reinstantiated and control is returned to the application.

The downside of this approach is a dependency on the instruction set architecture and the

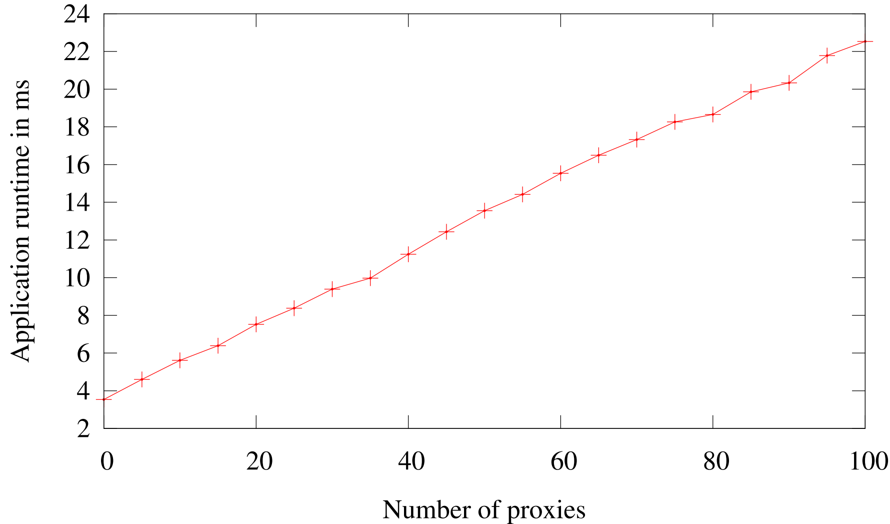


Figure 4.5.: Application runtime as function of proxies

calling convention. Hence, for other architectures or calling conventions, similar assembler code is necessary. Another problem are modern compilers with optimization routines that use the base pointer as general-purpose register to increase performance. In this case, the stack unwinding technique for exception handling can be used to retrieve the stack frame addresses without a linked list of frame pointers, e.g., using the libunwind library [1].

To evaluate the overhead caused by this approach, a simple application has been written that consists of a `for` loop in the main function and another function that is called from within the `for` loop. The second function is only executing a simple integer addition. Thus, for few loop iterations, most time is consumed by startup and initialization of the application while for a high number of loop iterations, the application runtime is dominated by the time required for repeatedly calling the second function.

At first, the application runtime is measured for a variable number of loop iterations with and without the runtime system. With a single iteration, the original application finishes after 2305 μs while the application with the runtime system lasts 3624 μs. This results in a one-time startup overhead of around 1.3 milliseconds for loading the runtime library and initializing one function pointer or a so-called proxy. A proxy represents a function pointer with a specific function signature and functionality. For multiple proxies, the initialization costs increase linearly by roughly 0.2 milliseconds per proxy as shown in Figure 4.5.

In Figure 4.6, the per-call overhead of the instrumentation-based and the explicit invocation approach, that will be introduced in the next section, is compared to normal execution is shown as a function of total function calls during one application run. This overhead constitutes the required time for the basic infrastructure of a call interjection. As it can be seen, with a growing number of iterations the static startup overhead becomes insignificant and the overhead for a single call settles below 1800 ns for both approaches.

4.2.2. Explicit integration using the DLS programming interface

In order to use the proposed automatism without dependency on specific ISAs, calling conventions or compiler optimizations, another approach was developed that relies on additional source code modifications. The main problem for a transparent integration of the runtime system are the function parameters as those cannot be preserved by a

[1] The libunwind project `http://www.nongnu.org/libunwind/`

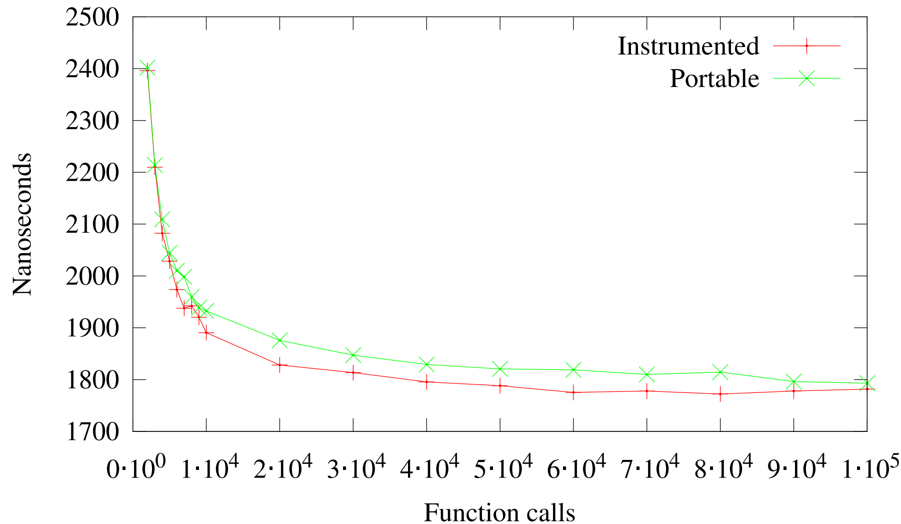


Figure 4.6.: Overhead introduced by the instrumentation-based and the explicit approach

generic function and passed to the actual target function in a portable way. Therefore, the approach presented in this section uses an explicit method to register arguments and query them during execution of a target function.

In order to preserve the arguments, this approach uses a storage that is organized as a stack of arguments. To add arguments, a push function is called from the application with arguments passed by reference. In turn, the actual target function calls a pull function to receive the arguments in reverse order. For convenience, multiple arguments can be passed with both functions and the order is reversed automatically for the pull function. Instead of calling a function pointer as in the previous section, the developer has to call a special function of the runtime system to initiate a call. In the following, the corresponding code for the matrix multiplication example is listed:

```
void matmul_CPU() {
        dls_pop_arguments(&A, &B, &C);
        /* calculate */
}
int main() {
        /* ... */
        dls_push_arguments(&A, &B, &C);
        dls_execute("matmul");
}
```

In the main function, the matrices A, B, and C are pushed onto the argument stack and afterwards the runtime system is instructed to start execution of a task `matmul`. If `matmul_CPU` is called it first pops the three matrices from the argument stack and then resumes its normal operation.

For all approaches to integrate the runtime system in this chapter, the runtime system provides two approaches to gain a list of possible target functions. A developer can add a function for a specific task with an API call or the runtime system will look for usable implementations itself as described later in Chapter 5.3.2. With the API call, one can add a function that is included in the application binary or from a dedicated library. For a function from a library, the name of the library is required and it must be in a path that is known to the dynamic linker. As an example, to add the function `matmul_GPU`

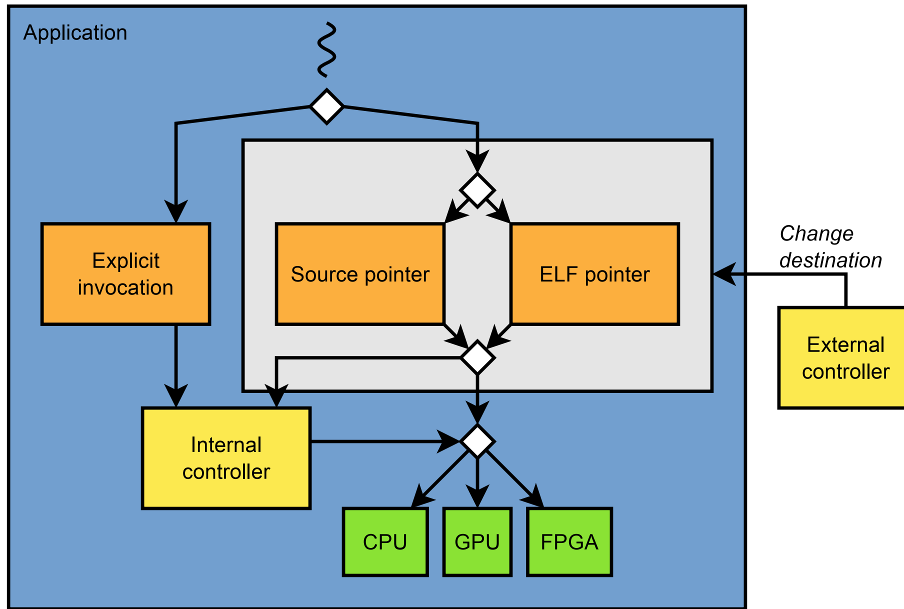


Figure 4.7.: Overview of the possible paths from a call to an implementation

from `libblas_gpu.so` to the list of implementations for the task `matmul`, one can use the following command:

```
dls_add_fct(matmul, "libblas_gpu.so", matmul_GPU);
```

4.2.3. Discussion

For applications written in C, several approaches for transparent function migration with a focus on light-weight and non-intrusive design have been presented. To summarize, an overview of possible paths from a call to an implementation is shown in Figure 4.7. The developer has either the choice to invoke the internal controller explicitly or to use one of the pointer-based approaches. Both pointer-based approaches can point either to the entry function of the internal controller or directly to one of the implementations. Besides, their target could also be modified by an external controller through the kernel module. The internal controller does not depend on a pointer mechanism and calls one of the implementations directly.

The benefit of the pointer-based approaches with external controller, is that decisions can be made by a central instance like the kernel scheduler. The downside of this approach is that the kernel has to read and modify data in the userspace which might pose a security threat. Also, the resource assignment through the kernel is not mandatory as in current systems, the right to access an accelerator is granted by user permissions. Thus, an uncooperative application might use an accelerator although it is not allowed by the kernel. As this approach requires a custom kernel module, additional efforts by the administrator are required before this approach can be used.

Instead, the in-application controller approach enables the bundling of the methods introduced in this work inside a library that is easier to maintain and does not require special preparations of the operating system. Furthermore, it avoids expensive crossings of the border between user and kernel space. While the instrumentation-based method requires less source code modifications, it also depends on an ISA that permits such instrumentation.

As we saw in Figures 4.5 and 4.6, the runtime overhead introduced by the in-application approaches is negligible compared to the usual runtimes of compute kernels and applications

on heterogeneous systems that can easily exceed several milliseconds per kernel and several hours per application run. Especially with the mechanisms introduced in the following chapters, this overhead is more than compensated by automatically choosing the best accelerator.

Although the binary-based approach offers interesting opportunities its applicability is also limited. For the mechanisms in the following chapters, additional information about the application is required to improve the performance. Currently, this information cannot be determined automatically and efficiently, e.g., the required data for an implementation, and thus additional help by the developer is required in terms of changes in source code anyway. Therefore, this approach is not pursued further.

Due to these circumstances, the methods in the following chapters are based on the in-application approach that can be integrated using either the instrumentation-based or the explicit variant.

4.3. Integration in Matlab for rapid prototyping

Matlab is a numerical computing environment with an integrated script language that is developed by MathWorks, Inc. [2]. It contains a multitude of functions and tools to ease the solution of mathematical problems and the development of algorithms. Hence, Matlab is often used for rapid prototyping of mathematical algorithms. If a working algorithm is implemented using the Matlab script language, it is also ported to C/C++ in order to benefit from the native code generated by highly optimized compilers and implementations for accelerators are written and evaluated as well. As usually only the compute-intensive algorithms are rewritten using separate languages and the main application logic should remain in Matlab, the so-called MEX interface was introduced that enables an extension of the internal script language with functions written in other languages like C/C++. This results in a situation similar to regular applications for heterogeneous systems where multiple task implementations and processing units could execute a task and the developer is responsible for finding the best in the current situation. In order to support rapid prototyping and to leverage the mechanisms for automatic task mapping presented in this work, a wrapper for the Matlab script language has been developed using the previously introduced C programming interface of the runtime system and the MEX interface.

If an unknown function is called in a script, Matlab looks for a library with the same name as this function. If such a library is found, it is loaded and a function with the following signature is expected inside the library that will be called with the function parameters and expected return values of the function called in the script:

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]);
```

The parameter `nlhs` determines the number of return values of the functions while `plhs` contains the list of pointers for the values. The actual function arguments are listed in the `prhs` array and their number is stored in `nrhs`.

Hence, as only one callable function per library is possible, the wrapper provides one demultiplexing `dls()` function that receives the requested command as string in the first argument and then passes control and the further arguments over to the corresponding command-specific handler that transforms the arguments and calls the actual function of the DLS runtime system. However, for convenience, it would also be possible to create an own library for each function of the runtime system.

[2]http://www.mathworks.com

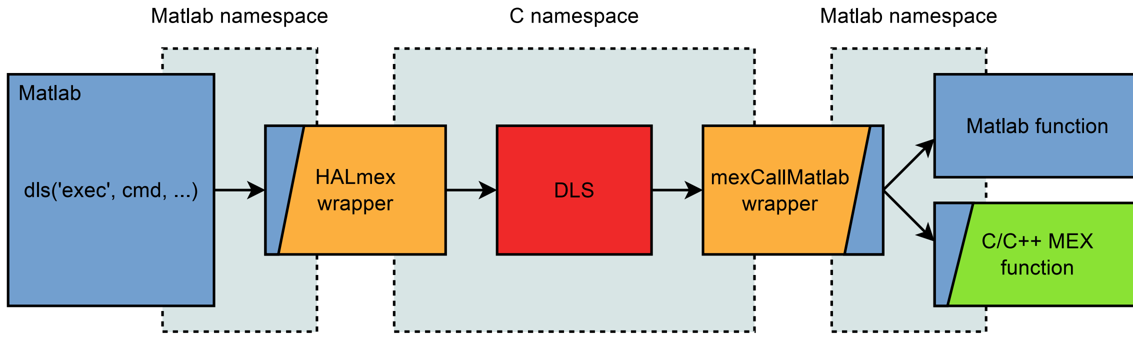


Figure 4.8.: The path from submitting a task to the possible task implementations

With the wrapper, a task type is implicitly created by adding an implementation. For example, to create a task type `matmul` and to add an implementation `matmul_GPU` that may execute this task, the following command can be used:

```
dls('add', 'matmul', 'matmul_GPU')
```

To issue execution, `exec` is passed as first argument followed by the task type and the corresponding arguments:

```
dls('exec', 'matmul', a, b, c)
```

Similar commands are available to, e.g., submit multiple tasks of a task graph or to register and request certain data that may reside in an accelerator's memory, for example.

An overview of the execution process is visualized in Figure 4.8. First, the variadic function `dls()` is called from within Matlab with the fixed arguments `exec` and `cmd` and an arbitrary list of further arguments. The wrapper pushes the optional parameters on the argument stack of the runtime system and afterwards calls the proxy with the name of the second parameter `cmd` that in turn passes control to the actual DLS runtime system. The runtime system chooses a target function and then calls again a special function from the wrapper that passes the call back into Matlab using the `mexCallMatlab` function. The function called using `mexCallMatlab` can be an internal Matlab function or another C/C++ function that implements the MEX interface.

4.4. Transparent OpenCL wrapper

Another method to employ this work is the OpenCL wrapper library that enables unmodified applications to benefit from this work. As we can see in the top of Figure 4.9, an OpenCL application can use different vendor libraries that implement the OpenCL API. As OpenCL requires the developer to pass all necessary information for kernel execution via the OpenCL API, the calls of the OpenCL API can be translated into corresponding calls of the DLS API by a wrapper library as indicated in the bottom of Figure 4.9. In turn, the runtime system is then able to choose the best of the vendor libraries for execution that is called through a hardware plugin as described later in Chapter 4.6.

Such wrapper libraries have been proposed before [57, 145, 63]. In contrast to them, this chapter and Chapter 7.4 will provide insight into the kind of problems that have to be solved in order to compensate the overhead introduced by an additional abstraction layer.

An initial problem for this approach was that OpenCL requires several API calls to submit all necessary information for a task, e.g., the source code of the compute kernel or required data, and the outcome of each of these calls depends on the chosen device, e.g., the resulting

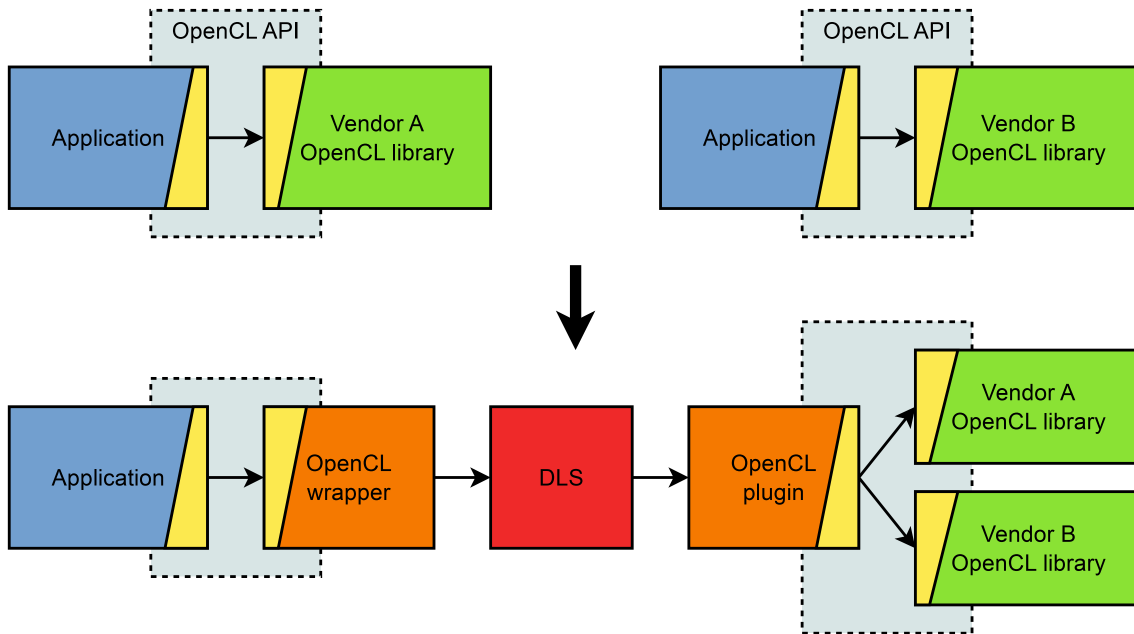


Figure 4.9.: Rerouting of OpenCL API calls through the DLS runtime system

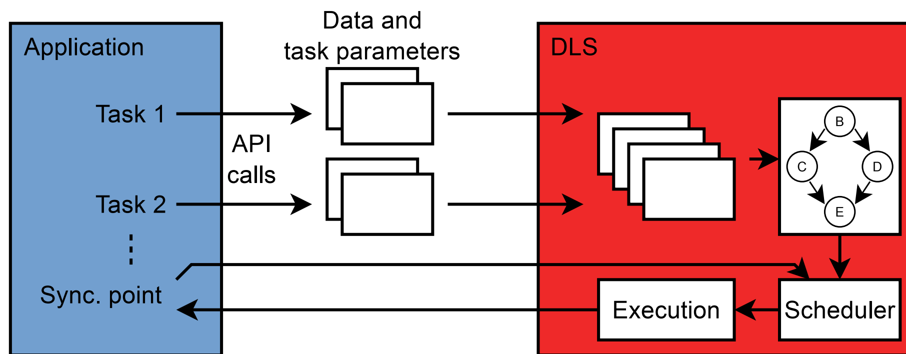


Figure 4.10.: Accumulating tasks until a synchronization point

machine code of the compute kernel or the location in device-specific memory. But, during the initial calls, the runtime system is not yet able to determine the most beneficial device as the following calls have a considerable impact on the decision. Therefore, the runtime system presents a virtual device to the application and logs the submitted data for each API call. Then, after all necessary information is available, it can choose the best device for a task and submit the logged commands to the device's OpenCL library.

As we will see in Chapter 7, the more information about the tasks of an application the runtime system receives, the better are the chances to find the optimal scheduling for an application. Therefore, the runtime system tries not to start the scheduling after one completely received task but delays execution until specific synchronization points as illustrated in Figure 4.10. Such points are, for example, read requests from the application or blocking commands in general. A downside of delayed execution is that implementations cannot tune themselves to specific devices. For example, with OpenCL, an implementation can query the maximum number of threads or size of shared memory in a block and adapt its execution accordingly. In such a case, one workaround is to determine and return the minimum of all available devices.

Due to the delayed execution and the additional work required for planning the execution, the potential optimization benefits of the runtime system come at a cost. To get an

Benchmark	Subm. tasks	Exec. tasks	Avg. exec. time	Exec. task graphs
b+tree	5	39	1462979	5
backprop	5	22	1983662	3
bfs	37	77	977409	13
cfd	16005	16031	218670	536
gaussian	2049	2059	732183	2049
heartwall	24	112	22520879	24
hotspot	2	10	3432565	2
kmeans	75	157	4829890	37
lavaMD	2	12	23876196	2
leukocyte	24	155	3079641	24
lud	47	51	669377	2
nn	2	8	3742468	1
nw	256	264	347964	11
pathfinder	7	18	6904782	2
srad	903	1129	277593	202
streamcluster	9667	16120	832097	8056

Table 4.1.: Statistics of Rodinia benchmark applications

impression of these costs, applications of the Rodinia benchmark suite were executed one time with the Nvidia OpenCL library directly and one time through the wrapper library. During execution with the wrapper, the execution has been limited to the Nvidia GPU in order to make a fair comparison without profiting from a better suited device. In Figure 4.11, the results of the measurements are shown that are also discussed further in Chapter 7.4. As we can see, the execution time with the wrapper is similar to directly using the Nvidia library in most cases. For a few applications, notably `cfd`, `srad` and `streamcluster`, there is a considerable increase of execution time. As we limited the wrapper to execute the application in the same way as in the direct case, these overheads are most likely caused by scheduling. To analyze this behavior, statistics about the scheduling of each benchmark have been collected during execution and listed in Table 4.1. The statistics include the number of submitted compute tasks by the application, the number of resulting tasks including supplementary tasks executed by the runtime system, the average execution time of each task and the number of resulting task graphs that had to be scheduled by the runtime system. As expected, the `cfd`, `srad` and `streamcluster` benchmarks have outstanding numbers of tasks and task graphs. Only the `gaussian` benchmark has similar high numbers but they do not result in a similar high overhead of application runtime. If the number of submitted and executed tasks are close, e.g., in case of `cfd` and `gaussian`, this indicates that the GPU was mostly busy with executing tasks and only a few supplementary tasks for memory allocation and data transfers between host and GPU RAM were necessary. In contrast, the number of executed tasks is 1.6 times higher than the number of submitted tasks for the `streamcluster` benchmark. This indicates that data was periodically synchronized and many transfers between CPU and GPU were necessary. By comparing the number of submitted tasks and the number of executed task graphs, we can see that the three benchmarks with a considerable overhead have a low average task execution time and a high number of tasks and task graphs. Consequently, dynamically determining the best way to execute each task becomes a considerable performance overhead in such cases. In Chapter 7, different methods to lower this overhead will introduced.

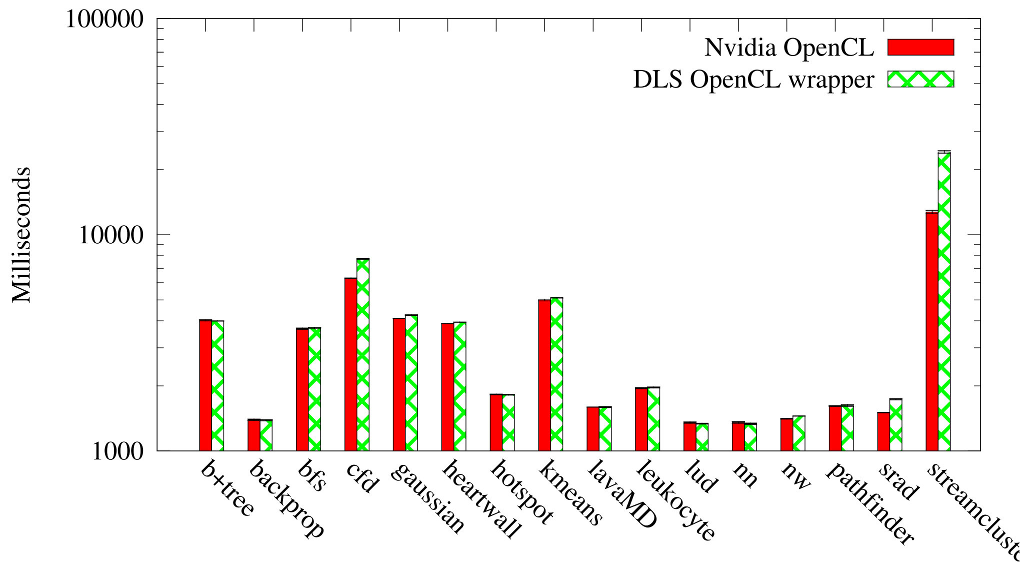


Figure 4.11.: Runtime comparison of benchmarks with and without the OpenCL wrapper

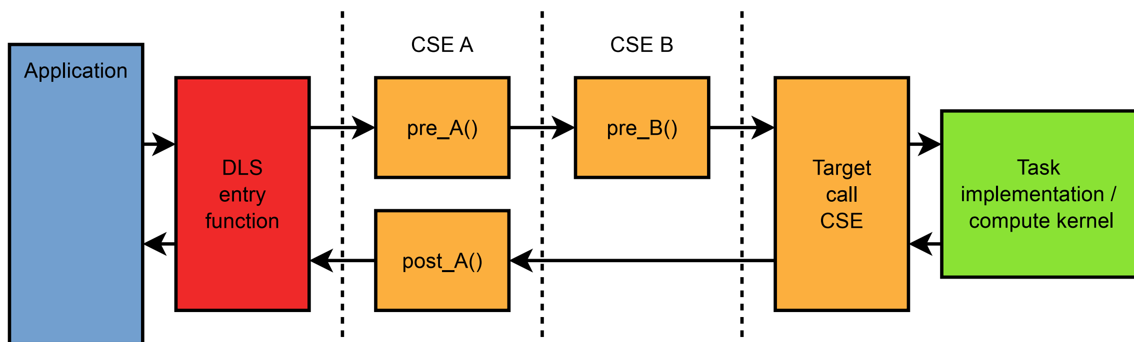


Figure 4.12.: Example of a call stack for a single task

4.5. Call stack infrastructure

Before and after the execution of a task implementation, several actions have to be performed as we will see in the following chapters, e.g., to measure the execution time. In order to make the list of actions extensible for other uses of the framework, a so-called *call stack* has been implemented that enables a developer or the runtime system itself to dynamically register specific actions that have to be executed before or after the actual execution of a target function. In Figure 4.12, an example of this design for executing a single task is shown. First, control is transferred from the application to the entry function of the runtime system using one of the previously introduced mechanisms. Afterwards, the entry function starts executing every so-called *call stack entity* (CSE) that may execute code before or after the target function respectively. In this example, CSE A execute code before and after the target function and CSE B only code before the target function. A special CSE provided by the runtime system is the so-called target call CSE that is usually the last CSE in the call stack and is responsible for actually calling a task implementation.

CSEs can also be used to narrow down the search for the best implementation and processing unit. In the entry function, the runtime system sets up a special structure named *call state* that, e.g., contains lists of available implementations and processing units for the current task. Each of the following CSEs can modify this lists to, e.g., exclude specific entries or

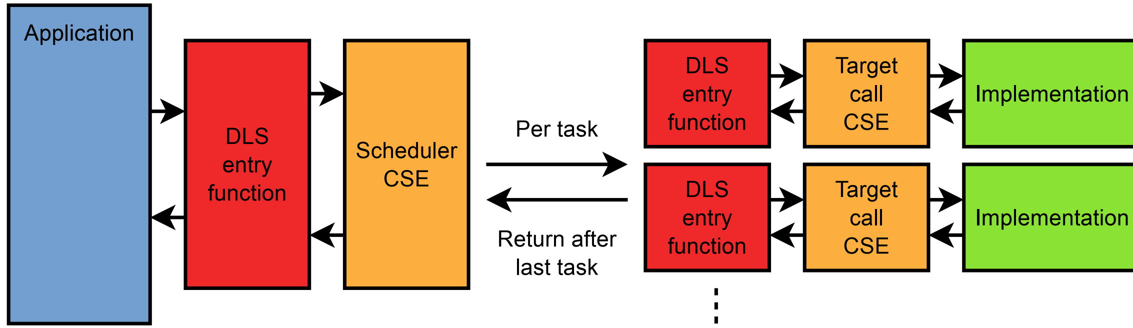


Figure 4.13.: Example of a call stack for multiple tasks

set the current favorite combination of implementation and processing unit that will be executed by the target call CSE.

If the application submits a task graph, the default procedure in the runtime system differs as shown in Figure 4.13. Again, control is first passed from the application to the entry function and it starts processing the call stack. While the last CSE in the single task case is the target call CSE for executing the actual implementation, the call stack ends here with the scheduler CSE. This scheduler CSE, that is described in Chapter 7, will determine a schedule for the task graph and start executing the graph. For each task, the entry function is called again and further CSEs, e.g., for profiling the specific task, are executed. Similar to the single task case, the last CSE is again the target call CSE that then calls the actual implementation.

4.6. Extensible hardware interface

To enable function migration, the runtime system must be aware of the available hardware. However, another design goal of the runtime system is independence from specific programming models and hardware. As specific methods are still necessary to employ certain hardware, e.g., to transfer data to and from device memory, the runtime system exposes a plugin API that – among other purposes – abstracts the individual communication with processing units for the core of the runtime system. With this API, the runtime system queries the amount and type of available devices or initiates transfers, for example. In Figure 4.14, examples for different hardware plugins are shown. As we can see, even the CPU and the host RAM are managed through a plugin as there are different ways to query and control them, e.g., using the standard C library or special libraries like hwloc [3] that provide more detailed information about the hardware but may not be available on any system. One problem of such a flexible approach is that a processing unit might be usable with different plugins. For example, an Nvidia GPU can be used with CUDA but also with OpenCL and each plugin would report one available GPU. In some cases, it is not possible for the runtime system to detect if two reported devices are actually the same. Therefore, additional help of a system administrator is necessary to avoid that the runtime system will create two waiting queues for the same device.

Similar to the method in Chapter 5, these plugins are only loaded if the necessary runtime libraries are installed. Hence, this approach also frees the runtime system from specific dependencies, e.g., runtime libraries, that may not be present on systems without the corresponding accelerator and could cause an abort of the application.

A considerable problem for hardware detection are high initialization costs. With some programming models, the querying and initializing the hardware can take up to several

[3] `http://www.open-mpi.de/projects/hwloc/`

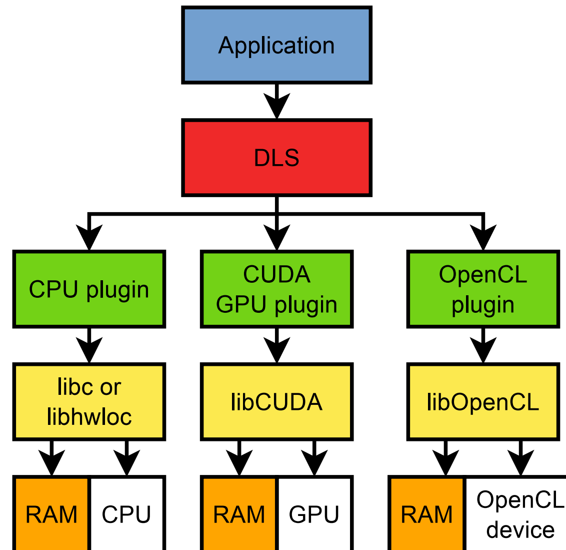


Figure 4.14.: Plugin interface for different types of hardware

hundred milliseconds. Therefore, during startup of the runtime system, a plugin only registers the type of processing unit it is responsible for. Only if there is later an actual task implementation under consideration for execution that uses this type of hardware, the plugin initializes the corresponding library for the hardware and looks for available devices. Hence, initialization costs are avoided for devices that will not be used.

4.7. Ad-hoc work offloading in local networks

In a common corporation or university network, most of the computer systems are under-utilized as the performance of those systems grows while the computational demand mostly stays the same for common tasks like office applications. In addition, even the low-end graphics processors in office desktops are only used for displaying graphical user interfaces although they are capable of massive-parallel general purpose computations. However, due to the latency of network communications, efficiently utilizing remote processing units is difficult. To maximize performance in such clusters, applications usually contain special implementations, e.g., written using MPI.

From the point of view of a scheduler, there is not much difference between local or remote accelerators as data has to be transferred to and from the remote system before and after the execution as well. Therefore, the scheduler of the runtime system – that is introduced later in Chapter 7 – can determine the benefit of remote processing units similar to the benefit of local units. However, to compensate the latency with such an approach, the remote units usually have to be considerably faster than local units or there has to be unexploited data or task parallelism in the local system. Furthermore, if multiple applications employ resources in the network, a remote unit might not be exclusively available in the moment it would be beneficial for the local application. Hence, in contrast to explicitly programming applications for a distributed system, this work automatically offloads work on demand depending on the current and remote system state and provides a simple network protocol based on TCP/IP as proof of concept.

Similar projects to integrate remote accelerators have been introduced before [10, 42]. However, these projects are limited to specific programming models while this work offers a generic approach to offload tasks. In contrast to established methods for remote procedure calls, this proof of concept expects that the remote systems use similar processor

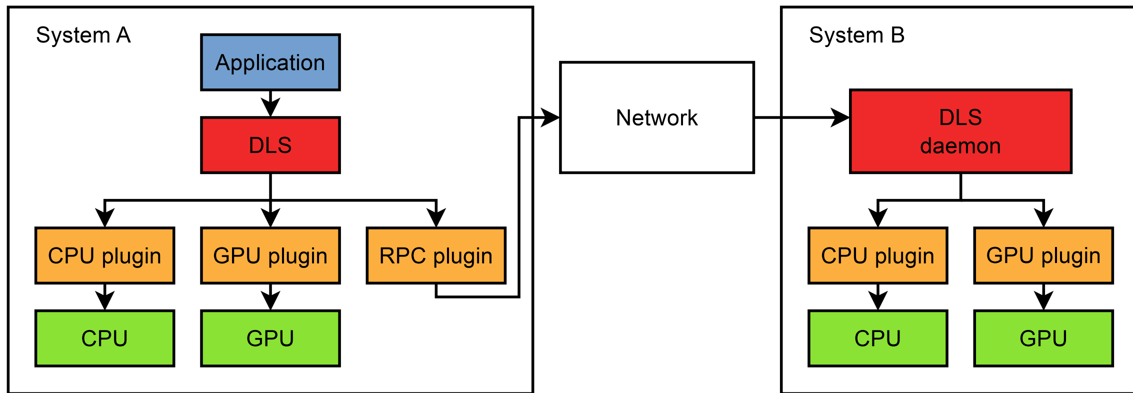


Figure 4.15.: Integration of remote processing units using a hardware plugin

architectures and calling conventions as the data is transferred without modifications. Furthermore, a task implementation must not access data that has not been registered before as described in Chapter 6.3. For example, accessing global variables or data in the local filesystem is not supported.

Remote processing units can be integrated into the runtime system similar to local units using the plugin infrastructure as shown in Figure 4.15. On the remote host, a special DLS process has to be started. This daemon initializes the local runtime system and listens on new connections. The main purpose of this daemon is to parse the incoming commands and pass them to the local runtime system, e.g., incoming data transfers or task submissions. For each connection the daemon starts an own thread that handles incoming data and starts new worker threads to process certain commands in parallel. By using threads, it is also possible for different clients to use the same namespace and to share data between these clients. However, if the daemon is used by different parties, a new process could be forked for every client, e.g., to enhance security. As, in the latter case, every connection would start a new separate process, each client process maintains only one connection to the host. As a client process may contain multiple threads and each thread may submit requests concurrently, e.g., to transfer data while executing a task, the connection is multiplexed. To distinguish the incoming messages, procedures that will send and receive multiple messages are encapsulated using so-called orders and each message that is sent contains the unique ID of the order. Upon reception of a message, the master thread that is responsible for the connection will use this ID to relay the message to the correct receiver. In Figure 4.16, an example of this concept is shown. Each client can have multiple active threads and each thread can execute an own order with multiple round trips of messages. As orders are only used for procedures that send and receive multiple messages, unsolicited messages like notifications do not require an own order and are send with empty order ID. Notifications are used to, e.g., indicate a status update of a data block after executing a task.

The structure of an actual message is shown in Figure 4.17. The first field indicates the ID of the order. This field contains zero, if it is an unsolicited message. The following fields contain the message type, the size of the payload and the size of the extra data. Depending on the message type, the payload contains a struct that carries additional information for this message type, e.g., the kind of task that shall be executed. To send a larger amount of data, e.g., the input data for a task, further so-called extra data can be appended to a message. Payload and extra data are handled as separate fields in order to keep the extra data also as a separate object in memory. For example, if the extra data contains the input data for a compute task, the memory space for the extra data is individually allocated and persists after the message has been processed and the header and payload have been freed.

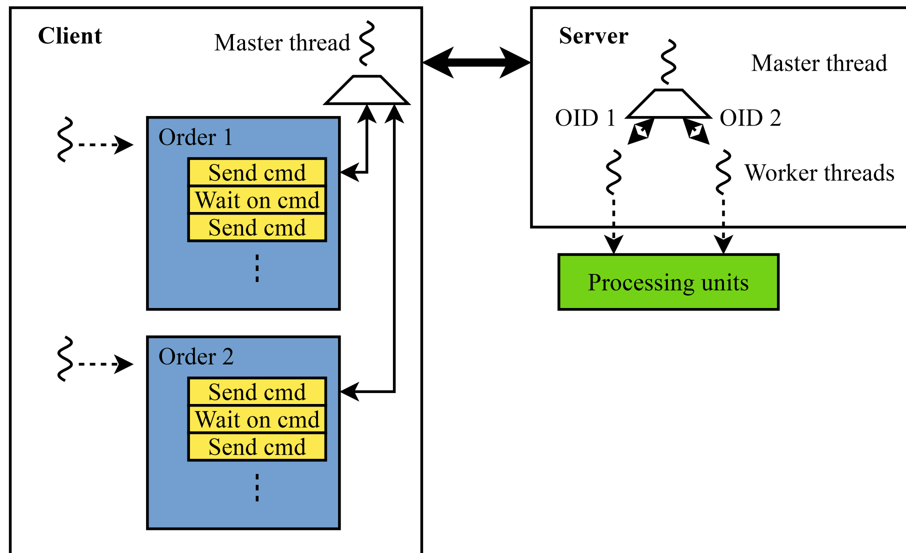


Figure 4.16.: Example for a multiplexed connection between client and server

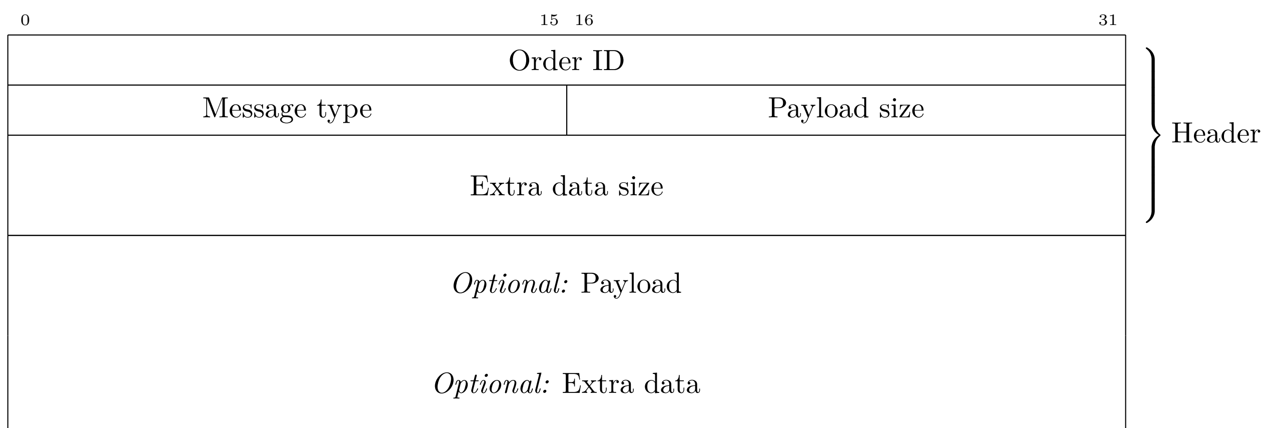


Figure 4.17.: Structure of a message

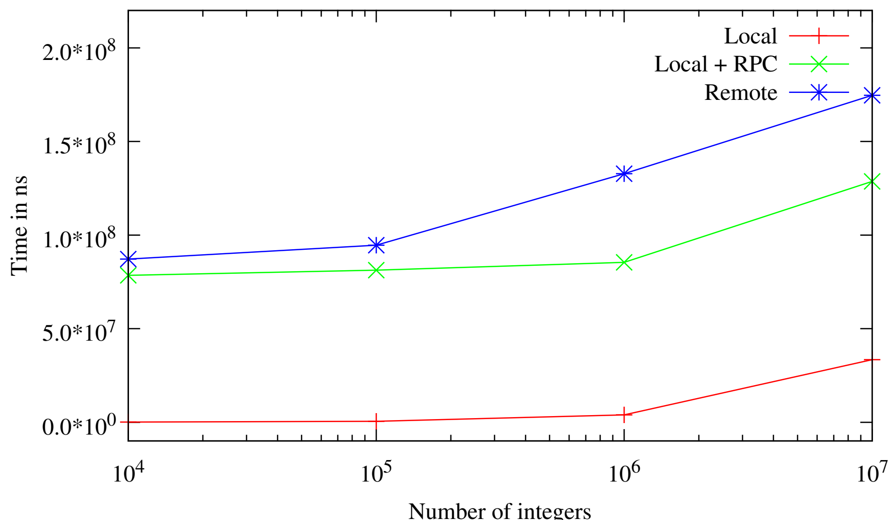


Figure 4.18.: Time for increasing integer arrays on a CPU using local and remote procedure calls

Compared to other hardware plugins that are loaded on demand, the RPC plugin is always initialized as it is unknown if a remote host contains an interesting processing unit. During initialization, the plugin connects to each given host, starts a new order, sends a message to requests the lists of available processing units and waits on returning messages. After receiving the request, the server will send the properties of each available processing unit as extra data to the client and the client will end the order afterwards.

To migrate a task to a remote processing unit, the runtime system starts again a new order and submits the properties of the task mapping to the remote host. The remote host starts the execution and if the implementation requests certain data, it checks if the data is already on the server and if not, requests the data from the client. After execution, the return value of the implementation is passed back to the client which can either immediately request the output data or close the order and fetch the output data later on demand. In the current version, a task migrated to another host is always executed on the other host. In future work, the remote host may also transfer the task further if it is aware of an even better suited processing unit in the network.

To evaluate the minimum overhead for executing a remote procedure call, an application was used that simply increases an array of integers with variable length. During the first two measurements, the kernel function is either called directly or through an RPC but always executed on the same system with an Intel i7-3610QM CPU. Hence, uncertainties caused by interconnect hardware can be avoided. During the third measurement, the kernel execution is migrated to a remote system with an Intel E5700 CPU over a 100 Mbit/s Ethernet network. As it can be seen in Figure 4.18, the size of the array has a similar impact on the runtimes on the local system but the execution through an RPC introduces an almost constant overhead of about 80 ms. For 10,000 integers, the call to the remote host consumes 90 ms and increases further for larger arrays as the remote CPU is also slower than the local CPU.

5. Implementation management and application portability

This chapter proposes a decoupling concept for applications to make them portable across differently configured systems by dynamic selection of suitable implementations and hardware as illustrated in Figure 5.1. First, the problems of current development models are explained and how they are solved with the new method. Second, the implications of the new method on the source code and the build systems are discussed, followed by a description of how the hardware-specific implementations are loaded on demand and how the development of implementations itself can be supported with the existing mechanisms of the runtime system. Afterwards, the mechanism to assure compatibility between applications, implementations and available hardware is introduced. In the final section, the proposed method is then evaluated. This chapter is based on a prior publication [79].

5.1. Introduction

In order to use the best processing unit in a system, an application needs a task implementation for this unit. If such an implementation is not present, another processing unit has to be used and this might lead to inferior performance. An initial idea to avoid such

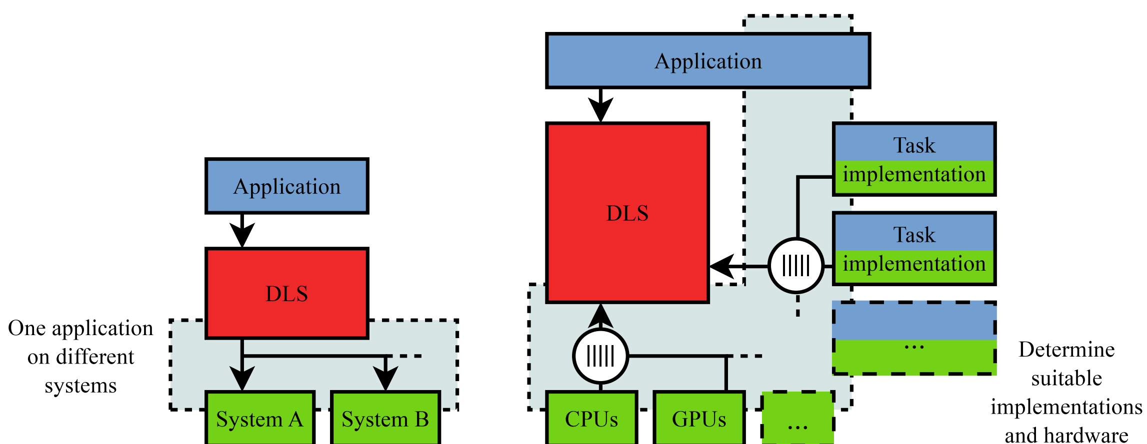


Figure 5.1.: Application portability with dynamic implementation management

problems is to include as many implementations for different architectures as possible in an application. However, such a so-called fat binary approach limits the portability of applications due to the additional dependencies for the individual toolchains and software stacks with runtime libraries. These dependencies have to be met both on the developers' systems and on each end-user system regardless of the availability of an accelerator because the application can neither be built nor executed without having all the corresponding libraries available. If a required library is not installed, e.g., because there is no corresponding processing unit in the system, application startup is aborted without the opportunity to fallback to CPU-only computation. Furthermore, those toolchains and runtime libraries can have their own dependencies, and in the worst case these can be incompatible due to different versions. Thus, installing an application as fat binary may rapidly require severe additional administrative work for potentially superfluous implementations and runtime libraries.

One approach to solve this problem are unified programming models based on just-in-time compilation like OpenCL that compile the kernel source code for an available processor during runtime. However, different studies have shown that OpenCL does not achieve the same performance of dedicated programming models due to less mature compilers and missing support for special hardware features [45, 86]. Another approach is to create an own application binary for each type of accelerator. However, this would prevent the application from using different accelerators in parallel.

Therefore, this chapter introduces a concept that decouples the applications from hardware-specific implementations and reduces the strict dependencies of the applications. The decoupled implementations can be shipped as independent libraries with the application but are only loaded on demand if their dependencies are met.

Besides finding a working implementation and a matching type of architecture for a task, developers might need to impose further restrictions on the implementations or hardware, e.g., a minimum amount of memory or specific hardware features. To ensure such requirements, this chapter describes so-called attributes [112, 113] that enable a developer or system administrator to denote requirements and abilities of applications, implementations and hardware. During selection of implementations and hardware for a task, the runtime system compares these attributes and excludes improper combinations from the selection.

In addition to increasing the portability of applications, this approach offers significant benefits regarding code reuse. By adding only small additional management information to the libraries, like implemented functionality and function signatures, libraries can even be automatically reused between different unrelated applications. If the management information of a library matches the requirements of an application, the runtime system loads the library, and the application can benefit from the corresponding accelerator without recompilation or binary modification. With such a mechanism, it is also possible to automatically extend old applications with support for future accelerators. As depicted in Figure 5.2, hardware vendors may bundle their hardware device like an FPGA with a tuned library that might not even have existed at the application's compile time.

5.2. Related work

Examples for a fat binary approach are Apple's Universal Binaries and EXOCHI [160]. The latter consists of the Exoskeleton Sequencer (EXO) architecture and the C for Heterogeneous Integration (CHI) programming model. The EXO part integrates heterogeneous accelerators through an MIMD extension to the x86 ISA and a shared virtual memory concept. The CHI programming model allows to include accelerator-specific assembly and domain-specific languages in a C/C++ environment by extending the OpenMP pragma approach.

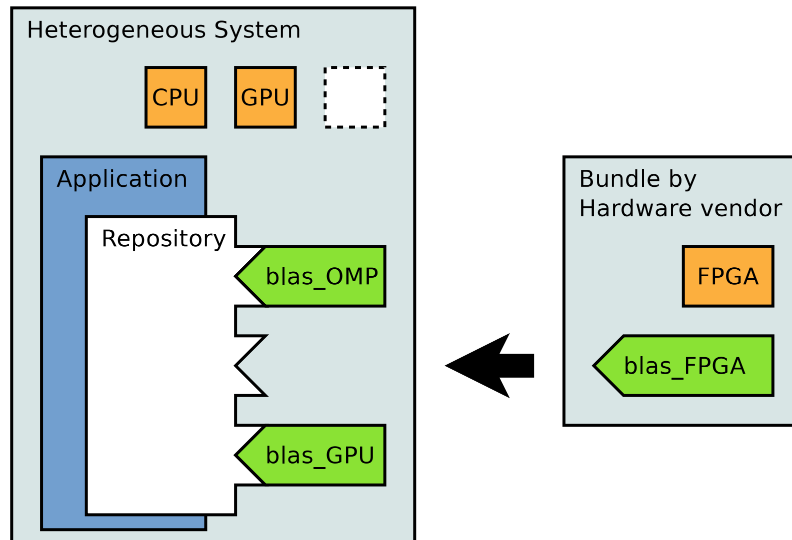


Figure 5.2.: Transparent extension of existing applications by a HW/SW bundle

To adapt applications to different heterogeneous systems without modifying source code, Sandrieser et al. present their XML-based Platform Description Language that can be used to express information about the platform and its architecture [135]. With a description of the platform, a source-to-source translator converts pragmas in the application source code into platform-specific code to utilize optimized implementations for this architecture. In contrast to this work, only the source code of the application is portable but not the resulting binary.

In [97], the authors also see the rising problem of diverse hardware in heterogeneous systems. They briefly introduce annotations to choose a suitable implementation from an available set. However, in their solution the compiler analyzes the annotations and, from them, creates so-called dispatch wrapper functions that determine whether an implementation is suitable. In contrast, the solution in this work is compiler-independent and allows varying amounts and values of attributes. An evaluation of their methods is completely missing.

Similarly, the so-called elastic computing framework is another project that separates general application logic from hardware-specific implementations [164]. Likewise, the application developer specifies the functionality and the runtime system chooses an implementation based on empirical measurements. The implementation can in turn call other implementations and start execution on multiple processing units in parallel. In contrast to the decoupled concept in this work, they do not consider the impact of the different implementations on the dependencies of the application which may inhibit application startup on systems without required runtime libraries.

A critical point regarding application portability is also the required CPU instruction set architecture. However, this work only considers portability with regard to compute kernels. In order to execute an application on different types of CPUs, other works, e.g., as presented by Cha et al. are necessary [28]. They discovered byte sequences that form a valid order of instructions for different instruction sets but on each CPU these instructions either perform a jump to the actual instructions for this CPU or perform pointless instructions similar to a NOP (no operation). Using this byte string, the application can be executed on any of the supported CPUs but only the parts of the application with the correct instructions for the respective CPU are executed.

Despite the binding to a specific CPU instruction set, this work enables dynamic exploitation of instruction set extensions. For example, implementations that use vector instructions

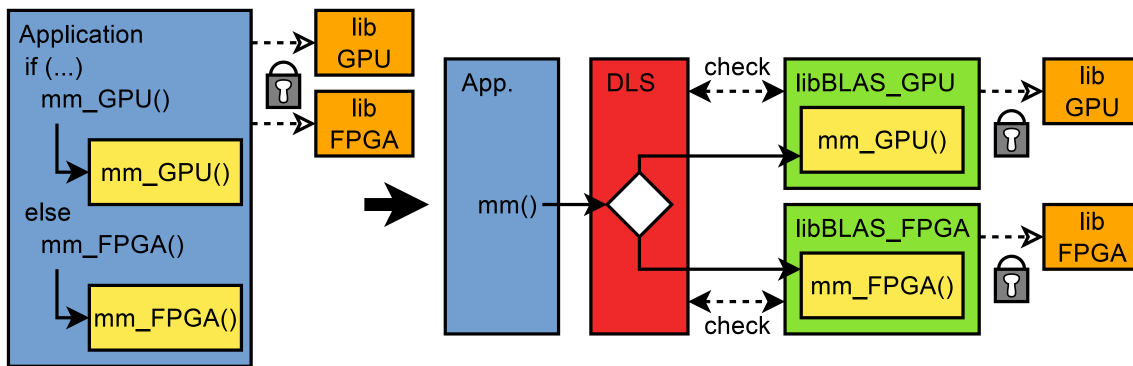


Figure 5.3.: Separating applications from accelerator-specific implementations

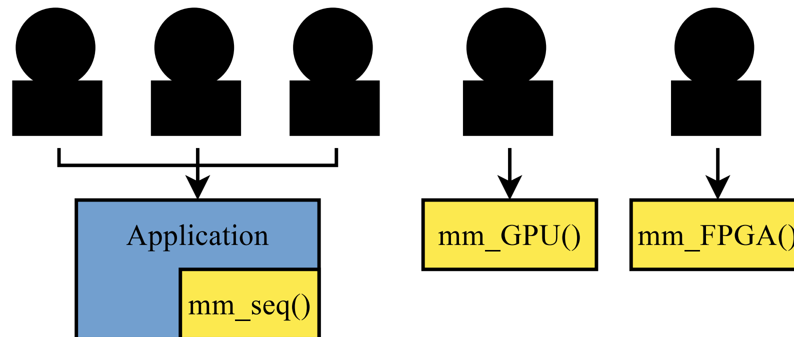


Figure 5.4.: Division of work in large projects

like SSE or AVX can be outsourced nonetheless in separate libraries and the runtime system will only use these optimized implementations if a CPU is present that supports these instructions.

5.3. Decoupled application development

To reduce the dependencies of applications, this work proposes a decoupled development concept that outsources special implementations into separate libraries. As shown in Figure 5.3, the two implementations `mm_GPU` and `mm_FPGA` are not included in the application binary anymore but reside in separate libraries called `libBLAS_GPU` and `libBLAS_FPGA`. Instead of the application, the two libraries will depend on the accelerator-specific runtime libraries. If the application wants to calculate a matrix multiplication, the runtime system will look for available implementations and only load those with satisfied dependencies. By using this concept, applications and implementations can be built and distributed individually. This also resembles the division of work in larger projects as illustrated in Figure 5.4. There are developers, e.g., engineers or natural scientists, working on the general application logic and specialized developers that create optimized task implementations for different processing units.

5.3.1. Implications on source code and build systems

In Figure 5.5, the process and the required components for compilation and execution of an application are depicted: first, the source code files for the CPU and accelerators are compiled with their respective compilers or toolchains. Then, the linker creates an

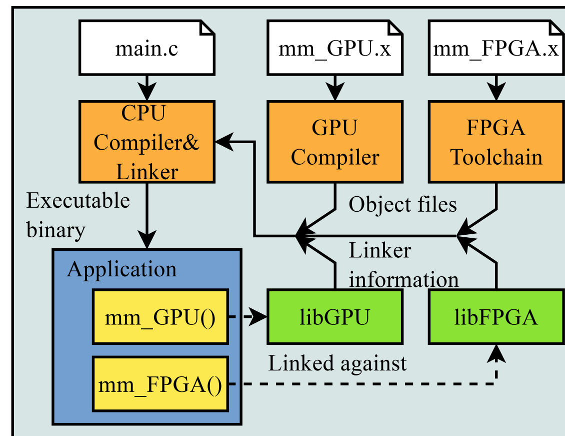


Figure 5.5.: Regular compilation – every toolchain and runtime library is required to create an application binary

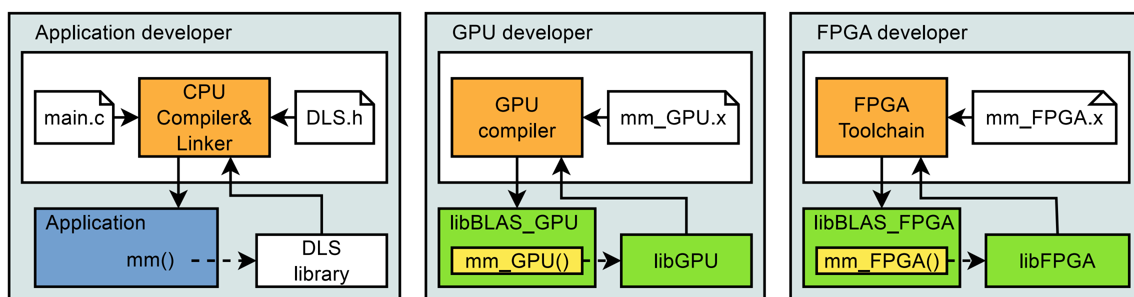


Figure 5.6.: Decoupled compilation of applications and hardware-specific implementations

executable from the resulting object files and links it against the runtime libraries `libGPU` and `libFPGA` of the accelerators.

With the decoupled concept, the resulting parts are compiled individually as depicted in Figure 5.6: the application is only linked against the library of the runtime system and every hardware-specific implementation is compiled separately with its respective toolchain as illustrated for the GPU and FPGA, for example. Hence, application and kernels can even be developed individually on different systems and not every developer needs all the accelerators and their software stacks for the experimental implementations.

5.3.2. Assembling applications and implementations

Theoretically, every library available in the system could contain an implementation with a desired functionality. However, as a large amount of system libraries exist, inspecting all of these would significantly increase the effort for finding appropriate implementations. Therefore, the runtime system uses an implementation database as a lookup table to quickly find libraries of interest on the system. An example of the implementation database is shown in Table 5.1. The list has two columns: the first column states the functionality, the second delivers the path to the library that contains at least one implementation with the respective functionality. New implementations and new functionalities can be inserted manually into this file or through an installer or package manager.

Before starting a task, the runtime system browses this database and if it finds a matching functionality, it tries to load the library using the dynamic loader of the operating system. If the library has unsatisfied dependencies, the loader returns an error code and the runtime system continues with the next library. If it loads successfully, the runtime system starts querying the symbol table or browsing the string table to find matching implementations.

Functionality	Path to library
mm	/usr/lib/libmm_CUDA.so
mm	/usr/lib/libblas.so
mm	/usr/lib/libmm_OMP.so
sort	/usr/lib/libsort_CPU.so
...	

Table 5.1.: Example of an implementation database

In an ELF binary file, functions and other entities can be found in the address space of an application using so-called symbols. A symbol usually has a unique name and, when resolved, points to a certain address in memory. The symbols are organized in a symbol table and each entry contains a reference to the corresponding entry in the string table. During runtime, the symbol table can be dynamically queried by passing the symbol name to the `dlsym` POSIX function. If a matching symbol is found, the function returns the address of the entity.

With this function, the runtime system probes the symbol table for entities that follow a specific naming scheme. To find functions for a proxy, it will look for symbols with the scheme `func_TAG`, where `func` is the name of the proxy, i.e. the required functionality, and `TAG` is one of several tags stored in the runtime system, e.g., the programming model or processing unit like `CPU` or `GPU`. This way, a programmer benefits twice from wisely chosen function names: he can easily identify the purpose of functions and the function is automatically considered as candidate for execution.

The downside of this approach is that every possible combination has to be tried and only implementations with a known tag can be found. Therefore, the runtime system is also able to directly browse the string table for interesting strings [1]. Only if a string is found that starts with the name of the required functionality, the string is passed to the `dlsym` function to test, if it points to a usable implementation.

5.3.3. Supporting fault diagnostics during development

During development of optimized implementations, hardware experts compare their implementations with a reference implementation to determine performance benefits and correctness of the results. Usually, the correctness is verified by storing the results of both implementations on disk and to compare them manually after an application run. However, in an application with multiple tasks, finding a software bug can thus become a time-consuming effort. To simplify this effort, this work proposes a tool based on the mechanisms of this framework to automatically detect a fault during application runtime and to supply a developer with additional information for the debugging.

In Chapter 7, this work will describe its mechanisms for fault tolerance in heterogeneous systems based on redundant execution. With this mechanism, the runtime system is able to overcome faults, e.g., caused by failing devices, during execution without human interaction. Besides tolerating faults during application runtime, this mechanism is also useful during development. Instead of solely executing on multiple devices, the runtime system can also enforce execution with multiple implementations, e.g., a reference implementation and an implementation that is currently in development. Hence, a fault in the new implementation can be detected automatically and a developer does not have to compare the results

[1] using libELF `http://directory.fsf.org/wiki/Libelf`

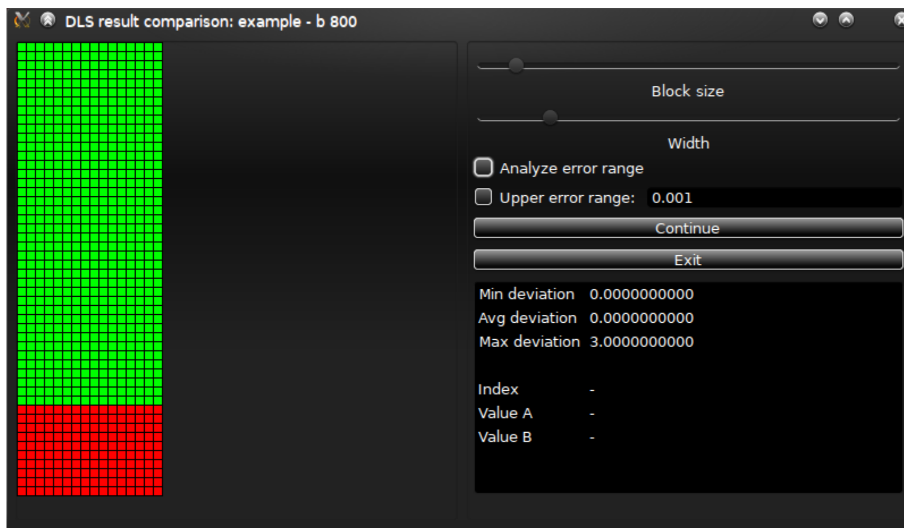


Figure 5.7.: Visualization of incomplete results

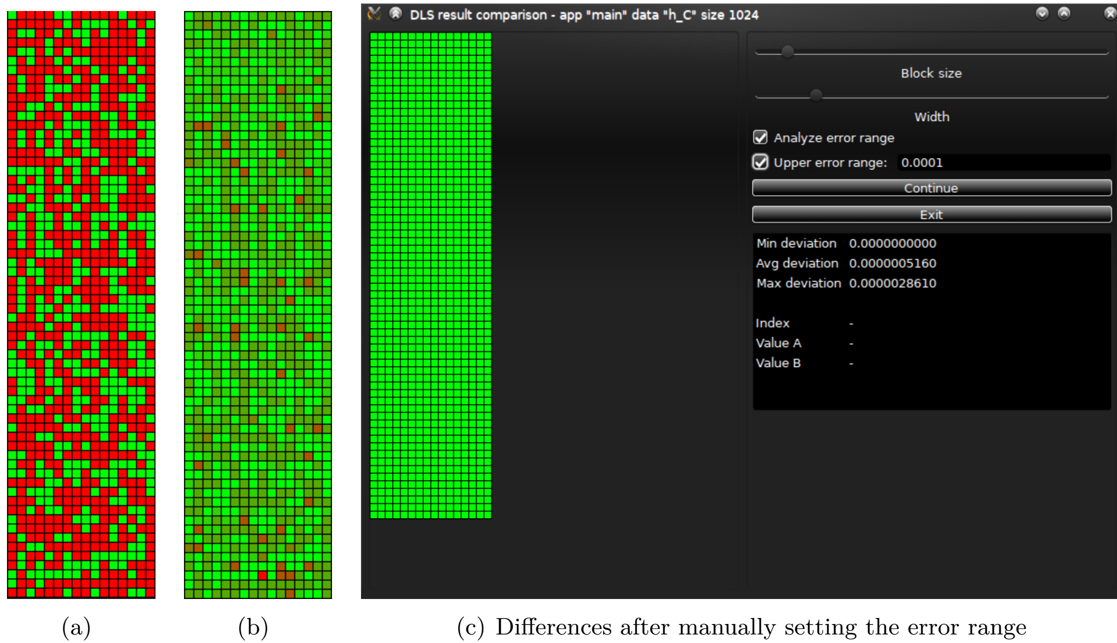
manually – which becomes even more time-consuming if multiple intermediate results have to be compared in order to locate a software bug.

To simplify fault diagnostics or if a mismatch has been found in intermediate results, the runtime system can write the results to disk and call an external program to show a graphical user interface which enables a detailed analysis of the results. In the following, this work introduces such a tool based on the existing mechanisms of the framework and gives examples for faults where such a tool can give valuable hints for debugging.

A common fault during GPU programming is, for example, a wrong calculation of the required threads and thread blocks. While starting too much threads usually results in a segmentation fault that is easy to perceive, forking less threads than necessary results in incomplete data. With the comparison tool of this work, such a case would be presented to a developer as shown in Figure 5.7. On the left side, each value is represented by a colored rectangle. A green rectangle represents a matching value and a red rectangle represents a mismatch. On the right side, control interfaces are available that can be used to, e.g., change the number of shown rectangles in each row in order to align this value with the number of threads in a block. In this example, we can easily see that the last values of the results differ.

As we will see in Chapter 7.3.4.2, a common problem during development of implementations for accelerators are floating point results that are both correct but still differ to a certain extent. In Figure 5.8, the comparison of results of a matrix multiplication on a CPU and a GPU are shown. In Figure 5.8(a), we can see that the differences are randomly distributed. If we activate the error range display, the drawing looks like Figure 5.8(b). In addition to the first figure, we can see that the differences vary randomly as well. For specific applications, it is necessary that the inaccuracy is below a certain threshold. For such cases, a developer can specify an upper relative deviation which results in a display of Figure 5.8(c) that shows that there are only deviations below 0.0001%. Hence, a developer can easily determine if the deviations are acceptable or indicate a problem with the code.

Another example for a fault during calculations is shown in Figure 5.9. On GPUs, the threads inside a block can be synchronized and share intermediate results. If such an intermediate result differs, the deviation may propagate through all threads in a block. In Figure 5.9(a), the resulting display of such an example is shown. As we can see, the first value in a block is correct while the others differ. If we activate the error range display as



(a) (b) (c) Differences after manually setting the error range

Figure 5.8.: Analysis of differences between CPU and GPU floating point calculations

shown in Figure 5.9(b), we can see that the the deviation grows inside a single block as the line length has been chosen to equal the number of threads in a block in this example. Hence, with little effort, a developer receives additional information which simplify the localization of bugs.

5.4. Balancing requirements and abilities

This chapter introduced a mechanism to assemble applications with related or unrelated implementations that implement the required functionality. However, especially with unrelated implementations, it is possible that the implementation is not compatible with the application despite the matching functionality, e.g., due to a different function signature. To solve this, it is either necessary to establish a convention that enforces compatibility, e.g., by creating special functionality strings like `mm_single` for a matrix multiplication with single precision, or to provide a separate mechanism that enables a developer to specify detailed requirements of the application and abilities of the implementations and hardware.

For the latter solution, the runtime system offers the possibility to tag tasks, implementations and processing units with so-called attributes [112, 113]. An attribute is either a `key=value` string tuple or a flag in a bitfield. With the attribute tuples, a developer can set a key to a specific value, e.g., the function signature: `signature=int f(char *)`. For numerical attributes, it is also possible to give a range of values. On the other hand, attribute flags are a special type of attributes that enable the runtime system to efficiently store and compare requirements and abilities which are either present or not. For example, flags can be used to indicate required hardware features like support for SSE or AVX if an implementation uses corresponding instructions. Further examples of existing attributes are shown in Table 5.2.

In general, the programmer is responsible for the declaration of attributes. Some attributes like the signature could be generated automatically by a compiler. Most other attributes like those presented later in the evaluation highly depend on the type of computation and usually result from a performance-quality trade-off. Thus, they are difficult to determine

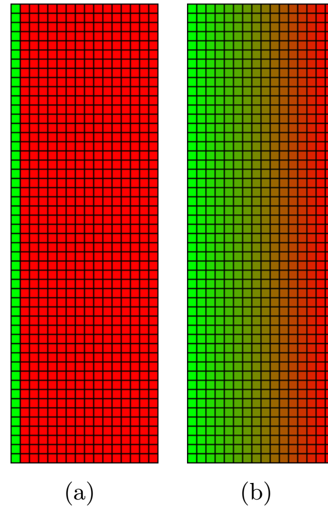


Figure 5.9.: Display of faults causing increasing deviations per thread block

Key	Type	Description	Example
`signature`	tuple	Function signature	`int f(char *)`
`psize`	tuple	Problem size	100, 5000
`pmodel`	flags	Programming model	CUDA, OpenMP
`cpu_features`	flags	ISA extensions	MMX, SSE*, AVX

Table 5.2.: Examples of attributes suitable to express requirements and abilities

automatically, e.g., because the compiler does not know how much accuracy is required. Therefore, the use of attributes is not constrained and the programmers can freely denote new attributes that fit their needs. However, special care is required as putting too much constraints on the implementation may result in an empty set of suitable implementations.

The runtime system offers different ways to specify attributes. To denote requirements of an application, the developer can include the attributes in the source code. For implementations, attributes can be specified in the source code as well or added later in an attribute database. Attributes for processing units have to be added to the attribute database. An example how attributes can be specified in the source code is shown in Listing 5.1. In this example, the application requires random numbers and two corresponding implementations are available, `rand_dev_random` reads real random numbers from `/dev/random` of the Linux operating system and `rand_MT_GPU` generates pseudo random numbers using the Mersenne Twister algorithm on the GPU [151]. To request specific abilities, the developer can set the requirements for the next call of `rand` using the `dls_call_attributes` function. In this example, the attribute `rand_type` is set to `real` in order to request real random numbers. To specify abilities of the implementations, a special global variable is set using the `dls_attributes` macro. The name of this variable is equal to the name of the implementation and the suffix `_attr`. During evaluation of the available implementations, the runtime system will lookup these variables similar to the function symbols and compare the attributes. In this example, the `rand_dev_random` function gets the attribut `rand_type=real` and `rand_MT_GPU` gets `rand_type=pseudo`. Hence, during the runtime of the application, the runtime system will choose the `rand_dev_random` and exclude the `rand_MT_GPU` function.

To set attributes independent from an application, e.g., for implementations in libraries or

Listing 5.1: Example for attributes in the source code

```
char * rand_dev_random_attr = \
        dls_attributes(1, "rand_type", "real");
void rand_dev_random( float *numbers ) {
  /* get numbers from /dev/random */
}

char * rand_MT_GPU_attr = \
        dls_attributes(1, "rand_type", "pseudo");
void rand_MT_GPU( float *numbers ) {
  /* get numbers with Mersenne Twister */
}

int main (int argc, char **argv) {
  float *numbers;

  dls_call_attributes(rand, 1, "rand_type", "real");
  rand(numbers);
}
```

Entity name	**Attributes**
mm_CPU	signature=void (float*, float*, float*)
FPGA1	memsize=128MB
...	

Table 5.3.: Example of an attribute database

for processing units, the runtime system provides an attribute database. Similar to the implementation database, the attribute database has two columns as shown in Table 5.3: the first column contains the ID of the entity, e.g., the implementation's symbol name or the device name, and the second column contains the corresponding list of attributes.

5.5. Evaluation

For evaluation, the systems given in Table 5.4 were used: two GPU systems (System A and System D), a 12-core SMP system (System B), and an FPGA system (System C). Each GPU system has one NVIDIA card that is programmed using CUDA. On the FPGA system (System C), H-MOL [87] is employed as the runtime system for the UoH HTX Board [50]. All systems use an x86-64 Ubuntu Linux operating system. If not stated otherwise, the numbers represent the average of 30 consecutive runs. In some of the following scenarios, more than one suitable implementation remained after applying the attribute filter. In such a case, the runtime system chose the fastest implementation that is determined using the mechanisms that are later explained in Chapter 6 and 7.

5.5.1. Overhead for matching requirements and abilities

In the first experiment, the overhead for using attributes is evaluated with varying numbers of attributes and functions. In order to get a worst-case estimation, the experiment is

System	Type	Hardware	Progr. Model
A	CPU GPU	2× AMD Opteron Quad-Core 2378 NVIDIA GeForce GTX 275	OpenMP CUDA
B	CPU	2× Intel Xeon X5670 Six-Core	OpenMP
C	CPU FPGA	AMD Opteron Dual-Core Processor 870 Xilinx Virtex-4 FX100	OpenMP H-MOL
D	CPU GPU	Intel Core2 Quad NVIDIA GeForce 8400 GS	OpenMP CUDA

Table 5.4.: Evaluation systems and the employed programming models

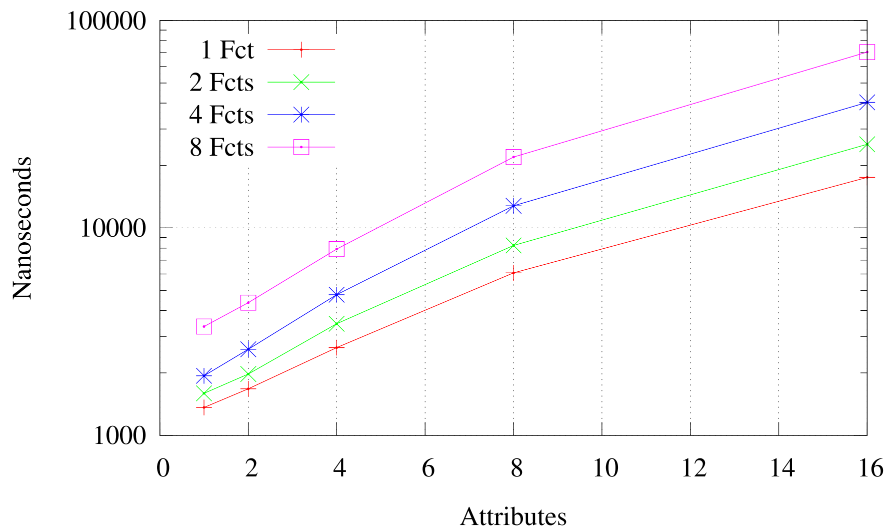


Figure 5.10.: Time overhead for a kernel invocation in relation to number of attributes and functions

performed as follows: for a single run, the application's function call as well as every called function are assigned the same number of attributes. The attributes are sorted in the most pessimistic way so that every attribute has to be compared and a decision cannot be made until the last attribute. Under realistic conditions, the comparisons will typically stop earlier upon detection of first mismatching attributes which causes less overhead. In Figure 5.10, the individual worst-case time consumption for different combinations is depicted. We see that the overhead depends on both, the number of attributes and the number of functions.

5.5.2. Performance with Rodinia benchmarks

In this experiment, it is shown that this work introduces a negligible overhead in applications targeting heterogeneous systems using the Rodinia benchmark suite and that it makes these applications portable across different systems. To assure that the executables and decoupled implementations are equal on all systems during the evaluation, the benchmark directory was mounted on every system using the network file system (NFS).

In Figure 5.11, we see the runtimes of the applications on System A with the different approaches: the DLS runtime system and the native versions using CUDA and OpenMP. As we can see, the application equipped with the runtime system has a similar runtime as

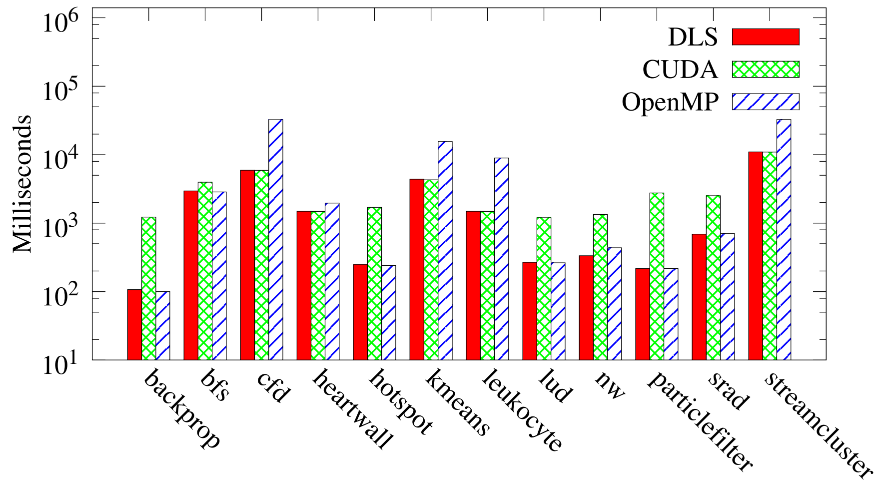


Figure 5.11.: Runtime of Rodinia benchmarks with our DLS, CUDA and OpenMP on GPU system A

the fastest one of the OpenMP and CUDA applications. This shows that the DLS runtime system successfully detects the available and the fastest implementations.

On System B, the very same applications were started. As this system has no GPGPU-capable GPU, only the DLS and OpenMP results are shown in Figure 5.12. Although there are no CUDA GPUs and CUDA runtime libraries available in this system, all applications can still execute because they are no longer linked directly against these libraries.

5.5.3. Use case: random number generation

Random numbers are essential for certain tasks in cryptographic applications and simulations. In this scenario, various ways to gain random numbers on a heterogeneous system with the Linux operating system, their differences and how a developer can use attributes are shown.

For evaluation, a repository of pseudo-random number (PRN) generator libraries based on the Mersenne Twister (MT) algorithm and on the mechanisms of the Linux operating system were built. For the CPU, there is a Mersenne Twister implementation based on the Dynamic Creator Library[2]. An OpenMP version calls several MT random number generators in parallel. An FPGA implementation of MT was created by integrating the freely available hardware description [3] into the H-MOL accelerator framework for FPGAs [87]. From the Linux system two methods were included: the GNU C library function `random()` and the kernel-based `/dev/random` virtual file. `/dev/random` delivers high-quality, *real* random numbers as long as the entropy pool of the kernel is filled. In contrast, `random()` produces only *pseudo* random numbers at a high rate. So, if a large amount of numbers is required, `/dev/random` can be unpredictably slower than `random()`. In general, `/dev/random` delivers the random numbers with the highest quality. The MT algorithm provides the highest quality for pseudo random numbers and `random()` achieves the lowest quality.

Figure 5.13 illustrates the measurements of the runtime of the individual implementations on System C. Due to initialization overhead of the multiple Mersenne Twisters in the

[2] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html`
[3] `http://www.ht-lab.com/freecores/mt32/mersenne.html`

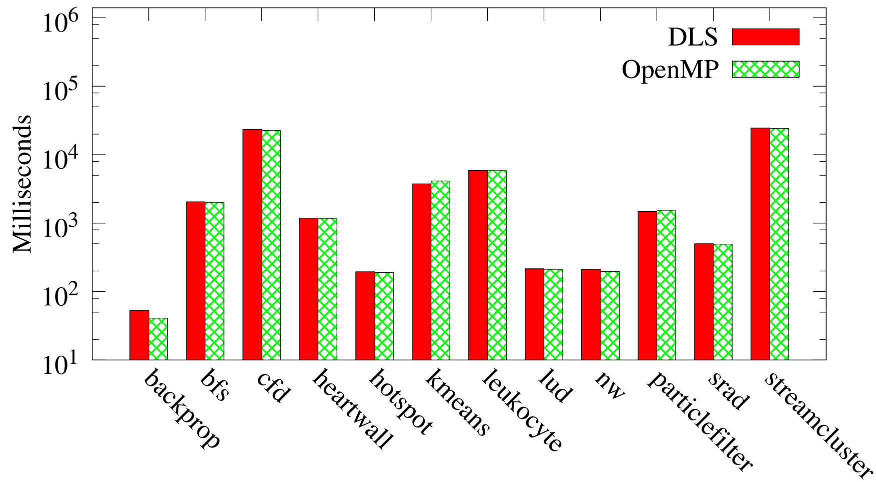


Figure 5.12.: Runtime of Rodinia benchmarks with our DLS and OpenMP on manycore system B

OpenMP version, speed-up is not obtained for less than 1 billion numbers. For problem sizes smaller than 1,000, the FPGA implementation (MT HMOL) performs very well because initialization is very fast. However, due to constraints of the FPGA implementation, results beyond 1,000 requested numbers had to be skipped. For 10 and 100 requested numbers, the time consumption of the `/dev/random` implementation increases rapidly. In order to maintain clarity of the graph, results for higher amounts of numbers were not included.

Due to the above restrictions, retrieval of random numbers poses an interesting challenge for the attribute concept. The *periodicity* attribute can be used to guarantee that the numbers are all different for a requested amount. With *random_type* it is possible to explicitly request *real* random numbers. With *psize*, the ranges for safe and reasonable operation are defined.

Therefore, the implementations were annotated as listed in Table 5.5 and the application requests a good level of quality by `quality=2+` in this example. In Figure 5.13, the time consumption to retrieve the respective amount of random numbers using this approach (DLS) is depicted. For 10 until 1,000 numbers, the runtime system selects the MT H-MOL FPGA implementation as the other MT implementations are too slow. It does not select the `random()` implementation either, although it is faster, because it produces numbers below the required quality. It does not choose the `/dev/random` implementation as well, although it provides high-quality random numbers, because it is much slower and because the quality of MT H-MOL is sufficient. Beyond 1,000 numbers, the runtime system chooses the MT CPU implementation, as using MT H-MOL is forbidden, MT OpenMP is too slow, and the quality of `random()` is still too low. As we see, the runtime system successfully chooses the most suitable from several executable implementations although the application has severe restrictions regarding the execution.

5.5.4. Portable MPI application

In a further experiment, an application calculating a double-precision matrix multiplication using MPI was created. The decoupled approach is especially interesting for such applications, as usually multiple instances of the same MPI application run in parallel on multiple hosts in a network. If the system is heterogeneous and the hosts contain different types of processing units, all required libraries for any type of processing unit in the system would

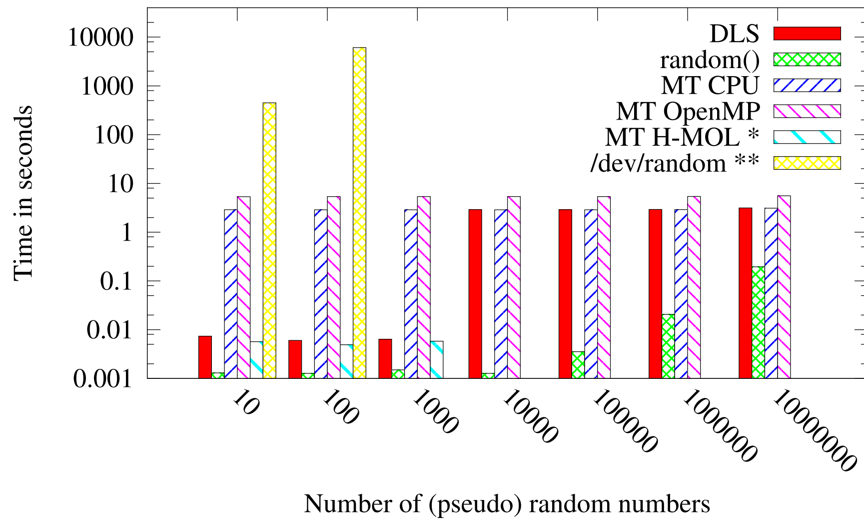


Figure 5.13.: Average runtime of Mersenne Twister library and built-in random-number generators on System C
(* = hardware restrictions apply; ** = 8 runs for 100)

Attribute	**Implementation**	**Value**
`quality`	/dev/random MT * random()	3 (high) 2 (mid) 1 (low)
`periodicity`	/dev/random MT * random()	- 2^{19937} 2^{35}
`random_type`	/dev/random MT * random()	real pseudo pseudo
`psize`	/dev/random MT H-MOL	0-10 0-1,000

Table 5.5.: Attributes for random-number generators

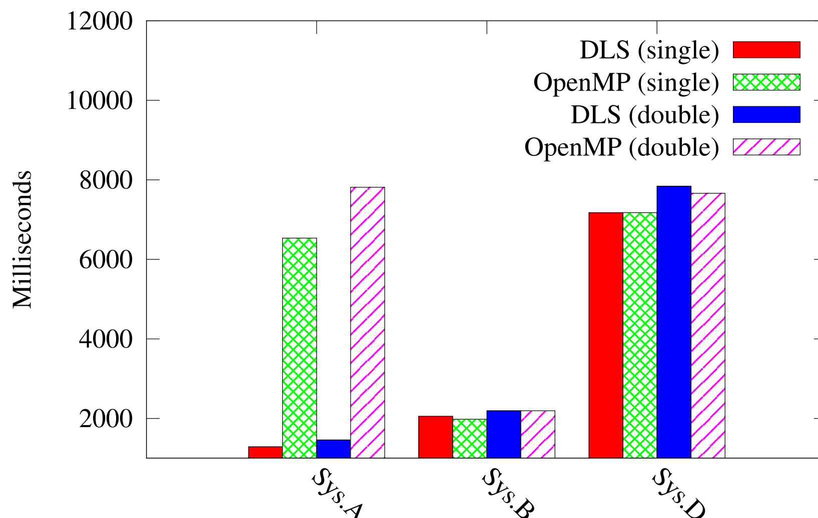


Figure 5.14.: Runtime of matrix multiplication example on the single MPI hosts

have to be installed on every host. With the decoupled approach, only the libraries for the processing units on the respective host have to be installed.

In this experiment, a serial, an OpenMP, and a CUDA implementation were again available for the application. Hence, with the runtime system, the MPI application is able to take advantage of locally available accelerators on every host without recompilation or installation of superfluous runtime libraries. On System D, a special obstacle is that the GPU does not support double-precision floating point numbers in hardware. Therefore, the GPU is marked with the `precision=single` attribute. To test the behavior of the runtime system nonetheless, a single-precision variant of the application was created as well. In Figure 5.14, the individual results of both variants on the three systems A, B, and D are shown. For better clarity of the diagram, the results of the CUDA implementation were omitted. As we can see, on System A, the application profits from the powerful GPU. On the CPU-only System B the application can only exploit the high number of cores. However, on System D, the system with the low-performance GPU, the runtime system chooses the OpenMP implementation not only for double-precision but also for single-precision computation. This is caused by the low performance of the GPU, as it can not even outperform the calculation on the CPU for single precision.

6. Establishing cost awareness

To make the right decision, the runtime system needs to be aware of the presumable costs of each necessary operation. These costs, however, depend on additional parameters like the problem size and the system state as illustrated in Figure 6.1. In the first section of this chapter, the importance of cost awareness in heterogeneous systems is motivated. In the second section, the elementary mechanisms of the runtime system to determine costs are presented. Afterwards, the impact of data locality and memory management on efficiency is shown as well as the mechanisms to account for these factors. In the fourth section, this work explains how the runtime system detects and resolves resource conflicts that would otherwise lead to inappropriate decisions. Similar, faults during execution can have a negative impact on the performance and falsify the estimations. In the last section of this chapter, a method to account for faults during cost estimations is presented and how the methods in this chapter can help to detect faults in the first place. Parts of this chapter are based on prior publications [76, 79, 77].

6.1. Introduction

The benefit of heterogeneous systems is the availability of different specialized architectures that provide improved performance for certain types of calculations. One of the major problems for applications on heterogeneous systems is the question which of the available

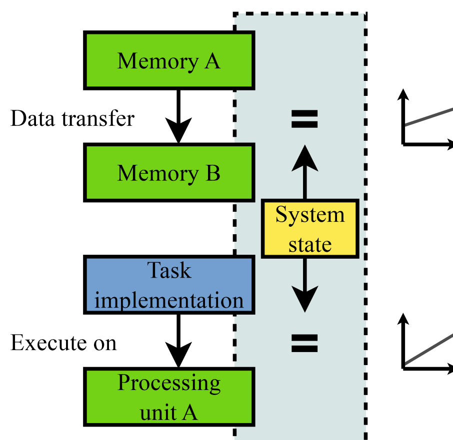


Figure 6.1.: Dermining costs for operations under given circumstances

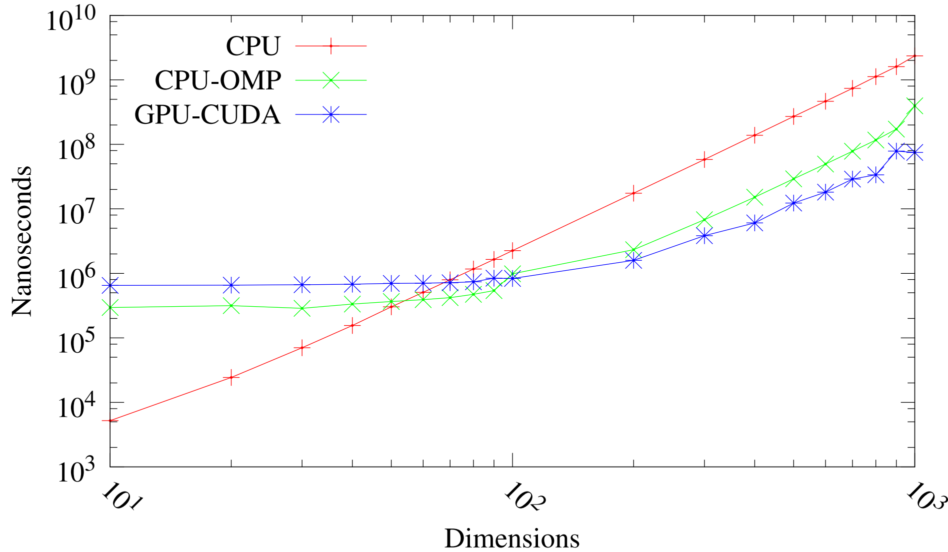


Figure 6.2.: Square matrix-matrix multiplication with different sizes and implementations

processing units provides the shortest execution time for a specific task. In many cases, different algorithms are required for the individual architectures in order to address the special features of the hardware, e.g., a CPU core is optimized for applications with a high amount of control flow instructions while the GPU is most suited for applications with high amount of calculations and parallelism. Besides the algorithms, the execution time also depends on the problem size or input data size of the specific task. As motivating example, the execution time of a square matrix-matrix multiplication was measured on different processing units with a varying size of the matrices. In Figure 6.2, the required execution time for a serial and a parallel OpenMP implementation for the CPU as well for an implementation executing on an Nvidia CUDA GPU is shown in logarithmic scale. As we can see, every implementation provides the shortest execution time of all three for a certain range of matrix sizes. Until a size of about 50x50, the serial CPU implementation provides the shortest execution time. From there until a size of approximately 100x100, the parallel OpenMP implementation exhibits the shortest time as the overhead for thread creation is compensated by the benefit of parallel calculations. Beyond 100x100, the benefit of massive-parallel execution on the GPU surmounts the overhead for initialization of the computation on the GPU and significantly reduces the time consumption compared to CPU computation. The problem for application developers is that these curves and thus their intersection points depend on the employed hardware and the system state. Therefore, statically relying on a certain processing unit on any system might significantly increase application runtime and hamper performance portability.

With the mechanisms from Chapter 4, it is possible to dynamically adjust application execution during runtime and choose one of the available implementations for a certain task. The following question is, how to determine the best choice for a system that is unknown at the application's compile time.

Estimating the benefit of possible options is an elementary task for decision making. In computing, assessing an option is usually achieved with either analytical or empirical predictions. Estimating the costs for execution on a certain processing unit analytically is difficult in a common computer system as there are many ways the execution can be disturbed by unforeseeable events like interrupts or competing processes in a multi-tasking operating systems. Even for predictable single-user environments, accounting for the details of all the hardware involved in execution, e.g., caches and interconnection networks like

PCIe, is necessary to make a precise assumption. An analytical method is not feasible as modeling all involved hardware is expensive and on a multi-tasking system, events that disturb execution are hard to predict. On the other hand, the downside of an empirical approach is the necessity to try and measure the available options which can result in considerable overhead in case some options constitute significantly worse behavior compared to the others.

The proposed mechanisms in this work rely on an empirical approach using the average time consumptions of past execution as it simplifies the determination of costs and also includes the average overhead in case of competing applications. As an empirical approach introduces additional overhead, this work also uses different mechanisms to avoid additional measurements.

6.2. Learning costs during application execution

In order to determine the best implementation and hardware for a task, the costs for executing the algorithm on all processing units have to be known. In many cases, these costs scale with the size of the to-be-solved problem of the task. This problem size can be influenced by different factors like the input data size, e.g., the size of the matrices for a multiplication. As the problem size can often be determined in advance, it is used by most dynamic schedulers to estimate the costs. These estimations are based on prior training that is conducted offline [58, 149] or online with actual data [122, 67, 13, 164, 121]. In this work, the scheduler is also trained online with actual data in order to gain as accurate estimations as possible about the performance. As the costs may vary between different runs, e.g., due to the scheduling of the operating system, and the first execution can be unusually high due to initialization effects, the runtime system requires that the costs are measured at least twice to gain an impression of the variances. These measurements are also repeated in regular intervals to review their correctness. To avoid too frequent checks, a slowdown factor can be configured and the runtime system will only schedule checks if the average overhead stays below this value.

Although variances of the costs are considered, this work and most related projects expect that the costs of a task are stable in principle, i.e. on an ideal system with no interrupts and resource competition, for example, the costs stay exactly the same during executions with the same problem size. However, there also exist algorithms that may exhibit unpredictable performance, e.g., if the performance depends on the actual values in the input data. For example, sorting algorithms may need to move every item in a list or may immediately return if the list is already sorted. Other methods have been proposed to handle such so-called irregular execution times [138] but such applications are not yet considered in this work.

To actually measure costs, the runtime system contains a plugin-based sensor interface. With this interface, different types of sensors, e.g., that monitor specific hardware, can be registered similar to the hardware interface introduced in Chapter 4.6. One sensor integrated in the runtime system is, for example, the runtime sensor that feeds the current wall clock time to the runtime system. Other examples are a sensor for Intel's Running Average Power Limit (RAPL) interface that provides an estimation of the current power consumption of the CPU and GPU cores on modern Intel CPUs starting with the SandyBridge architecture [130] or a `lm_sensors` [1] plugin that provides unified access to temperature sensors of different hardware components.

Before tasks are started, a so-called cost vector is created for every task. In this vector, every element belongs to one sensor and stores intermediate data. Depending on the processing

[1] `http://www.lm-sensors.org/`

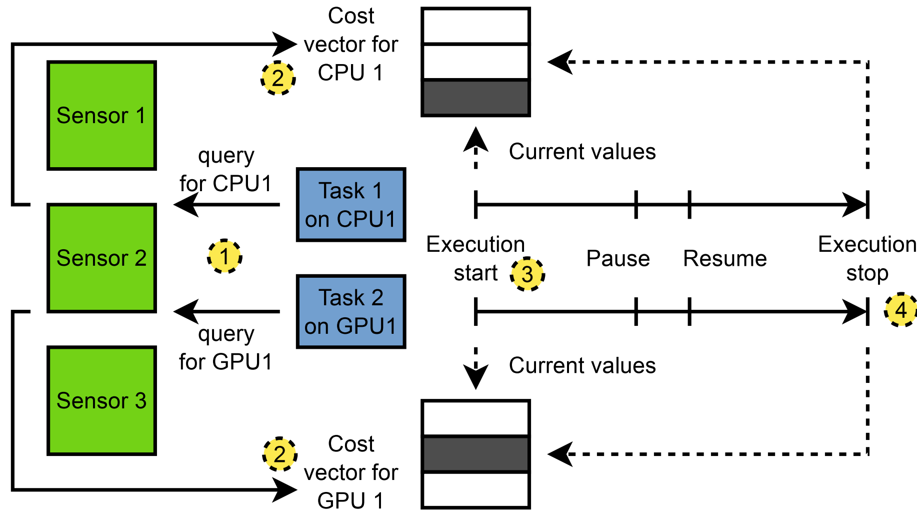


Figure 6.3.: Measuring process for two tasks using different sensors

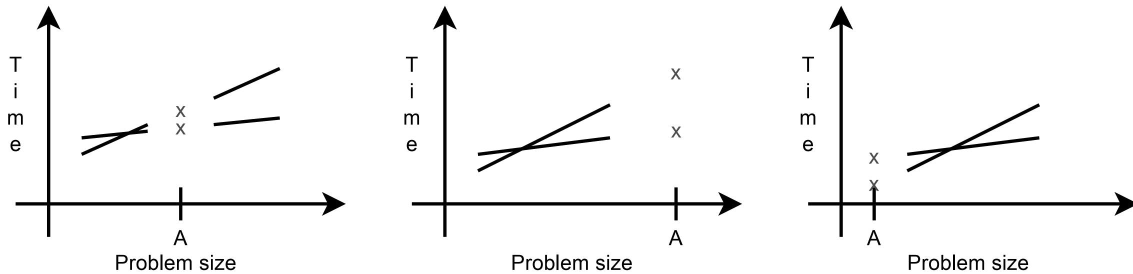


Figure 6.4.: Interpolation and extrapolation of missing cost values

unit the task is executed on, only some of the elements may be in use as indicated in Figure 6.3. For example, the runtime sensor can be used with every processing unit but the RAPL sensor only works with recent Intel CPUs. In Figure 6.3, only sensor 1 and 2 are active on the CPU and sensor 1 and 3 on the GPU. Just before the task starts, the active sensors are queried and their current value is stored in the cost vector. During task execution, the measurements can be arbitrarily suspended and resumed to exclude specific operations from the result. For example, later in Chapter 6.5.1, such breaks are necessary to exclude operations that may cause unpredictable variations. After the task finished, the difference between the measured values is then stored for following decisions.

In its current state, the runtime system also assumes that the costs of executing a certain task are high enough for the used sensors to give reasonable results. For example, if tasks have only a short execution time, the relative uncertainty raises due to imprecise sensors. To circumvent this problem, different statistical approaches have been introduced to generate useful cost predictions nonetheless [91, 38].

The downside of an empirical approach is that the benefit of the options have to be evaluated before a thorough decision can be made. Depending on the differences in execution time, evaluating a slow implementation may result in a considerable performance penalty. A common approach to avoid measuring costs for new problem sizes is interpolation and extrapolation [164]. In Figure 6.4, different situations are shown where the costs for a problem size A are unknown. In the left example, the costs for two alternatives at problem size A can be determined by interpolation of known values. In the other examples, the costs are determined by extrapolation. Besides linear interpolation, another approach is to use the k-Nearest Neighbor algorithm [121] to estimate the costs for new problem sizes.

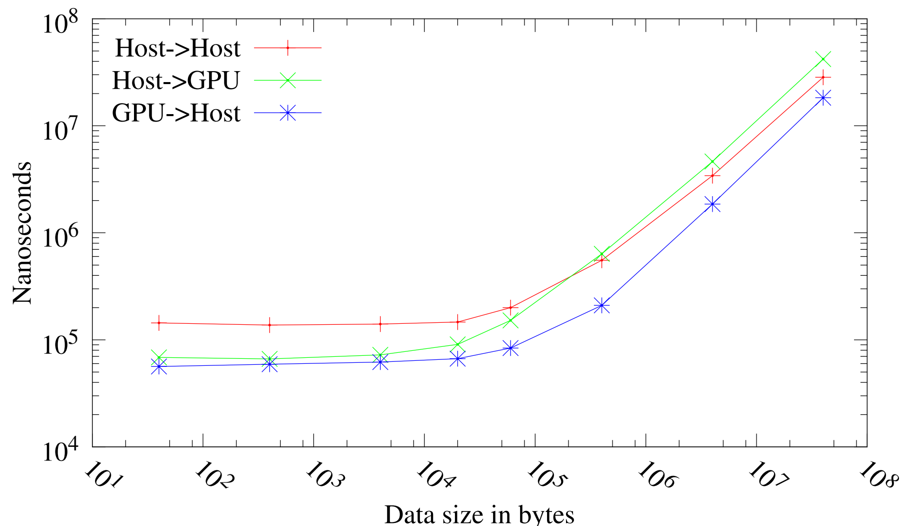


Figure 6.5.: Time consumption of data transfers between different memories

If measurements cannot be avoided, an evaluation of all implementations and processing units is ideally done only twice per implementation and accelerator. To preserve the results of such measurements for later use, the runtime system provides a database that stores the results of all measurements similar to related work [12].

Every type of cost in this database is identified by a unique string and is organized in an independent data set. For example, if we are interested in the past runtimes of the implementation `matmul_GPU` on `GPU0`, the database can be queried using the following variadic function:

```
history = dls_hist_get_history_va(3, "runtime", \
        "matmul_GPU", "GPU0");
```

Afterwards, the runtime for specific problem sizes can be retrieved from the resulting history object.

6.3. Memory management and the impact of data locality

As we saw in Chapter 2.1, dedicated accelerators usually possess an own physical memory as the access on the main memory, e.g., over PCI Express, is expensive or not possible at all and the own memory can be optimized for the specific use case. Hence, the data has to be transferred between main and device memory. In Figure 6.5, the time consumption for data transfers to and from the device memory of a NVIDIA GeForce GTX 275 GPU and for copying the data inside the main memory as comparison has been measured as function of data size per transfer. As we can see, the time consumption for transferring small amount of data is almost constant until a size of about 50,000 bytes. Afterwards, it increases linearly with the data size. An interesting fact is that the time consumption also depends on the transfer direction and is higher for transfers from main to device memory.

The result of similar experiment is shown in Figure 6.6. Instead of transfer times to and from local accelerators, data is transferred between host memories over a network connection using the RPC plugin introduced in Chapter 4.7 and compared with the time consumption of a data copy inside the host memory. Over the network connection, the data is transferred one time inside the local host over the loopback interface and one time to a remote host connected via an 100 Mbit/s Ethernet network. As we can see, copying

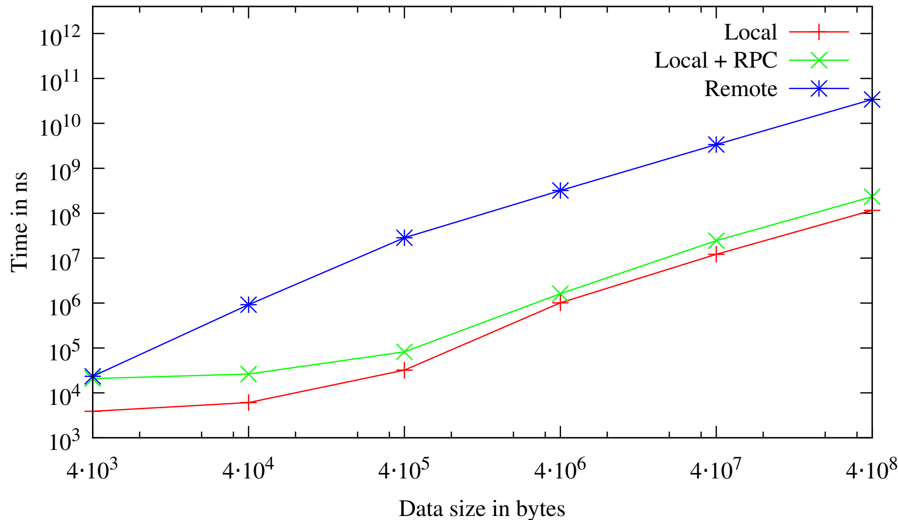


Figure 6.6.: Time consumption of data transfers in the same host and over a network

data over a network introduces considerable overhead. For example, copying 400 MB on the local system over a network connection introduces an overhead of 100 ms while copying to a remote host introduces an overhead of 34 s over a 100 Mbit/s Ethernet network.

For single kernel executions, data is transferred to device memory before the kernel execution and transferred back afterwards. Hence, when comparing performance between an implementation for the CPU and an accelerator, it is important to differentiate between raw execution time of the compute kernel and total time consumption including the data transfers. Only if multiple kernels are executed and, between, the data is not required on the CPU, data transfers between main and device memory can be avoided.

With the methods introduced by now in this work, the implementations are in charge of memory management and transfer data to the device before kernel execution and back afterwards. If multiple kernels are executed on the GPU this results in unnecessary overhead for data transfers. Therefore, an API of the runtime system is introduced that can be used to make the runtime system aware of the data. With this knowledge, the runtime system transfers data automatically between main and device memory on demand and also considers the required time for the transfer during selection of the fastest processing unit.

For each data block that the application registers the runtime system creates an own management structure. This structure contains necessary information like the start address and the size of the data in the host RAM. In order to keep track of the data copies in different memories, these structures are organized in linked lists. In Figure 6.7, examples of these structures is depicted. The data structures are organized per memory and the structures for every copy of the data is linked in a so-called siblings list. Each of the siblings also contains a version counter that is increased by one if a kernel writes data. Thus, when evaluating the possible destination of a migration, the runtime system can determine which of the siblings contains the most recent data and if a transfer is required.

The importance of efficient data placement has been motivated before. For example, on systems with a NUMA architecture, each core can access all memories but the access latency varies with the distance to the memory. Therefore, placing the data close to the core that will process the data is crucial for high performance and several projects have been introduced to automatically improve data and thread placement in such systems. To improve performance of OpenMP applications on NUMA architectures, Broquedis et al. introduce an extension of the GOMP OpenMP runtime system that combines dynamic

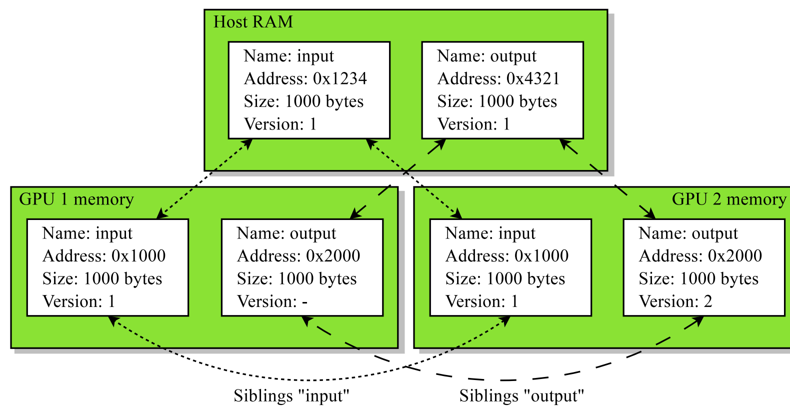


Figure 6.7.: Management structures for data in main and device memories

thread binding with efficient data placement on architectures with non-uniform memory access [26, 150]. With a synthetic benchmark, they show how their solution can increase memory bandwidth and decrease execution time using thread and data migration. In [137], Schmidl et al. introduce an extension to OpenMP that improves thread bindings for the execution of nested parallel regions on ccNUMA machines. With selected benchmarks and real-world simulations, they show that their mechanism increases the memory bandwidth and speed-up compared to unmodified execution with the OpenMP runtime. In contrast to this work, they only consider homogeneous systems with one address space spanning over multiple memories. In heterogeneous systems, efficient data management is more difficult as most heterogeneous systems have separate memory hierarchies and even if a software layer provides automatic data transfers, the latencies are much higher than accessing the memory in another NUMA domain. Furthermore, a thread cannot be simply bound to another core closer to the required data, as changing the processor usually requires different code.

But also for heterogeneous systems, different projects exist that simplify and automate data management [68]. A similar approach to parts of this work was presented by Becchi et al. [19]. They propose a runtime system that chooses either the CPU or GPU for execution based on past execution times and data location. However, in their work, they focus mostly on the memory management concept that is limited in contrast to this work. For example, they only consider a fixed set of memories and there is only one data block valid or all are synchronized. In this work, hardware plugins can add an arbitrary number of additional processing units and corresponding memories. Furthermore, there can be multiple valid and outdated data copies in different memories as the validity of each copy is determined using write counters. Also, although they claim that their approach is non-invasive with regard to the source code, they don't explain how this is achieved. In a further work [20], they describe an extension of this work that automatically detects out-of-band accesses to data blocks under control of the runtime system with the help of the operating system. They modify the page fault handling in a way that their runtime system is notified if an access occurs and it can update the data block status accordingly. In contrast to their previous work, they also describe that they use annotations of the kernel functions in order to determine the type of accesses in advance without changing the application source code.

Other projects use compiler analysis to automatically determine possible accesses to remote memories. Jablin et al. also introduced their work towards a similar goal [65]. Their compiler analyses the source code and data is automatically transferred between CPU and GPU with the help of a runtime system. In order to reduce transfers, they also identify short blocks of CPU calculations between GPU calls and transform this code into GPU functions to avoid the need for expensive transfers from GPU to CPU and back. Pai et al.

also introduced an approach to avoid the need for manual data transfers between host RAM and GPU memory [118]. Their approach uses a compiler-based code analyzer that inserts consistency checks into the source code which dynamically check if the data copy needed for following statements is up-to-date. They also compare their work with other approaches and show that their approach is able to avoid more redundant transfers. Compared to this work, the benefit of these approaches is that special compilers or kernels detect and handle accesses to remote memories. This work also automatically handles data transfers but the developer still has to signal following data accesses to the runtime system in advance.

So far, the presented approaches all work CPU centric. This means, the standard way to process data on a GPU involves several data transfers: first, data is transferred from an input device, e.g., an image sensor, into main memory, inside the main memory into a memory-mapped region for the GPU and from there into the actual device memory. As this significantly increases the latency, Kato et al. proposed a special approach that allows them to transfer data directly from input memory into the GPU memory by modifying the memory mapping of these PCIe devices to enable direct data transfers [73]. During experiments, they showed that they can achieve a latency of only several microseconds that allows them to control a fusion reactor using an algorithm executing on a GPU. A similar remarkable work is presented by Fujii et al. [51, 52]. Instead of reconfigure the memory mapping of the GPU, they reverse engineered the firmware of the GPU and modified the software executing on the management microcontrollers of the GPU to improve data transfers.

A project with similar concepts to this work is the Superglue project that also uses data versioning for dependency calculation [152]. Their task-oriented runtime system considers different access operations to determine if tasks depend on each other or if they are only mutual exclusive. In the latter case, tasks can be executed in arbitrary order but not at the same time. In contrast to this work, the project only targets homogeneous systems and does not consider dedicated memories or different implementations per task.

6.4. Reacting on competition for resources

A basic requirement for good decisions is the accuracy of the cost predictions. With the methods introduced in the past sections, the costs under normal conditions are determined. However, if other applications start to use shared resources, the kernel execution time will increase, predictions become incorrect and application runtime might increase even more due to unfortunate decisions. In the following, two approaches are presented that enable the runtime system to react on such events.

6.4.1. Passive checks

If the best processing unit for a task is affected by competition, the prediction automatically adapts as every execution of a task is measured and the average moves towards the new time consumption. Other task mappings are usually never executed except the fastest mapping suffers from, e.g., competing applications. Normally, this isn't a problem as the application is only interested in the fastest mapping. However, this might cause inefficient performance in case the fastest mapping becomes unusable or in case one of the alternatives was measured during a period of competition and therefore the runtime system afterwards wrongly believes it is slower than other mappings.

In order to circumvent such problems, the runtime system periodically checks the execution time of all mappings. But, if the time between those checks is chosen too short, the performance can decrease significantly. If it is too long, the probability of imprecise or outdated data in the database increases. In order to keep the overhead for checks low,

the runtime system considers the presumable loss of time and only schedules a check of an alternative mapping if the resulting overhead stays below a relative threshold. For every task mapping, the runtime system stores the accumulated execution time of other mappings that were preferred over this one for every execution since this mapping was executed the last time except for first time measurements or checks of other mappings. With this value, the runtime system can determine what a relative overhead a check of a combination would introduce. Given the set of all valid mappings A, the number of task executions $n(a)$ since the last execution of mapping a and the function $e(a, i)$ that returns 1 if the mapping a was executed as fastest mapping during the i-th task execution, else 0, the mapping c is checked if expression 6.2 is true and $t_{ignored}(c) > 0$:

$$t_{ignored}(c) = \sum_{i=0}^{n(c)} \sum_{a} t_{exec}(a) * e(a, i) \qquad a \in A \setminus c \tag{6.1}$$

$$\frac{t_{ignored}(c) + t_{exec}(c)}{t_{ignored}(c)} < threshold \tag{6.2}$$

The value of *threshold* can be set manually in order to reflect the system environment, e.g., it can be set to 1 in order to disable checks on systems where the application runs without competition.

To evaluate the impact of this threshold, an application with two task implementations has been used. The execution time of the first implementation is about 50 times larger than the execution time of the second implementation in this example. For this evaluation, the application executes the task 500 times during one application run. After each application run, the threshold has been increased and the resulting time consumption of the 500 task executions is shown in Figure 6.8. Besides the actual time consumption, the diagram also shows the function f that represents the execution time without checks multiplied by the threshold. By comparing these lines, one can see that the runtime system successfully keeps the overhead for checks below the given threshold. The size of the gap between them depends on the number of task executions and the time difference between the implementations.

In a further experiment, the behavior of the runtime system during periods of competition is evaluated. In Figure 6.9, the decision tree of the scheduler is visualized for an example that periodically calculates a multiplication of square matrices with either a sequential implementation on the CPU, a parallel implementation on the CPU or a parallel implementation on the GPU. The y-axis shows the range of problem sizes for which one of the three task implementations would be chosen and the x-axis shows how these ranges vary during the application runtime. In the beginning, the sequential implementation is the fastest for problem sizes between 1 and about 18 which equals matrices with a size of 1x1 until 18x18. The parallel CPU implementation is the fastest from about 18 until about 300. Afterwards, the GPU implementation is the fastest implementation for matrices with a size of up to 600x600 which is the largest size considered during this experiment. At time point A, a competing application is started that also uses the GPU. As we can see, the runtime system adapts its decision and would only choose the sequential or parallel CPU implementation until time point B when the competing application stops. Likewise, at time point C, a competing OpenMP application is started that occupies all CPU cores and the runtime system ignores the parallel CPU implementation as forking additional threads is not beneficial anymore. Instead, either the sequential or the GPU implementation is used until time point D.

Even during the periods without competition, the classification slightly varies over the time. This is caused by the natural variations of the execution time due to the scheduling

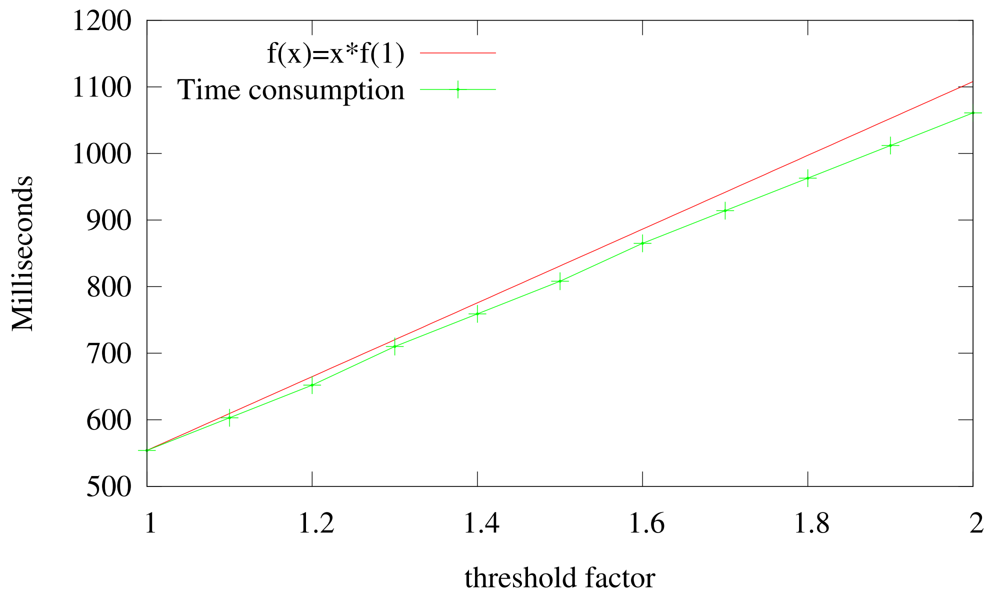


Figure 6.8.: Overhead caused by checks as function of threshold factor

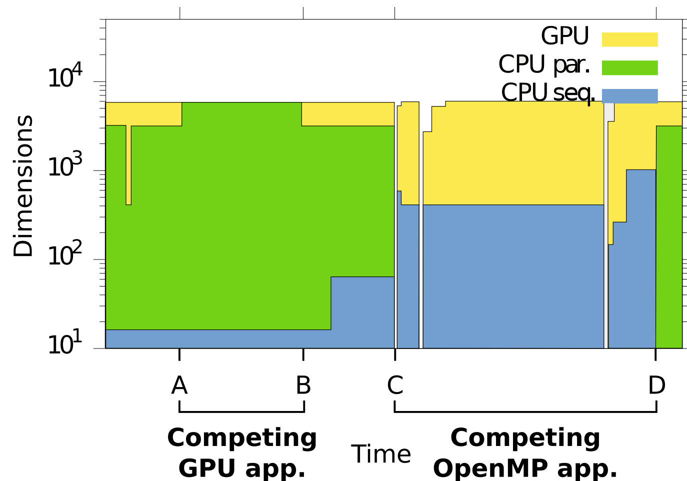


Figure 6.9.: Visualization of the decision tree during application runtime

of the operating system scheduler. Between time point C and D, also small gaps are visible that occur because the profiling data is not collected in time due to the high contention.

6.4.2. Shared-memory waiting queues

Queues are a common technique to organize a schedule on the CPU in a multi-tasking operating system. However, the schedulers of today's major operating systems are not involved in scheduling decisions on accelerators. If one or more users start applications that use an accelerator, the execution of the compute kernels is – depending on the device driver – either serialized or they have to share the device. In either case, a significant overhead may occur.

In the previous section, the runtime system detected such a case through the monitoring of the execution times. However, this method cannot detect competition in advance and a period of competition ends at an unknown time, thus the runtime system may either use a busy device for too long or unnecessarily avoid an idle device. To resolve this, maintaining own waiting queues in userspace has been proposed before [67]. In this work, the runtime

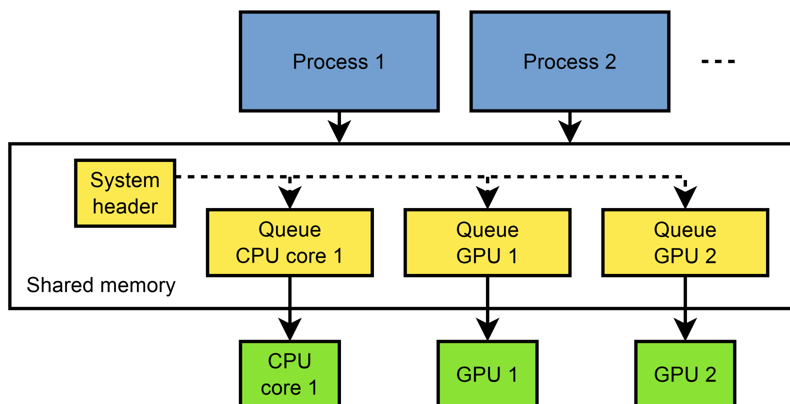


Figure 6.10.: Structure of waiting queues in shared memory

DLS	mutex_attr	mutex	#PUs
PU description 1			
⋮			
PU description N			

Figure 6.11.: Structure of the list of processing units in shared memory

system also provides queue structures in memory regions that can be shared between processes and that regulate the access on devices between cooperative applications. Every processing unit that the runtime system knows has an own queue and every instance of the runtime system can check, if another instance has scheduled a kernel on the respective device. With the entry in the queue, the runtime system also stores the presumable execution time, thus it can be determined in advance, how long one has to wait until the device can be used exclusively.

In Figure 6.10, an overview of this design is shown with an example. The first step to coordinate waiting queues in shared memory is to make all participating applications aware of the known processing units. As mentioned in Chapter 4.6, applications might not be aware of all processing units available in a system as they query the hardware on demand to avoid unnecessary initialization costs. Therefore, at first, the runtime system reads the so-called system header that contains a list of the currently known processing units. The detailed structure of this header is shown in Figure 6.11. The first field in this structure is the string `DLS` that indicates that the structure is already initialized. The next two fields are reserved for the mutex that protects the remaining fields: the number of known processing units followed by a list of their IDs.

For each queue a separate shared memory region is allocated that starts with a header structure as shown in Figure 6.12. The structure starts again with the string `DLS` followed by the name of the queue and the mutex that protects the remaining fields. `Head` points to the currently executing queue entry, `#used` gives the number of the entries that are currently in use, `max_entries` represents the number of entries currently allocated in the shared memory region and `start_time` indicates when the currently executing task started. This header is followed by the list of queue entries that each contains additional information about the corresponding task as indicated in Figure 6.13. The first two items, process ID and task ID are used to identify an entry. The third value states for how long the unit is presumably occupied. The fourth is a flag that indicates if the execution time is not known yet and a check will be performed. In such a case, other applications can either fallback to

DLS	Queue name				
mutex_attr	mutex	head	#used	max_entries	start_time
Queue entry 1					
⋮					
Queue entry N					

Figure 6.12.: Structure of a queue in shared memory

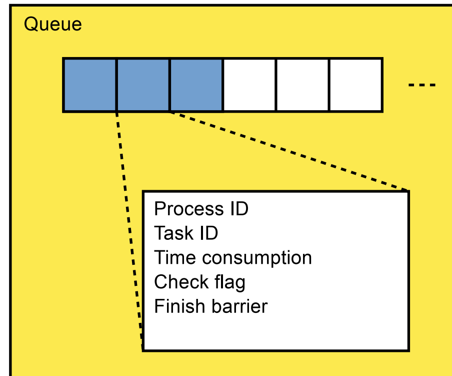


Figure 6.13.: Structure of a queue entry

a queue with a predictable waiting time or assume a default value as time consumption. The last entry is a barrier that can be used by threads that wait on the currently executing task to finish.

To give an example for the benefit of waiting queues in shared memory, this experiment uses an application implementing a Lattice Boltzmann method to simulate the flow inside a cube while moving one lid. The application is started two times in parallel on a machine with an Intel quad-core i7-3610QM CPU and an Nvidia NVS5400M GPU. The cube in this example has 100x100x100 points and is simulated for 20 time steps. For each time step, three compute tasks are necessary: `MoveTop` that simulates the movement of the lid, `CollideStream` that calculates the movements inside the grid and `StreamBoundary` that calculates the boundary conditions. In Figure 6.14, the Gantt diagram of the executed tasks with the runtime system that uses only per-process waiting queues is shown. As we can see, both processes compete for the GPU and only after the measured execution times exceeded a threshold, one of the processes switched to the CPU. In this example, the last task finished after approximately 12.3 seconds. With a naive approach that lacks a passive check mechanism, both would stay on the GPU and the overhead would increase further.

In contrast, the Gantt diagram of this experiment with enabled shared waiting queues is shown in Figure 6.15. As we can see, the process that started a short time later immediately switched to the CPU. In this case, the last task finished after about 8.1 seconds. Hence, by detecting competition in advance, a reduction of the execution time by over 30% was possible.

6.5. Impact of faults on costs

If the occurrence of faults is taken into consideration during task execution, they create two challenges for runtime systems: first, the faults must be detected and second, their impact on further decisions must be determined. A well-known approach to detect faults is redundant

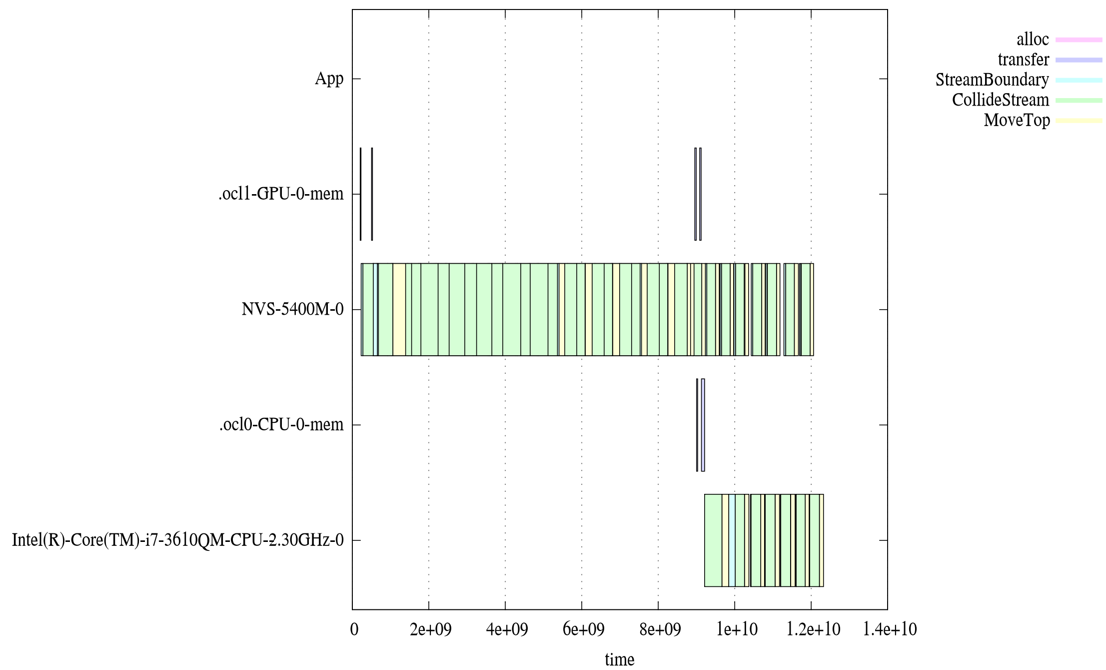


Figure 6.14.: Gantt diagram without shared waiting queues

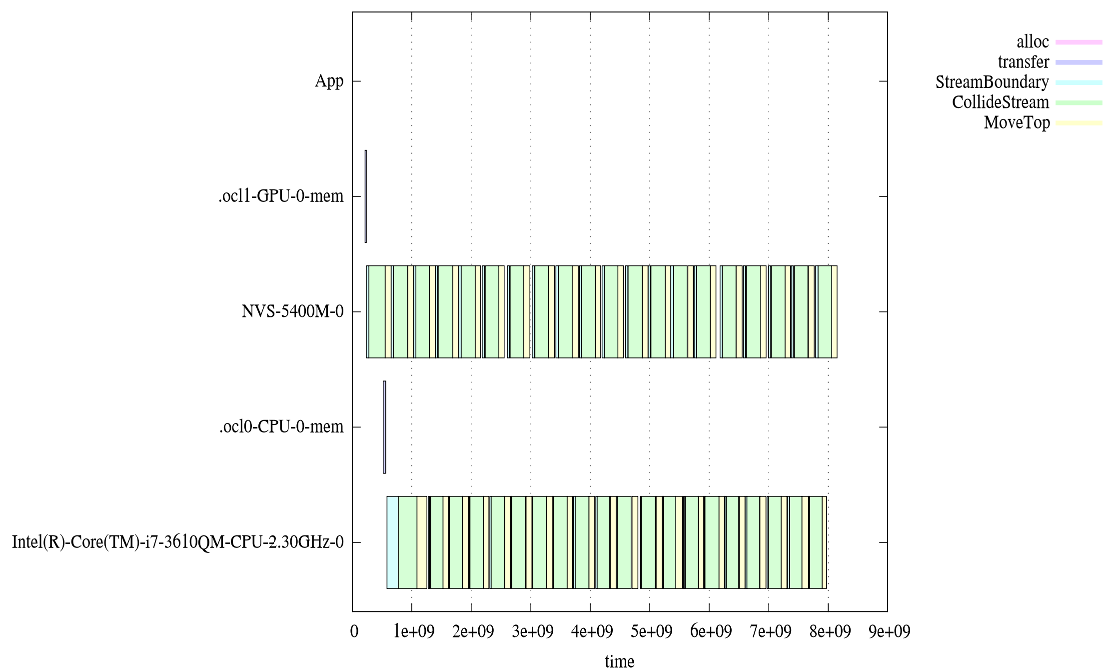


Figure 6.15.: Gantt diagram with shared waiting queues

execution. However, redundant execution has a high impact on the performance as a task has to be executed twice and the results have to be compared afterwards. Therefore, finding efficient methods to detect faults is an active topic in research. In the following Section 6.5.1, an adaptation of such an approach, called symptom-based fault detection, is introduced that leverages the methods presented in this chapter for fault detection.

As mentioned, the other challenge is determining the impact on further decisions in the runtime system. If a fault is not permanent, i.e., if faults occur only sporadic or for a limited amount of time on a processing unit, e.g., due to overheating, the processing unit can still calculate correct results and thus provide a benefit. However, a critical question is how to determine if a susceptible processing unit is still beneficial with regard to the other units available in a heterogeneous system. As faults cannot be predicted in advance, this work proposes a new metric in Section 6.5.2, called fault-aware runtime estimation, that enables the runtime system to estimate the averaged remaining benefit based on the observed fault rate and the fault-free runtime.

6.5.1. Symptom-based fault detection

Modern processor architectures provide a high number of hardware performance counters that enable performance analysis at a low cost. The events that are counted range from obvious events like executed instructions or cache misses to undocumented events that occur in the internal microarchitecture of the CPU [162]. Besides performance optimization and debugging [11], it has been shown that they are also useful for security, e.g., intrusion detection [43], or dependability, e.g., to detect anomalous execution caused by faults. In the latter case, performance counters are used directly in the hardware to enable fast detection and rollbacks [159, 123, 96, 111] or evaluated in software [166, 17, 46].

While these works focus on CPUs, this work also provides an initial evaluation of hardware performance counters on accelerator architectures and utilizes the suitable counters for symptom-based fault indication in addition to the redundancy-based methods described later in Chapter 7.3.4.

A common problem when using hardware performance counters are varying numbers between different runs of the very same application. Therefore, it is crucial to evaluate the behavior of the counters in order to determine if deviations caused by faults can be distinguished from natural fluctuation. For example, on `x86_64` architectures, the counters are incremented with every hardware interrupt [162]. As hardware interrupts can be measured as well, the inaccuracy can be reduced in this example but other counters might be influenced in an unknown manner. According to Weaver et al., other sources for inaccuracy can be instruction overcount [162]. However, as we only compare values on the same system, these inaccuracies do not vary between multiple runs.

To simplify the access to the hardware performance counters, the runtime system contains a sensor plugin for the PAPI library [2] that enables uniform access to counters on different architectures. Besides the counters for common CPU architectures, PAPI also provides access to GPU counters using a CUDA component that is based on the Nvidia CUDA Profiling Tools Interface (CUPTI). As the sensor infrastructure already stores observed costs and the average deviation for the scheduler, reusing this infrastructure enables symptom-based fault detection with only a small amount of additional code.

As mentioned earlier, the performance counters can vary although the executed instructions stays the same. The more the counters vary naturally, the more difficult it is to detect faults during execution that cause only small deviations. To determine the range of deviations,

[2] `http://icl.cs.utk.edu/papi/`

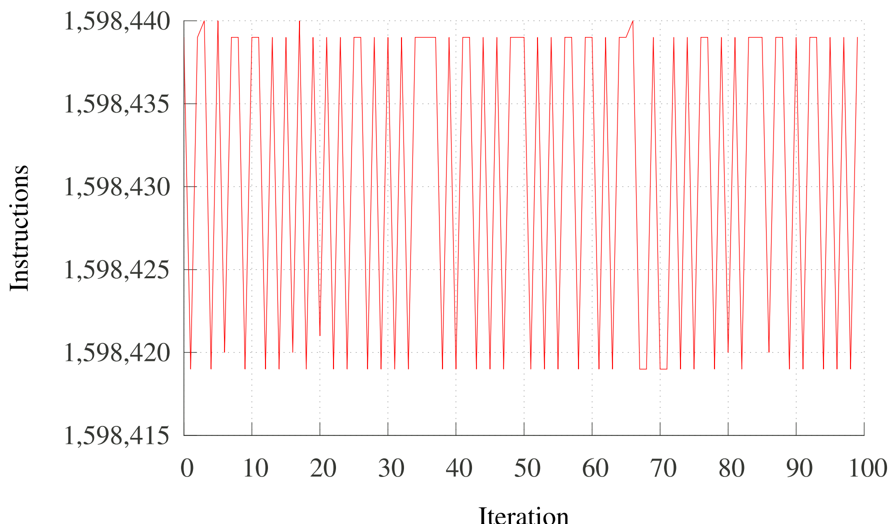


Figure 6.16.: Resulting number of counted instructions for single-threaded pathfinder

the pathfinder application of the Rodinia benchmark suite (with 1000 columns, 100 rows and a height of 20) and the total instruction counter have been chosen as example. The following experiments were performed on an eight core AMD Opteron 2378 with Shanghai architecture using the Ubuntu Linux 12.04.5 operating system with kernel version 3.2.0.

During early experiments, common functions like `printf` or `malloc` have been determined as sources for large sporadic deviations during runtime. After removing these disturbing sources from the measured area, the number of counted instructions vary only in a range of about 20 instructions for the single-threaded version as shown in Figure 6.16.

As PAPI does not handle OpenMP parallel regions automatically, additional effort was necessary to support the parallel version of the pathfinder benchmark. For each spawned thread, PAPI must be initialized individually. Hence, either instrumenting the parallel region is necessary or patching the OpenMP runtime library. As the latter is not possible with proprietary compilers, we added a function call to the runtime system at the beginning and at the end of the parallel region for the following results. In Figure 6.17, the initial results are shown. As we can see, the graph is largely dominated by a few peaks that add up to 50 million instructions.

As this behavior depends on the chosen OpenMP library and may even vary between versions, an independent approach would be beneficial. As all initialization routines like thread spawning are executed by the master thread, the numbers of the master thread have been left out which results in a graph as shown in Figure 6.18. As we can see, the peaks vanished and the deviations are reduced to a range of up to 3000 instructions. However, as the master thread also calculates a part of the results, it cannot be left aside without sacrificing fault coverage. As a trade-off, the runtime system also includes the numbers of the master thread but only from the part inside the parallel region. With this trade-off, the runtime system gets a complete view of the parallel region but loses information about the code surrounding the parallel region. However, as the parallel region is usually the most time-consuming part of an implementation and including the other part would make the numbers unusable, this trade-off is reasonable. The resulting numbers for the OpenMP version of the pathfinder benchmark are shown in Figure 6.19. As we can see, there are still peaks but the values vary only in a range of about 1500 instructions.

For the GPU, also the number of executed instructions is measured for the pathfinder benchmark. As the first GPU, a Nvidia GeForce GTX 275, has a compute capability below

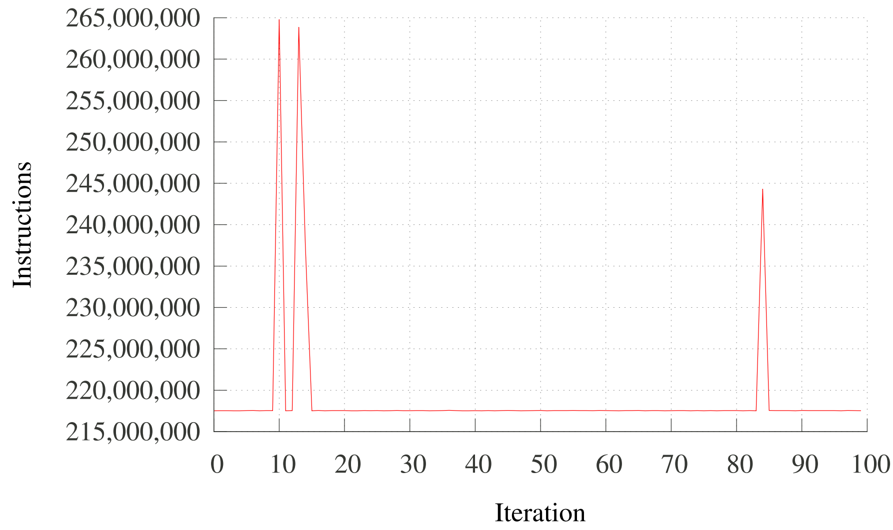


Figure 6.17.: Resulting number of counted instructions for OpenMP pathfinder

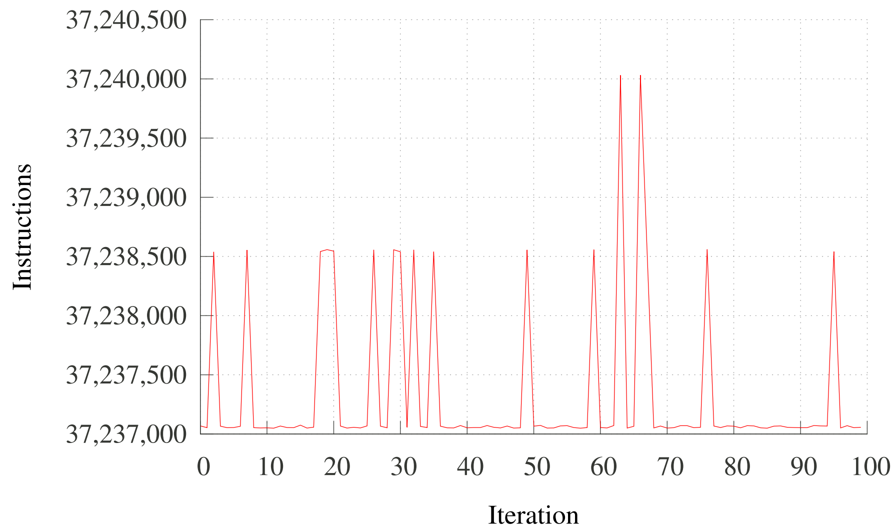


Figure 6.18.: Counted instructions for slave threads only

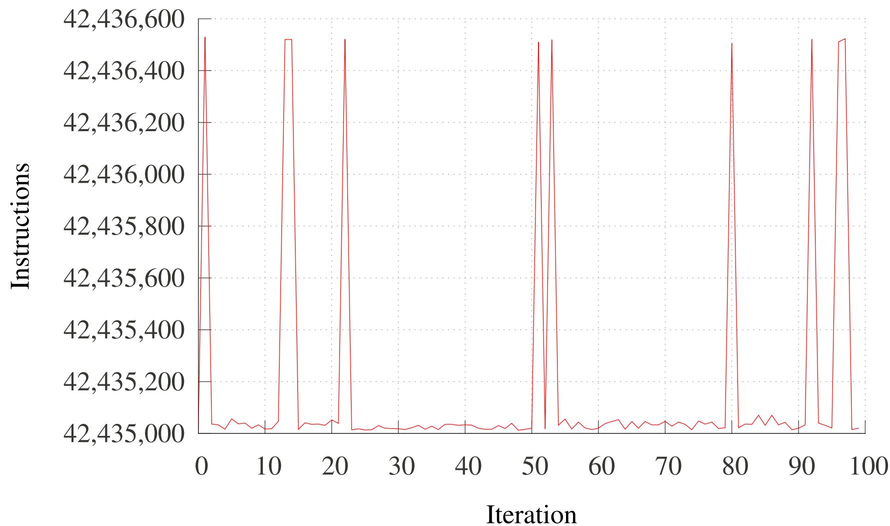


Figure 6.19.: Counted instructions for the parallel region only

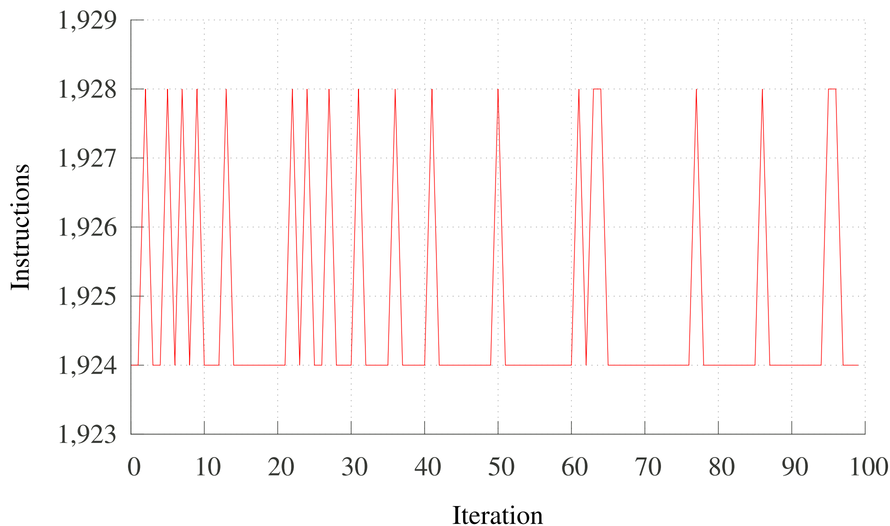


Figure 6.20.: Counted instructions for one shared multiprocessor on the GPU

version 2.0, only the `cuda:::GeForce_GTX_275:domain_b:instructions` counter could be used which represents the number of instructions executed by one shared multiprocessor on the GPU. Starting with compute capability 2.0, also the total number of instructions executed on the GPU is available according to the CUPTI user's guide in version 2. As we can see in Figure 6.20, the number only varies by four instructions but these numbers also only represent one of ten multiprocessors. On the second GPU, the GeForce GTX 560 Ti, the better suited counter `cuda:::GeForce_GTX_560_Ti:domain_d:inst_executed` is available. There, the instruction counter reported a constant value of 27104 executed instructions.

To analyze the accuracy of the symptom-based fault detection despite natural variations, the OpenMP compute kernel of the pathfinder benchmark has been instrumented to inject a specific fault under given circumstances. The OpenMP kernel consists of two nested loops where the inner loop is parallelized with OpenMP. With the instrumentation, the number of iterations executed by the inner OpenMP loop during the last iteration of the outer loop can be decreased by setting an environment variable. Thus, the number of executed

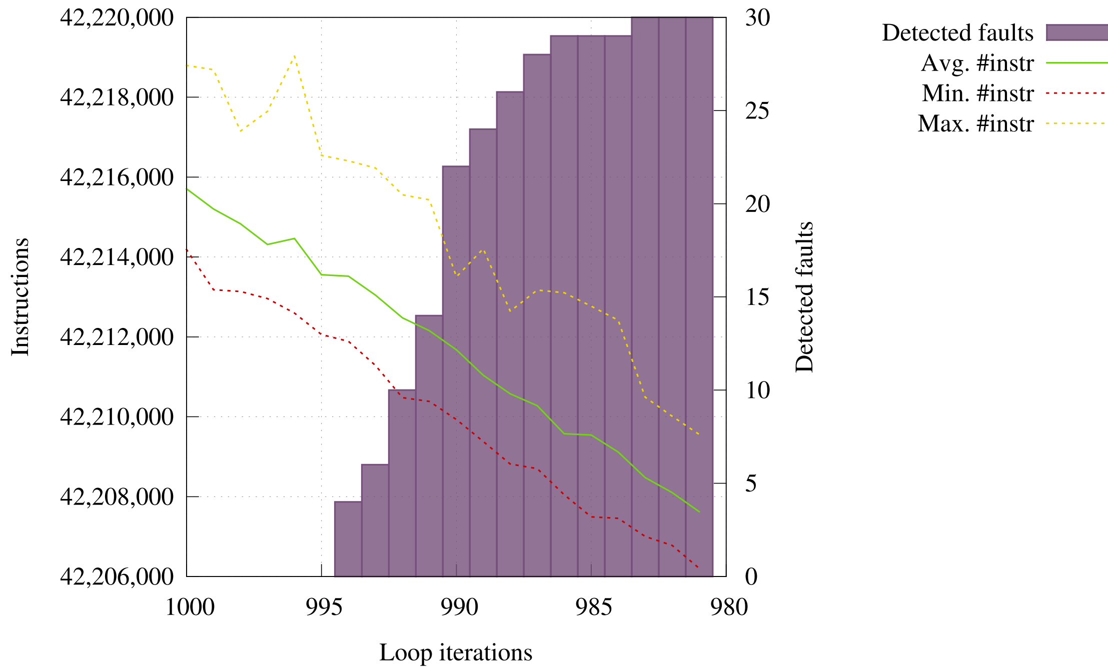


Figure 6.21.: Detected faults and counted instructions with a decreasing number of loop iterations

instructions can be decreased step-wise. In this experiment, the pathfinder benchmark is executed repeatedly and the number of iterations of the inner loop is reduced by one after every 30th application run. In the chosen configuration, the runtime system considers a run as suspicious if a counter exceeds a previously measured minimum or maximum value. In such a case, the runtime system repeats the execution with a backup copy of the input data and compares the results. If the results are the same, the runtime systems assumes a correct result and the measured values are added to the database. If not, the values are discarded. In Figure 6.21, the results of this experiment are shown. As we can see, the runtime system is only able to detect faults after the inner loop has been shortened by 6 iterations due to the comparatively high natural fluctuation of the counters. If we compare the minimum and maximum number of executed instructions during each batch of runs, we see that the first faults are detected after the minimum number of instructions decreases by roughly 2000 instructions and the last undetected faults disappear after the maximum value goes below the same threshold at 42,212,000 instructions. Therefore, the runtime system suspects a fault if the number of executed instructions decreases by roughly 2000 instructions in this example which is slightly higher than the peak values we saw in Figure 6.19.

For the next experiment on the second GPU, an `if` condition and an increment operation have been added to the kernel that increment one integer in the result during execution of one of the threads. As the instruction counter on the second GPU is free from deviations, an additional instruction is immediately visible. In our experiment, the number of executed instructions rises consequently from 27144 to 27159 whenever a fault was injected. Hence, even a fault that only causes a deviation of one instruction can be detected on the GPU.

As symptom-based fault detection has been introduced to lower to overhead for fault detection compared to other mechanisms, an important topic are the costs for querying the hardware performance counters. If we take a look at the average kernel runtimes with and without PAPI measurements in Figure 6.22, we can see that the measurements with PAPI add a considerable overhead. While the overhead for the CUDA version is

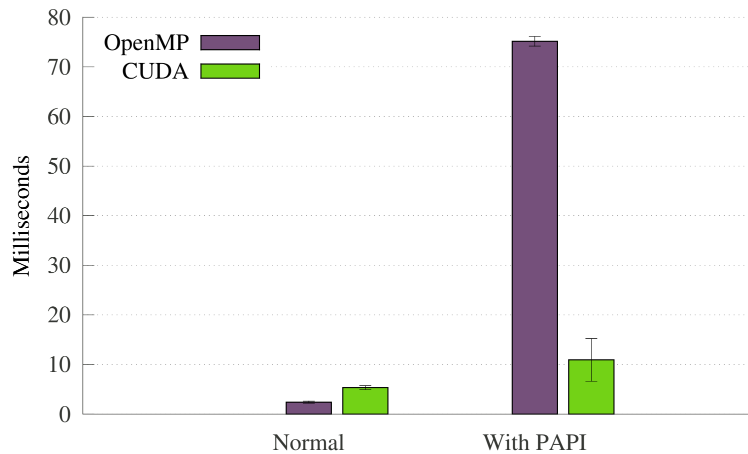


Figure 6.22.: Kernel runtime with and without PAPI measurements

comparatively moderate by doubling the execution time, the overhead for the OpenMP version is tremendous. In case of OpenMP, the overhead is caused by the frequent readout of the counters that happens during every iteration of the loop. With adequate support for profiling in the OpenMP library, a considerable reduction of the overhead should be possible. In addition, PAPI creates high startup costs. On the test system, simply initializing the PAPI library takes approximately 1.1 s. As PAPI is usually used for debugging and profiling during development, one can assume that it is not optimized for performance. Hence, for permanent profiling as part of symptom-based fault indication, either the performance of the library has to be improved or architecture-specific sensor plugins without the abstraction overhead of PAPI are necessary in future work.

To conclude, GPUs appear as promising architectures for symptom-based fault indication due to the noiseless counters while parallel execution degrades counter accuracy on the CPU. An interesting topic that is left for future work is the analysis of further counters and their combination to improve the indication – especially on the CPU to compensate the varying counters.

6.5.2. Fault-aware runtime estimation

The central topic in this chapter is to determine the single costs for executing a task on a specific processing unit. However, if a fault occurs during execution and it corrupts the results, the calculation has to be repeated. Hence, the actual cost for choosing the processing unit does not match the predicted costs which might cause an evitable loss of performance.

In a homogeneous system, a susceptible processing unit can simply be ignored as long as another idle unit is available as both usually provide equal performance. In heterogeneous systems, however, the processing units can exhibit a considerably different performance and switching to another unit might cause a higher loss than repeating the calculation. Therefore, avoiding a processing unit as soon as it calculated one wrong result is not advisable. But, sticking to a susceptible unit can also decrease performance if the calculation has to be repeated multiple times. Thus, another method is necessary to guide the decision if a valuable but susceptible processing unit is still beneficial compared to the other units.

This work proposes a new metric called fault-aware runtime estimation that enables the runtime system to estimate the remaining benefit of processing units compared to the other units in a system. The fault-aware runtime represents the fault-free execution time plus the

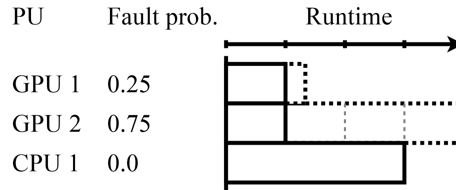


Figure 6.23.: Visual example for calculating the fault-aware runtime

average time required until the processing unit calculates a correct result. The fault-aware runtime F_i of the processing unit i with its fault probability $p_i \in [0, 1)$, the number of past valid runs v_i, total number of runs t_i and fault-free runtime R_i is defined as follows:

$$p_i = \frac{v_i}{t_i}$$

$$F_i = R_i * \frac{1}{1 - p_i}$$

A fault probability of 1 is handled like an infinite fault-aware runtime and the corresponding processing unit is only used in predefined check intervals to determine if it is still malfunctioning. A visual example for the metric is shown in Figure 6.23, where the solid boxes represent the fault-free runtime and the dashed boxes represent the fault-aware runtime for the given fault probability. In this example, GPU 2 is one of the fastest units but it is considered as the worst possible choice for task execution by the runtime system due to its fault probability of 0.75 which equals a fault-aware runtime that is four times higher as the fault-free runtime. The runtime system would choose GPU 1 for execution despite its fault probability of 0.25 as the calculation can be repeated up to three times before it requires more time than CPU 1.

An evaluation of this metric is presented later in Chapter 7.5.2.2 in combination with the other methods for fault detection.

7. Anticipatory scheduling in heterogeneous systems

The previous chapters described how usable implementations and processing units for a system can be determined and which single cost factors have to be considered. The remaining problem, this chapter tackles, is to combine this information and take the actual decision how an application and its tasks should be executed as illustrated in Figure 7.1. The first section introduces the fundamental concept for simulation of application execution on task level that enables a scheduler to evaluate the outcome of different decisions in advance. The second section then describes how different task mappings and other execution variants are implemented atop of this concept. Before presenting the results of the evaluation, section four will introduce the available mechanisms and schedulers for the actual decision making. Parts of this chapter are based on prior publications [76, 79, 77].

7.1. Introduction

Compared to common homogeneous systems, the scheduling in heterogeneous systems is considerably more difficult due to, for example, the disjoint memory hierarchies. If a task is mapped to a certain processing unit, additional data transfers into the unit's own memory might be necessary that delay the start of the task. Hence, decisions for

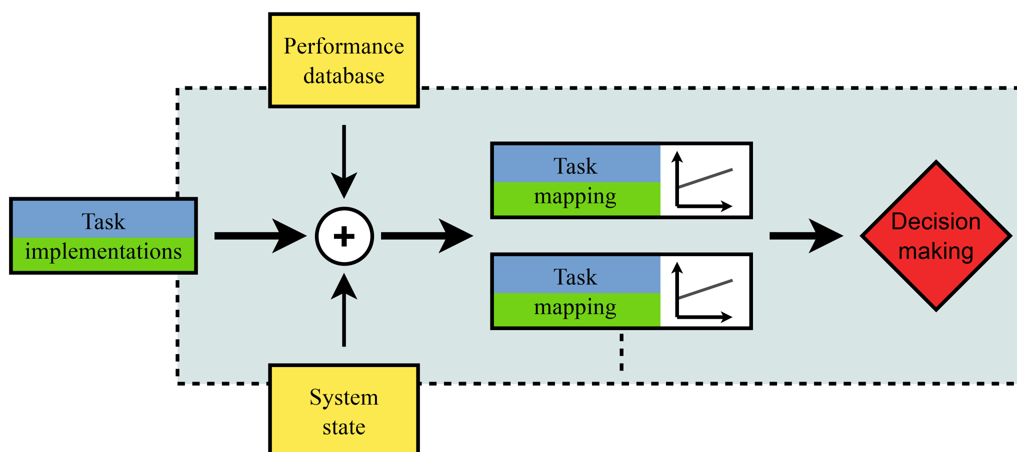


Figure 7.1.: Combining gathered information and initiating decision making

previous tasks determine the location of the most recent data and have a considerable impact on the best choice for the following tasks. With greedy algorithms like HEFT, that is preferred by related works, for example, tasks can be scheduled and executed one after another. The scheduler can simply analyze the current state of the memories and calculate necessary transfers for each possible mapping of a task before choosing and executing the best mapping. However, for more complex algorithms, multiple tasks might need to be scheduled at once, e.g., to evaluate a complete schedule of a task graph in advance. Hence, to determine the best mapping for a later task, the state of the memories before this task would be executed is necessary to make a prediction of required transfers for each task mapping. In such a case, the scheduler could either log intermediate data transfers and calculate the future state when required or, as in this work, the changes of the global state can be simulated during the scheduling in so-called containers. The benefit of simulation is that multiple schedules could be evaluated in parallel, that the latest state is always immediately known and the routines of the runtime system can be kept simple as they do not have to know if they modify the real global state or if they work on a hypothetical state inside a container.

Similar to the state of the memories, also the state of the waiting queues and the task graph itself can be modified inside a container. Especially the latter is important for the development of schedulers. For example, while task splitting can reduce the execution time of a task, it also introduces a management overhead, e.g., for splitting the data. Hence, if already enough task parallelism is available to utilize the system, task splitting might only decrease performance and determining such a case in advance can be difficult. Furthermore, if also additional objectives like reliability come into consideration, including all these execution variants and their implications on each other into the decision making results in increasingly complex schedulers.

Therefore, the runtime system in this work not only considers normal tasks, in the following referred to as *compute tasks*, but encapsulates every operation, that might be performed during execution and requires a considerable amount of time, into own special task types, e.g., for memory allocation or data transfers. Such tasks are in the following referred to as *supplementary tasks*. After a scheduling decision for a task has been made, the required supplementary tasks are also inserted into the task graph and the waiting queues as shown in the example in Figure 7.2. On the left, the submitted task graph with chosen implementations is shown and on the right resulting graph with the additional transfers between host and GPU RAM. Hence, based on the state of the waiting queues in the container, a scheduler can simply determine the resulting costs, e.g., the makespan, of the made decisions without having to know the purpose and implications of every task. Furthermore, before starting the execution, the entries in the waiting queues of the best container can simply be copied into the real queues without the need to recalculate necessary actions in the context of the global state.

In Figure 7.3, an overview of the execution planning process is shown. First, an application submits one or more tasks that the runtime system passes to the currently chosen scheduler. The scheduler can choose from different options to modify the tasks, e.g., duplicate a task for redundant execution, or to just map the tasks to a processing unit. It even can choose another existing scheduler to make a decision on its behalf. After a decision has been made for one or all tasks, the scheduler can then choose to directly execute the tasks or create further schedules with different decisions in a separate container, e.g., to compare different scheduling algorithms. If several schedules were created, they can be analyzed and compared before a voting algorithm of the scheduler determines the best according to chosen criteria, e.g., best performance. Afterwards, the container of the best schedule is then merged into the global state and executed.

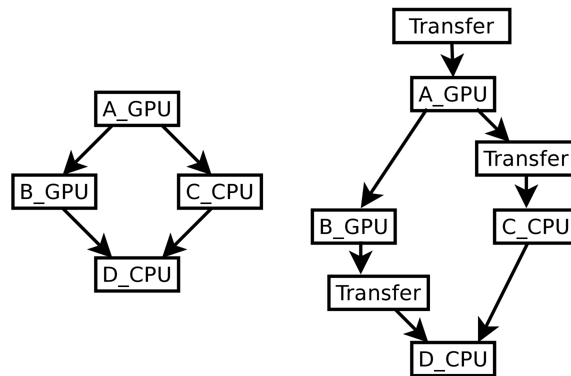


Figure 7.2.: Example of a graph with chosen implementations per compute task and the resulting graph with supplementary tasks

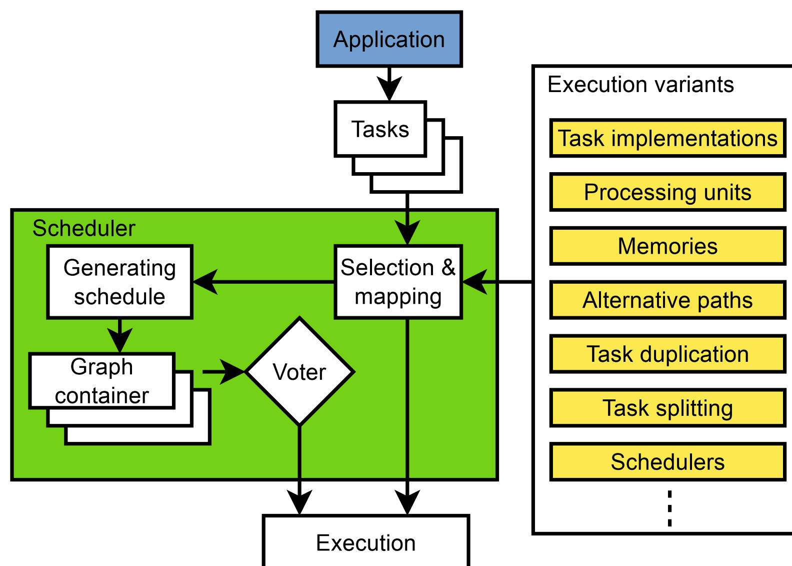


Figure 7.3.: Overview execution planning

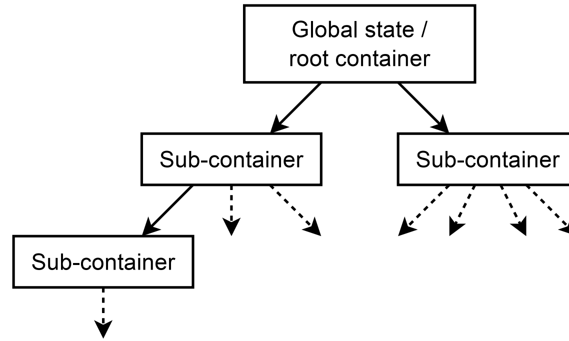


Figure 7.4.: Example of a container hierarchy

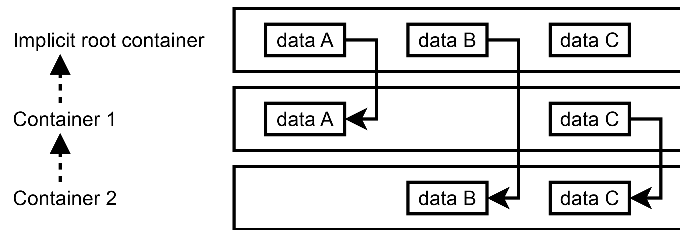


Figure 7.5.: Creating a modifiable copy of an object

7.2. Online simulation of task execution

In order to simulate task execution with different variations but without altering the global state, e.g., of the memory, waiting queues and the initial task graph, this work proposes the concept of so-called containers. Containers are organized in a hierarchy with the so-called root container, that represents the global state, at the top as shown in Figure 7.4. Every container can have an arbitrary number of sub-containers that each may contain new, removed or modified objects of its parent. While working in the context of a container, each operation is performed as if the previous alterations would represent the current global state and all changes that are made will also be only visible in the context of this container and its sub-containers. Hence, a scheduler can apply different modifications inside individual sub-containers and afterwards the resulting system states in each container can be compared. The container with the best result will be merged into the parent container and the other containers will be released.

New or deleted objects in a sub-container can simply be added or removed from the parent container if both are merged. However, if an object is modified, the modifications are performed on a copy of the original object. Due to the hierarchical organization of containers, the copy is created from either the original object or another copy in a superior container as depicted in Figure 7.5. In this example, container 1 is a sub-container of the root container with the original objects and container 2 is a sub-container of container 1. If container 1 modifies data object A and container 2 modifies data object B, both are copied from the root container. If container 2 modifies the data object C, it is copied from container 1 as container 1 already has a – potentially modified – copy of data object C.

While creating a copy is a comparatively simple operation, the opposite operation, merging an object into a superior container, is more difficult in this case. Objects usually contain multiple links to other objects that may have a different type, e.g., a transfer task links may link to its entry in a waiting queue, to other tasks it depends on and to the source and destination data blocks. In Figure 7.6, an example is given how a typical set of tasks, their queue entries and data objects are connected.

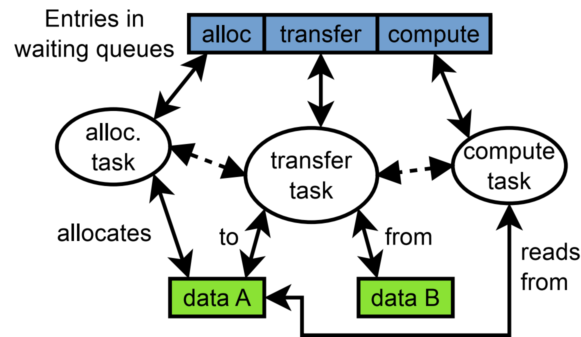


Figure 7.6.: Links between objects for a typical set of tasks

If an object is not present in the superior container, the object can simply be moved into the superior container and the links of other objects to this object are still valid as its address in memory does not change. However, if there is already a copy in the superior container, two approaches were implemented in this work. Following the initial approach, the object is merged into the object in the superior container. The runtime system had to compare the two objects, apply the changes to the superior object and update other objects that link to it as the inferior object will be released. The advantage of this approach is that the original object and its links stay valid but it requires additional efforts to determine and apply the changes as well as checking and updating all links.

Following the other approach, the inferior object simply replaces the object in the superior container. The benefit of this approach is that changes do not have to be applied again and the copy can be used as it is in the superior container. To avoid searching for links that have to be corrected, this approach is enhanced with automatic reference logging. Whenever an object references another, the address of the pointer in the source object is logged as well as the referenced object. Afterwards, if the source or the referenced object will be replaced, the runtime system simply walks through the list of references and replaces the address of the pointers with the address of the new object.

Besides the state of the memories, waiting queues and task graphs, also the progress of other types of objects can be simulated. For example, the runtime system also creates management structures for the available processing units in a system. Before mapping a compute task to a processing unit, it reads a flag in the management structure to check if the unit is already initialized. If not, an initialization task is created which the compute task depends on and the flag of the management structure is set in the current container. Hence, if the container is merged into the parent container, also the state of the processing unit is automatically updated and the initialization task is only created once. Similar to the initialization, also other properties of the processing unit can be of interest. For example, also the power state or the chosen frequency of the chip could be added to simulate if changing these properties would improve the performance and power consumption.

7.3. Building blocks for scheduling decisions

As on homogeneous systems, there are several generic parameters during application execution on heterogeneous systems that affect performance but do not alter the results of the computations. For example, the number of OpenMP threads can be varied or a different processing unit can be chosen for the execution of a task. Such variations exist on different levels and a few examples of possible generic variations are listed in Table 7.1. On task graph level, it might be beneficial to split a task among multiple processing units or use an alternative sequence of tasks that, e.g., target a different optimization goal. On

Task Graph	alternative sequence of tasks, task splitting, ...
↓	
Implementation	GPU thread block size, number of threads, ...
↓	
Hardware	Data location, Data transfer vs. mapping, ...

Table 7.1.: Examples for execution variations on different levels

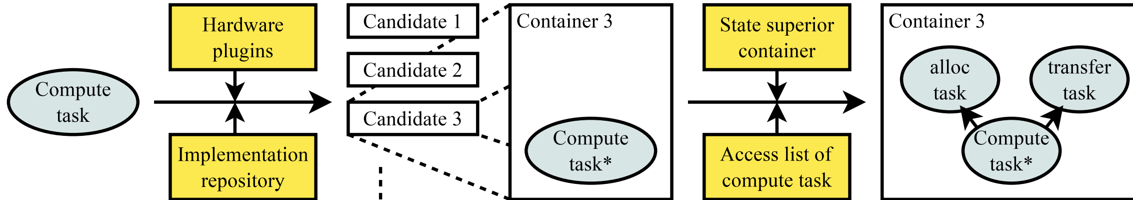


Figure 7.7.: Generating a candidate with container from a proxy task

implementation level, well-known parameters are the thread block size on GPUs, or the number of OpenMP threads on the CPU. On hardware level, it can be possible to choose between different memories that can hold input and output data or to choose different methods to make the data accessible like explicit data transfers to device memory or mapping the data from host RAM into device memory.

The following sections describe how the variations implemented in this initial work are mapped to the container concept and how they can be used by a scheduler as building blocks for decision making.

7.3.1. Task mapping

For each suitable combination of implementation, processing unit and memory, the runtime system creates a so-called mapping candidate as shown in Figure 7.7. Besides the chosen implementation, processing unit and memory for the data, each candidate also contains an own container. Depending on the state in the superior container and the list of data objects the compute task will access, the runtime system calculates the necessary supplementary tasks for this mapping.

In case the input data is not in an accessible memory by the processing unit, the data has to be transferred or memory mapped into the address space of the unit. In the best case, the data can be simply transferred from one memory into the other. However, if the accelerator shall be changed, e.g., from GPU to FPGA, direct transfers are usually not possible. In such a case, data has to be transferred into host RAM first. More intermediate transfers can become necessary if a remote accelerator shall be used as data has to be transferred into host memory, into the remote host memory and then into the memory of the remote accelerator. A visual example of these cases is shown in Figure 7.8.

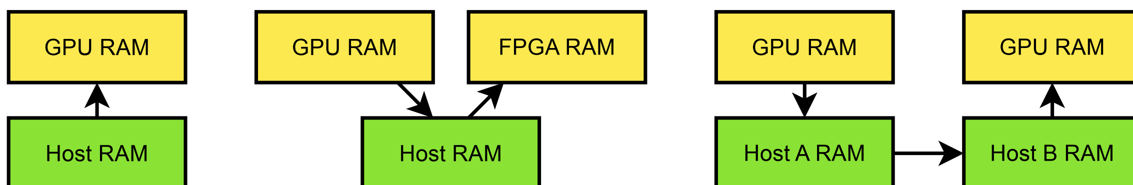


Figure 7.8.: Examples of necessary data transfers under different conditions

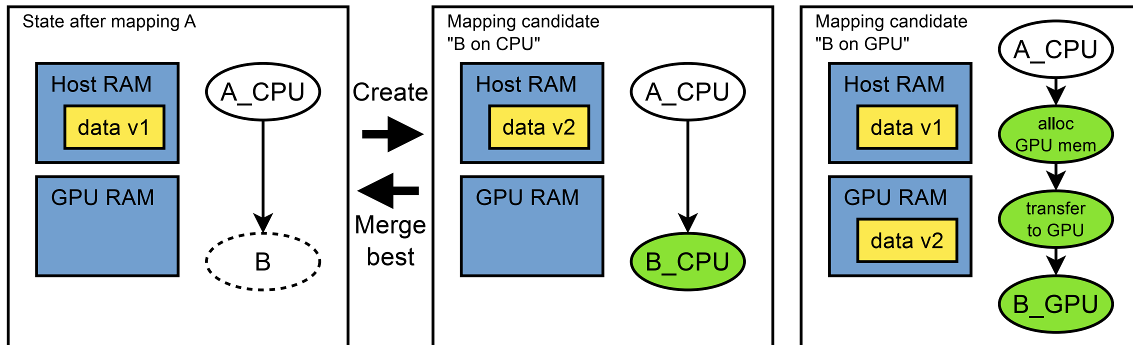


Figure 7.9.: Example of two candidates and their containers

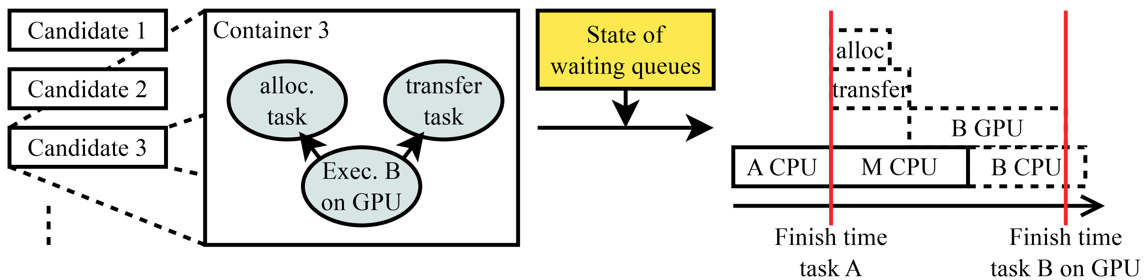


Figure 7.10.: Determining the final costs of a candidate

In Figure 7.9, an example of two candidate containers for task B that requires write access on the data is shown. On the left, we see the current state of the graph and the data after task A has been mapped to the CPU. On the right site, there are two candidates that map task B either to the CPU or to the GPU. If the CPU would be chosen for task B, the task can be started after task A has finished. If task B would be mapped to the GPU, two supplementary tasks are necessary. Memory has to be allocated in the GPU memory and the corresponding data structure has to be created.

To determine dependencies between tasks, each data structure contains a list of reading tasks and a link to the last writing task. If a task reads from the data, it will add itself to the reader list and add a dependency on the last writing task. If a task writes the data, it will add dependencies to all tasks in the reader list, so it won't write new data until all readers have finished, and then clear the reader list and register itself as the new last writer.

After this step, the runtime system knows the required actions for the input data and their individual costs. However, to determine the final costs, it is necessary to know when each action can be even started. Therefore the runtime system consults its internal waiting queues, registers the new tasks and calculates the resulting finish time of all tasks in the container as visualized in Figure 7.10. In this example, task B depends on task A that was mapped to the CPU. Although the single tasks for mapping task B on the GPU take more time than the execution on the CPU, the mapping on the CPU would result in a delayed finish time for task B as a parallel task M is already scheduled on the CPU. Hence, by comparing the finish time of the compute task in the candidate containers, a scheduler can simply determine the best mapping for this task without having to know how the task is executed.

7.3.2. Conditional task graphs

Similar to different implementations of a task, an application may contain alternative sequences of tasks that calculate the same end result but use a different algorithm designed

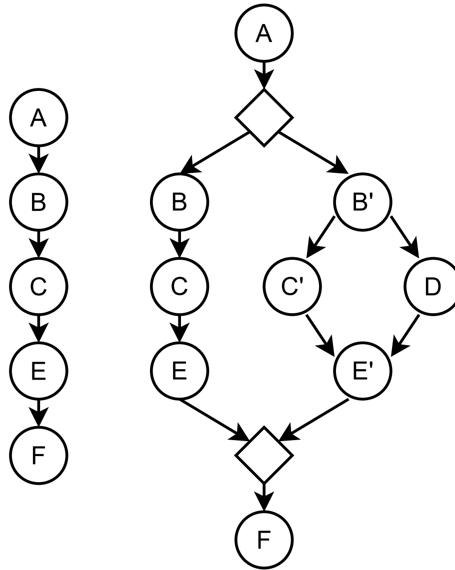


Figure 7.11.: Regular and conditional task graph with alternative sequences of tasks

for different situations like a different optimization goal, for example. With approaches proposed by related work, the developer would be responsible for evaluating and selecting the best sequence for the current system or situation. Instead, a developer can explicitly submit tasks in an alternative sequence with this work and the runtime system will use its mechanisms to determine the best sequence automatically under given circumstances.

In order to avoid duplicating the whole graph for each possible sequence, the runtime system uses so-called conditional task graphs. In a regular directed acyclic graph, every task of the graph is executed once. In a conditional task graph, there exist tasks that are mutually exclusive. The decision which of them will be executed can either be guided by an actual condition, e.g., if at least x bytes of memory are free, or be made dynamically by a scheduling algorithm, e.g., depending on the chosen optimization goal. In related work, conditional task graphs have only been used so far to model different sequences of tasks that may be executed during application execution which enables analysis of possible schedulings offline [99].

In Figure 7.11, an example of a normal task graph on the left side and a conditional graph on the right is given. In the graph on the right, one can either execute the tasks B, C and E or the tasks B', C', D and E' after the initial task A and before the final task F. In this example, the left sequence in the conditional graph can be beneficial in situations where only one processing unit is free and the right if there are two free processing units and they can execute task C' and D in parallel.

The actual design of such a conditional task graph in the code is shown in Figure 7.12. To simplify the algorithm for graph traversal, each task sequence has a special start and stop task. As we can see in this figure, the region with alternative sequences is replaced with a special task called *switch task*. In this task, each alternative sequence is managed as an own graph. This considerably simplifies code that operates on graphs and also enables nesting of graphs, e.g., to represent a case where alternative sequences contain multiple sequences themselves.

As the alternatives have to be self-contained, each one has an own container similar to the candidates in the previous section. In Figure 7.13, an example of a resulting container hierarchy is shown. On the top level, the container of the main task graph is shown. On the second level, we see the the container of one of the alternatives for the switch task. On

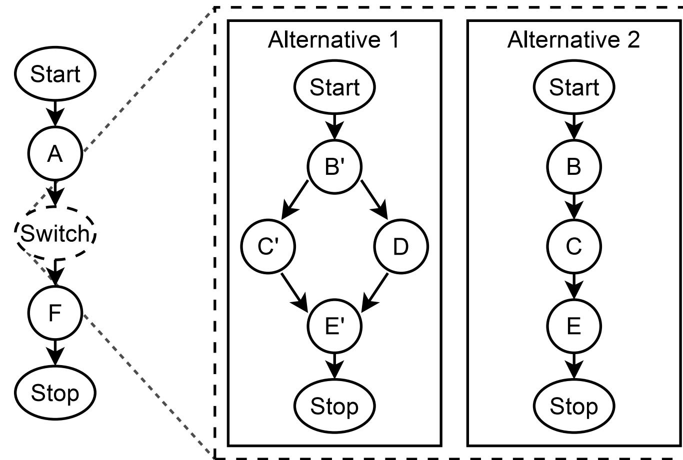


Figure 7.12.: Alternative sequences are encapsulated in special *switch* tasks

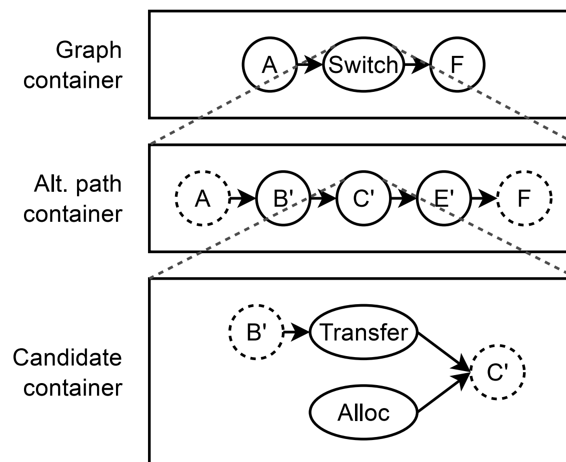


Figure 7.13.: Example of a container hierarchy

the bottom, one of the candidate containers is shown. In the second and third container, the tasks that are only virtual copies of the tasks in a superior container are shown with dashed circles. For example, in the candidate container, the task *B'* – that belongs to the path container – is required, as we need to modify its descendant list and replace the link to *C'* with a link to the transfer task that, in turn, links to *C'*. We cannot modify *B'* directly as a candidate that does not require a transfer keeps *B'* linked with *C'*.

Similar to the candidates, if a scheduling algorithm chooses an alternative path sequence, its container is merged into the superior container.

To submit a task, the runtime system provides the variadic function `dls_submit_task()`. An example is given in the following code:

```
dls_submit_task(matmul, "wrr", C, A, B);
```

As first argument this function expects the name or functionality of the compute task, e.g., `matmul`, and the second argument is a format string similar to the one of the `printf` function. Each character in this string represents one of the following arguments and determines the usage of the argument. In this example, `w` means the task requires write access on data object C and `r` means read access on the data objects A and B. Other example are `v` that states that the parameter shall be passed unmodified to the kernel, `p` that determines the problem size or `a` that appends this task to the path of the given task as we will see below.

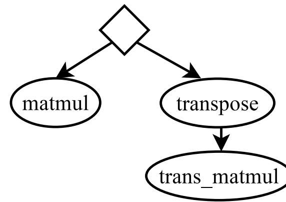


Figure 7.14.: Resulting graph of the given example

The submitted tasks are first organized in a linked list that represents the sequential order of execution. From this ordering, the runtime system calculates the dependencies between the tasks using the given data access modes.

If a developer wants to submit conditional tasks, he has to create the so-called switch task first. To start a new alternative path, the first task of each path has to be appended to the switch task using the `a` format specifier. As an example, a matrix multiplication can be either calculated using a regular multiplication or by transposing the second matrix and using a special multiplication function to improve cache utilization in the following code:

```
switch_task = dls_task_create_switch();

// path 1
mm_task = dls_submit_task(matmul, "awrr", switch_task, C, A, B);

// path 2
transp_task = dls_submit_task(transpose, "awr", switch_task, BT, B);
mm_task = dls_submit_task(trans_matmul, "awrr", \
        transp_task, C, A, BT);
```

As we can see, the `matmul` and the `transpose` task are appended to the switch task and each form the beginning of a new alternative path. In contrast, the second task in the transpose path `trans_matmul` is appended to the transpose task. The resulting graph of this example is shown in Figure 7.14.

7.3.3. Task splitting

To lower the application runtime, a high task throughput is necessary. Therefore, a scheduling algorithm tries to distribute tasks to all available processing units in parallel. If there is not enough task parallelism to employ all processing units, the system remains underutilized and an application cannot reach maximum performance. As most tasks that benefit from heterogeneous systems are inherently parallel, one approach to increase task parallelism is task splitting [85, 80, 133, 101, 92, 119, 70].

Common approaches for programming heterogeneous systems, e.g., OpenCL, simplify task splitting due to their programming models. With such models, a developer writes a compute kernel that processes one work item and submits the kernel and the number of work items to the respective runtime system. In turn, the corresponding runtime system spawns threads that process each work item, e.g., one thread per core on the CPU or one thread per work item on the GPU. Usually, the compute kernels are written in a way that one thread does not depend on other threads or only on neighboring threads in a work group. Therefore, single threads or groups of threads can be spawned on different processing units as visualized in Figure 7.15 where three work items are distributed among a CPU and a GPU. If, however, these processing units possess dedicated memories or exhibit different latencies to memories in the same address space, e.g., as in heterogeneous systems or on NUMA architectures, not only the threads but also the data has to be

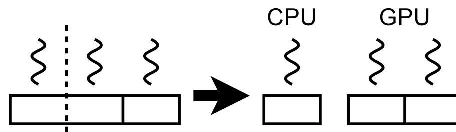


Figure 7.15.: Splitting work and data for parallel execution on CPU and GPU

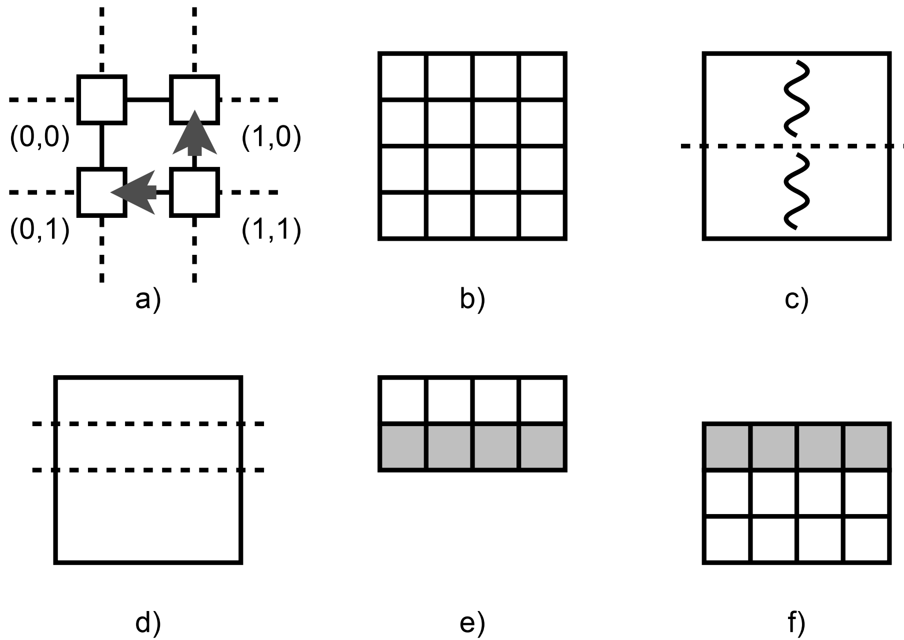


Figure 7.16.: Splitting work items and data for a grid where each point requires data from the left and top neighbor

distributed among participating processing units and memories to permit task splitting and to increase performance.

In the simplest case, each work item reads one element from one or more input arrays and writes one element into one or more output arrays. Here, the data can be split equally to the threads. However, if a thread requires other data, e.g., of neighboring points in a grid for stencil operations or simulations, partitioning the data becomes difficult. In Figure 7.16, we give an example where the data of the left and top neighbor in a grid is required as shown in sub-figure 7.16a). If we consider the 4x4 grid from sub-figure 7.16b) and want to calculate this grid using two threads, we can split the work load into two equally sized blocks as shown in sub-figure 7.16c). However, as we need the data in row two to calculate the data in row three, row two is required by both threads. Therefore, we get two cut lines as shown in sub-figure 7.16d) and thread one requires the data of row one and two, while thread two requires row two, three and four, as pictured in sub-figure 7.16e) and 7.16f). If the two threads run on devices with different memories, row two will be present in two memories. If this grid is calculated multiple times, e.g., for a simulation, data consistency becomes a critical topic. As thread one will calculate the new values for row two, the row has to be copied to memory two after each timestep.

In the following, the next section shows how work items and threads can be partitioned between multiple OpenCL devices. Afterwards, the fine-grained data management mechanism of the runtime system is introduced that also enables data partitioning and automatic synchronization across multiple devices. In the last section, the algorithm to determine a balanced partitioning is explained.

7.3.3.1. Thread partitioning with OpenCL

To start an OpenCL computation, a developer has to state the amount of threads that are required to calculate the total amount of work. To split the amount of work, e.g., in two parts, one could just start the computation twice with half the threads each time. However, this only works if the kernel source code is prepared for this case. In many cases, e.g., simulations, boundary conditions have to be considered. To determine if a boundary condition has to be considered, the threads check their assigned thread index that is used to calculate to corresponding position in a grid, for example. Without prepared code, these conditions are not determined correctly. For example, if a grid is split in two parts, a thread with thread index zero will be executed for each half while only one thread zero would be executed if only one computation would be started.

As this works strives for automatic task splitting, one approach could be automatic source code transformations. However, a simpler option is possible due to the OpenCL API. In addition to the amount of threads, an offset can be passed to the OpenCL library that sets the initial thread index. So, in the given example, the computation of the second half can be started with the corresponding offset and any thread index is only assigned once.

7.3.3.2. Fine-grained data management

With the method in the previous section, threads can be automatically distributed among the available OpenCL devices. If the involved devices share an address space, the remaining step is to efficiently balance the workload [70]. If not, the data has to be distributed before as well.

With most programming models, input and output data is allocated and transferred en bloc. Hence, it is unknown to the runtime libraries, which values in the data blocks a thread will access exactly. However, this information is necessary to determine which part of the data is required on each OpenCL device. To solve this issue, related work either copy all data into the device memories [119] or they depend on additional work of the developer [101], code instrumentation [80] or code analysis [85, 133, 92] to determine the access patterns of the threads. An important unanswered question for such automatic approaches is their versatility. For example, an approach based on code instrumentation may fail to detect a pattern that changes with the input data and that has not been experienced or trained before. An approach based on code analysis may not be able to correctly determine all accesses if they depend on input parameters as well, e.g., if they cause a varying number of loop iterations or they influence the array offset that will be read or written. Also, if the data is simply copied to all devices – which causes additional time overhead – the following problem is to determine which parts of the different data copies contain the actual results as the distributed results have to be merged back into the original data area before returning them to the application. Pandit et al. propose to keep a copy of the original data and compare it with the distributed results to detect new data [119]. However, this also creates additional overhead. In this initial work, the runtime system also depends on the developer to state the access patterns of individual threads as the other approaches either cause additional overhead or may not work in all cases.

In Chapter 6.3, the basic data management of the runtime system was introduced. This basic mechanism only works with complete copies of the original data. To split the data, a developer could use this mechanism nonetheless and simply manually divide the data. As the runtime system handles each data block individually, all necessary parts of the data will be present in the required memory. However, the data subsets might not be placed in contiguous memory locations as every subset is allocated individually. This is necessary, though, as the threads usually address the data using their continuous thread index as offset and they do not support gaps between the data. Therefore, an extended version of

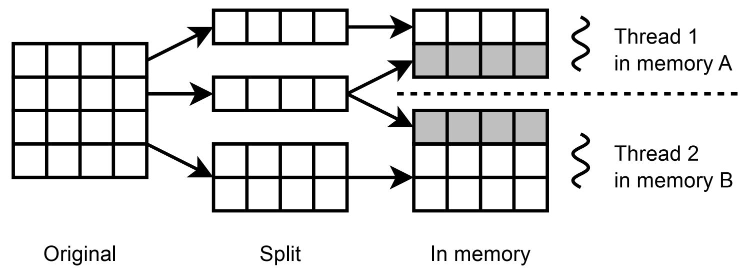


Figure 7.17.: Original data, split data and resulting data for two threads

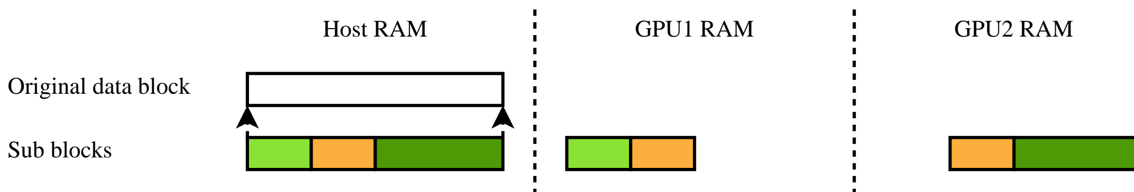


Figure 7.18.: Memory state example after subdivision

the data management is required that supports dividing data into smaller pieces but keeps the necessary parts at contiguous addresses in the memory.

If we reconsider the example in Figure 7.15, it is necessary to split the original data in three parts as shown in Figure 7.17. Row one is only required in memory A, row three and four only in memory B and row two is present in both memories. Therefore, the runtime system enables internal subdivision of data blocks into smaller parts that form own logically independent blocks themselves for the runtime system. If we split the given example in two parts and map the parts to two GPUs, the data would be split as shown in Figure 7.18. All rows are present in the host RAM, row one and two are present in GPU1 RAM and row two, three and four in GPU2 RAM.

As mentioned, the downside of the independence of each sub-block is that there is no guarantee that two subsequent memory allocations receive contiguous addresses. Due to the relative addressing, however, it is necessary that logical neighbors also lie in the spatial neighborhood as in the original memory. Therefore, the runtime system uses so-called intermediate blocks that span the size of the underlaying sub-blocks and are used for allocating and representing the memory area used for the sub-blocks as shown in Figure 7.19 with the dotted boxes. After the intermediate blocks are allocated in destination memory, the sub-blocks can be transferred independently but they share the continuous memory region of the intermediate block.

The downside of OpenCL for this approach is that it does not allow modifications of the

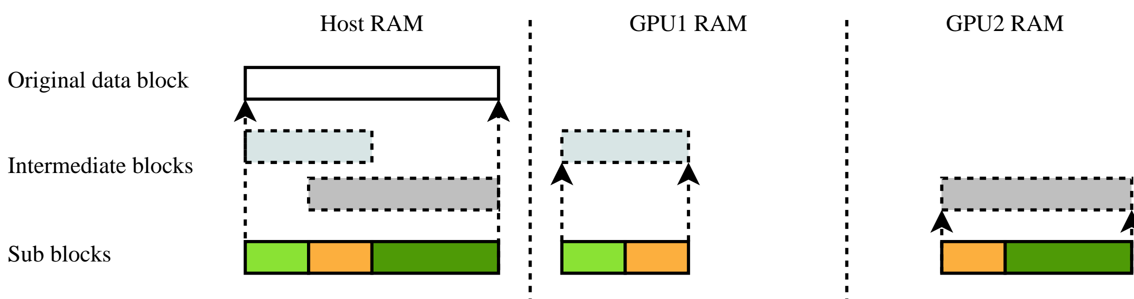


Figure 7.19.: Memory state example with intermediate blocks for allocation

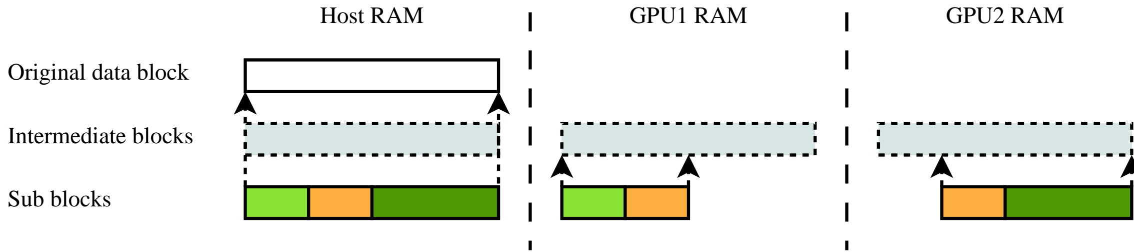


Figure 7.20.: Intermediate blocks for OpenCL kernels

data pointers passed to the kernels. The pointers may only contain the address returned by the allocation function and may not point to an address inside the allocated region. Therefore, adjusting the address to the offset of each device is not possible. This is a problem as kernels usually access the data relatively to their assigned thread index – passing only a part of the original data to kernel would result in incorrect data access. For example, if thread number 2 wants to access the grid data at index 32 and it receives the address `0x1010` to its part of the data starting at offset 16, it would consequently access the data at index `0x1010+32=0x1030` although index 32 starts at address `0x1000+32=0x1020`. For programming models that permit pointer modifications the runtime system can simply adjust the pointer by subtracting the corresponding offset from the starting address of the partial data so the kernel accesses the correct data nonetheless. In the given example, the runtime system would pass the address `0x1000` although the allocated block starts at `0x1010` and the thread would access the correct data at offset `0x1000+32=0x1020` without causing an incorrect access. However, the tested OpenCL implementation do not return the actual address in the device memory but a pointer to a management structure that is replaced with the correct address before kernel execution. Therefore, the procedure of the runtime system for OpenCL kernels slightly differs. As we can see in Figure 7.20, the runtime system allocates intermediate blocks in destination memory that have the size of the original data block in host RAM but it transfers only the required data to the respective position. This approach enables task splitting despite unmodifiable data pointers at the expense of increased memory usage.

7.3.3.3. Reaching equilibrium

To maximize the benefit of task splitting, the assigned work to each device should be chosen in a way that all devices finish their work at the same time. For heterogeneous systems, this is difficult as considering only the execution time of each device is not sufficient. E.g., due to the different memories, additional tasks for memory allocation and data transfers might be necessary. To make things worse, the time consumption of each task varies with the amount of work and data assigned to each device. While it can be assumed that the time consumption rises with the amount of work and data, the time consumption per problem size can only be roughly approximated and this approximation does not account special effects, e.g., caused by caches. Therefore, this work uses an iterative approach to determine the work portions resulting in an equilibrium of runtimes. This iterative approach is possible due to the container concept that enables an evaluation of different work distributions in advance including necessary supplementary tasks like data transfers.

To determine the best partitioning, related work uses machine learning [85], decision trees [92] or iterative algorithms [133] as in this work. Also, Pandit et al. chose an approach closer to typical load balancing [119]. Instead of calculating a distribution in advance, they start executing the single parts on a CPU and a GPU from each end of the workload in parallel. With proper synchronization, each processing unit continues processing the next part until the next part is already taken by the other processing unit. While this approach

avoids the overhead of calculating a balanced distribution, it is only usable for distributing data among two processing units.

In this work, the initial distribution is calculated using the time consumption for executing the complete task on each device. Based on the time consumption, the algorithm calculates the ratio of runtime per problem size and assigns the work accordingly. Afterwards, these steps are repeated with the time consumption for calculating the assigned work load.

To enable this approach, further problems have to be solved. One problem is that most OpenCL implementations and devices do not support arbitrary number of work items but expect a multiple of the local work group size. But even working with a multiple of usual work group sizes becomes problematic. Work group sizes are usually multiple of 8 in the range from 8 to 64. Especially for simulations with a large grid, the problem sizes can become quite large. Consequently, the intermediate result of an iteration can easily result in an unprofiled distribution. To avoid unnecessary evaluations of distributions with small variations, the algorithm starts with work sizes for each device that are a multiple of the required work group size and not smaller than 1/100 of the original problem size, if possible. The algorithm stops if a distribution has been found where to runtimes do not differ more than 50 µs. If the maximum difference does not decrease for two iterations, the algorithm decreases the granularity by factor 2. This loop is interrupted if no acceptable distribution is found after 16 iterations.

7.3.4. Taking precautions against faults

In order to detect and tolerate faults during compute kernel execution, special precautions have to be taken. The first challenge is to detect faults during execution. A well-known approach to detect faults is redundant execution. However, to employ redundancy in heterogeneous systems, different circumstances have to be considered as this work describes in Chapter 7.3.4.2. As redundant execution inevitably increases the costs in terms of execution time or energy consumption, approaches with less overhead are topic of current research but in most cases they trade the overhead with limited fault coverage. In Chapter 6.5.1, one such method to detect faults has already been introduced. In the following Chapter 7.3.4.1, another light-weight method to detect certain types of faults is described.

To tolerate a fault after it has been detected, this work repeats the calculation until a presumably correct result can be determined. Hence, in order to repeat calculations without altering the global state, a task implementation may only access data that is registered and preserved by the runtime system and must not access global variables or data in the file system, for example. In order to be considered as correct, the same result has to be calculated by at least two redundant executions. However, a system administrator can specify more strict requirements, e.g., demand at least x redundant executions and that the majority of the runs calculate the same result.

7.3.4.1. Light-weight fault tolerance

If a runtime system does not take special precautions before executing a compute kernel, a fault that occurs during kernel execution might lead to an inevitable abort of the application. For example, if a data block is used for input and output and a fault occurs during execution, the calculation cannot be restarted as the input data is in an inconsistent state and therefore lost. Or if a fault corrupts pointers and leads to a segmentation fault, the whole application is aborted.

In order to be able to recover from faults, the runtime system offers an operation mode for light-weight fault tolerance. In this mode, the data management of the runtime system

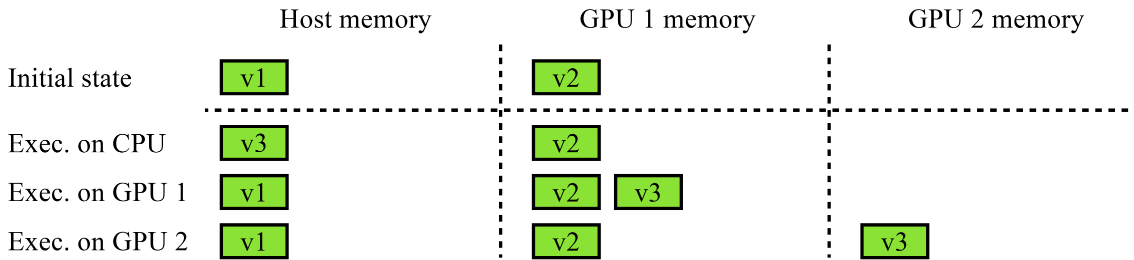


Figure 7.21.: Creating backup data on demand to enable rollbacks

slightly differs. Before execution, the runtime system checks the versions of the input data in the different memories. If there is only one data block with the most recent version, the data is duplicated to enable a rollback after a fault. In Figure 7.21, an example of this process is shown. Initially, the original data (v1) has been transferred to GPU 1 that modified the data (v2). Afterwards, if the second task is executed on the CPU, the data is transferred back and the CPU can use the original memory block to modify the data (v3) as the task's input data (v2) is still available in the memory of GPU 1. If a fault occurs on the CPU, the data can be transferred from the GPU memory again. If the second task is executed on GPU 1, the data has to be duplicated in the GPU 1 memory in order to preserve a backup of the data. In case the memory of GPU 1 is too small, an alternative would be to transfer the data back into host memory. If the second task is executed on GPU 2, a new data block is allocated in GPU 2 memory. Similar to the first case, it is not necessary to duplicate the data, as the input data is still available in GPU 1 memory.

For the task execution, the runtime system uses a separate thread from its thread pool and installs an own signal handler to catch segmentation faults. If a segmentation fault occurs, only the thread is aborted and the runtime system can restart execution on another processing unit. However, this approach works under the assumption that the thread did not modify other data except the data it has been assigned. If a fault caused corrupted pointers, it is possible that a thread has modified data at arbitrary addresses before an access to an unmapped address caused the resulting segmentation fault. Therefore, if the thread modified other data, using an own segmentation fault handler might only postpone the abortion of the application in some cases.

If a separate thread is executing a task, the original thread usually waits until the the task thread has finished. However, as the runtime system knows the usual execution time of the task, it can estimate if the thread is overdue and set a timeout in order to abort it in case the processing unit is non-responsive, e.g., if it is stuck in an endless loop. As the execution times always vary to a certain extent, the timeout is calculated from the execution time multiplied by a sufficiently large value. A good value for the timeout depends on the actual system, as other applications might delay the execution. However, choosing a too high or too low value has no impact on correctness and only degrades the performance as execution is either aborted too late or too soon.

7.3.4.2. N-modular redundancy

Despite its high costs under certain conditions, redundant execution with result comparison is still a common approach to detect and tolerate faults. Theoretically, it can detect and tolerate all kinds of faults that would lead to incorrect results except the very unlikely case that the faults falsify all results in the same way.

In homogeneous systems, usually only faults in hardware can be detected as the same task implementation is executed on different processing units. However, in heterogeneous

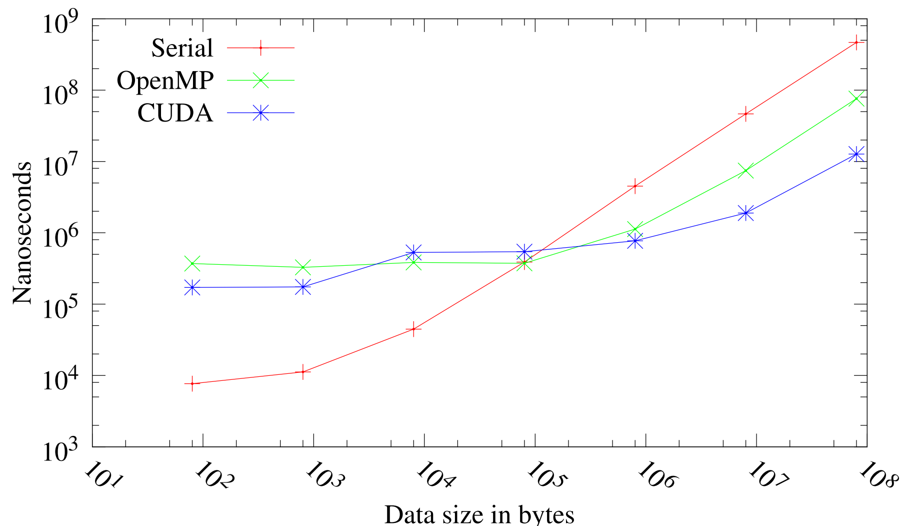


Figure 7.22.: Time consumption for data comparison with different programming models and sizes

systems, not only different processing units are available but also different implementations of a task. Therefore, also faults in software can be automatically detected if different implementations are used. To leverage this opportunity, different circumstances have to be considered, though.

After the redundant runs have finished, the results have to be compared. The comparison can consume a considerable amount of time that – similar to a regular compute task – depends on the input data size and location as well as the chosen processing unit as it can be seen in Figure 7.22. Therefore, the runtime system also considers the comparison as an own task that is mapped using the normal scheduler. If desired, the comparison can be even redundantly executed itself.

The actual comparison of the results is done by bit-wise comparison of values in most cases, e.g., if the values are common integer types. However, if the values have a special type such as floating-point numbers, also special mechanisms are necessary for heterogeneous systems: as it is possible that the order of instructions and the rounding modes differ between the various types of processing units, the results may diverge although they are correct. For such a case, the programmer may define a maximum that determines how much two floating-point numbers are allowed to differ while they are still considered equal by the voter in the runtime system. During the experiments with the OpenMP and CUDA implementations of the Rodinia benchmarks, the floating point values in the results differed by 0.001% to 0.1% depending on the application. Therefore, a delta of 0.1% was set as acceptable for the calculations with single precision.

7.4. Decision making

The previous sections introduced different possible execution variants and abstract models that uniformly encapsulate specific properties of these variants and thus enable a standardized estimation of costs. The remaining step before actual execution is determining the best combination of tasks and execution variants. In the following, this work introduces the implemented mechanisms that make such a decision under given circumstances.

For graph scheduling, an extensive amount of research has been conducted [27, 89, 131, 90]. For a scheduling algorithm, the system architecture of today's general-purpose computers

is most akin to heterogeneous distributed systems. There are multiple processing units that offer different performance for the same task, possess their their own memory and are connected over comparatively slow interconnect hardware which creates considerable communication costs. However, the number of individually controllable processing units are yet low in today's systems, e.g., threads can be bound to single CPU cores but a user has no control over the thread scheduling on a GPU with its several hundred cores. As CPU implementations usually exploit all CPU cores, e.g., using OpenMP, the remaining choice of a task mapper is either the CPU or the GPU, in most cases.

Most approaches consider performance differences between processing units and dynamically select the best unit based on previously measured execution times [67, 164, 21] or, in addition, maintain per-device waiting queues which equals the Heterogeneous Earliest Finish Time (HEFT) algorithm [154, 153, 13, 24]. In contrast to basing decisions on past experiences, other projects use source code features and machine learning to select the best processing unit for execution [58].

Many approaches also offer different scheduling algorithms but they commit themselves to a standard algorithm that is only changed on the initiative of a user. Instead, Dastgeer and Kessler introduced an initial evaluation of pattern-based selection of scheduling heuristics [36]. In their work, they use annotated source code to mark specific task patterns like concurrently executable or dependent tasks and show that the patterns benefit from different heuristics. In a subsequent publication, they extended this approach with annotations similar to the attribute approach in this work [37]. In this work, the runtime system is also able to switch between different scheduling algorithms but their benefit is evaluated dynamically for the given application.

Similar to the other modern approaches, this work provides different scheduling algorithms like the HEFT algorithm. In Figure 7.23, the resulting application runtime with this algorithm is shown using the OpenCL version of the Rodinia benchmarks. In the chosen system, three processing units are available: an Intel i7-3610QM CPU with HyperThreading resulting in 8 logical cores, an integrated Intel IvyBridge HD4000 GPU and a dedicated Nvidia NVS 5400M GPU. First, the application runtime was measured with each OpenCL library of the respective vendor and without the runtime system. Afterwards, the runtime with the OpenCL wrapper and the HEFT algorithm has been measured. For each processing unit, the scheduler can choose between two execution variants: one time, the data is explicitly transferred and the other time the data is mapped into the address space of the unit. Hence, for each task, the scheduler can choose between six candidates. In the figure, we used the runtime on the CPU as baseline and show the increase or decrease of the runtimes as factor relative to the CPU. As we can see, most of the benchmarks show better performance when they are executed on the GPU, except for the `gaussian` and the `kmeans` benchmark. In some of the cases, the runtime system with the HEFT algorithm is almost as fast as always executing on the fastest processing unit and for benchmarks with a short runtime even slightly faster as the runtime system only loads and initializes OpenCL libraries on demand. In some cases, however, the dynamic selection is considerably slower than always executing on the fastest processing unit. For example, in case of `heartwall` or `kmeans`, we can see that the runtime is similar to another result with a specific processing unit, which indicates a wrong choice of the HEFT algorithm. Also, if we reconsider Table 4.1, we observe an increased runtime especially for the benchmarks with:

- high number of tasks,
- a low number of tasks per task graph and
- a low average time consumption per task

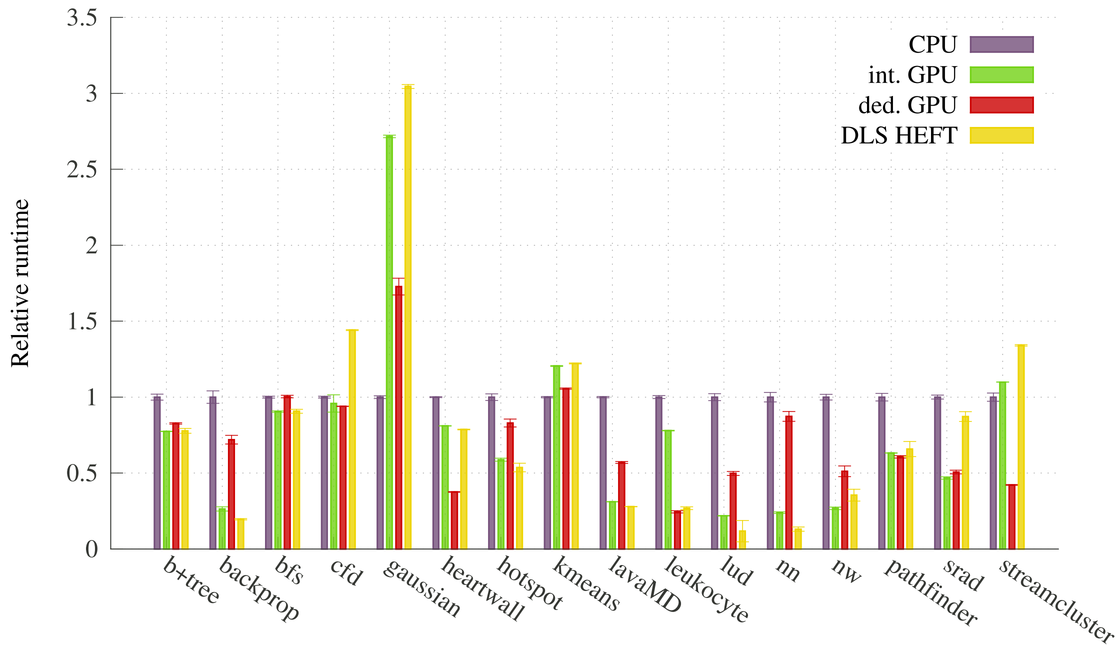


Figure 7.23.: Comparison of application runtimes with native OpenCL libraries and the DLS OpenCL wrapper with HEFT algorithm

namely `cfd`, `gaussian`, `srad` and `streamcluster`.

The problem with HEFT scheduling in these cases is that it is unable to determine the best choice which reveals the other problem that there is a high scheduling-to-execution time ratio per task for the latter benchmarks with a low average runtime per task. A common problem for HEFT are high startup costs and first tasks with a short runtime like initialization tasks that set up the structures for following long running tasks, for example. If we compare the initialization costs of the OpenCL libraries in the following table:

Intel CPU:	255 ms
Intel GPU:	25 ms
Nvidia GPU:	181 ms

with the runtimes of the first tasks, e.g., 28-40 µs for the `gaussian` benchmark, it is not suprising that HEFT tends to choosing the internal Intel GPU for execution. To make things worse, a switch to a faster processing unit for later long-running tasks becomes more unlikely, as they have to compensate their initialization costs plus the time required for transferring the data from the memory of the initial processing unit. To document this behavior, we listed the number of compute tasks that are mapped to either of the three processing units in Table 7.2. Only for the `leukocyte` and `pathfinder` benchmarks, the HEFT algorithm chose also a processing unit other than the internal Intel GPU. However, in both cases, the switch to the dedicated GPU is too late and the performance is slightly worse than choosing the dedicated GPU from the beginning.

An initial attempt to solve this drawback of the HEFT algorithm was to employ the container mechanism and simulate the execution with different algorithms. For cases with a strong bias towards a single processing unit, we added a scheduler that simply chooses the same execution variant for every task and repeats the simulation for every available variant. In Figure 7.24, the results with this multi-scheduler are shown. As we can see, this

Benchmark	CPU	int. GPU	ded. GPU
b+tree	0	2	0
backprop	0	2	0
bfs	0	24	0
cfd	0	16004	0
gaussian	0	2046	0
heartwall	0	20	0
hotspot	0	1	0
kmeans	0	38	0
lavaMD	0	1	0
leukocyte	0	2	10
lud	0	46	0
nn	0	1	0
nw	0	255	0
pathfinder	0	1	4
srad	0	502	0
streamcluster	0	4833	0

Table 7.2.: Number of tasks mapped to each processing unit by HEFT algorithm

approach still does not improve the situation and the runtimes get even worse due to the additional scheduling overhead. After analyzing the runtimes calculated by the simulation, the low number of tasks per task graph has been determined as cause for this behavior. Due to the frequent synchronizations, the graphs are too small and even when forcing an suboptimal mapping in the beginning, the number of following tasks is too small and the overhead cannot be compensated within one task graph.

During the following analysis of the source codes, also unnecessary synchronization commands were found in the `gaussian` and `heartwall` benchmarks for performance measurements. As we see in Figure 7.25, removing these commands improves the performance of the `gaussian` benchmark and results in an almost on-par performance with the best processing unit for the `heartwall` benchmark as in both cases, only at the end of the application a data transfer is really necessary and all tasks now appear in the same graph. The remaining overhead in case of the gaussian benchmark is caused by the scheduling of the numerous but small similar tasks. For most of the benchmarks with a high number of tasks, a large amount of the tasks are repetitions of a group of one to eight tasks. As most scheduling algorithms will make the same decisions for each repetition, the runtime system uses loop detection to avoid estimating the costs of each execution variant for each task. As shown in Figure 7.26, the repeating tasks are replaced by a special loop task that contains a list of tasks that represent one iteration of the loop body and a list of all tasks in this loop. As the task dependencies may differ between the beginning, the middle and the end of the loop, the loop detection only creates a loop task if the loop body repeats at least four times. For instance, if we consider another example in Figure 7.27, we see that the dependencies of the tasks C and D differ between the first iteration, the iterations in the middle and the last iteration. Therefore, the runtime system first unrolls two iterations of the loop body and uses the second iteration as sample for the remaining iterations. If a scheduling algorithm wants to reevaluate the decisions for the body, e.g., in case a competing application is running, the unrolling can also be interrupted after a chosen number of iterations and continued with a new body sample.

To remove the remaining mispredictions, a scheduler requires a complete view of all tasks executed during one application run. Therefore, the runtime system accumulates all

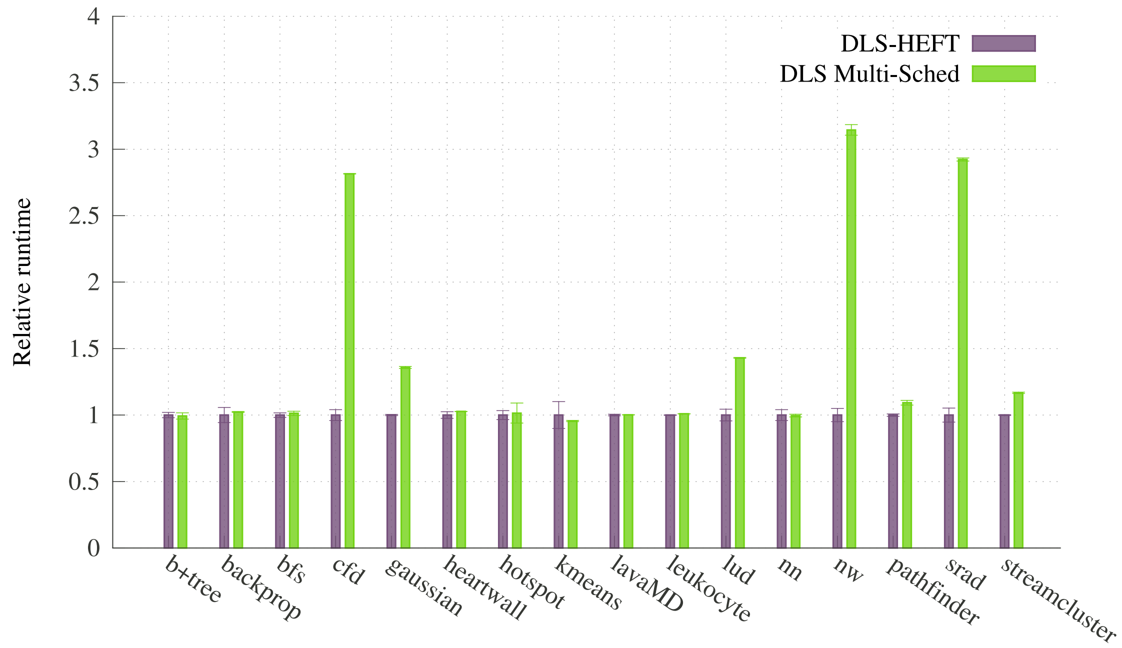


Figure 7.24.: Application runtimes with HEFT and the container-based multi-scheduler

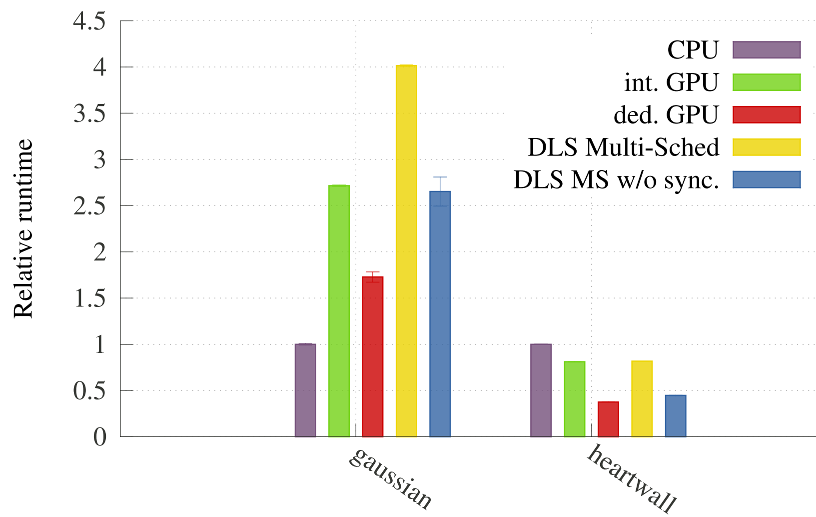


Figure 7.25.: Runtimes after removing unnecessary synchronization

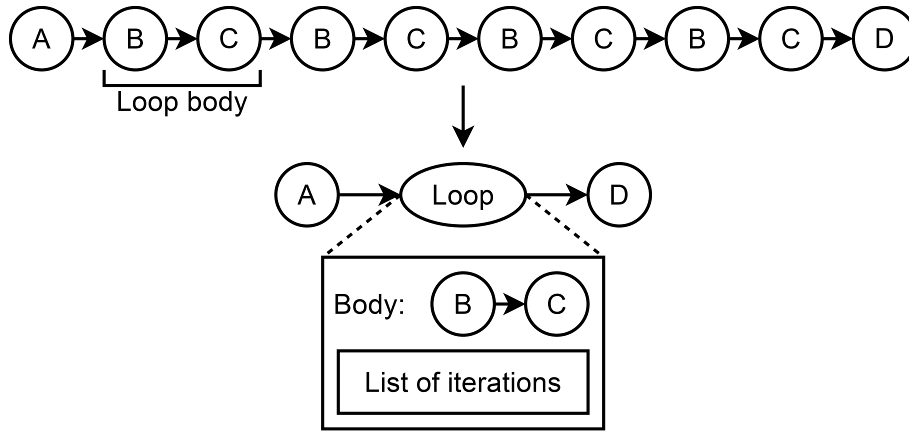


Figure 7.26.: Loop detection and resulting loop task

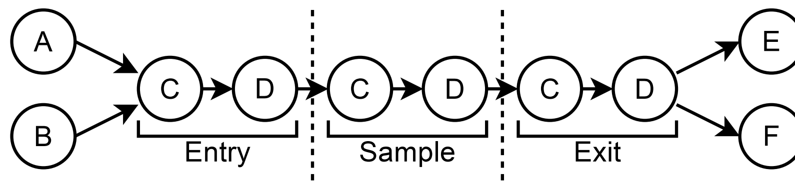


Figure 7.27.: Loop unrolling

submitted tasks during an application run and starts the multi-scheduler again at the end of the application run without executing the schedule. The best scheduler is then stored and used for following application runs. In Figure 7.28, the resulting runtimes with a cached scheduler and loop detection is shown in comparison to the original runtimes and the runtimes with the normal HEFT scheduler. As we can see, this technique enables the multi-scheduler to successfully determine the fastest processing unit. Except for the cfd, gaussian, srad and streamcluster benchmark the resulting runtime is equal or even slightly faster than choosing the best processing unit manually. If we consider Table 4.1 again, the performance of the runtime system is slightly worse for the benchmarks with a high number of task graphs. Similar to the loop detection for tasks, either avoiding synchronization or a caching mechanism for graph schedules would be necessary to lower the overhead in such cases.

While the multi-scheduler already alleviates the problem of the HEFT algorithm with high initialization costs, both may not find the global optimum in certain cases. To give an example of an algorithm that may find a global optimum, simulated annealing has been implemented as part of this work [83]. Simulated annealing is an iterative algorithm that chooses a random task in a schedule and a new random location for this task in one of the waiting queues for the available processing units in every iteration. If the resulting schedule is better, e.g., if the makespan is lower than any previous schedule this schedule is stored and used for further iterations. If the schedule is worse, it is either discarded or it may be accepted nonetheless with a certain probability. This probability depends on how much worse the schedule is and on the current so-called temperature value. Before the algorithm is started, a range of temperatures is calculcated and the temperature decreases from the highest to the lowest temperature. If the lowest value is reached the algorithm is terminated. The closer the makespan is to the previously chosen schedule and the higher the temperature is, the higher is also the probability for an acception of a worse schedule.

As proposed by Orsila et al. [117], the initial temperature T_0 is calculcated from a constant k with $k \geq 1$, the maximum execution time for any task on any processing unit t_{max} and

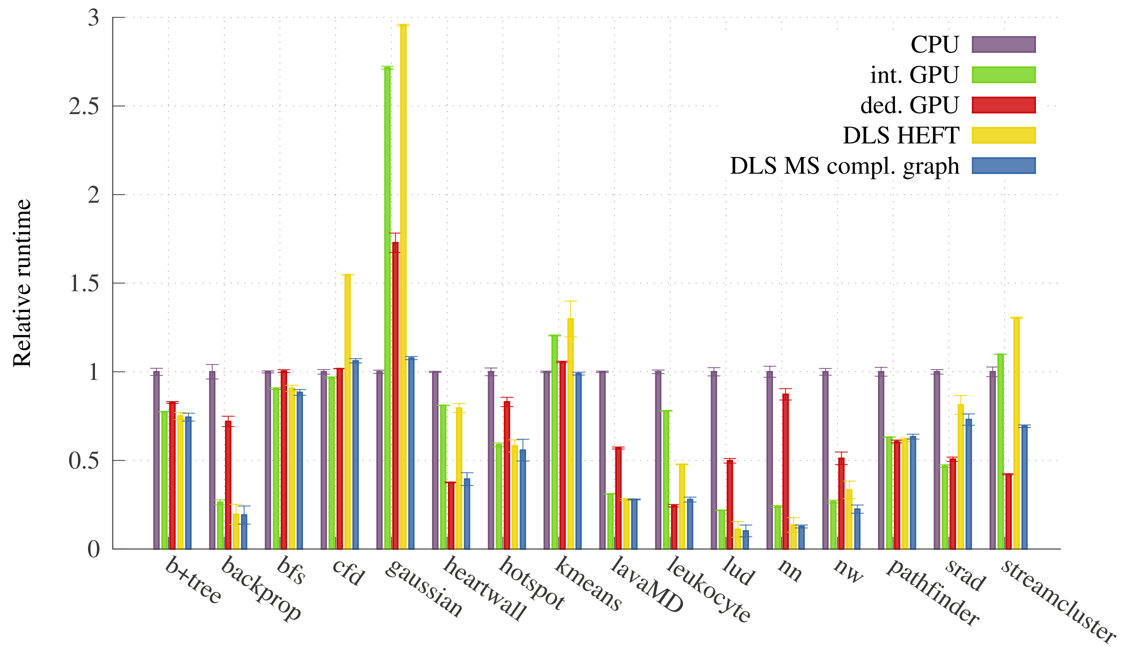


Figure 7.28.: Runtimes with complete graph analysis and best scheduler caching

the sum of execution times for all tasks on the fastest processing unit t_{minsum} as follows:

$$T_0 = \frac{k * t_{max}}{t_{minsum}} \tag{7.1}$$

In turn, the final temperature T_f is calculated from the minimum execution time for any task on any processing unit t_{min}, the same constant k and the sum of execution times for all tasks on the slowest processing unit t_{maxsum} as follows:

$$T_f = \frac{t_{min}}{k * t_{maxsum}} \tag{7.2}$$

In this work, the constant k has been set to 1. A new temperature T_i for iteration i is calculated from the previous temperature T_{i-1}, a geometric temperature scaling factor q and the number of mapping iterations per temperature L as follows:

$$T_i = T_{i-1} * q^{\lfloor \frac{i}{L} \rfloor} \tag{7.3}$$

In this work, q has been set to 0.95. Orsila et al. calculate the number of mapping iterations per temperature level L using the number of tasks N and the number of processing units M as follows:

$$L = N(M - 1) \tag{7.4}$$

As the runtime system offers more options per tasks as the number of processing units, e.g., multiple implementations for the same processing unit, M equals the average number of execution variants that are available for each task in this work. For example, for a switch task that includes alternative sequences of tasks, M would equal the number of alternative sequences.

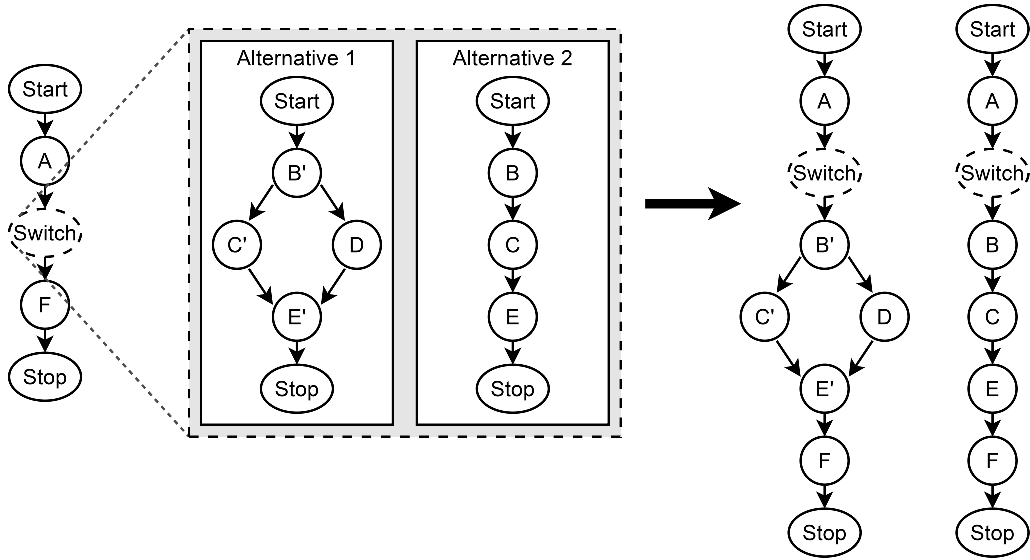


Figure 7.29.: Original graph and resulting graph after randomly selecting an alternative branch

If a worse schedule is accepted, depends on an accept function a that is calculated from the difference between the makespan of the last accepted schedule and the current makespan ΔC, the makespan of the initial schedule C_0 and the current temperature T as follows:

$$a(\Delta C, T) = \frac{1}{1 + exp\frac{\Delta C}{0.5 * C_0 * T}} \tag{7.5}$$

A worse schedule will be accepted if a new random value is below the result of the accept function a. In addition to the temperature underrun, the algorithm in this work is also terminated if $R_{max} = min(L, 50)$ consecutive schedules have been rejected.

In order to receive a schedule that is at least as good as the schedule calculated by the HEFT algorithm, this schedule is used as initial schedule. During every iteration, a new sub-container is created and the previously accepted schedule is applied again except for one random task. First, a new valid position for this task in the order of tasks is determined randomly. Afterwards, if the task is a compute task, a random task mapping is chosen. If the task is a switch task, a random sequence of tasks is chosen. This sequence is then inserted in the graph after the switch task as shown in Figure 7.29. Hence, for further iterations, the algorithm can also choose new positions and mappings for the tasks in a conditional branch and it can randomly select a different branch.

Afterwards, the makespan can be calculated and the container is either released if the schedule is not accepted or kept in the background if it is accepted. If the algorithm terminates, the container with the best schedule is merged and executed.

A comparison of this algorithm with the other schedulers is presented in the following Section 7.5.1.

7.5. Evaluation

In this section, this work presents the results of further experiments. First, a case study is presented for the preprocessing of medical imaging. Afterwards, the mechanisms for fault tolerance are evaluated.

Figure 7.30.: The 3D USCT II system for ultrasound computer tomography [22]

7.5.1. Case study: Preprocessing for medical imaging

One project at the Institute for Data Processing and Electronics (IPE) led by Prof. Dr. Marc Weber at the Karlsruhe Institute for Technology is ultrasound computer tomography (USCT) for early breast cancer detection [1]. Breast cancer is the most common type of cancer for women in western countries. To raise the probability for a successful recovery, it is very important to detect cancer as early as possible before metastases were built in the vital parts of the body.

The benefits of USCT are no radiation exposure and a painfree procedure. In the past, the resolution of such approaches enabled the detection of cancer with a size of 1 cm or more. For cancer of this size, the probability for metastases is at 30% [23]. The goal of the USCT project is to enable the detection of objects with a size of less than 1 mm in order to lower the probability of metastases below 5% while keeping the time consumption of the image processing low and enable an examination by a doctor during one consultation. To achieve this goal, the scientists at the IPE work on all parts of the process: from the ultrasound receivers in the device (q.v. Figure 7.30) to the actual image reconstruction on a computer.

In order to lower the time consumption of the image reconstruction, heterogeneous systems with different accelerators are evaluated [15]. As we can see in Figure 7.31, this procedure is separated in three main steps: the signal acquisition, signal processing and the actual image reconstruction. In this scenario, FPGAs are used for the signal acquisition and GPUs for the image reconstruction but, for the reconstruction, other types of architectures are evaluated as well. Besides the hardware, the scientists also research new algorithms to increase the performance of the system. Finding the best schedule in such cases becomes a considerable amount of work for the developers as each improvement of one task implementation or new hardware might change the best schedule. While related work already provides mechanisms to find the best schedule for multiple tasks in a heterogeneous system, the support of conditional task graphs in this work also enables the automatic evaluation of alternative algorithms. As an example, this chapter presents an evaluation of this work in combination with the preprocessing stage of the image reconstruction. This stage consists of five tasks that have to be executed in sequential order but each task can be executed on a CPU or GPU. In addition, alternative algorithms for three of these tasks are considered in this example. Each of these tasks can either operate in the so-called time or frequency domain and to switch between the two domains, the data has to be transformed. In Figure 7.32, the task graph the runtime system constructs from the submitted tasks is shown. On the

[1] http://www.ipe.kit.edu/167.php

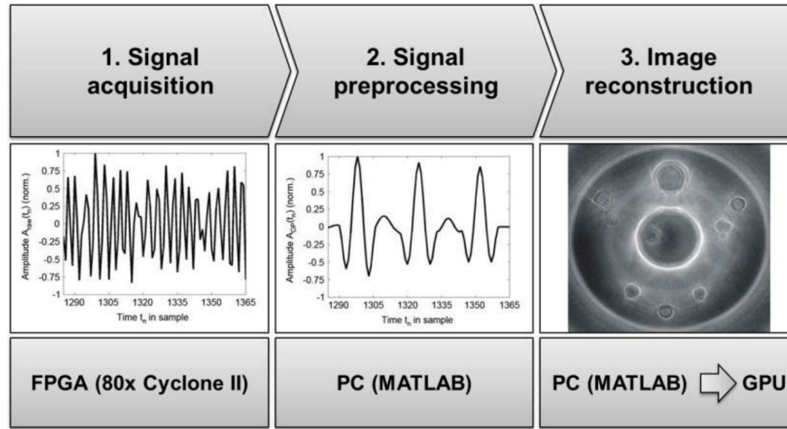


Figure 7.31.: The main steps for image reconstruction [88]

left, we see the branch with the tasks that operate in the time domain and, on the right, the tasks that operate in the frequency domain as well as two transformation tasks that convert the data into the required format.

To simplify development, the main procedure as well as new algorithms are written in Matlab and proven algorithms are ported to C/C++ and accelerator-specific programming models afterwards by hardware experts. To minimize the impact on this workflow, this work introduced the Matlab interface of the runtime system in Chapter 4.3. With this interface, the developers can still control the main logic as well as the runtime system from within Matlab and can add internal Matlab functions or external C/C++ functions as implementations for each task. As example, the code generating the above graph is listed in Appendix A.

The evaluation was performed on an Intel Core i7-3930K system with 12 CPU cores, two Nvidia GeForce GTX 690 and one GTX Titan running OpenSUSE 12.2 64bit. Hence, in addition to the alternative implementations, there are four execution variants for every task. During evaluation, the execution time of the task graph and the time consumption of the HEFT, simulated annealing, brute force and multisched algorithms has been measured. The average results after 40 runs with the HEFT algorithm as baseline for the speedup are presented in the following table:

Algorithm	Exec. time [s]	Speedup	Search time [s]	Ex.-to-search ratio
HEFT	0.262	-	0.177	1.48
Simulated annealing	0.223	1.17	0.462	0.482
Brute force	0.223	1.17	4.229	0.052
Multisched	0.218	1.20	0.198	1.101

Despite the small differences of the execution times between simulated annealing, brute force and multisched, all three find the best schedule that chooses the time domain branch and maps all tasks to the GTX Titan GPU. Due to the high initial costs, HEFT maps the first task of the time domain branch on the CPU. As the multisched scheduler provides the best execution time per search time ratio of the successful algorithms, it is the best scheduler in this example. To find the best scheduling, simulated annealing evaluated between 100 and 200 schedules and the brute force algorithm 2048 possible combinations.

7.5.2. Fault-tolerant task execution

To evaluate the mechanisms for fault-tolerant task execution, selected applications from the Rodinia benchmark suite [29] and the Nvidia CUDA SDK were used. However, some

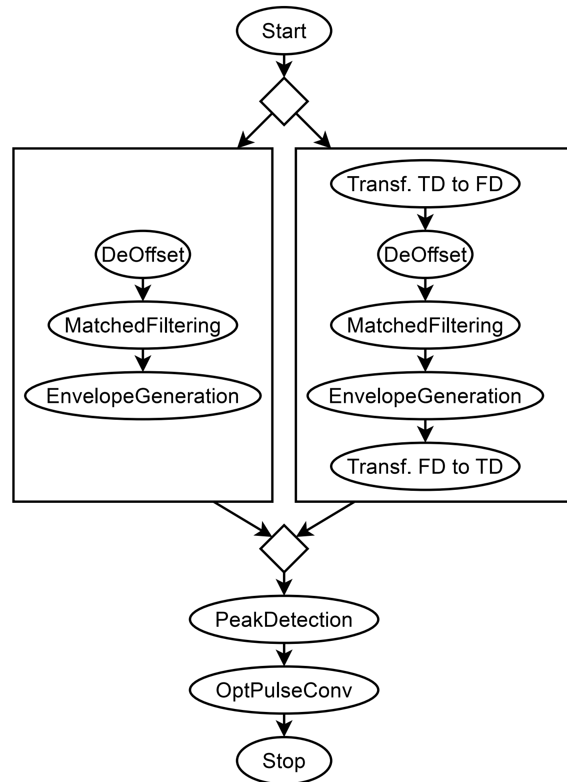


Figure 7.32.: Input task graph of the preprocessing example

applications of these collections were not used as they were not suitable for the DMR concept, e.g., benchmarks where the OpenMP and the CUDA versions calculate different results, e.g., due to different data structures, or the versions make use of static variables which exclude a parallel execution of the functions.

To simulate faults, a small routine was integrated in the kernels of the applications that causes a fault under predefined conditions. By setting a specific environment variable, the routine either causes a segmentation fault, sleeps for an infinite time or writes a wrong value into the results. This routine is called at the end of the kernel, therefore the following results represent the worst-case execution times.

The evaluation was conducted on a dual AMD Opteron 2378 machine equipped with 8 CPU cores, an Nvidia GeForce GTX 275 and a GeForce GTX 560Ti GPU running with Ubuntu Linux 12.04. In order to decrease runtime variation due to competing tasks on the system, the number of OpenMP threads was limited to 7. Therefore, competing tasks like maintenance routines can execute on the eighth CPU core without increasing fluctuations of the measurements.

7.5.2.1. Comparison of the different operation modes

The runtime system provides different fault detection mechanisms that have varying impact on the performance. In this experiment, this impact is quantified using the selected applications. First, the raw execution time of the kernels on the different processing units is shown in Figure 7.33. Then, the required time with the runtime system and performance-oriented mapping (Perf), with performance-oriented mapping and checkpointing (Perf+CP) for light-weight fault tolerance, normal Dual-Modular Redundancy (DMR) and heterogeneous Dual-Modular Redundancy (HetDMR) are depicted in Figure 7.34. As we can see, the checkpointing adds a small additional overhead for creating redundant copies of the output data and the DMR strategy increases the runtime by factor 1x to 3x. Except for the

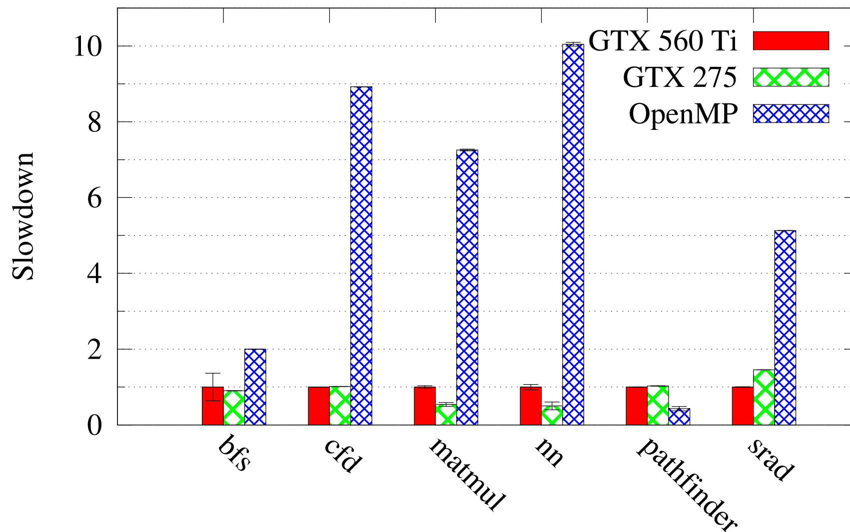


Figure 7.33.: Comparison of average kernel execution time on different processing units

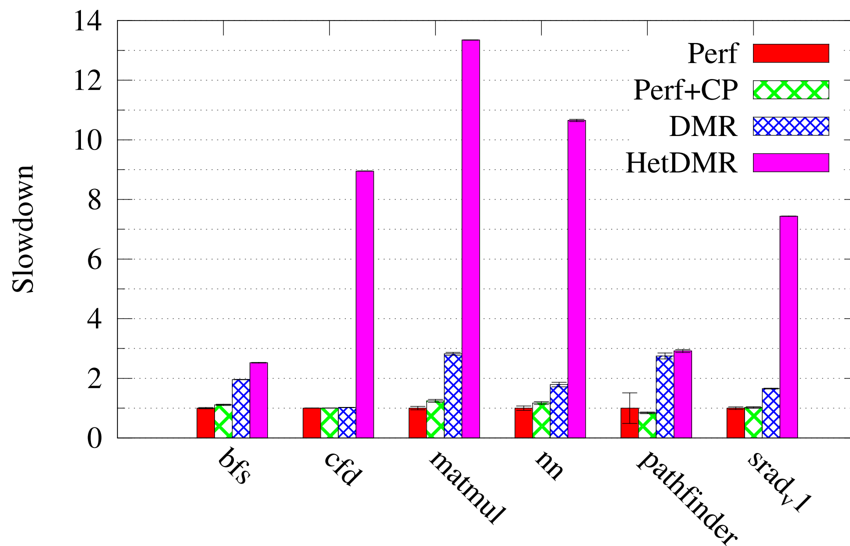


Figure 7.34.: Overhead of the three modes compared to performance-oriented task mapping

pathfinder benchmark, the HetDMR strategy enlarges the overhead significantly as the OpenMP kernel is considerably slower than the CUDA kernels. DMR and HetDMR are about the same for the pathfinder benchmark, as the OpenMP kernel is the fastest here and thus DMR and HetDMR choose the same processing units (the CPU and one GPU) for execution.

In order to determine the overhead in case of a fault, the described fault routine was integrated in the implementations. In the Perf+CP mode, a segmentation fault was induced while in each case one result during DMR and HetDMR measurements was falsified. The results of this evaluation are presented in Figure 7.35 with the fault-free performance-oriented mapping (Perf) mode as baseline. As we can see, a fault during Perf+CP mode causes a slowdown of around 2x while the DMR and HetDMR mode introduce a similar high overhead as in both cases the task is executed on all processing units, the CPU and the two GPUs, and the runtime system has to wait on the slowest processing unit in any case.

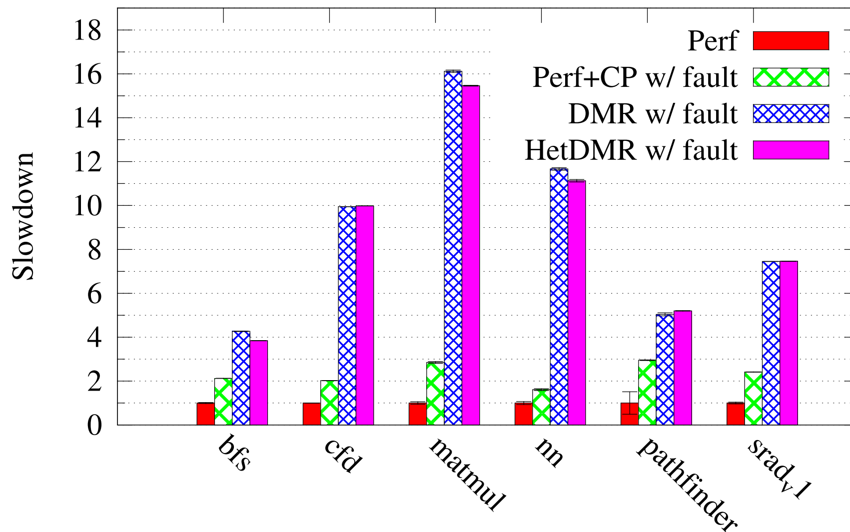


Figure 7.35.: Slowdown of the different modes in case of a single fault

7.5.2.2. Benefit of the fault-aware runtime estimation

In Chapter 6.5.2, a new metric, called fault-aware runtime estimation, was introduced. This metric is used by the runtime system to determine the remaining benefit of susceptible processing units. In this experiment, the benefit of this metric is shown using the pathfinder application of the Rodinia benchmarks. With a varying probability, a segmentation fault is injected during the execution of the OpenMP implementation and the required time until the correct result is returned to the application is measured one time with the normal, performance-oriented task mapping and checkpointing (Perf+CP) and one time with the same strategy which uses the fault-aware runtime estimation for selection, though (Rel+Perf+CP).

As the OpenMP kernel is roughly 3x faster than the best GPU kernel, the runtime system with the new metric should prefer the slower GPU kernel if the fault probability of the OpenMP mapping rises above $p = 1 - \frac{1}{3} = 66\%$. In Figure 7.36, it can be seen that, as expected, the reliability-aware approach reduces the required time beginning with a fault probability of 70%. Compared to avoiding the CPU as soon as it caused a fault, this approach is, e.g., for 10% fault probability, more than 2 times faster. In case of a permanent fault (100% probability), this approach saves 30% of the execution time compared to always choosing the fastest processing unit.

7.5.2.3. Handling multiple failing devices

So far, only cases where one processing unit caused corrupted results were considered. However, it is also possible that multiple processing units fail in parallel, e.g., due to a failing system fan which causes spatial overheating. Therefore, this experiment demonstrates the behavior of the runtime system in such a case using again the pathfinder benchmark. In this experiment, the runtime system executes all tasks with a dual-modular redundancy strategy. To tolerate two misbehaving devices, the DMR strategy requires four devices. Under normal conditions, only three units (CPU, GPU 1 and GPU 2) can be used individually in the test system as OpenMP occupies all CPU cores at once. Therefore, the number of OpenMP threads has been manually limited to one thread for each run or task. Consequently, the runtime system can use the eight CPU cores individually and it has enough redundancy to tolerating two misbehaving devices.

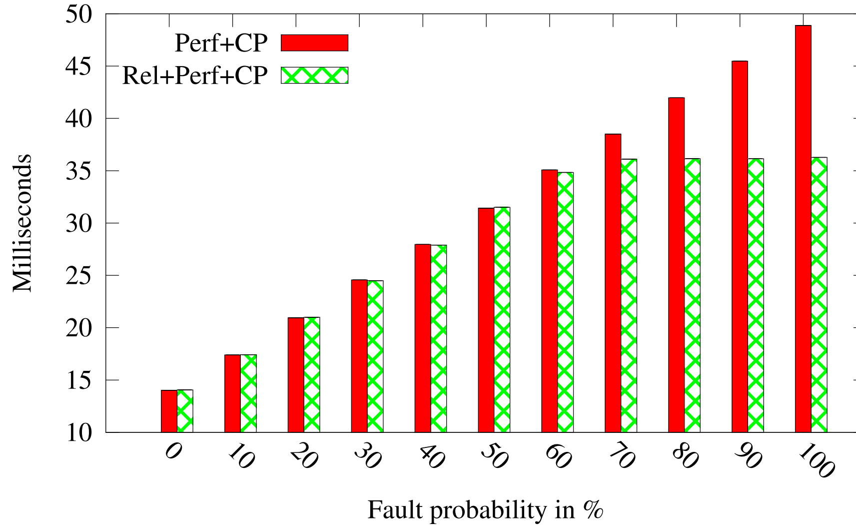


Figure 7.36.: Runtime with performance-oriented and fault-aware task mapping

In Figure 7.37, the results of this experiment are shown. The application was started several times and the X-axis shows the progress of time measured in number of application runs. The right Y-axis shows the fault probabilities of GPU 1 and GPU 2 and the left Y-axis shows the processing units that are used during this experiment. The probabilities are drawn as lines and the blue boxes represent a correct calculation on the respective processing unit during that application run.

At the beginning, all devices work properly, so the runtime system receives correct results from GPU 1 and 2 as indicated by the blue boxes. Between application run number 5 and 20, a segmentation fault was induced in GPU 1. This results in the expected increase of the fault probability and the runtime system starts to execute the task on CPU core 1 as well. Afterwards, a segmentation fault was also induced during execution on GPU 2 between application run 15 and 30. Thus, between run 15 and 20, both GPUs suffer from an intermittent fault and both CPUs are required for 5 runs to determine the correct result. At application run 20 and 30, the GPUs continue to produce correct results and their fault probability decreases to zero. Hence, in case there is no usable processing unit of a certain type left, the runtime system is able to continue execution on any other type of processing unit with an available implementation.

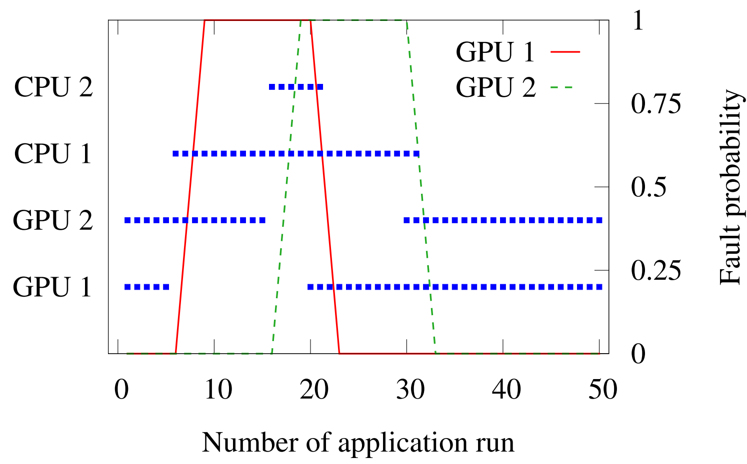


Figure 7.37.: Measured fault probability and correct calculations during intermittent faults on GPU 1 and 2

8. Conclusion

In this chapter, the presented work is concluded with a summary of the contributions and with an outlook on future works and opportunities.

8.1. Summary

A multitude of challenges await developers while adding support for heterogeneous systems to their applications and optimizing the performance. Considering and handling all these challenges in the source code of the application requires considerable efforts from a developer and makes the development and the resulting application considerably more complex.

To improve the portability of applications without loosing the benefits of hardware-optimized task implementations, this work proposed a decoupled development concept that outsources hardware-specific implementations into separate libraries and uses a new runtime system to dynamically load only executable implementations which also fulfill the requirements of the application. Using this concept, not only the portability of applications is improved but also already compiled applications can automatically benefit from new implementations, e.g., optimized BLAS libraries bundled with new accelerators, if they share the same interface. In the evaluation, it has been shown that, with this concept, the very same applications were successfully executed on systems with different configurations that would otherwise inhibit the execution of the application due to missing dependencies.

While most other related works expect exclusive usage of a flawless system, this work also considers competing applications and faults occurring during task execution. Different methods to detect such events are included and a new metric has been introduced that enables a scheduler to trade off performance benefits with the observed fault rate in order to dismiss a processing unit only if the average effort for fault recovery exceeds the performance benefits compared to other units in the system. Compared with naive approaches that either keep using a susceptible processing unit or avoid it as soon as it calculated one wrong result, this metric reduces the average execution time by up to 50%. Similarly, with the integration of shared-memory waiting queues, adjusting the scheduling decisions during periods of competition enables a reduction of the execution time by 30% compared to an approach that relies solely on the operating system and device drivers to resolve competition.

To improve the performance, related work mostly focus on specific optimizations of application and use a rather simple but effective greedy scheduling algorithm. However,

to gain good performance, multiple optimizations have to be evaluated and dynamically applied and a greedy algorithm might not find the best schedule for an application and system. Evaluating the benefit of multiple optimizations at once and their implications on each other as well as enabling demanding scheduling algorithms that can find the global optimum becomes considerably complex. Therefore, this work introduced a concept based on so-called containers that enable an online simulation of application execution on task level to evaluate the outcome of a combination of different techniques and to enable scheduling algorithms like simulated annealing which require a considerable amount of iterations before a good schedule may be found. By simulating application execution in advance with different scheduling algorithms, this approach achieved a reduction of the execution time by 20% compared to the common greedy scheduler.

As one goal of these mechanisms is to lower the complexity of an application's source code, a considerable topic is also the required effort for integrating these mechanisms themselves. Therefore, besides a native C interface that enables an integration in different other languages, this work also introduces wrapper libraries for Matlab and OpenCL applications that facilitate rapid prototyping and enable a completely transparent integration, respectively.

To conclude, while related works only simplify and automate specific aspects to lower the complexity, this work proposed a new framework implemented with an online-learning runtime system that provides a common basis for different mechanisms to simplify development and make applications more portable, efficient and reliable across different systems. Furthermore, the framework also provides a tool for computer scientists and hardware experts that encourages the research of new runtime optimizations and hardware-optimized task implementations.

8.2. Outlook

Due to the raising interest in accelerators, the programming of heterogeneous systems will become easier with new languages and tools that specifically target parallelism and heterogeneity. Especially due to unified address spaces, sharing data between the CPU and accelerators becomes significantly less troublesome and error-prone. However, hardware-specific optimizations and efficient data placement will still be necessary to maximize performance. In addition, with tightly coupled CPU and GPU cores, heterogeneous computing might become also ubiquitous for more general applications besides high-performance computing or games that use the parallelism available in modern CPUs but only offer a small profit margin for accelerators that is immediately neutralized by necessary data transfers. Hence, if the execution times on the available processing units become closer with a unified memory hierarchy, a general assumption which type of processing unit is the most profitable on the current system might inhibit optimal performance and carefully choosing the best processing unit for a task will be necessary. However, as we saw in this work, the performance cannot always be accurately predicted for certain processing units. Therefore, improving such predictions, e.g., by combining different hardware performance counters, will be a topic of future works that also benefits dependability. If also the overhead for querying these counters is reduced, symptom-based fault detection can be enabled for any task similar to the monitoring of the execution time in this work.

As the fault coverage of symptom-based fault detection is limited, this work also offers mechanisms for redundant execution to detect faults leading to corrupted results. Which of these mechanisms are enabled during execution can be chosen dynamically. Hence, the effort for dependability can be adjusted by a scheduler, e.g., for single tasks, which encourages future works that balance dependability with other objectives.

In this work, the experiments focused on the reduction of the execution time. However, due to its modular design, the framework in this work can be extended to consider other types of objectives. For example, with appropriate power sensors, the energy consumption of a task mapping can be measured in addition to the execution time and used as determinant for decision making. Also, the abstract model of processing units during simulation of task execution can be extended to include other parameters like the current frequency or voltage of a processing unit. Hence, in combination with power sensors, different schedulings can be evaluated in advance if they comply with given power limitations and deadlines, for example.

Similar to high-performance computing, also the demand for computational power in mobile devices like mobile phones or wearable computers rises for augmented reality applications or games, for example. However, due to the power and size limitations, such devices will stay less powerful than, e.g., desktop or server systems. Especially for wearables like Google Glass or smartwatches, the computational power is severely limited and some even depend on additional devices like the mobile phone. As the next generation of wireless communication protocols will provide a considerable increase of the data transfer rate to up to several tens of Gbit/s for 5g networks with a latency down to 1 ms [9] and up to several Gbit/s for 802.11ad wireless LANs [120], it might become beneficial to offload certain calculations to more powerful devices in order to improve the quality-of-service. As a connection might not always be available or it has to be shared with other participants, the benefit of offloaded calculcations can vary. Hence, a dynamic decision might become necessary here as well to determine if a remote device can be used under given constraints like data transfer speeds and latency.

Bibliography

[7] E. Ajkunic, H. Fatkic, E. Omerovic, K. Talic, and N. Nosovic. A comparison of five parallel programming models for C++. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 1780–1784, May 2012.

[8] Guilherme Andrade, Gabriel Ramos, Daniel Madeira, Rafael Sachetto, Esteban Clua, Renato Ferreira, and Leonardo Rocha. Efficient Dynamic Scheduling of Heterogeneous Applications in Hybrid Architectures. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 866–871, New York, NY, USA, 2014. ACM.

[9] J.G. Andrews, S. Buzzi, Wan Choi, S.V. Hanly, A. Lozano, A.C.K. Soong, and J.C. Zhang. What Will 5G Be? *Selected Areas in Communications, IEEE Journal on*, 32(6):1065–1082, June 2014.

[10] R. Aoki, S. Oikawa, R. Tsuchiyama, and T. Nakamura. Hybrid OpenCL: Connecting Different OpenCL Implementations over Network. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2729 –2735, 29 2010-july 1 2010.

[11] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 101–112, New York, NY, USA, 2013. ACM.

[12] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Delft, Pays-Bas, August 2009.

[13] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[14] Enes Bajrovic, Siegfried Benkner, Jiri Dokulil, and Martin Sandrieser. Autotuning of Pattern Runtimes for Accelerated Parallel Systems. In *PARCO 2013, September 2013, Munich, Germany*, Advances of Parallel Computing. IOS Press, 2013.

[15] M. Balzer, M. Birk, R. Dapp, H. Gemmeke, E. Kretzek, S. Menshikov, M. Zapf, and N.V. Ruiter. 3D ultrasound computer tomography for breast cancer diagnosis. In *Real Time Conference (RT), 2012 18th IEEE-NPSS*, pages 1–4, June 2012.

[16] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1 –7, sept. 2010.

[17] K.A Bare, S. Kavulya, and P. Narasimhan. Hardware performance counter-based problem diagnosis for e-commerce systems. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 551–558, April 2010.

[18] Michael Bauer, Henry Cook, and Brucek Khailany. CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[19] Michela Becchi, Surendra Byna, Srihari Cadambi, and Srimat Chakradhar. Data-aware Scheduling of Legacy Kernels on Heterogeneous Platforms with Distributed Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 82–91, New York, NY, USA, 2010. ACM.

[20] Michela Becchi, Srihari Cadambi, and Srimat Chakradhar. Enabling Legacy Applications on Heterogeneous Platforms. In *Proceedings of HotPar 2010*, June 2010.

[21] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A Dynamic Self-scheduling Scheme for Heterogeneous Multiprocessor Architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.

[22] M. Birk, A Guth, M. Zapf, M. Balzer, N. Ruiter, M. Hubner, and J. Becker. Acceleration of image reconstruction in 3D ultrasound computer tomography: An evaluation of CPU, GPU and FPGA computing. In *Design and Architectures for Signal and Image Processing (DASIP), 2011 Conference on*, pages 1–8, Nov 2011.

[23] Matthias Birk. *Effiziente Datenverarbeitung auf heterogenen Rechnerarchitekturen für die 3D-Ultraschall-Computertomographie.* PhD thesis, Karlsruher Institut für Technologie (KIT), München, 2014. Zugl.: Karlsruhe, KIT, Diss., 2014.

[24] Michael Boyer, Kevin Skadron, Shuai Che, and Nuwan Jayasena. Load balancing in a changing world: dealing with heterogeneity and performance variability. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 21:1–21:10, New York, NY, USA, 2013. ACM.

[25] Alexander Branover, Denis Foley, and Maurice Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 32(2):28–37, March 2012.

[26] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 79–92, Berlin, Heidelberg, 2009. Springer-Verlag.

[27] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering, IEEE Transactions on*, 14(2):141–154, Feb 1988.

[28] Sang Kil Cha, Brian Pak, David Brumley, and Richard Jay Lipton. Platform-independent programs. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 547–558, New York, NY, USA, 2010. ACM.

[29] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[30] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM Journal of Research and Development*, 51:559–572, September 2007.

[31] Dan Connors, Kyle Dunn, and Jeff Wiencrot. Adaptive OpenCL (ACL) execution in GPU architectures. In *Proceedings of the 3rd International Workshop on Adaptive Self-Tuning Computing Systems*, ADAPT '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[32] Pete Cooper, Uwe Dolinsky, AlastairF. Donaldson, Andrew Richards, Colin Riley, and George Russell. Offload – Automating Code Migration to Heterogeneous Multicore Systems. In Yale N. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 337–352. Springer Berlin Heidelberg, 2010.

[33] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[34] Mayank Daga, Ashwin M. Aji, and Wu-chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing*, SAAHPC '11, pages 141–149, Washington, DC, USA, 2011. IEEE Computer Society.

[35] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

[36] Usman Dastgeer and Christoph Kessler. Towards global composition of performance-aware components for GPU-based systems. In *Proceedings of the 17th Int. Workshop on Compilers for Parallel Computers (CPC-2013)*, 2013.

[37] Usman Dastgeer and Christoph Kessler. Conditional component composition for GPU-based systems. In *Proceedings of the 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2014)*, 2014.

[38] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194, New York, NY, USA, 2010. ACM.

[39] Carlos S. de la Lama, Pablo Toharia, Jose Luis Bosque, and Oscar D. Robles. Static Multi-device Load Balancing for OpenCL. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 675–682, Washington, DC, USA, 2012. IEEE Computer Society.

[40] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[41] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. Understanding Software Approaches for GPGPU Reliability. In *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 94–104, New York, NY, USA, 2009. ACM.

[42] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231, June 2010.

[43] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002.

[44] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*, 2014.

[45] Jianbin Fang, A.L. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216 –225, sept. 2011.

[46] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 385–396, New York, NY, USA, 2010. ACM.

[47] Wu-chun Feng, Heshan Lin, Thomas Scogland, and Jing Zhang. OpenCL and the 13 dwarfs: a work in progress. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 291–294, New York, NY, USA, 2012. ACM.

[48] Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL. In *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, LCPC'10, pages 215–229, Berlin, Heidelberg, 2011. Springer-Verlag.

[49] Jason Flinn and M. Satyanarayanan. Managing Battery Lifetime with Energy-aware Adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, May 2004.

[50] Holger Fröning, Mondrian Nüssle, David Slogsnat, Heiner Litz, and Ulrich Brüning. The HTX-Board: A Rapid Prototyping Station. In *3rd annual FPGAWorld Conference*, 2006.

[51] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, and Shinpei Kato. Exploring Microcontrollers in GPUs. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[52] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato, and Masato Edahiro. Data Transfer Matters for GPU Computing. In *Proceedings of the 19th IEEE International Conference on Parallel and Distributed Systems*, ICPADS '13, 2013.

[53] Benedict R. Gaster and Lee Howes. OpenCL C++. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 86–95, New York, NY, USA, 2013. ACM.

[54] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault Recovery for Chip Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 98–109, New York, NY, USA, 2003. ACM.

[55] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. libWater: Heterogeneous Distributed Computing Made Easy. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 161–172, New York, NY, USA, 2013. ACM.

[56] S. Grauer-Gray, Lifan Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Autotuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.

[57] Dominik Grewe. Prius: A Runtime for Hybrid Computing. In *Proceedings of the First International Workshop on Code OptimiSation for MultI and Many Cores*, COSMIC '13, pages 3:1–3:3, New York, NY, USA, 2013. ACM.

[58] Dominik Grewe and Michael O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 286–305. Springer Berlin / Heidelberg, 2011.

[59] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin Peaks: A Software Platform for Heterogeneous Computing on General-purpose and Graphics Processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[60] Azzam Haidar, Chongxiao Cao, Asim YarKhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. Unified Development for Mixed Multi-GPU and Multi-Coprocessor Environments using a Lightweight Runtime Environment. In *Proceeding of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2014.

[61] IS. Haque and V.S. Pande. Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 691–696, May 2010.

[62] Sylvain Henry. ViperVM: A Runtime System for Parallel Functional High-performance Computing on Heterogeneous Architectures. In *Proceedings of the 2Nd ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '13, pages 3–12, New York, NY, USA, 2013. ACM.

[63] Sylvain Henry, Alexandre Denis, Denis Barthou, Marie-Christine Counilh, and Raymond Namyst. Toward OpenCL Automatic Multi-Device Support. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 776–787. Springer International Publishing, 2014.

[64] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM.

[65] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU Communication Management and Optimization. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[66] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-performance Programming*. Morgan Kaufmann. Elsevier Science & Technology Books, 2013.

[67] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[68] Hai Jin, Bo Li, Ran Zheng, Qin Zhang, and Wenbing Ao. memCUDA: Map Device Memory to Host Memory on GPGPU Platform. In Chen Ding, Zhiyuan Shao, and Ran Zheng, editors, *Network and Parallel Computing*, volume 6289 of *Lecture Notes in Computer Science*, pages 299–313. Springer Berlin Heidelberg, 2010.

[69] Rajshekar Kalayappan and Smruti R. Sarangi. A Survey of Checker Architectures. *ACM Comput. Surv.*, 45(4):48:1–48:34, August 2013.

[70] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, New York, NY, USA, 2014. ACM.

[71] SeungGu Kang, Hong Jun Choi, Cheol Hong Kim, Sung Woo Chung, DongSeop Kwon, and Joong Chae Na. Exploration of CPU/GPU Co-execution: From the Perspective of Performance, Energy, and Temperature. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 38–43, New York, NY, USA, 2011. ACM.

[72] Thomas Karcher and Victor Pankratius. Run-Time Automatic Performance Tuning for Multicore Applications. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 3–14. Springer Berlin / Heidelberg, 2011.

[73] Shinpei Kato, Jason Aumiller, and Scott Brandt. Zero-copy I/O Processing for Low-latency GPU Computing. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 170–178, New York, NY, USA, 2013. ACM.

[74] A. Kawai, K. Yasuoka, K. Yoshikawa, and T. Narumi. Distributed-Shared CUDA: Virtualization of Large-Scale GPU Systems for Programmability and Reliability. In *FUTURE COMPUTING 2012, The Fourth International Conference on Future Computational Technologies and Applications*, page 7–12, 2012.

[75] Mario Kicherer. Design and Implementation of a Low-overhead Run-time System for Self-X Architectures. Diplomarbeit, Universität Karlsruhe (TH), 2008.

[76] Mario Kicherer, Rainer Buchty, and Wolfgang Karl. Cost-aware function migration in heterogeneous systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 137–145, New York, NY, USA, 2011. ACM.

[77] Mario Kicherer and Wolfgang Karl. Heterogeneity-aware Fault Tolerance using a Self-Organizing Runtime System. *ArXiv e-prints*, First Workshop on Resource awareness and adaptivity in multi-core computing, May 2014.

[78] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. Extending a Light-weight Runtime System by Dynamic Instrumentation For Performance Evaluation. In Michael Beigl and Francisco J. Cyzorla-Almeida, editors, *ARCS 2010 Workshop Proceedings*, pages 279–284. VDE, February 2010.

[79] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. Seamlessly portable applications: Managing the diversity of modern heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 8(4):42:1–42:20, January 2012.

[80] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[81] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[82] Volodymyr Kindratenko. Novel computing architectures. *Computing in Science and Engineering*, 11(3):54–57, 2009.

[83] Peter Koch. *Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors*. PhD thesis, 1996.

[84] L. Koesterke, J. Boisseau, J. Cazes, K. Milfeld, and D. Stanzione. Early Experiences with the Intel Many Integrated Cores Accelerated Computing Technology. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, pages 21:1–21:8, New York, NY, USA, 2011. ACM.

[85] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 149–160, New York, NY, USA, 2013. ACM.

[86] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[87] 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)*, pages 30–38. Computer Architecture Group, Institute for Computer Engineering (ZITI), University of Heidelberg, February 2009.

[88] Ernst Kretzek, Michael Zapf, Matthias Birk, Hartmut Gemmeke, and Nicole V. Ruiter. GPU based acceleration of 3D USCT image reconstruction with efficient integration into MATLAB. *Proc. SPIE*, 8675, 2013.

[89] Yu-Kwong Kwok and I Ahmad. Benchmarking the task graph scheduling algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 531–537, Mar 1998.

[90] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, December 1999.

[91] Jens Lang and Gudula Rünger. High-Resolution power profiling of GPU functions using low-resolution measurement. In *Proceedings of the 19th international conference on Parallel Processing*, Euro-Par'13, pages 801–812, Berlin, Heidelberg, 2013. Springer-Verlag.

[92] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[93] Jongwon Lee, Yohan Ko, Kyoungwoo Lee, Jonghee M. Youn, and Yunheung Paek. Dynamic Code Duplication with Vulnerability Awareness for Soft Error Detection on VLIW Architectures. *ACM Trans. Archit. Code Optim.*, 9(4):48:1–48:24, January 2013.

[94] Seyong Lee, Dong Li, and Jeffrey S Vetter. Interactive Program Debugging and Optimization for Directive-Based, Efficient GPU Computing. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium.* IEEE, 2014.

[95] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[96] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265–276, New York, NY, USA, 2008. ACM.

[97] Michael D. Linderman, James Balfour, Teresa H. Meng, and William J. Dally. Embracing Heterogeneity: Parallel Programming for Changing Hardware. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 3–3, Berkeley, CA, USA, 2009. USENIX Association.

[98] Cong Liu, Jian Li, Wei Huang, Juan Rubio, Evan Speight, and Xiaozhu Lin. Power-efficient Time-sensitive Mapping in Heterogeneous Systems. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 23–32, New York, NY, USA, 2012. ACM.

[99] Michele Lombardi and Michela Milano. Allocation and Scheduling of Conditional Task Graphs. *Artificial Intelligence*, 174(7-8):500–529, may 2010.

[100] ARM Ltd. *big.LITTLE Technology: The Future of Mobile*, 2013.

[101] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[102] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures. In *Proceedings of the 2012 41st International Conference on Parallel Processing*, ICPP '12, pages 48–57, Washington, DC, USA, 2012. IEEE Computer Society.

[103] Alberto Magni, Dominik Grewe, and Nick Johnson. Input-aware Auto-tuning for Directive-based GPU Programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 66–75, New York, NY, USA, 2013. ACM.

[104] Gabriel Martinez, Mark Gardner, and Wu-chun Feng. CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 300 –307, dec. 2011.

[105] Giuseppe Massari, Chiara Caffarri, Patrick Bellasi, and William Fornaciari. Extending a Run-time Resource Management Framework to Support OpenCL and Heterogeneous Systems. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '14, pages 21:21–21:26, New York, NY, USA, 2014. ACM.

[106] Michael Matz1, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface, AMD64 Architecture Processor Supplement*, 2006.

[107] Mojtaba Mehrara, Mona Attariyan, Smitha Shyam, Kypros Constantinides, Valeria Bertacco, and Todd Austin. Low-cost Protection for SER Upsets and Silicon Defects. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1146–1151, San Jose, CA, USA, 2007. EDA Consortium.

[108] Perhaad Mistry, Yash Ukidave, Dana Schaa, and David Kaeli. Valar: A Benchmark Suite to Study the Dynamic Behavior of Heterogeneous Systems. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 54–65, New York, NY, USA, 2013. ACM.

[109] Subhasish Mitra. Globally Optimized Robust Systems to Overcome Scaled CMOS Reliability Challenges. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 941–946, New York, NY, USA, 2008. ACM.

[110] C. Muller, S. Frey, M. Strengert, C. Dachsbacher, and T. Ertl. A Compute Unified System Architecture for Graphics Clusters Incorporating Data Locality. *Visualization and Computer Graphics, IEEE Transactions on*, 15(4):605 –617, july-aug. 2009.

[111] Satish Narayanasamy, Ayse K. Coskun, and Brad Calder. Transient Fault Prediction Based on Anomalies in Processor Events. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '07, pages 1140–1145, San Jose, CA, USA, 2007. EDA Consortium.

[112] Fabian Nowak, Rainer Buchty, and Mario Kicherer. Providing Guidance Information for Application-Mapping on Heterogeneous Parallel Systems. In *Parallel-Algorithmen und Rechnerstrukturen*, volume 26 of *Mitteilungen*, pages 115–122. Gesellschaft für Informatik e.V., December 2009.

[113] Fabian Nowak, Mario Kicherer, Rainer Buchty, and Wolfgang Karl. Delivering guidance information in heterogeneous systems. In Michael Beigl and Franciso J. Cyzorla-Almeida, editors, *ARCS 2010 Workshop Proceedings*, pages 95–101. VDE, February 2010.

[114] Nvidia. *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. v1.0.

[115] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[116] Tetsuya Odajima, Taisuke Boku, Mitsuhisa Sato, Toshihiro Hanawa, Yuetsu Kodama, Raymond Namyst, Samuel Thibault, and Olivier Aumage. Adaptive Task Size Control

on High Level Programming for GPU/CPU Work Sharing. In Rocco Aversa, Joanna Kołodziej, Jun Zhang, Flora Amato, and Giancarlo Fortino, editors, *Algorithms and Architectures for Parallel Processing*, volume 8286 of *Lecture Notes in Computer Science*, pages 59–68. Springer International Publishing, 2013.

[117] H. Orsila, E. Salminen, and T.D. Hamalainen. Parameterizing simulated annealing for distributing kahn process networks on multiprocessor socs. In *System-on-Chip, 2009. SOC 2009. International Symposium on*, pages 019–026, Oct 2009.

[118] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 33–42, New York, NY, USA, 2012. ACM.

[119] Prasanna Pandit and R. Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, New York, NY, USA, 2014. ACM.

[120] E. Perahia, Carlos Cordeiro, Minyoung Park, and L.L. Yang. IEEE 802.11ad: Defining the Next Generation Multi-Gbps Wi-Fi. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–5, Jan 2010.

[121] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. MDR: performance model driven runtime for heterogeneous parallel platforms. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 225–234, New York, NY, USA, 2011. ACM.

[122] Artur Podobas, Mats Brorsson, and Vladimir Vlassov. Exploring heterogeneous scheduling using the task-centric programming model. In *Proceedings of the 18th international conference on Parallel processing workshops*, Euro-Par'12, pages 133–144, Berlin, Heidelberg, 2013. Springer-Verlag.

[123] P. Racunas, K. Constantinides, S. Manne, and S.S. Mukherjee. Perturbation-based Fault Screening. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 169–180, Feb 2007.

[124] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual Use of Superscalar Datapath for Transient-fault Detection and Recovery. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 214–224, Washington, DC, USA, 2001. IEEE Computer Society.

[125] M. Rebaudengo, M.S. Reorda, M. Violante, and Marco Torchiano. A source-to-source compiler for generating dependable software. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 33–42, 2001.

[126] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 25–36, New York, NY, USA, 2000. ACM.

[127] Ruymán Reyes, Iván López, Juan J. Fumero, and Francisco de Sande. accULL: an OpenACC implementation with CUDA and OpenCL support. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 871–882, Berlin, Heidelberg, 2012. Springer-Verlag.

[128] Ruymán Reyes, Iván López, Juan J. Fumero, and Francisco de Sande. A preliminary evaluation of OpenACC implementations. *J. Supercomput.*, 65(3):1063–1075, September 2013.

[129] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM.

[130] E. Rotem, A Naveh, D. Rajwan, A Ananthakrishnan, and E. Weissmann. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *Micro, IEEE*, 32(2):20–27, March 2012.

[131] H.G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *Computers and Digital Techniques, IEE Proceedings* -, 141(1):1–10, Jan 1994.

[132] D. Sabena, M. Sonza Reorda, L. Sterpone, P. Rech, and L. Carro. On the evaluation of soft-errors detection techniques for GPGPUs. In *Design and Test Symposium (IDT), 2013 8th International*, pages 1–6, Dec 2013.

[133] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. Scaling Large-Data Computations on Multi-GPU Accelerators. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 443–454, New York, NY, USA, 2013. ACM.

[134] L.M. Sanchez, F. Fernandez, R. Sotomayor, and J.D. Garcia. A Comparative Evaluation of Parallel Programming Models for Shared-Memory Architectures. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 363–370, July 2012.

[135] Martin Sandrieser, Siegfried Benkner, and Sabri Pllana. Improving programmability of heterogeneous many-core systems via explicit platform descriptions. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 17–24, New York, NY, USA, 2011. ACM.

[136] Ute Schiffel, André Schmitt, Martin Süsskraut, Stefan Weigert, and Christof Fetzer. Parallelizing Software-Implemented Error Detection. In *Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '09, pages 215–226, Berlin, Heidelberg, 2009. Springer-Verlag.

[137] Dirk Schmidl, Christian Terboven, Dieter an Mey, and Martin Bücker. Binding nested OpenMP programs on hierarchical memory architectures. In *Proceedings of the 6th international conference on Beyond Loop Level Parallelism in OpenMP: accelerators, Tasking and more*, IWOMP'10, pages 29–42, Berlin, Heidelberg, 2010. Springer-Verlag.

[138] Jie Shen, Ana Lucia Varbanescu, Henk Sips, Michael Arntzen, and Dick G. Simons. Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 14:1–14:10, New York, NY, USA, 2013. ACM.

[139] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[140] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A Language and Runtime System for Perpetual

Systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 161–174, New York, NY, USA, 2007. ACM.

[141] Kyle Spafford, Jeremy Meredith, and Jeffrey Vetter. Maestro: Data Orchestration and Tuning for OpenCL devices. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 275–286, Berlin, Heidelberg, 2010. Springer-Verlag.

[142] Kyle L. Spafford, Jeremy S. Meredith, Seyong Lee, Dong Li, Philip C. Roth, and Jeffrey S. Vetter. The Tradeoffs of Fused Memory Hierarchies in Heterogeneous Computing Architectures. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 103–112, New York, NY, USA, 2012. ACM.

[143] John A. Stratton, Christopher Rodrigrues, I-Jui Sung, Nady Obeid, Liwen Chang, Geng Liu, and Wen-Mei W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, March 2012.

[144] Toshio Suganuma, Rajaram B. Krishnamurthy, Moriyoshi Ohara, and Toshio Nakatani. Scaling Analytics Applications with OpenCL for Loosely Coupled Heterogeneous Clusters. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, pages 35:1–35:10, New York, NY, USA, 2013. ACM.

[145] Enqiang Sun, Dana Schaa, Richard Bagley, Norman Rubin, and David Kaeli. Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 84–93, New York, NY, USA, 2012. ACM.

[146] Oussama Tahan and Mohamed Shawky. Using dynamic task level redundancy for OpenMP fault tolerance. In *Proceedings of the 25th international conference on Architecture of Computing Systems*, ARCS'12, pages 25–36, Berlin, Heidelberg, 2012. Springer-Verlag.

[147] Hiroyuki Takizawa, Katsuto Sato, and Hiroaki Kobayashi. SPRAT: Runtime processor selection for energy-aware computing. In *Proc. IEEE Int'l. Conf. Cluster Computing (CLUSTER)*, pages 386–393. IEEE, 2008.

[148] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCUDA: A Checkpoint/Restart Tool for CUDA Applications. In *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '09, pages 408–413, Washington, DC, USA, 2009. IEEE Computer Society.

[149] George Teodoro, Timothy D. R. Hartley, Umit Catalyurek, and Renato Ferreira. Run-time optimizations for replicated dataflows on heterogeneous environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 13–24, New York, NY, USA, 2010. ACM.

[150] Samuel Thibault, François Broquedis, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. An Efficient OpenMP Runtime System for Hierarchical Architectures. In *Proceedings of the 3rd international workshop on OpenMP: A Practical Programming Model for the Multi-Core Era*, IWOMP '07, pages 161–172, Berlin, Heidelberg, 2008. Springer-Verlag.

[151] Xiang Tian and Khaled Benkrid. Mersenne Twister Random Number Generation on FPGA, CPU and GPU. In *Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*, AHS '09, pages 460–464, Washington, DC, USA, 2009. IEEE Computer Society.

[152] Martin Tillenius. SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization. Technical Report 2014-010, Uppsala University, Division of Scientific Computing, 2014.

[153] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task Scheduling Algorithms for Heterogeneous Processors. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, pages 3–, Washington, DC, USA, 1999. IEEE Computer Society.

[154] Haluk Topcuouglu, Salim Hariri, and Min-you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, March 2002.

[155] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

[156] Xavier Vera, Jaume Abella, Javier Carretero, and Antonio González. Selective Replication: A Lightweight Technique for Soft Errors. *ACM Trans. Comput. Syst.*, 27(4):8:1–8:30, January 2010.

[157] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[158] Hao Wang, Vijay Sathish, Ripudaman Singh, Michael J. Schulte, and Nam Sung Kim. Workload and Power Budget Partitioning for Single-chip Heterogeneous Processors. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 401–410, New York, NY, USA, 2012. ACM.

[159] Nicholas J. Wang and Sanjay J. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Trans. Dependable Secur. Comput.*, 3(3):188–201, July 2006.

[160] Perry H. Wang, Jamison D. Collins, Gautham N. Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y. Yang, Guei-Yuan Lueh, and Hong Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.

[161] Ute Wappler and Martin Müller. Software Protection Mechanisms for Dependable Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 947–952, New York, NY, USA, 2008. ACM.

[162] V.M. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 215–224, April 2013.

[163] Rick Weber, Akila Gothandaraman, Robert J. Hinde, and Gregory D. Peterson. Comparing Hardware Accelerators in Scientific Applications: A Case Study. *IEEE Transactions on Parallel and Distributed Systems*, 22:58–68, 2011.

[164] John Robert Wernsing and Greg Stitt. Elastic Computing: a Framework for Transparent, Portable, and Adaptive Multi-core Heterogeneous Computing. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, LCTES '10, pages 115–124, New York, NY, USA, 2010. ACM.

[165] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[166] Cemal Yilmaz and Adam Porter. Combining Hardware and Software Instrumentation to Classify Program Executions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 67–76, New York, NY, USA, 2010. ACM.

[167] Bobby Dalton Young, Sudeep Pasricha, Anthony A. Maciejewski, Howard Jay Siegel, and James T. Smith. Heterogeneous Makespan and Energy-constrained DAG Scheduling. In *Proceedings of the 2013 Workshop on Energy Efficient High Performance Parallel and Distributed Computing*, EEHPDC '13, pages 3–12, New York, NY, USA, 2013. ACM.

[168] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264–266, Feb 2011.

[169] Lei Zhang, Yinhe Han, Qiang Xu, and Xiaowei Li. Defect Tolerance in Homogeneous Manycore Processors Using Core-level Redundancy with Unified Topology. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 891–896, New York, NY, USA, 2008. ACM.

Appendix

A. Case study: Preprocessing for medical imaging

The Matlab code for the preprocessing stage of the image reconstruction as described in Chapter 7.5.1 without the code for loading and storing the data:

```
if ~ exist('DLS_MATLAB_INIT')
   % add implementations only once
   DLS_MATLAB_INIT=1;

   dls('add_impl', 'doTransformT2F', 'doTransform_T2F')
   dls('add_impl', 'doTransformF2T', 'doTransform_F2T')

   dls('add_impl', 'doDeOffset_FD', 'doDeOffsetFDinFDout_CPU_OMP')
   dls('add_impl', 'doDeOffset_FD', 'doDeOffsetFDinFDout_GPU_CUDA')

   dls('add_impl', 'doEnvelopeGeneration_FD', \
       'doEnvelopeGenerationFDinFDout_CPU_OMP')
   dls('add_impl', 'doEnvelopeGeneration_FD', \
       'doEnvelopeGenerationFDinFDout_GPU_CUDA')

   dls('add_impl', 'doMatchedFiltering_FD', \
       'doMatchedFilteringFDinFDout_CPU_OMP')
   dls('add_impl', 'doMatchedFiltering_FD', \
       'doMatchedFilteringFDinFDout_GPU_CUDA')

   dls('add_impl', 'doDeOffset_TD', 'doDeOffsetTDinTDout_CPU_OMP')
   dls('add_impl', 'doDeOffset_TD', 'doDeOffsetTDinTDout_GPU_CUDA')

   dls('add_impl', 'doEnvelopeGeneration_TD', \
       'doEnvelopeGenerationTDinTDout_CPU_OMP')
   dls('add_impl', 'doEnvelopeGeneration_TD', \
       'doEnvelopeGenerationTDinTDout_GPU_CUDA')

   dls('add_impl', 'doMatchedFiltering_TD', \
       'doMatchedFilteringTDinTDout_CPU_OMP')
   dls('add_impl', 'doMatchedFiltering_TD', \
       'doMatchedFilteringTDinTDout_GPU_CUDA')

   dls('add_impl', 'doPeakDetection_TD', \
       'doPeakDetectionTDinTDout_CPU_OMP')
   dls('add_impl', 'doPeakDetection_TD', \
```

```
        'doPeakDetectionTDinTDout_GPU_CUDA')

    dls('add_impl', 'doOptPulseConv_TD', 'doOptPulseConvTDinTDout_CPU_OMP')
    dls('add_impl', 'doOptPulseConv_TD', 'doOptPulseConvTDinTDout_GPU_CUDA')
end

% --> loading input data <--

% registering data at the runtime system, returns references
data_A = dls('reg_data', ascans_A);
data_B = dls('reg_data', ascans_B);
mf_fdptr = dls('reg_data', mf_fd);
mf_tdptr = dls('reg_data', mf_td);
hilptr = dls('reg_data', hil);
op_fdptr = dls('reg_data', op_fd);
op_tdptr = dls('reg_data', op_td);
fftparamsptr = dls('reg_data', fftparams);

% create a switch as first task
switch_task = dls('create_switch');

%%%%%%%%%%%%
%%% FD branch

% submit a task "doTransformT2F" with:
%  - no return value,
%  - read access on data_A,
%  - write access on data_B,
%  - three regular parameters that are passed unmodified to
%    the implementation
%  - and append it to a new branch in switch_task
f = dls('submit_task', 'doTransformT2F', 0, 'rwvvva', \
    data_A, data_B, mf_fdptr, fftparamsptr, 0, switch_task);

f1 = dls('submit_task', 'doDeOffset_FD', 0, 'rwa', data_B, \
    data_A, f);

f2 = dls('submit_task', 'doMatchedFiltering_FD', 0, 'rrwa', \
    data_A, mf_fdptr, data_B, f1);

f3 = dls('submit_task', 'doEnvelopeGeneration_FD', 0, \
    'rrwra', data_B, hilptr, data_A, fftparamsptr, f2);

dls('submit_task', 'doTransformF2T', 0, 'rwvvva', data_A, \
    data_B, mf_fdptr, fftparamsptr, 1, f3)

%%%%%%%%%%%%
%%% TD branch

t1 = dls('submit_task', 'doDeOffset_TD', 0, 'rwa', data_A, \
    data_B, switch_task);

t2 = dls('submit_task', 'doMatchedFiltering_TD', 0, 'rrwa', \
    data_B, mf_tdptr, data_A, t1);

t3 = dls('submit_task', 'doEnvelopeGeneration_TD', 0, \
    'rrwra', data_A, hilptr, data_B, fftparamsptr, t2);

%%%%%%%%%%%%
%%% common

dls('submit_task', 'doPeakDetection_TD', 0, 'rvw', data_B, \
    2, data_A)
```

```
dls('submit_task', 'doOptPulseConv_TD', 0, 'rrw', data_A, \
   op_tdptr, data_B)

% calculate a task graph from submitted tasks
tgraph = dls('calc_tgraph');

% start execution
dls('exec_tgraph', tgraph)

% --> store/forward data <--

% free allocated data
dls('unreg_data', ascansptr, mf_fdptr, mf_tdptr, hilptr, \
   op_fdptr, op_tdptr, preprocessDataptr, fftparamsptr)
```