Hybrid Approaches towards Adaptive Heterogeneous Computing Platforms for Exchangeable High-Demanding Workloads

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN (Dr.-Ing.)

von der KIT-Fakultät für Elektrotechnik und Informationstechnik des Karlsruher Instituts für Technologie (KIT)

angenommene

DISSERTATION

von

Fabian Lesniak, M.Sc.
geb. in Donaueschingen

Tag der mündlichen Prüfung: 13. Februar 2026

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr. Andreas Herkersdorf

Hybrid Approaches towards Adaptive Heterogeneous Computing Platforms for Exchangeable High-Demanding Workloads

First edition: September 13, 2026
Fabian Lesniak

Abstract

The entire history of computer technology has been characterized by one persistent objective: the continuous enhancement of computational performance. This trend is currently driven by Machine Learning (ML) algorithms, whose immense computational demands push modern systems to their operational limits. Traditional methods for performance scaling, such as increasing clock frequency and shrinking transistors, are increasingly reaching physical limitations, making alternative approaches necessary. Alongside microarchitectural improvements and spatial parallelism, there is a growing trend towards highly specialized computing systems. These systems feature application-specific extensions and optimizations, such as instruction-set extensions or discrete hardware accelerators. This specialization, however, comes at a cost: the system may not be suitable for algorithms that were unknown at design time. In the field of neural networks, for instance, novel models and methods are developed continuously, and hardware development constantly lags behind the state of the art due to the long hardware design cycles. Consequently, the very specialization that is essential to achieve the required performance can cause a computing platform to become obsolete quickly. The challenge for computer architects is to find a balance between specialization and flexibility to maximize the operational lifetime of computing platforms.

FPGAs represent one approach to achieve greater flexibility compared to inherently immutable ICs. They allow the reconfiguration of their implemented logic functions, making them adaptable to new use cases. However, they are inferior to ASICs in terms of maximum clock frequency and area efficiency. This raises a critical question: How can the advantages of ASICs–high performance and energy efficiency–be combined with the adaptability of FPGAs? To answer that question, this thesis investigates hybrid architectures that combine fixed logic with FPGAs. In contrast to existing FPGA-SoCs, which feature large, monolithic processor and FPGA blocks, this work proposes a fine-grained interleave of ASIC and FPGA logic. Components such as processors and memories are implemented as fixed logic for maximum performance. Conversely, FPGA fabric is used for components that are less performance-critical but significantly influence the system's behavior. This approach provides the performance of an ASIC in the datapath while retaining the flexibility of an FPGA for defining the algorithm.

Architectures based on this hybrid approach are divided into two categories: primarily control-flow-dominant systems and dataflow-driven systems. A methodology based on the Roofline Model is used to estimate whether an existing algorithm predominantly focuses on control-flow or dataflow. For control-flow-dominant applications, run-time reconfigurable processors are investigated. These combine a processor core implemented in hard logic with FPGA units for special instructions (SIs). The application benefits from both high processor pipeline throughput and hardware acceleration of compute-intensive operations. Using the *i*-Core as an example, the application of such a processor for Near-Memory Computing (NMC) is examined. Here, the reconfigurable units benefit from a discrete, high-bandwidth memory interface, which allows for faster execution of memory-intensive operations. Furthermore, approximate computing is analyzed as an optimization method for reconfigurable processors. Certain algorithms, particularly in media processing and ML, can tolerate a certain degree of inaccuracies in their calculations while still delivering an acceptable result. It is shown that this method is particularly effective here, enabling a 75 % reduction in hardware resources with a reduction of accuracy of only 3.24 %.

For data-flow-driven applications, an accelerator architecture based on a Network-on-Chip (NoC) is presented. A heterogeneous, multi-level memory hierarchy provides the bandwidth required for modern algorithms like ML. The memory system, the NoC, and the host computer interface are realized as fixed logic. The accelerator tiles, however, are implemented using embedded FPGAs, allowing them to be reconfigured at run time. This partitioning into ASIC and FPGA domains achieves high memory bandwidth while allowing the algorithm in the accelerator tiles to be flexibly adapted. As an example application, accelerators for Convolutional Neural Networks (CNNs) are presented and implemented in the tiles. Individual tiles achieve a throughput of 6.28 GOp/s, corresponding to an efficiency of 89.7 %. The high throughput of the memory hierarchy enables nearly linear performance scaling across multiple tiles. For instance, with 20 tiles in a 5 × 5 NoC, the throughput can be increased by a factor of 10.3 × .

Zusammenfassung

Die gesamte Entwicklungsgeschichte der Computertechnologie ist von einem immerwährenden Ziel geprägt: die kontinuierliche Steigerung der Rechenleistung. Aktuell prägen Algorithmen für Machine Learning (ML) diesen Trend, denn durch die enorme Menge nötiger Rechenoperationen treiben sie modernste Systeme an ihre Grenzen. Klassische Methoden zur Leistungssteigerung wie die Verkleinerung der Strukturgröße und Erhöhung der Taktfrequenz stoßen zunehmend an physikalische Grenzen, weshalb andere Ansätze gefragt sind. Neben Verbesserungen der Mikroarchitektur und räumlicher Parallelisierung findet man zunehmend stark spezialisierte Rechnersysteme. Sie besitzen auf den konkreten Anwendungsfall ausgerichtete Erweiterungen und Optimierungen, wie Befehlssatzerweiterungen oder diskrete Hardwarebeschleuniger. Diese Spezialisierung hat einen Preis: für Algorithmen, die zur Entwicklungszeit nicht bekannt waren, ist das System möglicherweise nicht geeignet. Im Feld der neuronalen Netze werden beispielsweise kontinuierlich neuartige Modelle und Methoden entwickelt und die Hardwareentwicklung liegt aufgrund der langen Entwurfszyklen immer hinter dem Stand der Technik zurück. Somit kann jene Spezialisierung, die zum Erreichen der nötigen Rechenleistung unerlässlich ist, dazu führen, dass eine Rechnerplattform schnell obsolet wird. Die Herausforderung für Computerarchitekten besteht darin, einen Mittelweg von Spezialisierung und Flexibilität zu finden, damit Rechnerplattformen möglichst lange eingesetzt werden können.

FPGAs sind ein Ansatz für erhöhte Flexibilität im Vergleich zu inhärent unveränderlichen ICs. Sie erlauben eine Rekonfiguration der implementierten logischen Schaltfunktion und sind dadurch an neue Anwendungsfälle anpassbar. Allerdings sind sie ASICs bei maximaler Taktfrequenz und nötiger Fläche unterlegen. Somit stellt sich die Frage: Wie lassen sich die Vorteile von ASICs, hohe Performanz und Energieeffizienz, mit der Anpassbarkeit von FPGAs verknüpfen? Als Antwort auf diese Frage werden in dieser Arbeit hybride Architekturen untersucht, die feste Logik und FPGAs kombinieren. Im Gegensatz zu bestehenden FPGA-SoCs, die große monolithische Prozessor- und FPGA-Blöcke besitzen, wird ASIC- und FPGA-Logik feinmaschig verwoben. Komponenten wie Prozessoren und Speicher werden für maximale Performanz als feste Logik implementiert. FPGA-Bausteine werden hingegen für Komponenten verwendet, die weniger leistungskritisch sind, aber das Verhalten maßgeblich bestimmen. Dadurch erhalten wir die Performanz eines ASIC im Datenpfad, aber die Flexibilität eines FPGA für die Definition der Algorithmik.

Architekturen nach diesem hybriden Ansatz werden in zwei Kategorien unterteilt: primär kontrollflussbasierte und datenflussgetriebene Systeme. Mittels einer Methodik basierend auf dem Roofline-Modell kann für bestehende Algorithmen abgeschätzt werden, ob der Fokus auf Kontroll- oder Datenfluss liegt. Für kontrollflussdominante Anwendungen werden rekonfigurierbare Prozessoren untersucht. Sie kombinieren einen Prozessorkern, implementiert als harte Logik, mit FPGA-Einheiten für Spezialinstruktionen. Die Applikation profitiert von einem hohen Durchsatz der Prozessorpipeline sowie von Hardwarebeschleunigung rechenintensiver Operationen. Am Beispiel des *i*-Core wird die Anwendung eines solchen Prozessors für Near-Memory Computing (NMC) untersucht. Hier profitieren die rekonfigurierbaren Einheiten von einer diskreten Speicheranbindung mit besonders hoher Bandbreite, wodurch speicherintensive Operationen schneller abgearbeitet werden können. Zudem wird approximiertes Rechnen als Optimierungsmethode für rekonfigurierbare Prozessoren analysiert. Einige Algorithmen, insb. für Bild- und Videoverarbeitung sowie ML, können eine gewisse Ungenauigkeit in ihrer Berechnung tolerieren und liefern dennoch ein akzeptables Ergebnis. Es kann gezeigt werden, dass diese Methode hier besonders effektiv ist und 75 % der benötigten Hardwareressourcen gespart werden können, während die Genauigkeit um lediglich 3.24 % verringert wird.

Für datenflussgetriebene Anwendungen wird eine Beschleunigerarchitektur auf Basis eines Network-on-Chip (NoC) vorgestellt. Eine heterogene, mehrstufige Speicherhierarchie stellt die benötigten Kapazitäten für moderne Algorithmen wie ML zur Verfügung. Hierbei sind die Speicher, das NoC und die Anbindung an den Host-Computer als feste Logik realisiert. Die Beschleunigerkacheln sind hingegen als eingebettete FPGAs implementiert, wodurch sie zur Laufzeit rekonfiguriert werden können. Durch diese Aufteilung in ASIC- und FPGA-Partitionen wird eine hohe Speicherbandbreite erreicht, während die Algorithmik in den Beschleunigerkacheln flexibel angepasst werden kann. Als Beispielanwendung werden Beschleuniger für Convolutional Neural Networks (CNNs) vorgestellt und in den Kacheln implementiert. Einzelne Kacheln erreichen einen Durchsatz von 6.28 GOp/s, was einer Effizienz von 89.7 % entspricht. Der hohe Durchsatz der Speicherhierarchie ermöglicht eine beinahe lineare Skalierung der Performanz über mehrere Kacheln hinweg. So kann der Durchsatz bei 20 Kacheln in einem 5×5 NoC um einen Faktor von $10.3\times$ gesteigert werden.

Contents

Acronyms

ACC	Adaptive Cruise Control.
AES	Advanced Encryption Standard.
AI	Artificial Intelligence.
ALP	Adaptive Logic Partition.
ALU	Arithmetic Logic Unit.
ANN	Artificial Neural Network.
ASAP	Adaptive Scalable Accelerator Platform.
ASIC	Application-Specific Integrated Circuit.
ASIP	Application-Specific Instruction-Set Processor.
AXI	Advanced eXtensible Interface.
BAR	Base Address Register.
BE	best effort.
BRAM	Block RAM.
CA	Collision Avoidance.
CDC	Clock Domain Crossing.
CGRA	Coarse-Grained Reconfigurable Architecture.
CLB	Configurable Logic Block.
CNN	Convolutional Neural Network.
CONV	Convolutional.
COTS	Commercial-Off-the-Shelf.
CPLD	Complex Programmable Logic Device.
CPU	Central Processing Unit.
CU	Compute Unit.
DFG	Data Flow Graph.
DLA	Deep Learning Accelerator.
DMA	Direct Memory Access.
DNN	Deep Neural Network.

DRAM Dynamic Random Access Memory.
DSE Design Space Exploration.
DSP Digital Signal Processor.
DUT Design Under Test.

ECU Embedded Computing Unit.
EDA Electronic Design Automation.

FF Flipflop.
FFT Fast Fourier Transform.
FLIT Flow Control Unit.
FLOP Floating-Point Operation.
FLP Fixed Logic Partition.
FPGA Field Programmable Gate Array.
FPU Floating Point Unit.
FSM Finite-State Machine.
FTL flash translation layer.

GEMM General Matrix Multiplication.
GPP General Purpose Processor.
GPU Graphics Processing Unit.
GS guaranteed service.

HBM High-Bandwidth Memory.
HDL Hardware Description Language.
HLS High Level Synthesis.
HMC Hybrid Memory Cube.
HPC High-Performance Computing.

IDC Inter-Domain Communication.
ifmap input feature map.
ILP Instruction-level Parallelism.
IMC In-Memory Computing.
IP Intellectual Property.
IPC Inter-Process Communication.
IPI Inter-Processor Interrupts.

ISA Instruction Set Architecture.
ISP Image Signal Processor.

LLM Large Language Model.
LSB least significant bit.
LSTM Long Short-Term Memory.
LUT Lookup Table.

MAC Multiply-Accumulate.
MCM Multi-Chip Module.
ML Machine Learning.
MLP Multilayer Perceptron.
MMIO Memory Mapped I/O.
MPSoC Multiprocessor System-on-Chip.
MRAM Magnetoresistive RAM.
MSE Mean Squared Error.

NA Network Adapter.
NMA Near-Memory Accelerator.
NMC Near-Memory Computing.
NMCC Near-Memory Computing Controller.
NoC Network-on-Chip.
NPU Neural Processing Unit.
NRE Non-Recurring Engineering.

ofmap output feature map.
OI Operational Intensity.
ONNX Open Neural Network Exchange.
OS Output Stationary.

PCIe PCI Express.
PCM Phase-change Memory.
PE Processing Element.
PGAS Partitioned Global Address Space.
PLB Programmable Logic Block.
PPA Performance, Power and Area.

PTQ Post-Training Quantization.

QAT Quantization Aware Training.

RAM Random Access Memory.
RC reconfigurable container.
RNN Recurrent Neural Network.
RPC remote procedure call.
RPMsg Remote Processor Messaging.
RRAM Resistive RAM.
RS Row-Stationary.
RTCA Real-Time Communication Architecture.
RTL Register Transfer Level.
RTOS real-time operating system.
RX receive.

SAD sum of absolute differences.
SB Switch Box.
SDRAM Synchronous Dynamic RAM.
SERDES Serializer/Deserializer.
SHM Shared Memory.
SI special instruction.
SIMB Single Instruction Multiple Bank.
SIMD Single Instruction Multiple Data.
SISD Single Instruction Single Data.
SM Streaming Multiprocessor.
SoC System-on-Chip.
SOP Service-Oriented Protocol.
SPM Scratchpad Memory.
SQL Structured Query Language.
SR-IOV Single-Root I/O-Virtualization.
SRAM Static Random Access Memory.
SRS Spatial Row-Stationary.
SSD Solid State Drive.

TPU Tensor Processing Unit.

TRS Temporal Row-Stationary.
TX transmit.

UCIe Universal Chiplet Interconnect Express.
USAN Univalue Segmented Area Nucleus.

VC virtual channel.
VIRTIO Virtual I/O Device.
VLCW Very Large Configuration Word.
VLIW Very Long Instruction Word.
VOS voltage overscaling.
VPU Vector Processing Unit.

WS Weight Stationary.

Chapter 1

Introduction

For over a century, the evolution of computing has been governed by one consistent trend: the ceaseless demand for greater processing power to enable increasingly sophisticated applications. Over the course of this period, growing demands for performance and energy efficiency of processors could be met solely through miniaturization. Driven by tremendous efforts of numerous computer engineers, computers shrank from warehouse-scale machines to portable devices. At the same time their performance has been increased by many orders of magnitude. Dennard Scaling and Moore's Law served as models of this process, guaranteeing a steady increase of those performance and miniaturization figures by reducing the transistor size. However, over the last 20 years it became patently clear that neither of these principles still applies to the modern development of semiconductor technology. As a consequence, engineers gradually shifted their focus from miniaturization to architectural improvements. Prominent examples include multicore, deep pipelining and speculative execution. In addition, specialized architectures tailored to their target application, have been developed to address this problem.

Such custom hardware excels in performance and energy efficiency, yet specialization always comes at a price: On the one hand, the costs of developing and producing an Application-Specific Integrated Circuit (ASIC) are extremely high. On the other hand, this chip is hardly suitable for any other application due to its application-specific nature. However, some examples originating from specialization found their way into General Purpose Processors (GPPs), e.g. the execution of Single Instruction Multiple Data (SIMD) instructions from the MMX, SSE and AVX extensions. As some applications benefit from such specializations in GPPs, they were adopted widely by many processors as of today. However, if applications do not use them at all, large areas of silicon required for these extensions remain unused. The dark silicon effect

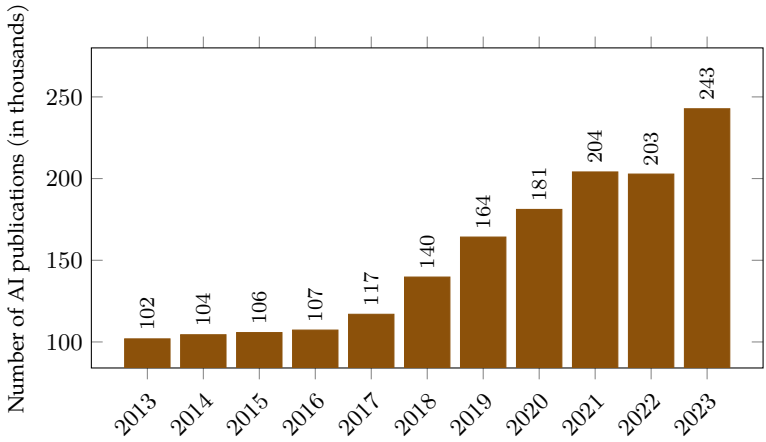


Figure 1.1: Number of worldwide Artificial Intelligence (AI) publications in computer science from 2013 to 2023. Data from [2].

also contributes to this, as the power budget, especially in the form of heat that can be dissipated by the silicon die, limits simultaneous operation of all implemented components. This also restrains engineers from further downsizing and scaling of the number of processors further [1]. Consequently, novel solutions are required to satisfy the computational requirements of modern workloads.

1.1 Motivation and Scope

To set the challenges put on today's systems into perspective, one can take a look at two prominent, demanding example applications of this time: AI and Big Data processing. The relevance of AI as a main driver for compute is illustrated by Figure 1.1, which shows the steady increase in scientific interest in AI, as the number of publications in the field has been constantly increasing over the last decade. Typically, all kinds of AI algorithms are implemented in the form of neural networks. Their complexity is growing excessively and has even leaped with the introduction of large-scale foundation models. These multi-modal models are very popular due to their broad applicability, but pose enormous challenges for computers due to their massive size of over 1 trillion parameters [3]. To run one of these huge models, which is referred to

as inference, a computer has to apply these billions of parameters to inputs and saved states. This easily adds up to over 1 trillion operations that have to be performed. Hence, computing them pushes many architectures to their limits, not only due to their vast number of operations, but also due to their high memory bandwidth requirements. Big Data processing is another prime example of a computationally demanding problem. Today, vast quantities of structured and unstructured data are collected from diverse sources. Often, the data is stored speculatively without a predefined purpose. This data only becomes relevant later for a wide range of applications, including search engines, business planning, and research in the human sciences. While the initial data acquisition is relatively straightforward, executing queries on often distributed, warehouse-scale data storage systems is computationally intensive and consumes significant amounts of energy [4]. Consequently, although application-specific hardware acceleration is being explored, the diverse and evolving nature of Big Data workloads complicates the development of universal processing solutions.

The rapid pace of algorithm development is fundamentally opposed to the long design cycles of integrated circuits, which involve extensive architecture design, physical layout, verification, and fabrication. For reference, software developers can deploy changes to algorithms within the range of a few months, while hardware architects need at least a year to create a tested and verified hardware architecture. This does not include the time needed to actually manufacture the chip itself, which can take up to eight months for recent process nodes [5]. Therefore, while software developers can adapt quickly to changing requirements, hardware designers must anticipate application needs years in advance. Nowadays, specialization is essential for performance, yet overly specialized hardware risks rapid obsolescence when applications evolve. As a result we are left with an important design decision: How specialized should the integrated circuit under design be? In this thesis, we address this overarching question in computer engineering by presenting an approach to balance the competing goals of specialization and flexibility. We explore this trade-off from two complementary directions:

1. First, we take a top-down approach starting from a general-purpose processor. Here we integrate Field Programmable Gate Array (FPGA) fabric to enable targeted hardware specialization. Simply speaking, this approach allows adding customized, accelerated instructions when they are needed and beneficial for a given application.

2. Second, we look at bottom-up methodologies, which add more flexibility to specialized dataflow architectures. Similarly, this can be achieved by integration of reconfigurable logic, which allows altering the behavior and data paths after the chip has been hardened in silicon.

Both approaches help to answer the central research question of this thesis: which parts of a computing systems can be implemented using FPGA fabric to increase the versatility of the platform, while maintaining processing capabilities and memory throughput? In this thesis, we will try to answer this fundamental question with contributions that balance flexibility and specialization as described above in mind—once from top to bottom and once bottom-up. Our contributions can be summarized as follows:

Run-time Reconfigurable Processors Our first contribution to these questions is the investigation of Near-Memory Computing (NMC) based on run-time reconfigurable processors. In the context of manycore systems, we place our processor closer to the memory than the remaining GPPs. In addition, the processor has a reconfigurable fabric, which features high-performance memory interfaces. This allows it to perform certain complex tasks significantly faster than the GPPs. With this approach we can unify the strengths of our heterogeneous processing approach. We can implement the computational kernels in the reconfigurable fabric for accelerated processing, and keep control and management tasks in software. In an experimental setup, we could show a 70 % improvement in throughput for an encryption workload. The second contribution analyzes the impact of approximate computing on run-time reconfigurable processors. The limited number of resources available in the reconfigurable fabric constrains the development of special instructions (SIs). Many algorithms for media processing or machine learning can tolerate a certain amount of inaccuracy in their computations, i.e., a small deviation in the result does not change its output. This characteristic can be leveraged to reduce the number of resources required for an SI. As a side effect, performance and energy consumption can be improved as well. Our approximate implementation allows for a 75 % reduction in resource usage with an accuracy reduction of only 3.24 %. The third contribution to the top-down approach improves software deployment on multiprocessor systems. Especially in those with multiple processors like our reconfigurable processor, unintended interference between multiple applications needs to be avoided. A hypervisor

offers isolation features commonly used in safety-critical embedded systems. Applications deployed using a hypervisor still need to communicate, which involves significant overhead when classical methods like emulated Ethernet are used. We present a novel lightweight mechanism for hypervisor guests requiring low-latency communication, outperforming traditional methods by more than 5 × .

Adaptive Accelerators for Dataflow Workloads We present a self-sufficient Convolutional Neural Network (CNN) accelerator based on a systolic array as our first contribution to this subject. Its novel and improved row-stationary dataflow allows for minimal data re-use to improve power consumption, since most energy is used for memory accesses. Thereby, it outperforms the original row-stationary dataflow for most layers of typical models. Additionally, we present an improved scratchpad memory layout to avoid time-consuming data reordering operations. This CNN accelerator serves as an example for our second contribution to this field, the Adaptive Scalable Accelerator Platform (ASAP). Our approach towards such an adaptive platform for accelerators consists memory and accelerator tiles connected by a Network-on-Chip (NoC). The NoC itself, the memories and the interfaces between them, including DMA engines, are implemented as hard logic to offer high performance. In contrast, the accelerator tiles are implemented using reconfigurable fabric to allow changing the architecture during run time. We can show that this hybrid combination of ASIC and FPGA platforms offers sufficient memory bandwidth to process modern Machine Learning (ML) models while enabling reconfiguration to adapt to future workloads. As a final step, we present a lightweight architecture for non-intrusive monitoring of tiled NoC architectures such as our accelerator platform. It can be adapted to heterogeneous designs with multi-level hierarchies and enables monitoring and evaluation without impacting the design under test.

1.2 Outline

In order to be familiar with the computing platforms at hand and have a common understanding of the terms used in the remainder of this thesis, fundamental principles are introduced in Chapter 2. We will cover basic concepts of computer architectures, looking in particular at the types of parallelism and the importance of a computer's

memory hierarchy. These key performance-impacting properties will be extended by scaling options such as networks-on-chip and systolic arrays. Next, we are looking at some of today's parallel computing architectures that utilize these concepts, looking at both processor-based and dataflow architectures. Finally, an introduction to neural networks is given, as these constitute a representative workload example for many of our analyses. We focus on the principles of machine learning and CNNs, for which we further introduce methods for efficient processing. In Chapter 3, we dive into the state of the art of adaptive hardware and parallel architectures. In line with our two-pronged approach, we look at both current concepts for reconfigurable processors and adaptive dataflow architectures. Additionally, current work on the acceleration of neural networks is presented, which provides a context for our proposed designs for dataflow architectures in Chapter 6.

Chapter 4 starts by presenting a method for workload analysis. Based on the Roofline Model, an early assessment is made of which architecture is better suited to a particular algorithm. In the following, the hybrid concept for hardware with both fixed logic and FPGA components is developed. The chapter concludes by applying the concept to processor-based and dataflow architectures in a generic fashion.

In Chapter 5, we take an in-depth look at run-time reconfigurable processors. After introducing the basic architecture used for our studies, the *i*-Core, we discuss utilizing the architecture as an accelerator for NMC. Secondly, we take a closer look at the architecture itself and investigate the use of approximate computing as a method of improving performance and reducing resource usage. Chapter 6 turns attention to monolithic accelerators for dataflow applications. We introduce an adaptive accelerator platform, which hosts reconfigurable accelerator tiles and provides a high performance interconnect and memory interface. Following the base platform, we then present an optimized hardware accelerator for CNNs. Finally, the accelerator platform is then evaluated using multiple tiles hosting our CNN accelerator design. Chapter 7 concludes this thesis by summarizing and discussing the results.

Chapter 2

Basic Principles and Fundamentals

This chapter establishes the fundamental concepts of computer architecture, providing the necessary background for the subsequent discussions. We will review traditional and modern performance enhancement techniques, highlighting the renewed importance of microarchitecture as traditional scaling effects diminish. Figure 2.1 shows the performance scaling of microprocessors throughout the last 40 years. A long phase of consistent growth of 52 % per year visualizes Moore's Law, dictating that the number of transistors doubles every two years. According to the Dennard Scaling rule, the power density of transistors remains constant, which allowed to steadily increase the frequency while shrinking the transistor size. In the following phase from around 2003 to 2011, further scaling was limited, as leakage currents and similar effects increased dramatically with ever smaller structure sizes. As it became increasingly difficult to increase the performance of a single processor core, several cores were combined to form multicore processors. However, the increase in performance for software was limited by the extent to which the algorithms can be parallelized, which is described by Amdahl's Law [6]. Subsequently, microarchitecture improvements became increasingly important. They range from classic methods like pipelining and spatial parallelism to modern approaches for improving scalability and constructing heterogeneous memory hierarchies. Afterward, we will look at several types of computer architectures particularly relevant in contemporary industry and academia, which leverage these optimization methods. Finally, an introduction to neural networks and machine learning is provided, underscoring their role as prominent workloads and laying the groundwork for Chapter 6, which addresses architectures for efficient neural network inference. For further details on many of these topics, *Computer Architecture* by Hennessy, a widely regarded standard reference, offers in-depth coverage. Other relevant references are provided throughout this chapter.

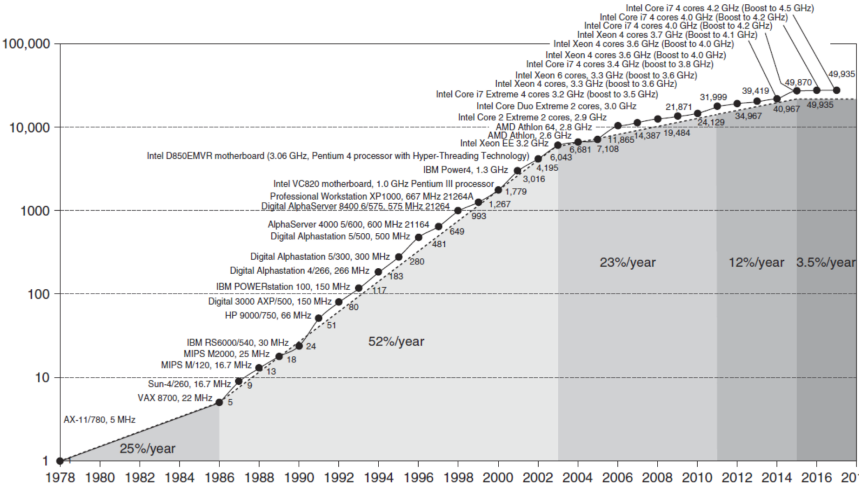


Figure 2.1: Performance scaling of microprocessors over four decades, illustrating initial continuous growth following Moore's Law and Dennard Scaling. After that, performance was still increased but limited through parallelization according to Amdahl's Law. Image from [7].

2.1 Computer Architecture Principles

Architecture design is one of the key disciplines of computer engineering, which transitioned from mechanical and electrical designs to semiconductors in the 70s. Microprocessors are arguably the most prominent category of computer architecture, where all components for computation, control and program execution are integrated on one chip. Historically, processors were designed around principles like the *von Neumann* architecture (shared memory for instructions and data) or the Harvard architecture (separate memory paths) [8]. Figure 2.2 shows a simplified system that builds upon the original von Neumann architecture by introducing a common system bus for transferring instructions, data, and input/output values, representing a clear progression.

Modern processors typically employ a hybrid approach, often utilizing a modified Harvard structure between core and caches for simultaneous instruction fetch and data access while maintaining a unified main memory. Microprocessors, which often take the role of Central Processing Units (CPUs) in a large computing system, are not

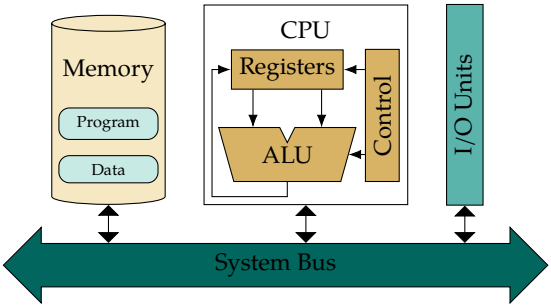


Figure 2.2: Simplified system based on a von Neumann processor, which evolved to an architecture with a common system bus. The memory shares both the program instructions and data used by the program.

the only type of processor commonly seen in today's systems. For example, a Graphics Processing Unit (GPU) is built from many simple processor cores processing multiple data at once. Generally speaking, the architecture and its method to process data are significantly influenced by workload characteristics, which vary in their control-flow and dataflow demands. Control-flow intensive tasks, like interactive applications or recursive algorithms, involve frequent branching, while dataflow intensive tasks, common in scientific computing or multimedia, apply similar operations to large datasets.

The efficiency of a processor's microarchitecture is quantified by metrics such as instructions per cycle or Floating-Point Operations (FLOPs) per second. The latter is often used in scientific computing, where floating point operations are predominantly used and represent the most expensive arithmetic operation. To enhance the performance indicated by these metrics, architects employ numerous techniques, including hardware-level parallelism like pipelining (overlapping instruction execution stages) and superscalar execution (issuing multiple instructions per cycle), as well as software-level parallelism such as multithreading, which allows concurrent execution of different program parts or programs on available processing units.

2.1.1 Temporal & Spatial Parallelism

The initial theoretical framework for processors, executing instructions in a strict step-by-step fashion, was quickly augmented and rapidly evolved through performance-

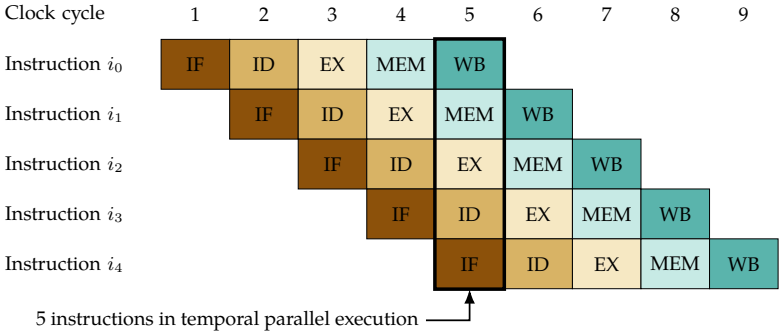


Figure 2.3: A five-stage pipeline in which the execution of five instructions is carried out in parallel, based on the exemplary DLX architecture [8].

driven optimizations. Parallelism is key to achieve high performance in modern computers and can be applied to different components. Hennessy defines four types of parallelism in computer architectures [7]:

1. Instruction-level Parallelism (ILP)
2. Vector architectures like GPUs
3. Thread-level parallelism
4. Request-level parallelism

ILP is very common nowadays and is already highly optimized in most architectures. Vector architectures like GPUs employ the Single Instruction Multiple Data (SIMD) scheme for spatial parallelism. Mechanisms 1 and 2 primarily focus on hardware and can be partially transparent to software. Thread-level parallelism is implemented in tightly coupled multiprocessor systems on data and/or tasks. Request-level parallelism applies the concept to loosely coupled systems like servers in datacenters, which operate independently on individual requests. Since this thesis focuses on single hardware architectures, we will take a closer look on ILP and vector architectures.

Pipelining is one of the first methods applying temporal parallelism on instruction level and can be found in all high-performance processors since 1985 [8]. In a pipelined processor, the execution of an instruction is split into consecutive steps of similar length. This allows the execution of individual instructions to overlap in time, as shown in Figure 2.3. For example, while instruction i_1 is being executed (e.g. an

addition using input operands is performed), the parameters of instruction i_2 are evaluated and instruction i_3 is being fetched from memory in parallel. Theoretically, an N-stage pipeline increases processor throughput by a factor of N; however, additional timing slack can occur in stages that finish their work early and thus under-utilize their cycle time, as pipeline stages need to be of equal size. This effect, paired with additional delays of pipeline registers, slightly increases execution latency of each instruction.

Pipelining can be applied not only to Single Instruction Single Data (SISD) processors, but any control-flow and dataflow architecture. GPUs use hardware-based schedulers to ensure continuous commands being dispatched to the streaming processor pipeline, see Section 2.2.2. Systems with focus on dataflow, like systolic arrays, can be considered as a large pipeline themselves. However, since they are not executing single instructions but rather process a stream of data in a predefined scheme, there is no need for dynamic scheduling and data dependencies do not have to be handled at run time. Refer to Section 2.1.4 for more details on systolic arrays.

In addition to temporal parallelism as leveraged by pipelining, spatial parallelism is also often used in processors. Multicore processors integrate multiple identical (homogeneous) or different (heterogeneous) processor cores. While the core itself and first-level cache are fully replicated, most other parts of the infrastructure are shared between the cores. Therefore, applications profit from increased processing power, but can still be limited by the shared memory bandwidth. A superscalar processor uses spatially parallel execution units to process more than one instruction per clock in a single core [9]. Instead of replicating the full core to form a multicore processor, components for instruction fetching and dispatching are enhanced to process multiple instructions per cycle. These are then dispatched to different execution units running in parallel and are therefore also referred to as multiple-issue processors. Dependencies between instructions are handled in hardware, as conflicts like data dependencies, register interference and branches can occur. The resulting hardware overhead is considerable, hence superscalar architectures are only found in high-performance processors.

2.1.2 Memory Hierarchy

The fundamental operation of a computer involves the association of instructions with corresponding data to generate an output. In a von Neumann architecture, data

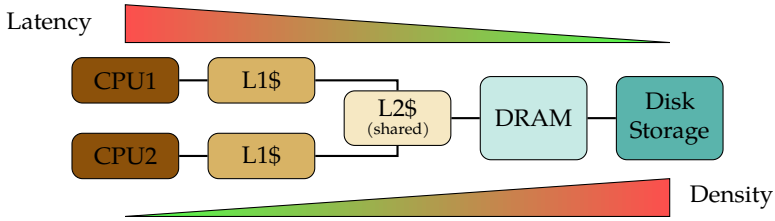


Figure 2.4: Typical memory hierarchy in a multiprocessor system. The reciprocal relationship between latency and storage density of different levels of memory is illustrated qualitatively. The CPU has immediate access to its register file and first-level cache, resulting in very high bandwidth, while the main memory has great capacity but high access latency.

needs to be present at the Arithmetic Logic Unit (ALU) when the corresponding instruction is executed. This requirement is referred to as *data locality* [10]. Data and instructions are stored in a central memory, creating the so-called von Neumann bottleneck. Whether an application is limited by memory or processor performance depends heavily on the algorithm. The Roofline Model is a simple graphical model that visualizes this relationship and is introduced in Section 4.2.1.

To overcome the von Neumann bottleneck, many architectures use a hierarchy of different memories to hide the main memory latency. The most common approach is adding caches for instructions and data, characterized by access latencies of a few cycles. The concept can be extended by adding further layers beyond the main memory, giving access to greater storage capacities with an additional increase in latency.

In large-scale systems, the memory hierarchy becomes more complex with several levels of different, heterogeneous memories being available. Figure 2.5 shows an exemplary memory hierarchy of a manycore system. Several self-sufficient multicore systems with local memory are connected using a Network-on-Chip (NoC) (Section 2.1.3) to form a manycore system. This introduces the notion of *local* memory, being accessible through the local bus, and *remote* memory, which involves a NoC transfer for each access.

A multi-level memory hierarchy adds additional challenges for memory orchestration on the software level. Programming languages like *X10* developed by IBM have distinct features to specify the location of data and computations [11]. The Partitioned Global Address Space (PGAS) is a memory organization model, mapping all available

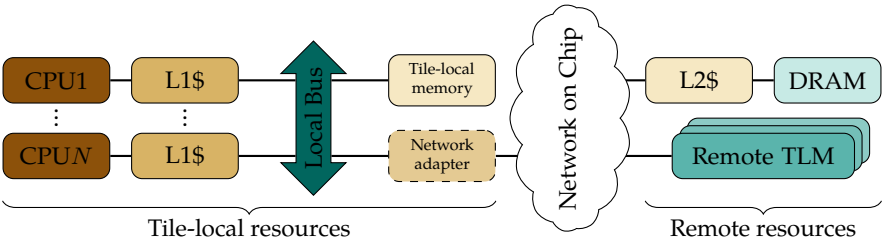


Figure 2.5: Example for a multi-level memory hierarchy in a manycore System-on-Chip (SoC). A NoC is used to connect multiple clusters of N CPUs, allowing each tile to access remote resources. Local resources can be accessed faster than remote resources, while local memory is limited and only a single, large shared Dynamic Random Access Memory (DRAM) is available.

memory resources into a single address space. The location of data within the memory hierarchy can always be deduced from its address. Hardware support for accessing any address within the PGAS from any location in the hierarchy is necessary and can cause additional overhead.

2.1.3 Network-on-Chip

A Network-on-Chip (NoC) is a network-based interconnect system implemented on a chip [12]. It is typically used to connect building blocks, like processors, memory, accelerators and periphery within a SoC. A NoC is preferred in large-scale systems for better scalability than a bus. Traditional bus architectures permit only a single master to initiate a data transfer, blocking the bus for other masters until that transfer is complete. Although techniques like bus splitting help reduce this contention, the bus architecture becomes unfeasible for systems requiring tens to hundreds of interconnected modules. Furthermore, the challenge of meeting timing constraints across greater physical distances—a consequence of larger die area in modern designs—additionally restricts the scalability of bus-based systems. An example application is given in Section 2.2.1 using a NoC in a manycore architecture.

Data in a NoC is transferred in packets, which are physically split into Flow Control Units (FLITs) on the link level. Routers are implemented at each intersection to handle forwarding of packets according to a routing algorithm. Traditionally, a static

XY-routing algorithm is used for planar homogeneous mesh topologies. Sophisticated dynamic routing algorithms can be implemented to support heterogeneous topologies or adapt to changes to the topology or traffic patterns at run time, but the additional hardware overhead needs to be considered [13]. In synchronous routers, each router holds registers to buffer data and reduce timing requirements on the path between tiles.

In theory, a NoC allows each node to send data in parallel. However, the total bandwidth is limited and depends on the number of hops and other transfers at the same time. A backpressure mechanism is therefore implemented in each router and connected node, i.e. in their Network Adapter (NA). The performance of a NoC is measured over varying the injection rate, at which packets are sent into the network by each node. A low-traffic scenario can be described with an injection rate of 0.025, while a rate of 0.15 packets per cycle represents high traffic [14].

2.1.4 Systolic Arrays

Systolic arrays are the basis for many dataflow-driven architectures, which are characterized by a regular grid of simple processing units. Despite their original publication by Kung et al. in 1978, systolic arrays continue to hold relevance today [15]. The term *systolic* describes the rhythmic, pulse-like flow of data through this network, inspired by the human heart. Data enters at the boundaries, passes through multiple PEs step-by-step, and results emerge from opposite boundaries. Each PE typically performs a fixed, basic computation, such as a multiply-accumulate operation, on the data it receives before passing intermediate results to its neighbors. Figure 2.6 depicts an abstract systolic array with 12 PEs, each containing an ALU and registers for results and forwarded input data. Hardware designs inspired by systolic arrays often extend the concept by implementing sophisticated PEs supporting multiple commands or an internal register file. The principle of cycle-based data forwarding enforces highly pipelined data movement, which allows for massive parallelism and high throughput, as many PEs operate concurrently on different parts of the data streams. A key advantage of systolic arrays is their efficient data reuse; data elements often interact multiple times within the array, significantly reducing the demand on external memory bandwidth. For correct execution, the dataflow path through the array must precisely align with the target computation, ensuring each data element traverses the appropriate PEs for the required operations. Advanced systolic architectures can incorporate reconfigurable dataflow paths, thereby improving their adaptability

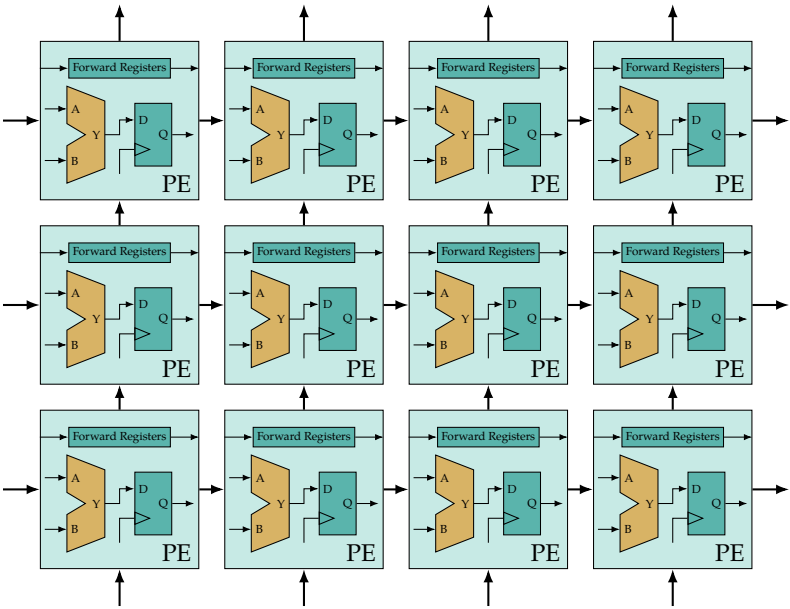


Figure 2.6: Abstract systolic array with 4×3 Processing Elements (PEs). Each PE performs arithmetic operations on data received from its predecessors, and the input data is forwarded to its successors in the next clock cycle.

to diverse applications. Consequently, systolic arrays are exceptionally well-suited for parallel algorithms exhibiting regular dataflow patterns, such as matrix multiplication, convolutions (prevalent in deep learning), FIR filtering, and solving linear equation systems. Their hardware-level optimization for these specific computational patterns enables them to achieve high performance and energy efficiency. Therefore, systolic arrays are a common concept for many parallel computing architectures.

2.2 Parallel Computing Architectures

This section provides an overview of key parallel computing architectures, detailing their basic organization and the foundational principles applied to them. The discussion covers, for instance, manycore systems—an evolution of multicore processor principles—and specialized dataflow architectures like Coarse-Grained Reconfig-

urable Architectures (CGRAs). These specific architectures are highlighted due to their relevance to the subsequent content of this thesis. Hennessy and Bobda provide a comprehensive overview of these and other architectures [7; 16].

2.2.1 Large-scale Manycore Architectures

As the persistent demand for higher performance drove a steady increase in core counts, the shared bus interconnect of early multicore processors became a critical performance bottleneck. This bus-based approach could not scale to handle the rising communication traffic, leading to prohibitive contention and limited bandwidth. Consequently, a fundamental paradigm shift was required, moving from shared buses to scalable, packet-switched interconnects known as NoCs. This transition enabled the development of manycore systems, which integrate tens to hundreds of cores on a single chip [8]. In this model, cores communicate by sending data packets through a network of routers and links, allowing for concurrent communication and providing bandwidth that scales with the number of cores. This foundational design principle supports the high degree of parallelism in modern manycore architectures.

Figure 2.7 shows an example for a meshed manycore architecture, heterogeneously built with CPU, memory, accelerator and input/output tiles. CPU tiles can be multiprocessor systems themselves and include tile-local memory to store the current working set. Memory tiles provide access to large Shared Memory (SHM), typically using DRAM technology, and usually incorporate a cache to mask DRAM latency. Multiple independent memory tiles allow for workload partitioning between the memories, such that the bandwidth of both memories can be used in parallel by possibly independent processes. Data locality, i.e. data stored within a tile or in neighboring tiles, can be leveraged manually by the programmer or automatically by the runtime environment to reduce data access latency.

Programming large, heterogeneous manycore systems presents its own set of challenges. Similar to scaling the hardware architecture, common software approaches in operating systems do not scale well beyond hundreds of processor cores. The presence of multiple small multiprocessor systems running independently of each other resembles warehouse-scale server systems, but scaled down to the size of a single chip. Each tile can operate independently, but needs to synchronize and interact with other tiles to execute large applications that do not fit on just a single tile. Operating

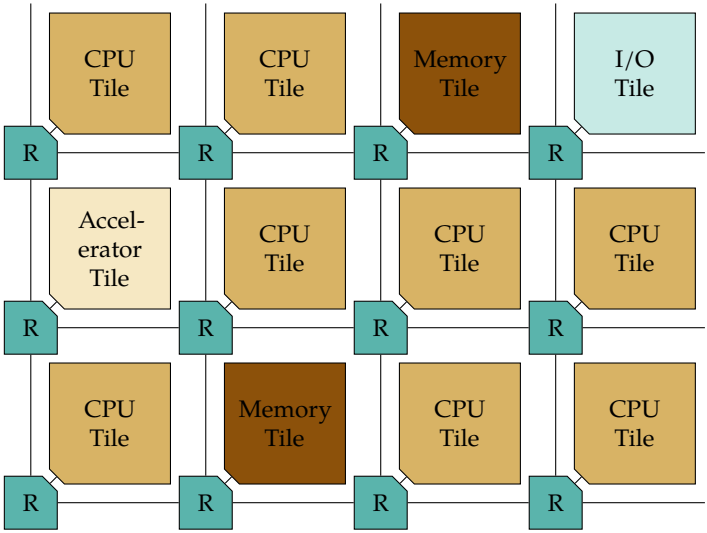


Figure 2.7: A 4 × 3 excerpt of a meshed manycore architecture using a NoC interconnect. The heterogeneous system comprises CPU, memory, I/O and accelerator tiles. Access latency can vary depending on the location of data and its requester, as a multi-level memory hierarchy is established.

systems for such architectures follow distributed approaches and rely on cooperation between the individual processing systems [17].

NoCs are not strictly used for multi-level manycore architectures. CPU manufacturers use custom interconnect systems in processors with high core counts, as AMD's *Infinity Fabric* [18]. This system is used to connect dies on the same package, as the *Zen* microarchitecture combines chiplets carrying processor cores with interconnect dies. Data transmission is implemented using high-speed Serializer/Deserializer (SERDES) units. Two modes are implemented, one for on-package communication with single-ended drivers for reduced power consumption, and one for inter-socket communication using differential drivers similar to PCI Express (PCIe). Although this infrastructure is not strictly a packet-based NoC with routers, it is a good example of the industrial use of the concept.

2.2.2 Graphics Processing Units

As their name suggests, GPUs were originally dedicated hardware for graphics operations. However, they have since evolved into formidable parallel computing engines, fundamentally designed for high-throughput, data-parallel workloads [19]. Their architecture is characterized by a multitude of processing elements, making them highly flexible vector processors for tasks in scientific computing and both inference and training of deep neural networks. The core of this parallel capability lies in the array of SIMD processor units, also known by vendor-specific terms like Streaming Multiprocessors (SMs) or Compute Units (CUs) in NVIDIA's and AMD's architectures, respectively. Each SIMD processor itself houses several simple ALUs, often referred to as CUDA cores or stream processors. Instructions are typically issued to groups of threads, known as warps (NVIDIA) or wavefronts (AMD), which execute the same instruction in lockstep across different data elements on these ALUs. This allows, for example, a single addition instruction to be performed simultaneously on 32 or 64 different data points. Each unit is further equipped with its own instruction cache, schedulers for managing warps, a large register file, and a dedicated, low-latency L1 cache. The L1 cache can be repurposed as local shared memory for compute applications, which enables data sharing among threads executing on that unit. The ability to execute tens of thousands of threads concurrently, with each SM independently processing multiple warps, allows GPUs to achieve massive parallelism. Operations like matrix multiplications and convolutions, central to neural network algorithms, can be mapped to this SIMD architecture, as well as many numerical methods in scientific simulations that involve operations on large vectors and grids. Their inherent parallelism allows GPUs to hide memory latencies through extensive hardware multithreading, rapidly switching between ready warps. Thus, the GPU's power as a vector processor stems from both massive spatial parallelism, due to the high number of vector processors, and temporal parallelism through hardware-based multithreading.

Figure 2.8 shows an overview of the architecture of an SM in NVIDIA's current architectures, *Hopper* and *Blackwell* [21]. Each SM contains 128 SIMD processing lanes clustered into four independent units. Additionally, a specialized tensor processing accelerator and four texture units are present per SM, and 128 KiB of shared memory and 256 KiB of registers are available.

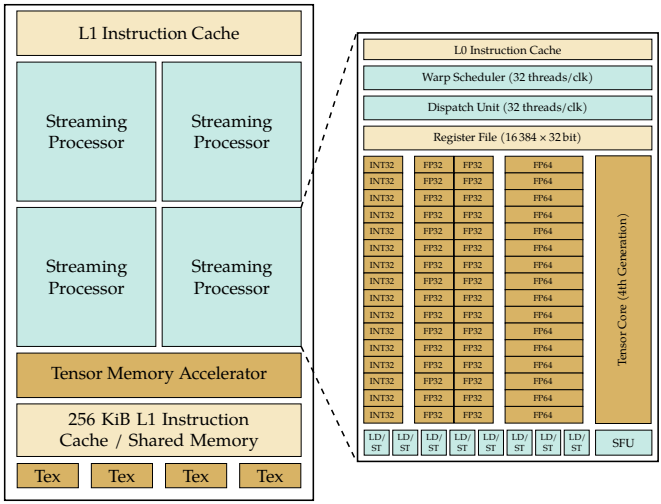


Figure 2.8: Architecture of a *Streaming Multiprocessor* in the NVIDIA H100 and Blackwell architectures. It consists of four independent SIMD units, each equipped with a *Warp Scheduler* for hardware-based scheduling of threads. Texture units and 256 KiB of cache are shared between the four units. Image adapted from [20].

2.2.3 Coarse-Grain Reconfigurable Arrays

CGRAs are processing arrays which can be categorized between processor-based architectures and Field Programmable Gate Array (FPGA) designs. They differ from classical, control-flow oriented processors in that the programming is primarily dataflow-driven [22]. On the other hand, they are characterized by the presence of the computational elements as fixed logic. These elements are called PEs and support a predefined set of instructions similar to a General Purpose Processor's (GPP) ALU. By using fixed logic for ALUs, CGRAs avoid the performance impact of reconfigurable hardware introduced by multiplexers, Lookup Tables (LUTs) and the interconnect on FPGAs. The dataflow between these elements, and in some designs also within them, can be reconfigured by multiplexers or similar logic. CGRAs thus aim for improved performance by providing commonly used arithmetic operation as high-performance fixed logic, and allow dataflow reconfiguration to adapt to different scenarios.

With the first CGRA designs like the *Xputer* published in the early 90s up to recent frameworks to create tailored CGRAs for different applications, they have been very popular in academia [23; 24]. Nevertheless, their incorporation into commercial products remains limited due to the challenges associated with their programmability.

2.2.4 Field-Programmable Gate Arrays

FPGAs are specialized integrated circuits that can be dynamically reprogrammed after manufacturing. Their behavior is defined through a configuration file, which allows them to represent most kinds of user-defined logic. They consist of a homogeneous array of Programmable Logic Blocks (PLBs) and a configurable interconnect layer, which connects logic blocks to realize custom logic [16]. The first commercially available FPGA has been introduced by Xilinx in 1985, and has been refined continuously to fit more logic and improve performance. Early designs featured just the basic logic elements and interconnects, whereas current architectures contain several additional fixed logic components. Block RAM (BRAM) elements are added to provide customizable, high-density memory, and Digital Signal Processor (DSP) blocks can be used for some common arithmetic operations instead of implementing them in programmable logic. All manufacturers offer several architectures with a varying number of logic elements, and multiple product families can be found targeting specific applications. For example, some provide an increased number of high-speed I/O interfaces, others focus on additional memory resources.

FPGAs are primarily used in two applications: to realize custom logic when the budget does not allow manufacturing the necessary Application-Specific Integrated Circuit (ASIC), and for prototyping of novel architectures for Design Space Exploration (DSE) and verification prior to ASIC production. From a financial point of view, FPGA vendors take over the costs for ASIC production and distribute them among their customers. Especially for small companies with limited financial resources, using an FPGA is often the only option to realize custom logic. For simple applications like glue logic, a Complex Programmable Logic Device (CPLD) with fewer logic resources can be used as an alternative.

The high initial costs of ASIC designs make an FPGA-based solution more efficient for a low and medium number of units. Logic density and clock frequency of FPGAs increased significantly since their initial release, resulting in continuous improvements in the efficiency of both ASIC and FPGA implementations. Thus, the crossover point

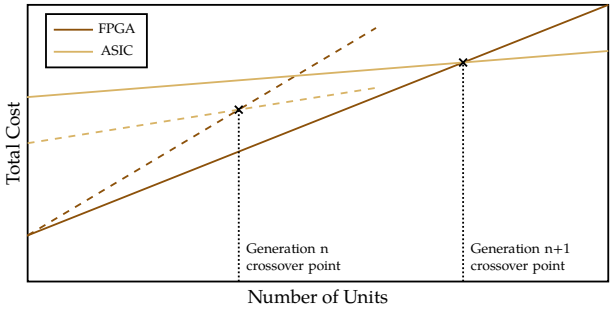


Figure 2.9: FPGA vs. ASIC crossover point in costs versus number of units. ASIC designs have a high initial cost, which falls below the cost of FPGAs at a high number of units. While newer ASIC process nodes tend to be more expensive, the next FPGA generation usually becomes more cost-effective. Figure adapted from [25].

of cost efficiency between ASIC and FPGA moves, as visualized in Figure 2.9. The Non-Recurring Engineering (NRE) costs increase significantly for modern process nodes, while FPGAs tend to become cheaper [25]. Advanced technology nodes allow FPGA designs to reach progressively higher clock frequencies and thus compete with ASICs for many workloads. However, the nature of FPGAs makes them generally less efficient than an ASIC implementation. Studies show a greatly reduced clock frequency (delay increased by 18 × to 26 ×) and lower logic density (area increased by 17 × to 27 ×) compared to implementing the same logic fully or partially as ASIC [26]. Still, these drawbacks can often be overcome by specialized engineering (e.g. parallelization), and the FPGA solution remains the preferred choice, particularly due to its cost efficiency.

Figure 2.10 illustrates the structure of a generic FPGA architecture. It shows an array of PLBs which can be connected using a configurable interconnect. At each intersection of the horizontal and vertical wires, a Switch Box (SB) allows connecting two intersecting wires. Additionally, inputs and outputs of each PLB can also be connected to the interconnect mesh using a SB. This allows to connect arbitrary PLBs to realize complex logic functions, although the number of routing resources is limited physically. The architecture of a PLB is detailed in the upper part of Figure 2.10. This design implements the logic path with LUT, adder and register twice, which is a common design choice in FPGAs but can still be different in some architectures. The most prominent component is the LUT with four inputs in this example. Its output for

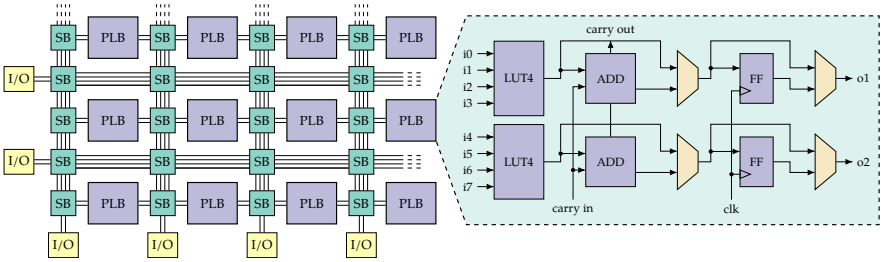


Figure 2.10: Exemplary FPGA architecture featuring PLBs, Switch Box (SB) and input/output buffers. The detailed view of a PLB highlights the programmable LUT and multiplexers implementing the programmed logic function.

any combination of inputs can be programmed and is usually stored in Static Random Access Memory (SRAM), while some devices use antifuse or flash memory. Since integer addition is a frequent operation, a discrete ripple-carry adder is implemented and can be programmed to be in the data path if needed. It uses the LUT result and a cascading carry bit from a neighboring PLB as inputs. If necessary, the output from the LUT or adder can be stored in a register, and the final signal is passed on to the interconnect network.

Architectural details can vary greatly between manufacturers and models. In Xilinx FPGAs, the PLB is called Configurable Logic Block (CLB), and uses 6-input LUTs in the recent *Versal* architecture. In addition to the standard components, the BRAM, DSP and clock generation components of modern FPGAs are also regularly distributed throughout the chip area. The facilities for programming have been left out from Figure 2.10 for clarity, but are of course of particular importance for the configurability of an FPGA. Configuration interfaces are provided both to the outside, i.e. for programming from external flash memory or JTAG interface, and to the inside for the logic design to interface with. This allows for partial dynamic reconfiguration, where the custom design on the FPGA can reprogram parts of the design by providing an appropriate configuration bitstream. Building on this feature, a research domain has emerged that deals with hardware designs that can be adapted at run time. In this work, we also make use of this feature to implement partially adaptive hardware components. An introduction of our approach is given in Chapter 4.

A notable family are SoCs with embedded FPGAs, like the *Zynq* Multiprocessor System-on-Chip (MPSoC) family by Xilinx. They incorporate one or more GPPs,

DRAM interfaces and additional periphery as hard logic. The embedded FPGA can interact with the processor system using bridges to the main memory bus or by emitting interrupts. This combined architecture enables deploying both software and accelerated custom hardware in a single chip, easing integration of FPGAs in embedded systems.

2.3 Neural Networks

Artificial Neural Networks (ANNs) are today's most popular approach towards Artificial Intelligence (AI), covering tasks like speech recognition, text generation and image classification. They are inspired by the biological structure of a brain, built from neurons and synapses, and map their functionality to computers. The original idea of realizing Machine Learning (ML) with ANNs has been around for several decades. In the 1960s, pioneers like Rosenblatt presented initial systematic descriptions of neurons and the neural processes within [27]. However, the limited computational performance of contemporary computing systems hindered the practical application of early ANN approaches. Consequently, the ideas were not well received and were not pursued further for the time being. The development of the backpropagation algorithm in 1986 marked the next step towards ANNs [28]. Still, the computational complexity, which was further increased by the requirements of the backpropagation method, could not be matched by computers available at the time.

It took another twenty years for ML to become popular again. The continuous improvement of GPUs and the associated increase in parallel processing performance paved the way for real-world applications of neural networks. Consequently, several ANNs using the backpropagation algorithm for image classification were shown between 2009 and 2012, outperforming traditional approaches [29]. The complexity of ANNs quickly increased, shaping the term *deep learning* referring to the growing number of layers [30]. Recent Deep Neural Networks (DNNs) target various tasks and can often exceed human capabilities. Large Language Models (LLMs) represent a significant advancement in the field of natural language processing, enabling computers to comprehend and interact with humans in a variety of applications [31]. Networks for image detection find various applications in robotics, autonomous vehicles, personal assistance systems and industry. In medical applications, ML helps experts to analyze imaging results reliably, achieving accuracies of 97 % to 99 % throughout several disciplines [32].

The growing computational demands for training state-of-the-art ML architectures have led to a corresponding and significant rise in their energy consumption. GPUs are running the majority of large-scale AI workloads, and while they can provide the necessary parallel processing power, their programmable nature leads to suboptimal energy efficiency. To illustrate the scale of this issue: a complete training cycle of a single LLM can consume hundreds of megawatt-hours (MWh) of energy [33]. The substantial power demands of modern ML applications raise environmental questions, as training of an LLM can emit over 284 t of carbon dioxide equivalent [34]. Therefore, there is a pressing need for efficient methods for training and inference of neural networks.

2.3.1 Machine Learning Principles

Modern ANNs are available in a wide range of variants, which are composed of a high number of neurons. To understand such neural networks, we start by looking at a single artificial neuron, the *perceptron*. An initial version of a generic perceptron was published by Rosenblatt in 1962 [27]. Today, neuron models have improved a lot, but the original theory remains applicable. A perceptron takes multiple binary inputs and produces a single binary output. Every input x_i is multiplied with a weight w_i, representing the importance of the respective input. The output of the perceptron is 1, if the sum of the weighted inputs is above a certain threshold, and 0 otherwise.

A perceptron represents a single-layer feed-forward neural network, handling binary inputs and producing a single output. Single-layer perceptrons can thus only learn linearly separable functions [36]. To learn more complex functions, multiple perceptrons are linked together to form a Multilayer Perceptron (MLP) [37]. A simple example of a fully-connected MLP is shown in Figure 2.11. Identical to a single perceptron, each neuron in the MLP holds weights for all of its inputs and a bias applied as an offset to the output. The neurons now support real numbers between 0 and 1 to allow learning using the backpropagation algorithm. The first layer of an MLP is the input layer, which does not consist of neurons itself but just represents the input values. Each input is connected to all neurons of the first hidden layer, making it a fully-connected MLP. The structure is now repeated several times, forming a set of consecutive hidden layers, each forwarding their outputs to the next layer. The final layer is called the output layer. Identical to the hidden layers, the output neurons compute a weighted sum of all preceding neurons. The number of output neurons can differ from the hidden layers and is adjusted according to the target application.

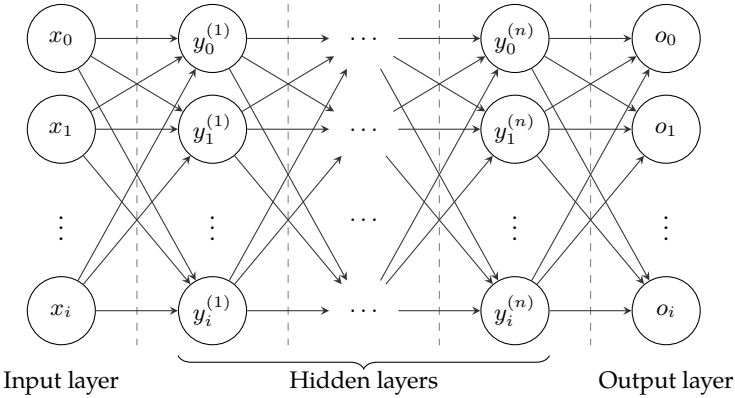


Figure 2.11: A simple Multilayer Perceptron (MLP) with an equal number i of inputs, neurons in hidden and outputs [35]. It is fully connected, i.e. each neuron is connected to the output of every preceding neuron. Within each neuron, a weighted sum of the inputs is calculated.

Various types of neural networks have developed on the basis of the MLP. For example, a Recurrent Neural Network (RNN) introduces a recurrent connection in addition to the inputs from the previous layer for processing of sequential data. This feedback loop allows the network to maintain an internal hidden state, which acts as a memory that synthesizes information from past inputs. A Long Short-Term Memory (LSTM) network advances RNNs to counter the vanishing gradient problem. It accomplishes this by adding gating units (input, forget, and output gates) that regulate the flow of information into and out of the neuron's state, allowing the model to selectively learn, remember, and forget information over long sequences. Transformer networks advance from traditional MLP-like sequential processing in favor of parallel computation. The core innovation is the self-attention mechanism, which simultaneously calculates the contextual relevance of every input token in a sequence to every other token, enabling the capture of complex, long-range dependencies without sequential iteration. Another widely used class of ANN is the Convolutional Neural Network (CNN), which is suitable for image processing tasks. As we target efficient Convolutional Neural Network (CNN) processing in Chapter 6, we take a closer look at their architecture in the next section.

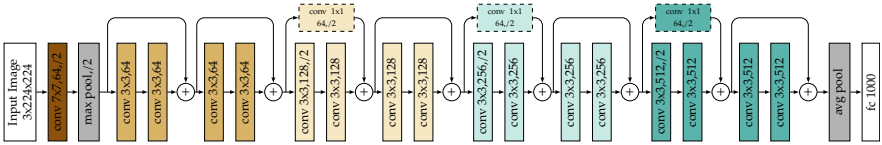


Figure 2.12: Structure of ResNet-18, built from a sequence of *residual* blocks.

2.3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of ANNs specializing in image processing tasks [35]. Most hidden layers in a CNN are a multidimensional convolutional operation, which helps them to recognize spatial features in pixel-based data. One of the first notable CNNs is *LeNet-5* published in 1998 [38]. Compared to the MLP, it is significantly more effective in recognizing handwritten digits. However, the real breakthrough of CNNs first happened when powerful GPUs became available. *AlexNet* was one of the first GPU-accelerated CNNs and quickly established itself as a benchmark in the field. Subsequently, numerous Convolutional Neural Networks (CNNs) have been developed for diverse applications, demonstrating constant gains in accuracy and efficiency.

The *ResNet* series is selected as representative modern, yet simple CNNs for image classification [39]. It introduces residual blocks as basic building blocks for networks, where the input to a residual block is added to its output. This shortcut connection helps to improve accuracy by increasing the network depth while being easier to train and optimize. The core component of each residual block is the convolution operation. Analogous to other networks, the first layer starts with a high resolution and few input channels. After a number of residual blocks, the resolution is regularly halved and the number of channels doubled. Finally, a fully-connected layer maps the results to an output vector, with each element representing one of the 1000 trained object classes. Figure 2.12 illustrates the structure of ResNet-18, the smallest model in the series. The network is composed of 18 layers in total: one input layer, 16 layers within residual blocks, and one final output layer. Starting from the third residual block, the first convolution in each block reduces the image size by $1/4$. In these cases, the shortcut connection uses a 1×1 convolution to match the output image size of the block. In addition to the residual blocks used for ResNet-18, the bottleneck residual block is introduced. It uses a 1×1 convolution before and after the actual convolution in the residual block to reduce the image size first, then operate on the

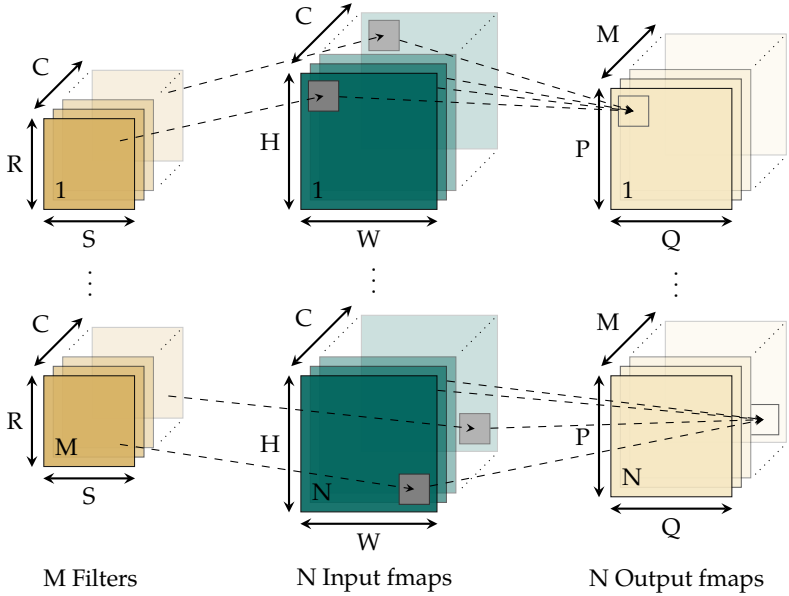


Figure 2.13: Multidimensional convolution operation typically used in CNNs. The dimensions are named according to [35] and this definition is re-used in Section 6.2. Refer to Table 6.2 for details.

reduced resolution, and return to the original resolution afterward. The authors could demonstrate a considerable reduction in parameters using bottleneck blocks while maintaining accuracy. Their networks ResNet-34, ResNet-50 and ResNet-101 achieved top-1 error rates on the ImageNet validation dataset of 21.53 %, 20.74 % and 19.87 %, respectively. Due to their efficiency, ResNets are often used as the basis for more advanced networks, such as U-Net [40]. Either ResNet-18 or its competitor EfficientNet is used for classification and enhanced with a decoder structure to classify individual pixels in the input image.

All CNNs share the name-giving element, the convolution operation, which will now be examined in detail. Figure 2.13 shows a graphical representation of a multidimensional convolution operation. Its input is a four-dimensional set of input feature maps (ifmaps) and a set of filters. The ifmap dimensions are $N \times C \times H \times W$, where $H \times W$ is the height and width of the image, C is the number of channels and N is the batch size. The filter set has dimensions of $M \times C \times R \times S$, with $R \times S$ being the

filter dimensions and M the number of output channels. This results in an output feature map (ofmap) of $N \times M \times P \times Q$, where $P \times Q$ is the image size of each output channel and M is the number of channels. Additionally, a constant bias per output channel is typically added to each output pixel, resulting in a bias vector of M elements. Equation (2.1) describes the operation to obtain one output pixel o of the ofmap from the ifmap i, filters w and bias b:

$$o(n, m, p, q) = b(m) + \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} i(n, c, p + r, q + s) \times w(m, c, r, s) \tag{2.1}$$

Some CNNs use additional parameters for the convolution operation like stride, dilation, and padding. The stride dictates the step size when the filter is moved over the ifmap. With its default value of one, no elements are skipped. When set to two, every second input element is skipped, effectively halving the ofmap size. Thus, stride is an alternative option to pooling operations for reducing the feature map size. The dilation parameter virtually increases the ifmap area covered by the filter. A horizontal dilation value of two results in applying a filter with $S = 3$ to input elements $q + 0, q + 2, q + 4$ instead of directly neighboring elements. Padding allows maintaining the ifmap size on the output. Without padding, the ofmap can only be calculated for $Q = W - S + 1$ elements in the horizontal dimension, the same applies for the vertical dimension. Padding the ifmap with additional values on each edge virtually increases $H \times W$ and allows obtaining an ofmap with the same size as the input.

2.3.3 Efficient Processing of CNNs

The convolution is the predominant operation in CNNs, thus it is crucial to compute it efficiently. Fundamentally, a convolution operation consists of a series of Multiply-Accumulate (MAC) operations. A straightforward implementation loops over the ofmap dimensions and applies Equation (2.1) to compute each element. The number of MAC operations can be computed based purely on the dimensions.

$$N_{MAC} = (N \cdot M \cdot P \cdot Q) \cdot (C \cdot R \cdot S) \tag{2.2}$$

$$P = \lfloor \frac{H + 2 \cdot pad_H - R}{stride_H} + 1 \rfloor \tag{2.3}$$

$$Q = \lfloor \frac{W + 2 \cdot pad_W - S}{stride_W} + 1 \rfloor \tag{2.4}$$

By using the number of ofmap elements and operations per element, Equation (2.2) calculates the number of MACs required to compute the layer. Output dimensions P and Q can be obtained based on input dimensions and attributes for padding and stride [35]. For example, this results in 1.18×10^8 MACs for the forward pass of the first layer of ResNet-18. The full network requires 1.81×10^9 operations.

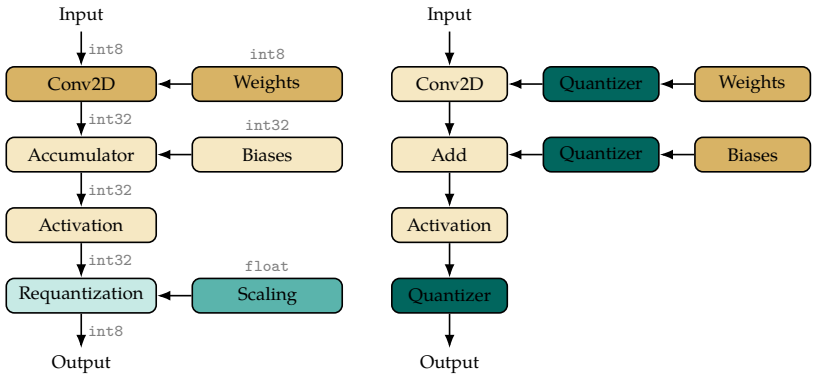
This immense number of operations is a challenge for many computers. While GPUs have proven to be more suitable than GPPs, there is still room for improvement. An advantage of the convolution operation is the limited number of data dependencies between input and output values. Each ofmap element is only dependent on a small excerpt of the ifmap tensor, allowing for extensive parallelization. This has made convolutions a popular benchmark for many parallel computing architectures, including some presented in Section 2.2. Driven by the huge attention for CNNs, many specialized architectures exploiting the parallel nature of convolutions have been published [41].

Many researchers have found that systolic arrays (Section 2.1.4) are particularly suitable for processing convolutions. Although the algorithm iterates over seven dimensions, the inner loop stated in Equation (2.1) is independent of the higher-dimensional iterations. This allows these loops to be processed in parallel, e.g. spatially separated in different PEs. In addition, PEs in systolic arrays are also arranged planar like the pixels in a two-dimensional image, while the other dimensions (N, M and C) are processed sequentially in time. The arithmetic operation in each PE is relatively simple because, apart from the bias addition at the end, only one MAC is required per iteration. On most platforms however, convolution algorithms are memory-bound, as the memory bandwidth is quickly exhausted due to the low Operational Intensity (OI). For a detailed analysis of the OI in the Roofline Model, refer to Section 4.2.1. Approaches to reducing the required memory bandwidth are being investigated from various perspectives. It can be observed that many ifmaps and filters are *sparse*, i.e. they are close to zero and their impact on the result is negligible. Such data can either

be compressed or skipped entirely, reducing bandwidth [42]. ANNs generally allow the application of approximate computing techniques, with quantization being the most common method.

Neural network quantization is a critical optimization technique that converts the high-precision floating-point numbers (typically 32-bit floats) used during model training into low-precision integer representations, such as 8-bit integers. The primary advantage of this conversion is a significant boost in efficiency: quantized models require less memory, consume less power, and can execute much faster on specialized hardware. This is because integer arithmetic is simpler and more energy-efficient than floating-point arithmetic, making quantized models ideal for deployment on resource-constrained edge devices like smartphones and embedded systems. However, this efficiency comes at the cost of a potential drop in model accuracy, as mapping a wide range of floating-point values to a limited number of integers introduces quantization errors. Most neural networks have considerable redundancy in their parameters, allowing for some degree of data reduction through quantization [43]. Quantization algorithms can be divided into two distinct classes: Post-Training Quantization (PTQ) and Quantization Aware Training (QAT) [44]. PTQ is a simple, fast approach where a pre-trained model is converted directly to a lower precision, often using a small calibration dataset to determine the optimal parameter ranges. In contrast, QAT simulates the effects of quantization during the training process itself, allowing the model to learn to be robust to approximation errors. This typically results in higher accuracy than PTQ, though it comes at the cost of increased training complexity. The choice between these methods depends on the specific application's tolerance for accuracy loss versus the need for a fast and simple deployment workflow.

Figure 2.14 shows the dataflow of a typical quantized convolution layer. In addition to the main convolution operation, such a layer includes biasing, applying an activation function, and requantization of the outputs. While the input and output of the layer is an 8-bit integer, the raw result of the convolution is stored with 32 bit. The reason is that an 8-bit accumulator quickly overflows when adding up integers from many channels. Typically, accumulator widths range from 20 bit to 32 bit, depending on the target networks and the quantization methods. In order to return to 8-bit granularity on the output, the values are scaled-down using a floating-point factor in the requantization step. The requantization factors result from the quantization algorithm based on the minimum and maximum values observed on a calibration dataset. According to [44], the multiplication of a quantized ifmap element $\hat{x}$ and filter weight $\hat{W}$ can be expressed as follows:



(a) Fused 2D convolution for inference (b) Fake quantization for floating-point

Figure 2.14: Dataflow graph of a fused convolutional layer, including activation function and quantizing steps. Integer operations are used for inference on specialized hardware; fake quantization can be used to simulate the behavior on training systems. Image adapted from [44].

$$\hat{W} \cdot \hat{x} = s_w(W_{int} - z_w) \cdot s_x(x_{int} - z_x) \tag{2.5}$$

$$= s_w s_x W_{int} x_{int} - s_w z_w s_x x_{int} - s_w s_x z_x W_{int} + s_w z_w s_x z_x \tag{2.6}$$

The last two terms of the sum (blue) in Equation (2.6) are independent of the input and can be precomputed. The second term (red) however, depends on x_{int} and needs to be evaluated in addition to $W_{int} \cdot x_{int}$, doubling the multiplications per element. As a consequence, symmetric quantization is usually applied to weights, resulting in the zero point z_w to be zero. Asymmetric quantization, where a non-zero zero point is used, can be applied to the input for increased flexibility in the quantization process. The factor $s_w \cdot s_x$ represents the floating-point requantization factor in Figure 2.14a. To simulate quantization effects on floating-point hardware, fake quantization can be used (Figure 2.14b). Here, the operations are still carried out in floating-point format, but scaling factors are applied and the inputs are rounded to integer numbers before the actual operation. This can be used in the quantization process to find suitable parameters for scaling and zero points.

In Section 6.2, a highly efficient hardware accelerator for CNNs is presented. It adopts a systolic array structure and supports quantized networks of 8-bit integer precision. Particular attention is paid to the memory interface so that performance losses due to a lack of bandwidth can be avoided.

Chapter 3

State of the Art in Adaptive Hardware Design and Parallel Architectures

A fundamental challenge in computer architecture since its inception has been to reconcile the inherently static nature of hardware, physically realized in silicon, with the need for operational flexibility. Achieving adaptability in hardware designs remains a persistent goal pursued by architects. This chapter provides a comprehensive overview of the current state of the art in the field of adaptive hardware architectures, exploring contemporary approaches designed to overcome the limitations of fixed hardware implementations. A fundamental distinction can be made between two types of architectures: control-flow based and dataflow-based. Consequently, we begin by examining control-flow based adaptive hardware, specifically in the form of run-time reconfigurable processors. Following this, our attention turns to dataflow-based architectures, looking for designs which enable a degree of run-time adaptivity. Presently, deep learning represents arguably the most significant trend within High-Performance Computing. As such, beyond general dataflow-based architectures, this review will also encompass hardware designs and accelerators specifically developed for Deep Neural Networks (DNNs).

3.1 Run-time Reconfigurable Processors

Reconfigurable processors try to bridge the gap between the flexibility of von Neumann processors and the computing power of specialized hardware [45]. To enhance a processor core with reconfigurability features, some kind of reconfigurable hardware needs to be added. In contrast to Coarse-Grained Reconfigurable Architectures (CGRAs), fine-grain reconfigurable architectures like Field Programmable

Gate Arrays (FPGAs) can map any hardware function, i.e. both the data path and the implemented logic function can be adapted. This makes them particularly flexible and applicable in a wide variety of domains.

Run-time reconfigurable processors are based on a General Purpose Processor (GPP) and incorporate reconfigurable hardware to extend the instruction set with custom instructions. Standard software can be executed, but computationally intensive operations can be identified, for which algorithms are being actively researched, and mapped to an FPGA-based logic block for faster execution. Ali et al. demonstrate a RISC-V processor with specific instruction set extensions for machine learning applications [46]. They follow the concept of an Application-Specific Instruction-Set Processor (ASIP) by tailoring the instruction set to a specific workload. The open source processor core *RI5CY* implementing the RV32IM instruction set is used as a basis. It is extended with a custom Vector Processing Unit (VPU) implementing Single Instruction Multiple Data (SIMD) instructions from the *V* extension of the RISC-V Instruction Set Architecture (ISA). Two different configurations of the VPU with a vector register length of 64 and 128 are examined. The larger configuration has been demonstrated to be more power efficient with 0.44 GOps/W in comparison to 0.41 GOps/W for the smaller configuration. Generally, their implementation is more efficient than software processing and shows an advantage in power efficiency of 1.2 × over a GPU implementation. This suggests that significant efficiency gains can be achieved by adding coprocessors. However, when applied to artificial neural networks, the effectiveness of a single VPU is limited due to the high number of Multiply-Accumulate (MAC) operations necessary.

Designing an ASIP requires identification of operations that benefit from custom instructions and implementation thereof in the hardware architecture. This principle holds true for both ASIPs fully implemented as Application-Specific Integrated Circuits (ASICs) and run-time reconfigurable processors. The distinction is the implementation method for the special instruction (SI): it is hardwired in ASIPs, whereas it is typically implemented on FPGAs in reconfigurable systems. In addition to the hardware implementation, software also needs to be altered to make use of the new SI. *Auto-SI* presents an approach to automatically detect loops during execution that profit from acceleration. Figure 3.1 shows a state chart of the process. To characterize the kernel currently executed, a real-time instruction trace is observed. When a consecutive sequence of operations forming an algorithmic kernel is recognized, a Data Flow Graph (DFG) is generated on-the-fly. This DFG is then used to determine potential SIs to replace the software implementation. If a match is found, the hard-

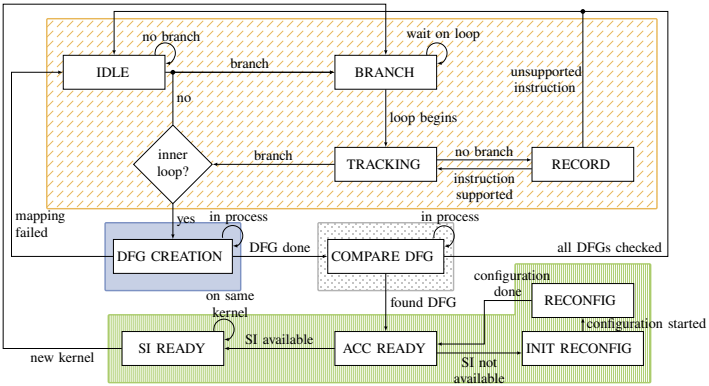


Figure 3.1: State chart of the *Auto-SI* approach to identify kernels for acceleration on a run-time reconfigurable processor during run time [47].

ware implementation is automatically launched the next time the respective kernel is executed. The approach assumes that a set of predefined SIs is present, which must match the operations executed in software. Sophisticated algorithmic kernels in software cannot be translated to an SI implementation automatically, which limits the applicability of this approach.

Spatzformer is a dual-core RISC-V architecture with vector units [48]. It is reconfigurable with regard to the vector units, which can either be used by one of the processors each or merged to be used by only one RISC-V core. Figure 3.2 shows the baseline architecture *Spatz* on the left, with the register file (VRF), load/store units (VLSU), and vector arithmetic units (VAU). On the right, components and connections to implement reconfigurability are highlighted. A single-threaded vectorized workload can achieve better performance by running in *merge mode*, where both vector units are used by one processor. The second processor can still support the application on any non-vectorizable control tasks. It demonstrates that slight hardware modifications can allow reconfiguration, without using FPGA hardware. However, the reconfigurability of these vector units is aimed solely at improving performance; it does not extend to alter the supported operations, which remain fixed after manufacturing. Still, this work highlights the performance impact of vectorization and execution of vectorized code on specialized hardware units. Additionally, the authors stress that many algorithms contain control tasks which can be run on a GPP.

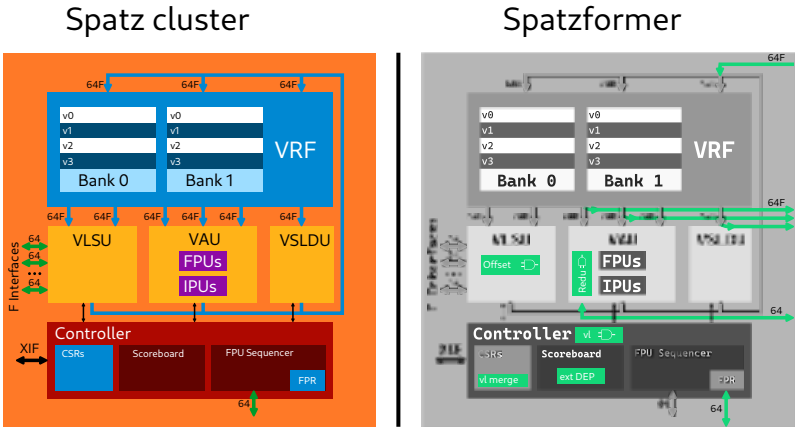


Figure 3.2: Reconfigurable vector units for RISC-V processors based on *Spatz*. Two vector units can be coupled to improve performance when used by a single core [48].

Classic vector units operate in SIMD fashion for a single command, and rely on the processor to determine the next operation. Taking single vector units a step further leads to arrays of vector processors. Graphics Processing Units (GPUs) are common examples for large-scale parallel processors, packaging up to 24 576 streaming processors in the current *NVIDIA Blackwell* architecture [21]. However, the overhead of instruction processing scales with the number of cores, increasing energy consumption.

Very Long Instruction Word (VLIW) processors offer spatial parallelism and reduced complexity compared to superscalar processors, since the instruction schedule is defined by the instruction encoding. They are often used for applications with predictable control-flow, which eases scheduling done by the compiler. Kumarathunga et al. present a reconfigurable VLIW processor for machine vision applications [49]. In contrast to standard VLIW processors, their design contains reconfigurable execution units for various tasks, including trigonometric functions, vector operations and a basic Arithmetic Logic Unit (ALU). Thereby, an application ships both the VLIW code and a corresponding configuration of execution units. The processor is designed as a coprocessor, so it is accompanied by a GPP to execute control tasks. Although the authors do not share details about the underlying reconfigurable logic, it can be assumed that they envision using an embedded FPGA, similar to their AMD

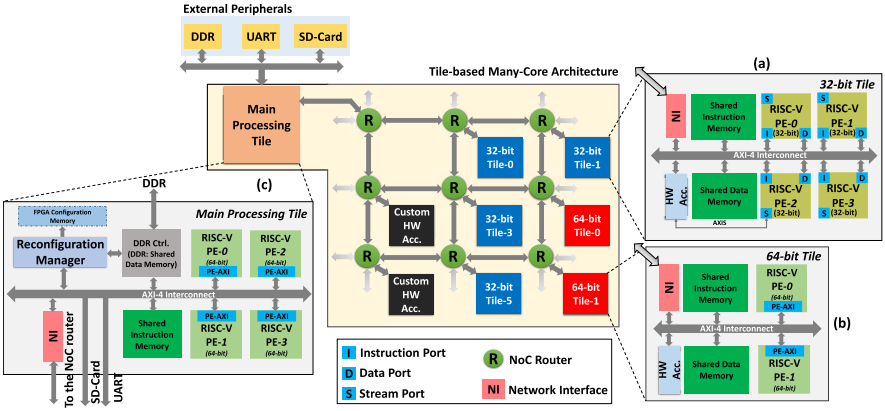


Figure 3.3: Overview of the *AGILER* platform with a 3 × 3 manycore architecture. It features RISC-V processors with local memory in each tile, but also allows including accelerator tiles.

Zynq 7000 prototyping target. A similar approach is followed by the *i*-Core, a run-time reconfigurable processor based on the LEON3 architecture. In addition to its fully-featured GPP core, so-called SIs can be executed using an attached FPGA-based accelerator fabric. Instructions for the accelerator fabric are called Very Large Configuration Word (VLCW), which resembles a VLIW design, but highlights the fact that the fabric is not a standalone processor. Details on the *i*-Core architecture are introduced in Section 5.1.

AGILER is an adaptive, heterogeneous manycore architecture [50]. It is based on a Network-on-Chip (NoC) connecting different tiles, containing *Compute Tiles* with 32 bit or 64 bit RISC-V processors. The open-source processor designs *RI5CY* and *Ariane* are used as the 32 bit and 64 bit respectively. The architecture is self-reconfigurable, enabling its processors to access the reconfiguration port of the host FPGA, and rewrite their own configuration. For this purpose, the design is equipped with a static region for the host tile and NoC components, as well as several reconfigurable regions for the individual tiles. An overview of the architecture is depicted in Figure 3.3. While this design theoretically supports including custom hardware accelerator tiles, the authors have not evaluated this feature. Run-time reconfiguration has been tested to switch between quad-core RI5CY tiles and dual-core Ariane tiles. Performance evaluation shows the highest performance for quad-core tiles for 2-D convolutions,

however no quantitative results are presented. This work allows choosing during run time between more simple processors or fewer, more powerful processors, whatever is better suited for the current workload. However, the authors concentrate on processor cores, which are more appropriate for control-flow workloads, and no comparison to dataflow architectures is made.

3.2 Adaptive Dataflow Architectures

Processors excel at handling the instruction fetching, decoding, and branching required by various software tasks. In contrast, many performance-critical algorithms, notably in media processing and DNNs, feature highly regular computations and predominantly linear control-flow, often processing large streams of data. The architectural overhead that grants GPPs their versatility can become a bottleneck for these specific, data-intensive workloads. Consequently, we now examine architectures tailored for such computational patterns, specifically focusing on their mechanisms for achieving run-time adaptivity to enhance versatility.

There are many specialized, dataflow-driven architectures in the scientific community. These designs excel at their individual use cases, but in most cases, cannot be transferred to other applications. In the following, we will look at some promising designs that can be used flexibly for different workloads thanks to their generic structure.

REVEL is a hybrid parallel processor architecture, which leverages the concept of systolic arrays for inductive matrix algorithms [51]. Inductive algorithms are characterized by a dependence of iteration counts of inner loops on outer loops. If the inner loop is inductive, the loop boundaries are dependent on results from the outer loop. Such workloads are hard to map to systolic architectures, since mapping and scheduling cannot be fully determined prior to execution.

REVEL approaches this issue by implementing two types of PEs, *dataflow* and *systolic* PEs. Systolic PEs are simple compute units that execute whenever valid input data is present. Dataflow PEs implement an ISA for stream processing, contain a register file and allow scheduling instructions dynamically, according to data dependencies. They are 5 × more expensive in terms of area, compared to their systolic counterpart. Figure 3.4 shows a simplified version of the architecture with four systolic and only two dataflow PEs. Accompanied by a GPP for execution control and a toolchain with dataflow-aware compilers, it achieves a speed-up of 11 × to 37 × compared to a

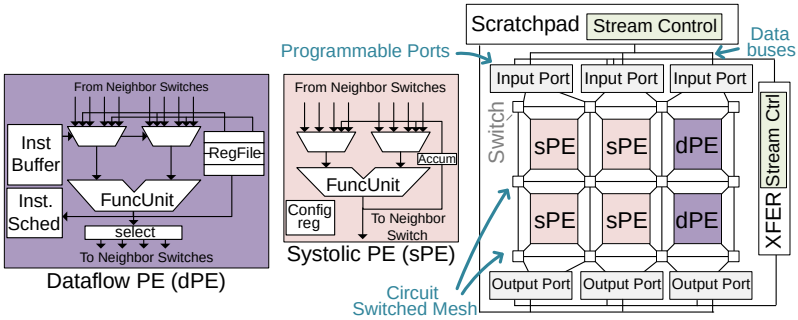


Figure 3.4: The *REVEL* architecture with both a processor-like Processing Element (PE) and a simple dataflow-driven PE. The NoC comprises few dataflow and several systolic PEs and allows reconfiguring the input and output data path [51].

Digital Signal Processor (DSP) with similar size. Apart from configurable switches for data input and output connected to the NoC, the architecture does not feature reconfigurability on hardware level. Nevertheless, it demonstrates a sophisticated hybrid of parallel processors and systolic arrays, increasing performance, but also versatility and programmability compared to static architectures.

The *MAERI* architecture focuses on reconfigurable interconnects [52]. It is specifically built for acceleration of DNNs, which commonly have many different types of layers with varying parameters. This leads to different dataflow patterns within an accelerator. The core contribution of *MAERI* (Figure 3.5) is a reconfigurable *Distribution Tree* and *Augmented Reduction Tree*, which distribute and collect data from and to the PEs in the accelerator. The Distribution Tree allows to either pass individual input data to a single PE or use broadcasting modes to supply the same input data to multiple PEs, which is required in many DNN layers. It is a binary tree, where both leaves get half the bandwidth of the root node, so that all PEs benefit from the full bandwidth of the root. The Augmented Reduction Tree is an enhanced binary tree of adders, which also allows the result from a neighboring adder to be incorporated. Its unique interconnect allows for very high utilization of the PEs, up to 95 %. Compared to *Eyeriss*, it is more area efficient since it does not require addressable local register files. It is, however, less energy efficient due to less data re-use and therefore increased communication on the interconnect. In summary, this work benefits a lot from focusing on the dataflow

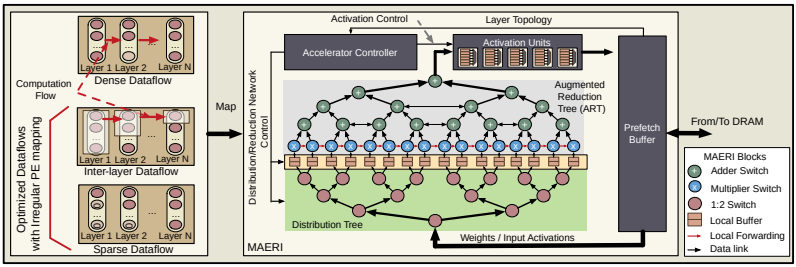


Figure 3.5: The *MAERI* microarchitecture for DNN acceleration. It uses tree-based interconnects to connect to the PEs, allowing to alter the dataflow during run time to adapt to different layers of a neural network.

required for different neural network layers. However, it remains unclear whether it is suitable for workloads requiring operations distinct from MAC.

Although the authors of *MAERI* do not mention it, their architecture resembles a one-dimensional CGRA. See Section 2.2.3 for an introduction to CGRAs. The set of operations supported by a CGRA is limited to instructions supported by the individual PEs. Common arithmetic and binary operations are usually supported, but sophisticated custom operations or arbitrary logic cannot be mapped. Fully-featured PEs result in a large area overhead of CGRA designs. Consequently, specialized designs are frequently engineered, with operators that are more closely tailored to the intended use. *CGRA-ME* is a framework to model and explore specific CGRA designs by specifying the architecture on a higher level [24]. The resulting hybrid platform commonly incorporates a RISC-V core to improve the versatility and allow the execution of standard software. Common sizes of generated architectures are 4×4 to 8×8 due to the area requirements of the PEs. This makes them considerably smaller than comparable accelerators with systolic arrays, which contain up to 1 000 PEs [54]. Performance improvements of up to 37 × were achieved based on a RISC-V software baseline. Compared to execution on a processor equipped with similarly-sized vector units, performance was improved by 2.6 × due to spatial parallelism.

3.3 Deep Neural Network Acceleration

The prevalence of DNNs has increased markedly in the last ten years. Due to their substantial computational demands and inherent high power consumption, DNNs

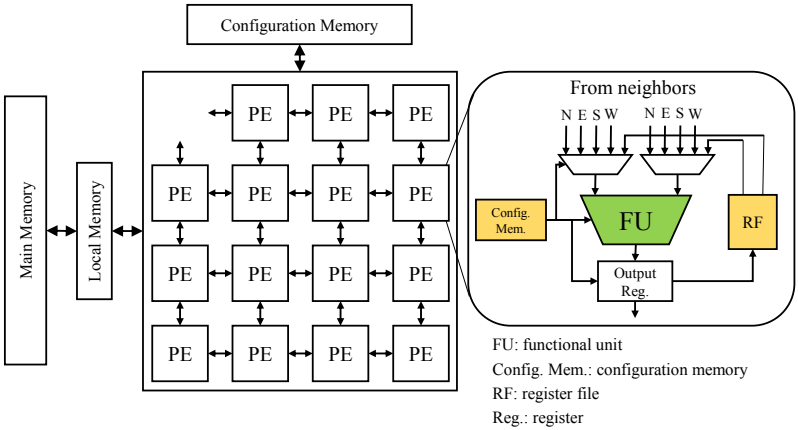


Figure 3.6: Generic example for a CGRA architecture with homogeneous processing elements. Multiplexers allow configuring the dataflow between the PEs. Figure from [55].

are prime targets for specialized hardware acceleration. Convolutional Neural Networks (CNNs), in particular, have so far been the dominant architecture for tasks like object detection and image segmentation. Given that convolutional layers often constitute upwards of 90 % of the computational operations within CNNs, numerous accelerator architectures have been developed in both industry and academia specifically targeting these layers [56]. A key driver for this development is the potential for dedicated hardware to significantly improve energy efficiency. Reuther et al.'s *Lincoln AI Computing Survey* (*LAICS*) offers a valuable survey of contemporary DNN accelerators, focusing specifically on power efficiency (Ops/W). Their findings reveal that while datacenter accelerators often achieve the highest peak performance, they generally operate at lower power efficiency. Figure 3.7 gives an overview of the compared architectures, comparing their peak performance and respective power consumption. A cluster of high-performance architectures forms at a power efficiency of 1 TOps/W, representing datacenter applications. In contrast, the design space achieving high power efficiency (i.e., 10 TOps/W to 100 TOps/W) is less populated, featuring primarily designs developed within academic settings.

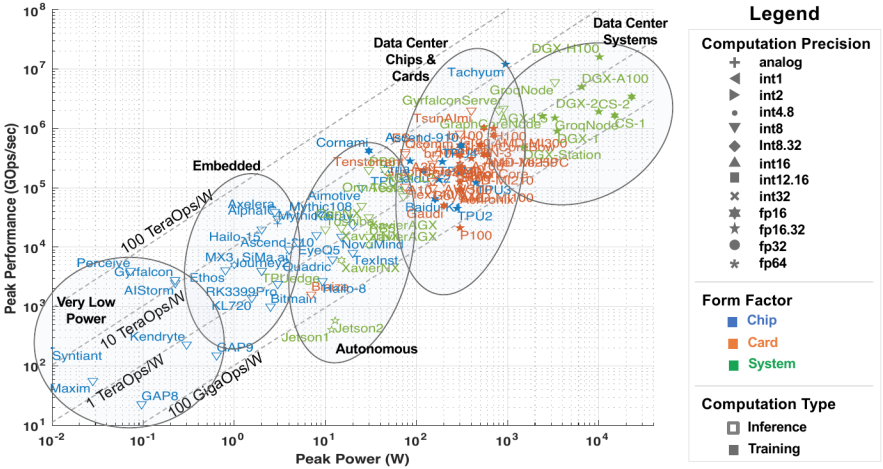


Figure 3.7: Comparison of recent Deep Learning Accelerators (DLAs) with respect to peak performance and peak power consumption. Original data collected by Reuther et al. [57].

3.3.1 Commercial DNN Accelerators

Hardware acceleration for both neural network training and inference is now a common feature in commercial chips. A primary distinction exists between accelerators designed for large-scale datacenter deployments (often synonymous with cloud infrastructure) and those optimized for resource-constrained embedded systems. In the High-Performance Computing (HPC) sector, datacenter GPUs remain the predominant platform for both training and inference tasks. Consequently, major GPU vendors like NVIDIA and AMD have integrated specialized neural network accelerators. NVIDIA's latest *Blackwell* architecture, for example, features Tensor Cores supporting the *FP4* data type, reducing memory requirements by half compared to *FP8* [21]. Similar advancements are occurring in PC processors to enable local DNN execution, as for example with AMD's integration of their *XDNA* Neural Processing Unit (NPU) architecture across various embedded and laptop Central Processing Units (CPUs) [58]. One of the first specialized DNN accelerators was the *Google Tensor Processing Unit* (*TPU*), exploiting the low complexity in control logic and high degree of both parallelism and data re-use possible in CNNs [59]. Despite supporting

only 8 bit integer operations, it outperformed a GPU in performance per watt by a factor of 29 × .

When looking into the embedded field, several architectures are available from both big tech companies and start-ups. *NVDLA* is a commercial, yet open source accelerator architecture for deep learning by NVIDIA [60]. Individual building blocks are provided to accelerate common CNN operations including convolution, fully-connected, activation, pooling, and normalization layers. Its computational units are implemented using multiple parallel vector units, in contrast to systolic architectures which forward data in a uniform, spatial fashion. NVDLA offers a wide configuration space including the convolution mode of operation, configurable hardware parameters and the option to include additional components. An additional Static Random Access Memory (SRAM) interface can be configured for low-latency access to input data, and a closely-coupled microcontroller can be added to offload tasks from the main CPU. This allows tuning the accelerator according to available resources and given workloads. The design is silicon-proven as it is integrated into the *Jetson Orin* System-on-Chip (SoC) series. However, the majority of its DNN performance still comes from the integrated GPU, which achieves 83.5 TOp/s compared to NVDLA's performance of 43.5 TOp/s [61]. Reconfigurable logic is not used in the design, as the manufacturer targets fully-custom ASIC integration, and the set of supported operations is fixed. With *X-NVDLA*, an architecture based on NVDLA featuring run-time configurable accuracy has been presented [62]. The approximate computing technique voltage overscaling (VOS) is used to reduce accuracy on the least significant bits (LSBs) of input and output data, both in the computational units and the SRAM. Figure 3.8 shows the structure of the MAC array in the convolution unit of NVDLA. Fine-grain configurable voltage supplies have been added to each multiplier, allowing to influence the accuracy of their operations. Energy consumption is reduced by about 35 % with a 3 % reduction in accuracy, demonstrated using an implementation in 15 nm FinFET technology.

The original NVDLA design is aimed towards embedded systems. *Simba* takes the basic design a step further to create a high-performance, full-chip inference accelerator [63]. A Multi-Chip Module (MCM) is used to create a chip from 36 chiplets, each containing several PEs, a management processor, memory, and communication interfaces. Each PE uses a simplified version of NVDLA implementing a Weight Stationary (WS) dataflow. On the chip built in 16 nm FinFET technology, the PEs are equipped with 32 KiB and 8 KiB of weight and input buffer each, and are clocked with up to 1.8 GHz. It is shown that a greedy mapping of a layer from the ResNet-50 CNN bottle-

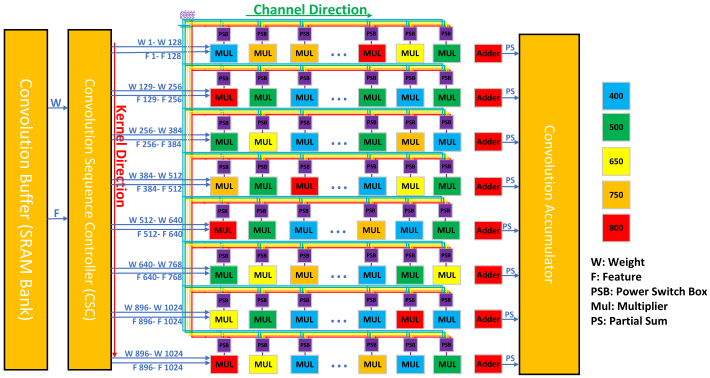


Figure 3.8: Structure of the MAC array of NVDLA enhanced with configurable voltage inputs for each multiplier unit [62].

necks when using more than 8 PEs on a single chiplet due to bandwidth limitations. An improved non-uniform workload partitioning approach is presented, leveraging the multi-level hierarchy of PEs and chiplets. Together with a communication-aware data placement mechanism, performance could be increased by 16 %. The full package achieves 128 TOp/s with a power efficiency of 6.1 TOps/W. Running ResNet-50, the authors were able to demonstrate 1 988 images/s with an inference latency of 0.50 ms.

Further examples for acceleration of DNNs can be found in modern smartphones, where on-device inference such as face detection or voice recognition is supported by NPUs on the device. The performance gains are significant: Snapdragon Mobile Platforms boosted the performance from 3 TOp/s on the Snapdragon 845 in 2018 to 26 TOp/s on the Snapdragon 888 in 2021 [64]. In addition to SoCs with built-in NPUs, external add-on modules like the *Hailo-8* are available to equip existing systems with DNN acceleration [65]. Using a standardized PCI Express (PCIe) interface, they interface to many Commercial-Off-the-Shelf (COTS) platforms, and allow making use of the SoC's components like the DRAM controller. Nevertheless, the high specialization comes at the cost of decreased flexibility, as these designs may not be usable for upcoming DNNs of the next years.

3.3.2 Academic DNN Accelerators

In contrast to the approach in industry, research on DNN accelerators has a stronger emphasis on the embedded field and, consequently, efficient computation [66]. The majority of current CNN accelerators utilize a systolic array comprising PEs arranged in a 2-D configuration [54]. The performance of systolic-based accelerators varies considerably based on the size of the array, workload mapping to the PEs, and the dataflow strategy employed [67; 68]. Depending on the network type and layer size, different choices may be optimal. Consequently, a considerable number of works present accelerator frameworks with the capacity to configure numerous hardware parameters.

For example, *Gemmini* is an academic open-source framework to generate DNN accelerators in the form of systolic arrays [69]. The framework can not only generate an accelerator, but also provides a full system-level SoC design (Figure 3.9). It comprises a RISC-V processor which is closely coupled with the systolic array, scratchpad memory and control logic. If processor integration is selected, the *Chipyard* SoC ecosystem is used to provide processors and peripherals [70]. Design Space Exploration (DSE) is supported to tune the design for a given context, adjusting the array size of the spatial array and defining properties of the scratchpad memory and Direct Memory Access (DMA) units. *Gemmini* maintains a high flexibility for the system designer, allowing to select between two different design approaches for the spatial array: it can either be a set of parallel vector engines, similar to NVDLA, or a systolic array with fully pipelined PEs as in TPUs. The systolic array dataflow can be set to either Output Stationary (OS) or WS dataflow dynamically during run time. In addition to the hardware generation framework, it comprises a software stack to run DNN workloads in Open Neural Network Exchange (ONNX) format. It also allows low-level access to execute command primitives on the spatial array directly, increasing programmability for non-standard operations. The framework does not include any run-time reconfigurability, as its design philosophy is to tailor its architecture template to the target use case.

Typical layers of CNN are too large to fully fit into hardware accelerators. The convolution problem is therefore tiled into sub-problems which are scheduled temporally on the architecture. This opens up the design space for different scheduling methods, which yield different utilizations depending on hardware and software parameters. Refer to Section 2.3.2 for an overview of loop nests for multidimensional convolutions. Tu et al. present a CNN accelerator with reconfigurable dataflow for different map-

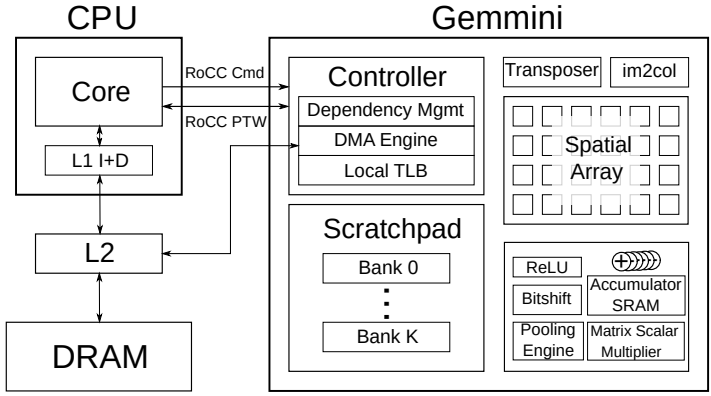
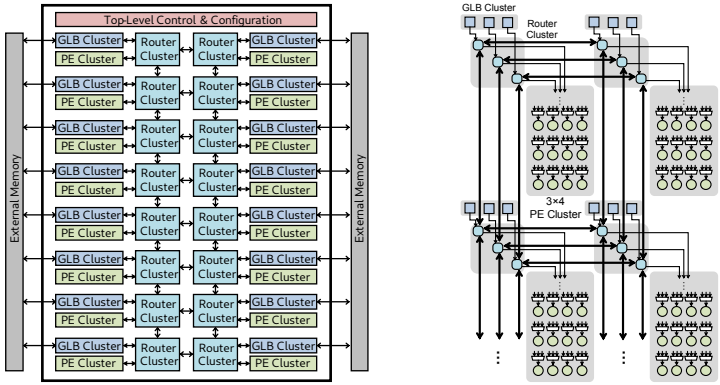


Figure 3.9: The hardware architecture of *Gemmini*. It allows generating a custom spatial array accelerator, which is closely coupled to a RISC-V processor [69].

pings [71]. Emphasis is put on data re-use, which improves energy efficiency by 5.9 × to 8.4 × compared to an approach without data re-use. The accelerator supports three different scheduling strategies for convolution layers, keeping either input, output or weight data local during computation. In order to find the optimal mapping, a simulation of Dynamic Random Access Memory (DRAM) and internal memory accesses is performed. The strategy with the lowest energy consumption and highest utilization is selected, which leads to a mean utilization of 93 % for the AlexNet, GoogLeNet, and ResNet models. The data path and the PEs are reconfigurable during run time to support all three mapping strategies. However, it is configured using designated multiplexers and reconfigurable hardware is not used. An ASIC implementation in 65 nm technology achieves 194.4 GOp/s at 200 MHz with a power efficiency of 152.9 GOps/W.

Eyeriss V2 is a scalable accelerator for DNNs on embedded devices [72]. Based on their previously published accelerator core, the authors employed a NoC to deploy multiple instances of this core on a chip, each called a cluster. The NoC consists of three data paths, one for each data type used in a common DNN: input activations, weights, and the output feature map. Figure 3.10b shows the NoC architecture. It implements unicast, multicast and broadcast modes to distribute input data efficiently to one or many clusters, to avoid transferring the same data multiple times. The authors can show that their original *Eyeriss* design does not scale well beyond 256 PEs, whereas the NoC allows overcoming this limitation. With 16 384 PEs, *Eyeriss*



(a) Top-level architecture (b) Network-on-Chip

Figure 3.10: Overview of the *Eyeriss V2* architecture. A specialized NoC is used for input feature, weight and output feature map data.

V2 runs 57.4 × faster than with 256 PEs, which corresponds to a scaling overhead of only 10.3 %.

Inside a PE cluster, the Row-Stationary (RS) dataflow is being used. It maximizes data re-use inside the cluster, thus minimizing energy consumption when compared to the OS or WS dataflow. This leads to each PE requiring large local buffers to store a full row of the output feature map, compared to only a single accumulator for the OS dataflow. However, by means of an adapted Roofline Model and DSE, the buffer size is carefully tuned and the resulting 65 nm chip uses only 246 kB of SRAM memory. The architecture achieves a 42.5 × and 11.3 × improvement on inference of a sparse AlexNet compared to its predecessor. In absolute numbers, benchmark results of 278.7 and 1 470.6 inferences per second are shown. *Eyeriss V2* does not allow reconfiguration on architectural level. The architecture is specialized in DNN acceleration, which is demonstrated by the separation of the three data types both in the NoC and in the global buffer memory. However, it allows for configuration of the NoC to adapt to different dataflow requirements. The authors stress that a single systolic array has scalability limits, and additional dataflow layers like a NoC have to be added when building a larger design.

RiSA extends concept of systolic arrays for CNN processing to support additional network types [73]. Some modern CNN architectures adopt depthwise separable convolutions, motivated by reductions in computational cost and model parameters.

A key characteristic of this operation, however, is its inherently lower potential for data re-use compared to standard convolutions. This poses a considerable challenge for hardware accelerators like *Eyeriss V2*, which are built to leverage high degrees of data re-use for efficiency. The *RiSA* accelerator specifically targets this limitation by introducing architectural extensions that enable high utilization even when processing depthwise convolution layers. Beyond CNNs, the framework demonstrates versatility by providing efficient mapping techniques for other critical operations, such as the large matrix multiplications fundamental to Transformer networks. A notable feature is the inclusion of intra-PE-array data reshaping capabilities, eliminating the requirement for off-chip or dedicated hardware units for data reordering. In the NVDLA architecture, such a data reordering module accounts for 14 % of the convolution unit area. Benchmarking against *Eyeriss V2* reveals substantial advantages: a 1.44 × performance increase, a 1.91 × area reduction, and a 1.31 × improvement in energy efficiency, achieved using a WS dataflow. The specific hardware augmentations for depthwise support incur relatively small overheads, adding 6.4 % to the area and 1.1 % to the power budget relative to a configuration lacking these features. *RiSA* serves as a good example illustrating how accelerator architectures must be extended with additional features to accommodate the diverse computational demands of novel neural network architectures, in addition to traditional convolution operations.

It can be concluded that there are several dataflow-driven accelerator architectures, primarily focused on processing of DNN. Current work places particular emphasis on scalability and maximum memory bandwidth to avoid being memory-bound. NoCs are a common choice for connecting computational units, whether there are processors or accelerators. Systolic arrays are a suitable structure to enable massive parallel computation, yet other approaches using binary trees are also feasible.

3.4 Summary and Open Challenges

The landscape of hardware acceleration encompasses a diverse range of architectures designed for computationally intensive tasks. Broadly, these can be categorized based on their primary execution model. One category includes run-time reconfigurable processor architectures, often targeting control-flow dominated applications. These typically integrate a general-purpose processor with tightly coupled reconfigurable fabrics or parameterizable vector units for efficient parallel execution. Another significant category comprises dataflow-driven architectures optimized for stream

processing. Many prominent examples, such as Gemmini and Eyeriss, leverage systolic arrays to exploit spatial parallelism, while alternative designs like MAERI and NVDLA employ parallel processing units interconnected with memory hierarchies, often via tree-like structures. Common to all these high-performance architectures is that leveraging spatial parallelism is essential for handling modern workloads.

The degree of reconfigurability observed in these systems varies significantly, ranging from adaptive interconnects within processors or dataflow fabrics to architectures incorporating extensive FPGA resources for implementing arbitrary hardware functions. Two primary trends emerge from prior work: the use of adaptive circuits targeted at performance optimization, and the integration of large FPGA fabrics to maximize functional flexibility.

To give a specific example, we looked closely at machine learning acceleration, which currently is a major focus in the scientific community. Numerous specialized architectures have emerged from both academia and industry, characterized by high degrees of parallelism explicitly tailored for efficient DNN execution. Energy efficiency is consistently identified as a major design constraint across these accelerators, particularly critical for deployment in resource-constrained embedded systems like mobile devices. Given that external memory accesses (e.g., to DRAM) constitute a significant fraction of total energy consumption, a primary focus of many architectural optimizations is the minimization of data movement. Maximizing on-chip data reuse is a widely adopted technique, particularly effective for CNNs. Other strategies include exploiting model sparsity through techniques like pruning, which enables skipping or compressing redundant computations and data. From the technological point of view, power consumption can be reduced by using energy-efficient memories. Figure 3.11 highlights the access energy efficiency of High-Bandwidth Memory (HBM) compared to preceding high-bandwidth DRAM technology. Given that the energy budget, particularly in the context of AI applications, is predominantly driven by memory access, the integration of efficient storage technologies is crucial for ensuring effective processing [74].

However, despite rapid advancements in neural network models, many existing accelerators are highly specialized for operations prevalent in current network types (e.g., CNNs). As research trends evolve, such as the current shift from CNNs to transformer networks, existing specialized hardware architectures may face challenges in being compatible with novel algorithms. Consequently, new architectures will need to be developed, potentially leading to the rapid obsolescence of previous designs. The in-

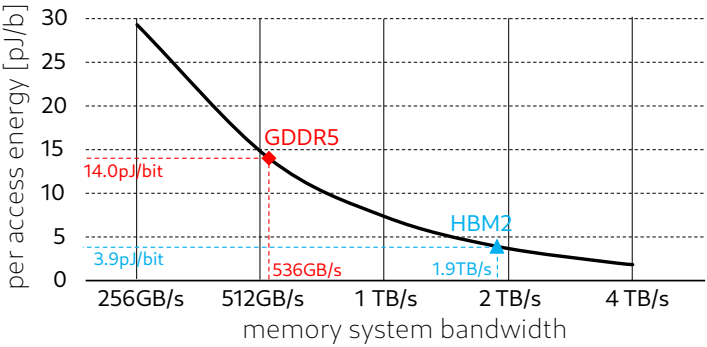


Figure 3.11: DRAM access energy of GDDR5 compared to HBM2 within a 60 W power budget. Figure from [74].

troduction of reconfigurable hardware components can help to improve performance and flexibility. A common approach is to variably connect multiple compute units, like the coupling of vector units to enhance vector processing performance of a single CPU as described by Perotti et al. [48]. Generally speaking, the family of CGRAs does also follow this paradigm by implementing a set of PEs with configurable interconnect [55]. On the large scale, we have observed the use of general-purpose NoCs allowing arbitrary communication patterns between nodes at the cost of resource overhead of routers [50]. However, the supported operations of the PEs are fixed in all the works that have been considered. The impact of fine-grained embedded FPGA fabric for increased flexibility of computational units remains to be assessed, representing a research gap in the field.

Addressing this limitation, the research presented in this thesis aims to enhance the versatility of hardware accelerators to better accommodate future, potentially unforeseen, workloads. While maintaining computational performance and energy efficiency remains important, the primary objective shifts towards improving flexibility. Conventional FPGA-based systems allow for full customization but typically suffer substantial performance and power penalties. Therefore, we investigate design methodologies for creating hardware architectures that strike a balance between the efficiency of hard logic and the flexibility of reconfigurable logic. A core principle is the strategic integration of reconfigurability specifically at points within the architecture where adaptability offers the most significant benefit, aiming to prolong hardware lifespan and reduce electronic waste. Additionally, ways to integrate novel

energy-efficient memory technologies are investigated to reduce power used by applications with high bandwidth requirements. The proposed approaches continue to follow the distinction between control-flow and dataflow dominated workloads, building upon methods for workload characterization detailed further in Chapter 4.

Chapter 4

Approaches for Acceleration on Adaptive and Heterogeneous Platforms

The selection of the target hardware platform is one of the initial and most important design decisions when developing an embedded system. Numerous different microcontrollers, System-on-Chips (SoCs) and multi-chip computing platforms are available, with large differences with regard to performance, power consumption, available interfaces and cost. Driven by higher abstraction and increased automation of Electronic Design Automation (EDA) tools, this number grows faster year by year, making specialized platforms more popular and available to a broad market [75].

One property of hardware platforms is the availability of hardware accelerators, which are increasingly common on platforms of all sizes, ranging from microcontrollers to embedded high-performance platforms [76]. Integration of acceleration for applications like image processing, high-speed communication and processing of Deep Neural Networks (DNNs) enables edge computation by increasing processing power for a specific workload while maintaining low power consumption. The preferred interface between the hardware accelerator and the processing system depends on several factors. It is influenced by the type of operation being accelerated, the origin and destination of the data, and the overall design methodology. Monolithic SoC integration can make use of third-party Intellectual Property (IP), Application-Specific Instruction-Set Processor (ASIP) designs allow for custom processor cores, whereas individual chips use standard interconnects like PCIe on board-level designs. Examples include Image Signal Processors (ISPs) that are commonly located between dedicated camera and DMA interfaces on systems specialized for video processing. In contrast, Instruction Set Architecture (ISA) extensions for Single Instruction Multiple Data (SIMD) operations like AVX512 are naturally integrated in the pipeline

and closely coupled with the CPU core. They are not only found in high-end server processors, embedded system ISAs also specify extensions for vector processing [77].

This work focuses on embedded systems and investigates three different approaches:

1. extending a Central Processing Unit (CPU) core with custom accelerated instructions
2. directly integrating an accelerator IP into a full-custom SoC
3. attaching the accelerator to an existing SoC via a high performance PCI Express (PCIe) link.

Hardware/Software Co-Design can be applied to decide on which approach is most suitable for a specific design. In this chapter, challenges and potential issues of these approaches are discussed. This includes an analysis of the target workload type, as well as additional parameters such as manufacturing technology, available interconnect mechanisms, and non-functional parameters including safety requirements.

4.1 Advantages of Adaptive Platforms

The demand for increased processing power is at an all-time high. Machine learning is currently one of the strongest factors in demanding more performance from computers. However, physical boundaries hindered continued performance improvements through architecture shrinks and increased frequencies. Instead, further development is diverse. On the one hand, the focus is shifting to improvements in microarchitecture. On the other hand, the architectures are being scaled up to larger areas, thus further increasing the number of transistors. However, the resulting improvements do not meet the requirements of modern architectures well enough. For this reason, specialized architectures are becoming increasingly common and replace generic computing platforms in many domains. Examples for such specialized architectures are Graphics Processing Units (GPUs), Digital Signal Processors (DSPs), ASIPs, and even ISA extensions of General Purpose Processors (GPPs) can be seen as a specialization. But specialization comes at a cost: platforms are far less versatile and rarely reusable for different workloads. For this reason, we see the need for *adaptive hardware*. Similar to the flexibility that we already know in the area of software, hardware must also be able to adapt to future workloads. Field Programmable Gate Arrays (FPGAs) (Section 2.2.4) offer this kind of adaptivity by allowing run-time reconfiguration. Practical examples for architectures making use of FPGA fabric are the *Xilinx Zynq*

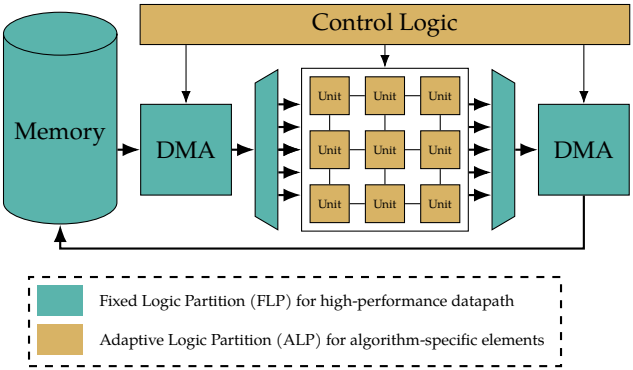


Figure 4.1: Example for a dataflow architecture with high-speed fixed logic in the datapath. Less performance-relevant, but behavior-defining elements are FPGA-based.

and *Versal* SoCs. They offer different kinds of fixed compute logic, such as GPPs and GPU cores, with a portion of adaptive FPGA hardware. This already indicates the need for a combination of both approaches: GPPs can be used very efficiently for standard tasks, while special tasks can be implemented in the adaptive part with lower performance per domain. The reduced performance of FPGAs can often be mitigated by parallelism, if enough resources are available. The disadvantage of these architectures is the higher cost due to the increased chip area for the large number of FPGA resources and fixed logic blocks that may not be used. Additionally, the coarse-grained partitioning requires the designer to choose the target architecture early in the Co-Design process, requiring upfront knowledge or leading to back-and-forth data transfers if multiple partitions are used.

The basic idea for this work arises from these difficulties: designing architectures with a fine-grained interleave of hard logic and reconfigurable logic to achieve the optimal trade-off between performance, efficiency and flexibility. This approach allows for high clock frequencies and high integration density in the hard logic partitions, referred to as the Fixed Logic Partition (FLP). The Adaptive Logic Partitions (ALPs), containing FPGA fabric, offer high flexibility through reconfiguration during run time. Designing such a hybrid system immediately raises several questions: Which logic needs to be implemented as hard logic? In contrast, which parts of the logic should be implemented in the ALP to significantly improve flexibility? This trade-off depends heavily on the base design of the system. An enhanced GPP will need

the core pipeline to be fully in the FLP to achieve reasonable performance, whereas parts of a streaming data processor can be implemented as ALP and make up for the performance penalty through parallelization. Figure 4.1 depicts an example dataflow architecture containing both ALP and FLP components. The partitions are not a single, monolithic block of Application-Specific Integrated Circuit (ASIC) and FPGA logic each; rather, the components are assigned to one of the partitions in a fine-grained manner. If the ALP regions are appropriately assigned, the architecture can be flexibly adapted to a wide variety of algorithms. Performance-critical components in the FLP are fixed to guarantee maximum bandwidth.

Choosing the basic architecture of a hybrid adaptive system is an early but critical step in the design process. For the system to be both powerful and flexible for future use cases, the FLP compute logic must be generic and specialization to the current use case is handled by the logic in ALPs. Section 4.2 presents a methodology to analyze current and possible future workloads as a tool in the design process to select a suitable basic architecture. Next follow detailed descriptions of two principal architecture classes that are differentiated in this work. For both classes, specific examples for additional functionalities for the use in embedded systems are given.

4.2 Methodology for Workload Classification

4.2.1 The Roofline Model

In 2009, Williams et al. proposed the *Roofline Model*, a visual performance model allowing to characterize computing systems regarding their memory and compute performance [78]. It was originally intended for floating-point applications on multicore processors, but its main idea is easily transferable to other computing architectures. In order to find a relation between processor performance and the—often limiting—off-chip memory bandwidth, the term Operational Intensity (OI) is defined:

$$OI_{workload} = \frac{\text{FLOPs}}{\text{Byte}}$$

A high OI of a workload indicates more computational operations per byte fetched from main memory. The performance of a processor is limited by the maximum number of FLOPs/s it can achieve. The model connects floating-point performance

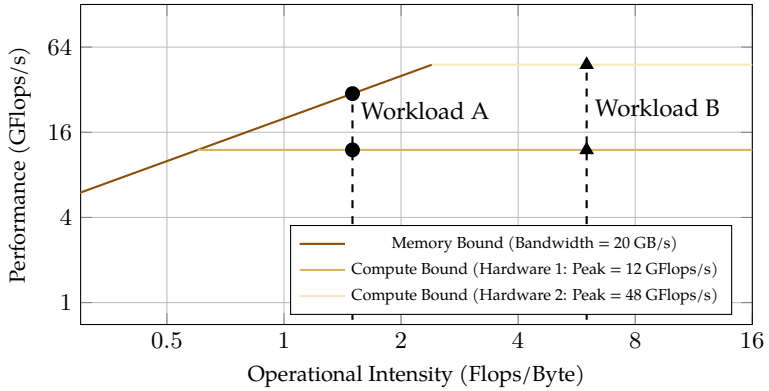


Figure 4.2: Example for a Roofline Model of two imaginary compute architectures

in FLOPs/s, the OI, and memory bandwidth to establish two performance bounds: the memory bandwidth bound and the compute bound.

Figure 4.2 is the resulting two-dimensional graph of plotting OI versus performance in FLOPs/s. Two fictional hardware architectures with peak floating-point performance of 12 and 48 GFLOPs/s and the same memory interface with 20 GB/s are shown. The performance of a workload with an OI of 1.5 FLOPs/Byte is compute-bound on hardware 1, but memory-bound on hardware 2. Example workload *B* shows a higher OI and is therefore compute-bound on both architectures.

In order to set up the model quantitatively, computational performance and memory bandwidth of the hardware as well as OI of the workload must be determined. However, if this is not feasible, the model can also be used qualitatively, allowing the system designer to compare workloads with respect to their complexity more easily. For example, it can be applied to integer arithmetic similarly. The performance of integers is generally higher than that of floating point numbers, which causes the algorithms to be more memory-bound. Many Machine Learning (ML) algorithms have a very low OI and therefore lie in the left-hand area of the Roofline Model, which illustrates that they are memory-bound on many platforms.

The principles of the Roofline Model can be used by system designers to determine a suitable target platform. In the early stages of the development process, the algorithmic kernel of the target workload can be outlined in pseudocode. The estimate of the

kernel's OI provides an indication of whether the target platform should focus on memory bandwidth or computing power.

Algorithm 1 shows the basic Gaussian filtering algorithm. An input image of $H \times W$ pixels is filtered with a Gaussian kernel of $K \times L$ elements. For each input pixel, excluding borders of $K-1$ in vertical and $L-1$ in horizontal direction, an element-wise multiplication with the kernel is performed. The sum of the multiplication results is the value of the corresponding output pixel. In total, $K \cdot L$ Multiply-Accumulate (MAC) operations are performed for each input pixel. Assuming 32-bit float values and cache locality for the Gaussian kernel, the OI of this algorithm is:

Algorithm 1 Simplified pseudocode for Gaussian filtering

```
Input: image I[H,W], Gaussian kernel G[K,L]
Output: filtered image E[P,Q]
  for y,x in range H-K,W-L do
    S = 0
    for gy,gx in range K,L do
      S+ = I[y + gy, x + gx] * G[gy, gx]
    end for
    E[y, x] = S
  end for
  return E
```

$$OI_{gauss} = \frac{K \cdot L \text{ FLOPs}}{4 \text{ Byte}}$$

For a typical kernel size of $K = L = 5$, this results in an absolute value of $OI_{gauss} = 6.25$ FLOPs/Byte.

In contrast, estimating the OI becomes far more difficult for algorithms with more control-flow operations. Algorithm 2 describes a simplified version of the SUSAN edge detection algorithm [79]. Similar to the Gaussian filter, an input image $I[H, W]$ along with a filter mask $M[K, L]$ is provided. Instead of a MAC per filter element, the intensity of each neighboring pixel is evaluated and compared to the center pixel. The number of pixels with an intensity delta larger than threshold T is counted, determining the size of a *USAN*. This size is used to compute the edge response in conjunction with the *geometric threshold* G, which is used only if positive. Some assumptions are necessary to determine the OI: Cache locality for M, a single operation to determine I, the use of 32-bit float values, and average distribution of *USAN* and non-*USAN* pixels are assumed.

$$OI_{susan} = \frac{K \cdot L \cdot 4 + 4 \text{ FLOPs}}{4 \text{ Byte}}$$

For comparison with the Gauss algorithm, choosing the same kernel size of $K = L = 5$, this results in an absolute value of $OI_{susan} = 26$ FLOPs/Byte. One can easily recognize that the more complex inner kernel of the algorithm leads to a substantially higher OI.

The actual OI of an algorithm cannot be determined in this way, but the estimate provides a proportional comparative value between different algorithms. As a decision aid, it can be deduced that processor-based architectures can be selected for high OI, which are more efficient for control-flow driven workloads. For low OI, a dataflow-driven architecture is more suitable, allowing the designer to focus on addressing the memory-boundedness of the workload.

Algorithm 2 Simplified pseudocode for the SUSAN algorithm [79]

Input: image I[H,W], thresholds T,G, mask M[K,L] with radius R
Output: edge feature map E[P,Q]
  for y,x in range H-R,W-R **do**
    $n_{USAN} = 0$
    for dy,dx in range K,L **do**
      $ny, nx = y + dy, x + dx$
      if $abs(I[y,x] - I[ny,nx]) <= T$ **then**
        $n_{USAN} += 1$
      end if
    end for
    $Response = 1 - (n_{USAN}/G)$
    if $Response > 0$ **then**
      $E[y,x] = Response$
    else
      $E[y,x] = 0$
    end if
  end for
  return E

4.2.2 Control-Flow and Dataflow-Dominant Workloads

When looking at a specific workload, we almost always see a mix of control-flow and dataflow parts. In Section 4.2.1, a method for comparing algorithms based on their pseudocode representation was introduced. If this approach is not applicable, e.g. due to the high complexity of the algorithm, a more detailed analysis must be carried out. Breaking down a complex workload commonly yields high-level control-flow logic connecting several computationally intensive algorithmic kernels [80]. Such kernels include Fast Fourier Transform (FFT), General Matrix Multiplication (GEMM), and Convolutional (CONV), which benefit from hardware acceleration in most cases. For example, FFT is used in radar algorithms to process received raw antenna data and obtain point clouds for further processing (see Figure 4.3).

Figure 4.3: FFT kernels are used in radar data processing to obtain a Range-Doppler-Time point cloud. Image adapted from Ahmed et al. [81].

The FFT kernels take up a significant portion of the iteration time and are a candidate for implementation in hardware. Then the FFT data is searched for detected points, and for each point a conversion to Cartesian coordinates and further filtering is performed. The variable number of detected points can result in dynamic data structures and may be a reason to perform this post-processing in software if it is too complex to implement in hardware.

In contrast, common DNNs have a fixed dataflow without any control-flow operations. Each computational kernel, mostly CONV or GEMM layers, is directly followed by the next kernel until the output layer is processed. Section 2.3.2 describes the structure of Convolutional Neural Networks (CNNs), a popular type of DNNs, with a linear dataflow as can be seen in Figure 2.12. Such highly linear dataflows are where hardware implementations perform best. Execution can be massively pipelined as no control-flow decisions stall the execution and data buffers for input and output are fixed. For neural network inference, there are a number of different TPUs from the academic and industrial environment that use a dataflow-driven architecture to solve this problem [57].

Generally speaking, there are several properties of algorithms or kernels which make hardware implementation more difficult.

Irregular or sparse data access Access patterns that involve accessing data in complex order are usually generated by a software algorithm with elaborate control-flow. Hardware implementation of this pattern is a time-consuming task.

Recursive algorithms: When the depth of a recursive algorithm is unknown, saving the state of each step to Dynamic Random Access Memory (DRAM) reduces performance and is highly optimized in GPPs. Evaluation of lazy functional programs requires specialized processors like the *Reduceron* [82].

Dynamic graph algorithms While graph traversal and analysis can benefit from hardware acceleration [83], altering a graph can require dynamic memory allocation and working with irregular data structures.

Highly dynamic data structures Handling dynamic data structures, where the layout must be derived from the data itself, can result in a high hardware overhead to handle variations of the data structure. Sequential processing using a GPP is more efficient in such cases.

The feasibility of accelerating algorithms with such properties in hardware depends strongly on the situation and cannot be generalized. *Nemesys* by Rheindt et al. is an example for graph operations in hardware [84]. The duplication of an object graph, originally created by the runtime system in software, is offloaded to hardware to improve Inter-Process Communication (IPC) performance. Compared to traversing and copying these graphs in software, run-time performance improved by a factor of 1.35 × to 3.85 × . In contrast, a software implementation can show similar performance when dealing with dynamic data, sparse data, increased recursion depth or larger graphs.

Evaluating the balance of control-flow operations and computational kernels along with their suitability for hardware implementation helps in deciding on a suitable target architecture. It is preferable to implement a workload on a dataflow-driven architecture if it consists mainly of compute-intensive kernels such as DNNs. While workloads with increased heterogeneity between dataflow and control-flow can be fully implemented in hardware, the use of a processor with tightly coupled hardware acceleration is much more flexible at a negligible performance penalty.

So far only the differentiation between software execution on a processor and ASIC hardware implementation has been discussed. Looking at the adaptivity of such systems, only the software parts can be altered after the design phase, as the hardware implementation is fixed in silicon. In the previous considerations, we only examined the algorithms known at design time. In order to assess the suitability for an adaptive system, the components of the algorithm must be analyzed for their reusability. Obviously, there is no oracle that can show us future use cases, but we must ask ourselves which components of the workload could possibly be modified. If the nature of the algorithm allows the top-level software to be changed while the algorithmic core remains unchanged, an adaptive processor platform with the option of software updates is an option. Since a processor is used as the target platform for that algorithm anyway, there is no additional loss of performance for customization. When aiming for a fully custom hardware implementation, adaptivity in the form of FPGA technology always comes at a performance penalty. It is therefore advisable to design a hybrid system with high-performance parts in the FLP. A central task in the implementation of

such a design is to identify components that can be implemented in the ALP and thus significantly increase flexibility for future changes, but hardly reduce the performance of the overall system. A design approach for both variants of adaptive systems is given in Sections 4.3 and 4.4 for processor-based accelerators and dataflow-driven accelerators, respectively.

4.3 Closely Coupled Processor-based Accelerators

A General Purpose Processor (GPP) can be found in many embedded systems. They offer great flexibility through programmability in software, which can also be exchanged in the field. Focusing on software, the system designer benefits from high-level languages such as C++ or Java, and very mature tools such as compilers, debuggers and runtime libraries make it easy to implement functionality. Whilst GPPs are great for control tasks, they quickly reach a performance boundary when it comes to handling data-intensive workloads. Examples for such workloads include signal processing, video encoding and encryption tasks.

A classic approach to solve this issue is to extend the ISA with SIMD instructions. One of the most prominent examples are the MMX extensions introduced by Intel in 1997 [80]. Execution performance of compute-intensive image-filtering kernels increased by up to 3.7 × by using SIMD instructions. In the following years, further SIMD extensions of the x86 ISA followed regularly, from SSE in 1999 to AVX10.2 in 2024. The main difference between the generations is the width of the SIMD registers, which has been continuously increased and is now up to 512 bit [85]. New extensions were needed to keep up with emerging algorithms with increased complexity and novel operations. This highlights a fundamental problem with fixed ISA extensions: modern workloads require new instructions to be added at the cost of more silicon and increased ISA complexity, while old versions must be retained for backward compatibility.

If a required instruction is not implemented, no acceleration is possible at all. Applications can detect this during run time and run a fallback implementation with no SIMD instructions present. In this case, performance is limited, and chip area used for SIMD instructions remains idle, if this is the only workload run on the system.

Processors with SIMD instructions are still register machines and require loading and storing data before and after processing. SIMD registers are larger than their GPP

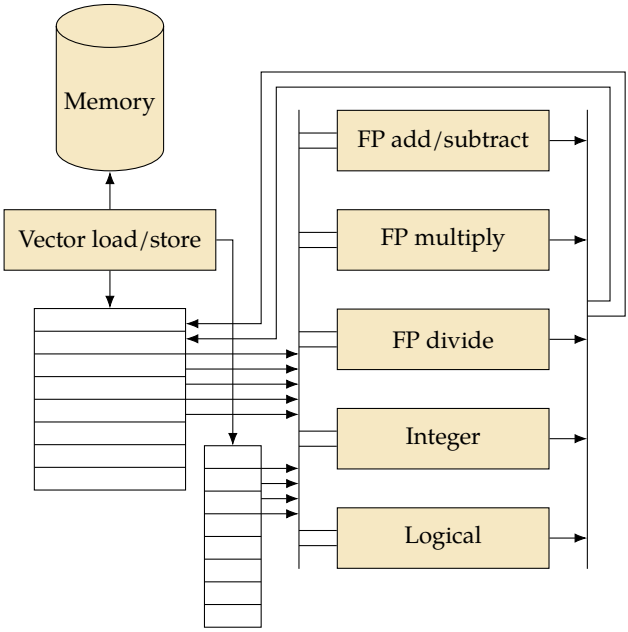


Figure 4.4: Excerpt of a vector processor architecture, VMIPS. SIMD instructions apply the same operation to all elements in a vector register concurrently. Image adapted from Hennessy [7].

counterparts in most ISAs, they can benefit from multi-word transfers like a 16×64 bit burst on DDR5 machines. However, maximum performance can only be reached if data loading, SIMD processing and result write-back are pipelined optimally. This is generally hard to achieve and as a consequence, we cannot universally reach the same performance on a linear data stream as a dataflow-driven architecture due to the memory interface. Speaking in roofline terms, the OI of an SIMD workload is higher and therefore bound by performance sooner.

4.3.1 Acceleration Principle for Control-Flow Workloads

The concept of hybrid processor-based accelerators fuses a GPP with reconfigurable FPGA fabric. This allows for execution of software along with all its benefits, but in addition enables application-specific extensions in the FPGA portion. Accelerator

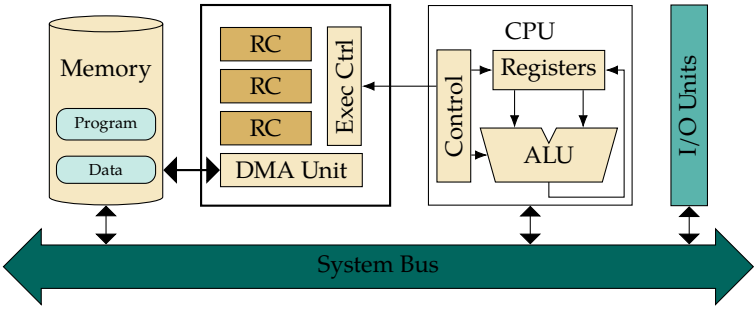


Figure 4.5: A GPP extended with a reconfigurable fabric. Each reconfigurable container (RC) can host a custom special instruction (SI) defined by the application. The Direct Memory Access (DMA) unit uses a distinct high-bandwidth interface to the memory.

and processor are closely coupled, which means that execution of the accelerator is triggered directly from the processor pipeline when encountering an SI. It allows deep integration of the accelerated function with the software, such as including compiler support for SIs and using the register set for parameter input and returning results. Figure 4.5 outlines the hybrid architecture, showing the processor core, caches and bus interfaces as part of the Fixed Logic Partition (FLP) for highest performance. The reconfigurable containers (RCs) are highlighted and form the Adaptive Logic Partition (ALP), hosting application-defined accelerated SIs. This concept allows performance-critical parts of the accelerator fabric to be implemented in the FLP to maintain high performance, e.g. for DMA.

A core principle of this concept is to integrate accelerator usage within the ISA. This means that in contrast to memory-mapped interfaces which are used with generic bus read and write transactions, distinct instructions to interact with the accelerator blocks are provided. A classic approach to implement this are coprocessor instructions, which are present in many old instruction sets. For example, MIPS architectures implement floating-point operations using an Floating Point Unit (FPU) which is accessible using coprocessor instructions [86]. Further processor extensions can include accelerators as additional coprocessors, since MIPS defines instructions for four coprocessors in total, of which two are unused in the standard. Coprocessors have two isolated register sets of 32 registers each, which are accessible using load/store instructions. Operations within the coprocessor can be arbitrarily encoded within 26 bit of the `COPz` instruction. Modern ISAs reserve encoding space for custom instructions.

In the RISC-V ISA, an instruction-set category for *custom* instructions is defined and reserved from use in future extensions of the ISA standard [87]. The opcode mapping for RV32G and RV64G defines four custom sections of 25 bit and 57 bit, respectively. This allows for maximum flexibility when implementing custom SIs. In contrast to MIPS coprocessor extensions, RISC-V custom instructions may also use the standard register file and exchange data by other means than load/store instructions. As a consequence of custom extensions being specified in the ISA, toolchains for RISC-V offer predefined interfaces to extend compilers and related tools with support for SIs.

Accelerator kernels are implemented in an ALP within FPGA logic. Therefore, configuration of the FPGA logic is necessary before use, which is possible either during system initialization or dynamically at run time. Implementation of SIs can be provided by the application, allowing distinct custom SIs per application. Alternatively, a runtime library can provide a set of commonly used SIs and offer means to request configuration of a subset for each application. If multiple applications are running in parallel, an operating system or runtime environment has to reconfigure the ALP on context switches to make sure that all SIs required by the scheduled application to be available. If the system cannot guarantee availability of all SIs used in the application, a lazy loading mechanism may be implemented. When executing an SI which is not configured in the ALP, the configuration can be triggered automatically and stall the processor until completion. Otherwise, a service interrupt can be emitted to allow the software to decide whether to reconfigure the ALP or fall back to a software implementation of the SI.

Chapter 5 takes a closer look at hybrid processor-based accelerator architectures. An example architecture is introduced and described in detail. Further, two application scenarios are described and the advantages of such an architecture are analyzed in depth.

4.4 Monolithic Hybrid Accelerators

Workloads that have few control-flow decisions and require maximum throughput benefit most from a dataflow-driven architecture as the analysis in Section 4.2.1 showed. Dataflow architectures often offer little or no software programmability, but excel in data throughput compared to processor-based systems. Still, they are often paired with a less powerful GPP taking over management and control tasks, however the majority of data is processed in a monolithic accelerator. The management pro-

cessor does not need a high performance interface to the dataflow architecture and can even be physically separated for eased maintenance or power saving.

One of the reasons for higher throughput in dataflow architectures is avoiding the von Neumann bottleneck. A monolithic accelerator can feature special high-bandwidth memory interfaces optimized for the workload, whereas processor-based systems commonly need to move all data over a central bus system. We can further implement multiple memory interfaces used in parallel, either to use independent ports for data input and output or for increased memory bandwidth in one direction.

While a management processor can perform control-flow operations, the algorithms implemented in monolithic accelerators have little internal state that affects their behavior. Many of these algorithms allow massive parallel computation, since similar to SIMD processing, the same operation can be applied to multiple data words and therefore spatial parallelization is possible. There are almost unlimited possibilities for the arrangement and type of processing elements, which vary in suitability depending on the application. In the following, we choose a systolic array for better illustration, which is well suited for a variety of parallel algorithms[1]. Similar to a processor, systolic arrays need control logic to select the correct operation of each Processing Element (PE), synchronize data loading and storing, and allow parameterization of the operation carried out. It is, however, not controlled by a stream of instructions as in a GPP, but implements algorithm-specific behavior.

Dataflow architectures offer very little flexibility at run time when implemented as ASIC. Compared to a GPP, they use little to no software to control their operation and are not updatable for inherently different workloads. This gives rise to the idea for the hybrid monolithic accelerator concept: the architecture is split into FLP and ALP, forming a partially reconfigurable yet high-performance architecture. Components which are critical to high performance and throughput of the system are implemented as hard logic in the FLP. The designer identifies components which have significant impact on the behavior of the system and implements them as reconfigurable hardware within the ALP. If properly designed, the resulting hybrid system can be reconfigured to carry out varying different operations and maintain high performance at the same time. Refer to Figure 4.1 at the beginning of this chapter for a simplified example of

[1] The exact classification of systolic arrays under Flynn's taxonomy is debatable. It can be described as a MISD architecture, since single words of data travel from one PE to the next and can therefore be processed by multiple instructions. However, the input data is a vector of multiple words loaded in parallel, which speaks in favor of classification as a MIMD system.

such a hybrid architecture. For example, the processing cluster can be a systolic array and the control unit is responsible for parameterization and orchestration of the PEs.

The identification of components to implement in the FLP is a critical design task. If too many components are implemented as reconfigurable hardware, performance may be limited unnecessarily. In contrast, flexibility is unnecessarily restricted if the right components are not mapped in the ALP. A thorough analysis of the accelerator design is necessary to get the mapping of FLP and ALP right. As this strongly depends on the design and the overall workload class, no generic approach can be given. Consider a systolic array as an example, similar to the design shown in Figure 2.6. In addition to the PEs and their interconnect, supporting circuitry like control logic and memory interfaces are necessary. Depending on their role in the system, they are implemented differently:

Processing Elements The PEs are performing the actual operation on the input data. The types of supported operations can therefore severely limit which workload can be processed on the array. If reduced performance of the PEs is acceptable, they should be considered for implementation in the ALP.

Memory Interface Memory bandwidth is crucial for most dataflow architectures. The memory interface needs to be implemented as an FLP in most cases.

Control Logic Units used for control signal generation are very significant for the behavior of the system. They are often suitable for spatial parallel implementation to achieve sufficient throughput when implemented as an ALP. Otherwise, slow generation of control signals in FPGA can slow down the system.

PE Interconnect Similar to the memory interface, efficient data forwarding between individual PEs is important to maintain high bandwidth and indicates an implementation as FLP.

In Chapter 6, an example architecture of a hybrid monolithic accelerator is described in detail. As a case study, a Deep Learning Accelerator (DLA) is designed and evaluated for processing CNNs. This accelerator is used as a building block to create a scalable, Network-on-Chip (NoC)-based computing platform featuring reconfigurable components.

4.4.1 Processing System Interfaces

Monolithic accelerators can be integrated into a processing system in several ways. Same-silicon integration is feasible when designing a SoC targeting deeply embedded systems. When building systems on a platform level, modularity and compatibility with industry standards can be a requirement. For example, a monolithic accelerator can be housed entirely on its own circuit board, which has its own memory and power supply. This way, it can be integrated in Commercial-Off-the-Shelf (COTS) server systems in the High-Performance Computing (HPC) sector.

A significant advantage of monolithic accelerators is the availability as building blocks for a system architect. Compared to a closely-coupled processor-based accelerator which requires extensive integration of processor and accelerator, only few standardized interfaces to the rest of the system are required. Common interfaces are bus interfaces like Advanced eXtensible Interface (AXI) to exchange data with a GPP and additional memory interfaces for high-bandwidth access [88]. AXI is also used in the Xilinx MPSoC series to connect their coarse-grained FLP and ALP regions (called PS and PL by Xilinx), see Section 6.2 for an example design [Les+24]. System integrators can buy accelerator IP cores and integrate them into their SoC design. Recent developments also suggest the availability of accelerator IP as individual chiplets that can be integrated on package level using Universal Chiplet Interconnect Express (UCIe) [89].

If compatibility to standard PC and server systems is the main focus and an extension card design is targeted, PCI Express (PCIe) is the standard interface. It allows memory-mapped access to registers and memory on the accelerator board from the host system, and interrupts can be issued for host signaling. PCIe may also be a sensible interface for a single-board design, in case both a SoC or host processor and the accelerator are discrete chips.

Chapter 5

Adaptive Heterogeneous Multicore Processors for Control-Flow Workloads

Two different approaches towards adaptive run-time reconfigurable accelerators have been superficially described in Chapter 4. One is closely coupled processor-based accelerators, which extend a fixed processor core with FPGA fabric and are particularly suited to control-flow-oriented applications. Building on the overview of approaches from the scientific community in Section 3.1, we will take a closer look here at a specific implementation of a reconfigurable processor. This design is used to demonstrate possible applications of such architectures in embedded systems, alongside presenting different optimization strategies for improved performance and flexibility.

5.1 The *i*-Core Architecture

Reconfigurable processors leverage FPGA-based hardware blocks to adapt to different workloads during run time. In this way the advantages of processors and hardware accelerators are combined: while the General Purpose Processor (GPP) allows for easy implementation and re-programming of software with a focus on control-flow, a hardware accelerator excels at processing of computational kernels. Figure 5.1 depicts the *i*-Core [90], a reconfigurable processor prototype based on a LEON3 processor core implementing the SPARC v8 instruction set [91]. The processor core is extended to support decoding additional instructions, so-called special instructions (SIs), in unused regions of the original SPARC v8 instruction set [92]. SIs allow making use of the reconfigurable framework specific supported operations, resulting in faster execution compared to a purely software-based implementation.

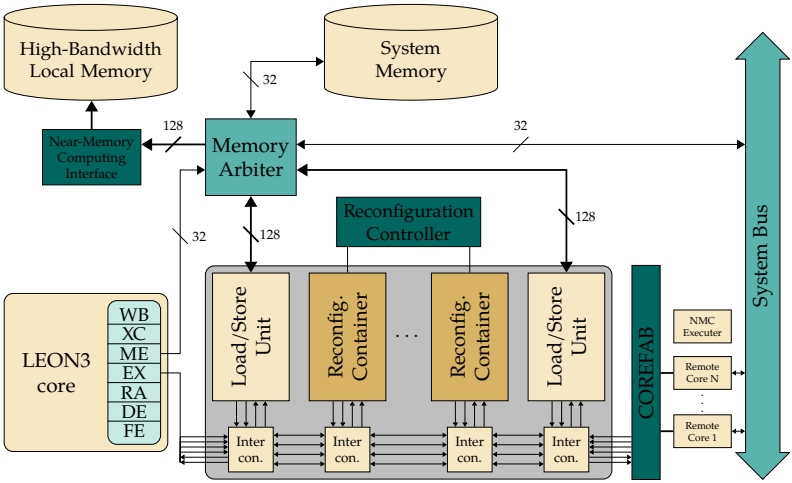


Figure 5.1: Reconfigurable processor architecture based on a LEON3 core with reconfigurable containers (RCs) and high-bandwidth interface to local memory

The accelerator fabric consists of multiple RCs which are connected through a linear data bus with a width of four 32-bit words. In addition to the RCs, two load/store units are connected at each end of the bus. They include an address generator to generate different address patterns and allow high-bandwidth access to memory with a 128-bit interface each. Because this interface is independent of the processor's load/store port, it is not subject to interference from other bus masters and can provide sustained bandwidth. The RCs can be configured at run time with application-specific accelerators and commonly contain combinatorial logic only. The Field Programmable Gate Array (FPGA) fabric implementing an RC will be configured with a bitstream included in the application. As one RC is limited in resources like Lookup Tables (LUTs), registers and memory blocks, it is necessary to split the accelerators of complex SIs into multiple RCs and to move data between them during execution. The interconnect between RCs is configured using a microprogram which we call Very Large Configuration Word (VLCW), as it controls the dataflow of each junction in the data bus.

In normal operation, the LEON3 processor executes a software binary containing SIs in computationally intensive parts of the application. When an SI is encountered, the processor pipeline stalls in the execute stage and input parameters from the register file are forwarded to the accelerator fabric. Subsequently, processing of the

VLCW referenced by the SI is started and executed step-by-step within the accelerator. These steps, which we call phases of the SI execution, usually consist of initial data loading, routing data to the respective RCs, moving data between related RCs and writing result data back to memory using the load/store units. When the VLCW microprogram exits, a return value can be written to the register file and the processor pipeline continues.

The VLCW microprogram can be created using a set of tools, starting from an abstract definition of the algorithmic kernel as Data Flow Graph (DFG). A scheduler is applied on the DFG to find a suitable mapping of abstract operations in the graph to matching RCs. This also includes components like the load/store units, which are not reconfigurable and reside in the Fixed Logic Partition (FLP). VLCWs have limited support for control-flow instructions for conditional jumps within the microprogram and counting iterations using a private register file. An enhanced C/C++ compiler toolchain is also provided. It allows using an SI within assembly code and translating it to the correct instruction encoding during compilation. The compiler also features correct register specifications for SIs, allowing to map input and output registers to C/C++ variables when using inline assembly. Resulting binaries can be executed on the reconfigurable processor directly.

An SI is implemented within one or more RCs, limited by the number of physically available RCs. The *i*-Core design has a configurable number of RCs and contains five equally-sized containers by default. An advantage of splitting an SI modularly across RCs is the possibility to load the configuration for multiple SIs at once, if the number of RCs is sufficient. Depending on the SI, different VLCWs are selected which use the required RCs only, whereas others can remain unused during execution. If the number of RCs is not sufficient to load the configuration for all required SIs, the application can decide at run time if reconfiguration is feasible or a fallback software implementation of the algorithm has to be used. This creates a trade-off between reconfiguration time, number of required accelerators and slow-down of a pure software implementation. Previous work also shows how to detect code suitable for SI-based acceleration and generate RTL code for appropriate accelerators automatically, reducing hardware design efforts [47].

The *i*-Core is a powerful platform to create and explore run-time adaptive embedded systems. In the following chapters, three different use cases for adaptive processors will be shown using the i-Core. In Section 5.2, we investigate the advantages of placing a reconfigurable processor close to memory. With its dedicated, high-bandwidth

memory interface, it can perform near-memory computing with high performance while retaining the performance and flexibility of a General Purpose Processor. In Section 5.3, we explore how approximate computing improves acceleration on the i-Core processor. In addition to increased performance and reduced energy consumption, approximate computing also reduces the resource usage of RCs and enables increased flexibility in scheduling and deployment of algorithms. Finally, we focus on security and safety of processor-based embedded systems in Section 5.4. Using a multicore processor system, possibilities for isolation by means of a hypervisor are shown, whereby the overhead at run time is to be minimized.

5.2 Adaptive Near-Memory Computing using Reconfigurable Processors

Based on the original publication *Transparent Near-Memory Computing with a Reconfigurable Processor* at ARC 2021. [LKB21]

In today's data-centric applications (e.g. big data, AI, media), the focus has shifted from pure high-performance processing to efficient data processing. These algorithms not only put a strain on the computing units, but above all on the memory connection. Within the Roofline Model, they often reside in the memory-bound area due to their low Operational Intensity (Section 4.2.1). Key to performance improvement for such algorithms is increasing the memory bandwidth of the computing units. Technology advancements in Synchronous Dynamic RAM (SDRAM) technology allow for increased bandwidth in main memories, and novel memory concepts such as High-Bandwidth Memory (HBM) and Hybrid Memory Cubes (HMCs) can be used in specialized hardware. Although these improvements have increased processing power and bandwidth, many workloads still saturate the available memory bandwidth. This especially becomes important in large-scale manycore systems, where memory transfers span longer distances and multiple levels of hierarchy.

Analogous to the findings from Section 4.2.2, most memory bandwidth is commonly used within the algorithmic kernels. The concept of Near-Memory Computing (NMC) is to move the processing of bandwidth-intensive kernels closer to memory, increasing available bandwidth. *Closer* to memory can have different meanings: next to a memory controller, between cache and upstream memory interfaces, or even on a memory tile in a Network-on-Chip (NoC)-based system. In every case, processing elements can profit from increased memory bandwidth compared to other processors in the system and potentially reduced interference from other bus members. Near-Memory Computing (NMC) must always go in conjunction with maximizing the total memory bandwidth of the system. If that is not the case, the overall memory bandwidth shall be maximized first to allow maximum performance for all processors. It can then be assessed whether NMC must be used to accommodate for insufficient bandwidth. Naturally, there is little space available near the memory interfaces, as most of the resources are used to maximize the memory bandwidth. It is not possible to place the entire processing system (i.e. GPPs including the accelerators) closer to the interfaces, as bus systems or NoCs are required to connect the processing units to the memory.

This results in a trade-off as to which algorithm can be implemented close to memory and merits the additional area requirement.

The design space is thereby greatly increased, as the designer has to choose multiple parameters when implementing a NMC accelerator:

- Select operations to be accelerated based on a workload analysis and those that should be excluded due to resource constraints or low improvement
- Determine where to position the accelerator within a complex multi-level memory hierarchy
- Place the accelerator within the Performance, Power and Area (PPA) trade-off

The key question of this thesis can be applied to the concept of NMC: Implementing a specific algorithm as fixed logic close to memory may not be flexible enough for future use of the platform, while using a GPP close to memory may not provide the required performance improvements. Therefore, the feasibility of utilizing an adaptive processor needs to be assessed. Adaptive processors combine the best of both worlds, reaching higher performance than software-based solutions and allowing reconfiguration to adapt to new use cases. However, the performance of an ASIC cannot be matched by reconfigurable hardware and FPGA designs are generally more difficult to build than a software solution. This section investigates the suitability of an adaptive processor for NMC, showcasing accelerated on-the-fly data encryption using Advanced Encryption Standard (AES).

5.2.1 Related Work

NMC is a common approach to improve performance of data-centric workloads. As many different approaches have already been published, this section classifies some of these approaches and compares them to our design. We are focusing on NMC and In-Memory Computing (IMC) approaches only, related work regarding adaptive processors is discussed in Section 3.1. Compared to placing computational units at the periphery of the memory (NMC), IMC leverages the internal structure of the memory itself to perform calculations.

An important property of NMC implementations is the type of memory the system is working on. Depending on the workload it can be feasible to place accelerators close to bulk storage, as shown with *Summarizer* [93]. The authors use the existing NVMe command interface to offload work from the host to a processor next to the

Solid State Drive (SSD) controller (Figure 5.2). Data stored in non-volatile flash memory on the SSD is not arranged trivially. Thus, the flash translation layer (FTL) of the SSD controller is used to fetch data into Dynamic Random Access Memory (DRAM), which is then processed by the user-defined function. After processing, data is stored back to flash memory using the FTL. The authors test their design with data aggregation and sorting functions formulated in Structured Query Language (SQL), and can show performance improvements of up to 20 % for some queries.

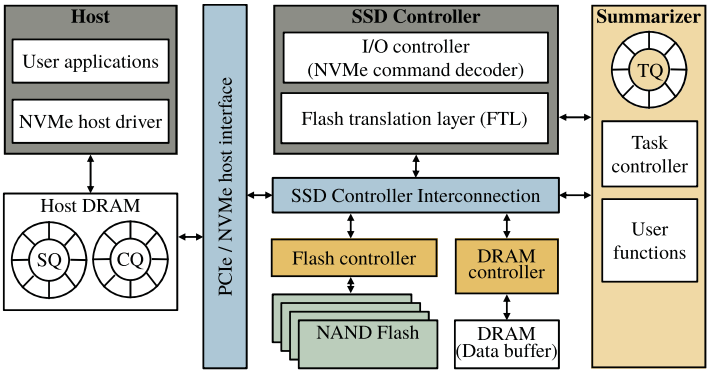


Figure 5.2: NMC accelerator within an SSD shown in *Summarizer*. The figure shows a common GPP system with an SSD attached via PCIe. The NMC accelerator is attached to the SSD controller and can access both flash and memory interfaces directly [93].

Types of possible operations depend strongly on the physical memory architecture, with novel non-volatile memory types such as Magnetoresistive RAM (MRAM), Phase-change Memory (PCM), and Resistive RAM (RRAM) being the focus of the scientific community [94]. Kang et al. present an approach to leverage an STT-MRAM chip to perform bitwise logic operations during readout [95]. Although only simple bitwise operations are possible, the memory cells do not need any additional circuitry, which results in minimal overhead for this method. In contrast, IMC approaches like *iPIM*, which features an in-memory image processing accelerator, are very flexible and still profit from very high memory bandwidth [96]. Flexibility is achieved by implementing a processor between memory banks (Figure 5.3), implementing the concept of Single Instruction Multiple Bank (SIMB) instructions, which apply operations to multiple memory banks at once.

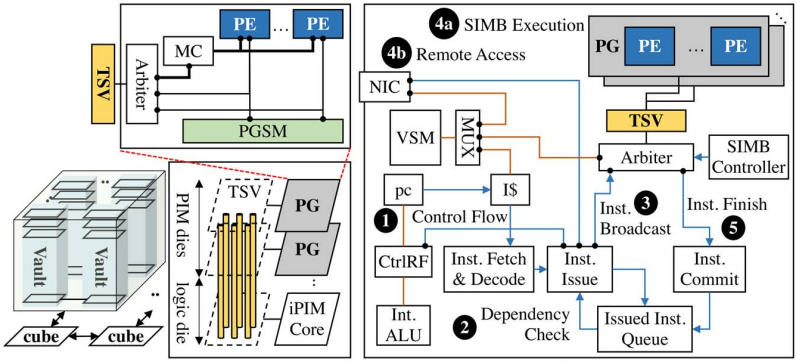


Figure 5.3: Processor-based IMC implementation by Gu et al. On the left, a 3D-stacked memory architecture is shown, consisting of several independent memory banks (*vaults*). The processor pipeline is implemented in a control block (right), which controls multiple processing elements of one vault [96].

Apart from the memory target, the fundamental design of the accelerator itself can be distinguished. Some of the approaches mentioned earlier, such as *Summarizer* and *iPIM*, use GPPs which are placed close to or within memory to perform operations. The benefits are high flexibility as software can easily be exchanged and low hardware costs due to the broad availability of processor Intellectual Property (IP) cores. However, some approaches choose FPGA or ASIC based accelerators for even higher processing power. Corda et al. demonstrated large performance benefits when using FPGAs with direct memory access for Fast Fourier Transform (FFT), outperforming both GPP and Graphics Processing Unit (GPU) based approaches [97]. To allow comparing different architectures, the authors use DDR4 and HBM directly attached to the processing element, i.e. GPP host memory and on-board memory for GPU and FPGA.

It can be concluded that approaches to NMC vary greatly with regard to memory technology and accelerator design. They differ from IMC, where memory technologies are augmented with processing capabilities for even higher bandwidth and implicit parallelism for data stored in different cells. Fully custom NMC accelerator designs implemented as Application-Specific Integrated Circuit (ASIC) provide peak performance, whereas processor-based approaches are programmable and still profit from increased memory bandwidth. However, none of the aforementioned works

combine the advantages of both approaches, which makes the investigation of our adaptive processor for NMC particularly interesting.

5.2.2 Design of a Reconfigurable Processor for Near-Memory Computing

This section will first introduce the architecture of our Near-Memory Accelerator (NMA), based on the reconfigurable *i*-Core processor. Afterward, our region-based approach for transparent operation on the memory bus is discussed, and its advantages are highlighted.

Near-Memory Accelerator Concept

The core component of the NMA concept is a reconfigurable processor which is placed close to memory. We focus on a simple Multiprocessor System-on-Chip (MPSoC) platform first and discuss the application of this concept in other scenarios, such as NoC-based manycore platforms, in Section 5.2.2. Figure 5.4 shows a high-level view of an example system containing multiple general purpose CPUs, peripherals and memory. Besides the interface to the main bus, the memory has a high-bandwidth connection to the NMA. The NMA features a standard processor core, similar to the other GPPs in the system, and a reconfigurable accelerator framework. Processor and FPGA framework are connected to allow SIs to be executed in the reconfigurable partition.

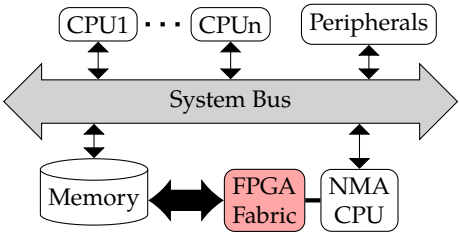


Figure 5.4: Near-Memory Accelerator consisting of processor and FPGA framework in a multi-processor system

By incorporating a standard CPU core, the NMA can execute code directly, without requiring RTL design. A key advantage is the low barrier to porting an existing algorithm to the accelerator: software can be easily adapted to run on the NMA. Even

without using the reconfigurable portion of the accelerator, software can benefit from improved memory performance. High-bandwidth load and store instructions can be used, which also do not generate any load on the main system bus. The generic concept does not require a specific processor architecture, allowing the designer to choose one that fits the design constraints. However, the processor must provide an interface to the reconfigurable framework that allows custom instructions to call the hardware accelerator.

The reconfigurable framework is an FPGA-based hardware accelerator. It facilitates the transfer of computationally intensive tasks from the near-memory processor, thereby enhancing the performance of the NMA. The implementation of an algorithm in hardware description language (HDL) is often a very time-consuming task. The proposed approach exploits the interaction between the processor and the hardware accelerator, leveraging the implementation of only the costly components of an algorithm in hardware to achieve substantial performance enhancements. Other work, such as initialization and control-flow tasks, can still be handled in software. In comparison to memory-mapped accelerator interfaces, the close coupling between the processor and the reconfigurable framework facilitates the engagement of the accelerator through the utilization of a suitable special instruction.

Thanks to its reconfiguration features, the accelerator framework can be used for a changing set of tasks. The required accelerator configuration can be loaded when required, falling back to a software implementation if the accelerator is being shared and other tasks have higher priority. Additionally, the NMA can adapt to unprecedented use cases in the future, still allowing acceleration of updated software where non-reconfigurable approaches need to fall back to software-only processing.

Region-based Near-Memory Computing

The reconfigurable framework is primarily controlled by software, as it requires a special instruction to operate. All clients using the NMA are thus required to synchronize with the software running on the accelerator. Typical schemes for sending a command to the NMA are remote procedure calls (RPCs), requiring a communication channel to be set up beforehand. To simplify and reduce communication effort between client and accelerator, we propose to make the accelerator transparent to memory access: The memory on which the accelerator operates is fully accessible via the bus, and thus accessible to all clients. A client can define regions within that memory, consisting of a starting address, size of the region and an operation. By monitoring the memory

bus, the accelerator can detect access to such an area and initiate the associated operation. This trigger can determine whether sufficient data is available, contingent on the selected operation, and initiate that operation automatically. Subsequent to this initialization, the user is only required to stream input data to the designated memory address, and subsequently, the output data can be retrieved after the completion of the processing operation. This approach significantly reduces the effort required to utilize the NMA, as it only necessitates the initial configuration of an appropriate region.

As in our concept the NMA contains both a CPU core and an accelerator framework, this yields two possibilities for the NMA to be triggered:

Interrupt-based trigger The processor core receives an interrupt after a region received enough data. The NMA starts processing in software and can use the hardware accelerator if suitable. We call this the *software-assisted mode*, as algorithms can be partly accelerated by hardware and run the remaining operations in software. The software-assisted mode can make use of reconfiguration by loading different accelerated instructions during one operation.

Direct hardware trigger The hardware accelerator is directly started without software intervention. As no software overhead occurs, this approach allows very low latencies. The NMA processor can execute other code in parallel, as the hardware accelerator does not interfere with the processor resources. However, the respective operation needs to be fully implemented in hardware. This may not be possible for complex algorithms due to limited hardware resources and the lack of reconfiguration.

The region-based NMC approach can be extended to allow overlapping regions. Thus, multiple algorithms can be applied consecutively to the same data without having to copy the data in between. A simple example of overlapping regions is shown in Figure 5.5, where two regions are filled with data to be sorted. First, the *quicksort* algorithm is applied to each region separately. Second, the *mergesort* algorithm can be used to merge both regions and obtain a fully sorted array.

Software interface

The region-based concept simplifies the steps necessary in software to use the NMA. Traditional approaches use RPCs or memory-mapped control registers to configure the accelerator and fine-grained execution control is handled by driver software. In contrast, our approach requires a one-time configuration, after which data transfer to

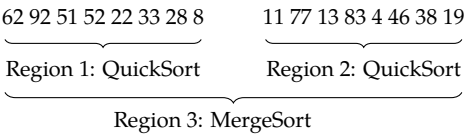


Figure 5.5: Combining sorting algorithms using overlapping regions

the configured region is sufficient to initiate operation. Figure 5.6 shows the interaction between client, NMA and memory.

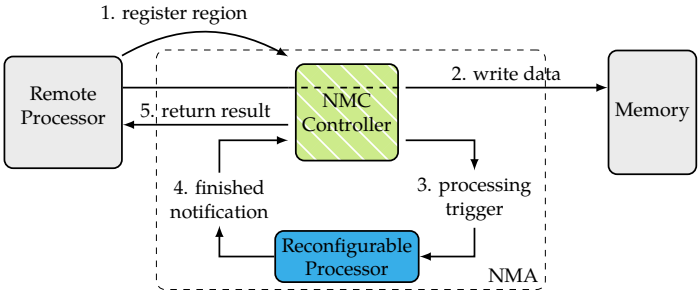


Figure 5.6: Configuring regions and using the NMA

The steps required to use the accelerator can be described as follows:

1. The client is required to register a region for use, including its size and an associated operation. It is essential to avoid region conflicts with other users, although the memory management of the operating system can be used to ensure unique buffer allocation.
2. Payload data is now written directly to memory, a call to `memcpy` or using DMA is sufficient.
3. The NMA observes memory writes and triggers the reconfigurable processor as soon as its processing condition is met. Either the reconfigurable framework is started directly or the software-assisted mode is used, depending on the selected operation.
4. The reconfigurable processor notifies the controller when it has finished processing the region.

5. A result is returned to the client, if applicable, and output data can be processed further.

Interaction with software on the client takes place in steps 1, 2 and 5 only. Steps 2 and 5 can be repeated arbitrarily to process further data. The client thus does not need to send any control commands after setting up the region, reducing communication overhead with the NMA to a minimum.

Synchronization mechanisms

As clients do not control the accelerator directly, it is required to establish means for synchronization to decide whether a region is in use and when an operation is finished. Additionally, the memory regions need to be protected from modification while the accelerator is working on them. These requirements can be realized by extending the behavior on the bus system. As a bus slave, the accelerator can delay or deny requests if they are currently not appropriate.

The behavior of the accelerator differs for the following three states of a region:

Region idle Writing data is allowed, waiting for enough data to start processing. Reading can be denied if necessary, e.g. secret data waiting for encryption.

Region busy Both read and write requests stall until processing is done, or a bus error is returned if non-blocking operations are preferred.

Region done Reading is allowed. Return to idle automatically if all output data is read or manually by register write to the controller.

To allow continuous writing and reading of data, multiple regions can be allocated and used in a round-robin fashion—similar to double or triple buffering methods. Our enhanced bus interface is not only useful for synchronization purposes, but also serves as a security feature. When the NMA is used to handle confidential data, unintended read accesses are denied until the region enters the *done* state. This ensures that no confidential data is being read until processing, like encryption, is fully carried out. Note that authenticity is not ensured, as data could be overwritten by an attacker while in *idle* state.

Large-scale platforms

In recent years there has been a trend towards ever larger multicore systems with heterogeneous memory architectures. To improve manageability of such platforms, such manycore systems are usually divided into tiles. Therefore, the notion of local and

remote resources can be established, depending on where a program is currently executed. While accessing a local accelerator resource on the same tile is fast and simple, accessing remote accelerators requires support by hardware and software. Software support in the operating system or runtime libraries uses on-chip communication mechanisms to communicate with remote resources and exchange data.

In such a scenario, our transparent approach brings several benefits: While operating system features for inter-process communication can still be used for allocation and arbitration of accelerator access, moving data to and from the accelerator does not require intervention by the runtime system. Especially in Partitioned Global Address Space (PGAS) systems, data transfer solely involves the remote user and its DMA controller. This saves additional API calls for submitting a buffer to be processed, which can be costly depending on the communication mechanism used.

However, there is a trade-off between copying data to an accelerated processor and processing without acceleration. Depending on the amount of data to process and the performance gain by hardware acceleration, it can be faster to process data without acceleration than moving it to the NMA memory prior to processing. This effect can be reduced by avoiding local data storage and using the storage backed by the NMA by default.

5.2.3 Hardware Implementation

We implemented our NMA concept using the *i*-Core processor as a framework [90]. The *i*-Core consists of a LEON3 core and a reconfigurable fabric, which is connected to the internal Scratchpad Memory (SPM). These two SPM interfaces are independent from the processor's memory interface and offer 4 × increased bandwidth. To implement our NMA concept, these interfaces are repurposed to interface the associated DRAM directly. Operations running in the reconfigurable framework will thus profit from the increased bandwidth and no interference from other masters on the bus. Additionally, the *i*-Core design includes supporting components like the bitstream loader, allowing for run-time reconfiguration of the framework. Nevertheless, the implementation of the proposed concept requires to extend the platform by modifying the processor itself and providing a new module called Near-Memory Computing Controller (NMCC) as shown in Figure 5.7. The controller is used by clients to register regions as described in Section 5.2.2, associate them with operations and check the state of the NMA.

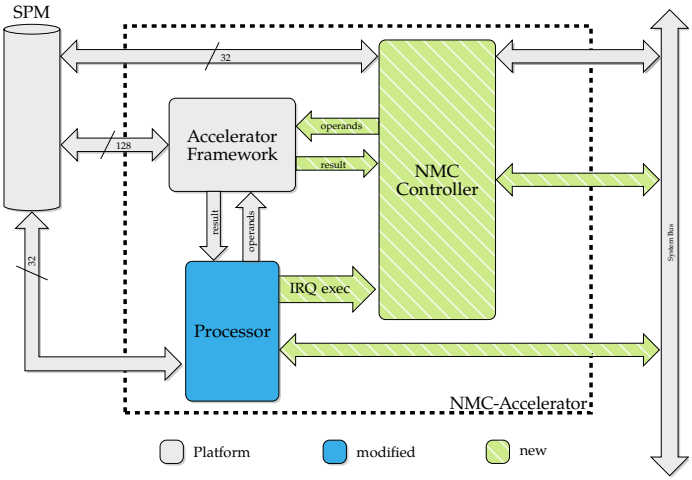


Figure 5.7: Block diagram of added and changed components of the NMA implementation

NMC Processor Extensions

Due to its ability to run a broad range of tasks the internal processor is a key element to implement a versatile NMA architecture along with the reconfigurable framework. As a bus master, it can access not only the SPM attached to the reconfigurable framework but also other system components to load data if necessary. Therefore, fetching additional data required for the task can be offloaded from the client to the NMA, assisted by software running on the internal processor if necessary.

To notify the processor about pending tasks in the software-assisted mode, it is directly connected to the NMCC. As the LEON3 processor allows customizable hardware traps, this feature is used to assert an interrupt each time the processor needs to start working on a software-assisted region. Moreover, this approach allows the processor to execute unrelated tasks in parallel, which get interrupted automatically whenever the accelerator itself requires action. In our reference implementation, the operating system treats the reconfigurable processor like a normal processor, as it is able to execute generic workload in parallel.

In addition to the interrupt interface, the processor requires information about the selected operation and the address range (region) to operate on. These values are

directly wired from the NMCC to the execution stage of the processor, where they are presented as operands to the interrupt handler. The software handler uses this information to identify the target region and initiate the associated operation. Upon completion of the NMC task, the processor stores the result in the NMCC and returns to its previous state.

Near-Memory Computing Controller

The main component of our implementation is the NMCC. To realize the synchronization mechanisms (Section 5.2.2), it is placed between system bus and SPM to be able to monitor and control read and write transactions on the SPM. During execution, the NMCC can deny or delay write access to the SPM to prevent the NMC result from being affected by inconsistent values in memory. Moreover, in case of encryption algorithms it also allows ensuring that clients are not able to read the plaintext. When delaying a memory access, bus splits are used to avoid locking up the system bus. Figure 5.8 shows the architecture of the NMCC implementation.

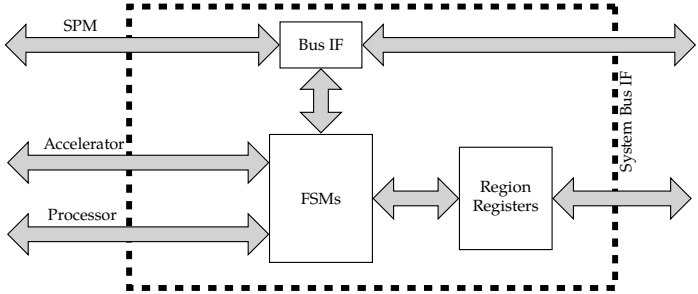


Figure 5.8: Schematic of the Near-Memory Computing Controller (NMCC)

When a write transaction to the SPM matches the processing condition of an enabled region, the corresponding action is triggered. Thereby our implementation offers two different possibilities to execute the desired task. Either an interrupt can be used in case of the software-assisted mode or the reconfigurable framework can be accessed directly using the *COREFAB* interface [98]. Originally designed to allow remote cores to execute SIs on the hardware accelerator, this interface allows bypassing the processor and executing a task directly within the accelerator framework. Input parameters required by the accelerator framework such as operands and the SI to run have to be configured as part of the region configuration prior to execution. However, the possibility of parallel execution of SIs on the processor is retained, as

the implementation of the extended framework has an arbiter that switches between local and NMC instructions. It also provides additional logic to merge accelerator operations if both local and remote processors request to run an SI at the same time but not on all available RCs in the framework.

Since overlapping regions are allowed, the NMCC has to decide which NMC task is executed first in case of multiple regions raising a request at the same time. In addition, requests can be separated into a queue for each operating mode. Hence, in case of one region requesting a software-assisted task and another one asking for a task being directly executed in hardware, both requests can be initiated at the same time. We therefore implemented two independent Finite-State Machines (FSMs) within the NMC Controller as shown in Figure 5.9 to ensure that only a single task is running on each processing entity.

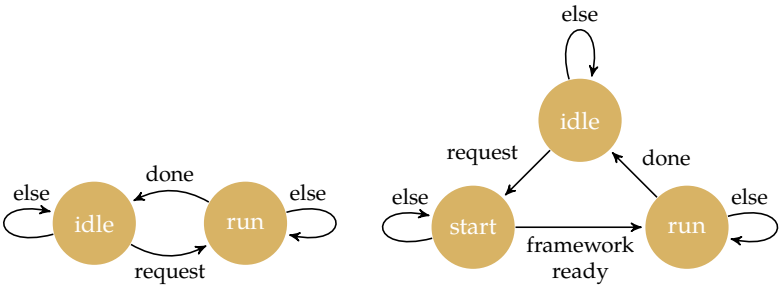


Figure 5.9: FSMs of software-assisted (left) and direct hardware tasks (right)

Region Configuration Interface

We implemented externally accessible registers of 32 bit within the NMC Controller to allow clients to configure regions. Table 5.1 shows the register set for a single region, which is repeated with a fixed offset for each region configured at design time.

The Region Range Register contains both start address and size of the memory region in the SPM. As operations are currently limited to 2^{16} words, it is sufficient to store both address and size in a single register to minimize overhead.

Furthermore, the Region Control Register allows the user to enable a region and to control whether the region should not be readable during execution. It also includes an operation field, which is either forwarded to the processor to identify the desired

Offset	Register	Usage
0x00	Region Control Register	Operation selection, bus protection, SW/HW mode
0x04	Region Range Register	Start address and size
0x08	Region Status Register	Busy and finished bits
0x0C	Region Result Register	Result of the last operation
0x10	Region Operand Registers	4×32 bit operand registers (HW mode only)

Table 5.1: Register set for a single region

NMC operation or to the reconfigurable framework to select the SI to be executed, depending on the selected operation mode.

The Region Status Register allows clients to check the execution state if interrupts or other OS interfaces for notification are unavailable. Its busy bit is set as soon as the execution condition is met and cleared when the task has been finished. After the busy bit is cleared, clients can load the processed data from the SPM. Additionally, it is possible to retrieve the return value from the Region Result Register if necessary.

5.2.4 Evaluation

We implemented a prototype of the NMA as a dual-core system, where one core is the augmented near-memory processor and the second core is a standard LEON3 core acting as the remote user of the accelerator framework. The resulting hardware design has been tested both in simulation and on a Xilinx VC707 prototyping board with a Xilinx Virtex 7 FPGA. Cycle counts have been determined using a clock cycle counter in the debug unit. During all tests, there was no additional traffic on the system bus by other applications. Such traffic would reduce the performance of software implementations further, with low impact on the NMA performance due to its isolated memory interface.

Quicksort performance

We compare the performance of the Quicksort algorithm implemented purely in software to an accelerated version using the reconfigurable accelerator. All three tests run in software on the NMA itself without making use of the region-based processing

yet. To be able to distinguish the performance gain from the high bandwidth memory interface of the NMA and its actual computational performance gain, the intermediate result of a test running in software, but using the high bandwidth memory interface, is also shown in Figure 5.10. It is important to note that the high bandwidth interface is independent of the main AHB bus, thus the processor is still using available AHB bandwidth to load instructions.

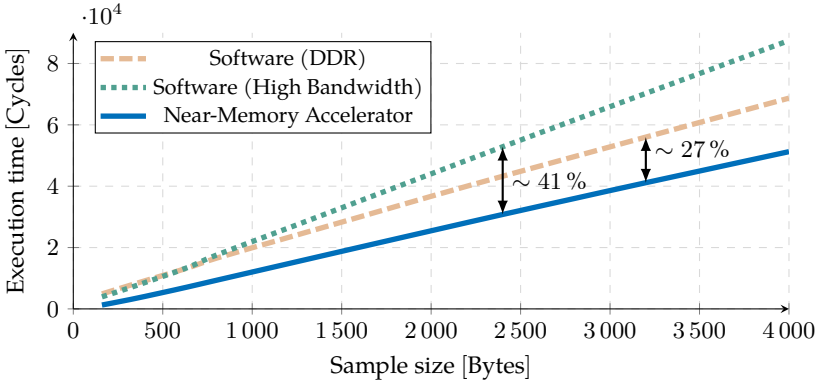


Figure 5.10: Quicksort benchmark

The results show that Quicksort performance is primarily limited by memory bandwidth, which can be improved by using a high-bandwidth memory interface. The findings reveal that reducing the load on the AHB bus enhances both data and instruction loading performance, thereby contributing to the observed enhancement in performance. Furthermore, the results demonstrate an additional speed enhancement when employing the accelerator framework alongside the high-bandwidth memory interface.

Video encoding acceleration

As part of an H.264 video encoder implementation on our platform, the sum of absolute differences (SAD) algorithm has been implemented within the NMA architecture [99]. It is used to calculate the similarity of two 16 × 16 image sections:

$$SAD(x, y, r, s) = \sum_{i=0}^{15} \sum_{j=0}^{15} |B_2(x + i, y + j) - B_1((x + r) + i, (y + s) + j)|$$

To support our approach, SAD is used as a benchmark to compare software-based execution to the NMA: A remote processor configures a region and streams pixel data to that region. As soon as enough data is available for a SAD 16 × 16 operation, the NMCC starts processing. In case of the accelerated implementation, the reconfigurable accelerator is triggered directly, not involving the processor at all. The software reference runs on the NMA processor, but does not use any accelerated instructions and is therefore limited to 32 bit bus width.

Interface	Software reference	Near-Memory Accelerator	
		4 Pixels/Iteration	8 Pixels/Iteration
32 bit	1 635 cycles	463 cycles	339 cycles
128 bit	N/A	223 cycles	126 cycles

Table 5.2: Run time comparison of SAD 16 × 16 on the NMA

Table 5.2 shows a run time improvement of 72 % when launching the NMA through the remote processor interface. The impact of the memory interface is also quantified, taking up only 48 % and 37 % of cycles respectively when using the high bandwidth memory interface. Additionally, reconfiguration allows the SAD accelerator to be implemented twice, as enough resources are available in our sample design. Thus, it can process eight pixels per iteration, almost doubling throughput of the SAD algorithm. Note that when using the 32-bit memory interface, execution is memory-bound when running two instances at once.

Accelerated Encryption using AES

To demonstrate both the flexibility and performance of the software-assisted approach, an AES encryption benchmark is used. The implementation is based on the *MiBench* suite [100]. As the AES algorithm involves a diverse control-flow as well as preliminary operations like key derivation, the software-assisted operation mode is used. Consequently, a finished write to the input data region triggers an interrupt and preparations are done in software. The operations for a single encryption round

are implemented in hardware, processing 128 bit of data per cycle. The hardware accelerator is therefore called for each round (10, 12 or 14, depending on key length). As a different implementation is necessary in the last round, we fall back to software processing in that case. This highlights a benefit of the processor-based approach: instead of having to implement the full AES algorithm in hardware, less demanding operations like preparation or finalization can still be handled in software. Although only parts of the algorithm are accelerated, we still see significant performance gains as shown in Figure 5.11. An intermediate result is also included to distinguish memory bandwidth and hardware acceleration gains. In the intermediate case, the AES implementation does not use hardware acceleration but still profits from increased memory performance of the NMA processor.

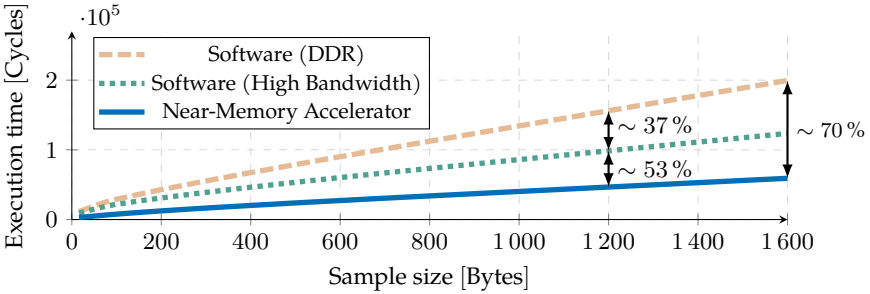


Figure 5.11: AES performance comparison

It can be seen that the software performance increases by 37 % when using the high bandwidth memory interface only. Using the LUT-based hardware acceleration, execution time improves by approx. 53 % compared to the software implementation. While this is a valuable improvement, it cannot meet the performance of full hardware-based AES implementations [101]. However, the hardware resources of the accelerator can be reconfigured and used for different workloads, trading performance against flexibility.

Resource Usage

The resource usage of our implementation can be divided into the processor core, the accelerator framework and the NMCC. To allow comparing our approach to other implementations we also provide numbers for a common LEON3 processor as well

as a modified LEON3 core that can access a remote framework within its execution stage. Table 5.3 shows Flipflop (FF), LUT and Block RAM (BRAM) resources for the individual components.

Component	FF	LUT	BRAM
LEON3	4 785	12 793	22
Reconf. processor	5 505	13 245	24
framework	7 922	16 326	59
instruction merging	1 032	9 661	0
Remote reconf. processor	5 471	13 741	24
NMCC (4 regions)	1 143	907	0

Table 5.3: FPGA resources by component

The reconfigurable framework holds the biggest share of resources. This is primarily caused by the reconfigurable containers, holding a fixed amount of FPGA resources required to fit different accelerators. The amount of resources for these regions represents a trade-off between low resource usage and flexibility to fit larger accelerators for applications, whose requirements may not be known at design time.

Figure 5.12 compares the resource requirements of the NMCC holding up to 32 regions to a configuration with N CPUs allowing to remotely access the reconfigurable accelerator using SIs. The difference between the resources allocated for a remote core with SI support and a common LEON3 processor yields the overhead necessary for remote execution. Our region-based approach uses standard LEON3 cores only, but requires specifying the amount of available regions at design time.

In terms of scalability, the resources used by the NMCC grow linearly with number of regions. Comparing to the original remote SI interface, the proposed concept is independent of the amount of processors in the system able to run instructions in the reconfigurable framework. Instead, all clients share the available regions and allocate regions on demand, allowing even more distant users in manycore environments (Section 5.2.2). Our evaluation shows a trade-off between processors with remote SI support and the number of regions for the region-based approach, e.g. a six-core system exceeds both LUT and FF requirements of a design supporting 16 regions. Especially in manycore architectures where there is no need for running multiple SIs the region-based NMC is a very good fit. However, the loose coupling

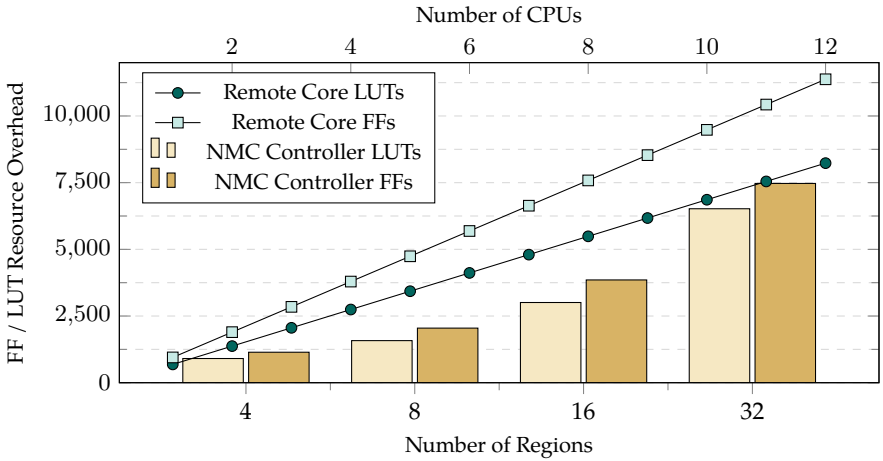


Figure 5.12: Resource overhead comparison of remote SI cores and the region-based approach

between standard processors and a region-based NMA can lead to increased latency of workloads using a variety of accelerated operations, due to the overhead of configuring regions instead of calling an SI directly. Therefore, in high performance computing environments that can constantly make use of the reconfigurable fabric, the remote interface might be a better choice.

5.2.5 Conclusion and Outlook

In this section, we investigated the application of a run-time reconfigurable processor as a Near-Memory Accelerator. The concept is easily accessible to developers thanks to its general-purpose processor, so existing software implementations can be ported with little effort. The attached accelerator fabric benefits greatly from the high-bandwidth interface to the attached memory, making it very suitable for operation close to memory. Run-time reconfiguration allows the NMA to adapt to different workloads dynamically, representing a trade-off between flexibility of software and throughput of an ASIC implementation. It can be flexibly decided if the accelerator is used directly or if it is sufficient to accelerate only parts of the algorithm and execute

remaining tasks in software. We benchmarked several different example applications and showed a performance gain of up to 13 × for SAD acceleration.

A core feature of our NMA is transparent operation on memory by defining regions with associated operations. The near-memory controller can dynamically execute these operations whenever a block of data is fully written to memory and notify the client after operation has finished. Especially in PGAS architectures, configuring and using the accelerator is greatly simplified, as clients can send data directly to memory backed by the NMA, and no further IPC is necessary to start operation. For secure applications, the near-memory controller can lock access to data if required.

In the future, we will see an increasing number of memories with integrated or closely-coupled processing elements. While our reconfigurable NMA concept is a flexible base design, thorough Design Space Exploration (DSE) is required to find optimal design parameters to balance memory bandwidth, hardware overhead for different usage models, and the number of GPPs and Processing Elements (PEs). The capabilities of the VLCW Instruction Set Architecture (ISA) are limited to linear execution and simple control-flow instructions. Incorporating features such as compressed VLCWs, in conjunction with enhancing its control-flow capabilities, has the potential to reduce reliance on the GPP and optimize performance.

Additionally, the impact of region-based NMC on large manycore platforms can be investigated further. The conceptual methods described in Section 5.2.2 have only been evaluated by means of emulation on the multicore prototype. Although portable to PGAS systems, side effects and overhead of our region-based approach in conjunction with Direct Memory Access (DMA) engines needs to be verified.

5.3 Approximate Computing in Runtime Reconfigurable Processors

Based on the original publication *Approximate Accelerators: A Case Study using Runtime Reconfigurable Processors* at SOCC 2023. [LHB23]

In many application areas of embedded systems today, we see very high demands on computing power, but also on adaptability and updatability in the field. Run-time reconfigurable processors represent a viable approach to these challenges by combining the flexibility of processors and the performance of FPGAs. However, many modern algorithms, such as neural networks, are extremely computationally intensive and pose a challenge even for the most sophisticated accelerators. The semiconductor industry is striving to address this escalating demand for increased computing power through technological advancements and architectural optimizations, but is increasingly reaching the limits of what is technically possible. To deal with these unsatisfied demands, a holistic approach is sensible: Not only should the computer architecture be considered, but it is also worth taking a look at the algorithmic side to find methods for reducing complexity. For example, neural networks benefit greatly from sparsity optimizations, allowing to skip data during calculation which does not influence the result [42]. Furthermore, algorithmic optimizations have the potential to reduce hardware requirements and offer greater flexibility within the PPA design space.

Approximate computing is a computing paradigm that aims to enhance energy efficiency and/or performance. It leverages the premise that many algorithms can tolerate a certain degree of inaccuracy during evaluation and still produce viable results. For example, scenarios that rely on sensor data from the real world, such as audio and image processing, but also machine learning approaches, need to tolerate deviations in their input. Similar to noise in the input data, deviations during evaluation may also be acceptable. By deliberately accepting certain computational errors, arithmetic operations can be performed more efficiently, reducing processing time, energy consumption, and resource requirements. In this section, we investigate the impact of approximate computation methods on reconfigurable processors. In particular, it can be shown that in addition to performance improvements, approximation allows accelerating multiple algorithms in parallel by overcoming resource constraints and I/O bottlenecks. All our scenarios have been evaluated on a physical hardware prototype of the *i*-Core architecture.

5.3.1 Related Work

Traditionally, computing architectures are built to produce results as accurately as possible. However, several applications can tolerate a certain amount of errors and still behave correctly from the user's point of view. Li et al. provide an analysis for the required application-level correctness of a program, where the necessary level of quality depends on the application [102]. They cover multimedia and Machine Learning (ML) benchmarks, where it is easy to understand their resilience to inaccurate calculations, but also applications from the SPECint benchmark suite, which are usually expected to yield deterministic results [103]. The authors refer to programs that can work with inaccurate data as *soft computations*, which are very resilient against data corruption on the application level. They further explain that errors in certain operations can have more serious consequences, such as changes to the program counter. Nevertheless, that does not necessarily lead to unusable results or crash an application, and even when it does, recovery is possible in many cases. Finally, each benchmark is given a fidelity metric indicating the user satisfaction with the result produced by the application. Although Li et al. focus on personal computers, we can still apply some of their observations to embedded systems. Many computationally intensive algorithms used in embedded systems also fall into the category of soft computations, like image processing and neural networks, and we can leverage their application-level error resilience. When implemented in hardware within the reconfigurable fabric of the *i*-Core, algorithms are processed with a focus on dataflow. Approximate computing therefore has no influence on control-flow mechanisms such as a program counter and the application is less susceptible to crashes.

Allowing approximate results in hardware can result in faster processing and lower demands on logic cells, thereby it can also increase the energy efficiency. Han et al. propose approximate computing as a paradigm for energy efficient design, including the design of approximate arithmetic blocks to allow for improved efficiency through methods like voltage overscaling [104]. An apparent approach lends itself to iterative procedures, which can be terminated prematurely at the expense of accuracy. Several different basic building blocks to create approximate hardware exist in academia, most commonly addition, multiplication and division units. *EvoApprox8b* is a library of approximate adders and multipliers [105]. The circuits have been procedurally generated and analyzed regarding worst-case and mean error rate, alongside power and area estimation. Figure 5.13 shows an example for an exact 8-bit adder built

from full- and half-adders, and an approximate implementation where the three least significant bits are generated by single logic gates.

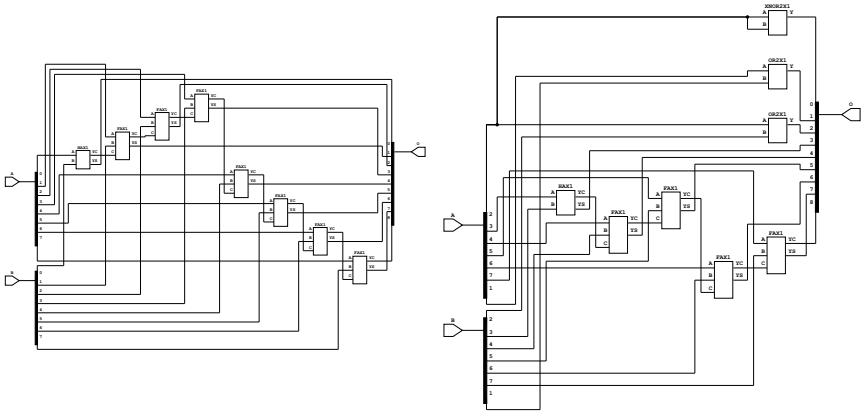


Figure 5.13: Schematics of an exact 8-bit ripple-carry adder (left) and the approximate adder *add8_136* (right) from the *EvoApprox8b* library [105]. While the exact implementation uses seven full adders, three full adders are replaced with XNOR and OR gates in the approximate design.

Systematic approaches can be used to generate approximated adders as Shafique et al. propose, as breaking up the carry chain significantly improves timing [106]. A generic approximate adder component called *GeAr* is proposed, which is highly configurable to meet different requirements in terms of accuracy, path delay and area. Every adder is composed of multiple sub-adders, each with its own carry chain, thus reducing the critical path length. Figure 5.14 shows a sample configuration of *GeAr* with three sub-adders. The result is constructed from partial results from the sub-adders and the carry-out of the most-significant sub-adder.

Approximate adders profit from optimization for the target platform, as they can leverage architectural features to be implemented more efficiently. Ahmad et al. present an efficient implementation of approximate adders for FPGAs, that is particularly optimized for LUTs of Xilinx Virtex-7 FPGAs [107]. Two different designs are presented, *LEADx* optimized on low Mean Squared Error (MSE) and area, and *APEx* optimized on area and power efficiency. n-bit approximate adders are constructed from an m-bit approximate adder and an $n - m$-bit accurate adder. A comprehensive library of

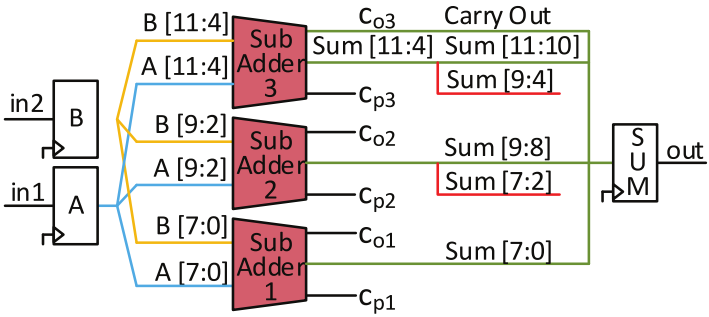


Figure 5.14: The approximate adder *GeAr* in the configuration $N, R, P, k = 12, 2, 6, 3$ [106]. It uses three sub-adders to generate the result, each sub-adder having its own carry chain.

approximate multipliers is presented by Ullah et al. [108]. In contrast to *EvoApprox8b*, the *SMApproxLib* is optimized for FPGA architectures and outperforms the former in terms of power and area. Yao et al. presented an improved approximate multiplier design with additional optimizations, which saves area while further reducing delay compared to *SMApproxLib* [109]. We opt for their approximate multiplier *LM-2* for moderately approximated multiplications, due to the good balance between the Mean Relative Error of 0.25 % and the resource requirement of 10 LUTs. For the stronger approximation, the *LM-NC* design is selected due to its unmatched low resource usage of 6 LUTs and critical path length of 4.059 ns.

It can be concluded that a multitude of approximate adders and multipliers are readily available. Given these components, the implementation of approximate hardware accelerators for our reconfigurable processor can now be addressed. Depending on the application requirements, a matching design regarding accuracy, resource requirements and performance can be selected.

5.3.2 Approximate Accelerators for Reconfigurable Processors

By applying approximate computing techniques, the performance of a hardware accelerator can be improved, and its resource requirements are reduced. This is based on the property of many algorithms to provide acceptable results even if they are not computed exactly. When looking at hardware accelerators, different approximate computing techniques can be applied [110]:

- approximation through simplification on the algorithmic level, like early termination of an iterative algorithm or omitting optimizations
- using approximate circuits like inexact adders and multipliers
- quantization or other accuracy reduction of data types, e.g. reducing the bit width of operands or LUT contents

The applicability and impact of the applied technique are highly algorithm dependent. However, all techniques are practically applicable to algorithms on reconfigurable processors and will thus be investigated in this work.

The usability of different approximation techniques is also dependent on the hardware architecture. In the context of our reconfigurable processor design, the size of RCs and the data width of the interconnect introduce additional parameters, which must be taken into consideration for a comprehensive assessment. The length of the VLCW microprogram dictates the length the SI's execution. If the number of phases in the VLCW can be reduced, e.g. through decreasing the required accuracy of data exchanged through the interconnect, the execution time of an SI can be reduced. Memory traffic can also be reduced when less input data is required or less output data needs to be written back, due to reduced data width. Reduced resource utilization of approximated circuits can result in a smaller bitstream, allowing for faster reconfiguration of an RC during run time. Finally, using fewer resources may allow combining two related accelerators, which previously required two RCs, into a single one. This eliminates data transfer between these RCs on the interconnect, thereby allowing other data to be transferred in parallel to execution. As an additional benefit of merging related RCs, other accelerators can be loaded into a vacant RC to allow execution of another SI in parallel. In the case that RCs cannot be merged, processing time will still be improved as fewer cycles in a VLCW phase are required for data transfers internal to the accelerator fabric.

As image processing algorithms can often benefit from approximation, a case study using the SUSAN edge detection algorithm is shown in Section 5.3.2. A closer look at the arithmetic of this algorithm is taken and ways to apply different approximation techniques to the exact hardware implementation are highlighted. Section 5.3.3 discusses the results and benefits from applying approximate computing to our reconfigurable processor architecture, the *i*-Core.

Case Study: The SUSAN Edge Detector In this section, we demonstrate the impact of approximation on an implementation of the *SUSAN* edge detection algorithm running on our reconfigurable processor design.

The SUSAN algorithm is used to detect edges and corners in a grayscale image [79]. A common approach to edge detection is to calculate the brightness gradient through derivation, as it is used in the Canny edge algorithm [111] along with other factors. SUSAN uses a different approach, where the similarity of each pixel to its individual neighbors is computed. The similarity factor $c(\vec{r}, \vec{r_0})$ is calculated according to Equation (5.1) using a threshold t, denoting the intensity difference between the center of the current kernel and a surrounding pixel. At its core, this operation is similar to other computer vision algorithms using a small convolutional filter mask applied to each pixel.

$$c(\vec{r}, \vec{r_0}) = e^{-(\frac{I(\vec{r}) - I(\vec{r_0})}{t})^6} \tag{5.1}$$

$$n(\vec{r_0}) = \sum_{\vec{r}} c(\vec{r}, \vec{r_0}) \tag{5.2}$$

A high difference in intensity indicates a candidate for an edge, while pixels of similar intensity are grouped into a Univalue Segmented Area Nucleus (USAN). The homogeneity of the area surrounding each pixel, named USAN, is computed to denote the similarity of the pixels within this area (Equation (5.2)). An edge is found based on the difference between two neighboring USANs and their relative location, if the USAN value $n(\vec{r_0})$ is sufficiently small. To determine the direction of an edge, a weighted average of neighboring USANs is used:

$$\bar{\vec{r}}(\vec{r_0}) = \frac{\sum_{\vec{r}} \vec{r} \cdot c(\vec{r}, \vec{r_0})}{\sum_{\vec{r}} c(\vec{r}, \vec{r_0})} \tag{5.3}$$

$\vec{r}$ can be expressed as $\sqrt{x^2 + y^2}$ using the Euclidean norm. In Section 5.3.2, a hardware implementation and approximate acceleration of the SUSAN algorithm is presented.

As a first step of implementing the SUSAN algorithm within our reconfigurable processor, it must be decided which parts of the algorithm are most suitable for hardware acceleration. The USAN and the edge direction (Equation (5.3), in most cases) need to be independently calculated for all pixels and can be processed in parallel. Figure 5.15 illustrates the dataflow of the hardware implementation and shows the evaluation, decision and data repacking modules involved. The interconnect bandwidth of 128 bit

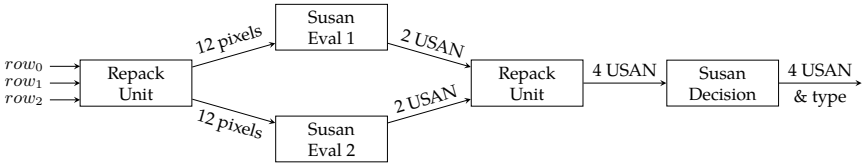


Figure 5.15: Dataflow graph of the exact SUSAN SI implementation with three different RC types

limits image data transfer to two adjacent kernels per phase, as a kernel consisting of nine 8-bit pixels is required for each USAN. In addition, the RC interface of 64 bit is not sufficient to transfer a full kernel at once, requiring two phases for the initial data transfer. Note that a kernel size of nine 8-bit pixels is an unfortunate fit for memory interfaces, which commonly have power-of-two data widths, i.e. 64 bit or 128 bit. Packed image data allows for loading consecutive memory words for optimal throughput, but data reordering is necessary to get aligned 72 bit words. To avoid expensive data reordering, our design evaluates four kernels at once per SI execution and maximizes usage of the 128 bit image rows fetched by the load/store units. Figure 5.16 shows the VLCW schedule for the exact SUSAN calculation. Each node color represents a different component of the reconfigurable fabric, where the SUSAN evaluation and decision units are implemented in RCs and the load/store units as well as the repack unit are implemented statically. Each unit can only perform one operation per phase, but multiple RCs can be programmed with the same accelerator. For example, SUSAN evaluation steps in phases 5 and 6 are performed by two RCs in parallel, and the phases are used as pipeline steps to compute all four kernels.

Three computationally intensive operations are necessary to evaluate each USAN: a division operation and the natural exponential function in Equation (5.1), and calculation of the square root in Equation (5.3). A straightforward High Level Synthesis (HLS) implementation of the algorithm includes a radix-2 divider [112] and a pipelined 6-stage CORDIC core for square root operations, as well as a LUT with fixed-point values of the natural exponential function. However, by analyzing the algorithm in detail and allowing for slight quantization errors, all CORDIC cores can be avoided and processing time is greatly reduced.

The threshold t in Equation (5.1) is constant and can be merged into the LUT for e^x, saving one division operation while maintaining accuracy. The center of gravity (Equation (5.3)) is used if the edge is on the border between two USANs. It requires

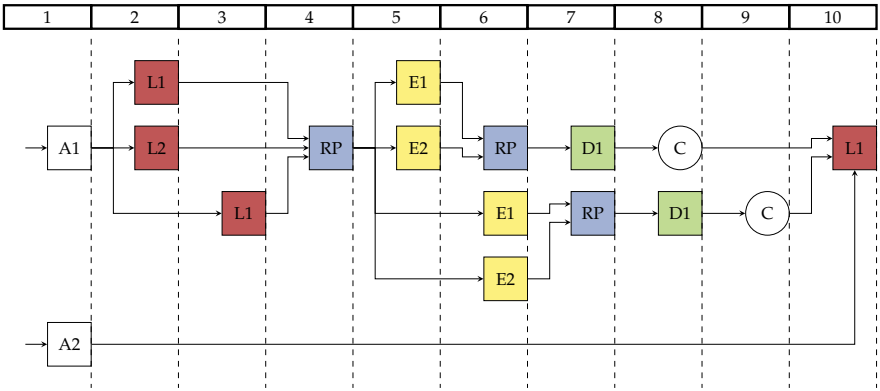


Figure 5.16: VLCW execution schedule of the exact SUSAN SI implementation with 10 phases. *A* denotes an addressing phase, *L* steps load or store data, *RP* is data repacking, *D*/*E* are SUSAN decision/evaluation steps, and *C* copies data between RCs.

the calculation of a square root for the Euclidean norm of $\vec{r}$, but since the result is only used in comparison to the *geometric threshold* G, it can be simplified as follows:

$$\sqrt{x^2 + y^2} > G = 0.4 \cdot n \tag{5.4a}$$

$$x^2 + y^2 > 0.16n^2 \tag{5.4b}$$

$$25 \cdot (x^2 + y^2) > 4 \cdot n^2 \tag{5.4c}$$

Now that there is no need for a square root operation, Equation (5.4c) can even be evaluated using integer operations only.

Finally, the ratio of $\overline{(y - y_0)^2}$ to $\overline{(x - x_0)^2}$ is used to determine the orientation of the edge by comparing it to 0.5 and 2.0 for the vertical and horizontal edge case, respectively. This can be rearranged to $2 \cdot |y| < |x|$ and $|y| > 2 \cdot |x|$, trading a division for two base-2 multiplications. These optimizations allow reducing hardware utilization and the number of cycles without sacrificing accuracy.

Approximation of the SUSAN algorithm Our optimized SUSAN hardware implementation requires the use of four RCs due to resource constraints. Approximate computing techniques can be applied to improve performance, further reduce resource usage and potentially merge related RCs. Figure 5.17 shows the execution

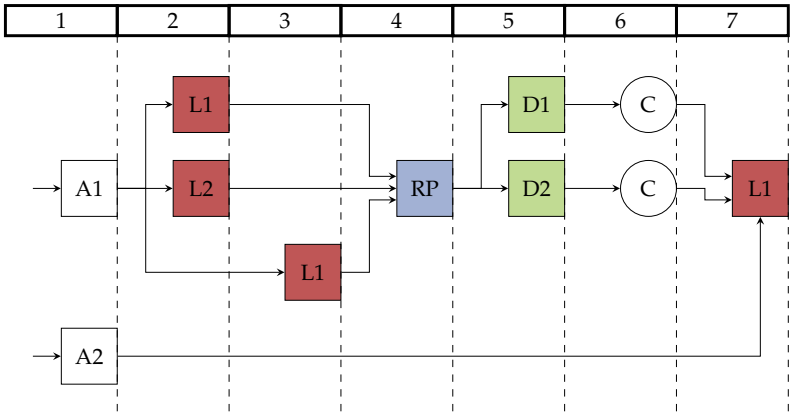


Figure 5.17: VLCW execution schedule of the first-stage approximated SUSAN SI implementation with merged evaluation and decision RCs

schedule with approximation for the exponential function LUT and the quadratic multipliers applied, which allows merging the evaluation and decision RCs. Calculating the Euclidean norm of $\vec{r}$ involves two quadratic multiplications. Using the technique proposed by Yao et al. [109], they are replaced with approximate multipliers optimized for FPGAs. Similarly, all adders in the evaluation and decision units are replaced by the approximate *GeAr* adder [106]. Both the adders and the multipliers feature a configurable level of approximation, which allow generating different variants of the hardware accelerated implementation.

In addition to the optimization of computational operations, advantages are also gained from the approximation of data types and their transfer between RCs. The exact implementation works on image data with 8 bit/pixel, which is also the source format in memory. An RC has two 32-bit inputs, thus two phases are required to load a 3×3 kernel into the evaluation RCs (phases 5+6 in Figure 5.16). By reducing the bit width to 7 bit/pixel using the repack unit, data can be loaded in a single phase and the subsequent repack step is no longer required. Nevertheless, the initial retrieval of the input data still takes up a significant portion of the run time. Processing a 3×3 kernel requires three rows of image data to be loaded, taking two phases (phases 2 and 3 in Figure 5.17) as only two load/store units are present. An *interleaved processing mode* is used for the second-stage approximation to compute the first kernel from only two rows of image data. The similarity of consecutive image rows is exploited as the data

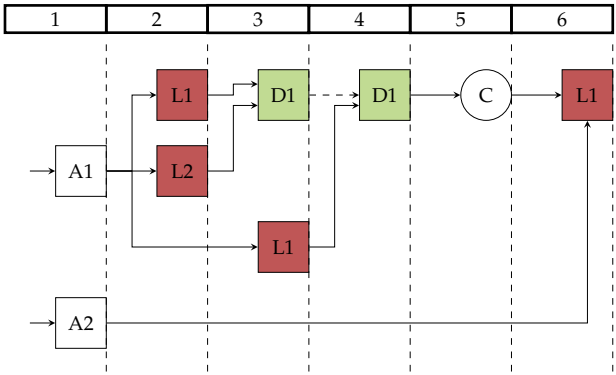


Figure 5.18: VLCW execution schedule of the second-stage approximated SUSAN SI implementation with reduced bit width RCs

of the second row is duplicated and also used virtually as the third row. Afterward, the second kernel is calculated with higher accuracy in the next phase, after the third row has also been loaded. The resulting schedule is shown in Figure 5.18.

5.3.3 Evaluation

The reconfigurable processor prototype is implemented on a Virtex-7 evaluation board (VC707) and runs at 50 MHz. We envision an ASIC implementation where the GPP core, interconnect and load/store units of the accelerator fabric are implemented as an FLP (i.e. hard logic), whereas the RCs are realized in the Adaptive Logic Partition (ALP) using an embedded FPGA. The length of a VLCW execution phase is set to four CPU cycles to emulate a critical path length 4 × higher on an FPGA compared to an ASIC. This also ensures an adequate timing budget within the RC, allowing them to implement multi-cycle operations. SIs are executed using the accelerator framework attached to the LEON3 processor core, consisting of five RCs of 1 200 LUTs and 20 DSP slices each. Both load/store units feature a 128-bit memory interface to 4 MiB of SRAM, which is also accessible through the main AHB bus of the processor. Run-time reconfiguration of the RCs is possible using the reconfiguration controller, which also allows automatic on-demand loading of bitstreams.

The SUSAN implementation of the *MiBench* suite is used as baseline software reference [100]. Edge evaluation and classification is replaced by a corresponding SI to

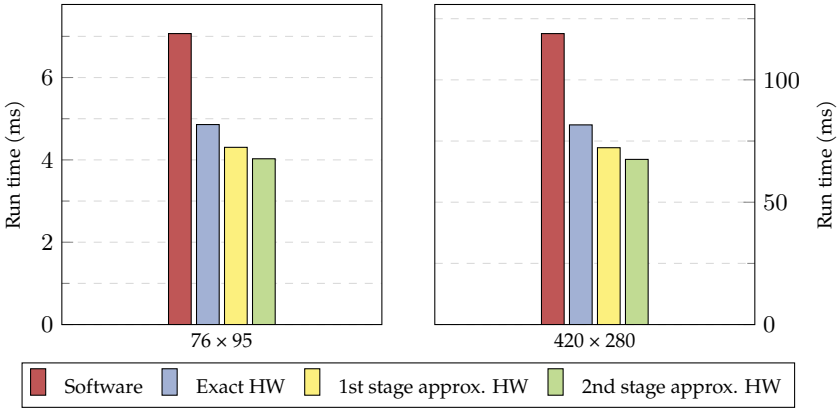


Figure 5.19: Execution time in SW and different HW implementations

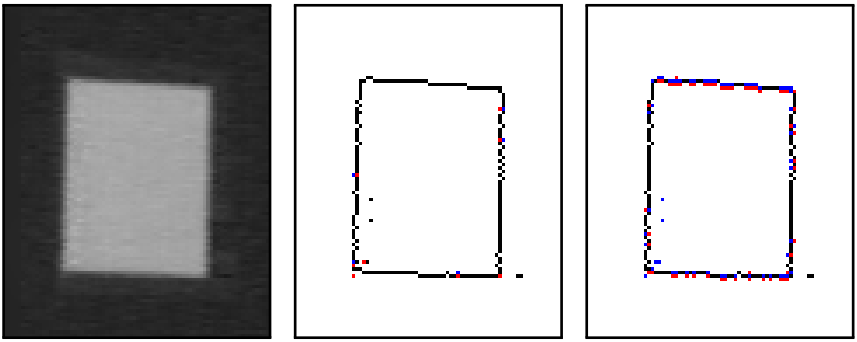
invoke the hardware implementation. Finally, edges are marked in the input image in software. Compared to processing in software, the exact hardware-accelerated implementation shows a performance gain of 43 %.

The approximate SUSAN implementation

In addition to the exact hardware implementation, whose output is identical to the software implementation, two approximate versions are evaluated: The first-stage approximate implementation uses *LM-2* multipliers [109] and *GeAr* adders with medium approximation settings[2]. The second-stage approximate implementation uses *LM-NC* multipliers, adders with twice the number of approximated bits, and the interleaved processing mode is used.

Comparing both a simple 76 × 95 pixels image showing a single rectangle and a 420 × 280 image, the first- and second-stage approximations reduce run time by 11.4 % and 17.1 % respectively (Figure 5.19). Note that the total run time of the algorithm is measured, including image read-in and highlighting of detected edges in the input image. The non-accelerated operations last 3.6 ms and 99.1 ms for the small and large test patterns. When only taking the process of edge detection and classification into account, the performance is improved by 44.6 % and 67.0 % for the 1st/2nd-stage approximations. It is noticeable that the pixel processing rate is independent of the

[2] Adder settings $N, R, P = 8, 2, 2$ according to Shafique et al. [106]



(a) Rectangular test image with noise, 76 × 95 pixels (b) First stage approximation (c) Second stage approximation

Figure 5.20: Edges detected in a small test image, misdetected edges marked in red or blue if they are additional or missing w.r.t. exact implementation

image size and almost constant, being 708, 751 and 775 pixels/s for the exact, 1st-stage and 2nd-stage approximated cases, respectively.

A visualization of the impact of the two approximation stages is shown in Figure 5.20. The input image 5.20a contains a single rectangle, filtered with Gaussian blur of $x = y = 0.5$ and 2 % of random noise. The first-stage approximation misdetects a total of 13 edges, 5 of which are incorrectly detected and 8 of which are undetected. In comparison, the second stage approximation misdetects 27 edges, with 12 false positive and 15 false negative edges. Edges that are not present in the output of the exact implementation are marked in blue, superfluous edges are colored in red. It can be seen that the second-stage approximation misdetects a considerable number of edges one pixel above the correct location, which can be explained as a result of the interleaved processing mode. Since in such cases the third image row is a duplicate of the second row, the center of mass for an edge is moved to the center of the two upper rows. Additionally, some pixels are marked as edges by both the exact and first-stage versions due to noise, where no edge is present in the input image. The second-stage approximation does not mark these pixels. This observation indicates that approximation introduces a kind of low-pass character, which stabilizes the actual algorithm. We can conclude that even with strong approximation, the design can still properly detect simple geometries.

Variant	1st stage approx.		2nd stage approx.	
	absolute	relative	absolute	relative
edge direction	2 807	2.39 %	3 805	3.24 %
edge presence	1 924	1.64 %	2 657	2.27 %
false-positive edges	976	0.83 %	1 365	1.16 %
false-negative edges	948	0.81 %	1 292	1.11 %

Table 5.4: Error analysis for a 420 × 280 image (*Hundertwasserhaus*)

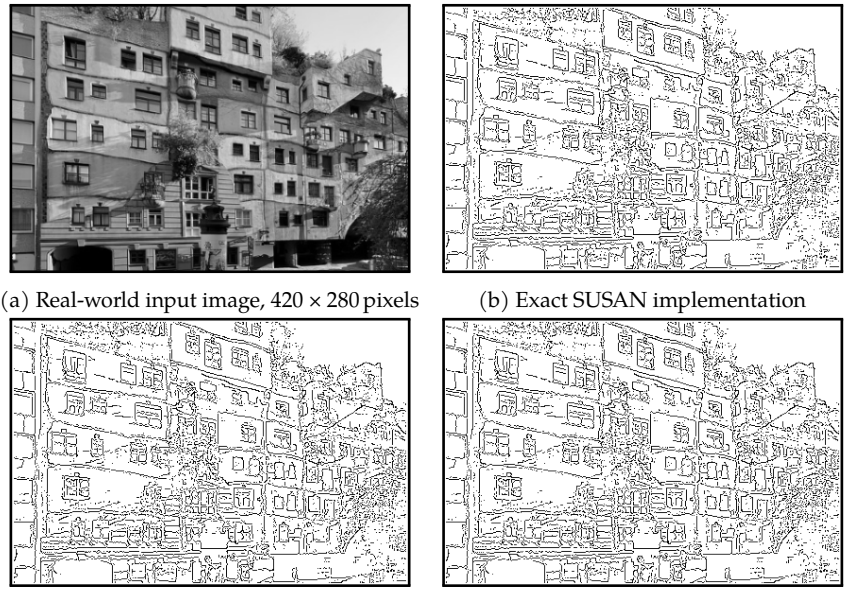
Variant	Module	# of RCs	LUTs	DSPs
Exact implementation	Evaluation	2	762	1
(4 RCs total)	Decision	2	748	0
1st stage approximate	Combined	2	837	0
2nd stage approximate	Combined	1	653	0

Table 5.5: Resource usage in numbers of reconfigurable container (RC) and FPGA primitives. The reduced resource usage of the approximated variants allows using fewer RCs in total.

Table 5.4 shows an error analysis for the large test image. The number of misdetected edges (edge presence) and incorrectly detected edge directions is evaluated separately. It can be seen that fewer edges are incorrectly predicted because fewer approximated operations are required, compared to determining the edge direction. In contrast, the evaluation result of the edge direction is only used if an edge is present, whereby many errors are suppressed.

For comparison, Figure 5.21 shows results for the large input image alongside the detection results, converted to grayscale. No additional filters were applied in order to obtain a test case with real-world data that was as authentic as possible. Visually, differences between the resulting output images are barely recognizable, since even in Figure 5.21d, less than 3.91 % of the pixels are incorrectly marked. This confirms that the approximate variants can also be used for more complex geometries, but the suitability for the respective application must be checked individually.

The greatest impact of the approximate implementations is seen in resource use as displayed in Table 5.5. The exact implementation requires a total of 4 RCs, consisting



(a) Real-world input image, 420 × 280 pixels (b) Exact SUSAN implementation

(c) First stage approximation (d) Second stage approximation

Figure 5.21: Comparison of the exact and 1st/2nd stage approximated edge detection on a photograph of the *Hundertwasserhaus*. Only minor differences in detected edges are visible with all major edges being represented for the approximate designs. *Original Photo by C.Stadler/Bwag; CC-BY-SA-4.0*

of two evaluation modules and two decision modules. Due to the resource limitations of an RC, the modules cannot be merged further. The approximate versions however show a greatly reduced resource utilization, therefore both the 1st and the 2nd stage allow the evaluation and decision modules to be combined. The first-stage approximate implementations require 2 RCs for parallel processing of two neighboring kernels. As the second-stage approximation makes use of the interleaved processing mode, one kernel can be evaluated earlier and one single RC is used in a pipelined way. Hence, the approximate implementations are especially useful, if the amount of available RCs is not sufficient to hold all modules required by the application running on the reconfigurable processor. By selecting an approximate implementation, fewer RCs are required for SUSAN acceleration and allow implementations for other SIs to be loaded concurrently. The resource limit of the RCs can of course be adjusted in the

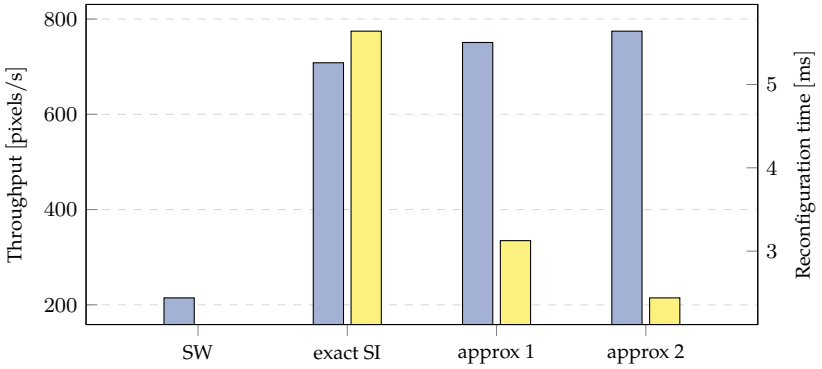


Figure 5.22: Performance of exact and approximate SUSAN designs compared to the time required to load the respective design into RCs.

FPGA prototype, but in a potential implementation as an ASIC one will always reach resource limits at a certain point.

Run-time Reconfiguration

Reconfiguration during run time allows to switch between exact and approximate implementations when necessary. Our experimental use case is a mixed-critical application running SUSAN edge detection with a constant frame rate. In addition, the system needs to apply AES-256 encryption to an incoming data stream using the implementation described in Section 5.2. The AES implementation requires two RCs to be available, whereas four RCs are used by SUSAN. Figure 5.22 shows the required reconfiguration time to switch between the exact, first-stage and second-stage approximate implementations. Loading the second-stage approximate implementation to free up two RCs takes 5.61 ms, which is just 8.31 % of the processing time of a 420×280 image. In our use case the additional delay is acceptable for one frame, allowing the encryption to be accelerated by 70 %. The implementation of different approximation levels offers a high degree of flexibility and allows for the selection of a Pareto-optimal operating point between desired accuracy and number of available RCs.

5.3.4 Conclusion and Outlook

In this section, the applicability of different approximate computing techniques to reconfigurable processors has been investigated. We have shown a significant impact especially when hardware limitations and bottlenecks due to resource constraints are encountered. Approximate computing allows overcoming such limitations and enables simultaneous acceleration of different algorithms on the same reconfigurable processor. We performed a detailed analysis of an image processing algorithm and, by applying different approximation methods, achieved significantly improved processing performance of the hardware accelerator while greatly reducing resource consumption. It can be seen that particularly good results can be achieved by approximation techniques specifically adapted and optimized for a given algorithm.

It must be emphasized that the primary benefit of approximation on reconfigurable processors does not stem from enhanced performance, but rather from significant resource savings. The fact that hardware resources are statically partitioned into reconfigurable containers leads to a quantization-like effect on the number of required containers. Even if a design requires just a little more than the resources of one RC, two entire containers are occupied and cannot be used by others. Approximate computing makes it possible to dynamically adjust the amount of resources required, if necessary only for individual, targeted operations. This results in a freely selectable design parameter between the accuracy of the application and required resources, which can be tuned using DSE. In this way, the use of resources can be reduced and the utilization of RCs optimized.

Given this work as a foundation to approximate computing within the *i*-Core platform, automatic DSE to find optimal configurations with respect to accuracy, performance and resource usage can be investigated in the future. We experienced that manual optimization and insertion of approximate circuitry is a tedious task. A first step towards automatic analysis and approximation of specific hardware accelerator blocks is creating the necessary tools to identify candidates for replacement in the design, for example in a netlist. Synthesis tools commonly parse Hardware Description Language (HDL) into an intermediate language, where approximation techniques could be applied to abstract Register Transfer Level (RTL) operations. Additionally, HLS tools could also be extended with a way to specify the required accuracy for an operation, and an appropriate approximated circuit will be selected automatically. Based on our results, we expect that automatic approximation will bring significant performance

improvements and substantial resource savings, enabling the acceleration of a wide range of algorithms while reducing implementation effort.

5.4 Resource Sharing and Runtime Segmentation

Based on the original publication *Low-latency inter-domain communication on the Xen hypervisor* at MCSoC 2023. [LHB24]

Throughout many technological domains, the requirements for embedded computing systems have risen tremendously in the last two decades, in terms of the number of implemented functions, their complexity and demands on communication [113]. As a result there is an increasing trend for mixed-criticality systems, where multiple components with different dependability and real-time characteristics (e.g. safety-critical and consumer functionality) are integrated into a shared computing platform [114]. The reason behind this trend is lowering costs and power consumption as well as improving maintainability [115].

In Sections 5.2 and 5.3, we have showcased run-time reconfigurable processors alongside techniques for performance improvement and different applications scenarios. We will now focus on the integration of such processors in embedded systems, following the goal of building highly flexible, yet powerful systems. The *i*-Core design supports being part of a multiprocessor System-on-Chip (SoC) and even allows other processors to share the accelerator fabric of a single *i*-Core. Consequently, a multiprocessor reconfigurable processor system serves as a solid foundation for the development of high-performance embedded systems. Multiple applications can run in parallel on the SoC and benefit from the accelerator fabric, which also supports resource sharing on the application level. For safety-critical applications, however, the concern arises as to how sufficient isolation between different applications can be ensured.

Virtualization technology can provide computational isolation among multiple applications on a common hardware platform [116]. It is increasingly used as a key technology in the middleware stack of embedded systems, whenever stronger isolation compared to an operating system is required. This includes mixed-criticality systems as well as cloud computing devices which execute applications from different ecosystems or with different requirements in terms of response time, computational performance or communication bandwidth in parallel. In many use cases, it is sensible to break up the functionality into multiple small partitions, as these can be developed independently and can be considered independently from a security perspective. Additionally, this simplifies the formalization and monitoring of requirements for each isolated module according to the safety standards targeted. In the automotive

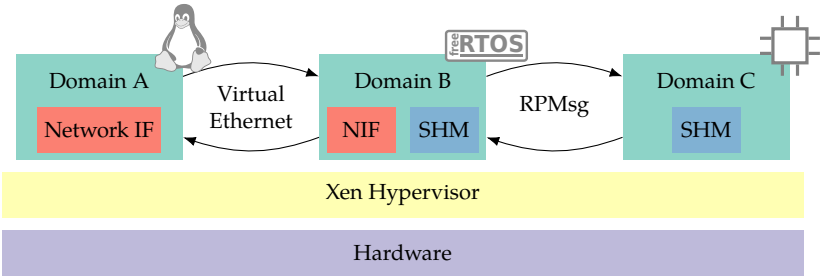


Figure 5.23: A typical hypervisor setup with multiple heterogeneous guest domains. Domains A/B are communicating using ordinary virtual networking with high latency, whereas B/C use the proposed concept using RPMsg and shared memory with very low latency.

industry, for instance, the requirements for functional safety alongside the increasing number of Embedded Computing Units (ECUs) are already leading to more centralization and thus to virtualized approaches [117].

The benefits of virtualization do not only apply to independent, strongly isolated applications. By dividing components of a large, interconnected system (e.g. control loops or image verification chains) into multiple partitions, it is easier to service, update and monitor the individual components. However, such fine-grained subdivided modules require reliable and rapid communication with other modules and the outside world. In industrial practice, two technologies are commonly used: On the one hand, a virtualized network can be used, either as an emulation of real hardware or as a paravirtualized device, similar to hypervisor applications in datacenters. This is flexible and easy to use but the latency overhead caused by network virtualization has to be considered. Especially on less powerful embedded systems, this overhead becomes significant when numerous guest domains are connected using virtualized network adapters. On the other hand, communication based on Shared Memory (SHM) can achieve higher performance, but is less flexible and typically more complex to implement. To enable efficient Inter-Domain Communication (IDC) on embedded systems, a solution is required that offers the performance of SHM but is similarly flexible and easy to use as virtualized networks.

In the following, an approach for low-latency IDC between guest domains on a virtualized platform is presented. The Xen hypervisor was selected for the availability and open source nature of the software, which allowed us to easily inspect its in-

ternals [116]. Figure 5.23 shows a typical hypervisor setup where two domains communicate using virtualized network adapters and another pair uses an improved IDC mechanism based on SHM with minimal latency. While covering the technical details, evaluation and an example use case, the following chapters also focus in particular on existing open-source standards and tested implementations thereof, which are combined to build the proposed IDC system.

5.4.1 Related Work

The Xen hypervisor is equipped with fundamental mechanisms for inter-partition communication [116]. Virtual network devices are the most common type of device in this category. These devices allow a guest domain to participate in Ethernet networks and reuse the network stack of the operating system or runtime environment used by the guest software. The conventional approach of emulating a real-world network interface within the hypervisor suffers from high overhead. Xen also supports the Virtual I/O Device (VIRTIO) backend for several device types, which is optimized for use in virtualized environments [118]. Therefore, while high bandwidths are achievable, they still fall short of the performance offered by physical Ethernet interfaces [119]. However, the requirement for networking stacks and virtual Ethernet endpoints on both sides significantly increases latency [120].

In contrast to virtual devices for guests, one increasingly finds hardware with support for concurrent access by multiple virtualized guests [121]. Such integrated virtualization features allow for higher performance than a paravirtualized variant and reduced hypervisor overhead and complexity. Figure 5.24 depicts a hypervisor-based setup with three virtual machines on a hypervisor. Peripherals in the system utilize PCI Express (PCIe) Single-Root I/O-Virtualization (SR-IOV) to present themselves individually to each guest. This allows the guests to access the devices directly without arbitration overhead, and enables the device itself to manage the segregation of its different users. Given that the primary focus of our approach is transfer of data between guests, the use of peripheral virtualization features is not an option.

A similar concept can be applied to applications in the automotive domain using CAN-to-Ethernet communication [120]. As software certification and testing is a tedious task in automotive system design, it is beneficial to modify existing solutions as little as possible when transferring them to a new architecture. This work presents an approach using CAN interface emulation for legacy applications, running as a

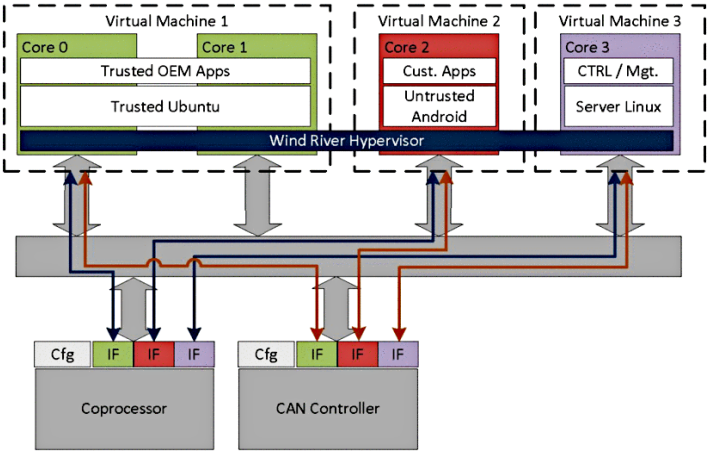


Figure 5.24: Heterogeneous multiprocessor system by Sander et al. where multiple hypervisor guests share PCIe-based peripherals. The devices implement multiple virtual endpoints which can be used by the different guests independently [121].

hypervisor guest, and converting it to Ethernet interfaces on the hypervisor level. However, in contrast to the design in Figure 5.24, the approach is purely software based and reaches latencies between 10 ms and 20 ms on embedded platforms. The biggest disadvantage is the overhead caused by the emulation of the native CAN interfaces, and the additional latency is particularly significant when there are several transitions between guests.

Further developments like *PrioIDC* [122] effectively improve latency by adapting the communication architecture to the schedule of the hypervisor. The authors deliberately stick to communication over network interfaces to support unmodified guest domains. As the existing scheduling and network communication mechanisms in Xen show several limitations, a Real-Time Communication Architecture (RTCA) is introduced in the management domain *dom0*. By combining this new architecture with improved scheduling, latencies of 324 µs can be achieved for the maximum batch size of 238 packets. However, *dom0* is still required to process each packet, and performance degrades to 4 643 µs when it is under heavy load. Thus, we focus on an SHM approach without involving *dom0* for handling packets.

	Ours	Xen	PrioIDC	C2E	VOSYS	TZ-VirtIO
Minimum latency	✓	✗	✗	✗	✗	❍
High throughput	✓	❍	✓	❍	❍	✓
Modularity	✓	✗	✓	✓	✓	✗
Service-oriented	✓	✗	✗	✓	✗	✗
Portability	✓	✓	✓	✓	✗	✗

Table 5.6: State-of-the-art Inter-Domain Communication approaches compared to our proposed mechanism

Monitoring is the main goal of *VOSYSmonitor* [123], a software layer based on ARM TrustZone. It can be used to run safety-critical and best-effort software on the same device with isolation while maintaining automotive standards. The communication interfaces are implemented as network adapters, which introduces additional overhead for IP stacks on each guest domain. Porting the approach to other communication methods like SHM is not discussed.

The authors of *TZ-VirtIO* [124] present a lightweight IDC mechanism for platforms using the ARM TrustZone isolation mechanism. Their method combines VIRTIO and Remote Processor Messaging (RPMsg) on a shared memory partition to connect secure and non-secure guest partitions. Inter-Processor Interrupts (IPI) are used as the event channel between both partitions, utilizing the hypervisor to deliver the interrupts to the guests. Their design uses the LTZVisor hypervisor, which specifically targets ARM TrustZone machines, and does not transfer seamlessly to other, generic hypervisors like Xen. Additionally, the achieved latencies are still in the order of ms and are therefore insufficient for our application. Latencies also increase linearly with the size of the payload, which indicates that they have not implemented true zero-copy behavior.

Table 5.6 summarizes the aforementioned IDC mechanisms and compares them with regard to different aspects. None of the approaches feature both sub-ms latency and still allow high throughput. In addition, most works do not support service-oriented communication frameworks natively or are not portable to other hypervisors. With our work, we are targeting this gap by providing a low-latency IDC with modular and service-oriented design. It is important to keep the overhead per guest as low as possible so that our mechanism can scale to a very large number of guests.

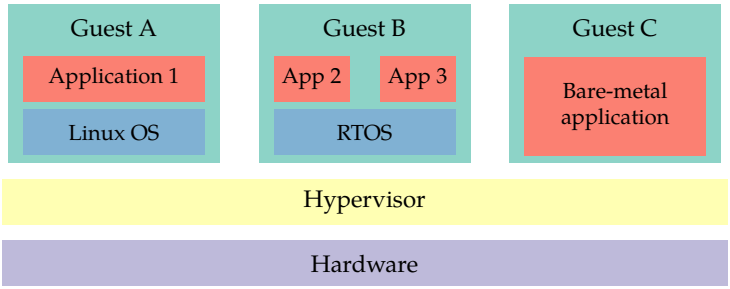


Figure 5.25: Type-1 hypervisor with multiple guest domains

5.4.2 Inter-Domain Communication

When multiple guests on a hypervisor are required to work cooperatively, they will need to exchange data. Mechanisms for such data transfers are called Inter-Domain Communication (IDC), and most hypervisors provide certain IDC mechanisms out-of-the-box. However, the standard implementations are often not optimized for minimal overhead and latency. As we aim to deploy safety-critical applications using multiple hypervisor guests, communication latency must be minimal and predictable. Additional functional features are also required: first, service-oriented communication frameworks are commonly used in the automotive industry and must also be supported by the communication mechanism. Second, run-time monitoring of the deployed applications is necessary to ensure correct functionality. This section will describe *Indoco*, our IDC mechanism with minimal latency, service-oriented communication framework and monitoring features.

To realize IDC, the hardware features of the target platform have to be considered. In most modern systems, means for communication between hypervisor domains or between hypervisor and guest domains are limited to SHM, interrupts and hypercalls. Mechanisms on higher layers like virtualized devices (e.g. Ethernet) are using these features in the background.

Figure 5.25 shows a type-1 hypervisor and a choice of different guests running on top of it: a Linux domain, a real-time operating system (RTOS)-based application and bare-metal software. The hypervisor provides computing and memory resources to guests while establishing isolation between individual guests and scheduling their execution as configured. Input and output is usually realized using virtual devices created by the hypervisor (emulated or paravirtualized devices [125]), e.g. providing

access to Ethernet networks [120]. It is also common to pass physical hardware to individual guests which then have exclusive access at near-native performance [121]. However, the number of physical interfaces is limited and even pass-through can introduce significant overhead [119] and is therefore out of the question for IDC.

Paravirtualized devices are designed with the hypervisor in mind leading to reduced emulation overhead and simplified drivers on the guest. However, a paravirtualized Ethernet controller implies a lot of overhead for guest domains: In addition to the device driver, an Ethernet, IP and TCP/UDP stack is necessary to participate in common networks. While easy to use in a fully fledged guest OS like Linux, the complexity of bare-metal guests and some RTOSs increases greatly if such functionality is included. Therefore, we propose a lightweight IDC system with very little overhead. Designs with many guest domains (e.g. in a microservice environment) will especially benefit from low latency and a small memory footprint of individual guests.

Shared Memory Management

From a guest perspective, SHM is the only resource allowing to exchange large amounts of data with other domains or the hypervisor. This SHM region needs to be managed in a uniform way by both communication partners. The VIRTIO [118] specification defines a standard interface for paravirtualized devices. Although the notion of frontend (guest driver interface) and backend (device model on host side) is tailored to paravirtualized devices, the VIRTIO mechanism can be used for IDC as well: To establish a peer-to-peer channel, one domain takes the role of the frontend domain, while the other domain acts like a device (backend). VIRTIO can define two ring buffers, called *virtqueue* in the context of VIRTIO, in the memory of the frontend domain. One queue is serving as the transmit (TX), the other as the receive (RX) channel.

Several operating systems and runtime libraries include drivers for VIRTIO. These drivers are usually well tested and can be expected to be more stable than a new custom driver. Multiple backends are specified for VIRTIO, most importantly PCIe and Memory Mapped I/O (MMIO). As a PCIe bus introduces additional overhead, the MMIO backend is preferred.

VIRTIO over MMIO is realized as follows: Both frontend and backend share a memory region containing virtqueues as well as configuration and status space (Figure 5.26). Therefore, one single SHM region is sufficient to set up the VIRTIO connection. Notifications can be delivered using status bits in the status region, additional features

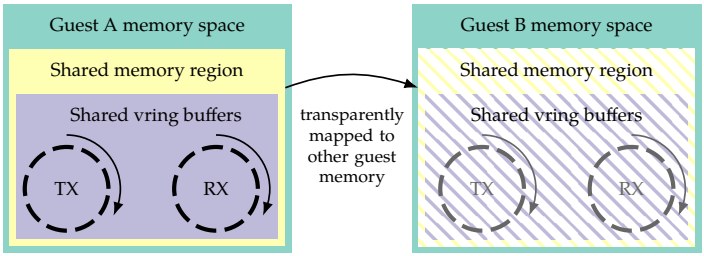


Figure 5.26: Two virtqueue ring buffers (vring) shared between two guests

like interrupts can also be used, although this is not explicitly defined in the VIRTIO standard.

To set up a VIRTIO channel, frontend and backend domains need to exchange information, which primarily consists of the shared memory location/size and the driver status on both sides. As VIRTIO does not define how such communication should be implemented, it is usually performed by the backend mechanism. When using PCIe, status information exchanged using a Base Address Register (BAR). However, for the MMIO backend, such a channel needs to be set up independently. In our case, the Xen hypervisor provides the `xenstore`, a key-value store to exchange status information with the hypervisor and optionally other domains. It is accessible via a SHM region available to every Xen domain by default. The hypervisor manager is supposed to publish values in the `xenstore`, thus deciding which domains are allowed to communicate. Usually this is done by the privileged domain *dom0*, but any domain with write permissions on the `xenstore` root can be used. The virtual device abstraction layer of Xen, `xenbus`, resides in the `xenstore`. The Xen project defines semantics for devices and drivers to exchange static information and state via the `xenbus` [126]. Note that accesses to `xenstore` are implemented using hypercalls and the end-to-end latency is in the order of ms. However, it is only used to exchange configuration data and set up the IDC channels, which are then used for application data.

It is possible to assign static SHM regions between domains, but doing so is only recommended between trusted domains according to the Xen manual [127]. Thus, Xen *grants* are used to dynamically set up SHM, using the `xenstore` to negotiate with the hypervisor. Compared to static allocation, no memory needs to be reserved prior to guest startup. To set up SHM between two domains A and B, domain A can select

a region within its guest memory to be shared and register it with the hypervisor as a grant. The privileged domain can define security attributes on grants to allow a second domain to use it. The second domain can then use a hypercall to mount the grant using information from the `xenstore`. The SHM region is now ready to use and can be shut down by reversing the setup procedure.

RPMsg

RPMsg is a scheme for data exchange between processors [128]. It was originally designed for asynchronous multiprocessors and is commonly used to access independent coprocessors in a SoC. However, since RPMsg is based on VIRTIO, it maps seamlessly to SHM between guests. It uses two virtqueues, one serving as TX, the other as RX channel. On top of the virtqueues, multiple independent *endpoints* can be created, allowing to virtually divide the channel into multiple logical channels for individual purposes. As an additional benefit, the RPMsg header is very short with only 16 B in size. Implementations of RPMsg are available in the Linux kernel starting from version 3.4 as well as in the OpenAMP framework, which can be used in both bare-metal and RTOS systems.

Small, irregular messages and continuous data streams can be transmitted in the same way, making it suitable for different applications. If the drivers are set up properly for the respective target platform, almost zero-copy behavior is possible. At minimum, a copy between user space and the kernel-managed virtqueue is necessary on Linux. On bare-metal, true zero-copy is possible.

Setup and teardown of VIRTIO via `xenbus`

The process of setting up and tearing down a communication channel is based on a state machine in both the frontend and backend driver. This state field resides in the `xenstore`, reusing the `xenbus_state` definition by Xen. After domain startup, both frontend and backend wait for configuration keys to appear in the `xenstore`. The process is visualized in Figure 5.27 and consists of the following steps:

1. Both frontend and backend wait for configuration keys to appear on the `xenbus`.
2. As soon as the frontend detects a matching root entry, it publishes vendor/device IDs and virtqueue information.

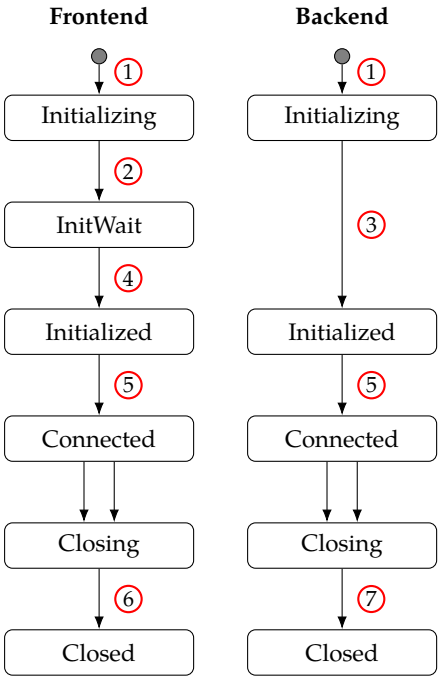


Figure 5.27: State machine life cycle on frontend and backend; numbers on state transitions refer to the detailed description in Section 5.4.2

3. The backend initializes virtqueues and allocates memory grants and event channel ports. It also creates the local VIRTIO and RPMsg devices, if the device ID matches.
4. The frontend can now mount the grants using the published information and create the VIRTIO and RPMsg device counterparts.
5. The communication channel is now ready for use. Both sides wait for the other endpoint to switch to the *Closing* state.
6. To close the channel, the frontend releases the device and SHM resource first, signaling completion by entering the *closed* state.
7. The backend can release all resources after the frontend has disconnected, the channel is now closed.

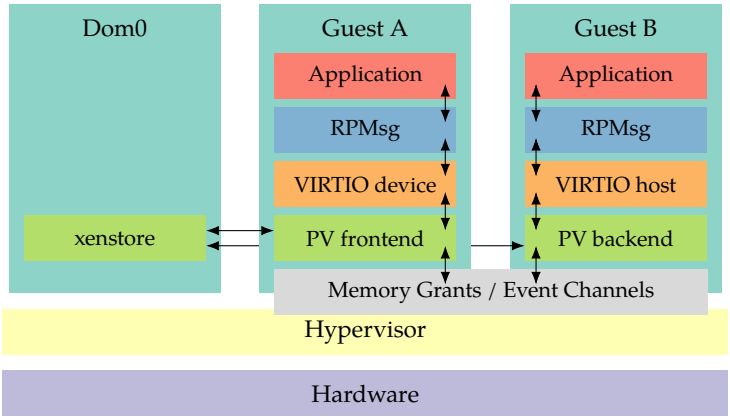


Figure 5.28: The full communication stack between two domains including xenstore for management

After the channel is closed, the `xenstore` values can be cleaned up by the hypervisor manager. We do not provide any measures for error handling during channel setup, as the cause must be an implementation or configuration error. In case of an error, the drivers stay in their current state and need to be reset manually.

Driver stack

Figure 5.28 shows the full software stack on both frontend and backend. Basically, the stack is identical for both sides, except for initialization differences between backend and frontend. Parts of the stack can reside in kernel space when using an OS, but the boundary between kernel and user mode can be chosen freely to fit implementation needs. In our reference implementation, a channel between a Linux domain implementing the frontend and a bare-metal domain in the backend role is shown. First, the Linux driver stack is discussed, followed by a description of a driver stack built into a bare-metal domain using OpenAMP.

The Linux kernel includes RPMsg drivers, which provide a character device to user space applications as well as a backend driver binding to VIRTIO (`virtio_rpmsg_bus`). Naturally, Linux also provides a VIRTIO driver infrastructure, whereof our design uses the `virtio_ring` module primarily. A framework to read, write and receive notifications via the `xenbus` is also available, providing an interface to allocate SHM grants and event channels as well. Here, our main contribu-

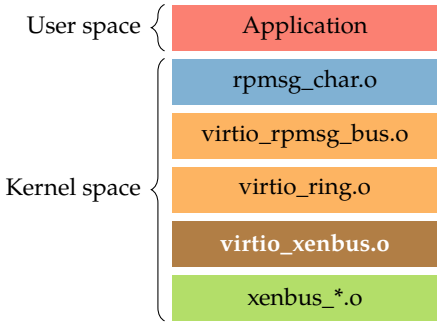


Figure 5.29: Driver stack on Linux side (frontend)

tion is the missing link between these frameworks: a driver called `virtio_xenbus`, which allows to create VIRTIO devices (and subsequently RPMsg sub-devices) based on `xenbus` entries. Figure 5.29 shows the stack of drivers and the location of the `virtio_xenbus.ko` driver in it, connecting `xenbus` and VIRTIO interfaces. Here, the Linux side implements the frontend role, allowing to use the Linux kernel interfaces to allocate grants and event channels as well as to initialize virtqueues and shared structures within the SHM. However, this is not an inherent limitation of the proposed architecture: frontend/backend roles can be exchanged easily and communication between Linux/Linux or bare-metal/bare-metal is possible in the same way.

In contrast to the Linux implementation, the bare-metal stack uses VIRTIO and RPMsg implementations of OpenAMP. They turned out to be well tested and stable for this use case and no modifications to the OpenAMP code are necessary. However, OpenAMP does not provide drivers to access the `xenstore` and manage `xenbus` data, and also lacks interfaces for the grant table. We contribute these features as part of an additional library, alongside the state machine logic (Section 5.4.2). It forms the functional equivalent to the `virtio_xenbus` Linux driver, but as an OS-independent library. This library makes it easy to realize our proposed scheme within other frameworks such as RTOSs.

Service-oriented communication

RPMsg provides a simple interface to send and receive unstructured binary data, allowing to easily adapt it to different use cases. To demonstrate its versatility, we

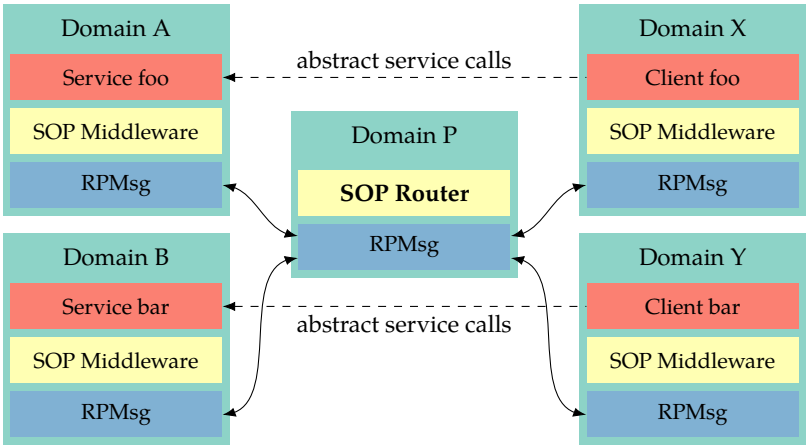


Figure 5.30: Structure of multiple domains communicating with SOP over the RPMsg connection

implemented a library for service-oriented communication on top of RPMsg. This library called Service-Oriented Protocol (SOP) allows domains to issue method calls to services implemented in other guest domains, handling serialization of parameters and results as well as asynchronous completion transparently. Since the underlying communication channel is a point-to-point connection, an extension is necessary to communicate with multiple domains. Therefore, we introduce a *router* component running on a Linux domain. Figure 5.30 shows an example architecture of multiple bare-metal domains where two clients on domains X and Y access services running in domains A and B.

It is also possible to establish multiple RPMsg connections per domain. However, the router component allows to enforce communication rules, for example for security purposes. It also opens up a possibility to monitor system behavior based on communication patterns. Using SOP, we can build service-oriented systems which are split into multiple guest domains on a low level, up to distinct domains for each available service. In Section 5.4.3, we demonstrate a control algorithm which is split into multiple domains using SOP with negligible overhead.

Real-time monitoring support

Monitoring is a common technique to verify if a component is operating in a specified environment. Especially for safety-critical functions, monitoring the execution state helps to improve the reliability of such a system [129]. The basic idea is to define valid execution states of a software component during development. At execution time, a component called *monitor* verifies that the application as well as the environment are within the specified, valid states of execution at all times. If a violation is detected, the system designer has multiple options to take action depending on the criticality of the failing component:

- Reporting: In case a misbehavior of a component is not critical to the system or the user, reporting the incident to the developer can be sufficient.
- Rollback: If the system was previously updated to a newer version of the component, returning to a previously known-good version can solve the issue. The known-good version can optionally be kept running (*hot-standby*) after an update to be able to perform a faster rollback.
- Degraded functionality: If feasible, the system can enter a state of degraded functionality where the violated state of execution does not occur. Manual intervention is required to return to a state of full functionality.

It can easily be seen that hypervisors fit very well into the monitoring concept. When large applications are split into multiple domains, it is very easy to replace single components (i.e. domains) to perform updates or rollbacks. However, monitoring individual functions of applications can be difficult in safety-critical conditions, as the performance impact cannot be neglected [130]. Thus, we propose monitoring on domain level, where IDC takes place. Figure 5.31 shows an example architecture for a dedicated partition monitoring another partition.

The router component of SOP is dynamically configured to forward specific requests to be monitored, as defined by the monitor partition, to said monitoring partition. The original communication remains unchanged, and the communication partners are not influenced by the monitoring. This approach is limited to monitoring of the state and parameters which are exchanged during communication, as internal values of each component cannot be observed. However, this limitation is less significant in cases where applications are divided into domains on a very low level of granularity, where monitoring takes place close to the class or module level. The impact on communication performance can be minimized by configuring the hypervisor to

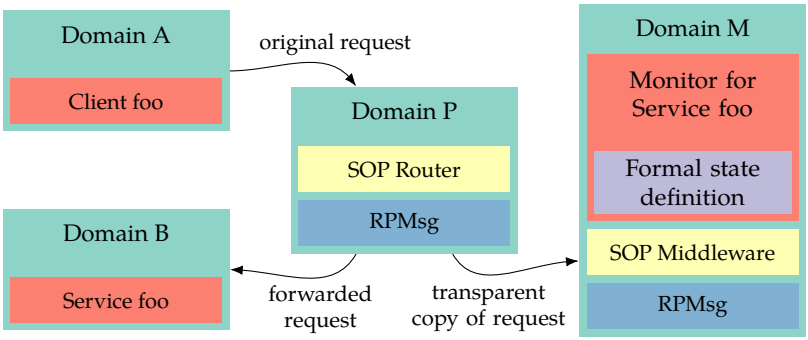


Figure 5.31: Architecture with a dedicated partition to monitor a service using IDC. The router component transparently forwards communication to the monitor.

execute the monitor on a separate processor. As a copy of the message data is created, memory bandwidth is the primary performance factor then.

5.4.3 Evaluation

This section details the performance evaluation of our approach, beginning with an analysis of round-trip latency. We measure this latency using synthetic benchmarks and compare it against common alternatives. To demonstrate practical relevance, we then showcase a real-world use case where an Adaptive Cruise Control (ACC) algorithm is implemented across several split guest domains. Our experimental setup is based on a Xilinx ZCU102 development board featuring the XCZU9EG MPSoC. It features an ARM Cortex-A53 64-bit quad-core processor, running Linux 4.19 as the privileged domain on the Xen 4.13 hypervisor.

Round-trip latency

The measurement setup for round-trip latency is as follows: an unprivileged bare-metal guest is connected to a Linux guest. The bare-metal guest is configured to return incoming messages unmodified as soon as they are received. On the Linux guest, we use the *benchmark* library by Google [131] to run tests of different block sizes, 10 000 iterations each. Figure 5.32 shows the results and compares them with other common approaches. Note that the *loopback*, *Unix datagram* and *UDP* (*local*) cases

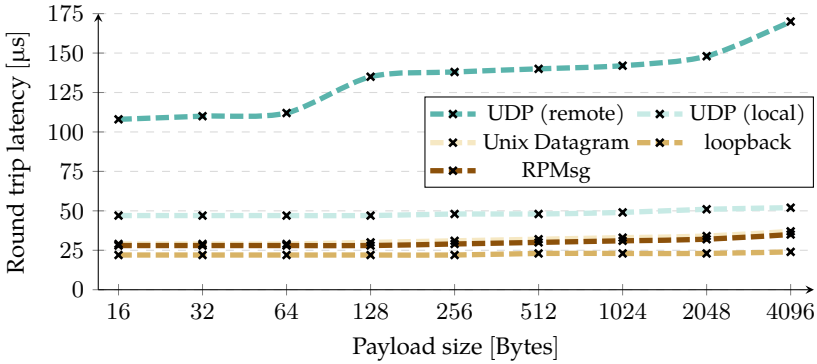


Figure 5.32: Round trip latency over payload size, compared to other common communication mechanisms

do not involve data transmission between different guest domains. These cases use loopback mechanisms locally within one Linux guest and are provided for reference.

We can show that the full round-trip latency is about 30 µs for a 512 B payload. Due to the zero-copy implementation, latency does not increase greatly for even larger payloads. UDP communication set up between two Linux guest domains for comparison has at least 110 µs round-trip latency. With the additional overhead of a second Linux guest compared to our bare-metal domain in mind, it can be seen that our approach is faster by a factor of 2.3 × . Additionally, we discovered that UDP communication between applications on the same Linux guest using the loopback interface is even slower than our IDC approach and on par with Unix datagram sockets on the same system.

Adaptive Cruise Control Use Case

Figure 5.33 shows a block diagram of an ACC algorithm used in the automotive sector. Each block is realized as a bare-metal guest domain implementing the respective function. All bare-metal guests are connected to one central Linux domain running the router application of our service-oriented communication framework discussed in Section 5.4.2. This leads to a number of domains for this use case: the *Tracking*, *ACC*, *Collision Avoidance* (*CA*) and *Switching* domains representing blocks in the dataflow diagram, an unprivileged Linux guest running the router component and an additional bare-metal domain to create the test stimuli.

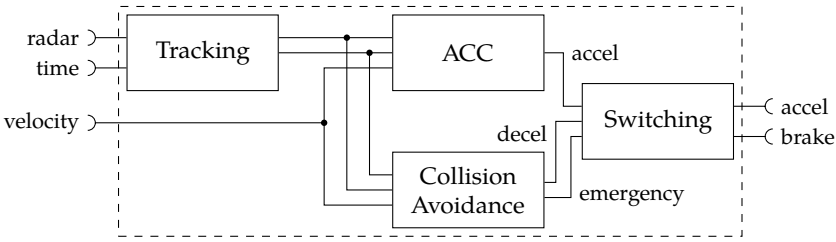


Figure 5.33: Block diagram of the Adaptive Cruise Control example use case

Application Type	1 step	1000 steps	ratio
Monolithic ACC application	6 µs	5 243 µs	∼873 ×
ACC split into four blocks	4 036 µs	4 313 µs	∼1.069 ×
Overhead due to splitting	∼672 ×	0.823 ×	

Table 5.7: Performance comparison of the ACC application, comparing the original single-application version with a variant split into four domains. Executing a single step is much slower in the split case, however continuous operation is even faster than the monolithic implementation due to implicit parallelism.

We compare the duration of a full run of the ACC algorithm using inter-domain communication as described with a monolithic version including all modules in a single bare-metal application. Table 5.7 shows the average duration of both a single iteration of the ACC algorithm and 1000 iterations of the same step to illustrate increased algorithmic complexity. Total run time is measured starting with the first call into the *tracking* service until final values are returned by the *switching* service.

We observe a very fast execution of a single step in the monolithic application due to the low complexity of the algorithm. When executing 1000 steps at once, the run time increases as expected, although slightly lower due to cache effects and measurement overhead. The single step for a split application is much slower, which can be explained by the round trip latency and scheduling impact. Each call to the four guest domains adds ∼30 µs of latency for communication, which does contribute to the total run time. The remaining overhead is caused by scheduling of the different guest domains, as no fixed schedule is used. The high standard deviation of 640 µs matches this explanation. However, when running 1000 iterations per step, we can even observe lower run time than with the monolithic application and only low addi-

tional overhead compared to a single iteration. The split implementation incidentally executes the ACC and CA services in parallel, provided that the hypervisor schedules both guests on different CPUs. This is a welcome side effect of the service-oriented communication framework. We can conclude that the feasibility of splitting an application across domains depends on the algorithmic complexity. The round-trip time is almost independent of the payload size and should not be significantly larger than the run time of the algorithm.

5.4.4 Conclusion and Outlook

In this section, we introduced a mechanism for communication between unprivileged guests on the Xen hypervisor. The design makes use of multiple standard technologies, such as VIRTIO, RPMsg and `xenbus`, allowing designers to reuse existing driver implementations in operating systems and runtime environments. During implementation, emphasis was placed on particularly high performance and although several software layers work hand-in-hand, the run time overhead in terms of latency is very low. Since SHM is used for data exchange, security benefits of the hypervisor through memory isolation remain, except for domains sharing a communication channel. We also introduced an example layer on top of the IDC mechanism, a simple service-oriented abstraction layer. It allows separating large applications into fine-grained guest domains providing individual services, profiting from communication between guests with very low latency. Additionally, we discussed how to implement networks on top of the peer-to-peer channel, allowing to monitor and enforce communication behavior. In comparison with conventional mechanisms, such as paravirtualized Ethernet, our approach allows communication at exceptionally low latency, albeit at the expense of flexibility with regard to interoperability with existing network protocols. This is a promising approach to implement extensively partitioned applications on hypervisor-based systems with mixed-critical workload. It has been demonstrated that especially large applications, which profit most from partitioning, can be enhanced using this IDC approach. This is similar to the ACC use case that was presented previously in this section.

The implementation of our IDC concept relies heavily on the Xen hypervisor. Future work will need to investigate how this approach can be implemented with other hypervisors. The main difference is the structure of the shared memory regions, which varies depending on the hypervisor. The remaining abstraction layers and associated drivers can then be reused. We demonstrated IDC on Linux as well as the

OpenAMP runtime library, however, there are many more runtime systems available. For example, we envision integration into RTOSs like FreeRTOS [132]. Integration is simplified by the fact that the OpenAMP library is already integrated into FreeRTOS. Other runtime environments, for example those used in the automotive sector, can require more work due to the lack of drivers for the technologies used by our approach.

5.5 Discussion & Summary

Embedded systems are being used in an increasingly diverse range of applications. As a result, the applications are becoming increasingly complex and error-prone, and the application may have to be updated during the system's service life or the intended use may even change. In the context of complex control and regulation tasks, a pure GPP no longer provides sufficient performance, but its flexibility compared to a pure hardware solution is still desired. This is where reconfigurable processors come into play: they combine both a GPP and an accelerator framework implemented as ASIC in an FLP, with an FPGA-based reconfigurable containers as the ALP.

On the basis of a reconfigurable processor prototype, the *i*-Core, we presented approaches to improve both performance and flexibility of embedded systems. We could show that the *i*-Core can deliver significant performance gains by offloading compute-intensive kernels to hardware, while maintaining software programmability through its closely-coupled design. There is, of course, a trade-off: on FPGAs, the performance of an ASIC implementation often cannot be matched. Reconfigurable processors blend the best of both worlds, pure software and pure hardware implementations, while retaining their advantages. Existing software development tools, such as sophisticated compilers, can still be used. Time-consuming hardware design by system integrators is reduced to only the computational kernels that require acceleration. We have not yet looked at approaches such as HLS, but they are promising to further simplify hardware implementation.

It is worth equipping the reconfigurable processor with additional features. A bus connection with a higher bandwidth is advantageous because, in contrast to GPPs, the reconfigurable fabric has the capacity to fully utilize it. This enabled the utilization of the *i*-Core as Near-Memory Accelerator and improve the performance of streaming tasks significantly. We also examined approximate computing as an optimization method, which turned out to be particularly suitable for run-time reconfigurable

processors. Algorithms that process sensor data, especially image and video processing, allow for certain inaccuracies in computations, which can be leveraged to improve performance, save hardware resources and reduce energy consumption. It not only yielded the anticipated performance enhancements; it also resulted in the use of fewer resources, allowing for a much more efficient allocation of reconfigurable containers. It can be expected that the use of approximate computing will continue to increase in the near future, especially in view of current developments in the field of machine learning. These algorithms possess an inherent degree of fault tolerance, and approximate computing techniques such as quantization and pruning are already actively researched [42].

Reconfigurable processors are particularly suited for control-flow focused tasks, which contain few computational kernels. For applications that are mainly dataflow-driven, we no longer benefit much from the capabilities of the GPP. Most of the processing time will be spent in the accelerator fabric, leaving the processor core mostly idle. Additionally, resource restrictions can be reached quickly when deploying highly parallel tasks. A single reconfigurable container is quite limited in hardware resources, as the design philosophy is to implement atomic, generic operations to be used by multiple different SIs. Specialized custom implementations quickly hit the resource limits of an RC and need to be split into multiple containers, which introduces communication overhead as discussed in Section 5.3.3. Additionally, the shared bus connecting the RCs can become a bottleneck. The *i*-Core concept has shown limitations when scaled up to tens of RCs, similar to the challenges encountered in MPSoCs with many cores. When hardware limits are reached quickly, the majority of the application will need to be executed in software. In Chapter 6, monolithic accelerators are discussed which are better suited for dataflow-driven applications.

In summary, it can be stated that reconfigurable processors have proven to be quite versatile and powerful in prototype use. However, there are still a few steps to go before it is ready for the market and use in embedded system products. We see a lot of potential for improvement in the area of tooling. Features such as auto-vectorization, common in today's compilers, could be transferred to hardware to simplify the extraction of kernels for acceleration. Finally, the area overhead must also be taken into account when moving to manufacturing of a reconfigurable processor, as the resulting costs must be covered. For each case, the hardware/software co-design process must be carried out individually to determine if a reconfigurable processor is appropriate.

Chapter 6

Dedicated Adaptive Accelerators for Dataflow-Driven Applications

An increasing number of workloads require specialized hardware due to their ever-growing computing requirements. In this work, we differentiate control-flow and dataflow dominated workloads. For the former, reconfigurable processors have been introduced in Chapter 5. The combination of a General Purpose Processor (GPP) with a closely-coupled accelerator framework using Field Programmable Gate Array (FPGA) technology renders them especially suitable for workloads that incorporate both control-flow and computationally intensive parts. Nevertheless, they are challenged by algorithms that rely largely on dataflow and high throughput. In this chapter, we are looking at dataflow-driven architectures as a platform for applications in Big Data and Machine Learning (ML). Focus is put on the reconfigurability of such architectures, allowing them to adapt to different workloads in the field.

An overview of related work from the scientific community has been presented in Section 3.2. With that background, we are introducing an adaptive accelerator platform that incorporates a Network-on-Chip (NoC) to connect individual accelerator tiles and facilitate scalability. It allows hosting different kinds of dataflow-driven accelerators and offer high-speed data interfaces to other accelerators, memories and off-chip devices. One of the most prominent data-intensive workloads today is ML, with modern Deep Neural Networks (DNNs) pushing the limits of even the most sophisticated High-Performance Computing (HPC) clusters. Consequently, we will then showcase an implementation of an accelerator for Convolutional Neural Networks (CNNs) based on a systolic array. As a third contribution, we investigate run-time monitoring of large-scale NoC systems. Using a tailored monitoring architecture, benchmarks of our platform hosting the CNN accelerator were conducted, which are presented to

conclude this chapter. These three contributions are ultimately combined, and the final design prototype is evaluated on an FPGA platform. Our evaluations cover the performance of individual accelerator tiles, data transfers throughout the platform, and end-to-end results from running CNN layers.

6.1 Concept for an Adaptive Accelerator Platform

Workloads characterized as dataflow workloads are often memory-bound and limited by memory bandwidth instead of compute power, as discussed in Section 4.2. This property makes the memory interface the most important performance indicator of computing systems for such workloads. Especially when scaling the systems up for increased performance through parallelism, it is vital to ensure that enough bandwidth is available to avoid starving of Processing Elements (PEs). Graphics Processing Units (GPUs) are often used for parallel computing due to their good programmability, advanced process nodes and high availability. Highly specialized architectures, both Tensor Processing Units (TPUs) and Neural Processing Units (NPUs) implemented as Application-Specific Integrated Circuit (ASIC) or backed by an FPGA, trade programmability for performance and power efficiency. Specialized hardware provides superior energy efficiency in terms of $\frac{\text{Ops}}{\text{W}}$ [133]. However, they lack flexibility because the designs are tailored specifically to certain algorithms and have difficulty adapting to other applications. An approach to overcome these limitations is the use of reconfigurable hardware to create hardware accelerators which can adapt to a given workload at run time. Changes in applications and algorithms then do not limit the scope of hardware-accelerated functions and the processor will stay up to date and energy efficient and updatable. We therefore apply the concept of hybrid adaptive hardware platforms, as introduced in Section 4.4, to dataflow architectures. Compute units, which are not a bottleneck for dataflow workloads, are implemented using reconfigurable hardware in the Adaptive Logic Partition (ALP). Maximum performance is required in the memory infrastructure, including memory interfaces, NoC, Direct Memory Access (DMA) engines, and thus implemented as Fixed Logic Partition (FLP).

We implement the adaptive accelerator platform as a reconfigurable tiled architecture. This ensures the scalability of the architecture and allows adaptation to conditions such as the available chip area and the number of memory interfaces. Additionally, we employ a uniform tile interface for high modularity and simple reconfigurability.

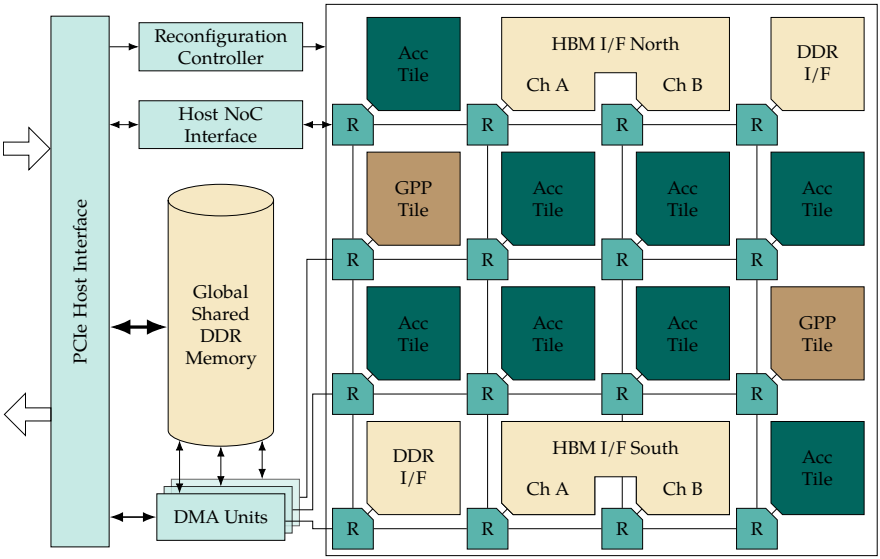


Figure 6.1: Small exemplary variant of the tiled accelerator platform with 16 tiles, two HBM and Dynamic Random Access Memory (DRAM) interfaces each. Reconfigurable accelerator tiles are highlighted in red (ALP), whereas all other components (NoC, memory and PCIe interfaces) are statically implemented as part of the FLP to achieve peak performance.

Run-time reconfiguration allows the platform to be repurposed for different use cases dynamically, adapting to different workloads as needed. The focus on modularity allows tiles to be designed in a generic way, independent of the number of instances in the platform. Users can load the number of accelerators needed for a specific workload and employ unused tiles for other workloads, enabling mixing of different tile types.

Adaptive Scalable Accelerator Platform

This section describes the base platform of our adaptive dataflow architecture. It is specifically focused on data-intensive workloads which are typically limited by memory bandwidth instead of compute, such as machine learning algorithms. We emphasize the use of reconfigurable hardware to enable changing the accelerator architecture for future needs. The following points were particularly considered in the design:

- *Scalability:* The base architecture is made scalable by the use of a planar meshed NoC, hosting memory tiles at its borders and compute tiles on the inside.
- *Adaptivity:* We envision the core memory interfaces and the NoC to be integrated as CMOS, while the compute tiles are implemented as FPGA logic and allow for run-time reconfiguration.
- *Memory interfacing:* Memory tiles can host DRAM interfaces to access a global shared memory, High-Bandwidth Memory (HBM) interfaces if more memory bandwidth is required (i.e. for larger configurations), or Static Random Access Memory (SRAM) interfaces to access low-latency memory.

Figure 6.1 shows the overall architecture concept. The accelerator core uses a regular meshed NoC, connecting all reconfigurable accelerator tiles (highlighted in red) and memory interfaces. It also features a PCI Express (PCIe) 4.0 host interface, which allows DMA access to both pure memory tiles and memory inside accelerator tiles. Memory tiles contain a memory interface and a multichannel NoC DMA unit and are typically placed on the mesh edges. In the small configuration, only four HBM channels are used, whereas typically at least 16 channels are available. The largest configuration has evenly distributed HBM interface tiles on all four edges, forming four HBM clusters.

Accelerator tiles represent the processing elements of the accelerator platform. While realized within the standard fabric in the FPGA prototype, we envision the accelerator tiles using embedded FPGA technology in an ASIC implementation. In the default configuration, each accelerator tile comprises 98 304 LUTs, 196 608 registers and 512 BRAM18 components. It provides enough resources for a set of reference accelerator implementations (Table 6.1), but can be freely configured at design time, if necessary. If the need for larger accelerators arises after design time, these can be realized with adjacent tiles, whereby communication via the NoC can be a bottleneck.

The NoC features wormhole switching, routers with configurable data link width and input buffer size [134]. By default, four virtual channels (VCs) are provided for improved handling of stalled communication in case of backpressure. Each VC can be configured as either best effort (BE) or guaranteed service (GS) mode. In BE mode, a channel is reserved for the duration of one packet only and the arbiter selects random VCs. GS mode in contrast allows reserving a channel indefinitely if guaranteed transmission latencies are required. Compared to specialized hardware accelerators featuring optimized NoC structures, a general purpose NoC is used to

	LUT	REG	BRAM
Available resources (default configuration)	98 304	196 608	512
Row-stationary CNN accelerator	84 265 (85.8 %)	56 513 (28.7 %)	512 (100 %)
NeoRV32 processor (8KiB IRAM/DRAM)	1 472 (1.5 %)	1 183 (0.6 %)	8 (1.6 %)
SUSAN edge detection pipeline	71 773 (73.0 %)	39 178 (19.9 %)	203 (39.6 %)

Table 6.1: Resources provided in an accelerator tile compared to a set of example implementations

support arbitrary dataflows. In addition, as all data transfers in the design essentially are memory-to-memory DMA transfers, using GPP-based tiles is also possible without the need to convert data streams into addressable memory. Bridging the gap to accelerators for control-flow workloads (Chapter 5), GPP tiles can also be implemented, using the available memory resources as instruction and data memory. DNN operations can also benefit from software execution directly on the accelerator. If a specific layer operation is not implemented in any hardware tile, it can then still be processed on the accelerator without being moved to the host.

Each tile is controlled by sending instructions to the NoC interface. Instructions can be sent by the host using the PCIe interface, but also from any other tile within the accelerator platform, e.g. GPP tiles. The instruction set comprises move-in, move-out, configuration and processing commands, which are stored in a command queue on reception. Tile control issues commands from this queue in-order and executes non-conflicting tasks out-of-order. On non-GPP tiles, move-in and move-out commands are processed by the DMA engines as part of our standard tile design. These DMA units can also be used in GPP tiles, however processing can also be done in software if resources are scarce. A sequence number can be set as a dependency to another command; this allows the user software to delay a move-out command until processing is finished.

Scalability considerations

The platform is specifically designed to be scalable for different use case scenarios. With a NoC as overall interconnect, the accelerator design can be easily scaled to

different sizes and tiles can be arranged freely. Considering sensor edge processing, a small accelerator with only a moderate amount of DRAM can suffice. For HPC applications, the NoC allows scaling up to 256 tiles and integrating both DRAM and HBM for maximum memory performance.

The main challenge of designing accelerators for data-intensive workloads is memory interfacing. To supply the highest possible memory bandwidth, the frequency and bandwidth of the different components are matched to each other. Using the prototyping platform Virtex UltraScale+ XCVU37P as a baseline, we selected the following clock configuration: The HBM stack operates at 400 MHz on a 256 bit Advanced eXtensible Interface (AXI) bus with a theoretical bandwidth of 12.8 GB/s for each channel [135]. While still close to the maximum frequency of 450 MHz, this allows for some more timing budget to drive the two NoC interfaces. The NoC links are 128 bit wide and operate at 200 MHz, using double-bandwidth links to the HBM interfaces. In addition to using the NoC clock for the NoC interface, each tile is also supplied a slower 50 MHz clock.

The Partitioned Global Address Space (PGAS) scheme allows accessing any part within the accelerator as mapped memory. Writing software for CPU tiles is greatly simplified, as an address map can be calculated at run-time based on hardware configuration information. However, the physical address width for the full accelerator platform needs to be higher when using PGAS to include all distributed (tile-local) and shared (DRAM/HBM) memories. While configurable on system level, we choose an address width of 48 bit for our prototype implementation.

6.1.1 Orchestration of Operations and Data Transfers

A general-purpose NoC is used in the Adaptive Scalable Accelerator Platform (ASAP). In contrast to designs with tailored NoC architectures as in *Eyeriss V2* [72], where distinct networks are present for each data type and the dataflow is fixed, our NoC allows data transfers between arbitrary tiles. While this approach introduces overhead in terms of addressing, routing, and packet decoding, it offers the essential flexibility to enable the implementation of future algorithms whose requirements have yet to be determined. The data format of a flit in our NoC implementation is shown in Figure 6.2. Each transfer starts with a head flit, which includes source and destination tile coordinates, and optional payload in the remaining bits. An arbitrary amount of data is transmitted with the control bit unset, containing 128 bit of payload data

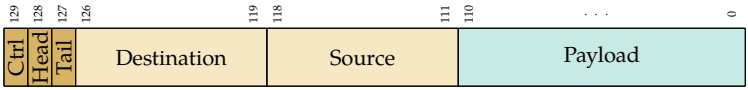


Figure 6.2: Layout of a flit in the NoC including a control bit to identify head and tail flits. The remainder of the 128 bit word can be used for payload data.

each. The transfer is finalized with a tail flit, which has the control and tail bits set, and may also contain trailing payload data if necessary.

Orchestration of large-scale manycore platforms based on NoCs is usually carried out by a distributed operating system running on each tile. Our platform however follows a coprocessor-like approach and relies on the host system for command generation and control. This has two advantages: on the one hand, the problem of synchronizing distributed systems can be avoided, and on the other hand, resources for processor and memory are saved in each tile. Managing the platform centrally from the host is unlikely to become a bottleneck as long as the atomic operations, like DMA transfers and accelerator iterations, are not too short. For inference of DNNs, this is generally the case, as execution times are in the range of 1 ms to 50 ms per inference.

Each tile is equipped with a tile controller, which handles DMA transfers and passes configuration and execution data to the user logic implemented in the tiles. A set of simple yet powerful instructions is defined, which are kept quite generic to make them suitable for a broad range of applications. In the following, the instruction set is explained, in accordance to their encoding on the network layer as shown in Figure 6.3.

Move-In A tile receives a move-in command with data to store to its local memory. It contains a target address, at which the standard DMA unit stores the received data. Alternatively, the data can be processed on-the-fly by user logic, if this benefits the application.

Move-Out When a tile receives a move-out command, it sends data from its local memory to a remote tile. This single-flit command includes source and destination for both the NoC coordinates and the memory address. Data sent to the remote tile is formatted as a move-in command, with the destination memory address taken from the received command. The destination can be another accelerator tile, a memory tile or the host interface, including the shared DRAM.

Configure The payload of the configure command is passed to user logic without modification. The tile controller just strips the control data from head and tail flit, leaving the remaining data for an application-defined use.

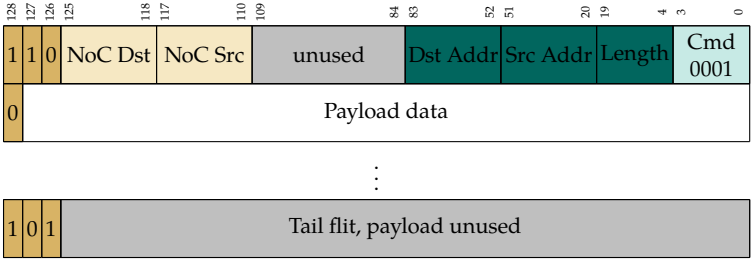
Execute The execute command is similar to the configure command, but is allowed to not contain any payload data. It signals the user logic to begin or continue processing of data.

Notify The notify command can be sent to the host interface, emitting an interrupt in the host system. The head flit payload carries an identification field passed to the interrupt logic. It is automatically generated after an execute command has finished.

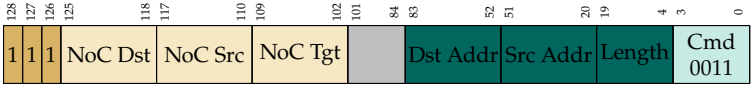
Although all memory resources in the architecture are mapped to a PGAS, the addresses in move-in and move-out commands refer to local addresses only. The address bits representing the tile are encoded in the NoC coordinates, avoiding overhead. Memory transfers are restricted to the PGAS within the platform. Transfers between accelerator platform and host memory can only be handled by the host interface, preferably targeting the shared DRAM. This reduces the hardware overhead in each tile's DMA units to improve scalability, and shifts the complexity of addressing host memory to the host interface. The command queue in the tile controller allows the host to enqueue up to eight commands in advance. Processing status can be monitored using interrupts generated by notify commands, which are also queued for processing at the host interface.

The basic tile infrastructure consists of tile controller, network adapter, and DMA units (Figure 6.4). NoC traffic is split into control and data flits, which get passed either to the tile controller or to the DMA units. User logic and local memory is implemented in the ALP, the remaining components are placed in the FLP. Three different clock domains can be used to support variable clock configurations for the user logic. This allows the NoC to run at its optimal clock frequency, independent of user logic constraints. Tile-local memory can either be connected directly to the DMA memory interface as shown, or user-specific handling of the data ports is implemented.

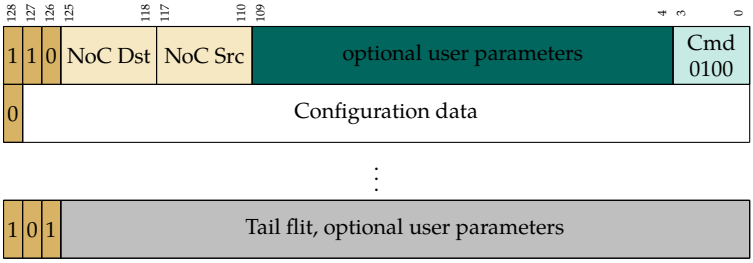
This simple communication protocol offers a high degree of freedom for implementing a wide variety of accelerator modules. Its applicability was demonstrated using the CNN accelerator module and shown simulatively for processor-based tiles. It is based on the fact that most modern workloads are primarily data-focused and require hardly



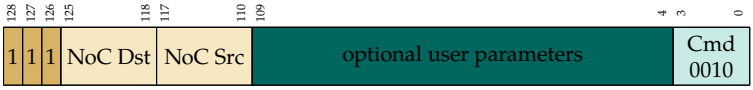
(a) Flit structure of the move-in instruction.



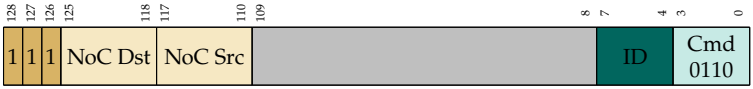
(b) Flit structure of the move-out instruction.



(c) Flit structure of the configuration instruction.



(d) Flit structure of the execute instruction.



(e) Flit structure of the notify instruction.

Figure 6.3: Flit data structure for the commands understood by the tile controller.

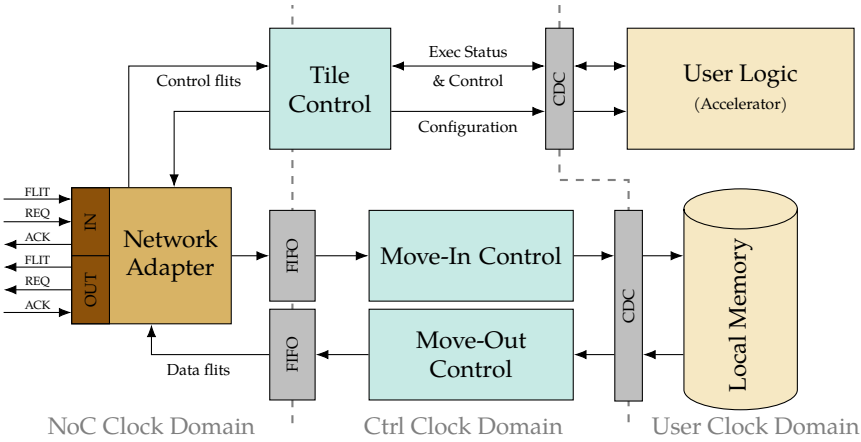


Figure 6.4: Tile controller and auxiliary components implementing the communication protocol.

any synchronization primitives, as shown in Chapter 4. A simple instruction set is therefore sufficient for the target workloads.

6.1.2 Host Interface and Software Libraries

When laying out the memory interfacing, focus was put on an add-on card style accelerator. Primary data I/O is done using a PCIe interface to the host. While sharing host memory as global memory may be possible, local accelerator memory (DRAM and HBM) is used to store intermediate results without the impact of a host interface bottleneck. DRAM is mapped to a PCIe Base Address Register (BAR) and can be accessed directly from the host. The DMA engines on the accelerator can access host memory, local DRAM, and both HBM and tile-local scratchpad memory. This enables multiple ways to exchange data between host and accelerator, while software on the host can directly allocate memory within the accelerator's DRAM when primarily used by the accelerator.

A driver library on the host provides software interfaces for initialization, configuration and control of the adaptive accelerator platform. It is split into general functions and application-specific functions. General functions include initialization of the base platform itself, sending and receiving NoC messages, triggering DMA transfers and

uploading bitstreams into the individual tiles. The application-specific libraries implement interfaces to control functionality in the accelerator tiles. The reconfiguration controller can be queried to identify the currently configured bitstream within each accelerator tile. This information is used by the application-specific library for the CNN accelerator to map convolution operations to the accelerator platform, taking the number of available CNN tiles, the PE array size, buffer sizes and input parameters into account.

6.1.3 Run-time Reconfiguration

The central reconfiguration controller supports loading configuration data into tiles on demand. It is triggered from the host system, which is responsible for ensuring that the tiles to reconfigure are unused. In the full-FPGA prototype, the reconfiguration controller uses AMD's ICAPE2 primitive. Thus, the bitstream encodes the target area of the design to be reconfigured. In a mixed ASIC and FPGA implementation, we envision the reconfiguration controller to be connected to each individual embedded FPGA, i.e. each ALP. The target tile for reconfiguration is then selected using control registers in the controller.

Reconfiguration performance is limited by the configuration primitive called the ICAP in former FPGA architectures by AMD. Due to its fixed width of 32 bit, it achieves a maximum bandwidth of 3.2 Gbit/s and 6.4 Gbit/s for 7 Series and UltraScale/UltraScale+ devices, respectively. There have been approaches in research that can achieve significantly higher data rates through sophisticated design of the ICAP control, which for Virtex-5 exceed five times the specified bandwidth [136]. However, it is not yet clear if this approach is applicable to recent architectures and if its limitations allow partial reconfiguration. Similarly, the performance of the CFI, which the ICAP has been rebranded to in AMD's latest Versal architecture, is yet to be assessed. We therefore operate the configuration port at its specified speed, resulting in a reconfiguration duration of 8.3 ms for an uncompressed tile bitstream.

6.2 Acceleration of Convolutional Neural Networks Using a Row-Stationary Dataflow

Based on the original publication *Enhanced Accelerator Design for Efficient CNN Processing with Improved Row-Stationary Dataflow* at GLSVLSI 2024. [Les+24]

The use of Deep Neural Networks (DNNs) is expanding into a multitude of domains. Recently, they are also being used in embedded systems for tasks like image or sensor data processing. Advanced architectures are required to sustain the high memory and processing demands of DNNs that include computationally intensive convolutional layers and frequent off-chip memory accesses. In the embedded field, neither GPPs nor GPUs are equipped to handle these requirements due to their high power consumption [56]. Specialized hardware accelerators, on the other hand, are tailored to the intended use case and enable particularly high energy efficiency. An overview of DNN accelerators from academia and industry is given in Section 3.3.

Our adaptive accelerator platform is particularly suitable for DNN processing, since it is primarily designed for applications with high memory bandwidth requirements. It exploits the fact that most DNNs are memory-bound, and reduced performance in the computational elements has negligible impact on performance. Therefore, an accelerator tile for CNN acceleration, which can be used for image segmentation and feature detection, is to be implemented. However, the majority of existing accelerator designs are either proprietary with undisclosed implementation details or lack flexibility in their implementation. One of the primary contributions in this section is an accelerator design that utilizes the Row-Stationary (RS) dataflow described as part of the closed-source *Eyeriss* accelerator [53]. We provide comprehensive implementation details alongside the full Hardware Description Language (HDL) sources and offer a practical approach to implement a high-performance scalable CNN accelerator. The accelerator module is both usable as a tile on the adaptive accelerator platform (Section 6.1) and in a stand-alone mode as a peripheral on an AXI bus. Further, we introduce an enhanced RS dataflow optimized for convolution tasks with a high number of channels. The section concludes with a detailed evaluation of design parameters and the demonstration of an FPGA prototype.

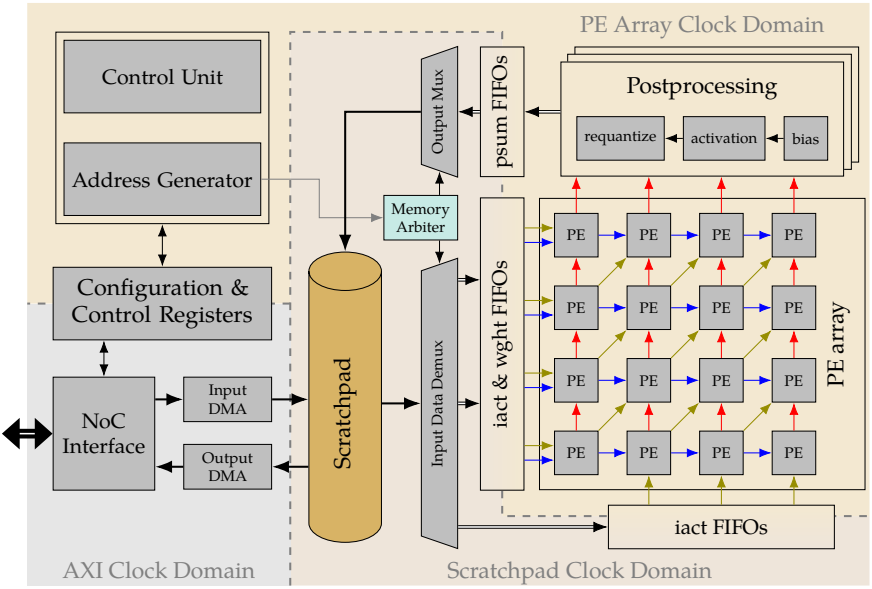


Figure 6.5: Architecture overview of a 4 × 4 CNN accelerator implementing a row-stationary dataflow. In the standard configuration, input activations move diagonally, weights to the right and partial sums upwards within the systolic array.

6.2.1 CNN Accelerator Design with Row-Stationary Dataflow

This section describes the design of a CNN accelerator in the form of a systolic array. It is dedicated to performing quantized convolution operations and includes functional units to apply floating-point scaling, offset biasing and activation functions. To maximize data re-use, the CNN accelerator implements a RS dataflow. This dataflow minimizes memory accesses, thereby reducing energy consumption and enhancing efficiency. In addition to the classic RS dataflow, our PE design implements an alternative dataflow that is more efficient for high input channel sizes. The accelerator features a NoC interface for use in tiled architectures and an AXI bus interface for integration in FPGA-enabled System-on-Chips (SoCs). While the initial design is prototyped on FPGA platforms, care was taken to avoid FPGA-specific Intellectual Property (IP) in favor of an ASIC implementation.

Variable	Description
H, W	Height & width of the input activations
R, S	Height & width of the filter kernels
P, Q	Height & width of the output matrix
C	Number of input channels
M	Number of output channels
M0	Number of output channels processed in parallel
C0	Number of channels that fit into the PE line buffers
Q0	Segment of the output width fitting into a line buffer
P0	Segment of the output height fitting the PE array
M1, P1, Q1, C1	Counterparts to the parallel loop ranges which need to be processed sequentially

Table 6.2: Parameters of a convolution problem and loop ranges for parallel and sequential processing.

Figure 6.5 shows an overview of the architecture of the CNN accelerator. It consists of a control block, scratchpad memory and the configurable PE array, shown with only 16 PEs for clarity. The accelerator is designed to operate stand-alone as long as possible in order to reduce communication effort with the host. The control unit implements all convolution loops in hardware so that a complete image with up to 1 024 channels can be processed with just one command. This is also limited by the scratchpad size, which is set to 1 MiB in our implementation by default and needs to hold all input and output data of one command. PEs are arranged as a systolic array, where input activations are forwarded diagonally from bottom left to top right, weights left-to-right and (partial) sums bottom-to-top. Each PE includes a multiplexer to allow using the vertical data path for forwarding of input activations as well. The width X and height Y of the PE array can be freely configured to meet workload requirements and technical constraints (available space, resources, etc.). The size of the line buffers within the PEs can also be adjusted. The memory within the PEs is supplemented by the scratchpad memory next to the array, which stores input activations, weights and (partial) sums of convolutions. For most modern CNNs, the scratchpad size does not fit a full convolutional layer and tiling is used to subdivide the problem. The size of the scratchpad memory can be adapted according to the intended use and the available resources, so that a balance of logic and memory requirements is achieved.

Configurable Clock Domain Crossing (CDC) is implemented within the PE array buffers and between the outbound interface and the scratchpad memory. Data is exchanged with other tiles, which can be both other processing tiles or memory tiles, using the input/output DMA engines. This isolates interface, scratchpad and PE clocks, forming three independent clock domains to achieve maximum performance in each region.

Algorithm 3 Loop nest for the SRS dataflow.

Input: input activations I[H,W,C], filter weights W[R,S,C,M]
Output: output feature map O[P,Q,M]

```
for m1,p1,q1 in range M1,P1,Q1 do
    parfor m0 in range M0                                  ▷ mapped to PE rows
        parfor p0 in range P0                              ▷ mapped to PE columns
            for c1,q0 in range C1,Q0 do
                parfor r in range R                        ▷ mapped to PE rows
                    for s,c0 in range S,C0 do
                        p = p1 * P0 + p0
                        q = q1 * Q0 + q0
                        c = c1 * C0 + c0
                        m = m1 * M0 + m0
                        w = q + s
                        h = p + r
                        O[p,q,m] += W[r,s,c,m] * I[h,w,c]
                    end for
                end parfor
            end for
        end parfor
    end parfor
end for
```

Algorithm 3 describes the loop nest for a classic RS dataflow, which is fully implemented in hardware within the control unit. Refer to Table 6.2 for parameter details. This dataflow, which we call the Spatial Row-Stationary (SRS) dataflow, maps columns of the output P spatially onto PE columns and both filter height R and output channels M onto PE rows. Spatially parallelized loops are identified by the keyword **parfor**, while the remaining **for** loops are processed temporally. Each PE buffers partial sums of an input row section W and the corresponding filter weights $S * C0$ in a line buffer. Consequently, one iteration of the loop nest processes a section of the input activation image which fits into the input activation line buffer within the PEs and has to be repeated to process the full input image. Convolution param-

eters P, Q, C, M are split into inner loop dimensions $P0, Q0, C0, M0$ which fit the line buffers and array size, and $P1, Q1, C1, M1$ in the outer loop which have to be processed sequentially or on another accelerator instance. PE rows are first allocated to process the current filter in parallel (height of the filter R). Afterward, further filters are mapped to the remaining rows if possible (number of output channels processed in parallel $M0$).

We propose a novel dataflow for RS CNN accelerators, which improves PE utilization in most cases. It especially suits hidden layers of CNNs, which often have a high number of output channels M. Algorithm 4 shows the altered loop nest for comparison.

Algorithm 4 Loop nest for the TRS dataflow

Input: input activations I[H,W,C], filter weights W[R,S,C,M]
Output: output feature map O[P,Q,M]

```
for m1,p1,q1 in range M1,P1,Q1 do
    parfor m0 in range M0                                  ▷ mapped to PE rows
        for r,c1,q0 in range R,C1,Q0 do
            parfor p0 in range P0                          ▷ mapped to PE columns
                for s,c0 in range S,C0 do
                    p = p1 * P0 + p0
                    q = q1 * Q0 + q0
                    c = c1 * C0 + c0
                    m = m1 * M0 + m0
                    w = q + s
                    h = p + r
                    O[p, q, m] += W[r, s, c, m] * I[h, w, c]
                end for
            end parfor
        end for
    end parfor
end for
```

Compared to the original SRS dataflow, output channels M are mapped spatially to PE rows and output columns P are mapped to PE columns. Filter weights are instead processed in sequence temporally, which requires the PE line buffers to fit $R \times S$ instead of only S weights. This overhead is negligible for small filters like 3×3, especially when existing memory primitives like Block RAM (BRAM) are used. Figure 6.6 shows a comparison of the SRS and Temporal Row-Stationary (TRS) dataflows. Note that the case shown has been constructed for the sake of clarity, but

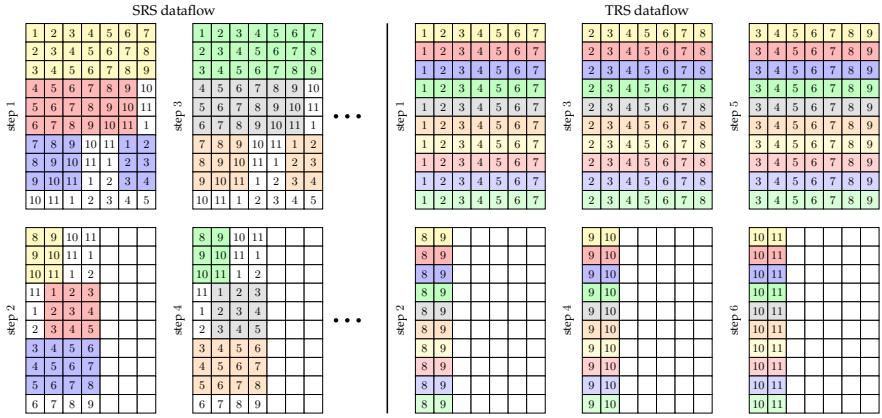


Figure 6.6: Exemplary mapping of a convolution with input dimensions of $11xW$, kernel size $3xS$ and $C = 10$ on an accelerator with 10×7 PEs. Each color is one kernel, each square represents one PE and the number inside is the input row being processed in the current step. Empty squares are idle PEs, uncolored squares are only forwarding data. The left part shows the first four steps in which six kernels are processed using the SRS dataflow, the remaining channels are processed likewise. The right part shows six steps in which all ten kernels are processed using the TRS dataflow.

the principle can be similarly applied to larger dimensions of input activation and different accelerator sizes. Each color denotes a set of weights for one filter that is processed on the respective PE. The numbers in the squares correspond to the input rows being processed in the PE. On the left, four processing steps of the SRS dataflow are shown that process a set of six kernels, which maps to processing three kernels every two iterations. As the PE array size is not a multiple of the kernel height R, the bottom row is not used. Due to input activations traveling diagonally through the array, six PEs do not contribute to the operation as they apply the filter to rows $10, 11, 1$ and $11, 2, 2$. Overall, 42 % of PEs are idle. Full utilization of the array is only achieved when the criteria

$$Y \bmod R = 0$$
$$H \bmod X = 0$$

are met. In comparison, the right side of Figure 6.6 shows six processing steps of the TRS dataflow, in which a set of 10 kernels is being processed. Steps 1, 3 and 5 show full utilization of the array, while steps 2, 4 and 6 have 50 idle PEs each, resulting in 36 % of idle PEs. For full utilization, the criteria are:

$$Y \bmod M = 0$$
$$H \bmod X = 0$$

The criterion for Y is easier to fulfill for the TRS dataflow in most cases. The number of kernels processed in parallel M can be set to Y if enough channels are being processed. In the SRS case, however, typical filter heights 1, 3, 5 and 7 have a common multiple of 105, so that very large PE arrays would be required for optimum mapping. Note that for full utilization, the input activation height H shall be a multiple of the accelerator width X. In case of a mismatch, the last step of each iteration has unused PE array columns which is negligible for high input channel counts. The design allows switching between the two dataflow types during run time, using the vertical data path for input activations for the TRS case. Based on the given criteria, the mapping tool can decide which dataflow is better suited and program the accelerator accordingly.

Custom Memory Design for efficient data transposition

The RS dataflow requires input data to be formatted differently than output data. Figure 6.7 visualizes different representations of image input/output data for convolution algorithms. In the *HWC* layout, the first pixel of each channel is consecutively stored to memory first. These elements are then repeated for the image width, and the resulting structure representing an image row with all channels is then followed by the remaining rows. This concept is similar to common image formats like *RGB24*, where one byte per red, green and blue channels is stored next to each other per pixel. In DNN toolkits however, the *CHW* data format is more common. For each channel, an image of $H \times W$ pixels is stored consecutively in memory, which results in placing the first channel's image first into the linear address space, following the remaining channels.

On the input, the RS dataflow expects data in the *HWC* layout. More specifically, $C0$ channels of each input pixel need to be loaded consecutively into the respective row of the PE array. Output data is retrieved in *CHW* layout, which results in a full $P \times Q$ image for the first output channel being written first, before the next output channel

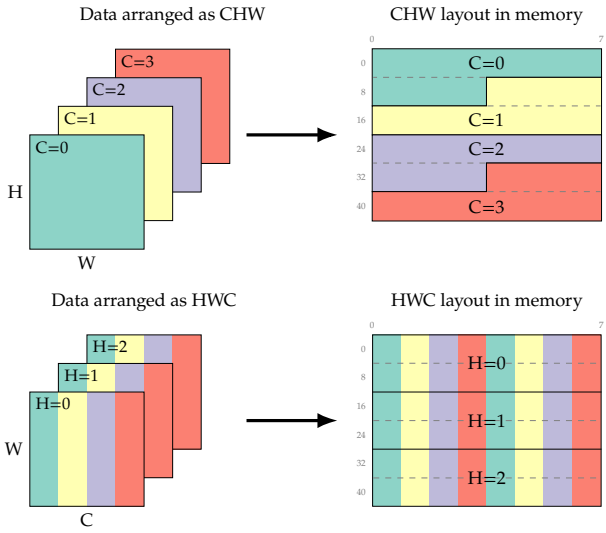


Figure 6.7: Exemplary visualization of *CHW* and *HWC* data layouts. Each color represents a different channel, the parameters are $C, H, W = 4, 3, 4$, resulting in 12 pixels per channel.

is retrieved. In order to process multiple convolutional layers without conversion, the same data layout needs to be used. If the memory allows single-byte access, a sophisticated address generator can load individual pixels from an *CHW* image to create an *HWC* data stream. This has been tested in our initial implementation, but leads the scratchpad requiring a clock frequency of 10 × the PE array clock to avoid stalling the array unnecessarily. To ease the clocking requirements on the scratchpad, multi-word memory access was implemented.

Loading multiple bytes of input data of a *CHW* image at once results in a sequence of pixels from a single channel. This is not suitable for an RS accelerator, which expects tuples of $C0$ single pixels from different channels at its input. We developed a custom memory architecture to support loading such tuples directly from memory without reordering overhead. Figure 6.8 shows the custom memory architecture for a memory with 64 bit words, consisting of 8×8 bit elements. The memory is organized in columns, which consist of dual-port Random Access Memories (RAMs) addressable in 64 bit words each. For simplicity, each column of the memory in Figure 6.8 holds only eight words, resulting in column addresses from 0 to 7. In our

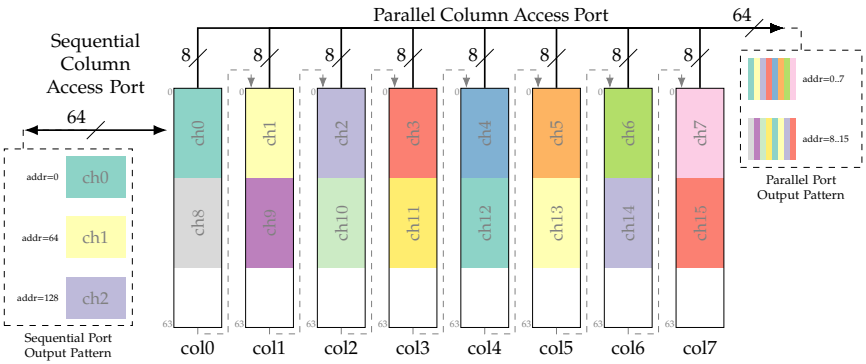


Figure 6.8: Scratchpad memory architecture supporting linear and column-wise parallel access modes. Data is effectively transposed when read and written through different ports.

standard design, each column holds 16 K words, resulting in a column size of 128 KiB and a total size of 1 MiB. To the outside, two interfaces are provided, each using one of the two ports of each column:

Sequential access port On this port, memory columns are concatenated linearly, so that the uppermost address bits identify the column. The lower part of the address is linearly mapped to the columns.

Parallel access port All memory columns are accessed at once on the parallel port. The lowermost bits of the address identify the offset of the 8 bit element within the 64 bit word to be loaded.

This scheme allows both the host and the output of the accelerator to write *CHW* data using the sequential access port. When loading input activations or kernel data, the parallel access port is being used to retrieve the data in *HWC* format. Care has to be taken to store *CHW* data with an offset equal to the column size per channel, such that different channels are located at the same offset in each column. The design is fully generic, making the number of columns, the address width, and the element size customizable. Observe that the memory can only work with multiples of 8 channels. If, for example, 12 channels are processed, two full words must still be loaded, whereby the second is filled with zero bytes.

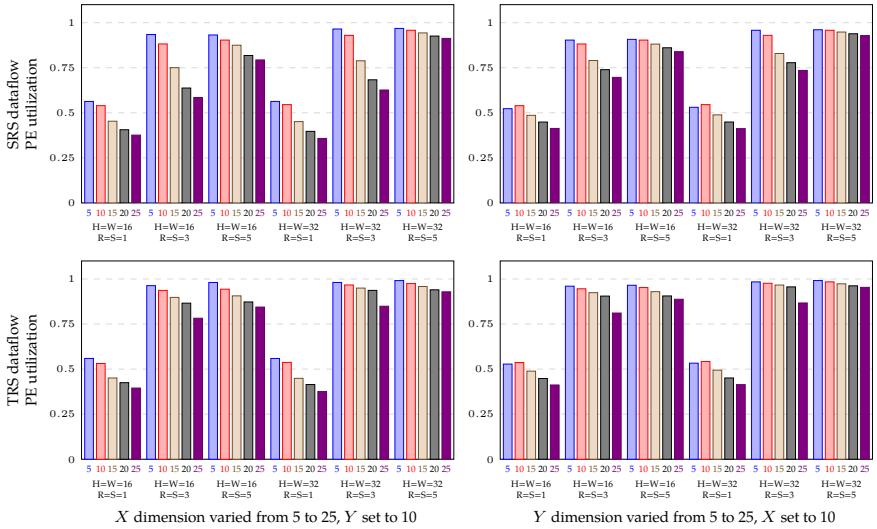


Figure 6.9: Utilization of the PEs relative to the total number of cycles for varied height and width of the PE array and different convolution workloads. Original SRS dataflow is shown in the upper graphs, our proposed TRS dataflow in the lower graphs.

The custom scratchpad memory design increases memory throughput by allowing to load multiple input elements at once. At the same time, the clock frequency can be decreased to a value only 1.5 × the PE array clock while maintaining enough throughput to avoid stalling.

6.2.2 Evaluation

Experimental Setup

In this section, the CNN accelerator is evaluated as a standalone design, using the AXI bus interface. Physical implementation results have been obtained using a Xilinx ZCU104 evaluation board with the Zynq UltraScale+ XCZU7EV FPGA. Xilinx Vivado 2024.2 is used for synthesis, implementation and bitstream generation. The prototype design of the accelerator uses a 10 × 7 array and 1 MiB of scratchpad memory.

Driver library and benchmarking software are running on the quad-core Cortex-A53 processor on top of Linux 5.15.19.

Simulation results have been obtained using automated cycle-accurate testing of the RTL model with Siemens Questa Prime version 2022.4. The simulation environment is automated using a Python script, which allows running permutations of parameter lists for most design variables in parallel. For each simulation run, random input activation and weights are generated and mapped to the accelerator using a greedy scheduling algorithm.

Evaluation of design parameters

We performed design space exploration for different scenarios to obtain suitable parameters for the PE array size, size of the line buffers within each PE and input/output FIFO depths. Figure 6.9 shows the utilization for varying PE array sizes and different convolution problems. The overall utilization per PE is given relative to the total number of cycles for the given parameters for both SRS and TRS dataflows. It can be observed that for small filter sizes of $R = S = 1$, the accelerator performance is limited by memory bandwidth as no input activation data can be reused. The scratchpad clock frequency is fixed at 1.5 × the PE array clock throughout all simulations and uses a single interface to the input activation arbiter. As expected, larger PE arrays, both in X and Y dimension, show degraded performance due to the memory bandwidth bottleneck. A single input activation arbiter can efficiently distribute data to the number of input FIFOs matching the scratchpad/PE clock frequency ratio. For a high number of input FIFOs, being $X + Y - 1$ for SRS and X for the TRS dataflow respectively, either the memory word size or the scratchpad clock frequency can be increased to achieve a higher memory bandwidth.

The impact of limited scratchpad bandwidth is reduced for the TRS dataflow. It shows a reduced performance penalty for large array sizes, as only the bottom edge of input activation FIFOs is being used. As expected, larger kernel sizes result in higher utilization of the PEs due to increased data re-use. While 1 × 1 convolutions perform similarly for both dataflows, the TRS dataflow can maintain higher utilization for 3 × 3 filters.

The size of the input FIFOs at the western and southern edge is crucial for performance, as the full array needs to be stalled if any FIFO runs empty. Figure 6.10 shows the result of an input FIFO size parameter sweep from 4 to 22 words on a 10 × 7 array. Sizes lower than 10 words result in a great increase in processing time, while more

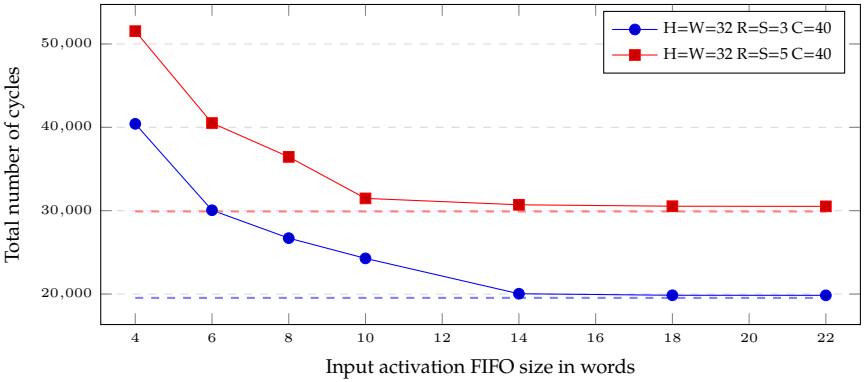


Figure 6.10: Total number of cycles for processing a 32 × 32 image, 40 channels, with 3 × 3 and 5 × 5 filters on a 10 × 7 PE array in TRS mode. The dashed line is the minimum baseline of cycles required for the full workload if no stalling occurs.

than 16 words hardly reduce the remaining overhead with respect to the theoretical optimum. Output buffers perform best at similar sizes, however their impact on the total processing time is limited due to their brief usage after slice processing.

The design benefits from a high scratchpad clock, as input activation data is distributed to the array input FIFOs in a round-robin fashion. A single memory port is used to allow using a monolithic SRAM block as scratchpad within each tile. However, the scratchpad could be split into smaller chunks with independent ports if FPGA BRAM is used, reducing the necessary clock frequency through parallelization. In any case, the maximum scratchpad interface frequency depends on the FPGA technology. We achieved a scratchpad clock of 250 MHz in our design, with potential for further improvement. This results in a clock ratio of 5:1 by running the PE array at 50 MHz. While this ratio is not ideal yet, further lowering the PE array frequency would still degrade performance.

Carefully balancing the trade-off between scratchpad bandwidth, number of arbiters and input buffer size is essential to maximize performance for the selected array size and clock frequencies. When using a single port scratchpad memory, an array size of 10 × 7 is a good compromise with sensible clock ratio, small input buffers are sufficient and common kernels up to 7 × 7 do not need tiling.

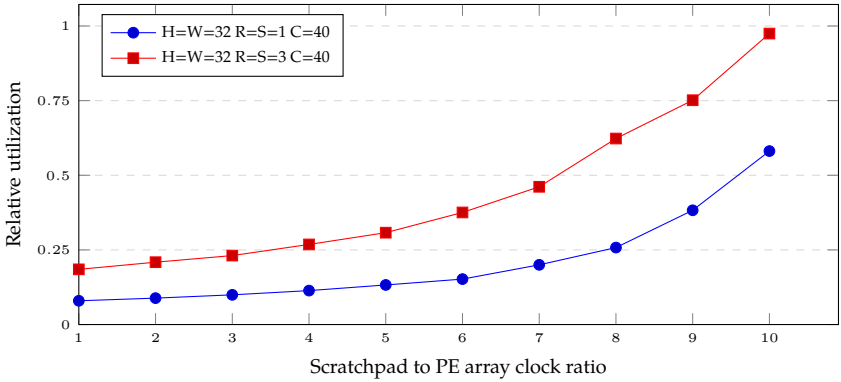


Figure 6.11: Relative utilization of a 10 × 7 PE array in TRS mode for different clock ratios with a single scratchpad arbiter.

Evaluation of SRS & TRS dataflow

We compare our proposed TRS dataflow with the original RS dataflow by Chen et al. [53]. Building a versatile CNN accelerator requires support for different input activation and kernel sizes. ResNet-50 [39], GoogLeNet [137] and MobileNetV3 [138] are chosen as a representative set of CNN models. All these models show the usual procedure for CNNs: the high resolution at the input is reduced quite quickly and fanned out over many channels. Further processing is done with small images and a high number of channels.

Table 6.3 shows utilization of all PEs for processing of a full higher-dimensional convolutional layer. The results are obtained using the mapping tool for our design, set to use the SRS and TRS dataflow, respectively. As the utilization of a mapping depends heavily on the accelerator shape, we analyzed both our default 10 × 7 size and a larger 14 × 12 array. It can be seen that the TRS dataflow is at least on par with SRS and exceeds it for most inputs.

We measured an end-to-end throughput of 4.012 GOp/s with a 10 × 7 array at 100 MHz on a workload with $R = S = 3$ and $C = M = 10$ using the TRS dataflow. Taking into account that no workload pipelining was used and time for preload and write-back of data is included, this already comes close to the theoretical maximum of 7 GOp/s for this configuration.

	H/W	R/S	C	SRS util	TRS util	Workload
10 × 7	28	3	256	82.92 %	**90.27 %**	ResNet-50 3rd last layer
	14	3	1 024	76.99 %	**85.21 %**	ResNet-50 2nd last layer
	7	3	512	64.16 %	**70.71 %**	ResNet-50 last layer
	14	1	528	**99.62 %**	98.87 %	GoogLeNet 3rd last layer
	7	1	832	**99.05 %**	**99.05 %**	GoogLeNet last layer
	28	5	120	**85.71 %**	**85.71 %**	MobileNetV3 IR layer 7
	14	3	240	77.14 %	**85.71 %**	MobileNetV3 IR layer 8
	7	5	960	**42.86 %**	**42.86 %**	MobileNetV3 IR layer 15
14 × 12	28	3	256	61.90 %	**69.01 %**	ResNet-50 3rd last layer
	14	3	1 024	42.86 %	**98.84 %**	ResNet-50 2nd last layer
	7	3	512	35.71 %	**42.73 %**	ResNet-50 last layer
	14	1	528	**57.89 %**	**57.89 %**	GoogLeNet 3rd last layer
	7	1	832	57.78 %	**58.49 %**	GoogLeNet last layer
	28	5	120	47.62 %	**92.06 %**	MobileNetV3 IR layer 7
	14	3	240	42.86 %	**95.24 %**	MobileNetV3 IR layer 8
	7	5	960	17.86 %	**25.67 %**	MobileNetV3 IR layer 15

Table 6.3: PE utilization for SRS & TRS dataflows for array sizes 10 × 7 and 14 × 12. Layers are chosen from state-of-the-art neural networks (ResNet-50, GoogLeNet, MobileNetV3).

End-to-End Performance

The performance of 2D convolutions of varying sizes, oriented within the common parameter ranges shown in Table 6.3, is measured to obtain a realistic performance indicator. Each benchmark run comprises a copy of random input data to the scratchpad memory, the computation carried out in hardware, and the subsequent transfer of results back to the main memory. Concurrently, a software implementation of the convolution algorithm is executed on a single core using data stored in DRAM. On the one hand, validation is carried out by comparing hardware and software. On the other hand, the speed-up compared to GPP execution is measured, including the time for copies between DRAM and scratchpad memory. Figure 6.12 visualizes the overall throughput in hardware for a selection of the most relevant results. In Table 6.4, the

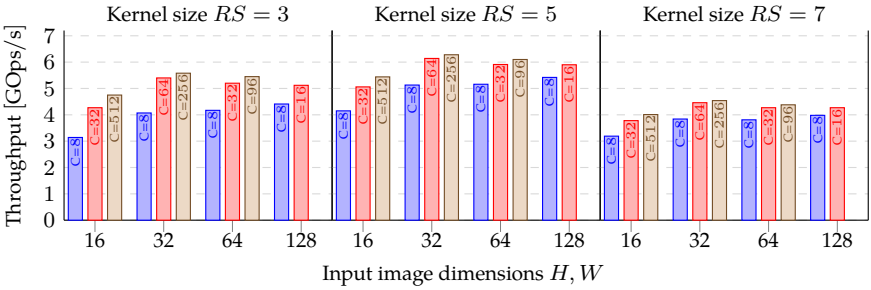


Figure 6.12: Overall throughput for various convolutions on a 10×7 PE array.

results are listed in detail including a comparison between hardware and software. All tests were run using the SRS dataflow with padding enabled, applying a ReLU activation function and final requantization to 8 bit.

It is apparent that the hardware performs optimally for medium-size input images and a high number of input channels. The highest performance is achieved for a 32×32 image, 5×5 kernels and 256 channels, with 6.28 GOp/s. This configuration constitutes the highest number of Multiply-Accumulates (MACs) to be executed, which explains the peak performance numbers. However, it can be seen that related configurations with a larger image size and fewer channels, or smaller images with increased channel count, also yield high performance. A notable influence on the results is the duration for copying data in and out of the scratchpad memory. In many cases, more time is spent on copying data than the actual processing in hardware. This has two reasons: On the one hand, single convolutions are tested. When a full neural network is run, data does not need to be copied to DRAM between individual layers. On the other hand, the copy mechanism is currently implemented in software due to the single-tile version not having DMA units. When using a DMA unit instead, the copy duration will decrease by a factor of about $10\times$. In summary, the design is very efficient in operation, since the achieved performance is close to the theoretical maximum of 7 GOp/s at 100 MHz.

Resource usage

Table 6.5 lists the resource usage of a 10×7 and a 14×12 accelerator on a Xilinx Zynq UltraScale+ FPGA, both for the full design and for the individual components. This implementation uses the SRS dataflow, but the TRS variant only differs slightly for

HxW	RxS	i-ch	o-ch	cpu [µs]	acc [µs]	copy [µs]	speed-up	GOp/s
16	3	8	3	1 280	17.67	84.74	12.49	3.13
16	5	8	2	1 984	24.79	58.66	23.77	4.13
16	7	8	1	1 765	31.34	33.98	27.02	3.20
16	3	32	3	4 982	51.76	103.81	32.02	4.27
16	5	32	2	8 016	81.01	80.26	49.70	5.06
16	7	32	1	7 024	106.11	52.67	44.24	3.78
16	3	512	3	80 179	744.74	436.29	67.89	4.75
16	5	512	2	126 674	1 204.15	439.50	77.07	5.44
16	7	512	1	112 609	1 600.19	418.95	55.77	4.01
32	3	8	3	5 265	54.41	328.48	13.75	4.07
32	5	8	2	8 258	79.89	226.32	26.97	5.13
32	7	8	1	7 472	104.43	125.28	32.53	3.84
32	3	64	3	43 385	327.45	476.02	54.00	5.40
32	5	64	2	68 169	533.45	376.13	74.95	6.14
32	7	64	1	61 082	719.76	273.97	61.47	4.46
32	3	256	3	176 767	1 269.54	979.21	78.61	5.58
32	5	256	2	275 628	2 088.66	891.32	92.49	6.28
32	7	256	1	246 496	2 829.38	781.43	68.27	4.54
64	3	8	3	21 769	211.99	1 302.61	14.37	4.17
64	5	8	2	34 073	317.42	895.69	28.09	5.16
64	7	8	1	31 188	421.10	490.38	34.22	3.81
64	3	32	3	94 858	680.97	1 554.34	42.44	5.20
64	5	32	2	141 880	1 109.36	1 150.63	62.78	5.91
64	7	32	1	126 850	1 505.40	736.26	56.59	4.27
64	3	96	3	283 770	1 949.08	2 196.83	68.45	5.45
64	5	96	2	425 570	3 221.36	1 802.41	84.71	6.10
64	7	96	1	379 841	4 396.63	1 396.94	65.56	4.38
128	3	8	3	95 030	802.01	5 204.30	15.82	4.41
128	5	8	2	142 858	1 209.10	3 580.02	29.83	5.42
128	7	8	1	128 832	1 613.02	1 956.92	36.09	3.98
128	3	16	3	188 842	1 382.72	5 539.45	27.28	5.12
128	5	16	2	285 553	2 221.39	3 906.72	46.60	5.90
128	7	16	1	257 323	3 005.01	2 283.64	48.66	4.27

Table 6.4: End-to-end performance of single convolution layers in hardware, compared to single-threaded execution on Cortex-A53. Copy duration involves both copies in and out. GOp/s are computed for hardware only.

	Component	LUT	Reg	DSP	LUTRAM	UltraRAM
10 × 7	Complete design	81 333	51 669	100	7 315	64
	of available	*35.3 %*	*11.2 %*	*5.8 %*	*3.2 %*	*20.5 %*
	Single PE	809	414	0	76	0
	Postprocessing	8 901	9 038	28	279	0
	Scratchpad	6 083	9 909	0	1 716	64
	Control Unit	8 333	3 643	72	0	0
14 × 12	Complete design	177 547	106 589	156	15 899	64
	of available	*77.1 %*	*23.1 %*	*9.0 %*	*6.9 %*	*20.5 %*
	Single PE	810	414	0	76	0
	Postprocessing	15 234	15 493	48	479	0
	Scratchpad	9 238	15 598	0	2 652	64
	Control Unit	12 877	5 691	108	0	0

Table 6.5: Resource utilization both of the complete design and divided into PE, scratchpad and control unit. Percentages relative to the available resources on Xilinx XCZU7EV. LUTRAM usage is included in the total LUT count.

the control unit. The control unit implements the full loop nest in hardware, which significantly reduces overhead through host interaction, and the necessary control and address generation units account for only 10.2 % of the resources. The high level of data reuse comes at a cost, so the RS dataflow requires a significant amount of distributed storage: In the 10 × 7 design, 9.4 % of the allocated Lookup Table (LUT) resources are used for buffers in the PEs. Note that these buffers were not implemented as BRAM, since the line buffers are shallow (64 elements for input activations & weights, 128 for accumulators). The number of BRAMs would be significant (1.5 per PE), but leave most of their capacity unused. However, implementation in BRAM can be enforced if necessary, e.g. when LUT resources are scarce. A distinct memory component of FPGAs by AMD, UltraRAM, is used to implement the scratchpad. Compared to BRAM, which stores 4 KiB plus bits for error correction, UltraRAM elements are 32 KiB in size. Thus, they are preferred to implement large memories, and the scratchpad implementation is optimized to allow inference of UltraRAM blocks. As a consequence, no BRAM is used in the design.

6.2.3 Conclusion and Outlook

In the previous sections, an FPGA implementation of a systolic-array-based convolution hardware accelerator was demonstrated. The RS dataflow is implemented, which enables a particularly high level of data re-use within the array. The alternative TRS dataflow was showcased with significantly improved utilization of the accelerator for most real-world applications. Various parameters of the hardware design were analyzed to highlight the adaptability of the accelerator for different use cases. This is supported by a Design Space Exploration (DSE) environment based on HDL simulation of the full architecture. The HDL design sources are publicly available, making RS accelerators accessible to a broader scientific community.

While the focus of this section is CNN acceleration, other operations can be mapped to the systolic array. The design is created with reusability in mind, making the PE array, data load/store facilities and host interface generic and independent of the workload. Modules defining the operation carried out in the accelerator are the control and address generator units only. It is sufficient to appropriately create these two units in order to implement other operations on the accelerator, if they are applicable to systolic arrays. For example, General Matrix Multiplication (GEMM) operations, the primary operation in transformer networks, have already been demonstrated to work on the PE array itself by supplying suitable control commands and addresses manually. For stand-alone operation, control and address generator units are required to create this command sequence in hardware.

The concept of closely coupled FPGA and ASIC can also be applied here. Given that the control and address generator units can run at a lower frequency than the memory interface, it is feasible to implement them using an embedded FPGA. This allows the behavior of the accelerator to be changed using run-time reconfiguration of the ALP, while the memory interface and PE array are implemented for peak performance as an FLP.

6.3 Run-time Monitoring for Tiled Architectures

Based on the original publication *Non-Intrusive Runtime Monitoring for Manycore Prototypes* at HiPEAC 2023. [Les+23]

The development of novel computing architectures requires in-depth verification and thorough validation. Especially when targeting ASICs, the *First Time Right* principle should be followed [139]. New components of any SoC are therefore tested extensively using hardware simulation. Most errors on component or unit level can be ruled out at this stage. However, the interplay of different components and the system as a whole is not verified. Cycle accurate system simulators such as GEM5 [140] offer a highly configurable framework optimized for architectures with GPPs. Such simulators provide high accuracy, but are often not suitable for comprehensive verification of manycore designs due to the high complexity of a large system simulation [141]. Therefore, other approaches for system-level simulation are required.

In general, simulation performance is improved by increasing the level of abstraction. Virtual platforms, for example, do not attempt cycle-accurate simulation. Instead, they make use of high-level models of the hardware design [142]. This is a valid approach for tasks like design space exploration or functional verification, but it omits important details of the hardware design. It is therefore desirable to verify cycle-accurate behavior to increase the test coverage. Rapid prototyping techniques can be used to solve this issue. A common approach is to implement the full architecture on an FPGA system. While not reaching the full performance of an ASIC realization, the execution is cycle-accurate by design and still faster when compared to simulation.

An FPGA-based prototype offers interfaces for programming, debugging and communication. FPGA bitstreams can be programmed using JTAG and software can communicate using UART, Ethernet, PCIe, and various other interfaces. However, the visibility of the internal state is limited compared to simulation. If the status of a signal or register is vital for debugging or evaluation, a software or hardware interface has to be setup in advance. However, both approaches have some drawbacks: accessing a software interface over shared resources can influence execution behavior, while attaching a hardware debugger is a tedious process and the available read-out bandwidth is limited.

In this section, we present a non-intrusive monitoring system for large NoC-based architectures. This includes scalable dataflow accelerators as described in Section 6.1 and SoC platforms like manycore systems, which are used as the target platform

of our monitoring system as an example. The design allows capturing numerous hardware- and software-based metrics of different types, which can be acquired and monitored outside the design. Special efforts have been made to prevent interference with the prototyped system in order to maintain the quality of the acquired metrics. In the following, the concept of the monitoring system is introduced, including both the hardware architecture and the corresponding software stack. Afterward, an implementation in an FPGA-based manycore prototype in Section 6.3.3 is discussed and an evaluation of performance and resource usage follows in Section 6.3.4.

6.3.1 Related Work

Run-time verification is a broad field focusing on the dynamic analysis of computing systems to ensure correct behavior. A formal specification of the expected system behavior serves as the foundation for synthesizing a set of monitoring instrumentation. The execution traces created by these during run time are then compared against the specification to identify deviations from the expected behavior [143]. Creating the execution traces in software involves a certain overhead which can alter the system behavior, therefore creating these traces in hardware has been examined [144]. The approach by Solet et al. greatly reduces the software overhead, although not completely avoiding it, and requires the hardware to be reprogrammable to include the hardware-based monitoring instrumentation. Instead of processing the generated trace data offline, Hoppe et al. propose to process this data directly within the system to ensure the security of the monitored processors [145]. Run-time verification methods commonly aim to create a complete trace of all significant events, which becomes a great challenge for large-scale and distributed systems [146]. In contrast to verification using a formal specification, the proposed system aims to be a tool for debugging and design space exploration, with absolutely no impact on the behavior of the running system. Additionally, our approach does not require a specification to verify against and instead the captured data is visualized to help the developer identify incorrect behavior.

Debuggers are tools to help a system developer to detect and analyze errors. Both hardware and software debuggers exist, however they take fundamentally different approaches and are built differently. Software debuggers either modify the software, which can affect the behavior of the system, or they rely on special hardware support, which may not be available or may also affect behavior [147]. Hardware debuggers have a different scope by capturing wires within a design and therefore give access

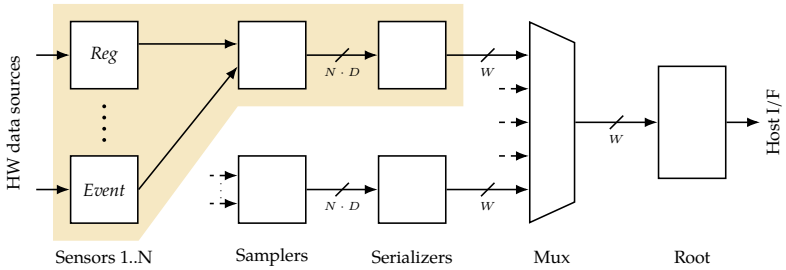


Figure 6.13: Overview of the proposed hardware architecture with one sensor group highlighted, N denotes the number of sensors in a group, D is the value width of a sensor, W is the data bus width of the mux tree

to otherwise hidden internal signals. However, the number of signals that can be detected is limited, and they usually need to be carefully adapted to the analysis of a particular problem. Lee et al. propose a framework for minimally intrusive online monitoring of embedded processors [148]. Hardware observability interfaces are used to capture specific predefined events from the CPU and peripherals. They are connected using a dedicated bus system together with an additional processor, which runs the observation software. This system allows very detailed non-intrusive analysis and is self-contained, but by using a dedicated processor within the hardware design, it does not scale well to manycore systems. For our monitoring system, we do not aim for cycle-accurate debugging. Instead, qualitative performance metrics shall be obtained, which allow for DSE and detection of bottlenecks in dataflow architectures.

6.3.2 Concept for Non-Intrusive System Monitoring

The overall goal of our non-intrusive monitoring system is to capture performance metrics from different data sources in the system prototype and forward this data to an external system for further processing and analysis. Several different sources of performance metrics are available across a SoC prototype, including single-bit events like interrupt lines, fixed-width registers like predefined performance counters, occupation of system buses in general or metrics calculated in software. A single SoC can contain thousands of potential interesting data sources. When such SoCs are scaled up to create a Multiprocessor System-on-Chip (MPSoC) or replicated to compose a manycore, this number quickly grows by orders of magnitude, scaling

proportionally with the size of the prototyped architecture. Therefore, a monitoring system needs to be scalable, but also flexible enough to adapt to heterogeneous and irregular architectures.

To reduce errors during measurement, the proposed monitoring architecture is designed to share as few resources as possible with the system being prototyped, which is referred to as the Design Under Test (DUT). Hence, it is conceived as an independent hardware design, from capturing the metrics up to sending the data stream out. Besides data provided actively by software, all sensors operate passively and can thus obtain data without impacting the execution of the prototype system. It is desirable that the resource overhead and the timing requirements are kept low, that the majority of resources and timing budget is still available for the DUT. Care has been taken to minimize the timing impact on the target design when integrating the non-intrusive monitoring system, since every data path in the monitoring system can be split using a specialized component if necessary. This component can even be replicated multiple times if necessary to reduce timing and routing complexity further, using a small amount of logic in return. The structure of the hardware design is shown in Figure 6.13 and consists of the following basic components:

Sensor A sensor captures raw data from the DUT and converts it into a consistent format for further processing. Different types of sensors need to be considered for different types of data sources (see Section 6.3.2).

Sampler Data from each sensor is being sampled either at a fixed time interval or based on an external trigger condition. The fixed time interval is usually shared with other sensors in a group, but can be reconfigured individually if a given source has to be monitored more or less frequently. An internal counter determines the time of sampling and can be shared with other samplers.

Serializer Since the monitoring data has to be converted into a data stream, the serialization component converts the sampled sensor data of arbitrary width to words that match the fixed internal bus width. Data is padded to multiples of the bus width and prefixed with an identifier number. The serializer node can be configured to omit sensor values from the data stream that have not changed since the last sample. This saves bandwidth, but requires additional hardware resources to compare the values and has to be enabled at design time.

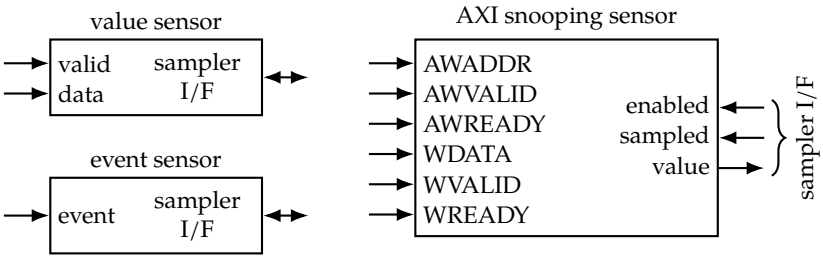


Figure 6.14: Interface of the value, event and AXI bus snooping sensors

Mux-based interconnect tree The flow of monitoring data is an N-to-1 communication pattern, as sensors are physically distributed throughout the whole design and their data needs to arrive at one single root node. This results in a tree structure with the sensors as leaves and the root node representing the destination of the data stream. At each fork, a fully buffered data multiplexer is inserted to merge incoming branches into a single data stream.

Root node The root node receives a single data stream containing monitoring data from all sensors. It is responsible for packetizing, generating a checksum and forwarding this data to an external interface, e.g. Ethernet.

Types of sensors

Sensor modules are responsible for gathering information from various sources within the prototyped system and converting it into a common representation. Potential data sources include CPU performance counters, bus usage and transfer error indicators, status registers of hardware accelerators and NoC usage statistics. Depending on what data should be monitored, different sensor types can be used. The monitoring system includes a library of different sensor types for both VHDL and Verilog designs, covering most common applications.

The simplest case is a sensor for a hardware register that directly represents the value to be monitored, e.g. a performance counter which is part of a CPU. This value is being passed through to the sampler component directly. Another basic sensor type is an event counter, which counts single-bit events within each sample interval. It is commonly used to count interrupts, bus transfers, cache hits/misses etc. Combining these two sensor types leads to a conditional counter: when a certain condition is met, a value is being captured and optionally accumulated. This behavior is useful for

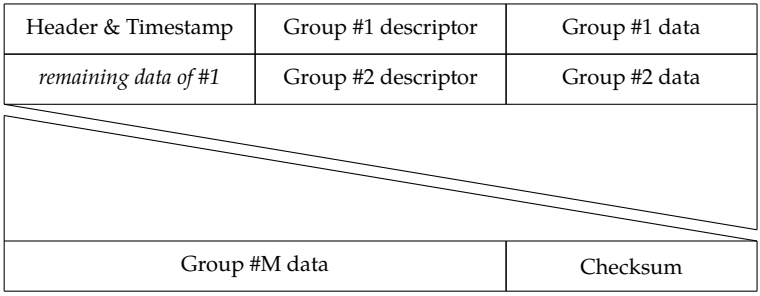


Figure 6.15: Data packet format of a full sample from M groups at the root node, assuming two sensors per group

values which are only present at certain points in time, like the duration of specific hardware operations. Specialized conditional counters for common bus systems are also provided: reads and writes to AXI and AHB buses can be snooped on certain addresses to monitor memory contents without additional accesses. Figure 6.14 shows the register-based value sensor, the event sensor used to count single-bit events and the AXI bus snooping sensor, including the interface to the sampling component.

A generic memory-mapped register file is also provided to monitor software-provided values. Using this interface violates the non-intrusive design paradigm, but has been proven to be quite convenient as a high-performance data logging path. If no predefined sensor matches a specific use case, a new sensor type can easily be defined. Multiple physically close sensors are grouped and assigned a specific identifier. Depending on the prototyped architecture, such a group can represent a single CPU, a tile of a hardware accelerator or a cluster of related processing elements.

Tree-based unidirectional NoC for monitoring data

In a large-scale prototype, data sources of sensors are distributed throughout the entire hardware design. We are using a hierarchically structured interconnect tree, which can be freely scaled to match the prototyped design. All intermediate components are fully buffered to avoid long paths, improving timing and allowing timing optimization to focus on the actual DUT. The dataflow is primarily from leaves (sensors) to the root node with a configurable width. However, a reverse path is also provided for configuration of the sampling component, i.e. setting a specific sample interval or disabling a sensor entirely without rebuilding the bitstream. As the sensor and serializer nodes are rarely reconfigured, the reverse path is a single-bit serial interface.

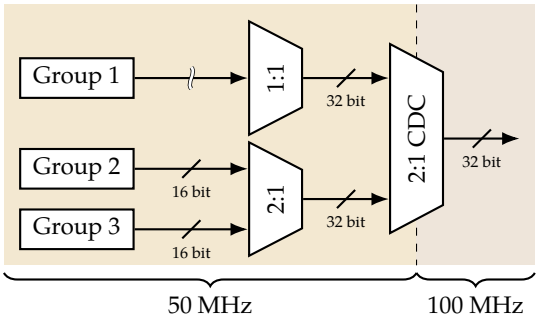


Figure 6.16: Example of a data collection tree including a 1:1 mux and a mux with Clock Domain Crossing (CDC) at the root

This results in very low resource usage for this data path, but increased transmission time for configuration commands.

On each fork, the multiplexing component (*mux*) merges the incoming branches of data. Figure 6.16 shows an example of the collection tree, based on the mux as central building block. The mux is designed to receive a full set of monitored sensors from one branch and proceed to the next after all sensor values have been forwarded. It utilizes round-robin scheduling of its inputs, thereby generating a continuous stream of sensor data at its output. This stream comprises all groups from the first input, followed by all groups from the remaining inputs, as shown in Figure 6.15. The packet header including a timestamp and a checksum is appended at the root node. A group descriptor contains a group ID to distinguish sensors with equal IDs but from different groups. Each group descriptor is followed by one record per sensor in that group, each record contains the sensor ID and its current value. The width of the ID and value fields is freely configurable, but must add up to a multiple of the bus width of the interconnect tree.

The mux tree features a backpressure signal generated by the output module. This signal is buffered within each mux to avoid long-distance wires in the design. To support continuous data transfer cycle-by-cycle, the mux component has the capacity to buffer one data word in the event of backpressure. This enables it to compensate for the delay in the backpressure signal by one cycle. Multiple mux instances can thus be connected in series to adapt to the layout of sensor nodes in a hierarchically clustered, heterogeneous system.

A special case is a single-input mux: there is no need for arbitration, however all input and output signals are still buffered. Such a mux can be inserted into the dataflow to improve timing of the monitoring system on large designs by reducing the critical path length and allowing for flexible placement of the intermediate mux registers by the design tools.

Lastly, a mux can also be configured as a data width converter including potential clock domain crossing. This compensates for the fact that the root datapath limits the bandwidth of the whole tree structure by increasing both clock and data width by a power of two. For example, an 8:1 CDC mux with a 16 bit input at 50 MHz can output a stream of 64 bit at 100 MHz, having a data rate of 800 MiB/s on both sides. However, this comes at higher hardware costs for FIFOs on each input and should only be considered if the full monitoring system cannot run at the desired data rate of the host interface.

Output module

The output module is at the root of the dataflow tree and receives the completed stream of monitoring from all attached nodes. This stream is prepared and transmitted to an external system for further analysis and storage. Several types of output modules can be implemented, depending on the I/O interfaces of the target system. As most FPGA platforms provide an Ethernet interface, the standard output module comprises a hardware TCP/IP stack to send monitoring data encapsulated in UDP packets. Figure 6.17 illustrates the structure of the hardware TCP/IP and Ethernet stack, including ARP and ICMP handlers for standard IPv4 compatibility. The Ethernet output module packetizes the data stream, adds checksums for error detection, and forwards it to the UDP module, which sends the data to a configurable IP address. Other output modules can be implemented to use different interfaces. In Section 6.3.3, we discuss interfacing the output module to a commercial rapid prototyping platform. In any case, the output module can also apply backpressure to the dataflow tree to throttle incoming monitoring data if the I/O interface bandwidth is insufficient. Section 6.3.4 provides both an estimate and real-world measurements of the required bandwidth.

Software stack for evaluation

To generate valuable insights on the execution of the monitored system, we have to store and preprocess the collected data. A Commercial-Off-the-Shelf (COTS) PC system can be used to receive monitoring data and store it for further processing.

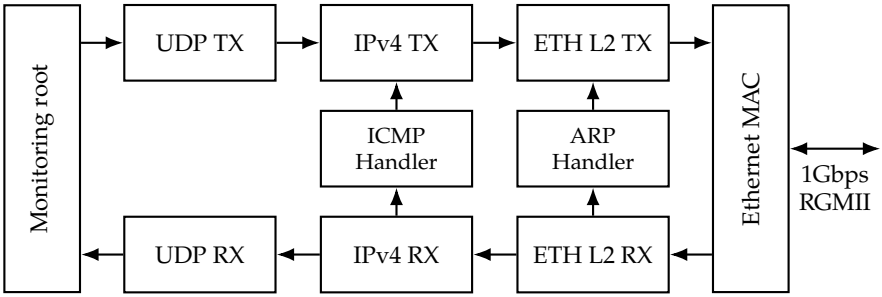


Figure 6.17: Architecture of the hardware Ethernet stack handling IPv4 and UDP layers and including support for ARP and ICMP

We developed a software stack to handle incoming monitoring data via Ethernet including checksum verification which stores each sample in JSON format. This data can then be further processed according the user's needs. Depending on the output module in use, it can be necessary to reassemble packetized data which is done as one of the first steps of the receiving process.

For example, we propose a setup using a time-series database and a web application. Incoming monitoring data is stored in the database according to the hardware timestamp, allowing for offline processing of the captured data after execution. The web application can display live data during execution, but also allows exploring previous runs based on data stored in the time-series database.

6.3.3 Monitoring of a NoC-based Manycore Prototype

We employ a tiled manycore system on a multi-FPGA platform, based on four Xilinx Virtex-7 XC7V2000T FPGAs, as a representative platform for a sophisticated prototype architecture. The manycore prototype is a tiled 4×4 architecture, connected using a meshed NoC. Each tile contains five LEON3 processors and 8 MiB of local memory. Peripherals include a network interface to access the NoC with a level 2 cache to improve read performance from the remote, shared DDR memory. Both the tiles and the NoC run at 50 MHz, while the DDR memory is clocked with 200 MHz. Debugger access is available for each tile separately including serial redirection. However, getting run-time performance data outside the system is still a tricky task: the serial console

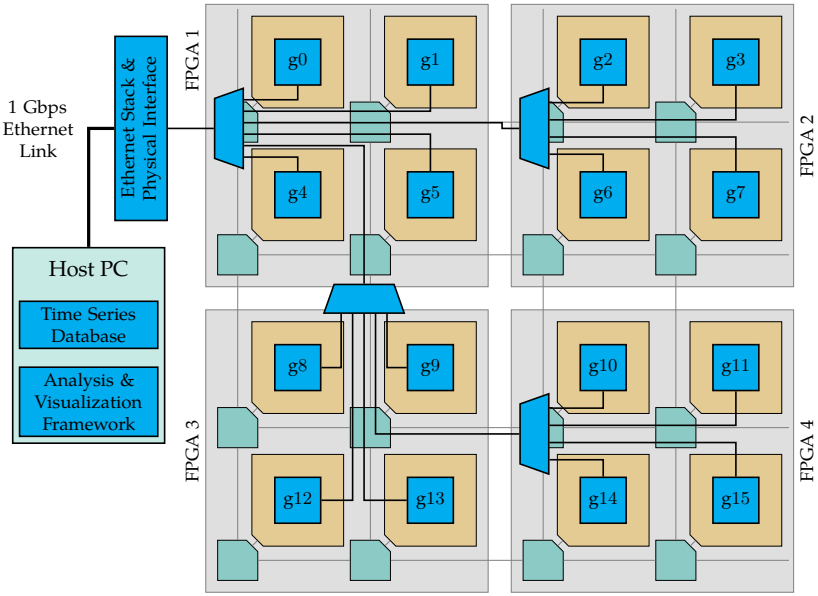


Figure 6.18: Proposed monitoring system mapped to a 4 × 4 tiled manycore architecture on a quad FPGA prototyping platform with 16 sensor groups connected over FPGA boundaries

performance is very limited and collecting metrics in software uses a considerable amount of CPU resources, while sending the results to the I/O interface creates additional NoC load.

To show how an existing system can benefit from our approach, we mapped the proposed non-intrusive monitoring system to this manycore prototype. Figure 6.18 shows the 16 tiles of the manycore prototype with one group of sensors per tile (g0 to g15), as each tile generates similar monitoring data locally. Since a multi-FPGA platform is used, the design is partitioned into four quadrants, resulting in four tiles per FPGA. Monitoring is first gathered from all four tiles in one quadrant before being sent over high-speed serial inter-FPGA links. Only one of the FPGAs features an Ethernet interface, thus the inter-FPGA links are used to connect the remaining FPGAs to the root node. To ease timing, a single-input mux is placed immediately before and after each inter-FPGA SERDES driver.

To show the broad range of applications, different sources of metrics are being monitored, including:

- Eight assignable performance counters per CPU, tracking I/D-cache misses, instruction counts, branch prediction and interrupt statistics.
- NoC transfers by type and buffer levels of the attached router.
- Usage of the tile-local AHB bus by type.
- Power consumption and temperature as estimated by the integrated power emulation system [149].

CPU and NoC performance counters as well as power and temperature are value-based sensors, as these values are already present in registers. For bus usage and interrupt statistics, event-counting sensors are used. In total, 60 sensors are monitored per tile, summing up to 960 sensors for the whole system.

Two implementations of the root node are provided: one using an Ethernet connection and another using a proprietary interface of the FPGA prototyping platform. The Ethernet root node makes it very easy to connect the processing host, but it is limited to 1 Gbit/s bandwidth. The MMI64 system is a custom bus system used within proFPGA ASIC prototyping systems [150], which is used to program the FPGAs but also for debugging access to the tiles during run time. It supports several transports among which PCIe 2.0 x8 offers the highest performance. This transport can be used if no Ethernet connection is available, however it does not offer significantly better performance due to the internal architecture of the MMI64 bus system.

Both the root node's interface to the interconnect tree and the one to the host PC can be bottlenecks in the system. Section 6.3.4 provides an estimate on the achievable sampling rate, assuming the host interface does not limit further. Apart from maximum bandwidth, the number of packets per second can be a limitation, especially for configurations with a high sample rate. Therefore, both the Ethernet and MMI64 root nodes combine monitoring data of two consecutive sample points if possible to lower the packet rate and maximize bandwidth.

On the receiving host, a combination of InfluxDB [151] and Grafana [152] is used to store incoming monitoring data and handle visualization during and after execution. In addition to the time-series database, live monitoring data is streamed to the Grafana instance directly during execution to display it as soon as it is received from the DUT. Detailed post-execution analysis can be done either using the provided web interface

or via any other compatible database client on the same granularity as originally produced. Monitoring data, both captured live and stored in the database, can be saved in JSON format using the included tools for further analysis.

6.3.4 Evaluation

We evaluated the maximum possible sample interval of our approach in theory and verified this result using the case study introduced in Section 6.3.3. In addition, an overview of the resource consumption of our proposed system in this configuration is compared to the manycore prototype. We also measured the impact of the data reduction algorithm when running a parallel benchmark suite on the manycore system. Finally, some specific use cases of our proposed system for debugging are highlighted.

The maximum sample interval of the system depends on several factors: the number of enabled sensors, the width of data and sensor ID values and the bandwidth of the root node. To give a theoretical estimate for the maximum sample interval, a system with N_{grp} sensor groups having N_{sens} sensors is assumed. Each sensor is configured with a w_{id} bytes identifier and a w_{data} bytes value, while the interconnect tree has a width of w_{bus} bits and runs at the frequency f_{bus}. The header size w_{hdr} and the size of each group descriptor, equal to a sensor, is also included in the following formula:

$$\Delta t = \frac{w_{\text{hdr}} + N_{\text{grp}} \cdot (1 + N_{\text{sens}}) \cdot (w_{\text{id}} + w_{\text{data}})}{w_{\text{bus}} \cdot f_{\text{bus}}}$$

Our reference implementation uses 32-bit sensor values and 16 groups with 60 sensors each. This results in a theoretical maximum sampling interval of 58.6 µs when running the design at 50 MHz with a 16-bit data bus. The theoretical maximum can still be limited by the host interface, which needs to support the bandwidth of 100 MiB/s in this case. Measurements on the FPGA prototype using gigabit Ethernet show an average bandwidth of 99.4 MiB/s for this configuration, missing about one single sample per second as the bandwidth does not fully suffice.

Table 6.6 lists the FPGA resource usage of the monitoring system including individual components. It focuses on one quadrant of the prototype as shown in Figure 6.18 on one FPGA, containing four tiles with 5 CPUs each. As it contains the root node, a 6:1 mux is used to merge all four local sensor groups with incoming data from the other design quadrants.

Component	FF	LUT	BRAM
Sensor group (60 sensors)	1507 (0.4 %)	707 (0.1 %)	0
6:1 mux	82 (0.01 %)	82 (0.01 %)	0
Ethernet root module	4066 (0.9 %)	2350 (0.4 %)	6 (0.7 %)
MMI64 root module	1651 (0.4 %)	1249 (0.2 %)	6 (0.7 %)
Full monitoring system (single FPGA w/ MMI64)	7858 (1.9 %)	4428 (0.7 %)	6 (0.7 %)
Single quadrant of the full design (1 FPGA, 4 tiles, 20 CPUs)	*423665*	*652652*	*879*

Table 6.6: Resource usage of the full system and individual components, portion of the full prototype design in brackets

The data reduction feature of the sampling components can greatly reduce the required bandwidth. The NAS parallel benchmark suite [153] contains a set of benchmarks for distributed systems which are also suitable for manycore prototypes. For example, running the BT test (block tridiagonal solver) allows data reduction of monitored data by 85.7 % when monitoring 60 sensors per tile as described in Section 6.3.3. This is due to algorithms used in the BT test, which contain only short communication phases and most time is spent in the computational kernel. Therefore, sensors for NoC traffic, temperature and peripheral operations are rarely changed and transmitted. This is however dependent on the application and monitored values, as for other benchmarks data reduction rates of 15 % have been measured. The maximum theoretical bandwidth should be supported in any case, otherwise samples could be dropped. The data reduction feature can improve the user experience if a shared interface like the MMI64 system is used, as latency for programming and debugging tools is reduced when the full bandwidth is used at all times.

We want to stress that the main benefit of this work, apart from the low resource requirements and easy applicability, is the debugging flexibility this system opens up. The software stack includes an easy-to-use, yet powerful tool to visualize the acquired measurements and quickly inspect the run-time behavior of the DUT, even already during execution. The monitoring system helped to successfully detect several bugs throughout hardware modules, operating system and applications and allowed to locate the actual error sources. As an example, we noticed an irregular deviation in the power & temperature emulation system [149]. Unaware of the actual cause of

this issue, the emulation system itself was investigated first. However, the monitoring system indicated that the issue only occurs close to the border between two FPGAs of the prototyping system. We could therefore quickly identify that the inter-FPGA SERDES was configured incorrectly. A second example was a rare scheduling issue of our operating system. Specific benchmarks showed degraded performance on some specific tiles on our manycore system. The monitoring system immediately made clear that for some time periods, tasks were not scheduled to some of the processors. The related scheduling bug was quickly solved. Apart from debugging purposes, the non-intrusive monitoring system has also been used to extract training data for neural networks to identify suspicious side-channel communication.

6.3.5 Conclusion and Outlook

When working with FPGA-based prototypes for verification of large architectures, access to run-time information is vital. Debugging on FPGAs is often a tedious process, since, similar to integrated circuits, they operate as a kind of *black box*. To examine an issue within the design, a specific debugging channel has to be engineered and integrated into the design. Alternatively, if resources which are part of the DUT itself are used to gain insight, the behavior of the system might be altered.

To approach these issues, a distributed monitoring infrastructure for FPGA-based hardware prototypes was introduced in this chapter. It is designed to be scalable for both homogeneous architectures like manycores and heterogeneous designs by supporting unbalanced sensor trees with asymmetric bandwidth requirements. Different kinds of value- and event-based sensor types have been discussed, providing an interface to commonly used data sources. Most importantly, the monitoring system can operate fully non-intrusively if required, so that the behavior of the prototyped system is not influenced. As the system resides on the same platform as the DUT, it uses some of the available hardware resources and timing budget. However, the resource overhead has been kept as low as possible by using a lightweight interconnect, and specialized mux components can be used to improve timing on complex designs. A standard 1 Gbit Ethernet interface, which is present on most FPGA prototyping platforms, is sufficient to greatly increase the observability of the DUT and gain useful insights at run time. In a case study with a 4×4 tiled manycore architecture, a sample set with 960 measurements could be acquired with a sample rate of up to 17 kHz. The non-intrusive monitoring system made visible several errors throughout the

prototyped platform, in hardware components, operating system and applications, and allowed us to detect the source of the error more quickly.

We have demonstrated that our monitoring system can be employed in large-scale prototypes. It has been manually integrated into many different components of the prototyping platform to collect sensor data and connect the data collection tree. For future work, this task could be automated, such that no manual changes to HDL code are necessary. Similar tools exist for the on-chip FPGA debugger *ILA* by AMD, which allows inserting debug probes on distinct wires after synthesis. However, the scope of ILA is limited to cycle-accurate analysis of data on a single FPGA, but a similar concept could be used to create tools for automatic integration of the monitoring system.

The monitoring system is not specifically tailored to manycore usage, but can be used to monitor any large-scale platform where the monitoring data sources are distributed throughout the design. Therefore, this system is used to collect run-time data on the adaptive accelerator platform introduced in Section 6.1. Each accelerator tile can export a set of sensors, which are collected by the monitoring system and streamed using Ethernet for evaluation on a secondary system.

6.4 Results & Evaluation

Both the reconfigurable accelerator platform and the CNN tile have been fully implemented in HDL. The platform is prototyped on a Xilinx VCU128 evaluation board with the Virtex UltraScale+ XCVU37P-L2FSVH2892E FPGA. It provides 8 GB of HBM and 4.5 GB of DDR4 on-board memory resources. The PCIe interface runs at 16 GT/s and x8 lanes. Xilinx Vivado 2024.2 is used for synthesis, implementation and generation of static and partial bitstreams. The host platform is an NVIDIA Jetson AGX Orin Development Kit, running the driver library and benchmarking software.

Simulation results have been obtained using automated cycle-accurate testing of the RTL model with Siemens Questa Prime version 2022.4. For each simulation run, random input activation and weights are generated and mapped to the CNN tiles using a greedy scheduling algorithm.

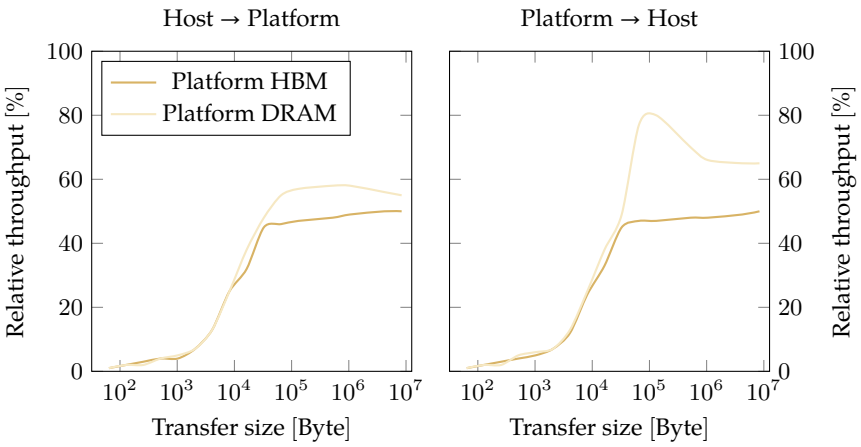


Figure 6.19: Performance of DMA transfers to and from the accelerator platform relative to the maximum PCIe bandwidth.

6.4.1 Memory & DMA Performance

First, the host interface performance is assessed as the limiting factor of input and output data transfers. DMA transfers are initiated from the host using preallocated buffers in its DRAM. The destination address is either located in the DDR4 DRAM on the accelerator platform, or located in the northern HBM tile. Transfers to DRAM only need to cross a single AXI bus bridge, which is directly connected to the PCIe DMA engine. When the HBM memories are target of a transfer, AXI transactions from the DMA engine are translated to NoC packets, forwarded to the correct coordinate, and translated back to AXI transactions towards the HBM controller. Benchmark results for DMA transfers in both directions for buffer sizes from 64 B to 8 MiB are visualized in Figure 6.19. Performance is given relative to the maximum throughput of the PCIe Gen 4 interface of 16 GT/s, which equals a gross bandwidth of 15 760 MiB/s for eight lanes.

It can be seen that the throughput of small transfers up to 4 KiB is very low. This can be explained with the overhead of setting up the DMA engines. Additionally, the maximum size of a single PCIe transfer is 4 KiB, thus small transfers cannot benefit from burst transfers. Larger transfers scale linearly with the transfer size and profit from consecutive bursts on both PCIe and AXI interfaces. Finally, DRAM bandwidth tops

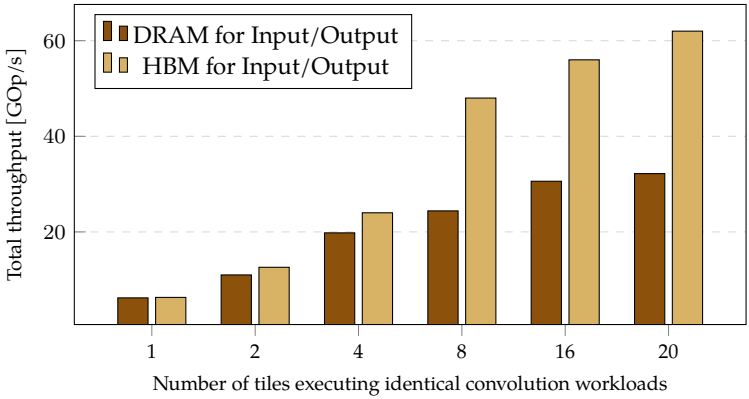


Figure 6.20: Scalability analysis performing convolution operations on 1 up to 32 tiles, measuring computational throughput when either DRAM or HBM is used to store input and output data.

out at 8.62 GiB/s and 10.01 GiB/s for transfers to and from the accelerator platform, respectively. Towards HBM, a throughput of 7.69 GiB/s is achieved in both directions. Note that HBM performs slower than DRAM because only a single channel is used in this benchmark. The main benefit is that all HBM tiles in the NoC can supply this bandwidth in parallel, while DRAM performance is shared between simultaneous transfers.

6.4.2 Parallel CNN Processing

In the following, we consider a 4 × 4 accelerator platform in which each tile was configured with the CNN accelerator from Section 6.2. In order to analyze the scalability of the architecture, we have mapped a CNN workload on one or more tiles. For each tile, a single convolution of $HW = 32$, $RS = 5$, $C = 128$, $M = 2$ is mapped. Input and output data is placed into Shared Memory (SHM) prior to the test run. Each run consists of copying data into the accelerator tile, executing the convolution operation, and copying data back into SHM. Running a single tile represents the baseline with the accelerator tile running at 6.28 GOp/s in the standalone case (Section 6.2.2). However, an overall throughput of 5.7 GOp/s is measured in this experimental setup, as we incorporate both copy-in and copy-out into the overall performance.

Figure 6.20 shows performance results for a single tile up to the full platform used for processing. We observed that performance scales worse when using data backed by DRAM, due to the fact that only a single tile is used to access the DRAM. In contrast, two independent memory interfaces can be used when targeting HBM, which can deliver the necessary bandwidth for multiple tiles working in parallel. This highlights that HBM is a key component for a scalable high-performance platform. For our small 4 × 4 design, two HBM channels are sufficient. However, there are 16 channels available in total on the XCVU37P FPGA, which can be attached when scaling to larger NoC topologies. Our design framework allows for flexible placement of the HBM tiles to match floorplanning requirements. For example, a 5 × 5 meshed NoC profits from placing two additional HBM tiles at the east and west edge.

Since only a single layer is accelerated in this test, this represents performance scaling in the worst case. The tiles cannot operate during copy-in and copy-out due to the lack of subsequent workload, thus performance is degraded. By applying sophisticated scheduling mechanisms, DMA transfers can run in parallel to processing within the tiles and thus further increase overall performance. Additionally, output channel tiling within the tiles can also allow the accelerator cores to operate continuously. The number of output channels in this test equals $M = 2 \cdot N_{tiles}$, therefore any workload requiring more output channels needs to apply tiling. Similar to multilayer operation, DMA transfers of the previous output channel set can also run in parallel. Figure 6.21 illustrates parallel data transfer and accelerator operation on multiple tiles with the workload mapped to the same SHM. In this example, input data for a subsequent run can be transferred to each accelerator tile before processing of the current workload is finished. This allows to copy result data to SHM while the next task is already being processed. The scheduling algorithm needs to consider hardware limitations such as the scratchpad memory size, which needs to fit two workloads for pipelined operation. In case of output channel tiling, input activation data is reused and only a new set of kernels is required, which is usually considerably smaller.

6.4.3 Discussion

Our concept for an adaptable accelerator platform highlights the advantages of a hybrid of fixed logic (FLP) and reconfigurable logic (ALP) to balance performance and flexibility. Implementing memory interfaces and interconnects in the FLP provides high bandwidth for memory-centric workloads. In contrast, accelerator tiles are composed primarily of reconfigurable fabric, only the NoC interface and DMA

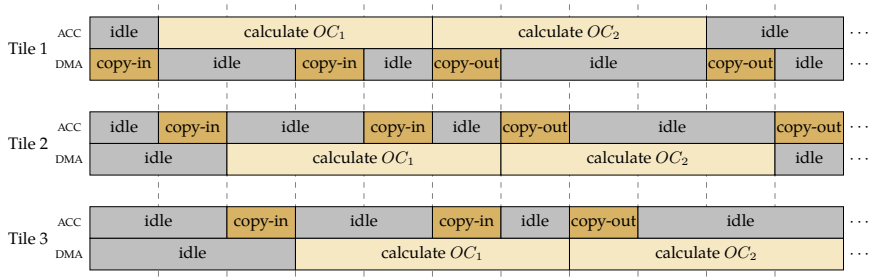


Figure 6.21: Pipelined operation of DMA and accelerator (ACC), avoiding stalls due to data dependencies.

engines are fixed. This allows for great flexibility within the tiles, but requires extensive parallelization and pipelining to achieve high performance. Our design study on CNN accelerators demonstrates that even a small amount of ALP can provide significant adaptability for closely related algorithms, such as our two convolution dataflows and GEMM, which share the same atomic MAC operation. However, this approach is incompatible with algorithms requiring different operators or dataflow paths in the systolic array. The main takeaway is that accelerator design is not a binary choice between fully fixed or fully reconfigurable hardware. Instead, a spectrum of viable hybrid solutions exists, allowing the ratio of fixed to reconfigurable logic to be tailored to the class of current and future workloads.

When evaluating our results, it is important to understand how our FPGA prototype compares to a final ASIC implementation. Since the entire prototype runs on an FPGA, the performance difference between the fast, fixed logic (FLP) and the slower, reconfigurable logic (ALP) that would exist in an ASIC needs to be emulated. For example, a hypothetical ASIC realization might have its fixed components (like the network and memory controllers) running at 2 GHz. Assuming a clock frequency penalty of 8 × in FPGA fabric compared to the same logic implemented in ASIC, the ALP clock would be 250 MHz. This artificial penalty is chosen lower than Wong et al.'s analysis from 2011 to adjust for the improved performance of modern FPGAs [26]. We can mimic this 8:1 speed ratio on our FPGA prototype to get a performance estimate. In our specific test, a much smaller 2:1 ratio (200 MHz vs. 100 MHz) is used to deliberately create a bottleneck and stress-test the memory system. This clock ratio of 2:1 is very optimistic for an ASIC realization of the platform, since FPGA designs can barely reach frequencies of 1 GHz. As a result, our test shows

memory limitations appearing sooner than they would on an actual ASIC. Despite this, the model's overall behavior is a valid representation. If needed, a more precise simulation can be achieved by simply increasing the clock speed difference to better match a real-world scenario.

6.5 Conclusion & Outlook

Throughout this chapter, a reconfigurable accelerator platform for high-performance applications was presented. It puts particular focus on sustainable bandwidth for many of today's data-centric workloads, like processing of Big Data and Machine Learning. To realize a platform that both offers high memory bandwidth and is adaptive to different use cases, a hybrid approach of combining ASIC and FPGA technology was followed. Parts of the system implemented as hard logic, the Fixed Logic Partition (FLP), comprise memory interfaces, the NoC interconnect and units for interaction with the host. This setup delivers peak memory performance of 1.60 GB/s between memories on the NoC. In contrast to the FLP, the majority of each accelerator tile implements the Adaptive Logic Partition (ALP) using FPGA fabric. This allows them to adapt to various use cases by loading an appropriate configuration during run time. The data transfer infrastructure within the tile, namely the network adapter and DMA units, is still implemented as hard logic. They operate at the NoC clock frequency and can process data at line rate to avoid a reconfigurable tile becoming a bottleneck. A simple instruction set was defined to orchestrate the platform, which is implemented on a host system.

We showed the applicability of the accelerator platform by implementing an accelerator tile for CNNs. The design uses the row-stationary dataflow to maximize data re-use and parallelism in its PEs. Its default configuration uses 70 PEs, which achieve 6.28 GOp/s at 100 MHz. Through its standalone design, no interaction with other components is necessary during execution. It is sufficient to copy the configuration and input data to the tile local memory prior to execution. After processing, the result data can be copied out or re-used within the tile for the next layer. To test the accelerator design, a software suite consisting of low-level driver, mapping and benchmarking tools was developed. Its modular nature allows the accelerator to be integrated in existing runtime environments for neural networks. Both bare-metal and Linux platforms are supported by the software suite, covering both deeply em-

bedded and high-performance systems. Compared to execution on an embedded ARM processor, the design achieves end-to-end speed-ups of over 90 × .

For the future, there are several further areas of interest to investigate related to our accelerator platform. The basic integration of GPP tiles has been shown in our evaluation, as they can help to accelerate tasks for which a distinct hardware accelerator is difficult to design. The next step is their integration into the host driver software and establishing cooperation with other tiles, both GPP and accelerator tiles. This poses challenges similar to the programming of manycore systems, thus the integration of a distributed operating system should be examined. The state of the art in neural networks is steadily advancing, with new types of networks and novel algorithms being continuously developed. Although CNNs are still popular in embedded systems, there is a shift towards Transformer networks in cloud services and high-performance datacenter applications. As a consequence, these networks will most certainly transition to the embedded area as well. To provide support for these networks on the adaptive platform, accelerator tiles for the involved operations like GEMM need to be implemented. The data transfer and interfacing infrastructure of the platform can be reused, reducing implementation efforts to just the compute cores. In this thesis, the accelerator platform was only prototyped on FPGAs. The performance delta between ASIC and FPGA was simulated by artificially decreasing the clock frequency in reconfigurable tiles, reducing the overall system performance as a result. Manufacturing the platform in silicon allows validating our results in comparison to state-of-the-art fabrication nodes and embedded FPGA architectures. Additionally, the software framework supporting the accelerator platform can be enhanced. Although the hardware design supports multiple tasks being executed in parallel, additional operating system support is necessary for simultaneous access to the platform. The runtime environment needs to ensure that no resource conflicts occur and data transfers between applications are secured. In summary, the adaptive accelerator platform can serve as a powerful basis for further research on specialized platforms.

Chapter 7

Summary, Conclusion and Outlook

As we have seen over the course of this thesis, modern computationally intensive workloads still pose many challenges for today's computers. A main challenge, exacerbated by the latest algorithms in Machine Learning (ML), is the growing disparity between processing capability and memory bandwidth. The memory-bound nature of these low Operational Intensity (OI) workloads demands novel solutions to prevent data starvation at the processing elements. A second major challenge stems from the variety and much quicker evolution of algorithms, with which computer hardware design cannot keep pace. This imposes constantly shifting requirements on hardware acceleration architectures. Finally, the aforementioned performance and adaptability targets have to be balanced with the essential goal of maximizing energy efficiency, thereby enabling the development of sustainable high-performance computing.

As an approach to solving these challenges, we investigated hybrid platforms consisting of tightly interleaved Application-Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) regions. While performance critical components of the architecture are implemented as hard logic in the Fixed Logic Partition (FLP), units outside the critical path can be moved to the Adaptive Logic Partition (ALP) to allow for reconfiguration during run time. Since the optimal architecture depends strongly on the target algorithm, we presented a methodology for workload classification in Chapter 4. Based on the Roofline Model, our approach allows us to assess whether a processor-based system or dataflow architecture is more suitable for a given application. Hence, we investigated both kinds of systems, each extended by reconfigurable hardware following our approach.

First, we looked at workloads in which the primary complexity lies in the control-flow part. In such cases, General Purpose Processors (GPPs) are a common and

effective solution. However, applications can still contain kernels with high computational complexity for which a GPP is not well-equipped. We investigated the use of run-time reconfigurable processors, that combine a processor core implemented as ASIC with an FPGA fabric to provide hardware-accelerated special instructions. We demonstrated that such processors are particularly beneficial in large-scale manycore platforms. Placing them close to memory resources enables to use them as Near-Memory Computing (NMC) units, which profit from high memory bandwidth while still allowing for reconfiguration during run time. Additionally, we looked into approaches to boost the efficiency of reconfigurable processors. By applying approximate computing techniques, both processing performance and resource utilization could be improved significantly for workloads that tolerate slight inaccuracies in their result. For our third contribution to this topic, we shifted the focus to software integration for platforms with a high number of GPPs, specifically manycore systems, using hypervisors for task isolation. We developed a framework optimized for low-latency inter-guest communication in safety-critical applications, which is particularly advantageous for service-oriented architectures. This approach reduces communication overhead to under 30 µs, thereby enabling fine-grained application partitioning.

Second, we focused on dataflow architectures. As of today, the most important dataflow heavy algorithms in this domain are Artificial Neural Networks (ANNs) for ML applications. Since they push many current computers to their limits, we developed a specialized accelerator architecture for Convolutional Neural Networks (CNNs) as basis for our flexibility considerations. The systolic array architecture uses a row-stationary dataflow to enable maximum data re-use and optimize memory bandwidth during operation. We showed that our novel Temporal Row-Stationary (TRS) dataflow, an optimized variation of the original dataflow, improves the utilization of many common CNN layers. A single instance of the accelerator achieves 6.28 GOp/s at 100 MHz, which corresponds to 89.7 % of the theoretical maximum throughput. Taking the step towards high-performance CNN acceleration, we proposed an adaptive platform for general dataflow algorithms. The architecture hosts run-time reconfigurable accelerator tiles, which are connected with a high-bandwidth Network-on-Chip (NoC). Two types of shared memory, DDR4 DRAM and High-Bandwidth Memory (HBM), are attached to the NoC to fulfill the bandwidth requirements of the data-centric workload. We could show that the additional area required for the FPGA fabric inside the tiles is well invested to improve the performance of multiple acceler-

ator tiles. Ultimately, this allows us to build a scalable, high-performance compute platform.

7.1 Conclusion and Future Work

Adaptive architectures are a compromise between generalization and specialization. Depending on the workload type, a decision can be made in favor of a specific basic architecture, which can then be adapted as required to a certain extent. Since the base architecture includes many hard logic components to improve the overall performance, it is critical to select a suitable architecture in the first place. Our workload classification methodology described in Chapter 4 is a first step for platform selection. However, further potential lies in improved algorithms for Design Space Exploration (DSE) in this area for future work. It has been demonstrated that both types of architecture can generally serve as a basic platform in areas where the intended use is unclear in advance or may change in the future. However, if all requirements and parameters of the workloads are known, using a pure GPP or ASIC solution is generally more efficient. This can be the case in deeply embedded systems, e.g. in industrial manufacturing or control devices. In contrast, the fast-paced development in the field of autonomous driving is an example of algorithms that are likely to change in the near future. Reasons for potential updates can vary, from higher accuracy and energy efficiency to legal considerations. It is difficult to predict whether the fixed hardware architecture of a vehicle currently on the road will remain suitable for an updated algorithm. For example, we are witnessing a shift in neural networks from CNNs to Transformers right now, which will presumably also extend to embedded systems in the future.

It must be noted that FPGAs have significantly lower clock rates and occupy more area than ASICs for equivalent logic. This introduces a critical question for system design: which components should be implemented in reconfigurable logic, where performance overheads might be compensated by parallelism? Conversely, which components are better suited for a generic (GPP) or highly specialized (ASIC) implementation? In this work, these trade-offs were made manually. Future research could investigate the development of tools to automate these decisions. Determining which architectural components provide a lasting flexibility benefit when made reconfigurable is a significant challenge. A valuable first step would be a broad study of the evolving similarities and differences across common compute workloads as input.

Based on this, tools could make a prediction about the influence on the adaptivity of certain components.

In the field of dataflow architectures, like many others in academia and industry, our approach relies on a systolic array of tightly coupled Processing Elements (PEs). We demonstrated that at the accelerator tile level, the implemented algorithm can be significantly altered after silicon hardening with only a small amount of reconfigurable logic. For instance, our architecture could easily switch from convolution to General Matrix Multiplication (GEMM), although this is due to the fact that both share the Multiply-Accumulate (MAC) as their atomic operation. This raises an open question: what set of operators is essential within each PE to maximize application generality? While today's available Coarse-Grained Reconfigurable Architectures (CGRAs) aim for broad applicability, this often comes at the cost of high resource overhead. Future work may also involve a DSE for common memory-intensive algorithms to determine the optimal operator set.

With the current advancements seen in computer architecture, run-time reconfigurable processors, i.e. a hard logic GPP with closely-coupled FPGA fabric, will presumably gain popularity in both datacenter and mobile applications. This will continue the trend of heterogeneous integration, as previously observed with loosely coupled FPGA System-on-Chips (SoCs), like the AMD Zynq series.

Bibliography

[1] Hadi **Esmaeilzadeh**, Emily **Blem**, Renee **St. Amant**, Karthikeyan **Sankaralingam**, and Doug **Burger**. "Dark Silicon and the End of Multicore Scaling." In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. New York, NY, USA: Association for Computing Machinery, June 2011, pp. 365–376. isbn:978-1-4503-0472-6. doi:10.1145/2000064.2000108

[2] Nestor **Maslej** et al. *Artificial Intelligence Index Report 2025*. Apr. 2025. doi:10.48550/arXiv.2504.07139

[3] Rishi **Bommasani** et al. *On the Opportunities and Risks of Foundation Models*. July 2022. doi:10.48550/arXiv.2108.07258

[4] Abraham **Gonzalez**, Aasheesh **Kolli**, Samira **Khan**, Sihang **Liu**, Vidushi **Dadu**, Sagar **Karandikar**, Jichuan **Chang**, Krste **Asanovic**, and Parthasarathy **Ranganathan**. "Profiling Hyperscale Big Data Processing." In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. Orlando FL USA: ACM, June 2023, pp. 1–16. doi:10.1145/3579371.3589082

[5] Mohamed **Shalan** and Tim **Edwards**. "Building OpenLANE: A 130nm OpenROAD-based Tapeout- Proven Flow : Invited Paper." In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. Nov. 2020, pp. 1–6

[6] David P. **Rodgers**. "Improvements in Multiprocessor System Design." In: *ACM SIGARCH Computer Architecture News* 13.3, June 1985, pp. 225–231. issn:0163-5964. doi:10.1145/327070.327215

[7] John L. **Hennessy**. *Computer Architecture: A Quantitative Approach*. Sixth edition. Cambridge, MA: Morgan Kaufmann Publishers, 2019. isbn:978-0-12-811905-1

[8] David A. **Patterson** and John L. **Hennessy**. *Computer Organization and Design: The Hardware Software Interface*. RISC-V edition, Second edition. Cambridge, MA: Elsevier, Morgan Kaufmann Publishers, 2021. isbn:978-0-12-820331-6

[9] David E. **Culler**. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco : Morgan Kaufmann Publishers, 1999. isbn:978-1-55860-343-1

[10] Randal E. **Bryant** and David Richard **O'Hallaron**. *Computer Systems: A Programmer's Perspective*. Third edition, global edition. Always Learning. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2016. isbn:978-1-292-10176-7

[11] Philippe **Charles**, Christian **Grothoff**, Vijay **Saraswat**, Christopher **Donawa**, Allan **Kielstra**, Kemal **Ebcioglu**, Christoph **von Praun**, and Vivek **Sarkar**. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing." In: *SIGPLAN Not.* 40.10, Oct. 2005, pp. 519–538. issn:0362-1340. doi:10.1145/1103845.1094852

[12] Giovanni De **Micheli** and Luca **Benini**. *Networks on Chips: Technology and Tools*. Elsevier, Aug. 2006. isbn:978-0-08-047356-7

[13] Nidhi **Anantharajaiah**, Fabian **Lesniak**, Tanja **Harbaum**, and Juergen **Becker**. "Reinforcement Learning Enabled Multi-Layered NoC for Mixed Criticality Systems." In: *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Dec. 2023, pp. 38–44. doi:10.1109/MCSoC60832.2023.00014

[14] Leonard **Masing**, Akshay **Srivatsa**, Fabian **Kreß**, Nidhi **Anantharajaiah**, Andreas **Herkersdorf**, and Jürgen **Becker**. "In-NoC Circuits for Low-Latency Cache Coherence in Distributed Shared-Memory Architectures." In: *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Sept. 2018, pp. 138–145. doi:10.1109/MCSoC2018.2018.00033

[15] H. T. **Kung** and Charles Eric **Leiserson**. *Systolic Arrays for (VLSI)*. Carnegie-Mellon University, Department of Computer Science, 1978

[16] Christophe **Bobda**. *Introduction to Reconfigurable Computing: Architectures, Algorithms and Applications*. Dordrecht: Springer, 2007. isbn:978-1-4020-6088-5

[17] Florian **Schmaus**, Sebastian **Maier**, Tobias **Langer**, Jonas **Rabenstein**, Timo **Hönig**, Wolfgang **Schröder-Preikschat**, Lars **Bauer**, and Jörg **Henkel**. "System Software for Resource Arbitration on Future Many- Architectures." In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2020, pp. 967–975. doi:10.1109/IPDPSW50202.2020.00160

[18] Kevin **Lepak**, Gerry **Talbot**, Sean **White**, Noah **Beck**, Sam **Naffziger**, and Kevin **Lepak**. *The Next Generation AMD Enterprise Server Product Architecture*. Aug. 2017

[19] Wen-mei W. **Hwu**, David B. **Kirk**, and Izzat El **Hajj**. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, May 2022. isbn:978-0-323-98463-8

[20] **Modal Labs**. *GPU Glossary*. Feb. 2025

[21] **NVIDIA Corporation**. *NVIDIA RTX Blackwell GPU Architecture*. https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf. Whitepaper. 2025

[22] Leibo **Liu**, Jianfeng **Zhu**, Zhaoshi **Li**, Yanan **Lu**, Yangdong **Deng**, Jie **Han**, Shouyi **Yin**, and Shaojun **Wei**. "A Survey of Coarse-Grained Reconfigurable Architecture and Design." In: *ACM Computing Surveys* 52.6, 2020, pp. 1–39. issn:0360-0300. doi:10.1145/3357375

[23] R.W. **Hartenstein**, A.G. **Hirschbiel**, and M. **Weber**. "Xputers: Very High Throughput by Innovative Computing Principles." In: *Proceedings of the 5th Jerusalem Conference on Information Technology, 1990. 'Next Decade in Information Technology'*. Oct. 1990, pp. 43–50. doi:10.1109/JCIT.1990.128268

[24] Jason **Anderson**, Rami **Beidas**, Vimal **Chacko**, Hsuan **Hsiao**, Xiaoyi **Ling**, Omar **Ragheb**, Xinyuan **Wang**, and Tianyi **Yu**. "CGRA-ME: An Open-Source Framework for CGRA Architecture and CAD Research." In: *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2021, pp. 156–162. doi:10.1109/ASAP52443.2021.00030

[25] Stephen M. **Trimberger**. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology." In: *Proceedings of the IEEE* 103.3, Mar. 2015, pp. 318–331. issn:1558-2256. doi:10.1109/JPROC.2015.2392104

[26] Henry **Wong**, Vaughn **Betz**, and Jonathan **Rose**. "Comparing FPGA vs. Custom Cmos and the Impact on Processor Microarchitecture." In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Fpga '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 5–14. isbn:978-1-4503-0554-9. doi:10.1145/1950413.1950419

[27] Frank **Rosenblatt**. *Principles Of Neurodynamics*. 1962

[28] David E. **Rumelhart**, Geoffrey E. **Hinton**, and Ronald J. **Williams**. "Learning Representations by Back-Propagating Errors." In: *Nature* 323.6088, Oct. 1986, pp. 533–536. issn:1476-4687. doi:10.1038/323533a0

[29] Alex **Krizhevsky**, Ilya **Sutskever**, and Geoffrey E. **Hinton**. "ImageNet Classification with Deep Convolutional Neural Networks." In: *Commun. ACM* 60.6, May 2017, pp. 84–90. issn:0001-0782. doi:10.1145/3065386

[30] Ian **Goodfellow**, Yoshua **Bengio**, and Aaron **Courville**. *Deep Learning*. MIT Press, Nov. 2016. isbn:978-0-262-03561-3

[31] Ashish **Vaswani**, Noam **Shazeer**, Niki **Parmar**, Jakob **Uszkoreit**, Llion **Jones**, Aidan N. **Gomez**, Lukasz **Kaiser**, and Illia **Polosukhin**. *Attention Is All You Need*. Aug. 2023. doi:10.48550/arXiv.1706.03762

[32] Georgios **Kourounis**, Ali Ahmed **Elmahmudi**, Brian **Thomson**, James **Hunter**, Hassan **Ugail**, and Colin **Wilson**. "Computer Image Analysis with Artificial Intelligence: A Practical Introduction to Convolutional Neural Networks for Medical Professionals." In: *Postgraduate Medical Journal* 99.1178, Dec. 2023, pp. 1287–1294. issn:0032-5473. doi:10.1093/postmj/qgad095

[33] David **Patterson**, Joseph **Gonzalez**, Quoc **Le**, Chen **Liang**, Lluis-Miquel **Munguia**, Daniel **Rothchild**, David **So**, Maud **Texier**, and Jeff **Dean**. *Carbon Emissions and Large Neural Network Training*. Apr. 2021. doi:10.48550/arXiv.2104.10350

[34] Emma **Strubell**, Ananya **Ganesh**, and Andrew **McCallum**. *Energy and Policy Considerations for Deep Learning in NLP*. June 2019. doi:10.48550/arXiv.1906.02243

[35] Vivienne **Sze**, Yu-Hsin **Chen**, Tien-Ju **Yang**, and Joel S. **Emer**. *Efficient Processing of Deep Neural Networks*. Morgan & Claypool Publishers, June 2020. isbn:978-1-68173-832-1

[36] Michael A. **Nielsen**. *Neural Networks and Deep Learning*. Determination Press, 2015

[37] Stuart Jonathan **Russell** and Peter **Norvig**. *Artificial Intelligence: A Modern Approach*. Prentice Hall/Pearson Education, 2003. isbn:978-0-13-790395-5

[38] Y. **Lecun**, L. **Bottou**, Y. **Bengio**, and P. **Haffner**. "Gradient-Based Learning Applied to Document Recognition." In: *Proceedings of the IEEE* 86.11, Nov. 1998, pp. 2278–2324. issn:00189219. doi:10.1109/5.726791

[39] Kaiming **He**, Xiangyu **Zhang**, Shaoqing **Ren**, and Jian **Sun**. "Deep Residual Learning for Image Recognition." In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Las Vegas, NV, USA: IEEE, June 2016, pp. 770–778. isbn:978-1-4673-8851-1. doi:10.1109/CVPR.2016.90

[40] Sylvia **Hochstuhl**, Niklas **Pfeffer**, Antje **Thiele**, Horst **Hammer**, and Stefan **Hinz**. "Your Input Matters—Comparing Real-Valued PolSAR Data Representations for CNN-Based Segmentation." In: *Remote Sensing* 15.24, Jan. 2023, p. 5738. issn:2072-4292. doi:10.3390/rs15245738

[41] Sparsh **Mittal**. "A Survey of FPGA-based Accelerators for Convolutional Neural Networks." In: *Neural Comput. Appl.* 32.4, Feb. 2020, pp. 1109–1139. issn:0941-0643. doi:10.1007/s00521-018-3761-1

[42] Tim **Hotfilter**, Julian **Hoefer**, Fabian **Kreß**, Fabian **Kempf**, Leonhard **Kraft**, Tanja **Harbaum**, and Jürgen **Becker**. "A Hardware-Centric Approach to Increase and Prune Regular Activation Sparsity in CNNs." In: *2023 IEEE 5th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. Hangzhou, China: IEEE, June 2023, pp. 1–5. isbn:979-8-3503-3267-4. doi:10.1109/AICAS57966.2023.10168566

[43] Misha **Denil**, Babak **Shakibi**, Laurent **Dinh**, Marc'Aurelio **Ranzato**, and Nando de **Freitas**. *Predicting Parameters in Deep Learning*. Oct. 2014. doi:10.48550/arXiv.1306.0543

[44] Markus **Nagel**, Marios **Fournarakis**, Rana Ali **Amjad**, Yelysei **Bondarenko**, Mart **van Baalen**, and Tijmen **Blankevoort**. *A White Paper on Neural Network Quantization*. June 2021. doi:10.48550/arXiv.2106.08295

[45] Russell **Tessier**, Kenneth **Pocek**, and André **DeHon**. "Reconfigurable Computing Architectures." In: *Proceedings of the IEEE* 103.3, 2015, pp. 332–354. doi:10.1109/JPROC.2014.2386883

[46] Muhammad **Ali** and Diana **Göhringer**. "Application Specific Instruction-Set Processors for Machine Learning Applications." In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022, pp. 1–4. doi:10.1109/ICFPT56656.2022.9974187

[47] Tanja **Harbaum**, Christoph **Schade**, Marvin **Damschen**, Carsten **Tradowsky**, Lars **Bauer**, Jörg **Henkel**, and Jürgen **Becker**. "Auto-SI: An Adaptive Reconfigurable Processor with Run-Time Loop Detection and Acceleration." In: *2017 30th IEEE International System-on-Chip Conference (SOCC)*. Sept. 2017, pp. 153–158. doi:10.1109/SOCC.2017.8226027

[48] Matteo **Perotti**, Michele **Raeber**, Mattia **Sinigaglia**, Matheus **Cavalcante**, Davide **Rossi**, and Luca **Benini**. "Spatzformer: An Efficient Reconfigurable Dual-Core RISC-V V Cluster for Mixed Scalar-Vector Workloads." In: *2024 IEEE 35th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Hong Kong, Hong Kong: IEEE, July 2024, pp. 172–173. isbn:979-8-3503-4963-4. doi:10.1109/ASAP61560.2024.00042

[49] Dilshan **Kumarathunga**, Omega **Gamage**, Asitha **Samarasinghe**, Nipuna **Saranga**, Ranga **Rodrigo**, and Ajith **Pasqual**. "VLIW Based Runtime Reconfigurable Machine Vision Coprocessor Architecture for Edge Computing." In: *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. New York, NY, USA: IEEE, July 2019, pp. 103–106. isbn:978-1-7281-1601-3. doi:10.1109/ASAP.2019.00-22

[50] Ahmed **Kamaleldin** and Diana **Göhringer**. "AGILER: An Adaptive Heterogeneous Tile-Based Many-Core Architecture for RISC-v Processors." In: *IEEE access : practical innovations, open solutions* 10, 2022, pp. 43895–43913. doi:10.1109/ACCESS.2022.3168686

[51] Jian **Weng**, Sihao **Liu**, Zhengrong **Wang**, Vidushi **Dadu**, and Tony **Nowatzki**. "A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms." In: *2020 IEEE International*

Symposium on High Performance Computer Architecture (HPCA). Feb. 2020, pp. 703–716. doi: 10.1109/HPCA47549.2020.00063

[52] Hyoukjun **Kwon**, Ananda **Samajdar**, and Tushar **Krishna**. "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects." In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. Asplos '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 461–475. isbn:978-1-4503-4911-6. doi:10.1145/3173162.3173176

[53] Yu-Hsin **Chen**, Tushar **Krishna**, Joel S. **Emer**, and Vivienne **Sze**. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks." In: *IEEE Journal of Solid-State Circuits* 52.1, Jan. 2017, pp. 127–138. issn:0018-9200,1558-173X. doi:10.1109/JSSC.2016.2616357

[54] Rui **Xu** et al. "A Survey of Design and Optimization for Systolic Array-Based DNN Accelerators." In: *Acm Computing Surveys* 56.1, Aug. 2023. issn:0360-0300. doi:10.1145/3604802

[55] Ensieh **Aliagha** and Diana **Göhringer**. "Energy Efficient Design of Coarse-Grained Reconfigurable Architectures: Insights, Trends and Challenges." In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022, pp. 1–11. doi:10.1109/ICFPT56656.2022.9974339

[56] Yiran **Chen** et al. "A Survey of Accelerator Architectures for Deep Neural Networks." In: vol. 6. Engineering. 2020, pp. 264–274. doi:10.1016/j.eng.2020.01.007

[57] Albert **Reuther**, Peter **Michaleas**, Michael **Jones**, Vijay **Gadepally**, Siddharth **Samsi**, and Jeremy **Kepner**. "Lincoln AI Computing Survey (LAICS) Update." In: *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. Boston, MA, USA: IEEE, Sept. 2023, pp. 1–7. isbn:979-8-3503-0860-0. doi:10.1109/HPEC58863.2023.10363568

[58] Alejandro **Rico** et al. "AMD XDNA NPU in Ryzen AI Processors." In: *IEEE Micro* 44.6, Nov. 2024, pp. 73–82. issn:1937-4143. doi:10.1109/MM.2024.3423692

[59] Norman P. **Jouppi** et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, June 2017, pp. 1–12. isbn:978-1-4503-4892-8. doi:10.1145/3079856.3080246

[60] **NVIDIA Corporation**. *Maximizing Deep Learning Performance on NVIDIA Jetson Orin*. https://developer.nvidia.com/blog/maximizing-deep-learning-performance-on-nvidia-jetson-orin-with-dla/. 2023

[61] **NVIDIA Corporation**. *Hardware Architectural Specification*. http://nvdla.org/hw/v1/hwarch.html. 2024

[62] Hassan **Afzali-Kusha** and Massoud **Pedram**. "X-NVDLA: Runtime Accuracy Configurable NVDLA Based on Applying Voltage Overscaling to Computing and Memory Units." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 70.5, May 2023, pp. 1989–2002. issn:1549-8328,1558-0806. doi:10.1109/TCSI.2023.3247743

[63] Yakun Sophia **Shao** et al. "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 14–27. isbn:978-1-4503-6938-1. doi:10.1145/3352460.3358302

[64] Hyunbin **Park** and Shiho **Kim**. "Overviewing AI-dedicated Hardware for on-Device AI in Smartphones." In: *Artificial Intelligence and Hardware Accelerators*. Ed. by Ashutosh **Mishra**, Jaekwang **Cha**, Hyunbin **Park**, and Shiho **Kim**. Cham: Springer International Publishing, 2023, pp. 127–150. isbn:978-3-031-22170-5. doi:10.1007/978-3-031-22170-5_4

[65] **Hailo Technologies Ltd.** *Hailo-8™ Century High Performance Pcie Cards*. https://hailo.ai/wp-content/uploads/2023/10/Hailo-8-Century-PCIe-Product-Brief-Rev1.1.pdf. 2023

[66] Christoffer **Åleskog**, Håkan **Grahn**, and Anton **Borg**. "Recent Developments in Low-Power AI Accelerators: A Survey." In: *Algorithms* 15.11, Nov. 2022, p. 419. issn:1999-4893. doi:10.3390/a15110419

[67] Dennis Agyemanh Nana **Gookyi**, Eunchong **Lee**, Kyungho **Kim**, Sung-Joon **Jang**, and Sang-Seol **Lee**. "Deep Learning Accelerators' Configuration Space Exploration Effect on Performance and Resource Utilization: A Gemmini Case Study." In: *Sensors* 23.5, Feb. 2023, p. 2380. issn:1424-8220. doi:10.3390/s23052380

[68] Chan **Park** et al. “Roofline-Model-Based Design Space Exploration for Dataflow Techniques of CNN Accelerators.” In: *IEEE access : practical innovations, open solutions* 8, 2020, pp. 172509–172523. doi:10.1109/ACCESS.2020.3025550

[69] Hasan **Genc** et al. “Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration.” In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. Dec. 2021, pp. 769–774. doi:10.1109/DAC18074.2021.9586216

[70] Alon **Amid** et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs.” In: *IEEE Micro* 40.4, July 2020, pp. 10–21. issn:1937-4143. doi:10.1109/MM.2020.2996616

[71] Fengbin **Tu**, Shouyi **Yin**, Peng **Ouyang**, Shibin **Tang**, Leibo **Liu**, and Shaojun **Wei**. “Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns.” In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25.8, Aug. 2017, pp. 2220–2233. issn:1557-9999. doi:10.1109/TVLSI.2017.2688340

[72] Yu-Hsin **Chen**, Tien-Ju **Yang**, Joel S. **Emer**, and Vivienne **Sze**. “Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices.” In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2, June 2019, pp. 292–308. issn:2156-3357,2156-3365. doi:10.1109/JETCAS.2019.2910232

[73] Hyungmin **Cho**. “RiSA: A Reinforced Systolic Array for Depthwise Convolutions and Embedded Tensor Reshaping.” In: *ACM Trans. Embed. Comput. Syst.* 20.5s, 2021. issn:1539-9087. doi:10.1145/3476984

[74] Mike **O’Connor**, Niladrish **Chatterjee**, Donghyuk **Lee**, John **Wilson**, Aditya **Agrawal**, Stephen W. **Keckler**, and William J. **Dally**. “Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems.” In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-50 ’17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 41–54. isbn:978-1-4503-4952-9. doi:10.1145/3123939.3124545

[75] Marcian **Cirstea**, Khaled **Benkrid**, Andrei **Dinu**, Romeo **Ghiriti**, and Dorin **Petreus**. “Digital Electronic System-on-Chip Design: Methodologies, Tools, Evolution, and Trends.” In: *Micromachines* 15.2, Feb. 2024, p. 247. issn:2072-666X. doi:10.3390/mi15020247

[76] Sebastián **Valladares**, Mayerly **Toscano**, Rodrigo **Tufiño**, Paulina **Morillo**, and Diego **Vallejo-Huanga**. “Performance Evaluation of the Nvidia Jetson Nano Through a Real-Time Machine Learning Application.” In: *Intelligent Human Systems Integration 2021*. Ed. by Dario **Russo**, Tareq **Ahram**, Waldemar **Karwowski**, Giuseppe **Di Bucchianico**, and Redha **Taiar**. Vol. 1322. Cham: Springer International Publishing, 2021, pp. 343–349. isbn:978-3-030-68016-9. doi:10.1007/978-3-030-68017-6_51

[77] Matheus **Cavalcante**, Matteo **Perotti**, Samuel **Riedel**, and Luca **Benini**. *Spatz: Clustering Compact RISC-V-Based Vector Units to Maximize Computing Efficiency*. Sept. 2023. doi:10.48550/arXiv.2309.10137

[78] Samuel **Williams**, Andrew **Waterman**, and David **Patterson**. “Roofline: An Insightful Visual Performance Model for Multicore Architectures.” In: *Communications of the ACM* 52.4, Apr. 2009, pp. 65–76. issn:0001-0782,1557-7317. doi:10.1145/1498765.1498785

[79] Stephen M. **Smith** and J. Michael **Brady**. “SUSAN — A New Approach to Low Level Image Processing.” In: *International Journal of Computer Vision* 23.1, May 1997, pp. 45–78. issn:1573-1405. doi:10.1023/A:1007963824710

[80] Alex **Peleg**, Sam **Wilkie**, and Uri **Weiser**. “Intel MMX for Multimedia PCs.” In: *Communications of the ACM* 40.1, Jan. 1997, pp. 24–38. issn:0001-0782,1557-7317. doi:10.1145/242857.242865

[81] Shahzad **Ahmed**, Sohaib **Abdullah**, and Sung Ho **Cho**. “Advancements in Radar Point Cloud Processing for Macro Human Movements in Healthcare and Assisted Living Domains: A Review.” In: *IEEE Sensors Journal* 24.22, Nov. 2024, pp. 36287–36305. issn:1530-437X,1558-1748,2379-9153. doi:10.1109/JSEN.2024.3452110

[82] Matthew **Naylor** and Colin **Runciman**. “The Reduceron Reconfigured and Re-Evaluated.” In: *Journal of Functional Programming* 22.4-5, Sept. 2012, pp. 574–613. issn:0956-7968,1469-7653. doi:10.1017/S0956796812000214

[83] Yu **Wang**, James C. **Hoe**, and Eriko **Nurvitadhi**. "Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform." In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. San Diego, CA, USA: IEEE, Apr. 2019, pp. 136–144. isbn:978-1-7281-1131-5. doi:10.1109/FCCM.2019.00028

[84] Sven **Rheindt**, Andreas **Fried**, Oliver **Lenke**, Lars **Nolte**, Thomas **Wild**, and Andreas **Herkersdorf**. "NEMESYS: Near-Memory Graph Copy Enhanced System-Software." In: *Proceedings of the International Symposium on Memory Systems*. Washington District of Columbia USA: ACM, Sept. 2019, pp. 3–18. isbn:978-1-4503-7206-0. doi:10.1145/3357526.3357545

[85] Jo **Van Hoey**. *Beginning X64 Assembly Programming: From Novice to AVX Professional*. Berkeley, CA: Apress, 2019. isbn:978-1-4842-5075-4. doi:10.1007/978-1-4842-5076-1

[86] **MIPS Technologies, Inc.** and Charles **Price**. *MIPS IV Instruction Set*. Sept. 1995

[87] **RISC-V International**. *The RISC-V Instruction Set Manual Vol. I: Unprivileged ISA*. Dec. 2019

[88] **Arm Limited**. *AMBA AXI Protocol Specification*. Sept. 2023

[89] Debendra **Das Sharma**, Gerald **Pasdast**, Zhiguo **Qian**, and Kemal **Aygun**. "Universal Chiplet Interconnect Express (UCIe): An Open Industry Standard for Innovations With Chiplets at Package Level." In: *IEEE Transactions on Components, Packaging and Manufacturing Technology* 12.9, Sept. 2022, pp. 1423–1431. issn:2156-3950,2156-3985. doi:10.1109/TCPMT.2022.3207195

[90] M. **Damschen**, M. **Rapp**, L. **Bauer**, and J. **Henkel**. *I-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems*. Jan. 2020. doi:10.1007/978-3-030-16949-7_1

[91] **Aeroflex Gaisler AB**. *LEON3: Multiprocessing CPU Core*. 2010

[92] Lars **Bauer**, Muhammad **Shafique**, and Jorg **Henkel**. "RISPP: A Run-Time Adaptive Reconfigurable Embedded Processor." In: *2009 International Conference on Field Programmable Logic and Applications*. 2009, pp. 725–726. doi:10.1109/FPL.2009.5272323

[93] G. **Koo**, K. K. **Matam**, T. **I**, H. V. K. G. **Narra**, J. **Li**, H. **Tseng**, S. **Swanson**, and M. **Annavaram**. "Summarizer: Trading Communication with Computing Near Storage." In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2017, pp. 219–231

[94] Yueting **Li**, Tianshuo **Bai**, Xinyi **Xu**, Yundong **Zhang**, Bi **Wu**, Hao **Cai**, Biao **Pan**, and Weisheng **Zhao**. "A Survey of MRAM-Centric Computing: From Near Memory to In Memory." In: *IEEE Transactions on Emerging Topics in Computing* 11.2, Apr. 2023, pp. 318–330. issn:2168-6750,2376-4562. doi:10.1109/TETC.2022.3214833

[95] W. **Kang**, H. **Wang**, Z. **Wang**, Y. **Zhang**, and W. **Zhao**. "In-Memory Processing Paradigm for Bitwise Logic Operations in STT–MRAM." in: *IEEE Transactions on Magnetics* 53.11, 2017, pp. 1–4. doi:10.1109/TMAG.2017.2703863

[96] P. **Gu**, X. **Xie**, Y. **Ding**, G. **Chen**, W. **Zhang**, D. **Niu**, and Y. **Xie**. "iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture." In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 804–817. doi:10.1109/ISCA45697.2020.00071

[97] S. **Corda**, B. **Veenboer**, A. J. **Awan**, A. **Kumar**, R. **Jordans**, and H. **Corporaal**. "Near Memory Acceleration on High Resolution Radio Astronomy Imaging." In: *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. 2020, pp. 1–6. doi:10.1109/MECO49872.2020.9134089

[98] A. **Grudnitsky**, L. **Bauer**, and J. **Henkel**. "COREFAB: Concurrent Reconfigurable Fabric Utilization in Heterogeneous Multi-Core Systems." In: *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 2014, pp. 1–10. doi:10.1145/2656106.2656119

[99] S. **Wong**, S. **Vassiliadis**, and S. **Cotofana**. "A Sum of Absolute Differences Implementation in FPGA Hardware." In: *Proceedings. 28th Euromicro Conference*. 2002, pp. 183–188. doi:10.1109/EURMIC.2002.1046155

[100] M.R. **Guthaus**, J.S. **Ringenberg**, D. **Ernst**, T.M. **Austin**, T. **Mudge**, and R.B. **Brown**. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite." In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. 2001, pp. 3–14. doi:10.1109/WWC.2001.990739

[101] Banraplang **Jyrwa** and Roy **Paily**. "An Area-Throughput Efficient FPGA Implementation of the Block Cipher AES Algorithm." In: *2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies*. Trivandrum, Kerala: IEEE, Dec. 2009, pp. 328–332. isbn:978-1-4244-5321-4. doi:10.1109/ACT.2009.88

[102] Xuanhua **Li** and Donald **Yeung**. "Application-Level Correctness and Its Impact on Fault Tolerance." In: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. 2007, pp. 181–192. doi:10.1109/HPCA.2007.346196

[103] **Standard Performance Evaluation Corporation** (**SPEC**) and **Robert Munafo**. *The SPEC Benchmarks*. Apr. 2022

[104] Jie **Han** and Michael **Orshansky**. "Approximate Computing: An Emerging Paradigm for Energy-Efficient Design." In: *2013 18th IEEE European Test Symposium* (*ETS*). 2013, pp. 1–6. doi:10.1109/ETS.2013.6569370

[105] Vojtech **Mrazek**, Radek **Hrbacek**, Zdenek **Vasicek**, and Lukas **Sekanina**. "EvoApprox8b: Library of Approximate Adders and Multipliers for Circuit Design and Benchmarking of Approximation Methods." In: *Design, Automation & Test in Europe Conference & Exhibition* (*DATE*), *2017*. Lausanne: IEEE, Mar. 2017, pp. 258–261. isbn:978-3-9815370-8-6. doi:10.23919/DATE.2017.7926993

[106] Muhammad **Shafique**, Waqas **Ahmad**, Rehan **Hafiz**, and Jörg **Henkel**. "A Low Latency Generic Accuracy Configurable Adder." In: *Proceedings of the 52nd Annual Design Automation Conference*. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015. isbn:978-1-4503-3520-1. doi:10.1145/2744769.2744778

[107] Waqar **Ahmad**, Berke **Ayrancioglu**, and Ilker **Hamzaoglu**. "Low Error Efficient Approximate Adders for FPGAs." In: *IEEE Access* 9, 2021, pp. 117232–117243. doi:10.1109/ACCESS.2021.3107370

[108] Salim **Ullah**, Sanjeev Sripadraj **Murthy**, and Akash **Kumar**. "SMApproxLib: Library of FPGA-based Approximate Multipliers." In: *2018 55th ACM/ESDA/IEEE Design Automation Conference* (*DAC*). San Francisco, CA: IEEE, June 2018, pp. 1–6. isbn:978-1-5386-4114-9. doi:10.1109/DAC.2018.8465845

[109] Shangshang **Yao** and Liang **Zhang**. "Hardware-Efficient FPGA-Based Approximate Multipliers for Error-Tolerant Computing." In: *2022 International Conference on Field-Programmable Technology* (*ICFPT*). 2022, pp. 1–8. doi:10.1109/ICFPT56656.2022.9974399

[110] Justine **Bonnot**, Alexandre **Mercat**, Erwan **Nogues**, and Daniel **Ménard**. "Approximate Computing at the Algorithmic Level." In: *Approximate Computing Techniques: From Component- to Application-Level*. Ed. by Alberto **Bosio**, Daniel **Ménard**, and Olivier **Sentieys**. Cham: Springer International Publishing, 2022, pp. 109–142. isbn:978-3-030-94705-7. doi:10.1007/978-3-030-94705-7_5

[111] John **Canny**. "A Computational Approach to Edge Detection." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6, 1986, pp. 679–698. doi:10.1109/TPAMI.1986.4767851

[112] Alexandru **Amaricai** and Oana **Boncalo**. "SRT Radix-2 Dividers with (5,4) Redundant Representation of Partial Remainder." In: *2013 NORCHIP*. Vilnius, Lithuania: IEEE, Nov. 2013, pp. 1–5. isbn:978-1-4799-1647-4. doi:10.1109/NORCHIP.2013.6702028

[113] Manfred **Broy**. "Challenges in Automotive Software Engineering." In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 33–42. isbn:1-59593-375-1. doi:10.1145/1134285.1134292

[114] Simon **Fürst**. "Challenges in the Design of Automotive Software." In: *2010 Design, Automation Test in Europe Conference Exhibition* (*DATE 2010*). 2010, pp. 256–258. doi:10.1109/DATE.2010.5457201

[115] David **Barton**, Philipp **Gönnheimer**, Florian **Schade**, Christopher **Ehrmann**, Jürgen **Becker**, and Jürgen **Fleischer**. "Modular Smart Controller for Industry 4.0 Functions in Machine Tools." In: *Procedia CIRP* 81, 2019, pp. 1331–1336. issn:2212-8271. doi:10.1016/j.procir.2019.04.022

[116] Paul **Barham**, Boris **Dragovic**, Keir **Fraser**, Steven **Hand**, Tim **Harris**, Alex **Ho**, Rolf **Neugebauer**, Ian **Pratt**, and Andrew **Warfield**. "Xen and the Art of Virtualization." In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. New

York, NY, USA: Association for Computing Machinery, 2003, pp. 164–177. isbn:1-58113-757-5. doi:10.1145/945445.945462

[117] Victor **Bandur**, Gehan **Selim**, Vera **Pantelic**, and Mark **Lawford**. "Making the Case for Centralized Automotive E/E Architectures." In: *IEEE Transactions on Vehicular Technology* 70.2, Feb. 2021, pp. 1230–1245. issn:0018-9545,1939-9359. doi:10.1109/TVT.2021.3054934

[118] Michael S. **Tsirkin**, Cornelia **Huck**, and **OASIS**, eds. *Virtual I/O Device (VIRTIO) Version 1.1*. Vol. OASIS Committee Specification 01. Apr. 2019

[119] Sara **Alonso**, Jesús **Lázaro**, Jaime **Jiménez**, Leire **Muguira**, and Alejandro **Largacha**. "Analysing the Interference of Xen Hypervisor in the Network Speed." In: *2020 XXXV Conference on Design of Circuits and Integrated Systems (DCIS)*. 2020, pp. 1–6. doi:10.1109/DCIS51330.2020.9268648

[120] Dominik **Reinhardt**, Maximilian **Güntner**, Markus **Kucera**, Thomas **Waas**, and Winfried **Kühnhauser**. "Mapping CAN-to-ethernet Communication Channels within Virtualized Embedded Environments." In: *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 2015, pp. 1–10. doi:10.1109/SIES.2015.7185064

[121] Oliver **Sander** et al. "Hardware Virtualization Support for Shared Resources in Mixed-Criticality Multicore Systems." In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. Dresden, Germany: IEEE Conference Publications, 2014, pp. 1–6. isbn:978-3-9815370-2-4. doi:10.7873/DATE.2014.081

[122] Sisu **Xi**, Chong **Li**, Chenyang **Lu**, and Christopher **Gill**. "Prioritizing Local Inter-Domain Communication in Xen." In: *2013 IEEE/ACM 21st International Symposium on Quality of Service (IWQoS)*. 2013, pp. 1–10. doi:10.1109/IWQoS.2013.6550267

[123] Pierre **Lucas**, Kevin **Chappuis**, Benjamin **Boutin**, Julian **Vetter**, and Daniel **Raho**. "VOSYSmonitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System." In: *2018 23rd Conference of Open Innovations Association (FRUCT)*. 2018, pp. 231–238. doi:10.23919/FRUCT.2018.8588018

[124] A. **Oliveira**, J. **Martins**, J. **Cabral**, A. **Tavares**, and S. **Pinto**. "TZ-VirtIO: Enabling Standardized Inter-Partition Communication in a Trustzone-Assisted Hypervisor." In: *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*. 2018, pp. 708–713. doi:10.1109/ISIE.2018.8433781

[125] Hasan **Fayyad**, Luc **Perneel**, and Martin **Timmerman**. "Full and Para-Virtualization with Xen: A Performance Comparison." In: *Journal of Emerging Trends in Computing and Information Sciences* Volume 10, Sept. 2013, pp. 719–727

[126] **Xen Project**. *XenStore Paths*. https://xenbits.xen.org/docs/4.13-testing/misc/xenstore-paths.html. 2020

[127] **Xen Project**. *Xl.Cfg - Xl Domain Configuration File Syntax*. https://xenbits.xen.org/docs/4.13-testing/man/xl.cfg.5.html. 2020

[128] **OpenAMP Project**. *RPMsg Messaging Protocol*. https://github.com/OpenAMP/open-amp. Oct. 2016

[129] Yosab **Bebawy**, Houssem **Guissouma**, Sebastian Vander **Maelen**, Janis **Kroger**, Georg **Hake**, Ingo **Stierand**, Martin **Franzle**, Eric **Sax**, and Axel **Hahn**. "Incremental Contract-based Verification of Software Updates for Safety-Critical Cyber-Physical Systems." In: *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*. Las Vegas, NV, USA: IEEE, Dec. 2020, pp. 1708–1714. isbn:978-1-7281-7624-6. doi:10.1109/CSCI51800.2020.00318

[130] Lihua **Gao**, Minyan **Lu**, Luyi **Li**, and Cong **Pan**. "A Survey of Software Runtime Monitoring." In: *2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. Beijing, China: IEEE, Nov. 2017, pp. 308–313. isbn:978-1-5386-0497-7. doi:10.1109/ICSESS.2017.8342921

[131] **Google, Inc.** *Google Benchmark – A Microbenchmark Support Library*. https://github.com/google/benchmark. 2020

[132] Richard **Barry**. *Mastering the FreeRTOS™ Real Time Kernel*. 1.1.0. Aug. 2024

[133] Eriko **Nurvitadhi**, **Jaewoong Sim**, David **Sheffield**, Asit **Mishra**, Srivatsan **Krishnan**, and Debbie **Marr**. "Accelerating Recurrent Neural Networks in Analytics Servers: Comparison of FPGA, CPU, GPU, and ASIC." in: *2016 26th International Conference on Field Programmable Logic*

and Applications (*FPL*). Lausanne, Switzerland: IEEE, Aug. 2016, pp. 1–4. isbn:978-2-8399-1844-2. doi:10.1109/FPL.2016.7577314

[134] Nidhi **Anantharajaiah**, Zhe **Zhang**, and Juergen **Becker**. "Multi-Layered NoCs with Adaptive Routing for Mixed Criticality Systems." In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by Steven **Derrien**, Frank **Hannig**, Pedro C. **Diniz**, and Daniel **Chillet**. Vol. 12700. Cham: Springer International Publishing, 2021, pp. 125–139. isbn:978-3-030-79024-0. doi:10.1007/978-3-030-79025-7_9

[135] Zeke **Wang**, Hongjing **Huang**, Jie **Zhang**, and Gustavo **Alonso**. "Shuhai: Benchmarking High Bandwidth Memory On FPGAS." in: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines* (*FCCM*). Fayetteville, AR, USA: IEEE, May 2020, pp. 111–119. isbn:978-1-7281-5803-7. doi:10.1109/FCCM48280.2020.00024

[136] Simen Gimle **Hansen**, Dirk **Koch**, and Jim **Torresen**. "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. May 2011, pp. 174–180. doi:10.1109/IPDPS.2011.139

[137] Christian **Szegedy** et al. "Going Deeper with Convolutions." In: *2015 IEEE Conference on Computer Vision and Pattern Recognition* (*CVPR*). 2015, pp. 1–9. doi:10.1109/CVPR.2015.7298594

[138] Andrew **Howard** et al. "Searching for MobileNetV3." In: *2019 IEEE/CVF International Conference on Computer Vision* (*ICCV*). 2019, pp. 1314–1324. doi:10.1109/ICCV.2019.00140

[139] Olga **Melnikova**, Irina **Hahanova**, and Karina **Mostovaya**. "Using Multi-FPGA Systems for ASIC Prototyping." In: *2009 10th International Conference - the Experience of Designing and Application of CAD Systems in Microelectronics*. 2009, pp. 237–239

[140] Nathan **Binkert** et al. "The Gem5 Simulator." In: *SIGARCH Comput. Archit. News* 39.2, Aug. 2011, pp. 1–7. issn:0163-5964. doi:10.1145/2024716.2024718

[141] Ayaz **Akram** and Lina **Sawalha**. "x86 Computer Architecture Simulators: A Comparative Study." In: *2016 IEEE 34th International Conference on Computer Design* (*ICCD*). Scottsdale, AZ, USA: IEEE, Oct. 2016, pp. 638–645. isbn:978-1-5090-5142-7. doi:10.1109/ICCD.2016.7753351

[142] Leonard **Masing**, Fabian **Lesniak**, and Jürgen **Becker**. "A Hybrid Prototyping Framework in a Virtual Platform Centered Design and Verification Flow." In: *IEEE Embedded Systems Letters* 13.1, 2021, pp. 1–4. doi:10.1109/LES.2020.2995084

[143] César **Sánchez** et al. "A Survey of Challenges for Runtime Verification from Advanced Application Domains (beyond Software)." In: *Formal Methods in System Design* 54.3, Nov. 2019, pp. 279–335. issn:1572-8102. doi:10.1007/s10703-019-00337-w

[144] Dimitry **Solet**, Sebastien **Pillement**, Jean-Luc **Béchennec**, Mikael **Briday**, and Sebastien **Faucou**. "HW-based Architecture for Runtime Verification of Embedded Software on SoPC Systems." In: *2018 NASA/ESA Conference on Adaptive Hardware and Systems* (*AHS*). 2018, pp. 249–256. doi:10.1109/AHS.2018.8541459

[145] Augusto **Hoppe**, Jürgen **Becker**, and Fernanda Lima **Kastensmidt**. "High-Speed Hardware Accelerator for Trace Decoding in Real-Time Program Monitoring." In: *2021 IEEE 12th Latin America Symposium on Circuits and System* (*LASCAS*). 2021, pp. 1–4. doi:10.1109/LASCAS51355.2021.9459137

[146] Adrian **Francalanza**, Jorge A. **Pérez**, and César **Sánchez**. "Runtime Verification for Decentralised and Distributed Systems." In: *Lectures on Runtime Verification: Introductory and Advanced Topics*. Ed. by Ezio **Bartocci** and Yliès **Falcone**. Cham: Springer International Publishing, 2018, pp. 176–210. isbn:978-3-319-75632-5. doi:10.1007/978-3-319-75632-5_6

[147] Fengwei **Zhang**, Kevin **Leach**, Angelos **Stavrou**, Haining **Wang**, and Kun **Sun**. "Using Hardware Features for Increased Debugging Transparency." In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 55–69. doi:10.1109/SP.2015.11

[148] Jong Chul **Lee**, Andrew S. **Gardner**, and Roman **Lysecky**. "Hardware Observability Framework for Minimally Intrusive Online Monitoring of Embedded Systems." In: *2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. 2011, pp. 52–60. doi:10.1109/ECBS.2011.13

[149] Alexandra **Listl**, Daniel **Mueller-Gritschneder**, Fabian **Kluge**, and Ulf **Schlichtmann**. "Emulation of an ASIC Power, Temperature and Aging Monitor System for FPGA Prototyping." In: *2018 IEEE 24th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2018. doi:10.1109/IOLTS.2018.8474284

[150] **Siemens Electronic Design Automation GmbH**. *proFPGA FPGA Prototyping System*. http://www.profpga.com. 2022

[151] **InfluxData Inc.** *InfluxDB Time Series Database*. http://influxdata.com. 2022

[152] **Raintank Inc.** *Grafana: Open Observability Platform*. http://grafana.com. 2022

[153] David H. **Bailey**. "NAS Parallel Benchmarks." In: *Encyclopedia of Parallel Computing*. Ed. by David **Padua**. Boston, MA: Springer US, 2011, pp. 1254–1259. isbn:978-0-387-09766-4. doi:10.1007/978-0-387-09766-4_133

Publications

[Les+24] Fabian **Lesniak**, Annina **Gutermann**, Tanja **Harbaum**, and Jürgen **Becker**. "Enhanced Accelerator Design for Efficient CNN Processing with Improved Row-Stationary Dataflow." In: *Proceedings of the Great Lakes Symposium on VLSI 2024*. GLSVLSI '24. Clearwater, FL, USA: Association for Computing Machinery, 2024, pp. 151–157. isbn:9798400706059. doi:10.1145/3649476.3658737

[LHB24] Fabian **Lesniak**, Tanja **Harbaum**, and Jürgen **Becker**. "Low-latency inter-domain communication on the Xen hypervisor." In: *16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip* (*MCSoC*) (Dec. 18–21, 2023). Singapore: Institute of Electrical and Electronics Engineers (IEEE), 2024, pp. 340–346. isbn:979-83-503-9362-0. doi:10.1109/MCSoC60832.2023.00057

[Ana+23b] Nidhi **Anantharajaiah**, Fabian **Lesniak**, Tanja **Harbaum**, and Jürgen **Becker**. "Reinforcement Learning Enabled Multi-Layered NoC for Mixed Criticality Systems." In: *16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip* (*MCSoC*) (Dec. 18–21, 2023). Singapore: Institute of Electrical and Electronics Engineers (IEEE), 2023, pp. 38–44. isbn:979-83-503-9361-3. doi:10.1109/MCSoC60832.2023.00014

[Ana+23c] Nidhi **Anantharajaiah**, Yunhe **Xu**, Fabian **Lesniak**, Tanja **Harbaum**, and Juergen **Becker**. "DREAM: Distributed Reinforcement Learning Enabled Adaptive Mixed-Critical NoC." in: *IEEE Computer Society Annual Symposium on VLSI* (*ISVLSI*) (June 20–23, 2023). Foz do Iguacu, Brasil: Institute of Electrical and Electronics Engineers (IEEE), 2023, pp. 1–6. isbn:979-83-503-2770-0. doi:10.1109/ISVLSI59464.2023.10238569

[Les+23] Fabian **Lesniak**, Nidhi **Anantharajaiah**, Tanja **Harbaum**, and Jürgen **Becker**. "Non-Intrusive Runtime Monitoring for Manycore Prototypes." In: *Proceedings of the DroneSE and RAPIDO: System Engineering for constrained embedded systems* (Jan. 17–18, 2023). Toulouse, France: Association for Computing Machinery (ACM), 2023, pp. 31–38. isbn:979-84-007-0045-3. doi:10.1145/3579170.3579262

[LHB23] Fabian **Lesniak**, Tanja **Harbaum**, and Jürgen **Becker**. "Approximate Accelerators: A Case Study using Runtime Reconfigurable Processors." In: *36th International System-on-Chip Conference* (*SOCC*) (Sept. 5–8, 2023). Santa Clara, CA, USA: Institute of Electrical and Electronics Engineers (IEEE), 2023, pp. 1–6. isbn:979-83-503-0011-6. doi:10.1109/SOCC58585.2023.10257090

[Gui+22] Houssem **Guissouma**, Carl Philipp **Hohl**, Fabian **Lesniak**, Marc **Schindewolf**, Jurgen **Becker**, and Eric **Sax**. "Lifecycle Management of Automotive Safety-Critical Over the Air Updates: A Systems Approach." In: *IEEE Access* 10, 2022, pp. 57696–57717. issn:2169-3536. doi:10.1109/ACCESS.2022.3176879

[LKB21] Fabian **Lesniak**, Fabian **Kreß**, and Jürgen **Becker**. "Transparent Near-Memory Computing with a Reconfigurable Processor." In: *17th International Symposium on Applied Reconfigurable Computing* (*ARC*) (June 29–July 1, 2021). Vol. 12700. Lecture Notes in Computer Science. Online: Springer Nature Switzerland, 2021, pp. 221–231. isbn:978-3-030-79024-0. doi:10.1007/978-3-030-79025-7_15

[MLB21a] Leonard **Masing**, Fabian **Lesniak**, and Jurgen **Becker**. "A Hybrid Prototyping Framework in a Virtual Platform Centered Design and Verification Flow." In: *IEEE embedded systems letters* 13.1, 2021. issn:1943-0663,1943-0671. doi:10.1109/LES.2020.2995084

[Ana+19] Nidhi **Anantharajaiah**, Fabian **Kempf**, Leonard **Masing**, Fabian Marc **Lesniak**, and Jürgen **Becker**. "Dynamic and scalable runtime block-based multicast routing for

networks on chips." In: *Proceedings of the 12th International Workshop on Network on Chip Architectures* (*NoCArc*) (Oct. 12–13, 2019). Columbus, OH, USA: Association for Computing Machinery (ACM), 2019, pp. 1–6. isbn:978-1-4503-6949-7. doi:10.1145/3356045.3360718

[MLB19] L. **Masing**, F. **Lesniak**, and J. **Becker**. "Hybrid Prototyping for Manycore Design and Validation." In: *15th International Symposium on Applied Reconfigurable Computing* (*ARC*) (Apr. 9–11, 2019). Vol. 11444. Lecture Notes in Computer Science. Darmstadt, Germany, 2019, pp. 319–333. isbn:978-3-030-17226-8. doi:10.1007/978-3-030-17227-5_23

[Wer+15] Stephan **Werner**, Leonard **Masing**, Fabian **Lesniak**, and Jurgen **Becker**. "Software-in-the-Loop simulation of embedded control applications based on Virtual Platforms." In: *25th International Conference on Field Programmable Logic and Applications* (*FPL*) (Sept. 2–4, 2015). London, United Kingdom: Institute of Electrical and Electronics Engineers (IEEE), 2015, Art.Nr. 7294020. isbn:978-0-9934280-0-5. doi:10.1109/FPL.2015.7294020

Supervised Student Work

[Kel25] Alexander **Keller**. "Automated Generation and Topological Verification of Address Descramblers for Custom SRAMs." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2025

[Sch25] Jakob **Schwarz**. "Laufzeitmodellierung für KI Beschleuniger zur Entwurfsraumexploration." Bachelor's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2025

[Kes24] Christopher **Kessler**. "Design of a Scalable High Performance Accelerator Platform." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2024

[Mül24] Rasmus **Müller-Both**. "Federated Reinforcement Learning for Resource Management for Edge Devices." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2024

[Tri24] Walid **Triki**. "Development of a mapping tool for adaptive AI accelerators." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2024

[Huo23] Johannes **Huober**. "Design of a Scalable Accelerator Architecture for Neural Networks." Mastger's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2023

[Lut23] Jonathan **Lutz**. "Design of a reconfigurable accelerator framework." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2023

[Wol23] Daniel **Wolf**. "Rekonfigurierbare approximative Beschleuniger." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2023

[Yin23] Keshuo **Ying**. "Automatic Generation of Approximated Hardware Accelerators." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2023

[Cao22] Zhenhao **Cao**. "Implementierung von dynamisch konfigurierbarer Approximation von Spezialinstruktionen." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2022

[Cha21] Jianxun **Chang**. "Implementing an AI Accelerator as a Special Instruction." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2021

[Dic21] Eugen **Dick**. "ARM-Debugger für den Xen-Hypervisor." Bachelor's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2021

[Kes21] Christopher **Kessler**. "Implementierung eines approximativen Beschleunigers." Bachelor's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2021

[Les21] Xavier **Lesage**. "Entwicklung eines dynamischen und flexiblen Speichercontrollers für Manycore- Prototypen." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2021

[Lie21] Simon **Liehmann**. "Dynamische CNN-Beschleunigung für den i-Core Prozessor." Bachelor's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2021

[Fre20] Antonia Alice **Frey**. "Distributed Realtime Monitoring of Manycores." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2020

[Kre20] Fabian **Kreß**. "Design eines prozessorbasierten Near-Memory-Beschleunigers." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2020

[Lei20] Simon **Leiner**. "Hypervisor-Gastüberwachung in Safety-Critical Systems." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2020

[Zhe19] Lihong **Zheng**. "Channel Estimation of Dynamic Intra-Tile Reallocatable Cache in Homogenous GPPs." Master's Thesis. Karlsruhe Institute of Technology, Institute for Information Processing Technologies, 2019

List of Figures

List of Tables