Reconfigurable Accelerators in the World of General-Purpose Computing

Dissertation

A thesis submitted to the
Faculty of Electrical Engineering, Computer Science and Mathematics
of
Paderborn University
in partial fulfillment of the requirements for the
degree of *Dr. rer. nat.*

by

Tobias Kenter

Paderborn, Germany
August 26, 2016

Acknowledgments

First and foremost, I would like to thank Prof. Dr. Christian Plessl for the advice and support during my research. As particularly helpful, I perceived his ability to communicate suggestions depending on the situation, either through open questions that give room to explore and learn, or through concrete recommendations that help to achieve results more directly.

Special thanks go also to Prof. Dr. Marco Platzner for his advice and support. I profited especially from his experience and ability to systematically identify the essence of challenges and solutions.

Furthermore, I would like to thank:

- Prof. Dr. João M. P. Cardoso, for serving as external reviewer for my dissertation.
- Prof. Dr. Friedhelm Meyer auf der Heide and Dr. Matthias Fischer for serving on my oral examination committee.
- All colleagues with whom I had the pleasure to work at the PC^2 and the Computer Engineering Group, researchers, technical and administrative staff. In a variation to one of our coffee kitchen puns, I'd like to state that *research without colleagues is possible, but pointless.* However, I'm not sure about the first part.
- My long-time office mates Lars Schäfers and Alexander Boschmann for particularly extensive discussions on our research and far beyond.
- Gavin Vaz, Heinrich Riebler and Achim Lösch for intensive and productive collaboration on joint research interests.
- The Bachelor and Master students and student assistants I have supervised, for their enthusiasm and for helping me to understand potential and limitations of the different approaches in their projects. The technical contributions of Henning Schmitz have particularly contributed to my research.
- The Deutsche Forschungsgemeinschaft (DFG) and the Intel Labs Braunschweig for funding different phases of my research in the Collaborative Research Centre (CRC) 901 *On-The-Fly Computing* and the Project *Multimodal Reconfigurable Processing Unit (MM-RPU).*

- The colleagues in the CRC 901 for insightful discussions.
- Michael Kauschke and Matthias Gries for contributing an industry perspective on my research.

Finally, I would like to thank my family for their encouragement and moral support. Special thanks go to Gaby Nordendorf. If anyone was ever more convinced than me that I would eventually complete this thesis, it was her.

Abstract

Growing and newly emerging computing workloads and markets keep pushing computer architecture forward. Power limitations and diminishing returns from wider and more parallel processors are driving the need for architectural innovations. By reducing overheads and customizing parallelism, specialized accelerators help to increase the performance of specific workloads efficiently. However, without programmability, they lack the flexibility for general-purpose computing and thus can't profit from shared costs among different workloads, users and markets. The architecture of field programmable gate arrays (FPGAs) combines programmability with a high potential for specialization for different workloads. The main obstacle for FPGA adoption in general-purpose computing is the lack of productive methods and tools for designers and maintainers of implementations running entirely or partially on FPGAs. In this work, by analyzing current approaches to tackle this productivity challenge along with their conceptual and practical trade-offs, we identify three pillars that can complement each other to jointly drive general-purpose adoption of FPGAs. These pillars combine, firstly, synthesis from parallel OpenCL designs, secondly, fast and automatic compilation targeting overlay architectures on FPGAs, and thirdly, the encapsulation of hand-optimized FPGA designs into application- or domain-specific libraries.

In this thesis, we focus on overlay architectures as one of theses paths towards productive FPGA design processes. A large variety of such architectures have been presented over the last years, but for most overlays it was poorly understood, which overheads they involve compared to custom designs implemented directly on FPGA resources. Our work quantifies such overheads with a diverse set of program loops from a state-of-the-art stereo-matching application for an instruction-programmable overlay. It demonstrates that the architecture can, despite overheads, serve as a practically usable accelerator, and identifies specific differences to fully customized FPGA designs that may help to reduce overheads through overlay customization. Even though one motivating aspect for related research on overlay architectures has been their potential for fast compilation or synthesis and for quick reconfiguration, in order to demonstrate their practical usefulness to raise productivity when aiming at FPGA acceleration, we had to go one step further with regard to the design entry. To this end, we present a fast tool flow that automatically extracts suitable loops from a high-level source or binary code and offloads them to a vector coprocessor realized as an FPGA overlay.

Besides the focus on productivity to foster FPGA adoption, in order to fit into established computing systems and markets, FPGAs need to architecturally coexist and cooperate with general-purpose processors. With a high-level performance estimation model that takes into account the interdependency of architectures and program designs, we explore the design space for such integrated systems. We highlight that integration of the memory hierarchy is not only helpful to reduce application design efforts, but also has considerable influence on the acceleration potential of the platform. Recent, high-profile trends in industry show interesting correspondences to our analysis. When the hardware integration of FPGA accelerators proceeds on this path, it will be foremost the interplay of design productivity and performance potential that governs the success of FPGAs in general-purpose computing. With our analysis and technical contributions in that field, we may help to shape the outcome of this process.

Zusammenfassung

Der Bedarf an immer höherer Rechenleistung für wachsende und neu aufkommende Rechenlasten und Märkte ist eine Herausforderung für die Rechnerarchitektur. Grenzen bei der Leistungsaufnahme und sinkende Erträge durch größere und zunehmend parallele Prozessoren machen Architekturinnovationen notwendig. Durch Effizienzsteigerungen und durch individuell angepasste Parallelität können spezialisierte Beschleuniger dazu beitragen, die Rechenleistung für bestimmte Rechenlasten zu erhöhen. Ohne Programmierbarkeit fehlt ihnen jedoch die Flexibilität für allgemeine Rechenaufgaben und sie können somit nicht davon profitieren, Kosten auf verschiedene Nutzungsszenarien und Märkte zu verteilen. Die Architektur von FPGAs, eine bestimmte Variante programmierbarer Logikbausteine, kombiniert Programmierbarkeit mit einem hohen Potenzial zur Spezialisierung für verschiedene Rechenlasten. Das größte Hindernis auf dem Weg zu einem verbreiteten Einsatz von FPGAs für allgemeine Rechenaufgaben ist der Mangel an produktiven Methoden und Werkzeugen zur Entwicklung und Wartung von Implementierungen, die ganz oder teilweise auf FPGAs ausgeführt werden. Wir analysieren in dieser Arbeit aktuelle Ansätze, die Produktivität bei der Entwicklung von FPGA-Anwendungen zu steigern, und arbeiten konzeptuelle und praktische Vor- und Nachteile heraus. Darauf aufbauend identifizieren wir drei Säulen, die einander dabei ergänzen können, die Verwendung von FPGAs für allgemeine Rechenaufgaben voranzutreiben. Diese Säulen kombinieren erstens eine Konfigurationsgenerierung ausgehend von parallelen OpenCL-Implementierungen, zweitens eine schnelle und automatisierte Übersetzung für übergelagerte Architekturen auf FPGAs, und drittens die Zusammenfassung von manuell optimierten FPGA-Konfigurationen zu anwendungs- oder bereichsspezifischen Bibliotheken.

In dieser Ausarbeitung konzentrieren wir uns auf übergelagerte Architekturen als Ansatz für produktive Entwicklungsprozesse für FPGAs. Im Laufe der letzten Jahre wurde eine Vielzahl solcher Architekturen vorgestellt, die durch eine Zwischenschicht das Abstraktionsniveau von FPGAs erhöhen. Allerdings existierte für die meisten dieser übergelagerten Architekturen nur ein unzureichendes Verständnis von Flächenmehrverbrauch oder reduzierten Rechenleistungen im Vergleich zu spezialisierten Konfigurationen die den FPGA ohne Zwischenschicht nutzen. Unsere Arbeit quantifiziert für eine instruktionsbasierte übergelagerte Architektur solche Nachteile anhand einer Auswahl unterschiedlicher Programmschleifen, die zu einer modernen Anwendung für stereoskopischen Bildabgleich gehören. Wir zeigen, dass diese Architektur trotz dieser Nachteile als praktisch

nutzbarer Beschleuniger dienen kann und identifizieren verschiede Unterschiede im Vergleich zu vollständig spezialisierten Konfigurationen. Dies könnte zukünftig helfen, durch Spezialisierung der übergelagerten Architekturen, deren Nachteile weiter zu reduzieren. Das Potenzial zur schnellen Übersetzung oder Konfigurationsgenerierung für übergelagerte Architekturen, sowie die Möglichkeit schnell Konfigurationen auszutauschen, war bereits ein Anreiz für die bestehende Forschung diesem Bereich. Um allerdings zu demonstrieren, dass sie tatsächlich dazu beitragen, eine erhöhte Produktivität bei der Beschleunigung von Anwendungen mit FPGAs zu erreichen, mussten wir bezüglich der Ausgangsdarstellung einen Schritt weitergehen. Dazu stellen wir Werkzeuge vor, die automatisch geeignete Schleifen aus Hochsprachenquelltexten oder Binärcode extrahieren und auf einem Vektorprozessor zur Ausführung bringen, der als übergelagerte Architektur auf FPGAs umgesetzt ist.

Neben dem Schwerpunkt auf Produktivität zur weiteren Verbreitung von FPGAs, müssen diese auch in etablierte Rechnersysteme und Märkte integriert werden, und dazu architektonisch mit allgemeinen Prozessoren zusammenpassen und zusammenarbeiten. Mit einem abstrakten Modell zur Abschätzung von Rechenleistung, das die gegenseitige Abhängigkeit zwischen Architekturen und Programmierentscheidungen berücksichtigt, erkunden wir systematisch Alternativen für derartig integrierte Rechnersysteme. Wir arbeiten heraus, dass die Integration der Speicherhierarchie nicht nur dazu beiträgt, Anwendungsentwicklung zu erleichtern, sondern auch einen erheblichen Einfluss auf das Beschleunigungspotenzial des Rechnersystems hat. Aktuelle, viel beachtete Entwicklungen bei kommerziellen Rechnersystemen zeigen inzwischen interessante Übereinstimmungen zu unserer Analyse. Wenn die Integration von FPGA-Beschleunigern in Systeme derartig weitergeht, wird vor allem das Zusammenspiel von Produktivität und erzielbarer Rechenleistung über den Erfolg von FPGAs für allgemeine Rechenaufgaben entscheiden. Mit unserer Analyse und den technischen Beiträgen in diesem Bereich könnten wir den Ausgang dieser Entwicklung entscheidend mitgestalten.

Contents

List of Tables

Algorithms and Listings

List of Figures

CHAPTER 1

Introduction

In more than two decades of research on reconfigurable computing, academia and industry have shown that field programmable gate arrays (FPGAs) permit huge gains in performance and efficiency for a wide range of suitable applications through customization and exploitation of parallelism on different levels. However, current usage of FPGAs is mostly confined to individual and specific tasks, for which the FPGAs were selected as most suitable or most economic architecture. Even the first large scale deployment of FPGAs in a data center was driven by a single application scenario [213].

In this thesis, we argue that in the light of current architectural challenges in general-purpose computing, FPGAs can play a much larger role in this domain. We identify current limitations and opportunities on this path, primarily in the area of productivity, but also with regard to system integration. We present original contributions in these two areas. In order to tackle productivity, we mainly propose and evaluate overlay architectures on FPGAs. With regard to system integration, we analyze design space options and compare our results to current trends.

In the remainder of this chapter, we first outline the motivation for this work in some more detail in Section 1.1. In Section 1.2, we lay out the contributions that this thesis makes towards general-purpose adoption of FPGAs and to specific technical challenges. In Section 1.3, we outline how the further thesis structure organizes these contributions.

1.1 Motivation

Computer architecture struggles to keep delivering ever more compute performance for growing and newly emerging workloads and markets. Well ahead of the eventual end of Moore's law [188], power limitations and diminishing returns from wider and more parallel processors are driving the need for architecture innovations. By reducing overheads and customizing parallelism, specialized accelerators help to efficiently increase the performance of specific workloads. However, without programmability, they lack the flexibility

for general-purpose computing and thus can't profit from shared costs among different workloads, users and markets. The architecture of FPGAs combines programmability with a high potential for specialization for different workloads.

The main obstacle for FPGA adoption in general-purpose computing is the lack of productive formalisms and tools for designers and maintainers of implementations running entirely or partially on FPGAs. For specific workloads, where performance or efficiency targets could otherwise not be met, sophisticated designs for FPGAs have been developed with the corresponding effort [121]. In general-purpose computing, such effort typically cannot be spent, particularly not for a target architecture that is not yet established in a wide range of systems. FPGA manufacturers and community have invested lots of effort in improving the accessibility and productivity of programming models, languages and tools. In this thesis, we discuss three essential pillars that can trigger a break-through in FPGA adoption, focus on overlay architectures as insufficiently understood paradigm, and investigate which overheads and trade-offs they involve.

In order to fit into established computing systems and markets, FPGAs need to coexist and cooperate with general-purpose processors (GPPs). The architectural integration of central processing units (CPUs) and accelerators is not only a central factor for performance and efficiency, but also has a huge impact on the design productivity for such systems. Industry is actively pursuing the integration of specialized accelerators, graphics-processing units (GPUs) and recently also of FPGAs with GPPs. In this thesis, we analyze the potential of such integration and discuss how we can predict it, even though architectures and workload implementations closely depend on each other.

1.2 Contributions of this Thesis

In pursuit of the main focus of this thesis, we present the following high-level contributions to promote the adoption of reconfigurable accelerators, in particular FPGAs, in the domain of general-purpose computing.

1. Based on an analysis of architectures, markets and workloads, we identify opportunities for FPGAs in data center and cloud environments, in high-performance computing (HPC) and mobile computers.

2. We present our vision of a three-pillar approach to FPGA usage with, firstly, accelerator-friendly Open Compute Language (OpenCL) specifications, secondly, overlay architectures to flexibly handle the trade-offs between productivity and performance of reconfigurable computing, and thirdly, libraries of reusable customized designs.

3. Further focusing on insufficiently understood characteristics of overlay architectures, we present the first broad quantification of the overheads of an instruction-programmable vector processor overlay architecture over fully customized kernel designs on FPGAs. For ten diverse kernels from a stereo-matching application, these overheads are on average in the range of 2.5x to 3x, which our experiments demonstrate to be small enough to still enable acceleration over GPPs.

4. We demonstrate that such an overlay architecture can serve as target for the desired fast and fully automated acceleration that existing tools didn't deliver.

5. In order to analyze the architectural integration of reconfigurable accelerators with CPUs, we present an estimation method that overcomes the interdependency of architectures and concrete implementations of workloads.

6. Based on this method, we give an overview of the design space of CPU-accelerator architectures that was insufficiently understood at the time of our publications.

Besides these high-level contributions, our work advances or complements the state-of-the-art in several technical aspects.

1. By avoiding optimization-induced reductions of algorithmic quality, we present the most accurate FPGA-accelerated stereo-matching implementation published to date.

2. Through fully compatible scalable kernel implementations and a memory management wrapper library, it runs on two very different hardware platforms with different programming models.

3. With the systematic variation of kernel patterns and data layouts, we isolate the effects of outer-loop vectorization for the vector processor overlay architecture and find measurable, but small effects.

4. Integrating our automation methods into the LLVM[1] compiler infrastructure allows us not only to target different variants of compiled and source code, but also to reuse components for analysis and offloading.

5. For dynamic code analysis, we found an interesting opportunity space between offline profiling and deterministic offloading decisions at program runtime.

We have presented and published peer-reviewed results of this work in nine conference contributions, one workshop, and two journal articles. Tobias Kenter is main contributor and first author of six of these twelve publications, including the article in one of the two premier journals on reconfigurable computing [6]. The author's publications are summarized starting on page 160, in front of the main bibliography.

1.3 Thesis Structure

The remainder of this thesis broadly follows the sequence of high-level contributions summarized in the previous section. Chapter 2 starts with a general background on computing terms, concepts and architecture trends. It leads to an analysis of market opportunities of FPGAs in general-purpose computing. Chapter 3 complements its predecessor by comparing the productivity perspective of FPGAs with established general-purpose computing

[1] `http://llvm.org/`

architectures. After introducing our three-pillar approach for FPGA usage in general-purpose computing, we review existing research on overlay architectures, which is the specific pillar we focus on in this thesis. This concludes the focus on backgrounds and general analysis.

The following three chapters present original research on offloading to overlay architectures and to customized kernels and on system integration of reconfigurable accelerators. In Chapter 4, we present the overhead analysis of an instruction-programmable vector processor overlay architecture. For this purpose, we offload ten different kernels from a general-purpose stereo-matching application to the overlay architecture and alternatively to fully customized kernel-specific FPGA designs. In the following Chapter 5, we focus on the productivity aspect when targeting this overlay architecture. While existing tools required considerable developer efforts and yet had limited coverage of supported kernel patterns, we demonstrate that the specific features of the overlay architecture can be used to guide a fully automated offloading process with wide applicability.

In Chapter 6, we focus on the system integration of reconfigurable accelerators with CPUs. We present firstly an architecture model that allows reconfigurable acceleration at different granularities, secondly the method of our performance estimation that avoids mapping-specific predictions and thirdly the design space exploration results for CPU-accelerator architectures obtained with this method. Chapter 7 concludes this thesis and points at directions for future research.

CHAPTER 2

Computing Concepts, Trends and Domains

In this chapter, we present the required background for the vision of FPGAs as accelerators for general-purpose computing. Firstly, in Section 2.1 we introduce basic terminology of compute devices and outline basic concepts of such devices, including FPGAs, on the architectural level. So, from a high-level perspective, this section is centered around architectural foundations of how computation can be done. The deeper roots of how computation is done, that is the fundamentals of semiconductors and process technology are out of the scope of this work. However their changes over time are driving factors for past and current trends in architectures and devices, which are discussed in Section 2.2. Thirdly, Section 2.3 extends this background by discussing what kind of computation is done for different purposes and markets, along with some of their requirements. We focus on the domain of general-purpose computing, designated in the title of this thesis and in this context introduce the paradigm of On-The-Fly (OTF) Computing, which motivated large parts of this thesis. Besides data centers and cloud computing, HPC and mobile computers, we identify On-The-Fly (OTF) computing as promising, but particular challenging target for FPGA acceleration.

2.1 Compute Devices: Basic Terms and Concepts

In this section, we introduce basic concepts of computing and compute devices and along the way lay out the terminology used throughout this thesis.

In a general sense, a *computer* is a device that transforms information in a way that can be described in an algorithmic form. *Computation* is the direct action of transforming information performed by the computer, whereas the field of *computing* summarizes any "goal-oriented activity requiring, benefiting from, or creating computers" [252]. However, the term computer is often associated with a narrower meaning [271] that implies the presence of an instruction-programmable *processor*, also denoted as *central processing unit (CPU)* (see Subsection 2.1.2) and some *general-purpose computing* capabilities (see

Subsection 2.3.1). In order to differentiate the more general computers that we discuss in this work form these narrower attributions, we denote them as *compute devices* or *compute systems*. The distinction of general-purpose devices to special-purpose or embedded systems is discussed in Subsection 2.3.1.

On the other hand, in the widest sense, computers encompass digital, analogue and even mechanical devices, whereas in this thesis, we restrict ourselves to synchronous digital compute devices implemented as integrated circuits. The basic elements of these digital circuits are transistors, which, arranged to elementary gates, implement basic logic functions on individual binary digits (bits). Inside a circuit, these bits are also denoted as signals, which after going through a sequence of combinatorial logic are stored in registers that keep their state until a clock signal triggers the register to assume the value of the input signal.

In the following Subsection 2.1.1, we continue the introduction of basic terms, here regarding target metrics of compute systems. In Subsection 2.1.2, we outline the basic concepts of computing with instruction-programmable processors and in Subsection 2.1.3 contrast them to concepts of computing in circuits. With FPGAs, a programmable architecture for computing in circuits is introduced in Subsection 2.1.4.

2.1.1 Target Metrics

Compute systems execute tasks, that can be specified in the form of software or as circuits, or as a combination of both. In this subsection, we outline target metrics of such systems as used throughout this work and point to relations between these metrics. The general goal for compute systems is to achieve high performance at low costs, with low power and energy consumption and with a low design effort. Those goals often compete with each other and several of the metrics are interdependent or form combined metrics like efficiency.

There are three typical metrics to characterize *performance perf*: First, the *execution time* t_{exe} is the time to execute one specific task or benchmark. Second, the *latency* λ is the response time between a specific input and its resulting output. Third the *throughput* $R = \frac{transactions}{s}$ is the rate at which similar tasks, in this context often denoted as transactions, are completed. Performance is proportional to the inverse of execution time $perf \propto \frac{1}{t_{exe}}$ and proportional to the inverse of latency $perf \propto \frac{1}{\lambda}$ but directly proportional to throughput $perf \propto \frac{transactions}{[s]}$. In synchronous devices, the performance depends on the clock frequency and the amount of work done per clock cycle.

When only a single task is executed at a time, the execution time is sufficient to characterize the performance a system, whereas for systems which execute more than one task in any form of concurrency, for example by pipelining, latency and throughput complement each other to characterize system performance. When comparing two different systems, a typical metric is speedup, which is the ratio of performance or execution times of the two systems. The speedup SU of a system S_{new} with regard to a reference system S_{old} is defined as $SU = \frac{perf(S_{new})}{perf(S_{old})} = \frac{t_{exe}(S_{old})}{t_{exe}(S_{new})}$.

In the conceptual parts of this thesis, when discussing performance we will have all of these metrics in mind unless any specific characteristic is highlighted. In our own

experiments in Chapters 4 and 5, the tasks of our workload are executed sequentially and therefore we will evaluate performance solely with the means of total execution times.

The manufacturing *cost* of a compute system is a monetary metric that reflects a complex interplay of aspects like chip area, fabrication process, yield rates during fabrication, assembly costs of components into a system and all steps of the design process. For comparisons, it is often desirable to express cost as per device, but this depends a lot on the volume of devices produced, since some fraction of costs occurs per device, whereas fixed costs like for design and mask generation, often denoted as non-recurring engineering (NRE) cost, can be split among all devices produced. Additionally, the variable costs per device change through scale effects and maturity of the production process. In this thesis, we consider general cost aspects within the remainder of this chapter, but don't report upon the concrete costs of the systems we used for the experiments in the following chapters, because the costs of individual low volume systems sold for academic use hardly reflect the actual costs that such systems would have in any volume market.

The *power* of a compute system while performing a workload determines both the amount of electrical power that must be supplied to the system and the amount of thermal power that must be dissipated, in many systems through active cooling. Both cause additional costs beyond the acquisition costs of a device. Together, all of those costs are often summarized as total cost of ownership (TCO) over a specified amount of time. *Energy* is computed by the integration of power over time, in our context the time to complete a specific task or set of tasks. Thus, energy puts power in relation to performance. When comparing two systems, the system with higher power consumption may still use less energy to complete a task if it has a sufficiently higher performance. Similarly to the costs, power and energy are a concern in the conceptual parts of this thesis, but due to the different nature of our hardware platforms are not evaluated in our experiments in Chapters 4 and 5.

Although *area* contributes to the cost metric as chip surface area, it is often considered individually, frequently expressed through proxies like number of transistors in a chip or fractions of utilized FPGAs resources (see Subsection 2.1.4). In the general case, this allows to abstract away the concrete decision for a specific manufacturing process and enables cost estimations prior to actual chip production. Similarly, in the FPGA case, this allows with numbers obtained for one specific FPGA model to estimate whether a design may fit to another FPGA or how much headroom there is to add additional functionality to the same FPGA. We use such estimations in Chapter 4 of this thesis.

The term *efficiency* puts one of the presented performance metrics of a system in relation to its cost or one of the cost related metrics. Therefore, typical efficiency metrics are energy efficiency as $\frac{perf}{energy}$, cost efficiency as $\frac{perf}{cost}$ and area efficiency as $\frac{perf}{area}$; when discussing efficiency throughout this work, we have all of those in mind, unless specified otherwise.

When we further talk about *productivity* of a design approach or programming model for a system, we relate either some achieved performance or achieved efficiency, to the design effort, typically expressed in time spent. Since for a fair assessment of productivity the required amount of knowledge and experience needs to be taken into account, it is harder to quantify than the previous metrics. We discuss productivity issues in Section 3.1. Based

on our comparison of two approaches in Chapter 4.7, we report upon the productivity we experienced during those experiments.

Finally, the *flexibility* to execute different tasks is an important characterization of a compute device. This aspect introduced in more depth in Subsection 2.3.1. Even though for each of the presented metrics a clear optimization goal exists, various trade-offs between different target metrics and different emphasis on either of the metrics are one contributing factor to the diversity of compute devices over time and now.

2.1.2 Instruction-Programmable Processors

The concept of instruction-programmable computers, as the first class of compute devices presented here, goes back to the Von Neumann architecture [263], but has long evolved since the first draft from 1945. At its core, a CPU contains an arithmetic logic unit (ALU) that performs the actual computation, registers to hold operands for the ALU, and a control unit that manages the operation performed by the ALU as well as the operands to be used. The CPU is connected to a memory, where both instructions forming a program and data for the computation are stored. Additionally, an interface for external input and output is part of the concept of any computer.

Instructions can encode the operations to be performed by the ALU and specify the registers that hold the operands. Other instructions specify data movements between registers and memory. The control unit decodes an instruction and identifies the operands, triggers instruction execution for example by the ALU and determines the next instruction by updating a program counter. In the simplest and typically most common case, the program counter is just incremented, otherwise a specific control flow instruction needs to specify, how the program counter it is updated. The next instruction specified by the program counter is then fetched from memory.

The basic concept of instruction-programmable computers turned out to be extremely powerful because of the flexibility to express any kinds of computing tasks with programs made from instructions. However, it also imposes two major drawbacks: firstly, besides the actual computation performed, there is an overhead, caused for example by the instruction fetch and decode steps, by instructions that don't contribute to the actual computations, or by the repeated movement of operands from and to registers and memory. This overhead limits both performance and efficiency of computing with processors. Secondly, the sequentiality of instruction execution facilitates reasoning about program execution, but also limits performance when each instruction can only be executed after the previous one is finished. Within the remainder of this subsection, we present some mitigation strategies in the areas of instruction set architecture (ISA) design, memory hierarchies and parallelism that are employed in current processors, mostly with regard to performance bottlenecks. The alternatives presented in Subsections 2.1.3 and 2.1.4 on the conceptual level and with more concrete architectures in Subsection 2.2.3 often closely combine efficiency and performance aspects.

The area of ISA design investigates, which types of instructions a processor should be able to execute. In early times of computer architecture, there was a trend to design a

high number of expressive instructions. Such expressive instructions can provide different ways to directly or indirectly encode operands in registers or memory, or designate special operations or groups of operations. With this approach, by getting more work done per instruction, the overhead of fetching and decoding each instruction can be mitigated and a compact program representation is achieved. In retrospective this strategy was denoted as complex instruction set computer (CISC), in contrast to the reduced instruction set computer (RISC) architecture, which was proposed in the 1980s [207, 208]. The latter concept proposes radically smaller and simpler instruction sets that allow memory operations only through explicit load and store instructions and have a fixed instruction size. This enables much simpler and smaller processor designs, which in turn allows to reduce the latency for executing each individual instruction. Another benefit, outside the processor itself, is that compilers can much easier generate efficient code for this type of architecture. In many contemporary GPPs, elements of both competing design approaches, CISC and RISC, have been combined.

Beyond this general distinction in ISA design approaches, a number of characteristic instruction set features have been developed to adapt processors to the needs of specific classes of tasks. For example, digital signal processors (DSPs) are optimized for signal processing applications and focus on data throughput in regular code. Microcontrollers on the other hand have very simple designs, optimized for control-dominant code and supporting only few and often slow arithmetic operations. These observations also point to the interplay of intended workloads and specialization that is an essential theme of this thesis and is more systematically approached in Section 2.3.1.

In order to supply a processor with instructions and to hold and supply most input, intermediate and output data, a word addressable memory is used. The fundamental challenge posed by such computer memory is that small amounts of memory can be made fast at the expense of area and power, whereas larger memory is slower. Therefore, a fundamental concept for memory design is that of a hierarchy, where the so-called main memory is typically designed to contain the entire program and data and two to four increasingly smaller and faster caches dynamically replicate transparently selected parts of the memory address space, so-called cache lines, for faster access. The processor registers form the fastest and smallest level of memory, but are typically not addressed like main memory and caches. At the other end of the hierarchy even larger and slower non-volatile storage like hard disks or flash are up to now not organized in data words, but in blocks and with file systems. The Von Neumann architecture's specific memory bottleneck is that both instructions and data are provided by the same memory and, which can lead to bandwidths limitations or contribute latency to the execution flow. This has been addressed by the Harvard architecture and modified Harvard architecture, which provides separate memory and interfaces for instructions and data. Most modern GPPs implement a modified Harvard architecture, where the main memory holds both instructions and data in a common address space, but separate first-level caches provide the processor with a distinct interface to each of them. Overall, modern memory hierarchies can provide good performance for many application scenarios, but contribute a significant fraction to the overall power consumption of the processors that use them.

After introducing ISA design and memory hierarchies, we now present three concepts of parallelism inside the area of instruction-programmable computers, that helped to mitigate the fundamental issue of program sequentiality over the the course of several decades, roughly following the structure and terminology of Hennesy and Patterson [118]. In the following Subsection 2.1.3 we outline how compute circuits that don't follow the basic principles of instruction-programmable processors may improve some of the aspects presented in this list.

- *Instruction-level parallelism (ILP)* occurs, when the correct execution of one specific instruction does not depend on the completed execution of all previous instructions from the instruction sequence of a program execution. This can be used to increase performance by executing several instructions in parallel or quasi-parallel. A compiler can increase the amount of exploitable ILP through instruction scheduling or loop unrolling. Scheduling can also be performed in hardware, then denoted as dynamic scheduling and allowing for out-of-order execution. The programmer is typically not involved in providing or exploiting ILP.

  In order to execute independent instructions in parallel, which is also denoted as horizontal parallelism [216], a processor needs to contain several independent execution units, either identical or different. When the processor front-end is able to decode and issue multiple instructions per cycle, the architecture is denoted as superscalar. Superscalar architectures aim at getting more work done per cycle, expressed in terms of instructions per cycle (IPC). There are two main variants of superscalar architectures. In very long instruction word (VLIW) architectures, multiple instructions are scheduled together into a single instruction word by the compiler, whereas in dynamic superscalar architectures, the scheduling of independent instructions to parallel execution units is performed at runtime in hardware. In order to increase the usable ILP, speculation is frequently employed, that is instructions after a branch can be completely executed before the outcome of the branch is fully validated. If it turns out that the speculatively executed instructions should not have been executed, their outcomes are rolled back. Also, many dynamic superscalar architectures can reorder the sequence of instructions, delaying those instructions that need to wait for dependencies and forwarding those instructions that can already start independently of all currently executing ones. In order to maintain the correct program behavior in such out-of-order architectures, the results of all instructions go through reorder buffers in order to be committed in the correct order.

  An orthogonal way to execute instructions partially in parallel is pipelining inside the processor. With pipelining, the process of executing a single instruction is split into several stages which overlap, for example while one operation is performed in the ALU, the next instruction is decoded and a third instruction can already be fetched from memory. Such pipelining is also denoted as a form of vertical parallelism. Since each stage can be completed faster than the entire sequence of stages required to execute an instruction, clock cycles are shorter, which increases throughput. However, pipelining does not increase the amount of work done per cycle, expressed as IPC.

On the contrary, in order to retain the same IPC than without pipelining, firstly the sequence of instructions to be known soon enough to fetch and decode the right next instructions. In practice, branch prediction is used to make educated guesses about the outcome of control flow instructions, and can achieve prediction accuracies between 80% and more than 99% depending on its design and the benchmark characteristics. Secondly, when the pipelined instructions are not independent, the pipeline has to deal with data dependencies, which however can for example be tackled by data forwarding, from one pipeline stage, where an operand is computed to another stage where the same operand is used by a subsequent instruction. Thus, pipelining does not necessarily depend on the availability of true ILP, but is easier with it.

- *Data-level parallelism (DLP)* denotes a form of parallelism, where inside a program the same operation can be applied independently on different data elements. The primary source of such DLP are loops without dependencies between subsequent iterations, so that each instruction of the loop can simultaneously can be performed on the data of several different loop iterations. Parallelism between different loop iterations is also denoted as *loop-level parallelism (LLP)*. As indicated earlier, through loop unrolling LLP can also be used to generate ILP and in that case is not dependent on fully independent loop iterations.

  The general concept of exploiting DLP is denoted as single instruction, multiple data (SIMD) [82]. In contrast to exploitation of ILP, SIMD concepts not only promise higher performance, but also better efficiency, because only a single instruction needs to be fetched and decoded in order to trigger several operations. Conceptually, DLP can often be automatically inferred, however efficient programs making use of DLP may still require considerable programmer interaction. This issue is also a subject of Chapter 5.

  Vector computers are the classical computer architecture to exploit DLP. Vector computers hold many data elements that can be computed in parallel together in vector registers. A single vector instruction causes the same computation to be performed on all elements of those vectors, or on a subset of the elements specified through a bitmask. Computation on the different elements is performed in a pipelined manner, here employing pipelining to hide the latency of an individual operation. In addition, several vector lanes can work in parallel on different elements of the same vector. Important features of those vector computers' instruction set architectures are different methods for data transfers between memory and registers and the mentioned concept of masking operations for some vector elements. Such a vector architecture is used in Chapter 4 and targeted in Chapter 5.

  A more recent way to exploit DLP is presented by SIMD instruction set extensions, sometimes marketed as multimedia instructions, that are integrated into many modern general-purpose (see Subsection 2.3.1) CPUs. In contrast to classical vector processors, their vectors registers typically contain less elements and their early incarnations contain a much less expressive instruction set. On the other hand, a

typical feature of SIMD instruction set extensions is the ability to treat data in SIMD registers as differently sized types, that is either as a few elements of a large data type, e.g. 64 bit integer, or as more elements of a smaller type, e.g. 16 bit integer.

GPUs form a further group of instruction-programmable parallel architectures that makes havy use of DLP. Instead of using large vector registers, GPUs are predominantly organized into groups of processing cores, which all have a distinct register file, but execute the same instruction, thus sharing the involved fetch and decode overheads. This principle is also referred to as single instruction, multiple thread (SIMT). The roots of GPUs started as fixed function graphics accelerators, which over time first gained programmability for a more flexible graphics pipeline, and ended up being increasingly used to and designed for many other tasks with similar parallelism. In this thesis, GPUs are not investigated in the practical parts, but are considered in Sections 2.2.3, 2.3.1 and 3.1 as reference for a commercially successful accelerator architecture with a mix of programmability and specialization.

- *Thread-level parallelism (TLP)* is a form of parallelism, where multiple instructions from different program contexts can be executed in parallel, an execution mode denoted as multiple instruction, multiple data (MIMD) [82]. Following [118], we distinguish two different types of TLP: in *request-level parallelism*, different threads can work relatively independently on separate requests. In *parallel processing*, several threads closely collaborate on a single computational task. This is often achieved by transforming DLP into TLP and distributing parallel work items to different threads.

  Considering only single-chip processors, two architectureal approaches to exploit TLP can be distinguished. In *multicore processors*, the entire structure of the CPU core, including control unit, registers and execution units like ALU, is replicated once or more times, so each core can work on a different thread independently. Often, parts of the cache hierarchy are shared among several cores in order to make collaboration between threads on different cores more efficient and in order to save some area. Systems where several identical processors or processor cores are connected through a common interface to a shared main memory are also denoted as symmetric multiprocessor (SMP) systems.

  Other designs don't replicate entire CPU cores, but rather just some parts. In a widespread approach, the central register file and parts of the control unit are replicated to support the parallel or quasi-parallel execution of different threads, whereas the execution units are shared among these threads. We denote this approach as *hardware multithreading*. We choose this terminology to differentiate from the multithreading term in the operating system context, which can be realized by context switches on a single processing core. Comparing hardware multithreading to multicore processors, the shared resources are a way to save area, however they may also turn out to be a performance bottleneck, so the overall impact on area efficiency depends on architectural details and workloads.

Table 2.1: Symbols used by the instruction set for our example.

Symbol	Meaning
`Ra`, `Rb`, ...	General-purpose registers.
`R0`	Register with fixed value 0.
`<var>`	Location of a variable in memory.
`<imm>`	Immediate operand that is encoded directly into an instruction.
`<label>`	Marker in the instruction sequence that is used as branch target.

> In hardware multithreading, the shared resources can be assigned to a different thread each clock in a round-robin way, skipping stalled threads (*fine-grained multithreading*), or only when the current thread incurs a costly stall like a cache miss (*coarse-grained multithreading*), or even dynamically within the same cycle *(simultaneous multithreading (SMT))*. In current processors, either fine-grained multithreading or SMT can be observed, whereas GPU architectures rely only on fine-grained multithreading. [118]

The effectiveness of all these concepts of parallelism depends on the fraction p of a task that can actually make use of this parallelism and on the remaining fraction $1 - p$ that for some reason remains sequential. Following Amdahl's law, when the parallel fraction p of the task achieves a parallel speedup SU_{max}, the speedup SU of the entire task can be computed as $SU = \frac{1}{(1-p)+\frac{p}{SU_{max}}}$. This means that in practice, both the maximal available parallelism SU_{max} in the parallel parts of the task and the fraction of sequential parts of a task put hard limits to the achievable speedups.

In this subsection, fundamental concepts of instruction-programmable processors have been introduced from a performance-centric perspective. In this spirit, current GPPs combine essentially all these concepts and more, in order to deliver good performance for any type of application. However, the means to mitigate performance issues of the Von-Neumann-based processor architecture don't necessarily improve efficiency, most of them leaving the general overhead of instruction fetch and decode unaffected and adding additional control logic that requires power and space. In this sense, the customizations of the briefly mentioned DSP and Microcontroller classes are to a large degree a reduction of features in order to improve efficiency. Microcontrollers are very simple and efficient designs, because they intentionally neglect arithmetic performance and data throughput, whereas DSPs with their focus on regular control flow can, for example, with zero-overhead loops reduce the number of executed instructions and at the same time reduce the need for sophisticated branch prediction. We illustrate computing with instruction-programmable processors with an example, before in the following subsection, discussing a computing approach that takes this customization much further.

Table 2.2: Selection of instructions for a RISC-like instruction set for our example.

Instruction	Meaning
`LD <var>(Ra), Rb`	**L**oa**d** into `Rb` a value from the memory location of variable `<var>`, modified by an offset value in `Ra`.
`ST Ra, <var>(Rb)`	**St**ore the value of `Ra` into the memory location of variable `<var>`, modified by an offset value in `Rb`.
`ADD Ra, $<imm>, Rb`	**Add** `<imm>` to value from `Ra` and put the result into `Rb`.
`ADD Ra, Rb, Rc`	**Add** value from `Rb` to value from `Ra` and put the result into `Rc`.
`BLE Ra, Rb, <label>`	**B**ranch to `label`, if the value in `Ra` is **l**ess than or **e**qual to the value in `Rb`.
`BLT Ra, Rb, <label>`	**B**ranch to `label`, if the value in `Ra` is **l**ess **t**han the value in `Rb`.

Introducing an Example

With an example, we want to illustrate some aspects of computing with instruction-programmable processors. We focus on some simple instruction set properties and on what we call instruction overheads. We extend this example in the following subsections to alternative approaches.

To outline a processor architecture for this example, we first introduce, based on a few basic symbols (Table 2.1), some instructions for a rudimentary instruction set in Table 2.2. The separation of memory instructions (load and store) and arithmetic instructions (and logic instructions not included here) is typical for a RISC architecture. The format of the memory instructions with a variable name and an offset register follows one of the addressing modes from the simple target machine model introduced in [15] and also shows up in the base instruction set of the instruction-programmable coprocessor that we target in Chapters 4 and 5. Eventually, during machine code generation, the compiler needs to replace the variable names by actual memory addresses, either in the *static* data area or in the *stack* data area. As branch instructions, we chose a pair of instructions that allows us to write the following example in a most comprehensible way.

With this instruction set, it is possible to execute a brief code snippet that is a simplified part of the real code used later in Chapters 4 and 5. Listing 2.1 shows the code snippet that sums up an input array `in` and stores the running sums into an output array `sum`. Listing 2.2 shows an instruction sequence that can execute this code snippet with the instruction set from Table 2.2. In order to keep the example simple, we omitted to specify different data word formats and assume a single, 32-bit (4-byte) integer data type. An important optimization that is included in this instruction sequence is the reuse of `sum[x]` values from the previous iteration as new value `sum[x-1]` in `R4`. By transforming the loop bounds, it would also be possible to remove one of the increment instructions (Lines 7, 8) inside the loop body, but we omitted this for better readability of the example.

```
1 for(int x=1; x<width; x++)
2   sum[x] = sum[x-1] + in[x];
```

Listing 2.1: Code snippet computing and storing running sums of an input array.

```
1  ADD R0, $1, R1       ## init loop counter x to 1
2  LD  (width)(R0), R2  ## load loop limit
3  BLE R2, R1, exit     ## branch to exit if width <= 1
4  ADD R0, $0, R3       ## init array offset to 0
5  LD  (sum)(R3), R4    ## load sum[0] into R4
6  loop:
7    ADD R3, $4, R3     ## increment array offset (4 byte word)
8    ADD R1, $1, R1     ## increment loop counter x
9    LD  (in)(R3), R5   ## load in[x]
10   ADD R4, R5, R4     ## compute new sum[x] in R4
11   ST  R4, (sum)(R3)  ## store R4 to sum[x]
12   BLT R1, R2, loop   ## branch to loop entry if x < width
13 exit:
```

Listing 2.2: Assembly code program for the code snippet in Listing 2.2. The actual work of the code snippet is performed in Lines 9-11.

When investigating this code sequence, one can argue that the instructions in Lines 7 to 9 perform the actual work of loading a new input value, adding it to the running sum, and storing it to the result location, whereas the other instructions are a form of overhead that is required to achieve this functionality. Other instruction sets aim to reduce such overheads. For example with a typical DSP ISA, one would replace the instructions in Lines 8 and 12 by instructions before the loop body to set up a zero-overhead loop. The ISA would also support special addressing modes making the instruction in Line 7 superfluous. A typical CISC-like feature present for example in the x86 ISA are arithmetic instructions with memory operands. Thus, it could for example replace the two instructions from Lines 9 and 10 by a single one. Like DSP ISAs, the x86 ISA also has addressing modes that allow to remove the instruction in Line 7. With such approaches, less instructions need to be fetched from program memory. However, inside x86 processors, such expressive instructions often get split again into several so-called micro-operations and thus internally require similar resources and time as the illustrated instruction sequence.

As platform to further illustrate the execution of the loop body from this instruction sequence on an instruction-programmable processor, we use a classic five-stage RISC pipeline like outlined in [118], with the following pipeline stages.

Instruction Fetch (IF) This stage fetches the instruction that is currently indicated by the program counter from memory. It also increments the program counter.

Instruction Decode (ID) This stage decodes the fetched instruction from the previous stage, that is it identifies the operation and the operands with their respective registers or immediate values. Additionally, it fetches the input operands from registers.

Table 2.3: Execution of two loop iterations from Listing 2.2 in a five-stage RISC pipeline. This schedule contains three data hazards (*marked with an asterisk). With data-forwarding, the value for `R3` computed in cycle 3 is already available in the EX stage in cycle 5 (could even be made available for cycle 4), even though not written back to the register file before. On the other hand, the result from the load instruction (#9) only arrives during cycle 6 and cannot be made available to the EX stage of the following instruction in the same cycle, therefore requiring a pipeline stall. The dependency of instruction #11 can again be resolved by data-forwarding.

		Clock cycle										
#	**Instruction**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
7	`ADD R3, $4, R3`	IF	ID	EX	MEM	WB						
8	`ADD R1, $1, R1`		IF	ID	EX	MEM	WB					
9	`LD (in)(R3), R5`			IF	ID	EX*	MEM	WB				
10	`ADD R4, R5, R4`				IF	ID	stall*	EX	MEM	WB		
11	`ST R4, (sum)(R3)`					IF	stall	ID	EX	MEM*	WB	
12	`BLT R1, R2, loop`						stall	IF	ID	EX	MEM	WB
7	`ADD R3, $4, R3`								IF	ID	EX	MEM
8	`ADD R1, $1, R1`									IF	ID	EX
9	`LD (in)(R3), R5`										IF	ID
10	`ADD R4, R5, R4`											IF

When operands are not valid, it either sets up data forwarding where possible or introduces pipeline stalls. Also, the comparison operation for branch instructions takes place in this cycle and if the branch is taken, the program counter is updated.

Execute (EX) In this stage, an ALU performs either the arithmetic operation specified by the instruction (in our case only `ADD`), or it computes the effective address for a memory instruction by adding the offset to a base address.

Memory access (MEM) This stage uses the effective address to either read a word from memory, or to write the value from the input register to memory.

Write-back (WB) In this stage, results from an arithmetic instruction or from a load instruction are written to the specified target register.

Table 2.3 illustrates how the loop body from Listing 2.2 can be executed in this five-stage pipeline. Due to a so-called data hazard, this schedule requires data-forwarding of the value for `R3` from instruction #7 to instruction #9 and of the value for `R4` from instruction #10 to instruction #11. The dependency between instructions #9 and #10 with regard to `R5` cannot be fully resolved by data-forwarding and therefore requires a pipeline stall. We assume no resource constraints between memory operations and instruction fetches, which would otherwise lead to structural hazards, and furthermore assume a correct branch-prediction to resolve the control hazard between instructions #12 and #7 in the second

iteration. In this example, due to the stall, the second loop iteration starts seven cycles after the first iteration, leading to an IPC value of $\frac{6}{7}$. In practice, further stalls would likely occur when load operations don't complete in a single cycle of the memory access stage.

This schedule illustrates how the pipeline of an instruction-programmable processor works on several instructions during the same cycle. However, the pipeline does not help to start or complete more than one instruction per cycle, but rather struggles to maintain this rate. Out-of-order execution can greatly help to keep pipeline throughput high, but requires additional area and energy for components that don't participate in the actual computation specified by the program code and thus are a form of overhead. When differentiating within the depicted five-stage pipeline between the actual computation specified by the code snippet (Listing 2.1) and overheads, one can argue that only the *execute* and *memory access* stages perform the actual work and the other stages are overhead.

2.1.3 Computing in Circuits

The discussed instruction-programmable processors are all implemented with digital integrated circuits. However, when the task of a compute system is fixed at design time, it is possible to design a digital integrated circuit that directly executes this task, without instruction-programmability and with a different structure than that of a processor. Such specialized compute circuits are often denoted as *hardwired logic*, or as *application specific integrated circuits (ASICs)*. In the circuit design community, the term ASIC is co-notated with a specific computer-aided design method for such a circuit, which involves the use of standard cells, and is contrasted to *custom circuit* design, that relies more on manual layout and optimization [55, 54, 94]. Therefore we stick to the terms compute circuit and hardwired logic here. Throughout this section, we present compute circuits directly in contrast to the previously discussed concepts of instruction-programmable processors, in particular how they can save area or improve performance at a much reduced or even removed flexibility.

Considering first the functional units that actually perform the computations, for hardwired logic, the departure from instructions removes the fixation to the specific operand sizes and to the specific set of operations specified by the ISA. A functional unit may thus be optimized exactly for operand sizes required by the compute task and the supported operations can be just a single one or a subset of the operations provided by general ALUs. A functional unit in hardwired logic can also execute required operations that are not included in a standardized ISA and therefore would in a CPU have to be performed through a series of other instructions. Contrasting hardwired logic to instruction-programmable processors, these techniques can be summarized as *functional unit customization* and can both save area and improve performance. Note a similar technique is used inside the realm of instruction-programmable processors to add specialized functional units to so-called application-specific instruction set processors (ASIPs) [96, 143].

The control information, that otherwise would have been encoded into instructions, is represented by several different parts of a application specific compute circuit. Fixed *wiring* between one functional unit that produces a data element and another functional

unit that uses this element in the next step can replace the operand encoding and at the same time obliterate the need for a central register file. Registers can instead be distributed throughout the circuit, for example as output registers after each functional unit. Wires can also be used to transmit synchronization signals like `valid`, `acknowledge` or `reset`. This way, direct wiring can also replace `move` instructions between different registers. Also arithmetic operations like `shifts` can be entirely replaced by wiring, by connecting specific bits of one source to different, shifted bits of the destination.

Considering the operators encoded in instructions, when a functional unit only has to perform a single type of operation, the correspondence of an encoded operation is not be needed at all. However, some control for selecting between different operations or different inputs is still required in many circuits. This control is typically provided by *state machines* (more precisely, but in this context less commonly, denoted as *finite-state machines (FSMs)*), described by a finite set of states and transitions between them. The compute circuit can for example select one of several inputs to a functional unit with a multiplexer whose control input depends on the current state. To summarize these two paragraphs, state machines, multiplexers, direct wiring and control signals enable *computing without instructions*. Some quantitative effects on efficiency will be presented in Subsection 2.2.3.

Without instructions, most typical stages of a processor pipeline, like instruction fetch, decode, operand fetch and write back are not relevant for compute circuits. For operations with high latencies, internal pipelining of functional units can still be valuable. Beyond that, pipelining is still a tremendously important concept for hardwired logic as a means to exploit additional vertical parallelism. The first target here is loop-level parallelism (LLP), by providing for each operation of a loop body a separate functional unit. These functional units are connected through wires representing the operand flow, but decoupled by registers into pipeline stages. For example while the first stage performs the first operation of the n-th iteration of the loop, the second functional unit can work on the second operation of the $(n-1)$-th loop iteration. Additional instruction-level parallelism (ILP) can be exploited by different parallel functional units within one stage, whereas additional data-level parallelism (DLP) often is addressed with entire parallel compute pipelines.

In between the two differently coupled forms of thread-level parallelism (TLP) introduced in Subsection 2.1.2 following [118], we denote as *task-level parallelism* the case, where a sequence of different tasks needs to be performed on several subsequent data elements or requests, a denotation used e.g. in [211]. In circuits, this type of parallelism can again be tackled with pipelining, here with a pipeline stage performing an entire task rather than an individual operation. Task pipelining is conceptually not limited to circuits [97, 27], but with customized FIFO buffers, multiplexing and demultiplexing, data width conversion or deterministic reordering, circuits can support many communication patterns much more efficiently than for example through shared memory regions.

Unlike the centralized control unit of a CPU, control for all these forms of parallelism can naturally be distributed over different pipelines and pipeline stages of a compute circuit, reducing complexity and overheads. The pipelining concepts in circuits are denoted as

streaming, emphasizing the flow of data through the functional units. A stream of data typically originates from memory or a device input, passes through several compute stages and is directed again to memory or to a device output. Thus, with intermediate buffers, streaming also is a natural form to decouple memory accesses from computation.

Beyond decoupled access to main memory, hardwired logic also has full control to explicitly manage any local memory. The registers and streaming buffers mentioned in the previous paragraphs are part of this local memory, but also addressable random access memory can be integrated into or attached to compute circuits. Optimizations over transparent caches can be achieved for example by adaption to known element sizes and data set sizes. Also, whenever data usage patterns are known, explicit data movement to and from local memory can be used to prefetch data ahead of its usage and avoid the overheads of the speculative prefetching that some transparent caches perform.

To summarize, compute circuits can achieve more performance and more efficiency than CPUs, through specialization and by removing the overheads involved with execution of instructions. However, their fixed functionality makes them unsustainable for usage scenarios where flexibility to execute different tasks is required. Also, since their fixed costs can only be distributed among devices for the same task, economically they require sufficient market sizes to pay off. Finally, time-to-market of specialized compute circuits is high, since it involves a lengthy process including hardware design and mask generation. In the following subsection, we present a technology that addresses these challenges by realizing compute circuits in a programmable way. Before, we extend the example from the previous subsection to a circuit design.

Extending the Example to Circuits

In order to highlight some of the differences between computing in circuits and computing with processors, we extend the example from Subsection 2.1.2 to a custom circuit with the functionality defined in Listing 2.1. The circuit is fixed to this specific functionality, but illustrates the performance and efficiency potential of computing in circuits. Figure 2.1 illustrates a possible design for such a circuit, with gray boxes summarizing the functionality of different parts of the circuit.

In the upper left corner of the figure, a counter is incremented in every cycle and can be reset (RST) to the initial value 1. In the lower half of the figure, memory units illustrate load and store operations from a common random access memory like specified by the listing. They compute memory addresses based on the counter and on the base address of their respective arrays and feed them into ports to the actual memory. With FIFO buffers at the memory ports we illustrate that memory accesses may involve some latency. The values read by the load unit are fed to the the upper right half of the figure, where an adder accumulates the inputs into a running sum that in turn is forwarded to the store unit. In the upper center of the figure, an enable signal (EN) is generated that ensures that the memory controllers only access data within the loop bounds. Beyond the scope of the figure, the base addresses for `in` and `sum`, as well as the values for `sum[0]` and `width` need

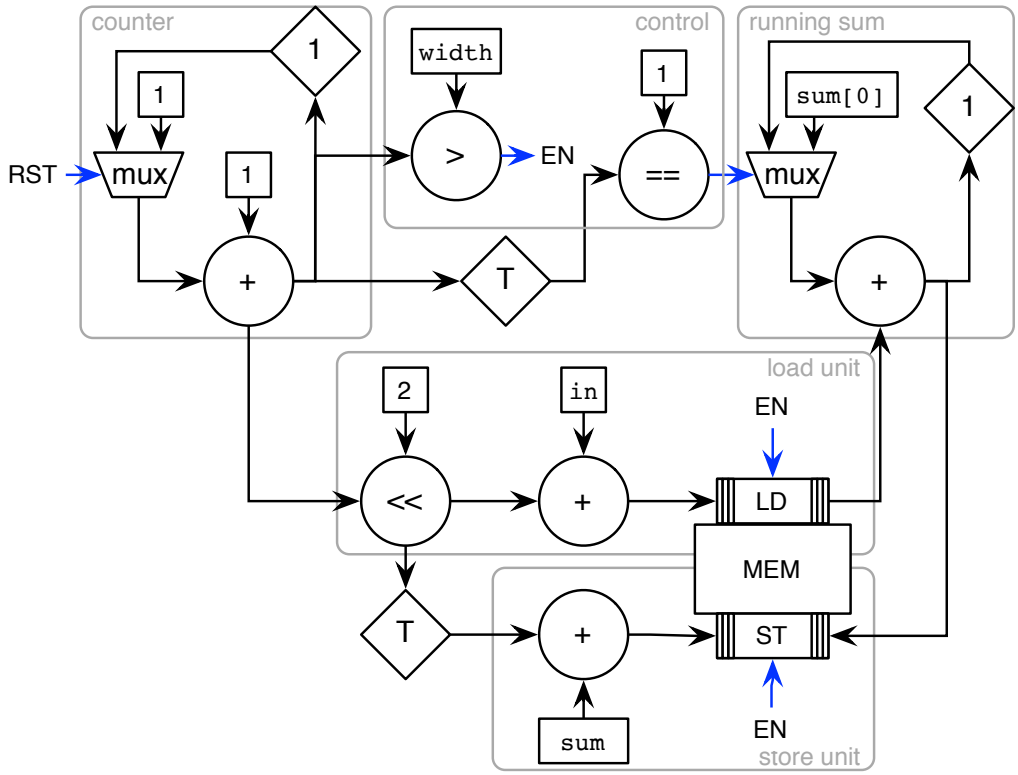

Figure 2.1: Illustration of a circuit that executes the code snippet from Listing 2.1. Blue arrows indicate wires for single bits, black arrows for entire data words. Circles indicate operators. Rectangular boxes contain input words, where `sum` and `in` are the base addresses of the respective arrays, whereas `width` and `sum[0]` contain the actual variable values. Trapezoid boxes indicate multiplexers to select one of two possible inputs. Diamond shaped patterns depict delay elements that may be realized by FIFO buffers. Access to a common memory is illustrated by a `load` port that contains an input FIFO for addresses and an output FIFO for loaded values, and a `store` port that has two input FIFOs, for addresses and corresponding values. The enable signal `EN` ensures that no memory access takes place outside the loop bounds. In the figure, the signal is split for clear arrangement.

to be loaded prior to the actual loop execution in the circuit. For this purpose, additional control logic, most likely in the form of a state machine, and additional wiring is required.

This example gives an impression how pipelining in circuits goes beyond that of a processor pipeline and can exploit loop-level parallelism (LLP). For each individual operation of the loop, the depicted circuit contains dedicated hardware components that each can process one element per cycle. Thus, the circuit can achieve a throughput of an entire loop iteration per cycle. To this end, different components may need to be decoupled by buffers to compensate for latencies in other parts of the pipeline. In our example, the load unit may have a considerable latency. In the example with the processor pipeline introduced in Subsection 2.1.2, such a latency would cause additional pipeline stalls and

significantly reduce performance. In the circuit pipeline in Figure 2.1, the delay elements denoted with `T` can fully compensate for this latency, as long as both memory ports can sustain a throughput of one element per cycle.

The performance potential through exploitation of LLP in this circuit comes at a certain area cost. For example the circuit contains four distinct adders. In contrast, in the processor pipeline, the functionality of all these adders is performed by a single adder (or more generally ALU) in different subsequent cycles. On the other hand, as driver for area and energy efficiency, the circuit requires no components to fetch or decode instructions and does not use a central register file, but rather forwards data directly to the components were it is used. The further advantage of customizing functional units to specific operators and data types is not well visible in our example, since we earlier decided that all adders work on the same 32-bit integer data type. However, the shift unit in our circuit illustration that transforms counter values to array offsets in bytes was just included for comprehensibility. In practice, it could be made obsolete by clever wiring.

Depending on the system in which this circuit is to execute the given functionality, the handling of input and output data could be handled very differently from the presented approach with a common random-access memory. For example, input data might be received directly from a sensor and forwarded to the next circuit or to an actuator of a mechanical device. However, within the practical parts of this thesis, computing is always performed on data in a memory, just like in this example.

2.1.4 Field-Programmable Gate Arrays

Field programmable gate arrays (FPGAs), are devices that implement compute circuits, but have no fixed functionality. Instead, they change their circuit behavior according to a number of configuration bits, which can be programmed and reprogrammed as the name implies in the field, that is by the user and after fabrication. The structure is denominated as gate array and describes an array of logic elements with a configurable interconnect. FPGAs are one architecture in a group of several different *programmable logic devices (PLDs)*. Architecturally, FPGAs differ from other PLDs by the utilization of *lookup tables (LUTs)* and are conceptually more suitable for complex and sequential logic than other PLDs. Currently, all of the largest PLDs are FPGAs.

The basic logic functionality in FPGAs is provided by *LUTs*, which can reproduce the output of any boolean logic function with n-inputs by configuring a small random access memory (RAM) with the desired result bit for each configuration of input bits. Typical sizes for input width of FPGA LUTs are 4-6. In order to allow for sequential logic, the output of a LUT can go to a *flip-flop (FF)*; a configurable multiplexer selects either the buffered or the unbuffered output. Together, these three components form the basic logic elements (BLEs) of an FPGA. In the academic view of FPGAs, a number of such BLEs can be grouped into clusters, internally connected by wires and with additional multiplexers to configure the BLEs' inputs as a combination of a cluster's external inputs and the outputs of other BLEs inside the cluster, and to configure the cluster's outputs [14].

The clusters are interconnected through *routing resources* in which configurable *switch boxes* determine the connections between different *wires*. The clusters form a first hierarchical structure designed to optimize area and delay of the routing network, because signals that remain inside a cluster don't need to go through the more costly routing network. The routing architecture can be further enhanced by providing wires of different length, with some spanning more than one cluster between two switch boxes [33]. Nevertheless, just like in ASICs [94], routing consumes most of the area of an FPGA chip and still for a number of circuits implemented on FPGAs it turns out to be the critical resource.

In commercial FPGAs, both design and naming of basic logic elements and clusters diverges a bit. The two FPGA vendors with the highest market share are Xilinx and Altera. In Xilinx Virtex-6 [275] and Virtex-7 [277] FPGAs, each LUT can be configured as one 6-input LUT or two 5-input LUTs. Four of those LUTs are combined with eight flip-flops, the required multiplexers and additional arithmetic carry logic into a slice and two of those slices form a configurable logic block (CLB) as equivalent of a cluster. In Altera's Stratix V [20] FPGAs, an adaptive LUT with 8 inputs can implement all 6-input logic functions, selected 7-input logic functions or be fractured into two smaller independent LUTs. One such adaptive LUT is combined with two embedded adders and four registers into an adaptive logic module (ALM), of which 10 are clustered into one logic array block (LAB).

Beyond the basic logic functionality through LUTs, several more specialized elements have been added to FPGAs over time. Parts of the first extra design element have been mentioned along with the vendor specific logic elements in the previous paragraph: In order to enable the generation of efficient adders on FPGAs, *carry-chains* and supporting logic have been added, which enable direct propagation of carry signals to neighboring logic elements, bypassing the global routing network with its comparably high area usage and delay. Additionally, dedicated *DSP blocks* have been introduced, which provide fixed function logic, typically for multiplication or fused multiply-accumulate (MAC) operations, often with configurable bitwidth. The functionality they provide can also be implemented with basic logic elements, but with much lower clock frequency and higher area consumption.

Another important resource of all current FPGAs is local memory, which is distributed within the array structure as *block RAMs (BRAMs)*, each of which can typically store several kBits of data with configurable address depths and data widths. Together with some logic resources, several BRAMs can be combined together to form larger memory blocks, both supporting deeper and wider memory. For local RAMs requiring even less memory bits or less address depths, often the configuration storage of LUTs can also be used as local memory, denoted as *LUT RAM*.

Last but not least, FPGAs need to communicate to the outside world or at least other chips in a system. For this purpose, they all have a number of IO pins, which are connected to the routing network as IO pads and often denoted as *general-purpose input-output pins (GPIO)*. Often a significant number of IO pads is available to support parallel interfaces like for external DDR-memory. Many modern FPGAs also contain a smaller number of *serial transceivers*, internally integrated as parallel interfaces but physically transferring

data serially at much higher frequencies. Those transceivers are used to communicate with serial interfaces like USB, PCIe or high-speed network links [223].

With this set of logic, interconnect and memory resources, and given a sufficient amount of resources, FPGAs can implement the functionality of arbitrary compute circuits, from fully customized ones to instruction-programmable processors. The structure and reconfigurability of FPGA resources forms a level of indirection and an overhead compared to circuits directly implemented in hardware, but is a prerequisite for their programmability. The process of programming an FPGA has conceptually more resemblances to circuit design than to software programming.

The classical design approach to implement a desired functionality on an FPGA is to specify the desired circuit design in a *hardware description language (HDL)* like VHDL or Verilog, which then is typically translated by a vendor-specific toolflow to a *bitstream*. HDL designs for FPGAs are typically specified on the register-transfer level (RTL), that is they specify how digital signals buffered in transistors change their state between two clock cycles. The register-transfer level (RTL) design combines structural information about the architecture components that are there and how they are connected by signals, with behavioral information, about how and when the components produce specific output signals. A bitstream, which is generated from an HDL design, contains all configuration bits for a specific FPGA, including configurations for LUTs, switch boxes and multiplexers, that configure the FPGA to behave like the specified design. A bitstream may also include initial values for BRAMs. The translation process from HDL to bitstream can be subdivided into several steps, which we only outline on an abstract level here.

In the first step, the HDL design with behavioral descriptions, is *synthesized* into a purely structural representation denoted as netlist. The netlist describes the circuit structure in terms of gates, which not yet need to correspond to the resource types that are actually available on an FPGA. In the *technology mapping* step, the appropriate resource types, excluding routing resources, are selected. In the *placement* phase, concrete instances of each resource are selected for each occurrence of a resource type in the technology mapped netlist. In the *routing* phase, the placed resources are connected through the wires and switch boxes of the routing network. Finding the best placement and routing are hard problems with huge search spaces, so heuristics, often with randomization, are employed and runtimes of these steps are high. Afterwards, the *timing analysis* determines the longest delay, through logic resources and routing, on any connection between a pair of flip-flops. If the timing analysis meets all previously provided constraints, it is followed by the final *bitstream generation*. Otherwise, one or more of the previous steps can be repeated with different parameters.

Even though strictly speaking the *synthesis* denotes only the netlist generation [86], the term is frequently used for the entire translation process from HDL specification to bitstream generation. We follow the latter, broader designation throughout this thesis. For the FPGA manufacturers, their closed source synthesis flows are an important part of their platform ecosystems, however in the academic world, an open source synthesis flow [168] was developed to enable research on aspects of synthesis as well as of FPGA design. On top of this basic HDL-based synthesis flow, different *high-level synthesis (HLS)* approaches

are building up to offer more abstract, productive and software-like design flows. Such concepts are presented in more detail in Subsection 3.1.3.

To summarize, FPGAs allow for a circuit-like computing paradigm instead of the instruction-based computing of processors. Programmability allows FPGAs to mimic very different circuits, but this comes at a hardware overhead that we detail further in Subsection 2.2.4 and with a programming model that requires structural information and an extensive synthesis flow.

Mapping Elements from the Example to FPGA Logic

There are several ways to achieve the functionality of the example code snippet from Listing 2.1 with an FPGA. Following the motivation to this subsection, one approach is to map the circuit from Figure 2.1 onto the FPGA and thus profit from the performance potential and efficiency of computing in circuits. A more indirect alternative approach is to first implement the five-stage processor pipeline introduced in the example of Subsection 2.1.2 on the FPGA and subsequently execute the assembler code on this processor. Advanced variants of both approaches are actually used and analyzed in Chapter 4. In the example at this point, we illustrate the flexibility of FPGAs by mapping three components of the circuit in Figure 2.1 to FPGA components.

In Subsection 2.1.3, we outlined that four distinct instances of an adder are used in the circuit in order to achieve a throughput of an entire loop iteration per cycle. On an FPGA, these adders can be realized by DSP blocks that can not only directly compute the sum or product of two input words consisting of several bits, but also typically contain accumulator functionality as required by the counter and running sum units from Figure 2.1. However, the adders can also be built from LUTs as the most fundamental resource of FPGAs. In Table 2.4, we illustrate the truth table for a 1-bit full adder that can be used as a building block to the 32-bit adder [111] required in our example. This truth table can directly be used to program two 3-input LUTs, one for each output bit. The special carry chains introduced earlier can be used to connect the carry bits to neighboring LUTs with low overhead. However, this first approach is not efficient, as current FPGAs have larger LUTs. In order to adapt to this, the building block for the adder can for example be modified to a 2-bit full adder, which can exactly be implemented with two 5-input LUTs that are one of the configuration variants of current Xilinx LUTs. For more details on adders implemented in FPGAs logic, we refer to [67].

Another component of the circuit in Figure 2.1 are multiplexers. Some small multiplexers are available as dedicated resources within the logic slices of most FPGAs. However, particularly to generate larger multiplexers, also LUTs are frequently employed. We illustrate the principle in Table 2.5 with the truth table for a multiplexer that selects one of only two input bits and thus requires only a single select bit S_m. In Section 4.5.1, we encounter a design that uses multiplexers with up to 35 inputs. A tree structure of smaller multiplexers can be used to build those. Chapman [51] introduces different design alternatives for large multiplexers on Xilinx FPGAs.

Table 2.4: Truth table for a 1-bit full adder with C_{in} as carry in bit from a previous full adder. This truth table can directly be used to program two 3-input LUTs, one for each output bit.

Inputs			**Outputs**	
A	**B**	**C**$_{in}$	**S**	**C**$_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table 2.5: Truth table for a 2-input multiplexer with select bit S_m. This functionality can be achieved with a single 3-input LUTs.

Inputs			**Output**
I$_0$	**I**$_1$	**S**$_m$	
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

As third component from Figure 2.1, we want to map the equality check in the control unit to FPGA logic. Notably, one of its two inputs is the constant value 1. Hence, no generic comparison between two inputs is required, but a LUT can directly be programmed to indicate whether the input variable is equal to 1. In order to keep the table size in check, we illustrate again a small truth table for a 3-input LUT, which covers the three least significant input bits of the input value. The comparison with an entire 32-bit input value can again be performed with a tree of smaller LUTs.

These three components mapped to the same type of FPGA logic resources may give an idea of the versatility of FPGAs and a glimpse to the amount of implementation decisions that need to be taken when mapping a circuit design to FPGA resources. However, the most time-consuming phases of most synthesis tool flows, component placement and routing, have not even shown up in this example.

Table 2.6: Truth table for the equality check with the fixed value 1. The three least significant input bits from I_2 down to I_0 are covered in this table that can be mapped to a 3-input LUT. Only one of the input combinations corresponds to the binary value 1.

Inputs			**Output**
$\mathbf{I_2}$	$\mathbf{I_1}$	$\mathbf{I_0}$	
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

This concludes our section about basic concepts of compute devices, where we have outlined the fundamental differences between computing with processors and in circuits as background to understand FPGAs. In the following Section 2.2, we present how the underlying process technology was and still is a major driver for how these concepts are combined into actual computer architecture and we again point at the special role of FPGAs in that context.

2.2 Trends in Technology, Architectures and Devices

In this section, we present how the trends in process technology summarized in Subsection 2.2.1 are a major driver for changes in processor architecture as outlined in Subsection 2.2.2. Accelerators are presented as one consequence of the ongoing efficiency challenge in Subsection 2.2.3, before we discuss in Subsection 2.2.4 how FPGAs architecturally fit into this trajectory.

2.2.1 Scaling in Process Technology: Continuity and Changes

The history of computing hardware has been governed for more than five decades by Moore's law, which predicts exponential growth of transistor counts per chip. From around 50 components in one integrated circuit in 1965, when Moore published his first projection for the next decade [188] to around 8 billion transistors on a single compute chip in 2015 [239], the number of components has doubled more than 27 times in 50 years. Research and industry continue to work new process technology to further increase transistor density [281, 23, 193]. The recently released International Technology Roadmap for Semiconductors (ITRS) 2.0 report, 2015 edition [139] projects a possible *technology scaling* path up to the year 2030, but contains technical challenges and preconditions that must be met for that roadmap.

Over time, the characteristics of this device scaling have significantly changed. The era of up to the early 2000's is denoted as classical, geometrically driven scaling [138], characterized by Dennard's scaling rule [71]. In this era, with decreasing physical dimensions of a transistor, supply voltage could also be linearly reduced. With this type of scaling, switching delay can linearly scale down that is clock frequency can linearly scale up and at the same time, power density, that is power per area, remains constant. This key insight was already qualitatively proposed by Moore [188], but Dennard became known for backing it up with a model and also quantifying the delay time with regard to the scaling factor.

The era of classical scaling ended, when supply voltage could no longer linearly reduced with feature sizes, which is attributed to device characteristics depending on the bandgap potential of silicon [83] and to thermal noise [152]. This caused a power density problem and put an abrupt end to frequency scaling. Also static leaking currents through insulating transistor components started to have an impact on the overall power consumption in addition to the traditional dynamic power for switching transistors [83]. With the introduction of new materials and transistor geometries [238, 131], in the era of so-called *equivalent scaling* [138], these challenges are kept under control sufficiently to retain the exponential growth of transistor counts per chip. However, these techniques mitigate, but don't remove the tendency of increasing power per area and absolute power per chip, which hit practical limitations and became known as power wall. 3D scaling or stacking of several chip layers, as the next trend in manufacturing that can increase device density, poses even more power challenges, because thermal power of several layers has to be dissipated from the same area.

Therefore, additional measures on the device and architectural level need to be employed to limit power consumption. Important concepts in this regard are to operate devices with dynamic voltage and frequency scaling [43, 225, 120] and with power gating [130, 13] of idle components. Complementary architectural concepts like SIMD instructions and multicore architectures, are introduced in Subsection 2.1.2 and are put into the context of equivalent scaling in the next subsection. Further steps ahead are proposed as near voltage threshold computing [75] and approximate computing [260, 105, 109], which encompass more radical ideas like lowering supply voltage into areas, where absolute performance or correctness start to degrade.

Beyond the technical concepts for scaling process technology, costs play a major role for the actual progress of semiconductor technology. The ongoing trend to follow Moore's law depends on the fact, that not only the area, but also the cost per transistor decreases exponentially over time. This development is still largely intact, but it requires ever-increasing production volumes, as one-time costs are growing with every new process generation. This affects the costs to develop a new process generation up to production level and the costs to equip each individual foundry with the required tools and environment for this process. Both costs have gone up to a point, that researchers speculate that not technical feasibility, but rather costs might put an end to Moore's law [219, 171]. Along with those process costs, also the costs for each individual chip design, with an increasing share of costs required for mask production, have grown considerably [267]. This requires higher

volumes for each design for amortization, or is forcing designers that expect lower volumes to resort to older process technology [171].

2.2.2 Impact on Processor Architecture

In this subsection, we put many of the processor design concepts introduced in Subsection 2.1.2 into the context of technology scaling just introduced in Subsection 2.2.1. Technology scaling is the enabling factor to — among others — build devices with more components. One goal of computer architects and application designers alike, is to transform this potential into *performance scaling*, that is the ability to execute given workloads faster or larger workloads in the same time. In this section, we focus on the computer architecture perspective, whereas in Section 3.1, the perspective of application developers is taken into account.

Parts of the increased transistor counts enabled by Moore's have always been used to add new features, rather than to boost performance for existing workloads. Such primarily feature oriented architecture innovations are increased bitwidths of data words and memory addresses, up to 64-bit addresses starting in the early 2000s [147], hardware support for floating point numbers and virtual memory, or more recently the introduction of cryptographic instructions [253, 103] and hardware support for transactional memory [264, 283, 44].

Performance gains in the era of geometric scaling and Dennard's scaling rule were mostly achieved by the pair of frequency scaling and of exploiting the additional transistors available per area and per chip for increased ILP (see Subsection 2.1.2). These two performance drivers were perceived as a form of "free lunch" [248], because any program can profit from them without any effort, just by waiting for the next processor generation. The techniques of superscalar execution, be it dynamic or explicit, and of out-of-order execution, contributed significantly to the increases in overall compute performance. Further gains are limited by the existing instruction level parallelism and by efficiency issues when aiming for wider superscalar execution. Explicit superscalar architectures with VLIW or Explicitly Parallel Instruction Computing (EPIC) [232] ISAs are particularly limited by the ability of compilers to fill their functional units with independent instructions based on static analysis. Dynamic superscalar architectures mitigate this difficulty by employing dynamic analysis and speculation at runtime in hardware, but may execute speculative instructions in vain and use much area and power for the required instruction issue logic [118]. To illustrate the slow pace of ILP scaling, in the era of equivalent scaling, it took 14 years to go from the AMD K7 architecture [92, 129] introduced in 1999 with 3 address generation units (AGUs) and 3 integer ALUs to 4 AGUs and 4 integer ALUs in a comparable processor class with the Intel Haswell architecture introduced in 2013 [136]. Admittedly, this comparison disregards significant changes in many other aspects of the architecture. In a different perspective, based on simulations of instruction traces conducted by Intel, eight subsequent processor generations to Broadwell architecture released in 2014, achieved a cumulative increase in IPC of close to 1.8x over the Dothan architecture introduced in 2004 [89, 193].

Starting during the era of Dennard scaling and continuing during equivalent scaling, a part of the additional transistors of GPPs were also dedicated to exploitation of DLP through SIMD instruction set extensions, e.g. Intel MMX [209]. This trend is still ongoing with SIMD units exhibiting ever more parallelism, since this technique can improve efficiency by getting more computations done per instruction. However, progress in this domain is also limited by the available DLP in applications and by the ability of developers and tools to exploit it through SIMD instructions. From MMX introduced in 1996 to AVX-512, which debuts in 2016, the bitwidth of SIMD has increased from 64 to 512 and thus doubled 3 times in 20 years, while transistor counts in the same time span doubled around 11 times. In Chapter 4, we demonstrate that some application patterns can profit from much wider vector instructions. This functionality is provided by FPGA accelerators, since application coverage is not sufficient to include such very wide SIMD units into GPPs. Chapter 5 covers aspects of the automatic code vectorization for this target and of partitioning process required for targeting an external accelerator rather than a tightly integrated SIMD unit.

With the limitations and efficiency problems of frequency scaling and exploitation of ILP at the transition from Dennard scaling to equivalent scaling, multicore architectures became popular [54]. In these architectures, additional transistors are spent to replicate entire processor cores to make use of TLP, thus avoiding the power limitations of frequency scaling and the diminishing returns of increasing ILP. Consequently, the usefulness of this approach is limited by the amount of TLP in the executed programs. In contrast to ILP, which is exploited by processor hardware or compilers and thus presented itself to application programmers just like frequency scaling as "free lunch" [248], most TLP has to be explicitly specified by the programmer. Also, performance scaling of multithreaded programs is limited according to Amdahl's law by serial code sections.

The vast majority of multicore processors are built with a single, shared main memory and a hierarchy of shared and private caches. In order to let the cores communicate and synchronize through memory accesses to this shared memory, cache coherence protocols are implemented in hardware to make sure all cores see the same data at identical memory locations, also through private caches. Scaling the required memory buses and interconnects for coherency traffic poses an ongoing challenge for hardware designers, but appears to be manageable as of now [173, 193].

Even though the multicore approach avoids the particular efficiency problems of single-core performance scaling, it is still is subject to power and power density limitations [127]. For the transistors that don't contribute to overall performance both because of limited TLP and because of power limits, Esmaeilzadeh et al. [78] have coined the term Dark Silicon and conclude that "adding more cores will not provide sufficient benefit to justify continued process scaling" [78]. Heterogeneity in multicore architectures [159], both with regard to clock frequencies [52] and core microarchitecture [197] helps to mitigate this issue for workloads with variable TLP during different phases, but ultimately remains limited with regard to the performance of the fastest core. Such concerns are even more pressing in the huge emerging field of mobile computing, where both power and energy are much more constrained than for typical desktop or server compute scenarios.

2.2.3 Accelerators

The difficulty of transforming more transistors into more usable computing performance has renewed interest in more specialized architectures to efficiently accelerate specific tasks. Most prominently, GPUs have evolved from fixed function logic for graphics rendering to fully programmable graphics pipelines that can also accelerate various other tasks [205, 166] with massive DLP, predominantly with floating point computations. From a different starting point, but with a similar design philosophy, so-called manycore processors emerged, which in comparison to multicore processors offer more cores by reducing frequencies and ILP performance per core, and tackle similar problem classes to GPUs through their strong emphasis on SIMD execution units [224]. Even more specialized to their specific tasks are video decoding and encoding blocks, which have been integrated into various CPU and GPU platforms [233, 135]. As such accelerator blocks, not only fully customized circuits, but also processors with a specifically specialized ISA, so-called application-specific instruction set processors (ASIPs) can be used, like demonstrated for video encoding in [74]. In the POWER8 architecture, IBM integrates a true random number generator, a cryptography accelerator and a compression accelerator on the same chip with a multicore CPU [242]. Also the datapaths that execute the cryptographic instruction set extensions [253, 103] introduced as feature extension in the previous subsection can be regarded as specialized accelerators, just more tightly integrated than the previous examples. Particularly for mobile computing platforms with their tight power budgets, ever more special function units are integrated into Systems-on-Chip (SoCs). For example, according to an analysis of Shao et al. -[231], in several recent Apple SoCs, more than half of the die area is spent on up to 29 specialized IP blocks. Typical candidates for such special function blocks are in the domains of image, video, audio and speech processing and of the wireless network functionality.

Through different degrees and forms specialization, all these accelerators try to reduce the overheads of instruction-based computing and data accesses through register files and generic memory hierarchies. Dally et al. [66] quantify these overheads over the actual arithmetic operations performed to amount for around 94% for embedded processors. For GPPs, Horowitz [127] illustrates that instruction cache access, register file access and instruction control together use between one and three orders of magnitude more energy than individual arithmetic operations of various complexity. Hameed et al. [108] observe a 500x efficiency gap between a fully customized ASIC and a general-purpose processor.

In order to reduce the relative impact of instruction overheads or "cost of programmability", Horowitz [127] points out that more operations per instruction can be helpful when the costs for instruction-cache access and control remain relatively constant, but also that the energy to access data in the memory hierarchy is important. Dally et al. [66] focus on the energy needed to move around instructions and data and propose to reduce this energy mostly through the use of more and better partitioned register files, whereas Hameed et al. [108] seek to get a higher number of basic operations done per instruction. For their video processing case-study, they show that SIMD and VLIW enhancements can increase efficiency by around one order of magnitude and algorithm-specific instructions can yield

Table 2.7: Power projections for the example loop from Listing 2.1 based on numbers from [127]. Six instructions use 70pJ each, including 6pJ for register file access. This part increases for SIMD-instructions, which we omitted due to a lack of data. A single 64-bit access to a 8KB cache uses 10pJ. Our loop example uses two 32-bit accesses. With the estimated 20pJ, we might be able to use a slightly larger cache instead. A single 32-bit add uses only 0.1pJ.

Design	Energy in [pJ] per Iteration		
	Instructions	**Data Cache Access**	**Arithmetic**
Unaltered	420	20	0.1
8x SIMD	420	160	0.8
32x SIMD	420	640	3.2
1024x SIMD	420	20480	102.4

more than an order of magnitude on top of that, closing most of the gap towards hardwired logic. We observe that current GPU and manycore architectures mostly revolve around such SIMD enhancements, whereas more specialized accelerators go more into the direction of algorithm-specific instructions and circuits.

We briefly illustrate the argument of Horowitz and Hameed et al. with our running example, based on numbers for simple in-order processor in 45nm technology from [127]. Based on these numbers, we project different components for the power consumption per iteration of the loop body from Listing 2.1 in the first data row of Table 2.7. Each of the six instructions uses 70pJ for a total of 420pJ. Each loop iteration requires two 32-bit data accesses. We approximate these with the values for two 64-bit accesses to a 8KB cache, which would require 10pJ each. Whereas the smaller data size in our example would likely save some energy, larger caches or an access to main memory would require much more energy. In the last column, the energy for the core arithmetic instruction (a 32-bit integer add) of the loop is outlined. Its contribution to the total energy of the loop iteration is minimal. Even a hypothetical floating point multiplication at 3.7pJ would be negligible compared to the involved overheads.

In the next rows of the table, we outline alternative SIMD-variants that would, with the same number of instructions, work on more data elements per iteration. Note that for the simple loop from our running example, this is not possible due to the loop-carried dependency. However, in the larger variants of this loop pattern, that we target in Chapters 4 and 5, outer loops enable vectorized execution. In the example here, we see that even with a huge amount of SIMD parallelism, the energy for arithmetic remains dominated by instruction overheads. However, unless data can be reused within registers, accesses to the memory hierarchy start to dominate the overall energy consumption already at a medium amount of SIMD parallelism. Note however, that due to a lack of data, our example neglects the energy scaling for accesses to ever larger vector registers.

As a complementary efficiency driver, most accelerators first use abundant parallelism suitable for their specific task and hence can in turn often be operated in efficient, relatively low, voltage and frequency ranges and yet provide good performance. Along with the

concept of simple and power-efficient processing pipelines, this was the guiding idea for manycore architectures, but also most other accelerators are run at much lower clock speeds than GPPs. An extreme example for this paradigm, is the IBM TrueNorth [91] chip designed to execute brain-inspired neural networks. With 5.4 billion transistors, it exceeds contemporary GPPs and GPUs in size and yet operates at three orders of magnitude lower power. Note however, that this is not just achieved by conventional voltage and frequency scaling, but the chip's computing cores rather operate in an event-driven way.

The trend to integrate different accelerators with processors into SoCs is jointly driven by efficiency, performance and cost considerations. On-chip communication can at the same time provide much lower latencies and higher bandwidths and save energy by avoiding physically long communication paths, transitions between different chips and circuit boards and separate voltage regulation for different distinct components. In many usage scenarios, heterogeneous SoCs with specialized accelerators are a more profound answer to the challenge of dark silicon than mere heterogeneous multicores. When all transistors cannot or even must not be productive at the same time, specialized accelerators can reasonably exist to only execute some very specific parts of the entire workload of the chip especially fast or efficient and remain unused at other times.

From the economic perspective, different aspects of on-chip integration of accelerators overlap. Given sufficient volume, a SoC is much cheaper than assembling a comparable system out of discrete components. However, considering the increasing design and mask costs, the potential to customize a SoC's composition for different customer demands becomes very limited. On the other hand, in a power constrained scenario with dark silicon characteristics, it can be viable to ship the same SoC with a wide selection of accelerators to different customers, who each just benefit from a subset of the accelerators and leave another subset unused as a form of deliberate dark silicon. Finally, while the chance to differentiate a product with specialized IP blocks form the competitors has contributed to a vibrant SoC ecosystem, the pressure to reduce time-to-market favors more programmable accelerators.

2.2.4 FPGA Accelerators

Considering the observations of the previous three subsections, we think that FPGAs can play an increasing role in the future computing landscape. In particular, the interplay of two effects leads to this conclusion.

1. The challenge of transforming the transistor budgets into more usable compute performance in the era of dark silicon is ongoing. Specialized accelerators successfully meet this challenge for specific tasks.

   Architecturally, though with a programmability overhead, FPGAs support all the features of specialized accelerators that account for their efficiency compared to GPPs, that is elimination of instruction overheads, task-specific flow of data through the circuit and customized parallelism that in turn allows efficient voltage/frequency design points.

2. When the most frequently executed tasks are covered with specific accelerators, adding more accelerators suffers from diminishing returns in terms of tasks profiting from these new accelerators. At the same time, increasing design and mask costs limit the potential to customize the mix of accelerators in SoCs and similar products. Also, the time consuming process from identifying acceleration demand to an accordingly specialized accelerator requires considerable time to meet new compute demands.

   With their programmability, FPGAs can be specialized for a large variety of tasks, either over time in the same device or with different utilization of different devices using the same FPGA chip, even for tasks that are not even identified during the design. In Section 2.3.1, we outline how these are characteristics of general-purpose computing that FPGA shine with.

Compared to circuits directly implemented in ASIC technology, FPGAs suffer from an efficiency disadvantage of around one order of magnitude. Comparing ASICs to FPGAs with DSP blocks and BRAM manufactured with the same 90nm process technology, Kuon and Rose quantify the dynamic power consumption ratio as around 12x, the area gap as around 21x and the ratio of critical path delay as 3-4x [161].

However, compared to instruction programmable processors, FPGAs still retain a considerable advantage. DeHon [70] quantifies the density advantage of FPGAs over processors in terms of ALU bit operations per area and time for different generations of FPGAs and processors as one to two orders of magnitude. Sirowy and Forin [237] compare FPGAs to a simple RISC processor and investigate with three case-studies how different circuit variants implemented on FPGAs need several orders of magnitude less cycles than the baseline processor and even idealized variants of this processor. They conclude that beyond eliminating instruction overheads, aligning memory accesses to computations and forwarding of data throughout compute pipelines have a major impact.

Overall, with an efficiency between that of GPPs and custom logic and the programmability as core advantage over more specialized accelerators, the role of FPGAs somehow resembles that of GPUs or manycores, yet with very different architectural approach. While GPUs just avoid parts of the inefficiency sources of GPPs, FPGAs avoid them much more radically when implementing compute circuits, but introduce programmability overheads compared to custom logic. We illustrate this correlation qualitatively in Figure 2.2.

Chung et al. [58] study the design space of GPPs, custom logic, GPUs and FPGAs integrated into heterogeneous single-chip architectures qualitatively for three different workload types. Their results support the joint role of GPUs and FPGAs between GPPs and custom logic both in terms of efficiency and performance. They also note that in several bandwidth limited scenarios both GPUs and FPGAs are performance-wise almost on-par to custom logic. Even though all applications in this study use floating point computations, which conceptually favor GPUs, the performance and efficiency relation between GPUs and FPGAs is a mixed picture. Considering this, there should be room for FPGAs to join GPUs and possibly match in their actual market role for acceleration.

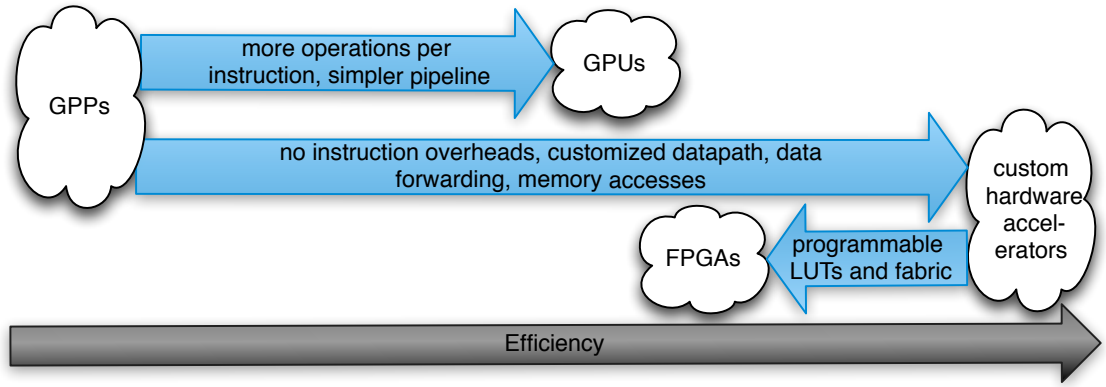

Figure 2.2: Illustration of efficiency drivers for different architectures. The relative positions of GPUs and FPGAs are neither a quantitative nor a qualitative statement and in practice vary depending on workloads and concrete architectures.

Indicators for such a trend are in fact visible. Considering on-chip integration with GPPs, both Xilinx with the Zynq series [214] and Altera with their SoC variants [19] are shipping products, which combine ARM processors and FPGA fabric in order to profit from the presented SoC integration advantages communication latency, power efficiency and possibly costs. The amount of integrated special function blocks of these products is not on par with non-reconfigurable SoCs and it remains to be seen, whether this is a significant disadvantage on the market, or whether the versatility of the FPGA can compensate for this. In the datacenter, Microsoft received considerable attention for a large-scale deployment of FPGAs to boost performance and efficiency of its Bing search engine and further upcoming workloads [213, 191]. Also, Intel, the world market leader for GPPs, acquired with Altera one of the two largest FPGA producers and announced first server products combining GPPs and FPGA.

For some workloads, a particular challenge for FPGA computing platforms is the off-chip memory subsystem, which trails GPUs in terms of latency, bandwidth, and transparent integration. Recent developments may help to overcome this issue, like products with Hybrid Memory Cube (HMC) attached via high-speed serial transceivers, products with coherent memory interfaces via the CAPI interface developed around IBM, and announced Intel products that will connect an FPGA to the processor bus. As the much earlier Convey HC-1 that is introduced in more detail in Subsection 4.3.2 illustrates, memory performance of FPGA computing products is primarily a matter of cost, volume and added value considerations and therefore can presumably quickly adapt to market demands.

Up to now, the market share of FPGA compute products in the general-purpose domain is much lower than the technical arguments in this subsection may suggest. In particular, there are no popular desktop or laptop or mobile computers with FPGAs, only first, small-

scale public cloud computing products with explicit FPGA resources[1][2] and only few HPC installations with FPGAs. In contrast, in all these domains, GPUs are well established, and manycores, despite sluggish availability, receive significant attention in the HPC and datacenter domains. In the following section, we have a closer look at computing markets and domains along with their requirements for compute devices, before in Chapter 3 discussing reasons for this low adoption.

2.3 Computing domains and markets

Computing has always been driven jointly by two factors, on the one hand by the devices themselves, which opened up new usage opportunities and on the other hand by applications and markets, which demanded and payed for new capabilities and features. Important conceptual and architectural aspects of the first driver, the device capabilities, have been mentioned in the two previous Sections 2.1 and 2.2, in particular with regard to parallelism.

In this section, we focus on the second driver, the markets that govern the economic development of computing and on application types that represent these markets. We start with the general distinction of the special-purpose and general-purpose computing domains (Subsection 2.3.1). We then discuss in Subsection 2.3.2 how current FPGA usage is mostly focused in the special-purpose domain. In Subsection 2.3.3 subdivide the general-purpose domain into several markets where we see further potential for FPGA technology.

2.3.1 General-Purpose and Special-Purpose Computing

Computing devices can be grouped into the domains of *special-purpose computing* and *general-purpose computing*, depending on whether at fabrication time, their functionality is fixed or not fixed, respectively. Most special-purpose computing devices are more commonly denoted as embedded systems or embedded computers, utilized in cars, household machines, photo and video cameras, many industrial machines and technical infrastructure like network switches. Typical examples for general-purpose computers are devices like personal computers (PCs) and laptops, servers and mainframes. The markets for general-purpose computers will be elaborated in more detail in Section 2.3.3.

There are two fundamental advantages of general-purpose compute systems. The economic one is that costs can be shared, be it NRE costs that are shared among different customer groups with different demands or be the individual costs of a specific system that fulfills different demands of one or several users. The second advantage is a functional one, the ability to perform computations that are not planned or possibly not even known at the time when a general-purpose system is designed or deployed. The competing advantages of special-purpose systems in contrast are, as indicated in the context of accelerators and specialized processor architectures, efficiency, higher or more predictable performance, and lower costs when the systems are produced in sufficient volume.

[1] `http://www.xilinx.com/products/design-tools/software-zone/sdaccel/supervessel.html`
[2] `https://xilinx-cloud.jarvice.com`

The distinction between general-purpose and special-purpose is not always as clear as it may appear at the first glance. We first present three criteria for distinction, then relate them to other definitions, before discussing their limitations at the hand of examples and drawing conclusions for the way these terms are used within this thesis.

1. Starting with the terms themselves, the purpose, functionality, workload or *computational task* itself serves as first criterion. Special-purpose computing here refers to a single specific task or a narrowly limited set of related tasks, whereas general-purpose computing implies a wider and more diverse task set.

2. The second aspect is the point in time, when the computational task is defined. If this fixation takes place before or at *fabrication time*, this is characteristic for special-purpose devices, otherwise for general-purpose devices. Note that the fabrication time of an individual chip is often different from that of the compute system it is used in, so the two can have different general- or special-purpose characteristics.

3. As third criterion, we summarize aspects of the *design process* of the devices. Any form of specialization, e.g. with customized functional units, local memory for specific access patterns, or interconnects for specific communication patterns, hints towards special-purpose devices. On the other hand, efforts to reuse components of a circuit for different functionality hint towards general-purpose devices. We here explicitly refer to the intent to and effort towards reuse, because any processor, even after heavy customization, exhibits some temporal reuse of functional components.

   As another aspect of the design process, for special-purpose devices, along with the workload, often a fixed performance goal is set, leaving cost, power and energy as typical minimization objectives, even though often subject to additional constraints. In contrast, in general-purpose domains, with explicit or implicit cost and power constraints given, the typical design objective is to maximize performance.

Looking at distinctions presented in literature, Hennessy and Patterson [118] use the ability of a device to *execute third-party software* to separate personal mobile devices like current smartphones as part of general-purpose domain from embedded systems. This combines the first two of the above criteria: third-party software is deployed after fabrication, but it is also a means to provide a wide and diverse functionality.

DeHon [69], who focused on the architectural perspective of reconfigurable devices for general-purpose computing, outlines two characteristics for general-purpose: To "Defer binding of functionality until device is employed - i.e. after fabrication" directly corresponds to our second criterion. His second key characteristic, to "Exploit temporal reuse of limited functional capacity" is one aspect that we included in our third criterion.

In the Berkeley position paper on the landscape of parallel computing research [25], the authors present the separation between fixed performance goals in embedded systems and the need for ever higher performance in HPC. They also note, that the separation currently is weakening, on the one hand because in both types of systems, power limitations already have become a major design concern, on the other hand because media-oriented server workloads may adopt more real-time, fixed performance goals from the embedded field.

In a much earlier consideration of parallel computing in general-purpose domains, Hey [122] makes an interesting distinction. He asserts that the first criterion for general-purpose computing from our list, in his words the ability "to deliver high-performance on many different types of problems" is only the view of vendors and enthusiasts. In contrast, the user of contemporary general-purpose devices expects to have "a very wide range of standard languages, libraries, applications packages, operating systems and tools at his or her disposal". We pick up this aspect later in Chapter 3.

We next look at concrete examples for compute devices or rather classes of such devices. We intentionally chose border cases, where the computing domains overlap and one criterion hints to one classification and another criterion to the other. We start with automotive systems and smartphones to illustrate this tension field before proceeding with a first assessment of the current position of FPGAs in Subsection 2.3.2.

Complex Embedded Systems in Automotive

As first device class, we consider the computer systems used in cars. In 2007, Broy et al. [41] characterize the hardware and software systems of premium cars built around that time with some numbers: around 70 electronic control units (ECUs), processors or microcontrollers of varying complexity, are connected with each other and with a multitude of sensors and actuators by several different bus systems. This high number of components is to a large degree caused by the component based design of vehicles, where many subsystems, mechanical components, sensors or actuators are developed along with their own control and compute systems. Together, these connected systems execute tens of millions of lines of code with up to 2000 distinct software-based functions, which are combined into around 270 functions that the user interacts with. As such, just on average each ECU executes around 30 different tasks. Beyond the tendency to bring the functionality of those systems from the premium segment into cheaper cars, there are two ongoing trends. On the one hand, on the path to so called smart cars and autonomous cars, ever more software-based functionality is added. On the other hand, for efficiency, reliability and cost reasons, system integration needs to be increased, that is ever more functionality is combined into individual components in order to keep the number of ECUs and the physical wires of the interconnect in check.

So, considering size and diversity of the workload, one could argue that such automotive compute systems as a whole already belong to the general-purpose domain. On the other hand, as of now, the end-users generally can't run third party software on these systems and software updates from the manufacturers are well known in the form of bug-fixes, but rather not to add new features, so functionality is mostly fixed after fabrication time. Also, most tasks are characterized by hard or soft real-time requirements, exhibiting the fixed performance targets typical for embedded systems. Therefore, overall we see automotive computing systems still in the domain of embedded systems, however individual parts, like subsystems for autonomous driving and infotainment currently seem to be in the process of adopting more general-purpose characteristics.

Modern Smartphones

As second device class, we examine current smartphones, which are already introduced in Subsection 2.2.3 because of their many accelerator components. In contrast to the automotive compute systems with their many individual components that are physically distributed all over the car, for smartphones a very high system integration is a governing design principle. This is caused not only be the form factor requirements, but also by the energy savings that are possible by the integration of many different compute components on a single chip into entire SoCs. According to a model in the the system drivers chapter of the 2011 ITRS roadmap [137], as of 2011, such mobile SoCs combine, along with memory and peripherals, 2 to 4 main processors and on average 129 additional processing engines, grouped and customized for a smaller number of specific functions.

As mentioned earlier in this subsection, one defining feature of smartphones is the ability to run third-party applications on an underlying operating system (OS). The extremely versatile workload enabled by this ability is not only a driving factor for the success of smartphones, but also a strong indicator for a classification as general-purpose devices. The design principles would not allow such a clear classification. The main processing cores are dedicated to be reused by the multitude of different applications and are adopting cache hierarchies, superscalar out-of-order execution and SIMD instructions from conventional GPPs, often clocked slower and scaled down for energy-efficiency. Their accelerators contain highly customized special-purpose processing elements to enable high efficiency. It is interesting to note, that parts of the functionality of special-purpose processing elements is exposed via application programming interfaces (APIs) to be used by the third party applications. Considering the optimization targets, the existence of benchmarking tools for both performance and battery runtime illustrates that these SoCs combine characteristics of general-purpose and embedded systems.

2.3.2 FPGAs in Special-Purpose Computing and Beyond

In this subsection, we approach the role of FPGAs as central components of special-purpose devices and beyond, primarily based on the revenue reports of the two most important FPGA vendors, Xilinx and Altera [276, 17]. These reports break down revenues into markets, where FPGAs are currently employed. Looking at FPGA-based devices from these markets, we can classify the majority of markets into the special-purpose domain, but also see an emerging general-purpose use-case. Competitors like Lattice are even more focused on the embedded marked [178]. As a side-note, even though the majority of FPGAs go into the special-purpose and embedded domain, the majority of those systems build upon embedded instruction-programmable processors like microcontrollers, DSPs and ASIPs, and on customized hardware.

FPGA revenue by end markets has traditionally been dominated by the communications sector, comprising wired and wireless communication infrastructure. In 2012, both Xilinx and Altera report [276, 17] more than 40% of their net revenue to be from this domain. Typical systems containing FPGAs are high-end routers and switches as well as cellular base stations. In these systems, FPGAs typically execute the same very specific

tasks over the entire lifetime of a device. They are valued for the capability to represent routing patterns in hardware and to process packets in a pipelined and highly customized way. Considering this, the role of FPGAs here is clearly that of special-purpose devices. Their direct competitor are ASICs, which promise lower costs at high volumes and higher energy efficiency, but incur higher NRE costs and longer times-to-market. However, in this comparison, FPGAs also excel with capability to fix bugs or even add new features after fabrication time. In computer networking, the approach of software-defined networking (SDN), which allows to define or modify network decisions by rules written in software, may lead to stronger dependence on the programmability of FPGAs in this domain.

Other end markets for FPGAs are industrial automation, the automotive domain particularly with entertainment and driver assistance subsystems, military applications e.g. for secure communications, medical imaging, and consumer devices like television set-top boxes. Overall the importance of these sectors increases, but no single domain matches the communications sector yet [278, 18]. For the vast majority of applications in these domains, the characterization is similar to that in the communications segment. FPGA-based systems are marketed as appliances with a fixed and narrow functionality. Their FPGAs are special-purpose components configured with a customized design and executing a task that is fixed before system fabrication. The FPGA's programmability however, is again valuable to reduce time-to-market and to allow for updates and bug-fixes. Considering these diverse markets, in contrast to the FPGA-based embedded systems, the FPGAs themselves, as fabricated chips from Xilinx or Altera, are general-purpose products, which considerably profit from the opportunity to share NRE costs among different customer needs. In particular, due to the high volumes of the combined FPGA markets, FPGAs can be produced with more advanced manufacturing technology than most ASICs and thus overcome or reduce efficiency and volume cost advantages of custom hardware.

In contrast to the above markets, there are two subdomains showing up in both FPGA manufacturers' reports [278, 18], where programmability of FPGAs is not just a bonus, but a prerequisite for the respective usage scenario. The first market is summarized as testing, the other one is high-performance computing (HPC). In the testing domain, FPGAs are used to generate test inputs for other circuits and analyze their outputs, but they are also employed to emulate or prototype new ASIC products. For these applications, the possibility to program and change the FPGA's functionality after fabrication time is crucial. Hence, if the testing domain can be at all characterized as one of the computing domains introduced above, it falls into the general-purpose realm.

For HPC, FPGA-based compute products have been introduced as accelerator components, attached to secondary CPU sockets of multi-socket mainboards [273], as PCIe accelerator cards in typical external GPU form factors or as addition to customized mainboards [213]. Two of these FPGA-accelerated compute systems, a Convey HC-1 [40] with in-socket FPGA accelerator and a Maxeler MPC-X system [175] with PCIe-attached FPGAs, are the target of the practical evaluations presented in this thesis and will be presented in more detail in Section 4.3. Architecturally such systems are capable of performing a wide range of general-purpose tasks, which is reflected by actual usage in academia. Just in our research group, the two systems have been utilized for diverse tasks like image pro-

cessing [4, 5, 6, 8], motion estimation for video encoding, finite difference time domain (FDTD) simulation [184, 90] for nanophotonics, short read mapping in bioinformatics, heat transfer simulation [8], Markov chain steady state computation [8], correlation matrix calculations [8], and cryptographic key search [10] and reconstruction [9].

However, it seems that, up to now, the commercial success of many FPGA compute products is closely linked to focused support for a few application classes and production systems often get deployed exclusively for one application. For example, in the financial sector, low-latency packet processing [266, 192] and asset pricing using Monte-Carlo methods [270] are applications where FPGAs are successfully utilized. Several standard methods from bioinformatics, such as Smith-Waterman, BLAST and hidden Markov models are, apart from academic research [287, 141], often marketed as integrated software and hardware solutions using FPGAs [115, 132, 28]. Stencil computations from seismic analysis for oil and gas exploration are implemented on FPGA products [84]. Academia has also presented various cryptographic applications for FPGAs [160, 104] and [10, 9]. We can assume that professional users in this domain have large FPGA installations for such applications, but prefer to remain unknown. To summarize the commercial usage of FPGA-based compute products, many systems are used like special-purpose products. They are purchased for a single task or narrowly defined task set, sometimes even designed for that purpose with specific performance targets. Third-party software often can be executed, but actually is not. Nevertheless, architecturally, most of these FPGA-based systems are general-purpose capable, as the academic use-case above underlines.

In contrast to the previous examples for commercial use of FPGA compute products, the recently emerging Microsoft Catapult system, first presented to accelerate the Bing search engine [213], is explicitly aimed at a broader range of workloads in the datacenter [191]. In other words, a key factor for the Catapult system architecture is to leverage the general-purpose computing capabilities of FPGAs. Research on new application fields for these systems, for example deep learning, are no longer purely driven by the desire to use the best available hardware design for an application, which might be a larger GPU installation as of now. Instead, in order to harvest consolidation benefits in its datacenters, the company also considers it beneficial to have further useful applications that can efficiently make use of the existing FPGA accelerators.

To summarize this subsection, we presented our view on the current state of FPGA markets. In terms of revenue, embedded devices still clearly dominate, but there are success stories of FPGAs accelerators in HPC. Yet, the trend to fully embrace FPGAs as general-purpose compute products is still in its infancy. In the next subsection, we further structure the general-purpose markets, before in Chapter 3, we discuss reasons for the current low adoption and requirements and our ideas for further adoption of FPGAs in general-purpose computing.

2.3.3 Markets and Workloads of General-Purpose Computing

In Subsection 2.3.1, we have defined general-purpose computing in contrast to special-purpose computing. In this section, we now give an overview of the markets for general-

purpose computing, characterize the workloads encountered there and assess the potential role of FPGAs.

General-purpose computing can on the highest abstraction level be subdivided into personal and server usage.

Personal and Mobile Computers

Besides the proverbial PCs, which fall into the categories of desktop and laptop computers, devices for personal use encompass tablet computers and smartphones. Used both for professional and consumer purposes, the common characteristic of these devices is that they are used by one person at a time and typically one or sometimes few persons over time.

A usage study on Windows 7 systems with Intel CPU, thus covering only PCs, classifies application domains as "Web, Communication, Office, Media Consumption, Media Editing, Game, Utility, Network Apps and IT" [215], with usage categories ranked by user interaction time. The analysis of CPU usage roughly follows this sequence, with gaming causing disproportionally more and office applications causing disproportionally less CPU usage than indicated by the interaction time. Additional load is caused by system processes and anti-virus software, not detailed further. For the groups of applications the users deliberately interact with, further descriptions and examples are given, for example "Communication" includes "VOIP, instant messangers [and] email", "Utiliy" includes "Backup, archiving, tuning [and] print", "Network Apps" include "Peer-to-Peer, remote desktop [and] FTP" and "IT" includes "Software development [and] databases" [215].

The diversity of these groups, which also tend to have very different characteristics in terms of available parallelism and computational requirements relative to memory and IO behavior, underline the challenge of general-purpose platforms to provide good performance for any given workload. The study also identifies distinct clusters of users. The usage patterns of the four largest groups put special emphasis on web, office, gaming and media consumption respectively [215]. Next to the challenge of versatility, this also underlines the value of sharing NRE among different user groups. CPU vendors are successful in this regard and differentiate between individual user demands and price points by matching performance classes and features, which are often not inherent to the sold products, but artificially inflicted. The development of GPUs has been fueled by demand from users with gaming emphasis, but now also serves other user interests, both within the same device class, such as media consumption and editing, and beyond. Upcoming FPGAs adoption in the PC market is unclear and may depend on a single, sufficiently important, usage scenario, where they provide sufficient added value alone, before being also used for other purposes. In 2011 and 2012, a small wave of FPGA computing devices was integrated in or attached to PCs for Bitcoin mining [251], however they were soon surpassed by fully custom ASICs and never formed a critical mass.

Within the market of personal general-purpose computing devices, there is a strong trend towards higher mobility of devices, with sales of desktop computers peaking in or before 2010, laptop computers in 2011, tablet computers in 2014 according to forecasts,

and smartphone sales still growing [134, 38]. With mobility, the requirements for energy efficiency are growing and have contributed to the trend to specialized accelerators in smartphones as discussed in Subsection 2.2.3. For the market of personal mobile devices, say smartphones and tablets, we have no detailed application domain data as for the PC market, but we can note that known accelerators can mostly be attributed to the media consumption and editing and gaming domains, or fall into the system domain. ARM, licensor for most processors in the mobile domain, predicts a trend towards technologies enabled by more and more accurate sensor data, such as noise cancellation, location awareness, recognition of gestures and handwriting [24]. Related tasks involve signal processing workloads, which are well suited for acceleration with custom circuits or FPGAs. NRE costs and time-to-market can ignite FPGA utilization in this domain. For tasks that don't require acceleration at the same time, temporal reuse of the same FPGA accelerator may provide an additional cost advantage, however in the era of dark silicon needs to take energy efficiency into account. Whereas FPGA energy efficiency cannot compete to custom accelerators implemented in the same process technology, it can conceptually do well compared to the general-purpose components of mobile SoCs. In practice, FPGAs may need to embrace energy savings techniques like power gating more resolutely in order to compete in this market. However, after FPGAs enter this market as accelerators with limited scope, there are incentives to use them for further, more general-purpose computing tasks.

Sever Class Computers

In contrast to personal devices, server class computers typically handle requests or tasks from many users, often concurrently, but also over time. Hennessy and Patterson [118] distinguish between servers and warehouse-scale-computers, distinguished not only by size, but also workloads, system architectures, and approaches for failure handling. In the revenue breakdown of Intel's data center group, we find as rough correspondences to servers and warehouse-scale-computers the categories enterprise and cloud and a third category combining HPC, workstation, networking and storage [190]. According to Intel's projection for 2016, each of the three categories will make roughly equal contributions to overall revenue.

Typical workloads for servers in the enterprise domain contain transactions on databases, along with business logic and websites. These workloads are dominated by integer operations and mostly contain thread-level parallelism through different requests, however the transactional character [107] of operations on the databases and business logic requires careful synchronization of threads. The three tasks of databases, business logic and web frontends are often distributed to distinct computers, such that each individual system often performs just a single task type over time. This offers room for customization [24], which started with configuring core counts and frequencies, memory hierarchy and storage, but can also include customization of processor ISAs to ASIPs. However, so far the tasks for which the systems are specialized seem to be too complex to be entirely performed on specialized accelerators.

Warehouse-scale-computers [118], or systems of hyperscalers as they are called by Morgan [190], combine many commodity or server class computers into huge clusters and achieve availability of services through redundancy of running nodes and management layers in software. They execute as one part of their workload similar tasks as enterprise servers. However, their typical search and social media services use much relaxed transactional requirements, denoted as eventual consistency [262] and thus can easier exploit the thread-level parallelism of different requests. Another typical application class for warehouse-scale-computers are map-reduce [68] algorithms through frameworks like Hadoop[3] and Spark[4], now both maintained by the Apache Software Foundation. The map-reduce pattern is specifically designed to distribute the processing of large datasets ("big data") among many server nodes.

Through the paradigms of software as a service (SaaS) and cloud computing, manifold tasks from personal and mobile computers, as well as from classical servers, have been partially or entirely moved to warehouse-scale-computers. For the hyperscalers, the central value proposition is consolidation. With resource demands that can be partially steered by pricing, the systems can run most of the time at the desired load levels that allow for maximal efficiency. For users, services running externally, "in the cloud", open up the chance to obtain task-specific performance levels without up-front hardware investments and with little impact on the local power budget. Through SaaS and cloud computing, the workload of warehouse-scale-computers is very diverse in terms of available parallelism and predominant operations, from integer dominated tasks with high requirements for single-thread performance over throughput oriented tasks to SIMD-favorable media processing. The striving for efficiency has led to the customization of warehouse-scale-computers or parts of them [213, 116, 190], which however somewhat conflicts with the consolidation paradigm. While GPUs are successful in the media processing domain and recently particularly shine for neural network processing and deep learning [116, 235], they are hardly suitable for other cloud workloads with predominantly request-level parallelism [213]. Hence, in retrospective, it comes as no surprise that such workloads inspired the first large-scale deployment of FPGAs in the data center [213], as indicated in Subsection 2.3.2. Once such FPGA installation are available, further workloads can use them, whenever they overcome absolute performance limitations as in [213], or are more cost effective as in [191]. Accordingly adapted applications can also migrate back to enterprise servers that for other reasons like privacy are not replaced by cloud solutions.

The paradigm of On-The-Fly (OTF) computing that is proposed and researched in the collaborative research centre (CRC) 901 "On-the-fly Computing" at Paderborn University goes beyond the established SaaS and cloud computing paradigms. OTF computing revolves around the idea of individually and automatically configured information technology (IT) services, whose components are traded on markets, can be flexibly combined and executed on-demand [185, 186]. The research of CRC 901 encompasses techniques for configuring and executing these services [110] as well as the required market mechanisms and infrastructure. Many results presented in this thesis are partially driven by the demand

[3] `https://hadoop.apache.org/`
[4] `https://spark.apache.org/`

of efficient execution of services in this context. We expect the workloads of On-The-Fly (OTF) computing to be at least as diverse as those of warehouse-scale-computers, however with much more versatility and variations over time because services are not executed as-is, but configured and adapted based on individual demands.

In order to be commercially successful, the execution such OTF services should not fall short of the efficiency obtained by established hyperscalers for their less configurable services and therefore needs to make use of any accelerator technology that is established there. Hence, as we see FPGAs gaining traction in the cloud, we want to make sure that the methods to use them can cope with the dynamics and versatility of the OTF paradigm through fast and automated compilation and offloading techniques. Due to the approach of decoupling software catalogs, service composition and execution in compute centers through markets, there is also an increased demand to enable target-specific optimizations directly prior to execution, at a compute center that may just have compiled binaries at its disposal.

Finally, HPC is yet a different segment in the market of server class computers. In this segment, typically compute nodes with higher individual performance than in warehouse-scale-computers are interconnected with higher bandwidths and lower latencies than in warehouse-scale-computers. The design goal for these systems is to achieve the highest possible performance that allows to solve computational problems that can otherwise not be solved within acceptable time frames. The workloads are mostly numerical simulations of phenomena from the physical world. These simulations tend to use many more floating point arithmetic than other workloads and often can and need to exploit DLP through SIMD instructions and through parallel processing on many nodes to achieve the required performance. Consequently, many applications from this domain are well suited for GPU architectures and many early success stories of GPUs as general-purpose devices [201] cover HPC applications.

When data-parallel floating point performance along with massive off-chip memory bandwidth is required, current FPGA computing products can currently not match GPUs. Similarly, GPPs can be superior when HPC applications map well to their cache hierarchies and profit from their SIMD units. In terms of energy efficiency FPGAs can already be the best platform even in such a domain [149]. As long as it is possible to reach the performance goals, users of HPC systems in academia, who often don't pay for power and cooling, are relatively insensitive to efficiency considerations. However, when HPC systems hit practical power limitations, more power efficient systems are mandatory and are likely to make use of FPGAs for their efficiency and versatility [229].

Also, whenever FPGAs can profit from customization of operations, data types or local memory reuse, they can already overcome GPUs and GPPs in raw performance [247]. Additionally, recent Altera FPGAs reduce the gap in floating point performance [191]. Even though sometimes bound to existing codes for a long time, computational scientists are willing to put extensive effort into overcoming performance boundaries, for example through higher-order methods that reduce communication overheads, as exemplary illustrated by [156], so low adoption of FPGAs in the HPC domain is particularly surprising. Beyond the general considerations around productivity that we introduce in Section 3.1.3

Table 2.8: Summary of opportunities for FPGAs in general-purpose markets.

Market	Drivers for FPGA usage	Alternative
Mobile computers	NRE costs, time-to-market, reuse	Custom accelerators
	Energy efficiency, performance	Mobile GPPs, GPUs
Datacenter, OTF	Energy efficiency, performance	GPPs
	Energy efficiency, versatility, performance	GPUs
HPC	Practical power limits, performance, customization	GPP, GPU, Manycore

and tackle within this thesis, a HPC-specific issue may be the focus of the FPGA community on strong scaling, that is acceleration of a workload with fixed problem size. Weak scaling, that is the ability to solve larger problem sizes on larger HPC systems is more important for many computational scientists, but may have been neglected by FPGA researchers because of the tight links to the embedded community.

Opportunity Matrix

After this brief excursion into markets of the general-purpose computing domain, we summarize the architectural and usage potential for FPGAs in this domain with Table 2.8. For mobile computers, we see them enter as alternative to a further growing collection of custom accelerators. In large-scale datacenters, they have recently entered for performance and energy reasons and seem to fit well to workloads with request-level parallelism. In the HPC domain, they will be needed under practical power limits and can also increase absolute performance.

To summarize this section, we have first characterized the distinction between general-purpose computing and special-purpose, embedded computing. We have outlined how FPGAs up to now are mostly used in the latter domain, but are architecturally and according to individual success stories interesting candidates for general-purpose computing. Investigating the markets for general-purpose computing in some more detail, we see indicators that FPGAs may be even more promising as accelerators than GPUs.

2.4 Chapter Conclusion

In this chapter, we presented different paradigms of computing and how FPGAs combine the programmability of CPUs with the design concepts of custom circuits. We have discussed how the technology and architecture trends, with increasing pressure towards efficiency and difficulties to translate more transistors into higher performance, point towards programmable accelerators, such as GPUs and FPGAs. Investigating computing domains

and markets, we have presented how FPGAs are up to now mostly used for special-purpose computing, but how various general-purpose computing markets could profit from their flexibility as accelerators. We have introduced individual success stories of FPGAs in this domain, but stated that overall adoption trails that of GPUs and manycore architectures.

We have intentionally not discussed productivity as central reason for this discrepancy between perceived potential and actual adoption within this chapter. In the following Chapter 3, we discuss this gap, and present our three-pillar approach to overcome it.

CHAPTER 3

The Case for general-purpose adoption of FPGAs with the help of Overlay-Architectures

After focusing in the previous chapter on architectures and their performance drivers for different usage scenarios, in this chapter, we discuss how well-performing implementations of general-purpose workloads need to find their way to FPGAs to reach this market. In Section 3.1, we observe how performance and productivity considerations go hand in hand for general-purpose architectures and investigate how FPGAs currently fit into this frame. Based on this analysis, we propose a three-pillar approach to general-purpose adoption of FPGAs. Besides libraries and accelerator-friendly OpenCL specifications, overlay architectures with accompanying productive design flows can play an interesting, but insufficiently understood role in this approach. Section 3.2 structures related work on overlays under this premise. Thus, this chapter combines on the one hand an analysis with own original positions and on the other hand a broad coverage of related work.

3.1 Between Performance and Productivity Walls

In this section, we first review how productivity, binary- or code-compatibility and performance scalability affected the adoption of new architectural paradigms in general-purpose computing. We briefly touch the effect of system architectures on productivity, before discussing how FPGAs are only slowly catching up with regard to performance productivity. Based on this observation, we introduce a three-pillar model for FPGAs in general-purpose computing, of which our specific focus is on overlay architectures.

3.1.1 Productivity of General-Purpose Architectures

The historic success of general-purpose CPUs largely built upon two pillars: firstly, on the straight-forward programming model with — from a programmer's perspective — se-

quential execution of a single instruction stream, and secondly on the "free lunch" [248] of *performance scaling*. In the era of geometric scaling, this "free" performance scaling allowed to get higher performance for the same applications from every new processor generation and to a lesser degree from more expensive processor models of the same generation. As discussed in Subsection 2.2.2, besides frequency scaling, this was achieved through the exploitation of ILP. Dynamic superscalar architectures achieve performance scaling through ILP in hardware, thus allowing for *binary compatibility*, whereas most VLIW architectures require recompilation to make use increased ILP, but offer performance scaling at *code compatibility*.

The comfortable situation of perceived free performance scaling allowed many software developers in general-purpose computing to focus much more on design productivity and on introducing new application features, than on low-level performance engineering. Naturally, the high-level runtime complexity of utilized algorithms could hardly be neglected for most programs running on all but trivial data sizes, but optimizations for specific features of processor architectures were often left aside, or even not exposed by high-level programming languages.

Multicore processors as major architectural paradigm in the era of equivalent scaling changed this situation. This paradigm essentially offers the choice between either binary compatibility at roughly stalling performance levels for sequential applications, or performance scaling at the expense of redesigning many applications around concurrency. This was perceived as a shock by the software development community [248, 249] and was dealt with in very different ways in the various market segments introduced in Subsection 2.3.3. In the domain of personal computers, acceptance for dual core processors was fueled by perceived performance gains, when a single interactive application can use all the single-threaded performance of one processor core while background and system processes run on the second core. However, individual applications only slowly moved towards exploitation of several cores, lead by media processing tasks with much DLP. In contrast, many server workloads contain sufficient request-level parallelism. However, particularly for transaction dominated workloads, programming for concurrency required considerably more effort. Virtualization and cloud computing facilitated the path to make use of multicore processors by deploying several service instances per server [193]. The HPC community overall was probably best prepared for multicore architectures, with established programming patterns for parallel processing. In this domain, shared-memory systems were already known before they became the dominating architecture for single compute nodes with multicore processors. For such shared-memory systems, the OpenMP[1] API is widely used to handle parallelism. In the most straightforward way, it distributes the iterations of parallel for-loops to different threads on different cores and thus primarily exploits DLP. More recent incarnations of the OpenMP standard focus on increased support for TLP through models for independent [196, 203] and dependent [204, 195] tasks. A particular benefit of OpenMP runtime system is the ability to specify the number of threads for each execution, thus for applications with the right form and amount of parallelism, it combines performance scaling with binary compatibility for multicore architectures. Intel's Cilk

[1] `http://openmp.org/wp/`

Plus[2] language extensions and compiler together follow a similar approach to OpenMP by exposing parallelism through pragmas and scheduling tasks to different CPU cores at runtime. Suitable parallelism from for-loops is also translated into SIMD instructions.

SIMD units are, orthogonally to multicores, another architectural feature for parallelism that often requires to choose between binary compatibility and performance scaling. Between the fully automated generation of SIMD instructions by compilers, which retains code compatibility but often yields limited performance gains [172], and the manual specification of SIMD instructions in assembly language, different approaches with intrinsics [288], pragmas [157][4] and code generation [102, 150] have been explored, and generally involve trade-offs between productivity and resulting performance. Along with partitioning and offloading aspects, SIMD code generation for an FPGA-based platform is a central theme in Chapter 5. Hennesey and Patterson [118] note that SIMD architectures are generally easier to program than MIMD architectures like multicores. This is mainly because they don't exhibit possible race conditions between different threads, but are limited to DLP. Also, many measures to use the SIMD units of general-purpose CPUs can remain local due to fine granularity and low overheads to switch between SIMD and sequential instructions.

In contrast, GPUs target DLP with SIMD and SIMT paradigms at a much larger scale. With the emergence of these architectures, binary compatibility and code compatibility to CPUs has been given up with CUDA as first successful specification language for GPUs as general-purpose computing devices. Soon after, with OpenCL another specification language was developed that promises portability between multicores, manycores and GPUs for kernels specified with it. Through online compilation, the actually code compatible kernels can be perceived as binary compatible between different architectures, however performance portability and scaling between different architectures is highly fluctuating. Later, with OpenACC, another programming standard has been presented that portably targets GPUs and manycore architectures through compiler directives. For an intermediate representation generated from OpenACC code, Sabne et al. [220] evaluated the performance portability and show that code variants optimized for one GPU model achieve on a GPU from a competing manufacturer 85% to 91% of the peak performance on the latter platform. Between a manycore platform and the GPU platforms, performance portability is more limited leading to between 65% and 74% of peak performance.

We see three essential factors that contributed to the success of GPUs accelerators in general-purpose computing despite the need to write new code with a new programming language or API when starting from a CPU implementation.

1. The promise of significant performance gains with one or two orders of magnitude [201] using widely available and competitively priced hardware and compelling roadmaps towards even faster upcoming GPU generations justified the development effort. As competitor, it is exactly the magnitude of these performance gains, that Intel challenged in [166].

[2] `https://www.cilkplus.org/`

2. An educated guess about the applicability of GPUs for an application type is relatively easy [269], for example based on available DLP, simplicity of control flow, arithmetic intensity and off-chip memory bandwidth.

3. GPU manufacturers, particularly Nvidia, invested heavily into compilers, debuggers, performance analysis tools, documentation and examples, and made most of these available free of charge [80, 162, 256]. This allowed for a steep productivity curve and low entry hurdles for the new architectures.

4. The SIMT programming pattern of CUDA and OpenCL allows to effectively cover parallelism both for SIMD and multicore execution, whereas with many other formalisms established in the CPU domain, these two need to be handled separately.

For many markets, another important consideration was the commercial availability of general-purpose GPUs from two independent manufacturers and the code compatibility between the two within the OpenCL language [80] and later on via OpenACC.

In the segment of manycore architectures, the first commercial product generation from Intel, based on the Knights Corner architecture, offers limited binary compatibility to the multicore CPU product line. Since the SIMD execution units have different instruction formats, code compatibility for performance scalability depends on the effectiveness of the compiler support for specific codes. The code compatibility is however perceived as an important factor, why manycore accelerators were able to catch up to GPUs in the HPC domain quickly [256, 162]. With the following Knights Landing architecture, Intel aims at full binary compatibility to the multicore products, with performance scaling that just depends on the the degree of exposed SIMD and multicore parallelism.

Beyond this rough overview of compatibility and specification methods between and for different architectural approaches, it has to be noted that for some applications, efficient utilization of the respective architectures is achieved through underlying libraries with highly optimized implementations. For example for the Basic Linear Algebra Subprograms (BLAS)[3] library for linear algebra, among other development communities, several hardware manufacturers like Intel, AMD and Nvidia maintain implementations that are tuned specifically to the respective features of their compute devices. Using libraries to achieve hardware specific optimized performance is compelling from a productivity perspective, but limited in scope. Libraries with very fundamental functionality are applicable to many applications, but often just cover small parts of the overall program runtime. More specialized libraries have a higher chance to cover more of the performance relevant parts of an application, but are only applicable within narrow problem domains.

Overall, we conclude that after the end of free performance scaling with binary compatibility, developers have taken up the challenge to employ different new programming techniques and paradigms in order to benefit from the performance opportunities that new architectural features offer in various domains. However, in order to keep productivity high, effort and expected gains seem to be carefully traded-off. Many of the successful

[3] `http://www.netlib.org/blas/`

new programming paradigms for parallelism offer not just one-time performance gains, but after the transition also open up a new path to future performance scaling with code or binary compatibility.

3.1.2 Impact of System Architectures

Within libraries and beyond, the granularity of individual code segments that can profit from specific hardware features and architectures tends to increase from SIMD instructions, which only use different registers within the same CPU core, over homogeneous multi- and manycore architectures, to heterogeneous architectures, where GPUs, manycores or FPGAs are used as external accelerators to a general-purpose CPU. The interplay of homogeneous or heterogeneous resources and functional and efficient memory management between those can be an additional hurdle for developers.

For homogeneous multicore architectures, the challenge for programmers to ensure correct functionality with concurrency and synchronization has already been mentioned in previous sections. In order to optimize performance, tasks and subtasks also have to be distributed to different cores with patterns that minimize communication and maximize data locality in memory blocks that are tuned to the specific cache hierarchy of a multicore processor. Depending on the applications, this can require significant efforts by the programmer, or can be automated, for example with polyhedral optimization methods [36].

For many current systems with accelerators, memory placement and synchronization are not just a performance challenge, but separate memory spaces and the process of offloading tasks outside of the scope of the OS require appropriate solutions already for mere functionality. Around this challenge, significant activity from hardware manufacturers can be observed. Nvidia in two steps introduced a runtime system that allows programmers to use a programming model based on shared memory, even though the underlying hardware still needs to move data between two distinct memory locations [109]. The Heterogeneous System Architecture (HSA) foundation[4] around AMD and ARM promotes shared memory architectures for heterogeneous on-chip integrated systems around CPUs, GPUs and DSPs, and introduced a common intermediate language to program such systems [128]. Similarly, under the roof of the OpenPOWER foundation[5], shared memory protocols for external accelerators are specified, which let IBM POWER CPUs interface with Nvidia GPU via NVlink and FPGA via Coherent Accelerator Processor Interface (CAPI). With the upcoming manycore products based on the Knights Landing architecture, Intel moves the manycore processors back from accelerator boards with an offloading model to homogeneous system architectures. The announced Xeon processors with FPGAs will be heterogeneous, but are expected to share main memory between CPU and FPGA. In Chapters 4 and 5, our presented solutions deal as one aspect with the challenges of distributed memory and shared memory with non-uniform memory access (NUMA) characteristics with library and compilation approaches respectively. Looking further into the future, in Chapter 6,

[4] `http://www.hsafoundation.com/`
[5] `http://openpowerfoundation.org/`

we analyze the potential of current trend to integrating reconfigurable accelerators into a shared memory hierarchy.

Overall, heterogeneity and separate memory spaces are a challenge for FPGA acceleration, and currently integration of FPGAs in heterogeneous systems is trailing that of GPUs. However, when general-purpose computing with GPUs gained traction, early GPU accelerators were no more closely integrated into systems than FPGAs are now. Therefore, the challenge of heterogeneous systems may have slowed, but obviously has not prevented the success of GPU accelerators in several general-purpose computing markets. In the PC domain, GPUs may have profited from a dual usage scenario of computing and graphics rendering, but in the HPC and cloud domains, the impact of graphics rendering capabilities to GPU adoption is minimal. Therefore, we further need to investigate the programming models for FPGA accelerators themselves as a major factor that currently limits their wide-spread adoption.

3.1.3 FPGA Productivity

On an abstract level, until recently, application developers needed to adopt new programming models to offload code from general-purpose CPUs to FPGA accelerators, just like for early GPU accelerators. However, the traditional design route via HDL with VHDL and Verilog, as outlined in Subsection 2.1.4, requires not only to learn new programming languages, but the underlying programming model is very different from software programming models. The designers not only need to combine structural and behavioral descriptions, the behavioral description also gives up the sequentiality of most software programming models and instead requires the designer to specify anything that can happen in each cycle [113]. Thus, the move to FPGAs as accelerators is much more time-consuming and difficult than to multicore and GPUs architectures. Also for experienced developers, design productivity is arguably lower for circuits and programmable hardware than for software. In particular, the compilation and synthesis flows takes much longer than software compilation, and simulation and testing is slower because the simulated target systems are structurally different from the host systems, on which they are designed.

Code compatibility of HDL designs between different FPGA models, also from different manufacturers, is conceptually given, but requires the new target FPGA to be sufficiently large. In practice many more problems can arise, from the availability of compatible interfaces in hardware and in configurable fabric, over incompatible low-level optimizations for specific hardware components like LUTs and DSP blocks, to timing issues during the synthesis phase. Even performance scaling to larger or newer FPGAs can be problematic, as many design decisions in HDL tend to depend on specific performance targets or resource constraints and may not be easily parametrized for flexible trade-offs between area and performance.

FPGA manufacturers, tool designers and academia have long worked on improving design productivity under the label of high-level synthesis (HLS). The idea of HLS is to generate FPGA or hardware designs from more abstract and more familiar, software-like programming models and language constructs. A main challenge for HLS is, that sequen-

tial, C-like programming models and languages seem to be most appealing to application programmers that want to accelerate their software with reconfigurable or customized hardware, but in order to generate efficient hardware, the synthesis tools need information about concurrency, timing and customized memories or buffers [76].

However, recently, increasing numbers of developers from the FPGA domain consider tools like Vivado HLS[6] (formerly AutoESL [177]) as viable design path that is able to produce functional FPGA designs from many C and C++ sources out-of-the-box and provides the designer with effective ways to influence the generated design for performance or area critical parts of the code through pragmas or directives. Thus, both for experienced and new FPGA developers, HLS significantly facilitates the steps to a first working design and opens up a path to incremental improvements. For such optimization steps, a deeper of understanding of the target architecture is still important.

The HLS tools typically perform source-to-source transformations form C-like languages to HDL and thus depend upon the HDL synthesis flows. Thus, some of its problems like less-then-perfect code portability and long tool runtimes can also be observed with HLS. However, HLS tools like Vivado HLS also support faster simulation on a higher abstraction level and thus require fewer of the very time-consuming simulation and synthesis steps of HDL. Compared to bottom-up HDL designs, the approach of incremental design optimizations also greatly increases productivity when design decisions like parallelism or data buffering need to be changed late in a design. This also brings HLS approaches closer to performance scalability at the source code level, because in some designs, parallelism can easily be adapted to the available resources.

One HLS variant, for which both Altera and Xilinx particularly pushed forward their respective tool chains recently is based on OpenCL as specification language [65]. This approach is attractive because the language is already designed around parallelism, offloading and different memory spaces and thus requires less target specific pragmas to generate corresponding FPGA designs. The OpenCL synthesis flows offer, within their respectively supported versions, full code compatibility to other accelerators and thus give access to existing code bases and to a pool of additional potential FPGA developers. However, performance portability from other architectures or between different FPGAs, as well as performance scaling are two more ambiguous properties. Sometimes, the parallelism and memory concepts from OpenCL can with little or no effort lead to efficient designs on FPGAs, for example when local and private memory regions are a useful way to use BRAM resources, or when data parallelism can effectively be used for latency hiding or unrolling. In other situations, the OpenCL code must be considerably re-factored to transform it from GPU to an efficient FPGA design [257, 148]. Other customizations that can contribute to the efficiency of FPGA, for example of data types and operations, cannot be expressed with OpenCL at all.

Orthogonally to HLS, a spatial programming model around the language MaxJ or its open source variant OpenSPL[7], retains much of the structural design aspects of HDL, but significantly raises the abstraction level. It recently emerged as a more productive

[6] `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`
[7] `http://www.openspl.org/`

alternative to HDL for dataflow-centric computations and is used and evaluated as one design path in Chapter 4.7. The core of MaxJ is an explicitly structural design approach with basic elements like directed streams of data, arithmetic or logic operators on these streams, and constructs to modify the streams, for example delay elements or multiplexers. However, the synthesis toolchain for MaxJ is currently limited to target FPGA boards from Maxeler. In our work, we present how parameterizable designs in MaxJ can provide some performance scalability, but are also constrained by other design considerations.

Despite this manifold progress to increase the productivity with FPGA design flows, we must note that the value proposition of performance of productivity for FPGAs does not appear as convincing as for GPUs.

1. Even thought the general performance potential of FPGAs has been demonstrated extensively, the potential for specific applications is often unclear because of the multitude of optimization strategies for FPGAs designs. Since FPGA optimization cannot start from a competitive baseline performance point, but rather first needs to overcome a large clock speed penalty. Leaving the actual computations aside, the memory interfaces of many FPGA accelerators neither offer the raw bandwidth of many GPUs accelerators, nor the convenience of extensive cache hierarchies that transparently offer reasonably good performance for many different memory access patterns, and therefore need to be included in the analysis.

2. The programming models for FPGAs involve trade-offs between difficulty, productivity and performance potential that are hard to grasp for software programmers. The most accessible ones, HLS and OpenCL are at best not more difficult than software programming models. Often, in order to come close to the available performance potential, either sophisticated optimizations for HLS are required, or only low-level HDL designs are suitable.

3. Once a design is successfully accelerated on an FPGA, portability to other FPGAs is more restricted than for other architectures. A somewhat limited code compatibility to other FPGAs exists, but even in the best case requires a new synthesis process that is orders of magnitude slower than recompilation of OpenCL kernels for a different GPU. Also performance scaling to newer or larger FPGAs can be difficult.

For the HPC domain, Laakso [162, 256] compares the adoption of manycores, Nvidia and AMD GPUs and FPGAs as accelerators after their first availability. He postulates that adoption of these architectures between fast, for manycores, over slower, for Nvidia and AMD GPUs respectively, to non-existing for FPGAs is closely related to the programming paradigms and development tools. The recent progress in synthesis from more abstract programming models may enable a first entry of FPGAs into this segment, but is it sufficient to pave considerable adoption in the HPC and other markets?

3.1.4 Three Pillars for General-Purpose FPGAs

Based on the observations in the previous subsections, we argue that three pillars can complement each other to make FPGA programming flows versatile enough to drive general-purpose adoption of FPGAs, similar to the general-purpose adoption of GPUs within the last ten years. Most practical contributions of this thesis pertain to the second pillar.

Progress in HLS and particularly OpenCL synthesis flows put FPGAs closer to other accelerators in terms of accessibility than they have ever been. Judging by the visible activity of tool development and documentation, OpenCL seems to be the tool of choice for Altera to further computing adoption of FPGAs and Xilinx seems to increasingly gear their HLS effort in this direction. Early case studies [53, 189] indicate, that FPGAs might be able to play the role of slower, but more energy efficient complement to GPUs with this approach. However, the OpenCL programming model's close ties to a GPU-like execution model limits or obfuscates the further customization and optimization potential of FPGAs. Also, in direct comparison to other target architectures, tool runtimes are still a huge disadvantage and are an additional factor that limits performance scaling. Therefore, we consider OpenCL not as comprehensive path to broad general-purpose adoption of FPGAs, but rather as the *first* of three *pillars*, upon which this outcome can be built.

Hence, as *second pillar*, we see the need for an approach that allows faster synthesis or compilation and can exploit other, orthogonal features of computing on FPGAs. For this approach, we propose to use overlays, that is computing architectures that are implemented as FPGA designs, but retain some programmability or configurability that enables them to perform different tasks. The indirection of an overlay limits the performance potential, but the abstractions and fixed design aspects of each specific overlay can enable faster and more automated compilation or synthesis than for the full configuration space of the underlying FPGA fabric. Such overlays have been researched in academia from various perspectives, which we discuss in more detail in the upcoming Section 3.2. The focus of overlay architectures thus far has mostly not been on general-purpose computing scenarios and the potential seems not yet sufficiently understood to play a major role in corporate roadmaps for FPGA acceleration. In Chapter 4, we show that the overheads of such an overlay can be acceptable in a practical application and in Chapter 5 we present contributions to a compilation flow for an overlay.

As *third pillar*, we summarize all efforts to encapsulate FPGA designs into reusable libraries. As outlined in Subsection 3.1.1 for established general-purpose architectures, this approach allows easy and fast acceleration for applications that have core functionality covered by libraries, and also allows highly optimized designs since design efforts can be shared within developer communities around such libraries, or taken over by hardware manufacturers. On the other hand, library coverage of applications and within individual applications is limited. When individual libraries and FPGA accelerator boards are closely tied together, like for some products in the bioinformatics domain [115, 132], the role of these boards remains limited to special-purpose acceleration, as discussed in 2.3.2. The ap-

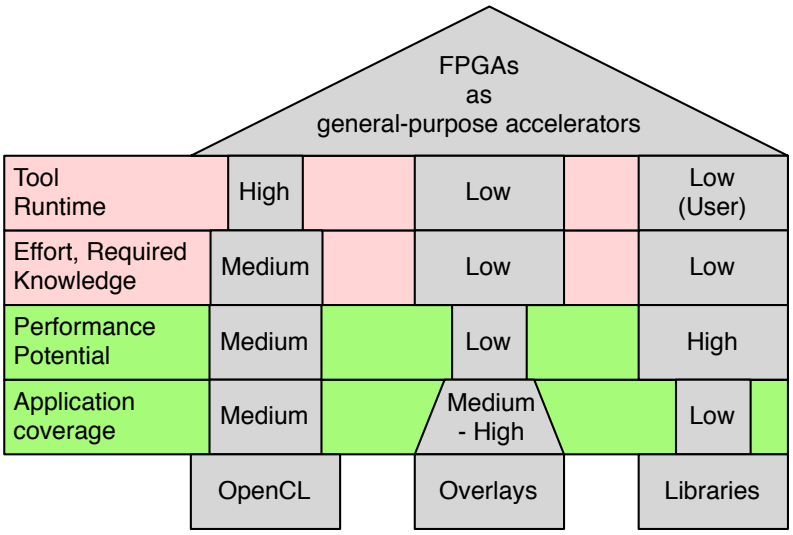

Figure 3.1: Illustration of three pillars for FPGAs as general-purpose accelerators. The width of each pillar segment indicates the suitability of each pillar per category. Note that high application coverage and performance potential, but low required knowledge and effort and low tool runtime are the target properties.

plication galleries from Maxeler[8] and IBM[9] with computational kernels among others from the domains of bioinformatics, big data analysis, compression cryptography and computational sciences, in contrast point towards a strategy of diversifying the use of accelerator hardware through a library approach. Intel is also rumored to work on application libraries for their upcoming CPU-FPGA products. A similar strategy by Nvidia earlier on played an important role to pave the way for GPUs in general-purpose computing.

We summarize key properties of these pillars in Figure 3.1. Together, we consider the three pillars of synthesis of FPGA accelerator designs from OpenCL, overlays with software-like compilation, and libraries with hand-optimized accelerator designs, as suitable to carry adoption of FPGA accelerators in general-purpose computing markets. In contrast to conventional HDL design, neither improves the performance side per-se, but they offer easier and faster entries to explore FPGAs as target platform and can together cover many optimization approaches. In case of a successful, wide-spread market penetration of FPGAs in general-purpose computing, further approaches, possibly more custom-tailored to the demands of specific application domains, may come up and gain traction. All three presented pillars can deliver portability and improved scalability to new FPGAs, the latter two without lengthy tool runtimes. This property is particularly important in dynamic OTF computing scenarios. Since OpenCL and libraries already seem to have strong industry backing, we see a particular opportunity for academic research in the overlay pillar.

[8] `http://appgallery.maxeler.com/#/`
[9] `https://cognitive.ptopenlab.com/accelerator`

We consider the combination of low effort and tool runtimes with the potential to cover many different applications through a few reusable designs, that the overlay pillar contributes, as particularly important for the initial adaption phase. While not many optimized library components or OpenCL-based designs exist, broader applicable overlay libraries can deliver at least some acceleration or efficiency with FPGAs and also offer a path to further performance scaling, either through better FPGA hardware, or through more specialized overlays or even hand-optimized designs. In Chapter 4, we quantify the optimization headroom that a general overlay architecture leaves open in our case-study, and show that the overlay can still contribute valuable acceleration.

3.2 FPGA Overlays

As FPGA overlay, we consider any configurable or programmable architecture implemented on top of FPGA fabric. In terms of raw performance or energy efficiency, overlay architectures typically involve overheads compared to direct low-level implementations on FPGA fabric. In this thesis, we propose to accept some overheads, in order to increase the potential to program FPGAs much faster and in a more automated way. As discussed in Section 3.1, portability and performance scalability to other underlying FPGAs are further important concerns in this regard. Many authors of related work also pursue some of these goals.

In this section, we present related work on FPGA overlays mainly in two regards, firstly the architectural approach and secondly the area of programmability, productivity and performance scaling of these architectures. Existing overlays can be subdivided into two categories, firstly processor-like instruction-programmable overlays, which we discuss in Subsection 3.2.1, and secondly structurally programmable or configurable overlays, including coarse-grained reconfigurable arrays (CGRAs) and virtual FPGAs, which we discuss in Subsection 3.2.3. The intermediate Subsection 3.2.2 covers configurable and reconfigurable processor architectures that are not or only prototypically implemented on FPGAs, but impact the overlay architectures in both domains.

3.2.1 Instruction-Programmable Overlays

As introduced in Subsection 2.1.4, FPGAs can be configured into arbitrary computing circuits, including instruction-programmable processors. Processors implemented on FPGA fabric are also denoted as soft or softcore processors, and can be considered as a form of overlay that transforms the structurally programmable FPGA to an instruction-programmable architecture, which can then be targeted by conventional software development methods and tools.

Softcore processors have been designed with various goals. Commercial designs from FPGA manufacturers, like Microblaze[10] and Nios II[11] have been introduced as host processor or controller for customized accelerator logic on FPGA [218] or as alternative to

[10] `http://www.xilinx.com/products/design-tools/microblaze.html`
[11] `https://www.altera.com/products/processors/overview.highResolutionDisplay.html`

embedded processors in order to increase the longevity of processor-specific software [206]. The LEON series with for example the LEON3[12] design has been developed as a processing platform for space missions [77] that can both be implemented as custom hardware or synthesized to FPGAs. Open-source soft processors like OpenRISC[13] are used as research vehicle, for example with regard to fault-tolerance [198, 47] or security [50, 42, 29].

Many soft processor projects consider performance of the instruction-programmable overlay as a relevant target metric, but don't try to surpass the performance of conventional processors, but rather focus on adding other additional benefits. For example when the soft processor is primarily used as control instance for custom logic, the performance focus is on the latter part, with all the involved performance potential and design challenges. Many performance optimizations presented for soft processors are only suitable to reduce the performance gap to hard embedded or general-purpose processors, where the respective techniques are already integrated. This includes pipelining to enable relatively high clock frequencies, and superscalar architectures with out-of-order execution [183, 217] or VLIW instructions [140, 57, 272].

However, there are also attempts to include features that general-purpose processors didn't adopt, because only narrowly focused application classes profit from them. For example with MARC, Lebedev et al. [164] focus on a manycore approach, demonstrated with the high number 48 processing cores on a single FPGA in 2010. With customized cores, they achieved one third of the performance of a fully customized FPGA design. The MARC design also makes use of up to 4-way hardware multithreading, which is common for many GPU designs. Kingyens and Steffan [151] put a particular emphasis on this design aspect in their GPU-inspired FPGA overlay by supporting up to 264 concurrent thread contexts through 64-way hardware multithreading and 4 parallel SIMT execution units. They show that this approach allows to keep an extremely deeply pipelined ALU with 53 clock cycles almost fully utilized.

Much attention has been given to instruction-programmable overlays that gain their performance from vector execution units, inspired both by classical vector processors and by modern SIMD instruction set extensions. The VESPA architecture [282] includes vector units with up to 256 vector elements and up to 32 lanes that support vector chaining. In VIPERS, Yu et al. [284, 285] add, in addition to a distributed register file, a local memory to each vector lane in order to support efficient table lookups, for example needed by Advanced Encryption Standard (AES) encryption. In the VEGAS architecture, Chou et al. [56] let the vector units read and write "directly [from and] to a scratchpad memory instead of a vector register file" and also support variable operator sizes like the SIMD units of modern general-purpose CPUs. With VENICE, Severance and Lemieux [228] introduce operations on 3D vectors and on unaligned vectors. They achieve good area efficiency also for applications with limited DLP and hence were able to implement a multicore design to exploit additional TLP.

All these vector architectures have been demonstrated in stand-alone embedded systems and mostly evaluated in comparison to a soft Nios II processor without vector units. For

[12] `http://www.gaisler.com/index.php/products/processors/leon3`
[13] `http://braap.org/or1200/lo3/spec.html`

VIPERS, Yu et al. [284, 285] also include a comparison to custom designs created with Altera's discontinued HLS tool C2H and found the vector processor to scale to much higher performance points, whereas HLS achieved better performance per area for small designs. A single, highly regular matrix multiplication kernel on a VEGAS design [56] was also compared to a contemporary general-purpose processor and achieved a speedup of around 1.5x, when the general-purpose processor's SIMD units were not used. On the Convey HC-1, which will be introduced in more detail in Chapter 4, a similar instruction-programmable vector overlay, denoted as Vector Personality, is shipped as part of a high-performance general-purpose and HPC computing system. The vector units are split between four FPGAs and support double precision floating point operations and vectors of up to 1024 elements. A comprehensive performance comparison to custom designs and general-purpose processors is the subject of Chapter 4.

The academic vector processor overlays have been programmed with hand-written inline assembly code [282, 284, 285] or C macros [56, 228], which are formalisms that are more accessible to software developers than hardware design. However, they lack portability to other architectures and allow generally much less productivity than high-level language constructs. For the Vector Personality of the Convey HC-1, a C/C++ compiler is shipped that compiles suitable pragma-annotated code segments to the vector coprocessor. In Chapter 5, we present shortcomings of this compiler and our extended toolchain that covers more suitable code segments and is independent of pragmas.

Beyond the presented parallelization aspects, instruction-programmable overlays can profit from the versatility of FPGAs by customization to specific applications or application domains. For the VESPA architecture, Yiannacouras et al. [282] vary different design parameters of the vector processor, including support for heterogeneous vector lanes, and show that choosing the right processor variant per application increases peak performance per area on average by 13%. Lebedev et al. [164] customize not only the multicore configuration, but also the datapaths of their MARC cores for their Bayesian network inference to achieve around one third of the performance of a manually designed fully custom circuit on the same FPGA. With the VectorBlox MXP design, Severance et al. [227] have extended their VENICE approach to support customized vector instructions that can internally pipeline several operations. With custom instructions for an N-body simulation, they achieve more than 100x speedup over the 32-lane base design. The VectorBlox MXP design can be commercially licensed[14].

The customization of datapaths along with the integration of custom instructions into the ISA has also been investigated independently from specific parallelization approaches, both for hard [96, 26] and soft [59] ASIPs. For example specific instructions for AES encryption have been integrated into a tiny soft processor to achieve 3x more performance on this application than a larger non-customized soft processor [95]. As mentioned in Section 2.2.2, custom instructions for cryptography have also been included into general-purpose processor designs due to their high performance potential with moderate hardware investments. However, a soft processor can be customized for each application individually

[14] `http://vectorblox.com/`

with a different instruction set for each application [59], thus exhibiting the reuse of the same configurable resources for different tasks that is one characteristic feature of general-purpose computing.

3.2.2 Reconfigurable Hardware beyond FPGAs

Further research in the domain of instruction set extension (ISE) has been conducted targeting an intermediate architectural approach, where a reconfigurable functional unit (RFU) for custom instructions is integrated with a conventional general-purpose processor. The Chimaera architecture features an RFU with FPGA fabric with an array of logic blocks and routing resources constrained to a downward data flow [114]. Automatically [194] or by hand [114], the data flow graphs (DFGs) from basic blocks without memory operations are mapped to this unit. Evans et al. investigate [79] graph coverage for different coarse-grained RFU designs without memory operations. In the OneChip architecture [48], the RFU additionally has a memory interface that allows it to perform load and store operations independently from the host processor.

Other projects like PipeRench [93], MorphoSys [236], GARP [45], MOLEN [258] or Zippy [210] assume more autonomous reconfigurable units that can execute entire loop nests instead of mere DFGs. They demonstrate a wide range of speedups over simple RISC processors, mostly for signal processing and media compression benchmarks. Even though some prototypes of such architectures were evaluated on FPGAs, the target platforms are coarse-grained reconfigurable arrays (CGRAs) implemented in hardware. These architectures contain configurable elements (denoted as configurable function blocks (CFBs), processing elements (PEs) or functional units (FUs)) and an interconnect fabric that both operate on data words instead of the individual bits of FPGAs. This higher granularity reduces the overheads associated with FPGAs, as well as the configuration space, both in terms of required configuration bits and in terms of design space for placement and routing. For a survey of early coarse-grained reconfigurable architectures, we refer to Hartenstein [112].

A number of commercial architectures with configurable or reconfigurable arrays of coarse-grained processing elements as accelerator units for embedded processors followed [30, 181, 250]. Amano [21] surveys more of these architectures. They had limited success only in signal and media processing domains, where they were mostly presented as solutions with fast time-to-market for special-purpose embedded systems [180, 179, 261, 37]. In such scenarios, CGRAs compete with fully specialized accelerators and the benefit of reconfigurability is limited to updates and bug-fixes. When the CGRAs' structure and FUs need to be customized for specific tasks in order to achieve competitive performance and efficiency, this customization needs to happen at the design time of a specific product [180, 37, 234], with the involved impact on NRE cost, time-to-market and required product volumes. This is one motivating factor, why CGRA have also been implemented as a form of structurally programmable overlay on FPGAs, with the option for specialization after device fabrication time.

3.2.3 Structurally Programmable Overlays

FPGA fabric attached in some form to a processor can be used to implement arbitrarily specialized accelerators. Yet, as discussed, the design of such accelerators is difficult and hard to automate, requires lengthy synthesis processes and also the actual reconfiguration times can turn out as performance issue. Overlay architectures that present a more restricted structure to designers and tools can facilitate the design and speed-up the synthesis and reconfiguration times. As outlined in the previous subsection, a commonly investigated structure is that of CGRAs, which were earlier envisioned as dedicated hardware and sometimes prototyped on FPGAs, but more recently get explicitly designed for realization on FPGAs.

A wide range of architecture variants for CGRAs on FPGAs has been explored [153, 234, 63, 81, 167, 46, 142, 170]. The predominantly considered execution model for CGRAs is to configure them to repeatedly execute a specific DFG, be it in a loop or with subsequent invocations. Considering the mapping of a DFG's nodes and edges that specify operations and data flows to a CGRA helps us to discuss many architectural aspects of CGRAs on FPGAs. PEs can either just cover a single node of the DFG [98, 46, 142], different ones according to a global schedule [210, 234, 63, 81, 167] or to a local control program [153]. The operations of PEs can be homogeneous [167, 142], heterogeneous [98, 46] or specialized when the overlay is generated [153, 234, 62]. The interconnects between PEs can be local or global [81]. They can have configurable buffering capabilities to compensate for non-balanced datapaths [46].

CGRAs on FPGAs are open to specialization for specific applications or application groups. Lin and So [167] show that customization of interconnect topologies improves the engergy-delay product by 9% - 28% for a set of synthetic and real world DFGs with several thousand nodes, mapped on an array with 16 processing elements. Building mainly upon DSP blocks as FUs, the intermediate fabrics from Coole and Stitt [63] use only on average 18% more resources than comparable circuits directly mapped to the underlying FPGA. After additional customization of the interconnect, this overhead is reduced to around 10%. For applications with several kernel loops, an overlay even uses on average 60% less area than a direct parallel FPGA implementation of circuits for the respective three to seven kernels [62]. This is achieved by using common subnets among different kernels.

Ma et al. [170] fix in their design only the routing overlay and advocate the use of application-specific PEs through partial reconfiguration. The PEs are synthesized according to computation patterns expressed in a domain-specific language (DSL) and are mapped to available slots dynamically at runtime.

For several variants of CGRAs on FPGAs, tool support has been presented for automated mapping of DFGs to the overlay, including placement, routing and, where applicable, scheduling. The tool runtimes for these steps are several orders of magnitude lower than when directly targeting FPGA fabric, in the order of milliseconds to few seconds for the investigated designs [63, 81, 46]. With little or no manual intervention, these tools also enable scaling to overlay designs of different sizes. When the tools are extended to support compatible high-level source formats, this lays out a path to convenient code portability

between different overlay approaches and different underlying FPGAs. In [61], Coole and Stitt present a synthesis flow starting from OpenCL specifications, which can both generate suitable application-specific overlays and generate configurations for existing overlays. The tool runtime for configuring overlays are short enough to match the just-in-time compilation paradigm for OpenCL kernels.

Other projects have gone even further, toward transparent just-in-time mapping of runtime detected DFGs or loops in binary format. For example, Beck et al. [31] and Bispo et al. [34] map instruction sequences to CGRA architectures, going beyond individual DFGs by speculation [31] or identification of so-called megablocks that cover typical control flow inside a loop nest [34]. Grad and Plessl [99, 100] generate for individual DFGs custom datapaths that are configured into regions for partial reconfiguration. For custom instructions through partial reconfiguration at runtime, Koch et al. [155] investigate methods to minimize the hardware overhead of reconfiguration modules.

Partial reconfiguration regions for customized datapaths of custom instructions are just one example of structurally programmable overlay architectures that do not belong to the category of CGRAs. We present a short selection of such approaches and their design goals. With SCORE [49], Caspi et al. investigate partial reconfiguration regions with a particular focus on scaling with different hardware sizes. Research on overlay architectures for finite-state machines [243, 60] aims at complementary control regions for the dataflow-centric CGRA architectures, but may also be useful for independent computing tasks like complex regular expression matching for security applications [73]. Virtual FPGAs have, beyond fundamental research questions about architectures and tools, been investigated as a portable intermediate layer for custom instructions [154] or more generic FPGA circuits [169, 101, 39]. The area and frequency overheads of fine-grained virtual FPGA seem prohibitive for general-purpose applicability [39], but customizations like coarse-grained virtual resources and time-multiplexing of resources may open up a path to lower overheads, area scalability and fast synthesis times [101]. Hung and Wilton [133] virtualize only the interconnect network of FPGA to dynamically insert trace buffers for debugging into synthesized FPGA designs.

To summarize this section, we conclude that structurally programmable overlays can have manifold incarnations that exhibit different patterns of computing with FPGA. Design-time specialization for different applications can reduce the performance and efficiency gap to fully custom designs. Synthesizing overlay architectures is generally much faster than to underlying FPGA fabric and some projects have demonstrated favorable scaling properties. Building upon these properties, a library of overlays with suitable binary or OpenCL synthesis flows [31, 35, 61] is a promising way to overcome FPGA productivity issues for general-purpose computing. Instruction-programmable overlays contribute to this mix and are equally dependent on compilation tools that start from common software or intermediate representations.

However, in contrast to the structurally-programmable overlays, where particularly Coole and Stitt closely investigated the overlay overheads compared to fully customized designs, these trade-offs are insufficiently understood for instruction-programmable overlays. Lebedev et al. [164] present a single comparison of their manycore overlay MARC with a custom

design and observe a 3x performance overhead. Yu et al. [285] compare their vector overlay with results of an early HLS tool and observe up to 8x more performance with their overlay, which mainly demonstrates that HLS on unaltered source code was not competitive in 2008.

3.3 Chapter Conclusion

In this chapter, we have analyzed how trade-offs between productivity and performance potential have mostly prevented FPGAs from entering the general-purpose computing domain. The integration into a common memory space and hierarchy can increase productivity, but has not been a prerequisite for GPUs adoption, where software solutions have masked this challenge. Instead, accessibility of the programming models and tools along with performance potential, portability and scalability are a major hurdle for FPGAs. Considering recent progress in FPGA tools and system architectures, we propose three pillars that jointly can provide beneficial characteristics for different developer requirements.

While with OpenCL and application libraries, two of these pillars already have strong industry backing, we focus on overlays and reviewed the mostly academic research in this area in the previous section. The combination of instruction-programmable and structurally-programmable overlays provides a diverse space of architectural patterns, tooling approaches and specialization opportunities. Based on this analysis, we identify four research questions around this overlay concept.

1. What amount of overheads do overlays on FPGAs involve when compared to the implementation of fully customized designs on the same FPGAs? As discussed in the previous section, this question is not sufficiently investigated, in particular not for instruction-programmable overlays. Furthermore, can these overheads be attributed to specific design differences between the computation patterns within the overlay and without it? On a broader scope, where does the usage of overlay architectures put the performance and efficiency of FPGAs compared to other architectures? Figure 3.2 takes up the qualitative illustration from Figure 2.2 in Subsection 2.2.4 and illustrates the context of this research question.

2. Can overlays sufficiently increase the productivity when targeting FPGAs with applications that originate from the domain of general-purpose computing and are designed with standard, software-based principles? For many overlays, fast back-end design tools have been presented that allow fast code generation respectively synthesis compared to the tools targeting FPGAs directly. However, these tools don't start from software implementations on abstraction levels typical for general-purpose computing, but rather require much more target-specific application descriptions.

3. How many of the manifold application patterns that are generally suitable for FPGA acceleration can be covered with overlay architectures and corresponding tools? The presented overlay architectures are very diverse, but several of the kernels and applications that demonstrate their usage either overlap or originate from similar domains.

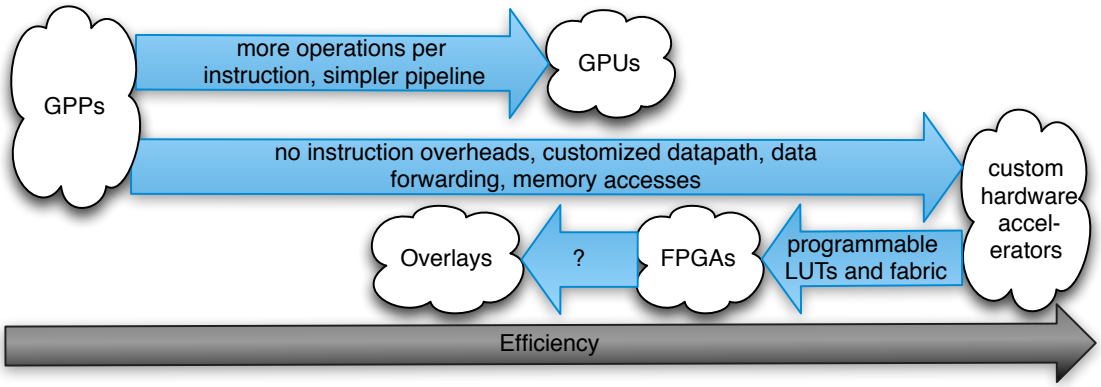

Figure 3.2: The usage of overlay architectures on FPGAs involves overheads that need to be quantified and analyzed. It introduces a new area of design points that need to be assessed in comparison to the existing alternatives, here illustrated with regard to efficiency.

4. How can the customization of overlay architectures, which has been shown to have great potential for increasing performance and efficiency, be systematically applied to the diverse set of overlay architectures and be completely or partially automated?

In the next two chapters, we present several contributions to these research questions. In Chapter 4, we primarily address aspects of the first research question by quantifying performance differences between fully customized FPGA designs and an instruction-programmable overlay in an extensive case-study. We also discuss design differences that contribute to these differences. The comparison to other hardware architectures is not in the focus of our practical work. In Chapter 5, we demonstrate for the instruction-programmable vector overlay already targeted in Chapter 4, that highly productive compilation flows for such a target architecture are actually possible, even though existing tools for overlay architectures typically lack a high-level design entry.

With regard to the third research question, we consider our contributions as one component to a required large-scale research effort to better understand the interplay of application domains and overlay architectures. In order to broaden the scope of this research, future studies can on the one hand systematically cover more application patterns, for example by using OpenDwarfs [158] implementations as starting point to target overlays, or on the other hand evaluate few application patterns with a larger set of overlay architectures and corresponding tool flows, as we propose in our thesis outlook (Section 7.2). The fourth research question is not practically addressed within this thesis except for the analysis of overheads that may point towards profitable customization directions. However, the related work discussed in the previous section outlines the considerable potential of overlay customization. Between performance and productivity, the actual relevance of overlay architectures as path towards FPGA adoption in general-purpose computing depends on the answers to all four presented challenges. Beyond the scope of these overlay-related research questions, we finally pick up the issue of system integration of FPGAs in Chapter 6.

CHAPTER 4

Stereo-Matching Kernels on Overlay and Custom Designs

In this chapter, we present two approaches for accelerating a stereo-matching application with general-purpose characteristics on FPGA platforms. Within this application, we identify and offload ten kernels, all with abundant DLP, but with different dependency patterns and computational intensity. As first solution, we implement individually specialized dataflow kernels in a spatial programming language for a Maxeler FPGA platform. As second approach, we target an instruction-programmable overlay on the application FPGAs of a Convey HC-1, which implements a vector coprocessor with large vector lengths.

The high-level contribution of this chapter is the demonstration that with average overheads of 3x in raw performance over fully customized FPGA designs after compensating for the different platform characteristics, the overlay architecture can still contribute to speedups over general-purpose CPUs in a general-purpose computing scenario. When practical reconfiguration overheads limit the parallelism of customized kernels, the overheads are reduced to 2.5x. The broad quantification of this overhead over custom designs is a novel contribution for instruction-programmable overlays. A distinction between different kernel patterns also allows us to point to possible reasons for these overheads.

This chapter contains also several technical contributions towards this goal. Besides the actual working kernel designs with state-of-the art optimization for two different FPGA platforms, a memory management wrapper library is introduced that greatly facilitated the practical implementation and evaluation with the different memory models of the platforms. Also, scalable designs for the customized dataflow kernels along with accompanying performance models allowed the guided synthesis of standalone kernels with maximal parallelism and a fused design with wider functionality. The presented stereo-matching implementation is to date the most accurate one with FPGA acceleration, though admittedly also slower than the ones running purely on FPGAs.

Most of this chapter was published in [6], supplementing two previous publications that cover the acceleration of this stereo-matching application on the two individual target platforms [4, 5]. Beside Tobias Kenter as lead researcher and main contributor to the

presented implementations and measurements, Henning Schmitz contributed during his Bachelor's project and as student assistant to the first implementation of the software application used in [4] and to parallelization ideas and implementations on both FPGA targets.

In Section 4.1, we first outline the general stereo-matching task, before presenting in Section 4.2 the concrete algorithm we accelerate. We then introduce in Section 4.3 the two accelerator platforms and how they are programmed in this work. Before the concrete kernel implementations are described side-by-side for both platforms in Section 4.5, we outline the common acceleration principles and memory management concepts in Section 4.4. Section 4.6 documents the setup of our experiments. Comparing both systems in Section 4.7 requires some normalization to account for the different hardware platforms, but gives us insights about the trade-offs in runtime, design effort and tool runtimes. Section 4.8 discusses related stereo-matching implementations on FPGA platforms, before concluding in Section 4.9.

4.1 Introduction to the Stereo-Matching Problem

Stereo-Matching is the computation of a disparity map from a pair of stereo images. The disparity specifies at every position in the image, how far the displayed object or feature appears displaced between the two images due to the different positions of the two camera lenses. In earlier work, the term *Stereo Correspondence* has been used for the same problem[259, 222]. By inverting the disparity information and scaling it according to the geometry of the camera system, actual depth information about the scene is obtained, which is denoted as *Stereo Vision* and is probably the most important method for computer vision.

Applications for computer vision in general and stereo-matching in particular range from classical embedded use cases in automotive and industrial contexts or robot navigation [254] to personal or professional usage on general-purpose hardware for 3D media generation or 3D data acquisition. Common design goals for all types of applications are high matching quality and high processing speed, yet with varying priorities and additional constraints, e.g. on image resolution, on latency or throughput, or on power and resource limitations. Up to now, the most accurate methods like the one employed here, do not match the typical real-time and power constraints of embedded systems. Of the markets outlined in Subsection 2.3.3, it fits the personal mobile devices, which may also be used for image acquisition, but also fits to PCs or cloud datacenters, to which the stereo-matching task may be offloaded from a camera or mobile device to avoid draining battery and to achieve higher total performance.

Most stereo-matching algorithms perform so called dense stereo-matching, that is, they compute a disparity map, containing a disparity value for each pixel. This value represents how far the object that this pixel belongs to appears shifted between the left and right stereo image. More formally, a disparity value $d = d_{left}(x, y)$ for a pixel in the left stereo image at position (x, y) signifies that the physical feature displayed by this pixel is believed to be found in the right stereo image at position $(x - d, y)$. If a corresponding right

disparity image is computed, to be consistent, the corresponding disparity in the right image, $d_{right}(x - d, y)$ should also contain the same disparity value d, pointing back to position $([x - d] + d, y) = (x, y)$. For this definition of disparity and consistency to be precise, the two images need to be perfectly horizontally aligned.

As auxiliary metric to compute disparities, many algorithms use a cost value, also denoted as matching cost, for each pixel at each possible disparity $C(x, y, d)$, thus forming a three dimensional cost volume, where a low cost signifies that it is plausible that this pixel should have the corresponding disparity. Throughout this chapter, we use the term cost purely for this matching cost and not for system cost as introduced in Subsection 2.1.1.

The general sequence of modern stereo-matching approaches comprises three steps [268]: first, computation of a matching cost volume, second, an optimization method which computes a disparity map from the cost volume and third, post-processing of the disparity map. For computing the initial cost volume, metrics for local color similarity and for local structural similarity are commonly employed [182, 123]. To smoothen the cost volume, aggregation techniques can be employed [85, 259]. Optimization in the simplest form, often called winner takes all (WTA) [123], just selects the disparity with the lowest cost for each pixel: $d(x, y) = \arg\min_d C(x, y, d)$. Other approaches like belief-propagation (BP) and graph-cuts (GC) seek to combine low matching costs with properties like low energy of the resulting disparity map. Beyond generic image augmentation approaches, post-processing often involves consistency checks between the disparity maps generated for left and right image and handling of the identified inconsistencies [124, 182].

For many stereo methods, there exist variants which incorporate more than two input images, typically but not necessarily captured from a set of cameras placed along one horizontal line. The additional viewpoints open up additional opportunities for consistency checks among derived disparity maps and can particularly help to fill occluded areas, when they are visible in one of the additional input images. However, these variants still rely on a good underlying stereo-matching algorithm, like the one utilized as case study here.

4.2 Stereo-Matching Algorithm with Inherent Parallelism

In our work, we algorithmically follow the stereo-matching implementation published by Mei et al. [182]. It follows the three basic steps outlined in Section 4.1, but splitting the first step of cost computation into two separate phases, we subdivide it here into a total of four phases. Figure 4.1 gives a high-level overview of the stereo-matching sequence. In the first phase, cost initialization, two similarity metrics are applied on the input images to compute for each pixel and each possible disparity a local cost value, thus forming the first cost volumes. In the second phase, cost aggregation, the costs of neighboring pixels of the same disparity are aggregated in adaptive support regions, which are determined by color differences and absolute distances. This smooths the original cost volumes. In the third phase, scanline optimization, an energy minimization approach is mimicked by dynamic programming along 1-dimensional scanlines. This produces a first pair of disparity maps, but also another pair of cost volumes that are used in the fourth phase, disparity

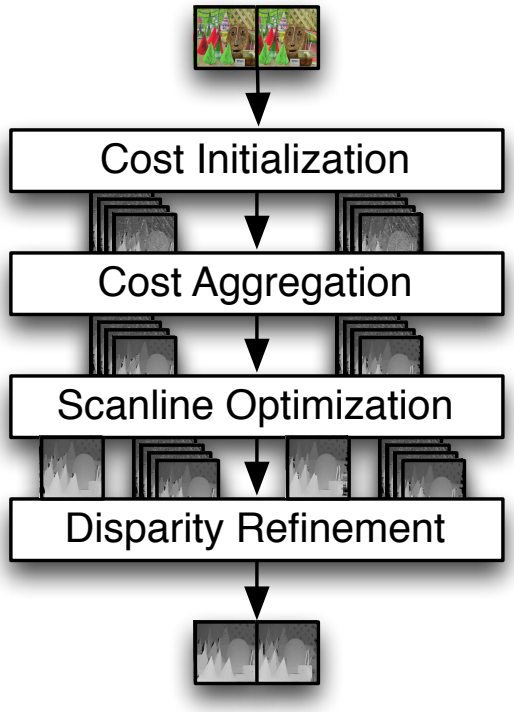

Figure 4.1: High-level overview of the stereo-matching algorithm following [182]. From a pair of stereo images, intermediate cost volumes are computed, which are used to generate disparity maps as final result.

refinement. This fourth phase performs a consistency check between the left and right disparity maps and applies several local optimizations for pixels which are not classified consistently.

As the most time consuming parts and parts where the accelerated kernel functions are located, we present some details about the mechanisms of cost aggregation and scanline optimization, and briefly outline the two less time consuming steps cost initialization and disparity refinement, which are executed on CPU in our work.

4.2.1 Cost Initialization

The cost initialization following Mei et al. [182] provides the first cost metric $C_i(x, y, d)$ for each position and disparity based on two individual components. The first component is called the absolute difference cost C_{AD} for a pair of left- and right-image pixels in RGB format. This cost is defined as the difference of pixel intensities I, averaged over the three color channels: $C_{AD}(x, y, d) = \frac{1}{3} \sum_{i=R,G,B} |I_{left,i}(x, y) - I_{right,i}(x - d, y)|$

The second component is the census cost C_{census}, computed as the Hamming distance of the census transforms of a left and corresponding right pixel. This census transform

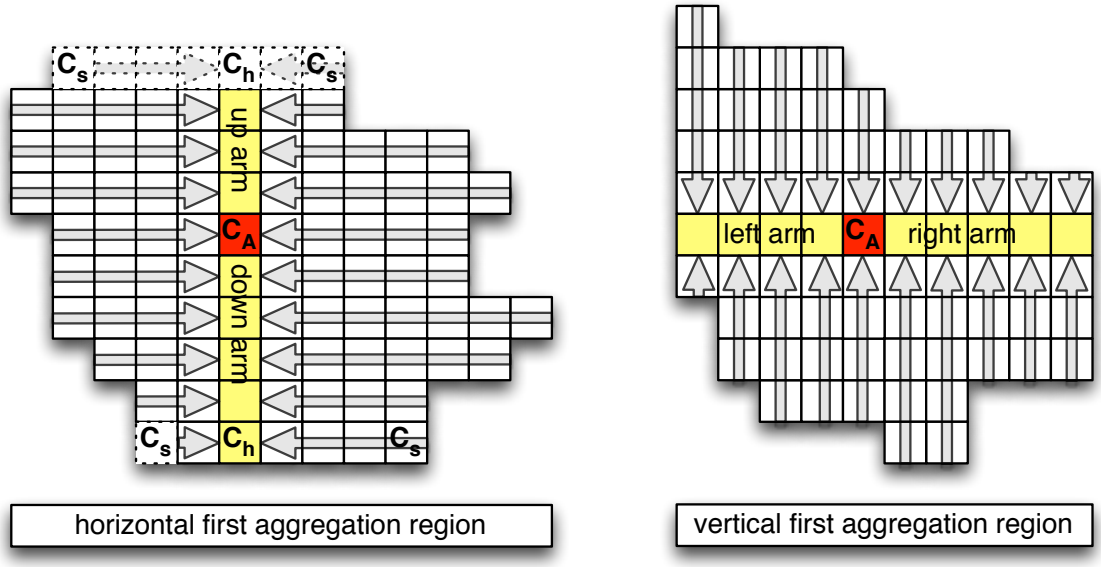

Figure 4.2: Illustration of cross-based cost aggregation regions. Left side: projection of horizontal arms on the vertical arms of a pixel. Right side: projection of vertical arms on the horizontal arms of a pixel.

captures the local structure in a 9×7 window around each pixel. Structural information is less sensitive to variations in lighting between the left and right image.

These two cost components are individually scaled by an exponential function that also enables the weighting of outliers and then added up to form the initial cost.

4.2.2 Cost Aggregation

The idea of cost aggregation is to reduce the huge amount of noise contained in the local cost metrics. Instead of simple smoothing, the costs for each possible disparity are aggregated over a limited area around each pixel, which likely belongs to the same objects of the image and therefore should have similar disparity values. Therefore aggregation areas should track object boundaries in shape and size as good as possible. However, computing individual aggregation areas for each pixel and summing up the costs inside them can be very compute intense. The cross-based aggregation method utilized here was first proposed by Zhang et al [286]. The areas are defined by the length of four arms for each pixel, two extending to the left and right, two up and down. The arm length are computed prior to the actual aggregation step depending on color differences and parametrized thresholds. Two possible aggregation areas are now formed by all vertical arms that belong to pixels on the horizontal arms of each pixel and respectively the other way round as illustrated in Figure 4.2. Horizontal first aggregation areas can cover vertical object boundaries better, vertical first aggregation is more precise for horizontal object boundaries.

Algorithm 4.1 Horizontal aggregation step. Note: **for-all**-loops are independent, **for**-loops are ordered.

Input: $\forall(x, y, d)$: $C_i(x, y, d)$ = input cost
Input: $\forall(x, y, d)$: $A_{left/right}(x, y, d)$ = precomputed horizontal arm lengths
Output: $\forall(x, y, d)$: $C_h(x, y, d)$ = aggregated costs of row segment around position (x, y) in disparity d

```
1:  for all d ∈ disparities do
2:    for all y ∈ rows do {compute integral sums}
3:      C_s(0, y, d) ← C_i(0, y, d)
4:      for x = 1 to #columns do
5:        C_s(x, y, d) ← C_s(x − 1, y, d) + C_i(x, y, d)
6:      end for
7:    end for
8:    for all y ∈ rows do {compute row segment costs}
9:      for all x ∈ columns do
10:       a_l ← A_left(x, y, d)
11:       a_r ← A_right(x, y, d)
12:       C_h(x, y, d) ← C_s(x + a_r, y, d) - C_s(x − a_l − 1, y, d)
13:     end for
14:   end for
15: end for
```

For both aggregation areas, the actual aggregation can be performed in linear time with the help of integral sums. Pseudocode for the horizontal aggregation step is given in Algorithm 4.1. As the outer loop indicates, the step is performed independently on each disparity d. The first loop nest computes for each row the integral sum of costs from the row's first element to the current element. In the second nested loop, for each position (x, y), the difference between two elements of the integral sum is taken, with element positions defined by the arm lengths at (x, y). This difference is exactly the sum of costs in the horizontal segment around (x, y) that is specified by the two arms. In Figure 4.2, aggregation for one disparity is illustrated for the topmost and bottommost rows of the horizontal first aggregation region, where the horizontally aggregated costs C_h depend on two elements of integral sums C_s. Afterwards, in the vertical aggregation step, vertically integral sums (not illustrated in the Figure) are computed and the aggregated costs C_A is computed again as difference between the running costs at two positions, here at the two positions that are marked with C_h in the example. Note that the left and upper positions, from which the respective integral sums are taken, are not part of the aggregation area itself.

A pair of horizontal and vertical aggregation steps forms one aggregation iteration with the illustrated horizontal first aggregation region. Following Mei et al. [182], we execute a total of four such aggregation iterations, the first and third one using horizontal first regions and the second and fourth one using vertical first regions. Not mentioned by Mei

et al. [182] is a normalization step after each aggregation iteration, where the aggregated cost is scaled by the respective aggregation area. This was already proposed by Zhang et al [286], also utilized by Shan et al. [230] and we found it to be important for the result quality of our implementation.

4.2.3 Scanline Optimization

The scanline optimization follows Hirschmüller's [124] semiglobal matching strategy. Global matching would perform a 2-dimensional energy minimization for the entire image, minimizing the weighted sum of the energy in the final disparity image and of the involved matching costs for this disparity image. The scanline optimization mimics this idea along 1-dimensional lines, but avoids costly minimization steps and instead uses a dynamic programming approach, where the previous disparity decisions along the scanline are fixed and only the energy trade-off for the current step is considered. Equation 4.1 outlines the basic recursion equation and Algorithm 4.2 illustrates pseudocode for one scanline direction.

$$
\begin{aligned}
C_r(x,y,d) = C_A(x,y,d) + \min\,[\, & C_r(x-r_x, y-r_y, d), \\
& C_r(x-r_x, y-r_y, d \pm 1) + P_1(x,y), \\
& \min_k C_r(x-r_x, y-r_y, k) + P_2(x,y)\,] \\
& - \min_k C_r(x-r_x, y-r_y, k) \\
r \in & \{right, left, down, up\} \\
(r_x, r_y) \in & \{(1,0), (-1,0), (0,1), (0,-1)\}
\end{aligned}
\tag{4.1}
$$

The scanline cost C_r in the equation is computed along a scanline path that depends on the direction $r = (r_x, r_y)$, which in the pseudocode example is $right = (1, 0)$ to define a scanline to the right, with accordingly denoted scanline cost C_{right}. The scanline cost depends on the aggregation cost C_A and a term requiring all scanline costs at the previous pixel position along the scanline path. This previous pixel position is given by $(x-r_x, y-r_y)$ in the equation and by $(x-1, y)$ in the pseudocode example. This term depending on the previous position reflects the energy minimization concept, selecting either the scanline cost from the previous position at the same disparity, or the scanline cost from the previous position at a neighboring disparity plus a small penalty $P_1(x, y)$, or the minimal scanline cost of all disparities at the previous position plus a larger penalty $P_2(x, y)$. These paths trade-off energy components added by the matching costs with energy components from the disparity profiles represented by the penalties $P_1(x, y)$ and $P_2(x, y)$. Both penalty values are chosen for each specific position based on the color differences of the original images. Finally, for normalization, the minimal scanline cost at the previous position is subtracted.

Figure 4.11, shown later in Section 4.5.2 on page 89, serves us mainly to illustrate the compute and parallelization pattern of our implementations, but also contains a numeric example of a scanline computation, here of a downward scanline. For simplicity, costs are represented as integer values and with an aggregation cost of 0 for the second line. Green

Algorithm 4.2 Scanline optimization step in left to right orientation (denoted as `ScanRight`). Note: **for-all**-loops are independent, **for**-loops are ordered.

Input: $\forall(x, y, d)$: $C_A(x, y, d)$ = aggregated cost
Output: $\forall(x, y, d)$: (x, y, d) = right scanline cost

1: **for all** $y \in$ rows **do**
2: **for all** $d \in$ disparities **do**
3: $C_{right}(0, y, d) \leftarrow C_A(0, y, d)$
4: **end for**
5: $C_{min}(0, y) \leftarrow \min_k C_{right}(0, y, k)$
6: **for** $x = 1$ to #columns **do**
7: **for all** $d \in$ disparities **do**
8: $c_0 \leftarrow C_{right}(x - 1, y, d)$
9: $c_{-1} \leftarrow C_{right}(x - 1, y, d - 1) + P_1(x, y)$
10: $c_1 \leftarrow C_{right}(x - 1, y, d + 1) + P_1(x, y)$
11: $c_k \leftarrow C_{min}(x - 1, y) + P_2(x, y)$
12: $C_{path} \leftarrow \min(c_0, c_{-1}, c_1, c_k)$
13: $C_{right}(x, y, d) \leftarrow C_A(x, y, d) + C_{path} - C_{min}(x - 1, y)$
14: **end for**
15: $C_{min}(x, y) \leftarrow \min_k C_{right}(x, y, k)$
16: **end for**
17: **end for**

arrows indicate the minimization paths taken to compute the scanline costs in the second row depending on the previous row and the input aggregation costs. These green arrows reflect the best trade-off between minimization of the input costs and the scanline energy for any given position.

In the abstract description of stereo-matching approaches in Section 4.1 the optimization step was described to yield a disparity map. In the more elaborate approach we use, the scanline equation for each direction produces a new cost volume, now incorporating a trade-off between raw matching costs and energy of the disparity map. This is convenient, as now the results of scanline optimization steps along different directions can simply be combined into an average cost volume, instead of having to combine different discrete disparity maps. On the combined scanline costs, now a WTA optimization selects the actual disparity for each pixel.

We use four directions, right, left, down and up, like proposed by Mei et al. [182]. Each scanline by itself produces some streaking artifacts in the direction of the scanline, because the penalty values only favors persistence of previously optimal disparities along the scanline, but not in the reverse direction. Therefore it is important to utilize not only several different scanlines like in [265], but also to have pairs of reverse scanlines to symmetrically offset the streaking.

4.2.4 Disparity Refinement

The previous three phases are executed for both the left and right image, producing one disparity image for each side. As indicated earlier, their computed disparity values should match: $d_{left}(x, y) = d_{right}(x - d_{left}(x, y), y)$. Pixels for which this is not the case are classified as outliers and are treated with the refinement steps *Iterative Region Voting* and *Proper Interpolation* from Mei et al. [182]. Due to insufficient details given, we skip their *Depth Discontinuity Adjustment* step, but again perform the subsequent *Sub-pixel Enhancement* step, which aims to reduce errors caused by the discrete disparity levels.

4.2.5 Software Implementation

As starting point for our acceleration, we use our own software implementation for stereo-matching, which follows these concepts, but offers additional features, such as different, parametrizable cost initialization metrics (for more metrics see e.g. [125]), an adjustable sequence of aggregation steps, and an optional OpenGL visualization of aggregation areas, cost volumes and cost metric profiles. The precision of intermediate cost values required for stable results depends highly on the actual images processed. In general, quality degradation with reduced precision is graceful, but in some cases with single precision floating point, costs after computing differences in the aggregation step can falsely get values of 0, leading to artifacts. Therefore, we use in our software implementation double precision and also require this from the FPGA acceleration. The extensible, feature-rich software implementation with an algorithmic structure driven by runtime complexity and result quality, but not by low-level performance optimizations, are characteristic for general-purpose applications as outlined in Subsection 3.1.1. With the settings of Mei et al. [182], our implementation reaches an accuracy in the Middlebury benchmark [222] of average 5.73% bad pixels and we make sure during our acceleration process to still produce the same results.

4.3 Utilized FPGA Platforms and Programming Models

In this section, we introduce the two hardware platforms we target and outline how they are programmed in this work. We conclude the section with a brief comparison of the accelerator resources as used in our experiments.

4.3.1 Maxeler Platform and Programming Paradigm

The Maxeler MPC-X platform we use [175] is illustrated in Figure 4.3. It comprises two 6-core (12 threads) Intel Xeon X5650 (Westmere microarchitecture) CPUs, running at 2.66 GHz, as host platform and is equipped with four *MAX3424A Vectis* PCIe accelerator cards, of which in this work only one is used. Each card contains a large Xilinx Virtex-6 SX475T [275] FPGA for user logic, a smaller, non-user-programmable FPGA for the PCIe interface, and 24GB of local SDRAM memory. This local memory is called LMem and has to be read or written in bursts of 384 adjacent bytes. However, in order to come

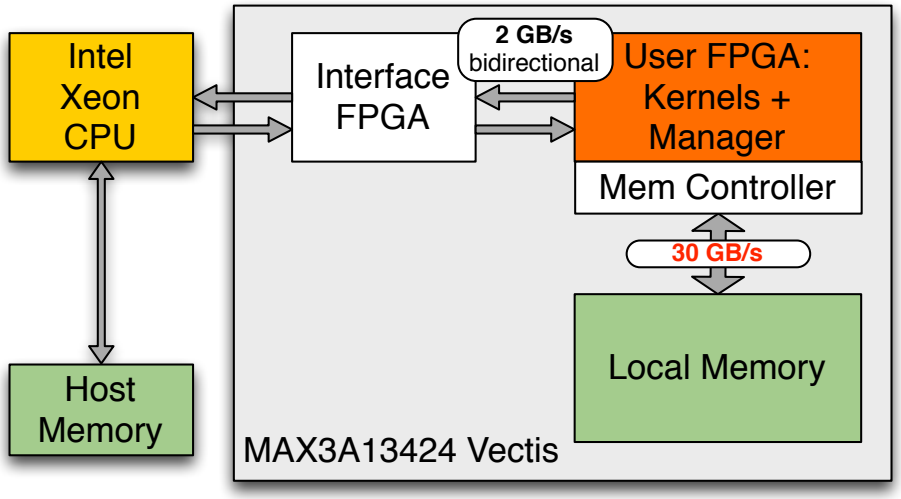

Figure 4.3: Illustration of the Maxeler MPC-X platform with only one of four MAX3 Vectis accelerator card shown.

close to the possible bandwidth of around 30 GB/s (with memory controllers synthesized at 300 MHz; up to 400 MHz are supported by the DDR3 DIMMs), several bursts, either adjacent or with a fixed stride, should be accessed with a single memory command. For example, commands with only 1 burst each lead to an efficiency of only 11%, whereas with 8 consecutive bursts, an efficiency of 80% is reached. The PCIe interface on the other hand can be used to stream data from or to host memory and reaches a bandwidth of 2 GB/s. Note that the memory controller is synthesized by the Maxeler tools onto the user FPGA alongside the custom logic.

The distinctive feature of the Maxeler systems is their development environment [176], which allows programming the FPGAs with a spatial programming language, denoted as *MaxJ* and realized as a Java extension. The kernel functionality implemented on FPGA is integrated with the host (CPU) part of an application through calls to an API automatically generated for the specified functionality. The *MaxJ* language offers a much higher abstraction than HDL languages like VHDL and Verilog, but much finer control on the design than when generating hardware via HLS. Conceptually, *MaxJ* is built around streams of data, where typically one data element per cycle is processed in a so-called hardware kernel. A sequence of operations on one or several streams is automatically translated into a corresponding compute pipeline, where pipelining may also happen inside individual operations, in particular when they utilize DSP blocks. The streams can be connected to other kernels or to LMem or via PCIe to host memory and the Maxeler toolflow automatically generates the required buffers and interfaces.

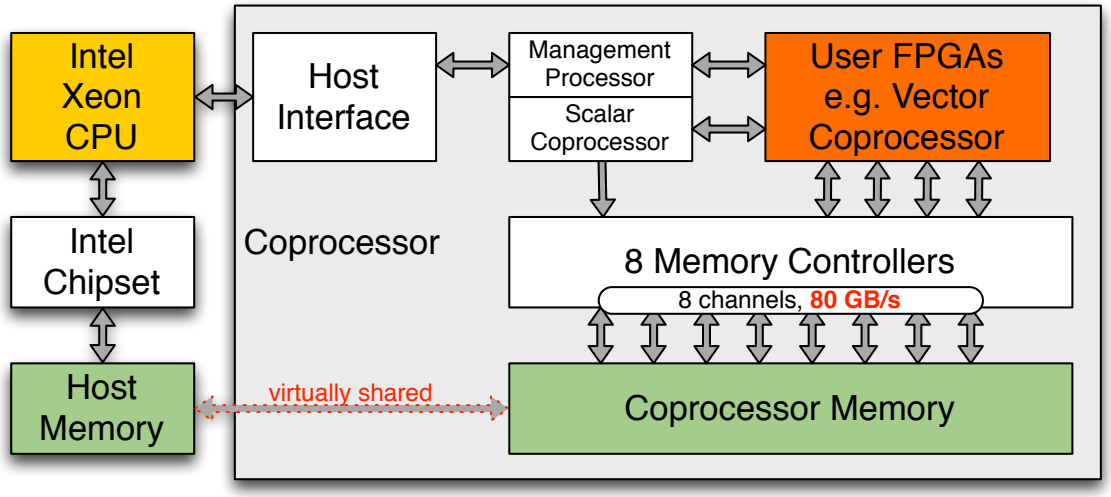

Figure 4.4: Illustration of the Convey HC-1 platform.

4.3.2 Convey HC-1 Platform with Vector Processor Overlay

The Convey HC-1 [40], illustrated in Figure 4.4, is a dual socket server system, where one socket is populated with a dual core Intel Xeon 5138 (Core microarchitecture) CPU, running at 2.13 GHz, while the other socket is connected to a stacked coprocessor board. The two boards communicate using the Intel Front-Side Bus (FSB) protocol. Both processing units have their own dedicated physical memory, which can be transparently accessed by the other unit through a common cache-coherent virtual address space, which distinguishes this platform from the Maxeler system. The coprocessor consists of multiple, individually programmable FPGAs. One FPGA implements the infrastructure that is shared by all coprocessor configurations. These functions include the physical FSB interface and cache coherency protocol as well as configuration and execution management for user programmable FPGAs. For implementing the application-specific functionality, four high-density Xilinx Virtex-5 LX330 [274] FPGAs are available. Eight memory controllers are implemented on one distinct Virtex-5 LX150 [274] FPGA per memory controller. Each of them accesses two DIMMs, which leads to an aggregated bandwidth of close to 80 GB/s with 16 memory modules. In our system configuration, custom-made scatter-gather DIMMs are installed, which allow accessing memory efficiently in 8-byte data blocks, while standard modules are designed for 64-byte block access. The memory controllers implemented on dedicated FPGAs and the custom-made DIMMs of this platform are, with reference to Subsection 2.2.4, indicators of a platform with high performance along with low volume and high unit-cost. For a high-volume product with similar performance characteristics, lower unit-cost and higher efficiency could be achieved with fixed-function memory controllers as present in every commodity GPP and GPU.

The user FPGAs can be configured with different fixed-function designs or programmable overlays. Fixed-function, problem-specific designs can be fully customized by developers

and integrated into the rest of the system by interface libraries written in Verilog. Convey also offers a number of synthesized, ready-to-use designs, so-called *Personalities*, that offer a fixed functionality with few configuration options, for example for graph traversal or local string alignment. On the other hand, the so-called *Vector Personality*, which we use in this work, is an instruction-programmable overlay. It represents a highly optimized design that contains the type of abstraction benefits and overheads that we want to quantify in this work.

The Vector Personality provides the functionality of a vector coprocessor that executes programs targeting its vector instruction set. It comes in two variants, optimized for single- or double-precision floating point operations; both also support integer operations, e.g. for vectorized address calculations. According to our application, we use the double-precision Vector Personality. The vector instructions are implemented for up to 1024 elements. A total of 64 vector registers are available and each can store such a set of 1024 elements. Besides the usual element-wise arithmetic vector operations, the vector instruction set contains memory instructions that distinguish it from typical SIMD vector instruction set extensions for general-purpose CPUs. It can load and store vectors where the elements are individually indexed and do not need to be aligned in a continuous memory location.

Convey includes a compiler to target this vector personality by annotating source code with pragmas, however we found it to be limited to simple array data structures and simple loop nesting patterns, which often requires significant code adaptations besides adding the vectorization pragmas. We fixed many of these shortcomings with the toolflow proposed in [7] and discussed in Chapter 5, however for the comparison of architectural overheads of the overlay, we wanted to achieve the best possible performance. Therefore for the work in this chapter, we designed all kernels by hand in assembly code, particularly exploiting on top of the capabilities of the automated toolflow additional opportunities as vector partitioning, vector register rotation and enhanced reuse of partially computed addresses.

4.3.3 Comparison of FPGA Platforms

Comparing the two hardware platforms, the Convey HC-1 is a few years older, with the utilized FPGAs being one generation behind, and the CPUs being two process shrinks (Intel *Tick*) and one microarchitectural change (Intel *Tock*) behind. On the other hand, when we compare a single Maxeler MAX3424A Vectis accelerator card to the coprocessor of the Convey HC-1, the latter incorporates a lot more hardware resources. Table 4.1 gives an overview of the accelerator hardware as used in our experiments. Together, the four FPGAs for the HC-1's application logic contain almost 3x more LUTs and some more BRAM resources than the single application FPGA of the MAX3424A. Similarly, the peak memory bandwidth of the Convey HC-1's coprocessor is around 2.5x higher than that of the Maxeler MAX3424A accelerator. This is essentially achieved by using more memory modules. Additionally, the Convey HC-1's memory controllers are implemented on dedicated FPGAs, in contrast to the Maxeler MAX3424A platform, where the memory controller is synthesized along with the application logic onto the same FPGA. For the Convey platform, this saves space on the application FPGAs and avoids timing issues

Table 4.1: Hardware resources of the two FPGA platforms as used in our experiments. *Maxeler MAX3424A memory clock and bandwidth depend on user design.

Platform	Maxeler MAX3424A	Convey HC-1
Application FPGAs	1× Virtex-6 SX475T	4× Virtex-5 LX330
#6-input LUTs	**297600**	4× 207360 = **829440**
#36Kb BRAMs	**1064**	4× 288 = **1152**
#DIMMs	6	16
Memory controllers	on User FPGA	8 dedicated FPGAs
Memory Clock	300 MHz*, variable	300 MHz, fixed
Peak Bandwidth	**28.8** GB/s*	**74.4** GB/s
Min. Access Size	384 bytes	8 bytes

when synthesizing new user designs. Finally, even though both platforms come closest to their peak bandwidth with linear access patterns, physically a much smaller access granularity is supported in the Convey HC-1 configuration we utilize.

In Section 4.7, where we assess the effects of the two different approaches to kernel design, we need to compensate for the outlined differences of the hardware platforms.

4.4 Kernel-Centric Acceleration

The general idea of kernel-centric acceleration as followed here, is to identify runtime intense kernels with acceleration potential and execute them on FPGA, and to keep other, possibly complex parts of the application with small contributions to the overall runtimes on CPU. This is a typical approach for accelerating existing software solutions on general-purpose accelerators like GPUs.

In order to identify the candidate kernels, we first performed profiling on CPU. The runtimes of all kernel functions with non-negligible runtimes, aggregated over all their invocations when they are executed more then once, are illustrated in Figures 4.5 and 4.6 for a FullHD input image pair on both CPU platforms. The kernels are sorted by the time of their first invocation, which reflects the overall sequence of cost initialization, aggregation, scanline optimization and disparity refinement, however there are repetition patterns spanning several of those kernels. Based on this result, we selected the 5 aggregation kernels from `horSum` to `scale` and the 5 scanline kernels from `scanUp` to `sumScanlines`. They cover 87% of the total program runtime, which permits by Amdahl's law a speedup of at most 7.8x.

Since both platforms investigated in this work have physically distinct accelerator memory, whenever possible, we want to leave data in this accelerator memory when it is read or modified by several different kernels or several invocations of the same kernel. Therefore beyond the raw execution times, possible data reuse between the kernels was considered. In case of our stereo-matching implementation, the selected kernels cover all cost-volume

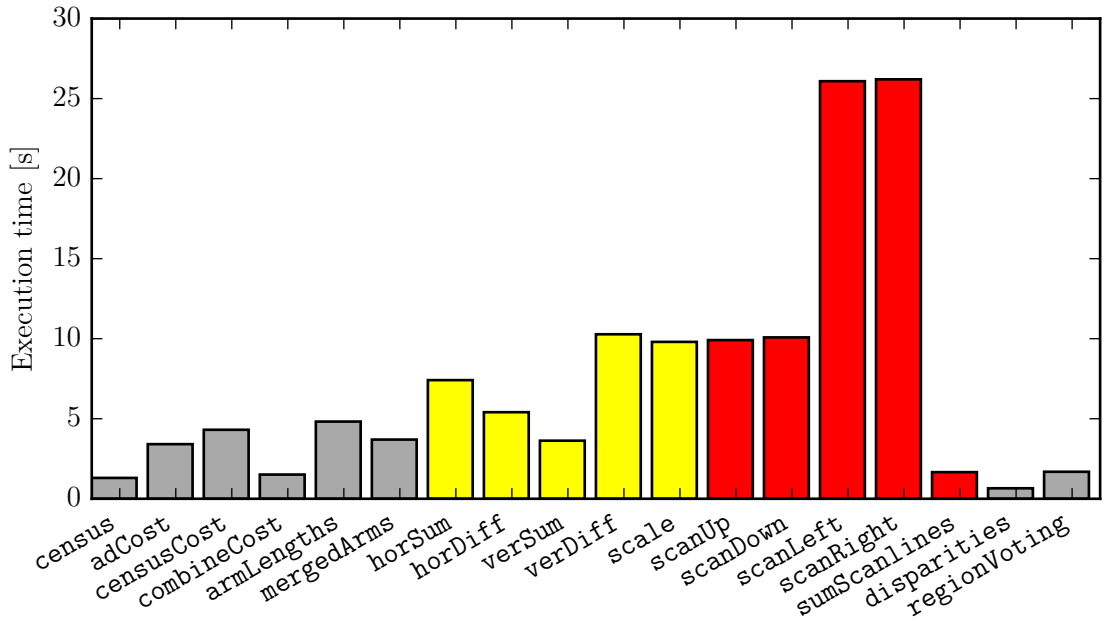

Figure 4.5: Runtimes that different kernels contribute to overall runtime on the Maxeler platform's host CPU for a FullHD image pair (1920x1080) with a maximal disparity of 80. Yellow and red bars indicate kernels in aggregation and scanline phase respectively.

related compute steps of aggregation and scanline optimization, thus maximizing the reuse potential of data in accelerator local memory. Based on pure profiling runtimes, the final step of scanline optimization, `sumScanlines`, would be a less worthwhile acceleration target then e.g. the computation of census costs, but it reduces the amount of data to be transferred from accelerator memory to host memory significantly from four cost volume instances to a single one.

Both utilized target platforms require data to be moved between CPU and accelerator memory, but in different ways. The Maxeler platform [175] requires explicit data movement functionality added to each design by the designer and the accelerator memory space is entirely managed by the developer [176]. The Convey platform [40] provides a shared memory space and different API functions for allocation on and movement between physical memory locations. In order to abstract these differences away from the application side, we modified and extended the memory manager presented in [5] for the Maxeler platform. An important feature of the memory manager, particularly useful during accelerator kernel development, is to support easy switching between CPU and accelerator execution of individual kernels with all required, but no unnecessary data movements.

Our means to achieve this was to express at the beginning of every kernel, which data structure it uses, whether it uses it at the host CPU or the accelerator and whether it reads or writes to this data structure. With this information, the memory manager keeps track of all data locations and initiates all required transfers prior to actual data accesses.

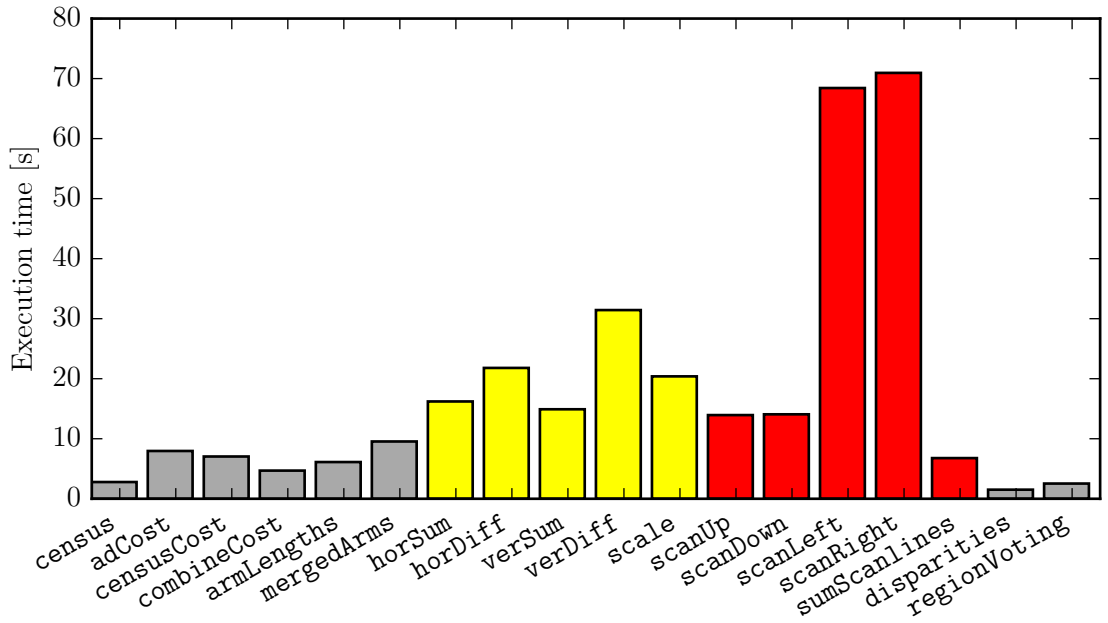

Figure 4.6: Runtimes that different kernels contribute to overall runtime on the Convey platforms host CPU for a FullHD image pair (1920x1080) with a maximal disparity of 80. Yellow and red bars indicate kernels in aggregation and scanline phase respectively.

In our new, extended memory manager concept, we applied these kernel annotations to both the kernels remaining on CPU and the wrappers for kernels executing on FPGAs. This goes beyond the modifications required for the methods presented in [5], where only data usage on FPGAs was indicated. The extension is however advantageous to the kernel centric acceleration concept, because it removes the only high-level application knowledge required for the previous version, where transfers from accelerator memory back to CPU had to be initiated manually, requiring changes for each accelerator kernel that is enabled or disabled during development.

Listing 4.3 illustrates some kernel functions using the memory manager interface. Before they actually use data, they indicate by calls to the memory manager API how (`mm.reads`, `mm.writes`) and where (locations `CPU, ACC`) they are going to use it. When a kernel both reads and writes data, or when it doesn't completely overwrite a structure, so previous data may still exist after writing, this has to be stated explicitly like in this example for the first function, using `b` both as input and output. The accelerator kernels (starting with `cny` for Convey, `max` for Maxeler) are mere wrappers and subsequently invoke execution on the respective accelerator. Due to the shared address space, the Convey kernel uses the original addresses, whereas the locations in Maxeler local memory are provided by the memory manager (`mm.getLMem`). Just like in [5], a memory region in Maxeler local memory is allocated lazily before the first usage of some data structure in this memory.

```
1  cpuABtoB(double* a, double* b){
2    mm.reads(CPU, a);
3    mm.reads(CPU, b);
4    mm.writes(CPU, b);
5    // CPU kernel code here
6  }
7  cnyAtoB(double* a, double* b){
8    mm.reads(ACC, a);
9    mm.writes(ACC, b);
10   callCnyKernel(a, b);
11 }
12 maxAtoB(double* a, double* b){
13   mm.reads(ACC, a);
14   mm.writes(ACC, b);
15   callMaxKernel(mm.getLMem(a), mm.getLMem(b));
16 }
```

Listing 4.3: Different kernel functions using Memory Manager.

Listing 4.4 now illustrates usage of two of those kernels. First, dynamic arrays are allocated through the memory manager, per default in host CPU memory. Then, for the first kernel call on CPU in Line 4, the memory manager determines at runtime that both arrays are already in the right location and no movement is required. In this example, the second kernel, Line 5, is executed on the Maxeler accelerator. For the data it reads, `c_init`, accelerator memory is lazily allocated and data is moved there from host. `c_agg` on the other hand is only written to, so it gets allocated in accelerator memory, but no data is actually moved. Line 6 now performs another kernel call on the host CPU. `c_init` was not modified in accelerator memory, so the memory manager internally still has it in a `shared` state and no data needs to be moved. `c_agg` however was `modified` in accelerator memory and on CPU it will now be read before it is possibly overwritten, so its data is transferred back by the memory manager.

```
1 double* c_ad = (double*) mm.alloc(size);
2 double* c_init = (double*) mm.alloc(size);
3 double* c_agg = (double*) mm.alloc(size);
4 cpuABtoB(c_ad, c_init);
5 maxAtoB(c_init, c_agg);
6 cpuABtoB(c_init, c_agg);
```

Listing 4.4: Code sequence using Memory Manager with Maxeler kernel.

Listing 4.5 repeats the same kernel pattern, just with the accelerated kernel being executed on the Convey platform instead of Maxeler. This time at the coprocessor kernel call in Line 5 no more memory is allocated since host CPU and accelerator share the same memory space. For the input data `c_init`, a similar data transfer is initiated as on the

Maxeler platform, just using a different API with different arguments internally. For the output data `c_agg`, again no physical data transfer is required. For this purpose, the Convey API contains a `migrate_virtual` function, which doesn't actually move any data, but just lets the affected shared memory area point now to the physical accelerator memory. This function comes in two flavors, one that touches all affected memory pages to update internal state such as the TLB (Translation Lookaside Buffer) and the other one without this touching. The version with page touching guarantees the fastest raw execution time of subsequently executed accelerator kernels and therefore is important for the later evaluation of kernel acceleration. On the other hand, we found the no-touch version in combination with allocation on host to yield fastest overall matching performance, because it partially overlaps the page touching effort with actual computation. It is even slightly faster than the alternative direct allocation as accelerator memory, even though the latter would require additional a-priori knowledge about the first usage location of a data structure. Therefore we measure and evaluate both versions in our experimental section.

```
1 double* c_ad = (double*) mm.alloc(size);
2 double* c_init = (double*) mm.alloc(size);
3 double* c_agg = (double*) mm.alloc(size);
4 cpuABtoB(c_ad, c_init);
5 cnyAtoB(c_init, c_agg);
6 cpuABtoB(c_init, c_agg);
```

Listing 4.5: Code sequence using Memory Manager with Convey kernel.

These examples conclude this section on the selection of kernels for acceleration and the concepts and infrastructure to support memory management for both platforms through a common interface.

4.5 Kernel-Designs for two FPGA platforms

In this section we present the compute and data-access patterns of the identified time-consuming kernels and outline their parallelization opportunities, taking dependencies and data locality into account. Subsequently, we discuss the compute and memory access and data reuse patterns we implemented on the two accelerator platforms. The kernels for the Maxeler platform [176] are designed with a flexible amount of parallelism, which is specified by an unrolling factor f_u prior to synthesis. The actually utilized amount of parallelism, typically low two-digit numbers, is either limited by resource or timing limitations during synthesis (HorDiff and Scanline kernels) or by the known limits of the memory interface to feed the compute pipeline (all other kernels). In order to hide feedback latencies in some kernels, in addition to this explicitly utilized parallelism, we also loop through different groups of work items in different clock cycles. For the Convey vector coprocessor [40], the desired amount of parallelism to be expressed by our kernel implementations is given by the size of the vector registers with up to 1024 elements. It internally contains 32 parallel function pipes and additionally makes use of further elements for latency hiding. We

present for each kernel the designs for both platforms side-by-side to emphasize similarities and differences. We outline the designs of the first kernel in some detail whereas for the other kernels we restrict ourselves to noteworthy aspects.

4.5.1 Aggregation Kernels

The cost aggregation involves five different kernels: horizontal integral sums and differences, vertical integral sums and differences and scaling. All aggregation steps are independent for each different disparity value and also for at least one of the image dimensions.

For the Maxeler platform, the independent image dimension suffices to support the required parallelism and latency hiding, so we restrict ourselves to unrolling in this dimension. The work of Shan et al. [230] suggests that utilizing disparity level parallelism in addition to image dimension parallelism might allow to save BRAM resources at the expense of additional logic utilization, which we did not investigate further for our kernels.

On the Convey platform, the image dimensions of small images do not suffice to make best use the available vector width. With the feature *vector partitioning*, loops can be partially unrolled in a second dimension to increase the exploitation of DLP. For vector partitioning, addressing requires, besides the address stride of the primary vector dimension, so-called partition offsets for the second dimension. For the aggregation kernels, we use this feature to exploit both parallelism in image dimensions inside each partition and parallelism in disparity dimensions by multiple vector partitions.

Horizontal Integral Sums

After Section 4.2 already presented simplified pseudocode for the horizontal aggregation step, Listing 4.6 presents the corresponding function with the actual indexing used in our software implementation. There are dependencies along the rows, but we can parallelize computation by vertical unrolling, that is computing several rows in parallel, and additionally work on independent disparity dimensions for Convey vector partitions.

```
1  void horSum(double *in, double *out){
2    long slice = height * width;
3    for(int d=0; d<=maxD; d++){
4      for(int y=0; y<height; y++){
5        out[d*slice + y*width] = in[d*slice + y*width] ;
6        for(x=1; x<width; x++){
7          out[d*slice + y*width + x] =
8            out[d*slice + y*width + x-1] + in[d*slice + y*
                 width + x];
9        }
10     }
11   }
12 }
```

Listing 4.6: Horizontal integral sums.

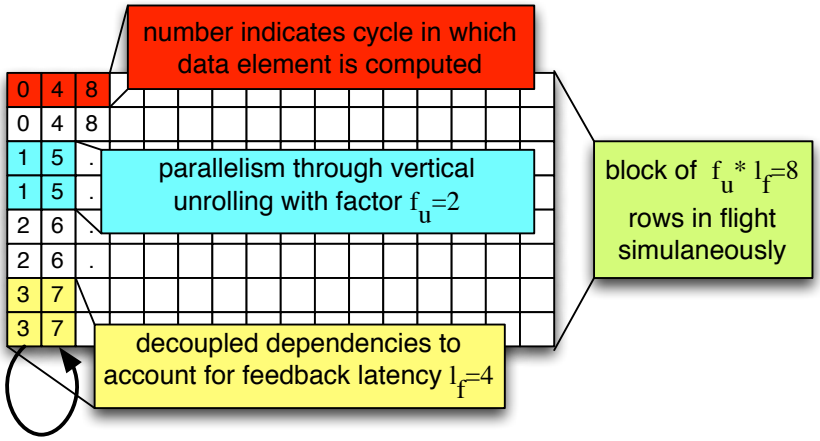

Figure 4.7: Illustration of compute order for horizontal sums on Maxeler. Here, an unrolling factor $f_u = 2$ and a feedback latency $l_f = 4$ are illustrated.

Figure 4.7 illustrates the computation pattern implemented on the Maxeler platform. The product of unrolling factor f_u and feedback latency l_f determines the number of rows that are in-flight at the same time as one common block. The latency is given by estimates from the Maxeler tools, whereas f_u is limited either by bandwidth estimations or synthesis results. More than $f_u * l_f$ rows in the same block are possible, but require larger buffers and provide no further advantages. After an entire block of rows is computed, the next block of rows, not shown in the illustration, is started. Finally, also not illustrated, after one entire image (a slice of the cost volume) is finished, computation continues with the next disparity. In this description, the presented compute pattern now governs the required memory access pattern, however in practice both are closely codesigned.

In memory, elements are arranged in row-major order, which means that entire rows are stored in continuous memory locations one after the other, because LMem uses the same data layout as the host application to allow for the memory management outlined in Section 4.4. Thus each burst of 384 bytes reads 48 subsequent double values from each row. Figure 4.8 illustrates the way data is read from LMem with an appropriate memory command generator. We see that inside each block, between memory accesses and compute step, the data needs to be reordered from horizontal to vertical order. This is relatively easy to do with the *MaxJ* concept of stream offsets, however the actual design may need considerable amounts of BRAM. Additional simpler, non-reordering buffers are required to keep the memory and compute pipelines fed.

The implementation for the Convey vector coprocessor, as indicated in the introduction to this section, not only uses the same unrolling into the independent image dimension, here vertically, but additionally can work on more than one disparity dimension in different vector partitions. Figure 4.9 illustrates this pattern with 4 partitions and 256 rows covered by each partition. The innermost loop runs horizontally inside the rows to reuse the vector register containing the previous integral sum as one of the two inputs for the next step. Before entering this innermost loop, for each group of rows, the number of partitions and

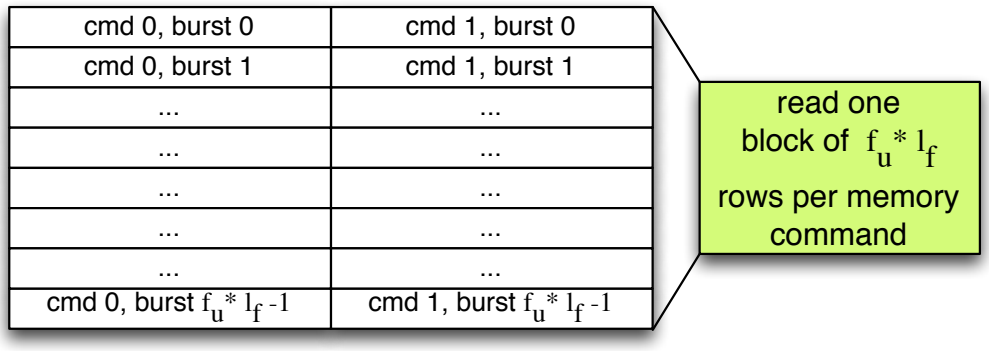

Figure 4.8: Illustration of memory accesses pattern for horizontal sums on Maxeler. Each burst spans 48 elements, every command generates accesses for an entire block of data. Inside each block, data needs to be reordered for the compute pattern.

size of the partitions is computed based on remaining dimensions and two offsets are written into configuration registers. One is the row offset between two consecutive vector elements inside each partition, the other one the image slice offset between two consecutive disparity levels in the cost volume. Vector load and store instructions use these offsets to determine the memory addresses of each vector element and, in this loop profiting from the small access granularity of the scatter-gather RAM, only access the specified vector elements in memory.

Vertical Integral Sums

The vertical integral sum kernel (`VerSum`) is orthogonal to the `HorSum` kernel and contains vertical dependencies. Consequently, we now unroll computation horizontally for the implementations on both platforms.

Our Maxeler compute kernel combines the same combination of unrolling and feedback latency hiding as the `HorSum` kernel illustrated in Figure 4.7, just horizontally. When we buffer entire rows instead of blocks inside each row, the compute pattern exactly fits the data layout in memory, so we can use a linear memory access pattern instead of a customized memory command generator.

Similarly, the Convey `VerSum` vector kernel contains the same features, vector partitioning and data reuse in the innermost loop, as the `HorSum` kernel, but with vectors in horizontal image dimensions. Now the memory accesses inside each vector partition are continuous, which is beneficial for effective memory performance. In the vector processor instruction set, the only difference is, that the element stride is now set to the element size of 8 bytes.

Horizontal Differences

After the computation of the horizontal integral sums (`HorSum`) follows the step of computing horizontal differences (`HorDiff`). For each pixel, a left and a right arm length are

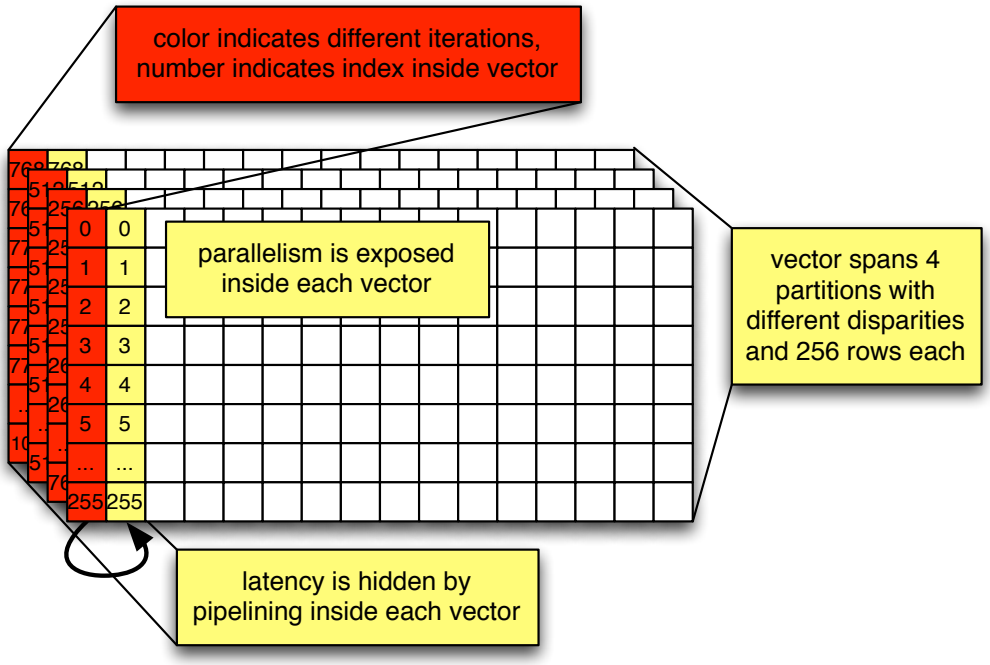

Figure 4.9: Illustration of compute order for horizontal sums on Convey vector coprocessor. Here, 4 vector partitions with 256 elements each are illustrated.

required, which define the two positions in the integral cost rows to access, before the corresponding cost values are subtracted from each other. So in this kernel we have data dependent memory accesses, however only with bounded offsets from a given position, which are limited by the maximal arm lengths. There are no dependencies in this kernel, so both horizontal and vertical unrolling are possible.

Since this kernel does not contain feedback, latency hiding as used on the Maxeler platform for the integral sum kernels is not needed here. With the burst oriented Maxeler memory interface, we need to have the window of possibly required integral cost data available in local buffers. This seems straightforward when unrolling horizontally, since neighboring pixels in one row require largely overlapping areas of possible input values defined by the arms. Figure 4.10 illustrates the use of multiplexers for the selection of the position specified by right arms for an unrolling factor f_u of 4 and with possible values for the arm length of 0 to 4 (in practice we use the a length of up to 34 as proposed in [182]).

Figure 4.10 illustrates, that the overlapping of the possible access windows makes the buffer very space efficient, in the example actually using only 8 registers to buffer the possible inputs for 4 parallel access operations with 5 different input options each. However, even though resource utilization would permit it, the synthesis tools were not able to route such a design with more than $f_u = 8$ anywhere near 100 MHz. The illustration in Figure 4.10 may give an intuitive idea that high number of overlapping signal routes to the different multiplexers causes this problem.

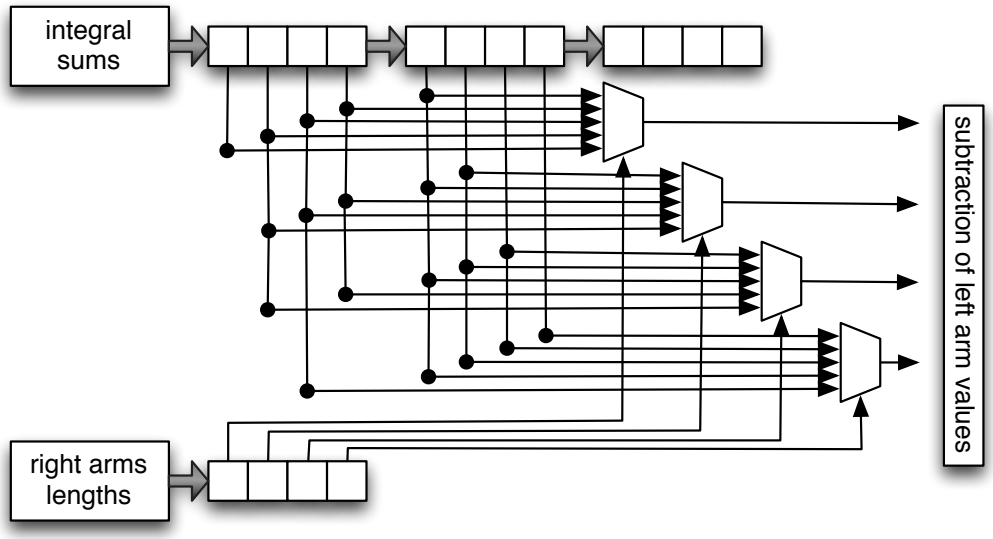

Figure 4.10: Illustration of the selection of the integral sum element specified by a right arm. Together with the left arm counterpart and computation of the difference of their values, this forms a `HorDiff` kernel. From the left side of the illustration, integral sum values and arm lengths are streamed in in groups of 4 according to the unrolling factor f_u and per cycle 4 selected values are output (to the right). In this example we limited the arm length to between 0 and 4, such that a multiplexer of size 5 suffices to select an integral sum element from the group streamed in during the same cycle or from the group of previous cycle.

As an alternative, we tried vertical unrolling like in the previous kernel. Here, in addition to the resource consumption of reordering between row oriented memory access and column oriented compute step, for each parallel row a buffer for the possibly accessed input elements needs to be instantiated. With this approach, unrolling was limited by resource consumption after synthesis. Therefore, specifically for this kernel, in our final Maxeler design we combined horizontal and vertical unrolling, achieving the largest synthesizable design with overall parallelism of 16 through horizontal and vertical unrolling factors $f_{uh} = 4$ and $f_{uv} = 4$.

On the Convey vector processor platform, conceptually the vector registers might provide a similar line buffer for input cost values selected by arms. However, the instruction set does not support any form of parallel access to specific indexed elements of the vector, so this is not possible. Instead, we resort to computing the address of each element of the horizontal integral sums that needs to be accessed by adding the arm length value to each respective base address. Then these addresses are used for indexed vector load operations, which are however less efficient then the accesses with regular strides as outlined for the previous kernel.

We again use multiple vector partitions covering several disparity values in each computation step for efficient utilization of the vector size with small images. Since there is no

dependency of the two inner loops, the parallelism in each partition can be provided either by horizontal or by vertical vectors. After implementing and measuring both alternatives, we use vertical unrolling to form the vectors. When forming horizontal vectors, several loads of input cost values inside a vector load may point to the same location. This works functionally correct, but apparently causes additional latencies in the memory interface.

Vertical Differences

Similarly to the `HorDiff` kernel, the computation of vertical differences (`VerDiff`) does not contain any dependencies. On the Maxeler platform, horizontal unrolling does not suffer from the routing and timing difficulties of the `HorDiff` kernel, because now selection of arm positions is realized independently for each column. Thus, we can restrict unrolling to the horizontal dimension here and still reach unrolling values up to $f_u = 24$. On the Convey platform, we again use vector partitioning and this time unroll the vectors horizontally, following the data access pattern of the vertical summation and again avoiding indexed vector loads to contain several instances of the same address.

Scaling

Finally in the scaling kernel (`Scale`), each aggregated value is scaled, that is divided by the size of its specific aggregation region. It is a straightforward streaming kernel without dependencies and on both platforms can be readily unrolled horizontally, following a linear memory access pattern. On the other hand, the division of a double precision floating point values is neither easy nor efficient to implement on the Maxeler platform and not supported in the vector instruction set of the Convey coprocessor. Fortunately, there is a only fixed number of discrete sizes A_a that any aggregation region can have, so we can precompute the inverse values $1/A_a$ and replace division operations by multiplications with the inverse values. On the Maxeler platform, those precomputed factors are stored in BRAM and looked up locally. For each parallel function pipe, a separate block of BRAM is instantiated. Due to the indexed access pattern, the Convey vector coprocessor again cannot use the vector registers to hold those values, but instead reads them with indexed vector loads from memory. Again, lookups to the same address impair performance, so on this platform we replicate the block of lookup values in memory and use the vector indices to distribute lookups to different blocks.

4.5.2 Scanline Kernels

In contrast to the aggregation, the scanline optimization is not independent for different disparity values. On the contrary, for the computation of the scanline costs of a new pixel, the minimal scanline costs of the previous pixel over all disparities need to be known. There is also a dependency along the scanline direction, such that unrolling can only be performed orthogonal to the scanline direction.

Vertical Scanlines

On the Maxeler platform, we implemented a common vertical scanline compute kernel (`ScanVer`), suitable both for the `ScanUp` and `ScanDown` kernels of the host application, switching between both modes by configuring the accompanying memory command generator for different access directions. Figure 4.11 illustrates the dependency pattern for a downward scanline computation and how it can be unrolled horizontally, here with boxes of size 4. All yellow boxes are required as inputs to compute the red boxes. The aggregation costs are read in in the required pattern as inputs, as well as the color difference information (not illustrated in the Figure) needed to determine the penalty values for each row (boxes P_1 and P_2). The resulting scanline costs are written out to LMem, but for an entire disparity range also buffered locally in BRAM for reuse in the next row. Therefore, computation is performed in blocks, but not in entire rows, as this would require excessive buffer space or additional read-backs.

In our actual Maxeler implementation, due to the burst size of the LMem interface, actual data blocks of 48 horizontal elements are loaded from memory and computed in $^{48}/_{f_u}$ cycles before proceeding to the next line. Since the previous minimum from step 0 is required to update the minimum for step 1, we reordered the datapath for the recursion of Equation 4.1 to have a deeper pipeline for the computation of the individual scanline costs and a simple comparison for the selection of the current minimal scanline value. Nevertheless, similar to the integral sum kernels, we incur a feedback latency l_f of four cycles, which for the block size of 48 limits the possible unrolling in space with unrolling factor f_u to 12 ($l_f * f_u = 48$). We also tried out larger block sizes to obtain more possible compute throughput, but the resulting large designs failed to meet timing.

On the Convey vector processor platform, we implemented two individual assembly kernels for `ScanUp` and `ScanDown` to save unnecessary selection instructions for the direction, but both implementations have identical structures. According to the dependence pattern, vectors cover entire rows or parts of rows, depending on image sizes. In contrast to the aggregation kernels, vector partitioning into different disparity dimensions is not possible.

The compute order looks very similar to the one illustrated for Maxeler in Figure 4.11, just with much larger blocks formed by the vectors. On this platform, the scanline costs of the previous line can't all be buffered for the next line, so they are read back at every iteration of the vertical loop. However, only one disparity block needs to be read for every newly computed block, the other two are reused from the previous iteration of the innermost loop, the disparity loop. E.g. in the Figure 4.11, the yellow block with scanline cost 3 was newly read for the current compute step, the other two yellow scanline costs are reused, which is done efficiently by using the vector register rotation feature of the vector instruction set. Also, our code makes heavy use of vector mask generation and vector element selection to find the different possible scanline paths inside one single vector, which can be used by the vector internal streaming of the coprocessor to skip masked out elements.

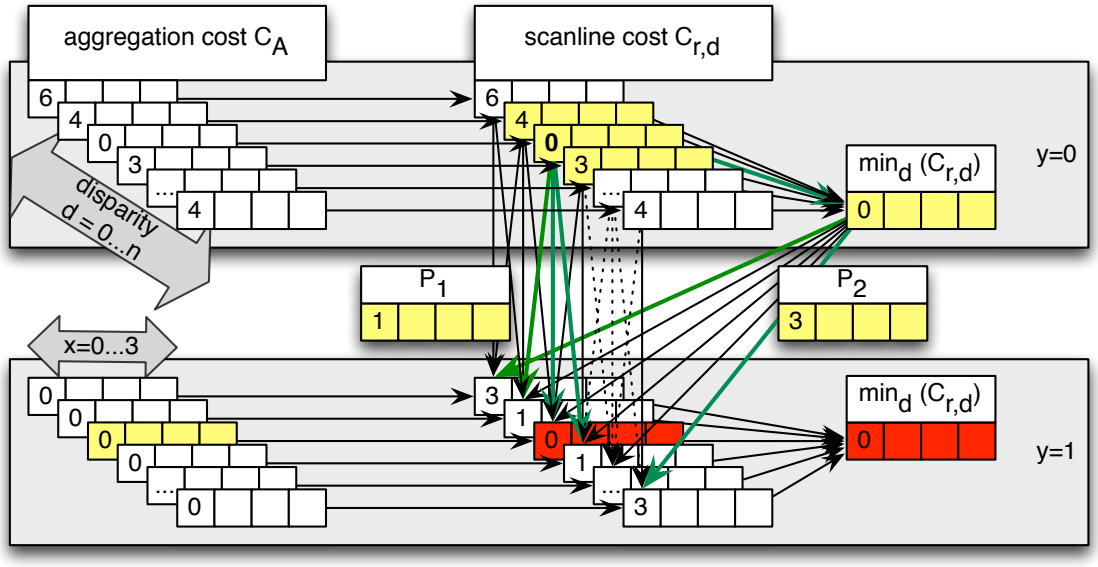

Figure 4.11: Illustration of downward scanline compute pattern. Arrows indicate dependencies. In x-dimension there are no dependencies, blocks of width 4 indicate unrolling in this dimension. For the computation of the red boxes (scanline cost is computed, minimum is updated after every disparity steps), data from all yellow boxes is required. The penalty values P_1 and P_2 additionally depend on pixel color differences not shown here. Green arrows indicate, which paths are chosen to compute the illustrated scanline costs for $y = 1$.

- Red box (value 0): previous scanline cost (value 0) at the same disparity without any penalty.
- Neighboring white boxes (value 1): previous scanline cost (value 0) at neighboring disparity plus penalty P_1 (value 1).
- Outer white boxes (value 3): minimum over all disparities of previous scanline cost (value 0) plus penalty P_2 (value 3).

Horizontal Scanlines

On Maxeler, for horizontal scanlines (`ScanLeft` and `ScanRight`) the orthogonal unrolling concept from vertical scanlines with buffering the entire previous scanline costs was not applicable due to prohibitive BRAM requirements. This is because bursts were still aligned horizontally, but unrolling would have to be done vertically and additionally the buffers would have to cover all disparity dimensions. Therefore we decided to implement an auxiliary turn kernel (`Turn`) that reads cost arrays in row-major data layout and writes them back to LMem in column-major data layout, or vice-versa. Now we can execute horizontal scanlines by a sequence of turning input aggregation data, applying vertical scanlines and turning scanline result data back. The overhead of this turning steps gets mitigated, because both horizontal scan kernels use the same turned input data by utilizing the `ScanUp` and `ScanDown` variants of the vertical scan kernels.

The `Turn` kernel uses 48 BRAM blocks in which data is written to and read from with a diagonally shifted addressing scheme, which provides the flexibility that either an entire row or column of 48 values can be accessed. The size of blocks to be turned has to match at least the 48 elements per burst from the LMem interface, so in contrast to all other kernels we implemented this with a fixed unrolling factor f_u of 48.

On the Convey coprocessor platform, the finer grained access capabilities of the memory interface allow direct implementation of the horizontal scanline kernels without prior turning. The kernel structure is very similar to the vertical one, just using row strides between the vertically unrolled vector elements as for the `HorSum` and `HorDiff` kernels.

Average over Scanline Directions

After computing the costs along all scanline directions, the final scanline costs for each position and disparity is computed by averaging the values of all directions. On both platforms, the resulting `ScanAvg` kernel is a straightforward streaming kernel with four linear input and one linear output streams. As outlined in Section 4.4, its particular value for the overall implementation lies in the reduction of output data size that has to be transferred back from accelerator memory to host memory.

This concludes the part of this section covering kernel designs for both platforms. On the Convey platform, the kernels were directly integrated into an heterogeneous executable by filling empty proxy kernels with the proper signature compiled by the Convey compiler with the actual assembly code providing the described functionality. On the Maxeler platform, the kernels defined in the *MaxJ* language are synthesized to kernel-specific FPGA dataflow designs. The results of the synthesis process are summarized in the following subsection.

4.5.3 Synthesis and Integration

As indicated, most Maxeler dataflow kernel designs are parametrizable at synthesis time with an unrolling factor f_u, which is often constrained by several rules. It must be a whole number divisor of burst sizes, the product of f_u and feedback latency l_f must not exceed burst or block sizes. The `Turn` kernel has a fixed size for the diagonal buffer addressing scheme. The practically possible unrolling factors are further constrained by resource utilization and our decision to aim for a clock frequency of at least 100 MHz for the datapath. We furthermore analyzed the bandwidth requirements and did not investigate unrolling factors which would exceed those by much. The first data column of Table 4.2 summarizes the final unrolling factors utilized for individual kernels. Out of eight individual kernels, six were able to reach or exceed the bandwidth limit. Details of this analysis can be found in [5].

Anticipating some performance results from Section 4.7, in the aggregation phase, there is a high overhead for reconfiguring the FPGAs for the sequence of kernels. Hence, for the five aggregation kernels that are repeated in different cycles during the application, we created a common design (denoted as *MaxFused*) implementing all of their functionality in the same FPGA configuration and thus saving the reconfiguration overhead. We had some headroom for this integration, because not all of the individual kernels hit resource

Table 4.2: Unrolling factors f_u of synthesized kernels. *Asterisks mark unrolling that suffices to reach bandwidth limits. For the aggregation phase, an alternative multi-kernel design (*MaxFused*) with reduced f_u was synthesized.

Kernel	f_u	**Reduced** f_u
HorSum	24*	12
VerSum	24*	12
Scale	24*	12
HorDiff	16	4
VerDiff	24*	12
Turn	48*	—
Scan	12	—
SAvg	12*	—

Table 4.3: Resource utilization of the implemented kernels. Suffixes denote unrolling factors of individual kernels. Critical resources are highlighted.

	Logic	**LUTs**	**Primary FFs**	**DSP**	**BRAMs**
Available	297600	297600	297600	2016	2128
HorSum_{24}	27%	19%	23%	1%	**39%**
VerSum_{24}	**30%**	21%	25%	0%	18%
Scale_{24}	**23%**	15%	19%	6%	22%
HorDiff_{16}	38%	27%	33%	1%	**48%**
VerDiff_{24}	**47%**	40%	42%	1%	23%
MaxFused	63%	50%	57%	8%	**77%**
Turn_{48}	**32%**	23%	27%	3%	29%
Scan_{12}	**53%**	42%	46%	1%	31%
SAvg_{12}	**35%**	25%	30%	0%	23%

limitations, but still we had to decrease the unrolling factors. The final integrated aggregation design was chosen as the optimal trade-off between unrolling and achievable timing and runs at 130 MHz. The second data column of Table 4.2 summarizes the decreased unrolling factors for *MaxFused*. For the scanline phase, no integrated design was found that increased overall performance, not even for small images. In this phase, less reconfigurations are required, so the overhead that can be saved is much smaller. On the other hand, severe reductions of the unrolling factors were required to get routable designs.

In Table 4.3, we finally summarize the resource usage of all used dataflow kernel designs. The table highlights that the individual kernels don't hit hard limits in resource consumption, however for `HorDiff` and *Scan*, no larger design with valid unrolling factor could successfully be synthesized. The critical resources of all kernels are either logic slices or BRAMs.

4.5.4 Kernel Summary

In Subsections 4.5.1 and 4.5.2, the kernels and their implementations are introduced in the order they are used throughout the stereo-matching application. In this subsection, we summarize the kernels based on the encountered computation and data access patterns and identify three groups based on common features. In Subsection 4.7.4, we refer to these groups when analyzing the overheads of the overlay architecture or the advantages of customized kernels.

In the first group, we combine the kernels with a *Regular Streaming* pattern, `HorSum`, `VerSum`, `Scale` and `SumScanlines`. In each of these kernels, there is a common streaming pattern for input and output data, either governed by dependencies (`HorSum`, `VerSum`), or fully unrestricted (`Scale`, `SumScanlines`). Also, the kernels share a low arithmetic intensity and offer no specific data reuse opportunities beyond the simple reuse of the `sum` value from the previous iteration that was already introduced in the example in Subsection 2.1.2 and is used in both applicable kernels on both platforms. In the individual kernel designs on the Maxeler platform, all kernels of this group are sufficiently unrolled to reach the platform's bandwidth limits. In the `Scale` kernel, the multiplication by inverse values of the region size looked up from different memory locations on the two different platforms is an outlier within this group, however because of the other correspondences, we decided to group it here.

In contrast, a second group, denoted as *Complex Streaming* is formed by the four directed scanline kernels (`ScanUp`, `ScanDown`, `ScanLeft`, `ScanRight`). They require to simultaneously stream data from different iteration spaces and have a higher arithmetic intensity on both platforms, because different alternative scanline paths are computed from reused data. With customized block buffers, the customized kernels for the Maxeler platform achieve additional data reuse over those for the instruction-programmable overlay. On the vector overlay, the higher number of operations per volume element compared to the first group of kernels is executed sequentially, whereas the customized kernels pipeline these instructions like introduced in Section 2.1.3 and compute alternative scanline paths in parallel. However, as disadvantage, the customized kernels also use an auxiliary `Turn` kernel and we were not able to unroll them sufficiently to fully saturate the memory bandwidth.

The particular feature of the third group, *Irregular Index Offsets*, is shared by the `HorDiff` and `VerDiff` kernels. By exploiting the known limits of the offsets, we were able to transform this data access pattern into streaming kernels with customized buffers on the Maxeler platform. Due to a combination of resource consumption and routing problems around this buffer, one of the customized kernel designs could not be fully unrolled to saturate the platform's bandwidth limits. In contrast, the instruction set of the vector coprocessor requires and allows us to use indexed vector load instructions to fetch data for each specified offset location from main memory.

4.6 Experimental Setup

In this section, we first present the setup and notation for the evaluated systems and their configurations. We then discuss the generation and selection of our input data.

4.6.1 Evaluated Systems

After implementing and testing all described kernels on both accelerator platforms introduced in Section 4.3, we integrated them into our stereo-matching application and tested it in a total of six different configurations. All host code was generated with the latest GCC versions available as packages for the respective CentOS installations of the two systems, GCC 4.4.6 on the Convey platform and GCC 4.4.7 on the Maxeler platform. Both versions vectorize some simple loops to generate SIMD instructions for the Xeon CPU, but due to limitations of the instruction set and of the analysis don't come close to the vectorization potential manually exploited on the instruction-programmable vector coprocessor of the Convey platform. Our stereo-matching implementation does not support multithreading. Earlier experiments with OpenMP showed a mix of speedups and slowdowns, depending on kernel patterns and utilized CPUs. The coprocessor of the Convey HC-1 is configured with the double precision Vector Personality in version 1.1.1.3 and the Maxeler kernels were designed with MaxCompiler 2013.3 and executed running MaxelerOS 2014.2.

Host CPU of Maxeler platform (*CPU1*)

The entire execution is performed on the 6-core Intel Xeon X5650 CPU with Westmere microarchitecture, running at 2.66 GHz, which is the host CPU of the Maxeler platform [175].

Host CPU of Maxeler platform (*CPU2*)

The entire execution is performed on the dual-core Intel Xeon 5138 CPU with the older Core microarchitecture, and running at only 2.13 GHz, as host CPU of the Convey platform [40].

Kernels with Maximal Parallelism on Maxeler Platform (*MaxKern*)

The first accelerated configuration executes the individual, maximally unrolled dataflow kernels on the Maxeler accelerator card. This design point guarantees the highest raw kernel performance, but induces considerable configuration overheads, in particular during the aggregation phase. The parts of the application that are not accelerated are executed on *CPU1* and the memory manager presented in Section 4.4 handles transfers between host and accelerator memory.

Multi-Kernel Design on Maxeler Platform (*MaxFused*)

The second configuration using Maxeler accelerator card uses the integrated aggregation design, containing five kernels with less unrolling and thus reduced parallelism. The remainder of the execution is identical to *MaxKern*, including utilization of individual kernels

for the scanline phase. This configuration saves a lot of reconfiguration overhead during the aggregation phase in exchange for reduced raw kernel performance.

Coprocessor on Convey platform (*CnyVecTouch*)

On the Convey platform, the host parts of the application are executed on the slower *CPU2* and the accelerated kernels are executed on the vector processor overlay on the FPGA accelerator. Thus, no bitstream reconfigurations are required during application runtime, but only the much smaller kernel code executed on the coprocessor is changed in the different matching phases. As coprocessor memory interleaving mode, we use a 31-31 interleaving mode, which maps memory addresses to the individual memory banks in a way that allows near peak throughput for most possible stride patterns. For our tests, we setup the 24 GB of physical host memory and 16 GB of physical accelerator memory with a windowed memory mode with an 12 GB window of mapped coprocessor memory, 12 GB of pure host memory and 4 GB of pure coprocessor memory. As suggested in Section 4.4, when no actual data has to be transferred, we use two different strategies to migrate allocated memory areas between the distinct physical memory locations. Here, with the first one, all involved pages are touched on the new location to guarantee best raw kernel performance.

Alternative Coprocessor Usage on Convey platform (*CnyVecNt*)

With the second strategy, no-touch, the migration step is much faster and overall matching performance is a bit higher, at the expense of some increased kernel runtimes. All other settings are identical to *CnyVecTouch*.

4.6.2 Input Data

Conceptually, all accelerated configurations profit from larger image sizes and higher maximal disparity values, as parallelism and pipelining can be exploited better and overheads are amortized better by longer computation times, whereas smaller sizes may help the pure host execution by better caching opportunities. Beyond this general rule of thumb, there are different characteristics specific to either the Maxeler or the Convey accelerator platform. On Maxeler, all LMem accesses need to be 384 byte aligned, so in practice we pad all data structures and memory accesses to fit these requirements. This padding is an overhead that doesn't occur for multiple-of-384 dimensions. On the Convey platform, for best performance it is important to fill the 1024 vector elements. This is trivially the case for multiple-of-1024 dimensions, but in the aggregation phase can also be achieved by nicely fitting vector partitions, e.g. for horizontal sums with height 256 and multiple-of-4 disparities as in our earlier illustration in Figure 4.9. Additionally more subtle effects occur when the image sizes interfere with the memory interleaving mode which defines distribution of memory space to different memory bank. However, the mentioned 31-31 interleaving mode makes our experiments relatively robust in this regard.

Table 4.4: Dimensions of low-disparity image series generated from *Tsukuba* image pair.

Name	Width	Height	Disparity	Pixels [$\times 10^6$]	Volume Elements [$\times 10^6$]
Tsukuba	384	288	16	0.11	1.77
HVGA	480	320	20	0.15	3.07
Macintosh LC	512	384	22	0.20	4.33
EGA	640	350	27	0.22	6.05
VGA	640	480	33	0.31	8.29
WVGA	768	480	32	0.37	11.80
SVGA	800	600	34	0.48	16.32
DVGA	960	640	40	0.61	24.58
XGA	1024	768	43	0.79	33.82
XGA+	1152	864	48	1.00	47.78
SXGA	1280	1024	54	1.31	70.78
UXGA	1600	1200	67	1.92	128.64
FullHD	1920	1080	80	2.07	165.89
TXGA	1920	1400	80	2.69	215.04

To summarize, absolute and relative performance significantly depend on the input dimensions for our stereo-matching systems. We therefore decided to perform our measurements with a series of different input dimensions and to use standardized real-world image sizes or screen resolutions, regardless of their suitability for either architecture. In order to generate the input images, we scaled image pairs from the Middlebury benchmark set [222] to the desired resolution with cubic scaling in Gimp. The number of disparity steps to investigate is scaled according to the scaling factor of image width. This is important, because with a too low limit to the possible disparities, matching artifacts occur, which lead to dis-proportionally longer runtimes of the disparity refinement step.

We created two test series, one starting from the *Tsukuba* image pair, which has a low ratio of maximal disparity to image width and one starting from the *Cones* image pair, which has a high ratio of maximal disparity to image width. Tables 4.4 and 4.5 show the two series of input dimensions we investigated. We scaled the two image pairs to different commonly used sizes with pixel ratios between 5:4 (SXGA) and 64:35 (EGA), most of them at 4:3 like the original *Tsukuba* pair. We selected the set of sizes in a way that the number of pixels between two consecutive sizes increases by factors between 1.08 (from UXGA to FullHD) and 1.46 (from SXGA to UXGA) and the number of elements in a cost volume increases by factors between 1.29 (from UXGA and FullHD) and 1.82 (from SXGA to UXGA). In the remainder of this chapter, we reference inputs with `<width>x<height>x<disp>`, for example `1920x1080x80` for FullHD images from the low-disparity image series. This series are currently limited by two aspects. Firstly, a maximal line width of 1920 is synthesized in one of our Maxeler kernels. Secondly, for larger input dimensions, total memory usage starts to become an issue. On the Maxeler platform, with our current implementation of

Table 4.5: Dimensions of high-disparity image series generated from *Cones* image pair.

Name	Width	Height	Disparity	Pixels [$\times 10^6$]	Volume Elements [$\times 10^6$]
Cones	450	375	60	0.17	10.13
Macintosh LC	512	384	68	0.20	13.37
EGA	640	350	85	0.22	19.04
VGA	640	480	85	0.31	26.11
WVGA	768	480	102	0.37	37.60
SVGA	800	600	107	0.48	51.36
DVGA	960	640	128	0.61	78.64
XGA	1024	768	137	0.79	107.74
XGA+	1152	864	154	1.00	153.28
SXGA	1280	1024	171	1.31	224.13

the MemoryManager, the 24GB of accelerator memory put a hard limit to the maximal input dimensions. On the Convey platform, we were able to execute tests with larger input dimensions than in tables 4.4 and 4.5, but performance was impaired by the Linux kernel starting to swap data between main memory and hard disk.

4.7 Evaluation and Comparisons

We first present overall system performance for both platforms. For the main comparison between the approaches of specialized kernels and the reusable vector processor overlay, we focus on the raw kernel performance with both methods and abstract the underlying hardware away as far as possible. Finally, we give some estimates of the design efforts for both approaches.

4.7.1 Stereo-Matching System Performance

Our first charts present speedups for the execution of the entire stereo-matching process for different image sizes compared to pure host execution. For the two respective image series, Figures 4.12 and 4.13 show the speedups of the four configurations with accelerators, *MaxKern*, *MaxFused*, *CnyVecTouch* and *CnyVecNt*, compared to the host execution on the faster *CPU1*. Since the host components of the two *CnyVec* versions are executed on *CPU2*, we also exemplarily include Figure 4.14, where *CPU2* is used as baseline for speedups of the low-disparity test series. Speedups of a hypothetical platform with the Convey HC-1's coprocessor and the faster Xeon X5650 host CPU from the Maxeler platform should be somewhere in between the values from Figures 4.12 and 4.14.

For both test series, we see that both *CnyVec* configurations on the Convey HC-1 [40] can achieve speedups, already at small image sizes which don't fully fill the vector registers. In both series, 512x384 is the first image size, where *CnyVec* achieves little speedups

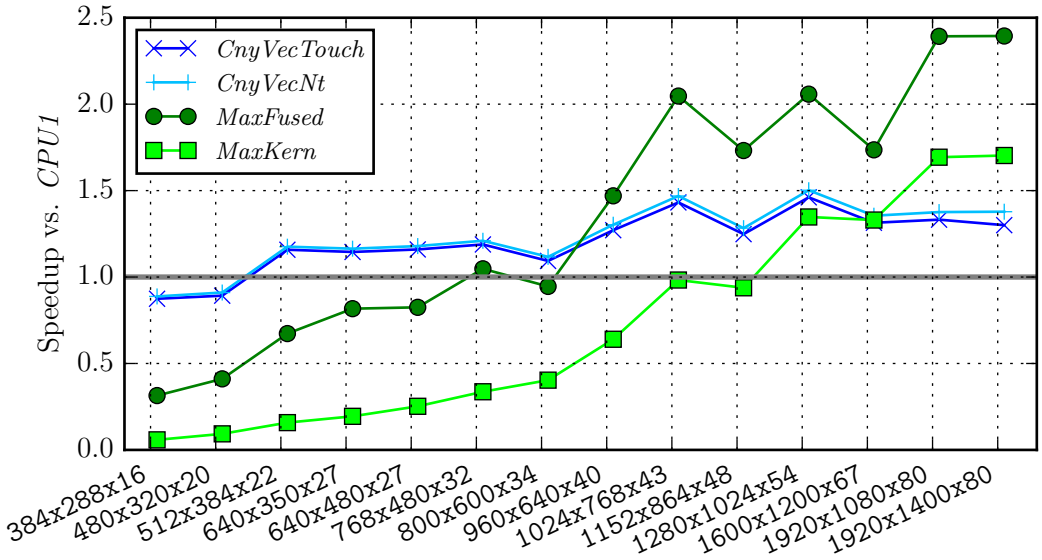

Figure 4.12: Low-disparity image series: Speedups of accelerated configurations compared to faster *CPU1*. *CnyVec* versions are at disadvantage because their host parts run on slower *CPU2*.

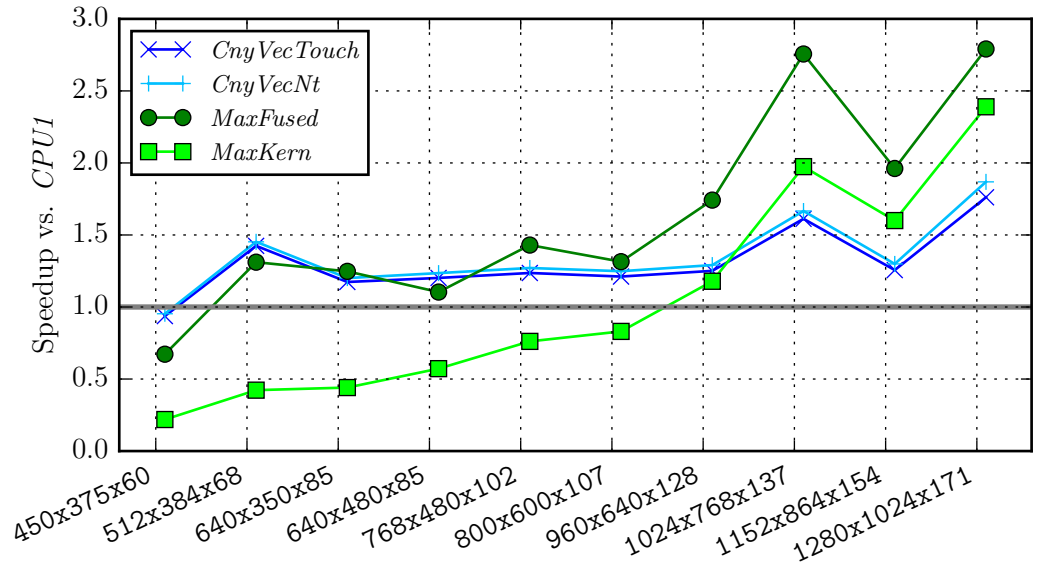

Figure 4.13: High-disparity image series: Speedups of accelerated configurations compared to faster *CPU1*. *CnyVec* versions are at disadvantage because their host parts run on slower *CPU2*.

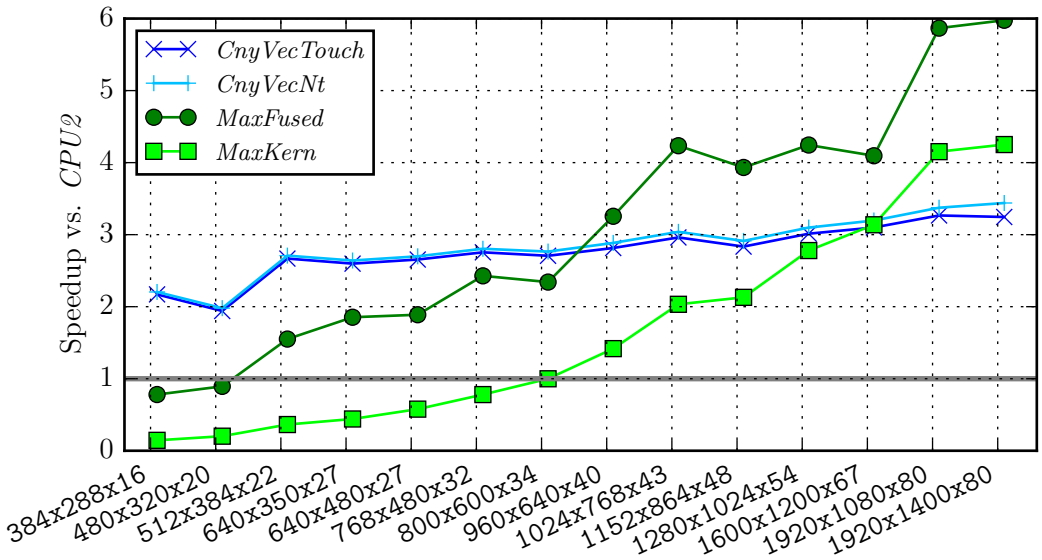

Figure 4.14: Low-disparity image series: Speedups of accelerated configurations compared to slower *CPU2*. *Max* versions have an additional advantage because their host parts run on the faster *CPU1*.

over *CPU1*. The speedups increase slightly with increasing image sizes, but show some variations for specific sizes fitting vector register sizes or memory interface a bit better or worse. At `1280x1024x171`, *CnyVecNt* reaches a peak speedup of 1.9x over *CPU1*. With the slower *CPU2* as baseline, the speedups are around 3x.

On the Maxeler platform [175], the *MaxFused* configuration with a common design for all aggregation kernels persistently outperforms the *MaxKern* configuration, with its individual, maximally parallel aggregation kernels. However, for small image sizes, *MaxFused* is still slower than *CPU1* and both *CnyVec* configurations. In the low-disparity test series, it takes the lead over all other designs for the first time at `960x640x40`. In this test series, its speedup peaks at `1920x1400x80` with 2.4x compared to *CPU1*. *MaxFused* profits from the higher disparities of the second test series, achieving a first speedup over *CPU1* already at `512x384x68` and a peak speedup of 2.8x at `1280x1024x171`.

Figure 4.15 displays some additional details of the underlying data for *CPU1* and the respective faster versions for both accelerators, now showing absolute execution times and subdividing them into aggregation phase, scanline phase and all remaining parts of the application. We see that in the aggregation phase, only *MaxFused* outperforms *CPU1* by a small margin. However, execution of this phase on the accelerator has the additional benefit that afterwards, intermediate results are already in accelerator memory for the following scanline phase. During this scanline phase, now both accelerators achieve significant speedups compared to *CPU1*. During the execution phases remaining on the respective host, *CnyVecNt* notably loses some of its earlier speedups compared to *CPU1*, because its host code is executed on the slower *CPU2*.

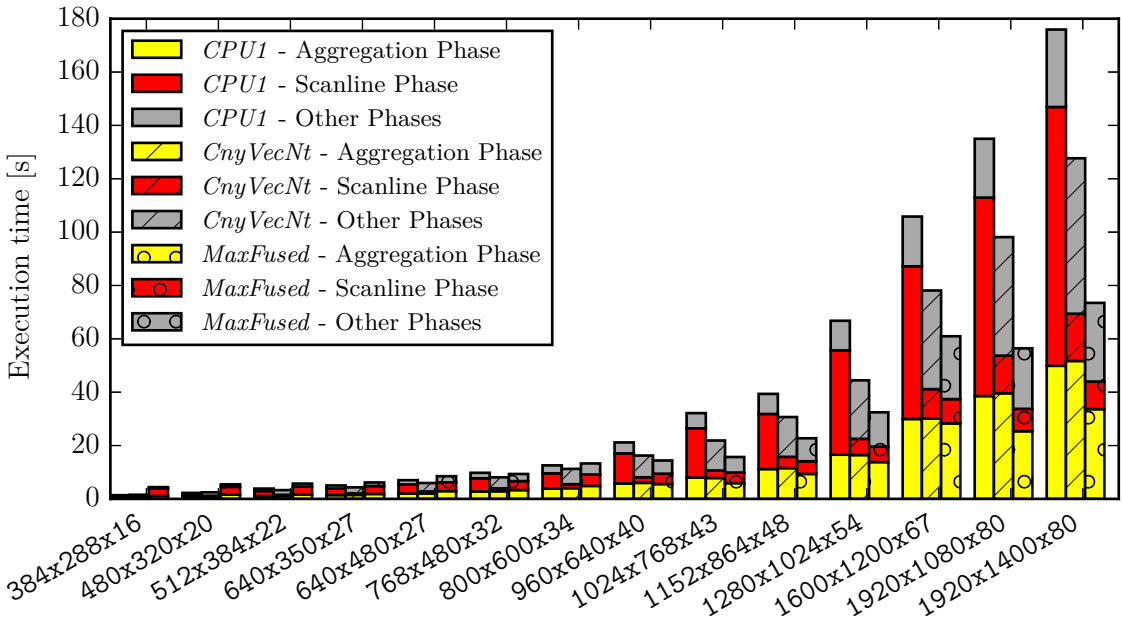

Figure 4.15: Low-disparity image series. Execution times with three different platform configurations (*CPU1*, *CnyVecNt*, *MaxFused*) next to each other for each input size. Stacked bars are divided into aggregation phase (yellow), scanline phase (red) and all remaining parts on host (grey).

4.7.2 Platform Overheads

All further comparisons are only performed with regard to the faster *CPU1*. We proceed with the analysis of the two accelerated phases, in this subsection on the basis of results from `1920x1400x80`, and compare *CPU1* to all accelerated designs. Figure 4.16 breaks down the total execution time of the aggregation phase, splitting the entire yellow blocks from Figure 4.15 into individual components. The first component is the raw execution time of the five described aggregation kernels, still summed up together. We see that this raw kernel execution time is significantly reduced on all accelerator platforms and configurations compared to *CPU1*, down from 47s to between 15s and 24s on the accelerators, with the design with highest parallelism, *MaxKern* executing fastest.

The next component is the total time of all data transfers between host and accelerator memory, that are initiated through our memory manager. For pure CPU execution, naturally no such transfers are needed. Here we see that part of the lower execution times observed on the Maxeler platform in comparison to the Convey platform, are caused by lower transfer times, either because of faster physical interconnect, or because of the overhead incurred for the realization of the shared memory space on Convey. Masking some of this overhead in the *CnyVecNt* configuration overcompensates for the increased raw kernel runtimes compared to *CnyVecTouch*. As third component, we summarized the time spent outside of the five kernels selected for acceleration. In the aggregation phase, this *Host*

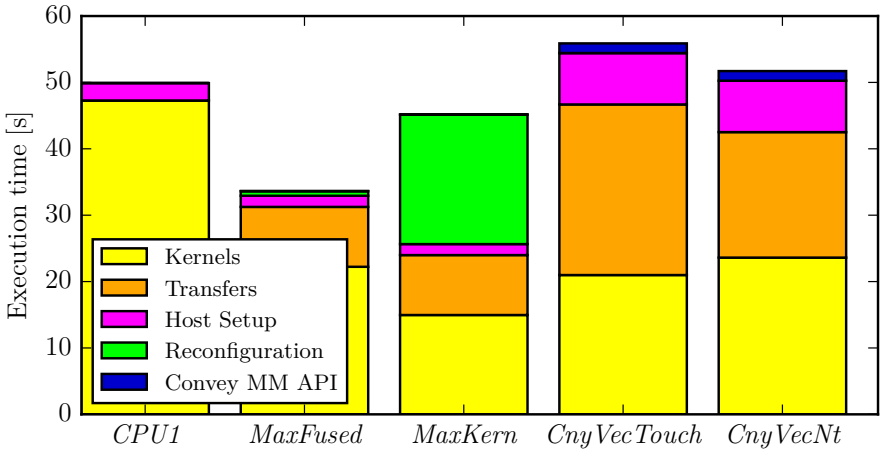

Figure 4.16: Breakdown of aggregation phase into raw kernel execution times and overhead components for different platforms for `1920x1400x80` input image pair.

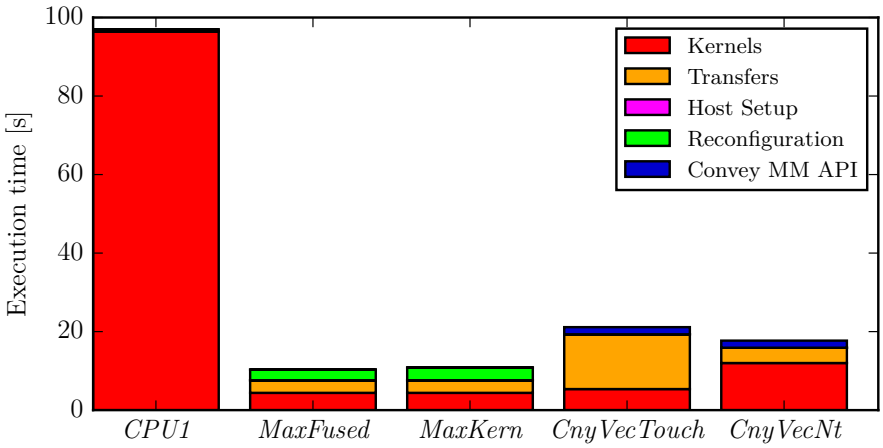

Figure 4.17: Breakdown of scanline phase into raw kernel execution times and overhead components for different platforms for `1920x1400x80` input image pair.

Setup time includes for example the time to initialize the aggregation regions needed for scaling. This phase is notably slower on the Convey platform again because of the slower host *CPU2*.

The fourth component, reconfiguration times, only occur on the Maxeler platform. We see that for the *MaxFused* design, this overhead is negligible as only one reconfiguration is performed, whereas for the individual aggregation kernels in *MaxKern*, it more than eats up the additional speedups achieved in raw kernel execution times. As final component, we measured the time spent in platform specific allocation and free API calls on Convey, which turns out to be relatively minor in the two configurations observed.

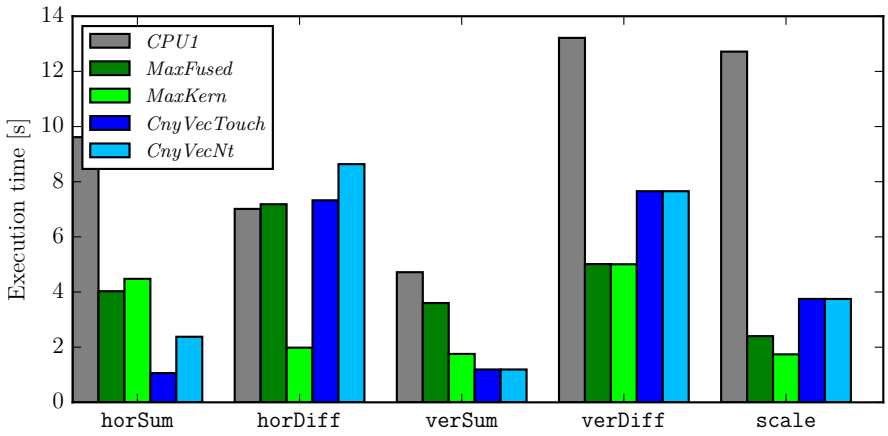

Figure 4.18: Execution times of individual aggregation kernels.

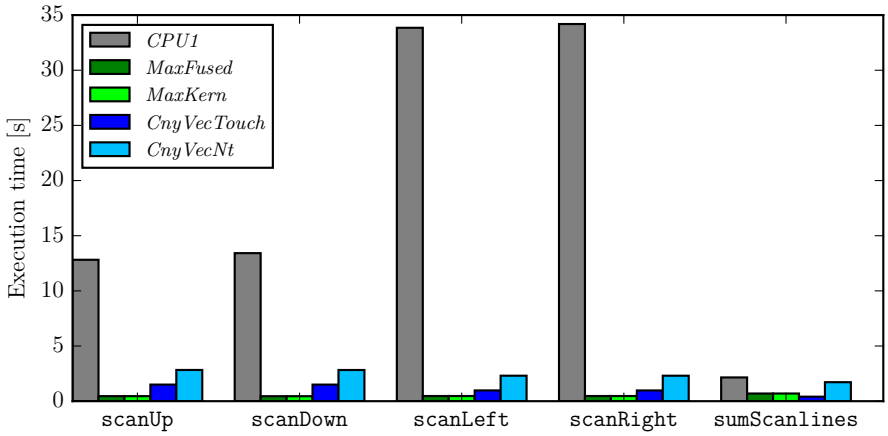

Figure 4.19: Execution times of individual scanline kernels.

Figure 4.17 displays the same components for the scanline phase. In this phase, both Maxeler configurations execute the identical designs and thus perform identically. Due to higher computational intensity and higher data reuse, all accelerator platforms in all configurations reduce the raw kernel execution times much more than in the aggregation phase. Compared to its kernel execution times, *CnyVecTouch* incurs a huge overhead for data transfers, which *CnyVecNt* can again partially mask during kernel execution. Compared to the CPU execution times, these overheads are smaller than during the aggregation phase, thus allowing considerable overall speedups.

4.7.3 Kernel Performance

In order to compare the kernel specific dataflow design approach with the vector processor overlay in regard to their suitability for kernel-centric acceleration, we now look at individ-

ual kernel execution times and disregard the platform overheads discussed in the previous subsection. Figures 4.18 and 4.19 show the raw kernel execution times of the aggregation and scanline phases, again for the largest image pair, now separated into individual kernels, but summing up the execution times of all invocations of the same kernel to be comparable to the previous plots.

Like in the previous subsection, we compare the faster *CPU1* with all accelerated versions. Since the goal in this step is to assess raw performance potential without platform overheads of instruction-programmable overlay and specialized dataflow designs, on the Convey platform we focus on the faster *CnyVecTouch* variant, since with *CnyVecNt*, some of the data transfer overheads are contained within the kernel execution times. These transfer overheads are not related to the overlay approach as such. For the Maxeler platform with specialized dataflow designs on the other hand, we consider both design points. The *MaxKern* implementation is designed to maximize raw performance, whereas the *MaxFused* implementation is designed to mitigate a particular overhead, the reconfiguration step, that the alternative overlay approach deliberately avoids.

For the aggregation kernels in Figure 4.18, we see quite diverse results, with either *MaxKern* or *CnyVecTouch* achieving the best kernel runtimes. Furthermore, we see an unexpected artifact for `HorSum`, where *MaxFused* in spite of less compute parallelism is faster than *MaxKern*. Presumably both kernels are limited by effective memory bandwidth, with *MaxFused* generating a slightly more favorable memory access pattern. The `HorDiff` kernel, in turn, was already projected to be compute bound for the *MaxKern*. The *MaxFused* design with four times less compute parallelism is around 4x slower, supporting this assumption. The `verSum` and `scale` kernels seem to have become compute bound for the *MaxFused* design, showing smaller slowdowns compared to *MaxKern*.

A detailed discussion on the underlying effects for the comparison of dataflow and vectorized kernels needs to take into account the achieved compute parallelism, memory reuse properties as outlined in Section 4.5, bandwidth requirements and the impact of memory access patterns on the achieved bandwidths. Also, during our optimizations of the vector overlay kernels, we saw that performance cannot be easily modeled as a function of compute throughput or of effective memory bandwidth, as it also depends on latencies and sequence of dependent instructions. Therefore, a detailed attribution of certain results to possibly dominating performance factors would be mostly speculation without additional measurements. However, the effects of sensitivity to latencies and those of different arithmetic intensities caused by data reuse and design of operations are attributable to the kernel design paradigm and thus form the actual subject of our comparison. On the other hand, effects of different amounts of available compute resources and memory bandwidths distort this comparison. Hence, in the following subsection, we try to extract the design effects of instruction-programmable overlays versus customized dataflow kernels by compensating for the performance effects purely attributable to the underlying hardware platforms. However, first we proceed with the comparison of kernel performance.

Comparing the runtimes of scanline kernels in Figure 4.19, we see a more homogeneous result pattern, with the most notable observations the difference in CPU performance in horizontal and vertical directions and that the specialized kernels dominate for the

Table 4.6: Kernel-Ratios (geometric mean of all kernel speedups) relative to *CPU1* for the SXGA image with high disparity (`1920x1400x80`) and for the geometric mean over all sizes.

Architecture	**Speedup over *CPU1***	
	All Sizes	**SXGA-high**
MaxKern	6.59	9.86
MaxFused	5.20	7.76
CnyVecTouch	5.80	8.37
CPU2	0.40	0.38

actual scanline computation whereas the vector overlay takes the lead for the subsequent summation step.

In our concrete stereo-matching application, the various kernels do have their individual, well defined contributions to the overall runtime. However, when comparing the two design methods in regard to their general suitability for kernel-centric acceleration, we want to abstract from these individual kernel weights and just profit from the variety of compute and data-usage patterns represented by different kernels. Therefore, we consolidate these results into a single metric, the geometric mean of individual kernel speedup factors that each approach achieves over the reference CPU execution. We denote this metric as *Kernel-Ratio*, analogical to the similarly computed SPECRatio[1]. This metric has the nice property that the choice of reference platform doesn't impact relative ratio between two other platforms.

Table 4.6 summarizes those Kernel-Ratios for three accelerated designs with reference to *CPU1* for the geometric mean of all input sizes and individually for largest problem size tested, SXGA with high disparity (`1920x1400x80`). The reference invariance of the Kernel-Ratios metric allows to directly derive additional ratios between the platforms in the list. So, considering the comparison of the two kernel design approaches, for all image sizes the Kernel-Ratios of *MaxKern* with reference to the slower *CPU2* is computed as $\frac{MaxKern}{CPU2} = \frac{9.53}{0.38} = 24.84$. Similarly, the kernel acceleration methods can be compared directly to each other without any CPU implementation as reference, e.g. again for TXGA resolution the the Kernel-Ratio of *MaxKern* with reference to *CnyVecTouch* is computed as $\frac{MaxKern}{CnyVecTouch} = \frac{6.59}{5.80} = 1.14$. Similarly, for SXGA high-disparity test, it is computed as $\frac{MaxKern}{CnyVecTouch} = \frac{9.53}{8.37} = 1.18$.

The results, when comparing the Kernel-Ratios of the two specialized dataflow kernel approaches on Maxeler with the vector overlay on Convey, are surprising. In the geometric mean, the specialized kernels are just marginally faster than the vector overlay. When trading off parallelism for the integration of several specialized kernels in *MaxFused*, the specialized kernels are even slightly slower than the overlay, for all image dimensions with $\frac{MaxKern}{CnyVecTouch} = \frac{5.20}{5.80} = 0.90$ and for high-disparity SXGA with $\frac{MaxKern}{CnyVecTouch} = \frac{7.76}{8.37} = 0.93$. However, as indicated above, these numbers do abstract away the data transfer and re-

[1] `https://www.spec.org/spec/glossary/#specratio`

Table 4.7: *Hardware-Normalized Kernel-Ratios* (geometric mean of all kernel speedups, normalized with regard to compute resources and bandwidths) relative to *CnyVecTouch* for the SXGA image with high disparity (`1920x1400x80`) and for the geometric mean over all sizes.

Architecture	**Hardware-Normalized Speedup over *CnyVecTouch***	
	All Sizes	**SXGA-high**
MaxKern	3.04	3.16
MaxFused	2.41	2.48

configuration overheads, but still contain the mismatch in available compute resources and memory bandwidths.

4.7.4 Quantifying Overlay Overheads through Hardware-Normalization

We try to extract the effects of different design approaches, that is the overhead of the vector overlay over custom kernel designs, on the two platforms by compensating for the effects of underlying hardware. For this, we need metrics to compare the hardware platforms and approach this by looking at compute resources and memory bandwidth. When we compare basic compute resources in terms of 6-input LUTs, which are common to Virtex-5 and Virtex-6 FPGAs, we can observe a ratio of Maxeler to Convey hardware of $\frac{297600}{829440} = 0.359$. Similarly, the ratio of theoretical peak memory bandwidth can be computed as $\frac{28.8\ GB/s}{74.4\ GB/s} = 0.387$. For a somewhat sophisticated compensation of hardware configurations, we would like to offset each observed kernel speedup with one of those factors, depending on whether the kernel is compute or bandwidth bound. However, since the factors are similar, we just average those two ratios to 0.373. We multiply Maxeler to Convey Kernel-Ratios by the inverse of the combined hardware ratio in order to normalize performance to comparable hardware platform characteristics. This leads to a metric we denote as *Hardware-Normalized Kernel-Ratio* and present in Table 4.7. We can summarize these results as central contribution of this chapter as follows:

> In a diverse set of compute kernels with data parallelism, specialized dataflow kernel implementations on FPGAs are on average around 3x more efficient in terms of performance than a reusable vector processor overlay implemented on comparable hardware. In a concrete scenario, due to trade-offs between reconfiguration overheads and exposed parallelism, this advantage shrinks to around 2.5x.

The 3x performance overhead notably corresponds to the comparison of the manycore overlay MARC to a single customized FPGA design in [164], however this agreement may well be coincidental. We here need to discuss the circumstances and limitations that govern the general applicability of our results. First of all, the utilized method of normalizing for different hardware platforms by a single compensation factor depends on the similar ratios

of compute resources and bandwidths. Once those differ considerably, such scaling needs to be done on a per-kernel basis after an analysis whether compute or bandwidth would be the limiting factor. For the dataflow kernels, the foundations for such work are present in [5], but for the vector overlay, the performance bounds are hard to quantify since all our kernels are actually constrained by a combination of computation, latencies and bandwidth. Also, after migration from one hardware platform to the other, the performance bounds can be different, requiring a more elaborate compensation step.

Secondly, we need to discuss aspects of memory bandwidth. The peak bandwidth data we utilized for our normalization already incorporates two aspects from our practical results. On the Maxeler platform, the memory controller is part of the synthesized FPGA design. The theoretical bandwidth maximum can be achieved with the memory controller clocked at 400 MHz. Due to difficulties to meet the timing of this controller after synthesis, we targeted 300 MHz in our experiments and the peak bandwidth value used in our calculation reflects this. On the Convey platform, the memory controllers are implemented in separate FPGAs and their design is fixed, running at 300 MHz. As reported, we utilize a 31-31 interleaving scheme, which maximizes actual performance in our measurements, but technically reduces peak bandwidth to $\frac{31}{32}$ of the physical interface capabilities, which we also included in our numbers.

The practically realizable bandwidths of both memory interfaces depend, beyond those peak numbers, on additional influence factors, like burst sizes, strides and granularity, which are hard to quantify without extensive tests on both platforms. However, we can qualitatively state, that the efficient support for element-wise vector memory operations, in particular indexed ones, of the vector overlay depend on the capability to access individual 8-byte blocks enabled by the scatter-gather RAM modules of the Convey platform we use. So we need to constrain our Hardware-Normalized Kernel-Ratio results for this design approach with:

> The vector processor overlay requires a memory interface with sufficiently fine access granularity in order to achieve the indicated performance efficiency.

Thirdly, we want to discuss the compute resources. Our scaling method depends on the implicit assumption, that performance scales linearly with available hardware. When it comes to parallel execution units that operate on unrolled data and are implemented primarily with LUTs, this assumption makes sense. However, other aspects of resource usage often doesn't scale linearly with compute throughput. On the one hand, some parts of the designs remain constant, e.g. in our experiments the control logic of the dataflow kernels and the scalar processing units of the vector overlay. On the other hand, resource demands of some components grow more than linearly, e.g. those of some data reordering buffers or input selection multiplexers.

Also, the FPGAs of the two utilized platforms have different ratios of additional resources as BRAMs and DSP-blocks to LUTs, which the scaling in terms of raw logic resources neglects. In particular, as seen in Table 4.3, the current designs of several of our dataflow kernels rely on the high ratio of BRAMs to LUTs on the Maxeler platform's Virtex-6

SX475T FPGA, which is $\frac{\#36Kb-blocks}{\#LUTs} = \frac{1064}{297600} \approx \frac{1}{280}$. On the Virtex-5 LX330 FPGAs of the Convey platform, this ratio is lower: $\frac{\#36Kb-blocks}{\#LUTs} = \frac{288}{207360} = \frac{1}{720}$. However, again as a qualitative statement from our design experience of the dataflow kernels, we note that many of the BRAM resources are directly dedicated to buffering or reordering kernel inputs, outputs and intermediate results in order to properly utilize the burst-oriented memory interface of the Maxeler platform. So, the second addendum to our Hardware-Normalized Kernel-Ratio result now states more precisely for the other design approach:

> Dataflow kernels can achieve the indicated performance efficiency even with a burst-oriented memory interface, but require FPGAs with a sufficiently high ratio of BRAMs to LUTs for this.

Finally, we need to discuss clock frequencies and low-level optimization. Our data-flow kernels are generated with the Maxeler design flow, which enhances design productivity by transparently applying a number of best-practice decisions, e.g. to pipelining or organization of buffers. Many of these can be modified manually, but in our designs such optimizations were mostly performed demand driven, in response to specific timing or resource problems. In order to relax the need for deep pipelining and along with it the need to very carefully optimize the balancing of pipeline stages and their physical layout, most compute paths of our dataflow kernels run at modest 100-130 MHz. For the much more widely distributed and reused vector overlay on the Convey platform on the other hand, common sense and anecdotal evidence suggest, that a huge amount of effort and expertise was invested into low-level optimizations. Consequently, this design runs at 300 MHz, which has a large impact on the performance we measured and compared in this work. We do consider this difference as characteristic for the relationship between reusable and problem-specific designs and as such not as a weakness of the comparison, but nevertheless want to state this in a third addendum to our overall findings:

> Our comparison premises that much more manual low-level optimization effort is put into a reusable overlay design than into problem-specific dataflow kernels.

4.7.5 Overheads by Kernel Groups

In order to proceed from the quantification of overlay overheads in a practical application with diverse kernel patterns, towards insights into the nature of these overheads, we apply the hardware-normalization step from the previous subsection to the three distinct groups of kernel types as identified in Subsection 4.5.4. Table 4.8 presents the results for both design alternatives of customized kernels.

When analyzing the custom kernel designs with highest parallelism as represented by *MaxKern*, we observe a bipartition of results. For the *Regular Streaming* kernels, the customized kernel designs are on average just slightly faster than the kernels running on the vector coprocessor overlay. As the custom variants of these kernels are sufficiently

unrolled to reach the bandwidth limits, we may attribute the small performance difference to improved overlapping of memory access and computation compared to the instruction-programmable overlay and to the more local table lookup for the `Scale` kernel.

For both other kernel groups, we observe similar speedups slightly above 5x. This illustrates that different customization aspects can increase performance over a reusable overlay, or to put it the other way round, that the fixed structure of the instruction-programmable overlay incurs different forms of overheads. In the *Complex Streaming* group, the customization allows for internal pipelining of operations, the parallel execution of different branches and improved data reuse through a blocking-optimized compute order. In the group with *Irregular Index Offsets*, the central customization advantage is the data buffering and reordering that on the one hand improves effective memory bandwidth through long bursts instead of individual word reads, and on the other hand avoids latency-induced limitations by decoupling memory accesses from computations.

For some of these differences, there are approaches to reduce vector overlay overheads through architectural enhancements or customization. With scratchpad memory instead of vector registers, like included in VEGAS [56] and VectorBlox MXP [226], the improved data reuse in our *Complex Streaming* kernels would likely be possible, but might involve compromises on the size of vector instructions that could mitigate the benefits of large vector instructions on the instruction overhead. The *Irregular Index Offsets* of the other kernel group, in contrast, could likely not be supported efficiently by banked scratchpad memories like proposed in the VEGAS and MXP designs. The custom vector instructions introduced for the VectorBlox MXP design [227] additionally aim at transferring the advantages of data pipelining, like exhibited in the *Complex Streaming* kernel group, from custom designs to vector processor overlays. The performance results of their research are very promising, however it remains unclear, how this approach can be combined with toolflows for high productivity.

4.7.6 Estimates on Design Efforts

As final step of our comparison of the two design approaches, we want to present some empirical data about our experienced productivity when performing kernel-centric acceleration with two different design philosophies and targets. This subsection contains a subjective assessment from the perspective of developers with special interest in reconfigurable architectures and designs and does not correspond to the perspective of software developers from the general-purpose domain, as discussed in Section 3.1. Also, as we did not systematically track the design process and many factors which are hard to quantify impact the perceived productivity, these results need to be contemplated with at least a grain of salt. The design and implementation results presented in this work were done in several disjunct phases and with different levels of experience gained from other projects.

Overall we would describe the dataflow kernel design process as two phases, the first starting with some limited amount of experience in dataflow kernel design with the Maxeler toolflow, spanning the equivalent of 8-10 full-time developer weeks for conceptualization of kernels and their unrolling patterns, implementation and many stand-alone tests in simu-

Table 4.8: Grouped *Hardware-Normalized Kernel-Ratios* (geometric mean of all kernel speedups, normalized with regard to compute resources and bandwidths) relative to *CnyVecTouch* for the SXGA image with high disparity (`1920x1400x80`) and for the geometric mean over all sizes. The results from Table 4.7 are now split into the three groups of kernel types identified in Subsection 4.5.4. The *Regular Streaming* group contains the kernels `HorSum`,`VerSum`,`Scale` and `SumScanlines`. The group *Complex Streaming* consists of the four directed scanline kernels that combine different streaming patterns with data pipelining opportunities. The group with *Irregular Index Offsets* combines the `HorDiff` and `VerDiff` with their irregular, but bounded index offsets. *The *Complex Streaming* kernels are identical in the *MaxKern* and the *MaxFused* configurations.

Architecture	**Kernel Pattern**	**Hardware-Normalized Speedup over *CnyVecTouch***	
		All Sizes	**SXGA-high**
MaxKern	*Regular Streaming*	1.28	1.36
	*Complex Streaming**	5.50	5.84
	Irregular Index Offsets	5.27	4.99
MaxFused	*Regular Streaming*	0.98	1.04
	*Complex Streaming**	5.50	5.84
	Irregular Index Offsets	2.79	2.60

lation, along with early synthesis results to get a feeling the resource usage characteristics. The second phase, conducted with much additional background of the Maxeler platform, took another 6-8 weeks with focus on integration, synthesis and optimization.

This phase was in practice prolonged by the process of waiting for synthesis results, which we tried to exclude from the above reported time span, because it to some degree depends on the amount of parallel synthesis resources and to some degree can be covered organizationally, e.g. by running synthesis over night. As an illustrative number: the total tool runtime for the final design of *MaxFused* was reported as 22 hours, 5 mins, 11 secs. Within this time, for the place and route step, a total of 11 different guiding parameter sets (denoted as cost tables by the toolflow) were explored, with four parallel instances running concurrently. Another special challenge was posed by one kernel instance, where the Maxeler simulation tool was not able to reproduce a memory interface related error actually encountered in hardware.

The design of the vector coprocessor kernels was also performed in two major phases. An equivalent of 4-6 full-time developer weeks was spent for first concepts and prototypical implementations with no preliminary knowledge of the concrete vector ISA, but with some general background in assembly programming. With a lot more experience with the architecture, another 6-8 weeks was spent for the final kernel designs and optimizations, including a considerable fraction of the time that was spent in exploring performance impacts of memory settings and data transfer patterns triggered through our memory manager. On this platform, assembly of a kernel design and integration into an executable was completed within seconds, allowing for much faster optimization iterations. A special

challenge was posed by repeated crashes of the accelerator hardware that occurred when using the debugger for the vector coprocessor.

To summarize our concrete experience with both design approaches, designing specialized dataflow kernels with Maxeler's spatial programming language and design flow requires some more time and some more expertise than developing assembly code for a vector coprocessor, but not a whole lot. However, the time-consuming synthesis adds some tedious waiting to the process.

From the perspective of suitability for general-purpose computing, as contemplated in Section 3.1, the assessment is very different. Designing dataflow kernels is hard with a pure software background, whereas a number of software developers are to some degree familiar with assembly code. However, the design of low-level assembly code is not a promising option with regard to productivity and portability. Hence, in order to serve as a path towards general-purpose computing, the speed of the compilation flow for the instruction-programmable overlay needs to be combined with a high-level design entry and a high degree of automation, as we present it in Chapter 5.

4.7.7 Limitations of the Comparison

The presented comparison leaves out two important aspects of the broader computing landscape as outlined in Chapters 2 and 3, a comparison to GPUs and evaluation of energy efficiency.

GPUs are left out here to restrict the scope of the comparison and because of limited development resources. However, it is clear from the characteristics of the workload with abundant DLP and bandwidth sensitivity, that GPUs with sufficient double precision floating point units fit the application demands very well. The orthogonal data access pattern in successive aggregation steps are likely to also impact the memory and cache efficiency of GPUs, but the raw bandwidths of high-end GPU computing products is still useful. The well performing single precision GPU implementation for small images in [182] underlines this. Without further customization of algorithms, data types or operators, for this type of application, FPGA accelerators are generally not likely to surpass GPUs in terms of performance.

Considering a GPU comparison, but also from a general perspective, an evaluation of the energy efficiency of stereo-matching on the two FPGA platforms would be extremely interesting. The accelerator boards of the Maxeler platform, with a peripheral component interconnect express (PCIe) form factor that appears close to commodity hardware, show a with peak power consumption of around 60W when running our kernels, which is much lower than high-end GPU computing products. On the other hand, they are much less optimized for idle power, drawing around 50W when a kernel design is loaded and around 25W when an explicit idle bitstream is loaded. For the Convey platform, we have not performed corresponding measurements. However, the concrete platform characteristics as laid out in Section 4.3 would heavily restrict the validity of any energy measurements. Beside the generational difference of utilized FPGAs, the higher number of individual FPGAs and DIMMs, as well as the customized DIMM modules themselves, of the Convey platform

cause overheads for off-chip communication that are hard to quantify. These may or may not easily overshadow any energy differences caused by the different design approaches, including clock frequencies, instruction overheads and activity rates of functional units.

4.8 Related Work

In this section we present related work on stereo-matching systems on FPGAs. Related work in systems and overlays is discussed in Chapters 2 and 3 and compilation for vector architectures is the topic of Chapter 5.

Apart from our own previous work in [5] and [4], stereo-matching on FPGAs has been tackled with co-design of algorithm and hardware, typically implementing the entire processing pipeline without off-chip memory accesses. Different algorithmic approaches have been explored with different design goals in mind. For example Tippetts et al. [254] present a complete stereo-matching system with less than 10,000 LUTs and 30 BRAMs, at much lower result quality, but robust in respect to uncalibrated and unrectified images. Apart from simple pre- and post-processing steps, their approach employs an intensity profile shape matching algorithm, that directly works on row-local intensity data.

The FPGA implementations with highest matching accuracy reflect more of the matching patterns utilized in this work. Shan et al. [230] implemented a slightly modified variant of the presented cost aggregation for adaptive support regions on FPGAs. By aggregating only once and in a fixed order, first vertically and then horizontally, they are able to stream the required data only through on-chip buffers. Wang et al. [265] try to follow the algorithm of Mei et al. [182] in their FPGA implementation more closely. In addition to the aggregation technique of Shan et al. [230], they propose a reduced scanline optimization which runs in three downward directions, following the order the data is generated in the previous aggregation stage. Both implementations try to exploit parallelism both in the spatial domain of the images, working on several rows at once, and in the disparity domain of the cost volume, working on several disparity images at once.

Jin et al. [144, 145] use a similar single-pass aggregation phase and winner-takes-all disparity selection and combine it with a voting scheme, denoted fast locally consistent (FLC) [174], which is more sophisticated than the one utilized in the post-processing step we employ. Between these two phases, intermediate disparity results are actually buffered off-chip, but require much less bandwidth than for our approach, since no volume data is stored.

These implementations come quite close to our results in terms of matching quality, with Wang et al. [265] reaching an average of 6.17% bad pixels and Jin et al. [145] of only 5.86% bad pixels, compared to 5.73% of our implementation. They are more limited in problem dimensions than our approach that works on blocks of memory, with Jin et al. [144, 145] projecting a design that supports our largest test inputs to exceed the LUT and BRAM resources of their and our current hardware platform, but may be suitable for large Virtex-7 FPGAs. In terms of performance, these co-designed implementations are orders of magnitude faster than our implementation, by executing less computation steps on volume data and by integrating the compute pipelines more tightly. Therefore, these

approaches are superior when algorithmic trade-offs can be made, whereas our approach is justified, when exact reproduction of results or a simpler, structured design process are required.

4.9 Chapter Conclusion

In this chapter, we have presented and compared two design approaches for kernel-centric acceleration of a general-purpose application on FPGAs, specialized dataflow kernels versus an instruction-programmable vector processor. We have shown that the fixed vector overlay can be used to achieve actual speedups over GPPs, but compared to specialized dataflow kernels incurs around 3x slower raw kernel execution times. By distinguishing different groups of kernels, we were also able to point towards design aspects that cause this performance gap. Due to trade-offs between reconfiguration overheads and exposed parallelism, in our concrete scenario the advantage of specialized dataflow kernels is reduced to around 2.5x, which may be an interesting indicator for particularly dynamic workload characteristics like in OTF computing.

Also not explicitly designed for it, this work may also serve as starting point for a stereo-matching library on various interface levels. In compiled binary, for the Maxeler platform along with synthesized designs, it can be exposed as entire application that gets fed a pair of input images and outputs a resulting disparity map, or as the two sets of kernel implementations that are encapsulated behind common interfaces. For different Maxeler platforms, a library maintainer might be requested to provide synthesized designs to ensure sufficient resources and a working placement, routing and timing. For different ISA and management interface compatible vector overlays, note that none is known to us, the assembly code would be sufficient to ensure quick portability.

The vector assembly code uses a built-in constant register that specifies the maximal vector length and thus can transparently scale to designs with different vector lengths, however, some design decisions may then be non-optimal, for example with regard to vector partitioning. As discussed, the spatial dataflow kernels also have parameterizable parallelism, but require lengthy re-synthesis, possibly several runs to find the best fitting resource consumption and timing.

We have furthermore reported on the design efforts for these solutions. While the runtimes for compilation for the overlay vs. synthesis of the fully custom designs are vastly different and did have practical impact on overall productivity, the assembly-language implementations for the vector overlay were chosen to come as close as possible to the maximal performance of each platform, but does not represent a high-productivity approach as proposed in Chapter 3 as overlay pillar. Towards such productivity, compilation from high-level source code is required, which we show in Chapter 5 is possible, but was lacking heavily in features and performance in the tool chain shipped with the Convey HC-1. The spatial programming model for the Maxeler platform on the other hand allowed to generate kernel specific FPGA designs at a reasonably high abstraction level. However, with a design paradigm that is unfamiliar to software developers and as of now not compatible with targets outside the Maxeler ecosystem, we see it suitable for a number

of special-purpose computing [10, 9] and selected HPC scenarios, but not as major driver of FPGA adoption in general-purpose computing, were we see OpenCL as more familiar and more portable alternative.

With a focus on structured and productive offloading, this project intentionally left out manual customizations on the algorithmic level, or of data types and operations. Irrespective of these considerations, a very attractive subject of future work is the customization of a vector overlay to more efficiently execute the offloaded kernels. For example when not all vector registers are used, their number could be reduced, freeing resources towards larger vector registers. Also, after analysis of the used vector operators, the functional units of the vector lanes can be customized, possibly even as heterogeneous vector lanes as proposed in [282]. Other customizations require a more active codesign of kernel and overlay architecture. For example a local memory for each vector lane, as included in VIPERS [284, 285] may significantly boost the performance of the lookup operation of scaling kernel described in Subsection 4.5.1. Similarly the vector registers could be used as local buffers for the two difference kernels from the same subsection, requiring a non-straight-forward ISA extension before being targeted by manual assembly code or compilation.

CHAPTER 5

Compilation and Runtime Techniques for FPGA Accelerators

At the core of this chapter, we address the challenges of productive tool flows targeting instruction-programmable overlay architectures, using the example of the vector coprocessor implemented on the FPGAs of a Convey HC-1. In [4], we encountered shortcomings of the compiler shipped with the Vector Personality in terms of supported code constructs and in terms of performance of generated coprocessor code. We have developed a compilation flow that addresses these shortcomings and additionally provides fully automatic offloading without being guided by pragma annotations.

Thus, as high-level contribution, this work shows that highly productive compilation is possible for instruction-programmable overlay architectures, where encountered shortcomings of existing solutions are not inherent to the approach, but a matter of engineering resources. Through fully automatic detection of suitable code segments, along with highly productive offloading decisions at runtime, the initial effort to create an FPGA accelerated variant of some application is minimal, similar to using accelerated variants of commonly used libraries. Since our tool flow works on compiled code in LLVM[1] intermediate representation (IR) format, it is particularly suited for OTF computing scenarios as introduced in Subsection 2.3.3, where source code may not be available at the data center that tries to execute the workload as efficiently as possible.

As technical contributions of this chapter, we focused on code generation for outer-loop vectorization and on support for pointer-based addressing of data structures, which were both not possible with the previously existing toolflow. With a set of benchmark kernels that systematically varies these properties, we quantify, beyond the value of added functionality, the performance impact of these features.

Most of this chapter has been presented and published at ARC 2014 [7]. Beside Tobias Kenter as lead researcher, Gavin Vaz greatly contributed to the implementation and many design choices. Joint follow-up research with Gavin Vaz and Heinrich Riebler around improved solutions for and broader potential of deferring offloading decisions to application

[1] `http://llvm.org/`

runtime has been presented and published at ReConfig 2014 [11] (Received Best Paper Award) and in a much extended version in [12]. Complementary joint research, lead by Achim Lösch and Tobias Beisel, investigates runtime management of heterogeneous resources on a system level in scenarios with several competing tasks.

In the remainder of this chapter, we firstly in Section 5.1 present the motivation and goals of the presented work in more detail. Section 5.2 presents the approach and implementation ideas of our solution. In Section 5.3, we discuss the design of our test suite that we use for the evaluation in Section 5.4. Section 5.5 presents related work specific to this chapter. In Sections 5.6 and 5.7, we give a summary of the joint research that builds upon respectively complements the ideas from this chapter, before we conclude in Section 5.8.

5.1 Motivation

The results from Chapter 4 have not only demonstrated that instruction-programmable overlay architectures with wide vector instruction sets can be used to achieve speedups over CPU implementations, but also for the first time quantified the involved overheads in a general-purpose computing scenario. In Chapter 3, we have argued based on observations from other target architectures, that such overheads can be acceptable if they go along with sufficient gains in productivity. In our approach from Chapter 4, one aspect of productivity, the execution times of design tools, was clearly superior to the alternative approach that involves lengthy synthesis processes. However, with regard to required developer expertise and effort, the value proposition of the vector coprocessor design was much weaker due to the chosen low level assembly implementation. As discussed in Subsection 3.2.1, academic projects on vector processor overlays on FPGAs are either programmed on the same abstraction level [282, 284, 285], or slightly above with C macros that generate assembly instructions [56, 228].

The vectorizing Convey Compiler shipped along with the Vector Personality promised a much more productive design entry with compilation from C/C++ source code that only needs to be annotated with pragmas to guide the offloading and vectorization process. We built upon this compiler in our first work on stereo-matching [4] and our expectations were deeply disappointed. In order to vectorize promising loop nests, significant refactoring and annotation efforts were required. Some transformation steps, for example the consolidation of multi-dimensional data structures into flat arrays, were familiar from other acceleration and offloading paradigms and reproducible improved the chances for successful vectorization. For other design decisions, like the order of loops in loop nests or the computation of array indexes, the impact on vectorization was more erratic and required to try out many different variants, often along with significant code-bloat, before finding a code representation suitable for compiler vectorization. Finally, for the generated vector code, performance was far below expectations, often exhibiting slowdowns compared to pure host execution. A recurring pattern we observed during our analysis was, that the compiler only performed inner-loop vectorization, which often lead to data accesses in non-favorable stride patterns and also missed data reuse opportunities.

Given this experience and considering the very limited tool support for other academic vector overlays on FPGAs, there are two possible explanations. Either, automatic vectorization is inherently too hard to cover more than just a few basic computation patterns in canonical representation, or the available compilers are just not suitably engineered to handle the more complex computation patterns with dependencies, indirect indexing and multi-dimensional data structures beyond flat arrays. Depending on the correct answer to this choice, instruction-programmable overlays with vector execution units are either disqualified as productive path towards FPGA computing in general-purpose scenarios, or can still fill this role. Based on the observations during manual design of stereo-matching kernels and considering that vector supercomputers were fairly successful for a decade, we assumed the latter and for this chapter set out to demonstrate this. We aimed to design a compilation flow that

1. functionally supports a wide range of dependency and indexing patterns and flexible multi-dimensional data structures.

2. generates well-performing vector code that is not limited to inner-loop vectorization and smartly reuses data in its vector registers.

3. requires no refactoring or user guidance through pragma annotations.

5.2 Approach

We decided to build our tool flow upon the LLVM compiler infrastructure. It starts with binary applications in the form of LLVM IR. LLVM provides front-ends to generate IR from several source languages, which gives access to many active developers and legacy applications in the general-purpose domain and beyond. Even though IR is no machine code, it is a binary format in which applications can be distributed or it can be generated from machine code [22]. Thus, the toolflow can even be applied in OTF scenarios where source code is not available [110]. Beyond this very wide base of design entries, the LLVM infrastructure provides us with an extensive set of code analysis methods, encapsulated in so-called passes, on which we build our methods for identifying hotspots to move to the coprocessor and to perform suitable and valid vectorization.

In the remainder of this section, we first present our overall toolflow to generate heterogeneous executables for CPU and coprocessor. After this general overview, we discuss in more detail the extraction of code parts for execution on the coprocessor, the actual vectorization and our approach to runtime checks to guide the execution and data movement.

5.2.1 Toolflow for Heterogeneous Executables

Since we use and extend the LLVM compiler infrastructure in this project, we use its terminology. In particular, a module denotes the top level compilation unit, e.g. an entire program or a library that will be linked with the main executable later. A module contains a set of functions which consist of basic blocks. Control flow between basic blocks is denoted

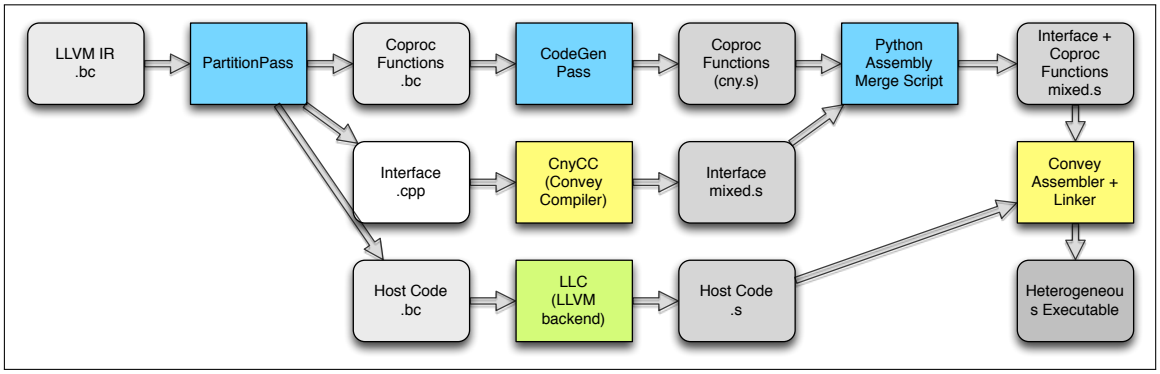

Figure 5.1: Toolflow for generating heterogeneous binaries for execution on Convey HC-1 coprocessor and host. Blue: our implementation; yellow: Convey Compiler infrastructure; green: LLVM infrastructure

by edges. Figure 5.1 depicts our overall toolflow for generating heterogeneous binaries for execution on the host CPU and coprocessor. The toolflow integrates some existing LLVM and Convey Compiler tools, but most of the partitioning-related aspects, as well as the vectorization and coprocessor code generation are new contributions for this work.

We start with LLVM IR code, which we generated for our tests in Section 5.4 with the Clang compiler frontend. In our *PartitionPass*, we then split the module into code that is to remain on host and code that is to be executed on the coprocessor. The details of this phase will be presented in Subsection 5.2.2. The *PartitionPass* also includes the planning of a vectorization strategy described in Subsection 5.2.3 and the inclusion of runtime decisions as detailed in Subsection 5.2.4. The modified host code is then translated by the LLVM backend to x86 assembly code. Note that, where applicable, this will generate short vector instructions for the host CPU's SIMD units.

For generating the interface between the host and coprocessor code, we use the Convey Compiler to match Convey's calling conventions and to avoid reimplementing that functionality. For that purpose, the *PartitionPass* additionally emits a .cpp file containing stubs of all the functions we want to implement on the coprocessor along with their signature of arguments. We also generate pragmas indicating to the Convey Compiler that those functions are to be executed on the coprocessor. The Convey Compiler then generates an x86 function entry, which contains a runtime check for availability of the coprocessor and puts all arguments properly on the coprocessor stack. Then control is handed over to the coprocessor entry of this function, where arguments are loaded from the stack into coprocessor registers.

From the code extracted for the coprocessor, we generate vectorized coprocessor assembly code in our *CodeGenPass* following the vectorization strategy determined by the *PartitionPass*. With the help of a Python script we then merge this code with the headers including function arguments of the function stubs compiled by the Convey Compiler. Finally we assemble and link the generated assembly and object files, again using the Convey Compiler tools.

5.2.2 Code Extraction

We want to identify parts of the code that can be executed on the coprocessor and are likely to yield a speedup. This subsection focuses on the feasibility whereas the performance depends on the outcome of the subsequently described steps.

On our platform, the control flow between host CPU and coprocessor is based on function calls. The only way to transfer control from the CPU to the coprocessor is to call a function that is compiled for the latter; the only regular way to transfer control back is to return from the called function. The coprocessor may call other coprocessor functions but it cannot call functions on the host. The following process identifies coprocessor suitable code regions in two phases, before the actual extraction starts.

In the first phase, we identify all function calls to libraries on the host CPU, e.g. I/O, as direct incompatibilities. The basic blocks containing these calls cannot be moved to the coprocessor, except for a few selected functions, for which we can generate coprocessor code directly, e.g. a `std::min()` with appropriate data types can be directly translated into assembly functions later. In the second phase, we search for indirect incompatibilities, where function calls inside the compilation module point to functions that need to be at least partially executed on the host. We repeat this second phase until no new incompatibilities are detected. The outcome of these two phases are functions that can be entirely moved to the coprocessor and functions that are only partially coprocessor feasible.

```
1  //Integral horizontal sums
2  for(int y=0; y<HEIGHT; y++)
3    sum[y][0] = in[y][0];
4    for(int x=1; x<WIDTH; x++)
5      sum[y][x] = sum[y][x-1] + in[y][x];
6  writeIntermediateResult(sum); // call with IO
7  //Horizontal differences
8  for(int y=0; y<HEIGHT; y++)
9    for(int x=0; x<WIDTH; x++)
10     right = rightArms[y][x];
11     left = leftArms[y][x];
12     diff[y][x] = sum[y][x+right];
13     if(left > 0)
14       diff[y][x] = diff[y][x] - sum[y][x-left-1];
```

Listing 5.1: Example for loops that can be extracted for coprocessor execution. Note: for a more visible representation of the dependencies, we chose a two-dimensional array indexing in contrast to Listing 4.6.

For those latter functions, we want to extract the basic blocks that can be executed on the coprocessor into new functions. For this extraction to be possible, a set of blocks must have a single entry edge and a single exit edge [146]. Some sets of basic blocks may not have this property, but can be transformed to satisfy it. In particular this is the case if they have a single basic block as target for all entry edges and a single basic block as

source for all exiting edges. As such, all loops in any nesting level have this property. As speedups are only expected from vectorizing loops, we restrict our toolflow to extract only sets of basic blocks that form a loop. We proceed from outer to inner nested loops, so if all basic blocks of an outer loop are marked as coprocessor feasible, the outer loop gets extracted, otherwise inner loops are tested in the order of their nesting level until we reach the innermost loop. We use LLVM's refactoring capabilities to perform this extraction after a suitable loop is detected.

The implementation variant of vertical differences in Listing 5.1 shows a simple code example where this function splitting is required. After the first vectorizable loop nest, some intermediate result is written to a file, before a second vectorizable loop nest follows. Our toolflow will extract the two loop nests into two new coprocessor functions, leaving the calls to those functions along with the other call inside the original function on the host. Note that we chose the source code listing just for illustration purposes, whereas internally our tools operate on LLVM IR.

5.2.3 Vectorization

The vectorization phase checks for two important conditions on each loop nest level. Firstly, dependencies between loop iterations are detected, which would prevent vectorization of this loop. The example from Listing 5.1 computes integral horizontal sums and differences respectively, thus the inner loop from the first loop nest has a dependency, leaving only the outer loop for vectorization. Secondly, the dimensions of the loops are checked, whether they permit any speedup. As heuristic, we use an iteration count of 100, when plain array data structures are detected and 500, when following pointers to inner dimension, as threshold below which vectorization often isn't sufficient to allow speedups on the coprocessor. When the iteration count of loops is constant, like indicated by capitalized loop bounds in our example, this decision can be made at compile time. However, often those counts can only be determined at runtime, which will be covered in Subsection 5.2.4.

When the conditions for vectorization specify, that only an outer loop can be vectorized, many compilers, including the Convey Compiler will try to interchange the loop nests and afterwards vectorize the inner loop. Depending on the compute and data access pattern, this may be inefficient, or infeasible, e.g. if the loops are not perfectly nested. Therefore we prefer to vectorize the outer loop directly.

```
1 for(int y=0; y<HEIGHT; y+=VL_max)
2   VL = min(VL_max, HEIGHT-y)
3   out[y:y+VL][0] = in[y:y+VL][0];
4   for(int x=1; x<WIDTH; x+=VL_max)
5     out[y:y+VL][x] = out[y:y+VL][x-1] + in[y:y+VL][x];
```

Listing 5.2: Vectorized loop nest from horizontal integral sums. Note: for a more visible representation of the vectors, we chose a two-dimensional array indexing in contrast to Listing 4.6.

Listing 5.2 illustrates the outer-loop vectorization of the first loop nest from Listing 5.1, using the array index notation from Cilk Plus[2], where a `[a:b]` statement indicates, that the elements from `a` to `b` will be processed in parallel. The outer loop is strip-mined. It now increments by the size of the vector registers `VL_max`. The actually used vector size `VL` is computed in line 2, because in the last iteration, there will often be less than `VL_max` elements left.

Our toolflow does not actually produce code like shown in this listing, but rather the *PartitionPass* plans vectorization and marks the identified loops for vectorization. When mapping the LLVM IR instructions to Convey coprocessor assembly code, the *CodeGenPass* then replaces all instructions involving the induction variable of the vectorized loop, in this case `y`, by corresponding vector instructions. This can in turn require to vectorize further instructions and variables, even if they are scalars independent of `y`. We support this scalar expansion, but no vector code generation of reduction operations.

In this simple example, loop exchange with inner-loop vectorization would easily be possible as well. However we can note, that the computation of `VL` with outer-loop vectorization is executed only $\frac{HEIGHT}{VL_{max}}$ times, whereas after loop interchange and inner-loop vectorization it would take place in the inner loop $WIDTH * \frac{HEIGHT}{VL_{max}}$ times. Additional benefits can be exploited when loop-invariant instructions from the inner loop can be moved to the outer loop. For example an address calculation or pointer load for an outer dimension of an array, like `out[y:y+VL]` in Listing 5.2 can be moved to the outer loop, which may not be possible after a loop interchange.

Note, that the pattern of vectorized memory operations is independent of the decision between outer-loop vectorization and inner-loop vectorization after loop interchange. In this example, assuming C-like row-major order, vectorization requires strided or indexed loads, which impose an overhead compared to continuous loads. When manually optimizing an application for vectorization, adapting the data layout, also combined with tiling, can be a major source of speedups. However for our automated acceleration approach, we leave the data layout unchanged.

5.2.4 Runtime Decisions

When the iteration space of a loop nest is known at compile time and promises speedups according to our heuristic threshold, we statically replace the execution on the host with execution on the coprocessor. This method is sufficient for the experiments presented in Section 5.4. However, often the iteration space depends on concrete input data to an application or on unknown function arguments when accelerating a library. In many of those cases, the iteration space can be determined at runtime of the program at the entry of the actual loop. In this case, we generate code to compute the iteration space before executing the actual loop, using LLVM's ScalarEvolution analysis. Then we add a comparison instruction to compare this value to the threshold for coprocessor execution. If the threshold is not met, a branch instruction will point to the original entry of the loop

[2] https://www.cilkplus.org/

on the host, otherwise to a new basic block where we generate data movement statements and a call to the according coprocessor function. If the iteration space cannot be computed at this point, e.g. when following a linked list, execution will remain on the host.

For achieving best performance on the NUMA architecture of our platform, data should be migrated to the physical memory location where it is most frequently accessed. Therefore we insert calls to Convey's data movement API to transfer data to coprocessor memory, before transferring control to the coprocessor. For these data movement statements, we need the data space of the accessed data structures. Similar to the iteration space, it can either be statically computed at compile time, dynamically at runtime before execution of the loop or it is uncomputable at this point. If it is computable, we add the according data movement statements, either with static size arguments or with runtime computed size arguments. After the coprocessor function execution, similar statements could move the data back to host memory. However, we would need to analyze the further control flow of the application to determine whether the next intensive data access will happen on the host or the coprocessor. Currently we don't support this, so we optimistically assume that typically the runtime relevant code sections will be executed on the coprocessor and leave the migrated data in coprocessor memory. Thus, subsequent coprocessor loops working on the same data will still have calls to migrate data to coprocessor memory, but will need very little time because no data actually needs to be moved.

5.3 Experimental Setup

In order to assess the impact of different dependency patterns, vectorization strategies and data layouts systematically, we designed a synthetic loop test suite. As base, we used the fundamental compute patterns, which we observed during our practical work with the vector coprocessor overlay of the Convey HC-1, in [4] and Chapter 4. We started off with three base patterns of nested loops.

1. A simple loop carried dependency in one loop nest, realized through an array offset of -1. Variants of this basic pattern, as presented in Listing 5.3 show up with the horizontal and vertical sum kernels and are one incarnation of the *Regular Streaming* kernels identified in Section 4.5.4.

```
1 for(int i=0; i<HEIGHT; i++)
2   for(int j=1; j<WIDTH; j++)
3     arr[i][j] = arr[i][j] + arr[i][j-1] * scalar;
```

Listing 5.3: Basis pattern with simple loop carried dependency.

2. A simple loop carried dependency in one loop nest, realized through an array offset of -1, that is combined with access to another array with different iteration space. This base pattern as presented in Listing 5.4 exhibits simplified features of the scanline kernels from Subsection 4.5.

```
1 for(int i=0; i<HEIGHT; i++)
2   for(int j=1; j<WIDTH; j++)
3     arr[i][j] = arr[i][j] + arr[i][j-1] * vec[j];
```

Listing 5.4: Extended pattern with loop carried dependency.

3. No loop carried dependencies, but indirect indexing in one loop nest, as presented in Listing 5.5. This corresponds to the group of *Irregular Index Offsets* with the difference kernels of the aggregation phase.

```
1 for(int i=0; i<HEIGHT; i++)
2   for(int j=0; j<WIDTH; j++)
3     arr1[i][j] = arr1[i][j] + arr2[i][j + vec[j]];
```

Listing 5.5: Basis pattern with irregular index offsets.

Building upon these three base patterns, we systematically generated additional variants.

1. We varied the nesting level of the loop nest between two and three. Two loop nests correspond to kernels that work on individual images or slices of a 3D cost volume, whereas three loop nests work on an entire 3D cost volume.

2. We varied the layout of the multidimensional data structures. Multi-dimensional data structures are either put into a flat *Array* or are accessed by following a *Pointer* for every dimension to a dynamically allocated memory location. Note that for the implementation in Chapter 4, only flat arrays are used, following a general design principle for external acceleration, to offload computations and data in as large chunks as possible.

3. We further varied on which loop nest's induction variable the loop carried dependency respectively the indirect indexing depends. Thus, for the loops with dependencies, this variation enforces either direct inner-loop vectorization (denoted as *Inner*) respectively direct outer-loop vectorization (denoted as *Outer*), or requires active loop-interchange. Straight-forward implementations of the summation and scanline kernels from Subsection 4.5 exhibit such pairs of opposite dependencies between the respective horizontal and vertical variants, however it is at the programmers discretion to interchange the loop nests. For the loops with indirect indexing without dependencies, we created a third variant with offsets in both loop dimensions as motivated for example by a census transformation that remained on the host in Chapter 4 as result of the profiling phase.

4. Finally, for each of these loops, we generate one data layout which is *Favorable* for vectorization by enabling continuous vector loads and one *Transposed* layout, which requires strided or indexed vector loads. In the stereo-matching algorithm of Chapter 4, the same data structures are accessed in two orthogonal directions and

thus, independent from loop orders decided by designers or compiler optimizations, form the pair of horizontal and vertical steps with dependencies, necessarily one step accesses a favorable layout and the other a transposed layout. For the loops without dependencies, we didn't generate transposed layouts, since the indexing variations from the previous step already cover the risk of working on a transposed layout because of limited knowledge. In the absence of dependencies, there is no further reason beyond limited knowledge to iterate over data in a transposed layout.

For the two basis patterns with loop carried dependencies, these four variations result in a total $2 \times 2 \times 2 \times 2 \times 2 = 32$ loop nests generated. For the basis pattern without dependency, the only the first three variations are applied, with three variants for the dimension in which indirect indexing points. This results in $1 \times 2 \times 2 \times 3 = 12$ variants. Additionally we evaluate two simple test patterns for the generation of mask instructions, which run on linear loops without dependencies.

We integrate all these loop patterns into a single test application, where all required data structures are allocated and filled with reproducible sequences of random data. Subsequently, all loop patterns are executed with 5000 iterations for each of the two inner loop nests and five repetitions through an additional outer loop for the 3D patterns. In the presented evaluation, all trip counts can be statically determined by the compiler. Thus, each non-vectorized loop executes for 5000 sequential iterations, whereas vectorized loops on the coprocessor need only five iterations, where the final iteration is using less than the maximum possible width of 1024 vector elements in order to demonstrate correct functionality of this feature. For each loop, the execution time is determined by calling a timer function before loop invocation and after loop termination. These timing measurements serve as natural border for the offloading mechanism. After the time measurement, from the output arrays of each loop, a checksum is computed.

5.4 Evaluation

We evaluate our approach by comparing for each loop pattern the performance achieved after applying our toolflow and running the heterogeneous executable on host CPU and coprocessor together, to the baseline performance when compiling to pure host code with the Clang backend and executing only on the host CPU. In terms of tool productivity, our entire toolflow just adds a few seconds to the default Clang compilation time. A comparison of the checksums demonstrates the correct functionality of the entire test setup.

In our first experiment, all measurements are performed with data already present in local memory of the target platform, in host memory for the baseline measurements and in coprocessor memory for the offloaded loops. This setup is close to the practical performance if our optimistic data movement strategy works out and corresponds to the raw kernel performance as analyzed in Subsection 4.7.3.

Table 5.1 summarizes the geometric mean speedups of our toolflow compared to pure host execution, for all loop nests together and in individual groups. According to the systematic from Section 5.3, we group our total of 46 benchmark loops into five times two

Table 5.1: Observed speedup as geometric means for different groups of loops. Subscripts indicate the size of each group. Note that identical loops show up in the Inner/Outer and Favorable/Transposed column pairs.

	Inner	*Outer*	*Favorable*	*Transposed*	*Independent*	Geomean
Array	6.51_{08}	6.67_{08}	13.10_{08}	3.31_{08}	2.05_{08}	4.46_{24}
Pointer	2.00_{08}	2.01_{08}	2.62_{08}	1.52_{08}	0.88_{06}	1.52_{22}
Geomean	3.60_{16}	3.66_{16}	5.86_{16}	2.25_{16}	1.35_{14}	2.61_{46}

groups in Table 5.1, with subscripts indicating the size of each group. The 32 loops with dependencies can be found in their corresponding line of either column *Inner* or column *Outer* and in order to enable a different point of view they are contained again either in column *Favorable* or in column *Transposed*. Additionally, there are ten *Independent Array* loops (including two vector mask patterns) and eight *Independent Pointer* loops, where the vectorization target and utilization of data layout solely depends on compiler decisions. Like in Chapter 4, we use geometric means within each group and beyond because of the scale invariance of this metric. Since geometric means weight positive outliers less, the values are a bit lower than published in [7], where arithmetic means are used. We summarize the geometric means of groups by columns and rows, with the overall mean in the lower right corner.

We see that the automatic compilation flow is not only functional, but is also able to achieve speedups for all but one of the investigated groups of loop patterns. The geometric mean of speedups of all 46 kernels is 2.6x and the eight kernels in the best performing group of flat arrays with favorable data layout achieve a speedup ratio of more than one order of magnitude.

Investigating the influencing factors individually, we firstly see that *Array* data structures allow much higher speedups then *Pointer* data structures, around 3x higher in many groups. To ensure correct functionality, we make no assumptions about the actual data layout of dynamically allocated partial arrays and thus need to load many different base addresses that can be computed as offsets in the loops with flat arrays. There may be some improvement potential through more clever reuse of once loaded base addresses, but a gap to the *Array* groups will definitely remain unless additional knowledge about the allocation process can be gathered by the compiler, from the developer, or dynamically at runtime. Outer-loop vectorization (*Outer*), that we implemented as alternative to loop interchanges with the goal of inner-loop vectorization (*Inner*) yields slightly higher speedups for *Array* data structures. Based on earlier experiments with hand-written code, we had hoped for a higher impact, but were not able to separate out the effects of data layout. The systematic variation of data layout between *Favorable* for vectorization and *Transposed* to it reveals that the impact of this factor by far dominates the vectorization strategy, particularly in *Array* loops, were a favorable layout enables around 4x higher speedups than a transposed one. Nevertheless, even all groups with transposed layouts were able to gain some speedups. Only in the group of *Pointer* data structures with indirect indexing the

high number of irregular memory operations causes a slight slowdown compared to the host CPU. In these cases, the vector overlay can no longer compensate its clock frequency disadvantage and lack of a generic cache hierarchy with increased parallelism.

5.4.1 Comparison to Hand-Written Kernels of Chapter 4

When comparing these results with the results in Table 4.6, were kernels with hand-written assembly code run actual stereo-matching tasks on the same platform, we first need to summarize common features and differences. Both speedup numbers refer to raw kernel execution times with data present at the coprocessor. The input sizes of the synthetic tests are larger in the two inner dimensions, but much smaller in the third dimension of the 3D kernels. This has an impact on the comparison, but reduces the effect of vector partitioning, that is used in the hand-written kernel but not generated by our compilation flow. All hand-written stereo-matching kernels use flat arrays and fall into the first row of Table 5.1. The summation and directed scanline kernels fall into the *Outer* group, with a forced variation between *Favorable* and *Transposed* data layouts. The other kernels fall into the *Independent* group, even though the sumScanlines requires neither indirect indexing nor select operations and hence would performance-wise be best represented by the *Favorable* group.

The kernel-ratio of 5.80 in Table 4.6 is computed with reference to the host CPU of the Maxeler platform, there denoted as *CPU1* and therefore needs to be divided by the kernel-ratio of *CPU2* to obtain results comparable to those in Table 5.1. Thus, a kernel-ratio of 14.50 would represent the hand-written kernels for all input sizes from Chapter 4 as most comparable number to the results from this chapter. It is much higher than the speedups of the group with *Array* and *Outer* kernel variants, to which six of its ten kernels belong. We attribute this difference mostly to three reasons. Firstly, knowledge of the application allowed higher data-reuse in the hand-written kernels than the compiler could prove. Secondly, the actual computational intensity of the scanline kernels is higher than covered by the basic data access pattern in the synthetic test suite. Thirdly, manual, measurement-driven optimization of the instruction sequences of some hand-written kernels yielded additional speedups for which we were not able to find any design rules applicable to an automatic code generation.

5.4.2 Further Experiments

As shortcomings of the shipped Convey Compiler motivated this work, we compared our results to the ones generated with the latest version of this compiler. When testing it on the same benchmark in its fully unguided mode, it produces wrong results, probably due to some unsafe optimizations. However, according to the documentation, the unguided mode is mainly intended for finding possible vectorization candidates. Hence, we next added the applicable pragmas about data layouts, vectorization goals and hints about dependencies or their absence in each loop nest. After experimenting with the most suitable compiler options, we were finally able to vectorize and offload 20 of the 24 loops with array data structures, but none of the pointer variants. This outcome is a notable progress over the

older internal vectorizer of the Convey Compiler that motivated our project when we used it in [4] and that can only vectorize four of the array loop patterns without further manual extraction of index computation and manual loop-interchanges. However, our contribution both significantly increases the functionality and reduces the effort for targeting a vector coprocessor overlay compared to all tested variants of the Convey Compiler.

When comparing the runtimes after applying our toolflow to those from the loops successfully vectorized with the Convey Compiler, in nine of those 20 loops, our generated kernels are faster between $2.88x$ and $10.05x$, mostly because our more direct vectorization approach allows higher data reuse. For seven loops, the runtimes are almost identical (speedups of $1.00x$ to $1.06x$). In four examples, we have slowdowns of 0.44 to 0.49, because our compiler misses a data reuse opportunity by a loop interchange the Convey Compiler performs. Thus, our contribution reveals considerable additional speedup potential for automatically generated code for vector coprocessor overlays, but is not optimal for all investigated patterns.

In practical applications as in Chapter 4, there is a mix of required data transfers between host and coprocessor and reuse of data residing in coprocessor memory. As complementing experiment to our optimistic scenario from Table 5.1, we measured the performance impact when all used data needs to be migrated to the coprocessor prior to each actual invocation of a loop vectorized with our toolflow. The geometric mean of all speedups in this scenario is 0.85, which corresponds to a slight slowdown. Looking at individual loops, we found 21 loops with speedups and 25 loops with slowdowns, in both categories between a few percent and around one order of magnitude. Reusing the same data from coprocessor for further loop invocations reduces these overheads until the speedups asymptomatically approach the results in Table 5.1.

5.5 Related Work

In this section, we give an overview of related work on compilation for vector architectures. This work was motivated by serious gaps between what can be achieved using the Convey vectorizing compilers and what is possible by making best use of the vector instruction set. Maleki et al. [172] have investigated similar problems for compilers targeting vector extensions for current general-purpose CPUs. Our work underlines that overall such shortcomings can be addressed, but the details can be challenging.

The foundations of automated loop vectorization driven by data dependency analysis were established during the era of vector supercomputers in the '80s, first by Allen and Kennedy [16]. In their source-to-source compilation system they apply loop interchange and then vectorize inner loops or entire loops nests that are fully vectorizable from the innermost loop on. Later on, the vectorizing Fortran compiler by Scarborough et al. [221] also featured direct outer-loop vectorization, like Ngo's [200] compiler framework integrated into the "Cray Fortran-90 compiling system". More recently, outer-loop vectorization has also gained interest for SIMD instruction sets with short vector units [202] as present in modern GPPs.

Other test suites for automatic vectorization, like the one from the GCC auto-vectorization project [3] and PolyBench [212] contain some additional code patterns, but lack the systematic variations of our test suite and thus don't allow to isolate effects of the vectorization strategy from the data layouts.

5.6 Excursion to Offloading Decisions at Runtime

In this work, we have introduced, but not evaluated the concept of embedding offloading decisions into application code, where they often have access to the runtime variables that determine the iteration counts of loops and loop nests. The joint follow-up research in [11] and [12] improves the implementation of offloading decisions at runtime and demonstrates their effectiveness in scenarios where an actual mix of input data sizes requires dynamic decisions for or against offloading at runtime. For this evaluation, we furthermore improve the selection of data to be transferred between host and coprocessor memory and depart from the optimistic approach of lazily leaving data in coprocessor memory.

In [11, 12], we also investigate the potential of offloading decisions at runtime beyond the compilation for this concrete FPGA overlay target. Firstly, we analyze all loops from the common benchmarks SPEC CPU2006 [119], MiBench [106] and MediaBench [165] to find out, whether the trip count to guide an offloading decision can be determined at compile time, or later at runtime when the loop is entered, or cannot be determined with the means of the LLVM compiler framework at all. It turns out that the trip counts of 40% of all loops identified in the code or 26% of all loops that are executed at least once can be determined at runtime, but not before. These loops account for 53% of all cumulative loop iterations in all benchmarks. These numbers show, together with more detailed per-benchmark analysis in [12], that offloading decisions that are automatically inserted into program code and take deterministic decisions based on actual runtime data, can have a great impact beyond the case-study within this chapter.

Compared to decisions based on expert knowledge from the application developer, or based on profiling results, runtime decisions require much less effort at compile time and thus are much more productive as postulated in Section 3.1. Conceptually, they however incur a performance overhead compared to statically taken decisions at compile time. In order to assess this overhead, as second experiment we insert such runtime decisions at the entry blocks of every applicable loop of the SPEC CPU2006 benchmark, without actually generating corresponding accelerator code and ensuring with an according threshold parameter, that all execution actually remains on the host CPU. With an impact on overall benchmark execution time of less than 1%, we found the performance trade-off of such runtime decisions to be negligible.

As third new contribution beyond [5], we introduce and analyze in [12] the decision slack as the distance between the earliest program position or point in time, where a runtime decision can be computed based on all required input data, and the program position or point in time, when the potentially to-be-offloaded loop starts. We found that in the

[3] `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`

SPEC CPU2006 benchmarks, 88% of all loops applicable for runtime decisions exhibit such a decision slack. Evaluating in a conservative analysis only the shortest possible control flow path, we found the average decision slack in SPEC CPU2006 to span 80 instructions within 10 basic blocks. The resulting time span may be used for small configuration or data transfer tasks of an accelerator or for taking scheduling decisions on the system level like the ones outlined in the following section.

5.7 System-Level Scheduling and Task Migration

Up to now, we have within this chapter and in Chapter 4 only investigated how to optimize the performance of a single application at a time with exclusive usage of a heterogeneous system with host CPU(s) and an FPGA accelerator. This is similar to the scenario analyzed in [215] for the PCs market, where emphasis was given to one application running interactively in the foreground. However, particularly for the consolidated execution of cloud and OTF workloads in datacenters, at the same time the performance of many tasks may need to be optimized. These tasks then compete for host and accelerator resources, which is the perspective we jointly investigate in [8]. Instead of individual offloading decisions embedded into application code, here a system scheduler controls the assignment of tasks to host and accelerator resources.

In order to demonstrate the importance of this perspective, we present three technical contributions in [8], extending earlier work of Beisel et al. [32]. Firstly, we introduce a programming pattern around checkpoints within the kernels of our target applications. This allows us to interface different kernel implementations for multicore CPUs, GPUs and Maxeler FPGA accelerators with common functions to setup and retrieve intermediate application status and to trigger the next computation step. In contrast to the automated toolflow presented in this chapter, the checkpoint-centric programming pattern allows to voluntarily generate repeated decision points at which computation can be migrated between arbitrary resources. Secondly, we characterize the performance of different kernels with a range of different input dimensions on all three resources and transform the concrete measurement results into a more abstract affinity metric. This measurement-based affinity metric can be considered as the counterpart of the target architecture specific threshold used throughout this chapter. Thirdly, we present a heterogeneous scheduler that assigns tasks to their most suitable resource whenever possible, but also tries to fully utilize all resources in order to maximize the total task progress of the system. This can involve executing tasks on a non-optimal resource that would otherwise fall idle. When a new, better suited task for this resource arrives, or when the optimal resource of the running task becomes available, the scheduler can update an assignment decision and migrate a task at any checkpoint.

For the evaluation in [8], we generate different task sets with 32 to 72 tasks from the applications Gaussian blur, heat transfer simulation, Markov chain and correlation matrix with different input sizes. The sets contain for each resource some tasks that are most affine to it and the distribution of task execution times mimics large-scale datacenter workloads [187, 72]. For these task sets, we compare the total completion time with our

heterogeneous scheduler with task migration to two alternatives without migration. The first alternative, denoted as affinity-conserving, executes all tasks on the most suitable resource and thus may leave other resources idle, whereas the second alternative, denoted as workload-preserving, also assigns non-optimal tasks to otherwise idle resources and thus corresponds to the heterogeneous scheduler without migration capabilities.

It turns out that the second alternative on average does not deliver competitive performance because it sometimes causes long-running tasks to execute on a non-optimal resource with large slowdowns. The heterogeneous scheduler can recover from such situations and compared to the first alternative provide an average speedup of 7% through better resource utilization. This shows that offloading decisions at the system level can be superior to decisions on the application level, when the system has sufficient capabilities to correct non-beneficial decisions.

In the current implementation of offloading decisions embedded into application code as presented throughout this chapter, execution falls back to the host, when the coprocessor is not available at the decision point. In a multi-tasking scenario, this is conceptually similar to the workload-preserving alternative from the previous paragraph, but with less degrees of freedom. Considering the above results, it might be better to retry the coprocessor allocation after a brief waiting period, in particular when iteration counts well above the threshold hint towards high speedup potential.

5.8 Chapter Conclusion

In this chapter, we have presented an automated, unguided acceleration process for binary applications targeting a heterogeneous platform with a coprocessor realized with an instruction-programmable FPGA overlay. Our toolflow introduces decisions made at application runtime and beats existing pragma-based tools in versatility and in many cases in performance. This shows that with the help of overlays, acceleration with FPGAs can be achieved not only without the lengthy synthesis processes, but also without difficult and time-intensive application refactoring. Even though the presented design flow seems to involve additional performance overheads on top of the overlay-inherent ones analyzed in Chapter 4, with its extremely high productivity, it provides an interesting entry point for FPGA acceleration. The automation and the independence from source code availability can be particularly important in OTF computing scenarios. A possible next step is to exploit the fast acceleration process by moving it from compile time as presented here to the actual runtime of the program, e.g. by running the program in LLVM's just-in-time execution engine and then accelerating applications fully transparently to the user.

The presented thresholds for our offloading decisions work in our scenario, but are conceptually a rather rough heuristic. An interesting starting point for future work can be to either refine these thresholds through a learned model based on code-features as presented in [269] for OpenCL kernels, or to use auto-tuning techniques [255] to at the same time find good-performing kernel variants and characterize their performance for different input dimensions. Similar to the hand-written assembly code from Chapter 4, the generated code is scalable to overlay architectures with different vector lengths. Either of the two tech-

niques to automatically find more accurate thresholds would be particularly important for scaling overlay variants. As discussed in Sections 3.2 and 4.9, customization of the overlay architecture can be an approach to reduce the involved overheads. Some customizations proposed in Section 4.9 can easily been applied after kernel code generation, whereas others are challenging, because of the interdependence of the compilation flow and the overlay architecture.

CHAPTER 6

CPU-accelerator System Integration

In this chapter, we broaden the view from concrete systems and solutions to design options for future system architectures combining GPPs and reconfigurable accelerators including FPGAs. For this purpose, we not only depart from the constraints imposed by existing hardware platforms, like the Convey and Maxeler systems targeted in the two previous chapters, but also from the design methodology aiming for concrete solutions. Instead, we use an abstract model of computation that is necessarily less accurate, but that overcomes the limitations to concrete architecture features, parallelism types and sources of efficiency for FPGAs.

The high-level contributions of this chapter are on the one hand a new approach for fast and fully automated performance estimation of CPU-accelerator architectures and on the other hand high-level design insights for such architectures based on the estimation method. The technical contributions around an LLVM-based analysis tool flow combining static and dynamic code analysis have influenced the solutions presented in Chapter 5 and the joint work in [1].

The core of this chapter has been published in [3], extending the design space exploration and improving the partitioning method introduced in [2]. The presented work is based on the Intel funded project Multimodal Reconfigurable Processing Unit (MM-RPU) and beneath the main research conducted by Tobias Kenter contains ideas from Michael Kauschke, Marco Platzner and Christian Plessl, mainly on the architecture model.

In Section 6.1, we present the challenge of design space exploration for the integration of heterogeneous architectures that motivates our work in this chapter. In Section 6.2 we introduce the type of architectures we investigate. Section 6.3 contains our new performance estimation method and the partitioning approach used in our framework. In Section 6.4 we discuss selected results of our design space exploration. In Section 6.5 we compare our framework to related work, before giving a conclusion and outlook to future work in Section 6.6.

6.1 Motivation

The integration of accelerators with CPUs into heterogeneous systems is not only a technical challenge, but it also offers many design alternatives with regard to data transfers, communication and synchronization. In particular, the characteristics of a given interface have a major impact on the granularity of functions that can be offloaded to the accelerator, on feasible execution models, and on the achievable performance or performance or efficiency benefits. As outlined in Sections 2.2.3 and 2.2.4, both GPU and FPGA accelerators have entered the general-purpose computing domain as loosely coupled architectures with separate physical memory and memory space, but are more recently also integrated with CPUs into common memory spaces, cache hierarchies and SoC products (see Subsection 3.1.2). This tighter coupling has been driven by performance and efficiency considerations, but also to increase the productivity of application designers targeting accelerators.

However, as illustrated in the domain of GPU accelerators, where the HSA foundation tries to shape the integration process [240], the progress towards tightly coupled heterogeneous systems is slow. Additionally, the programming models and tools need time to follow architecture innovations before their benefits can be harvested beyond case studies. Of course, all previously possible coarse-grained offloading concepts still work with and often profit from tighter integration, but it takes time to make best use of new, more fine-grained offloading opportunities. This observation is even more applicable for heterogeneous systems with reconfigurable accelerators, due to the more numerous ways they can increase performance through different forms of parallelism and customization. In this chapter, we attempt to gain insights about the potential integration of reconfigurable accelerators independent of these limitations.

In this work we focus on a subclass of CPU-accelerator architectures, where the accelerator has both a direct low latency interface to the CPU and independent access to the memory hierarchy. We discuss this architecture and its motivation in Section 6.2. Performance estimation and design space exploration for this and other classes of CPU-accelerator architectures are challenging problems. Simulation is the most common approach to evaluate the architectural integration of reconfigurable accelerators before prototyping. For example, Garcia et al. [87, 88] rely on co-simulation to evaluate an architecture where CPU and accelerator work on the same memory hierarchy. While such a pure co-simulation approach provides some insight, its time-consuming design process often limits it to assume a specific interface and a hardware/software partitioning that is custom-tailored to the characteristics of this interface. The challenge for an automated design space exploration is that the specifications of the interface affect what parts of the application can be mapped to the accelerator during hardware/software partitioning. We consider this interdependency between interface and partitioning the reason why the systematic exploration of the design space for the architectural integration has so far not received significant attention in research.

In this chapter, we present a new approach for fast and fully automated performance estimation of CPU-accelerator architectures. By combining high-level analytical performance modeling, code analysis and profiling and automated hardware/software partitioning we

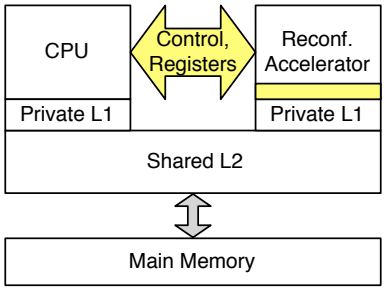

Figure 6.1: Reconfigurable accelerator with dual interface in an architecture with shared L2 and two private L1 caches. The two interface components (to adjacent general-purpose CPU and to memory hierarchy) are highlighted in yellow.

can estimate the achievable speedup for arbitrary applications executed on a wide range of CPU-accelerator architectures. The intended use of our method is to quickly identify the most promising areas of the large CPU-accelerator design space for subsequent in-depth analysis and design studies. Consequently, we emphasize modeling flexibility and speed of exploration rather than a high accuracy of the estimation method. The main benefit of our method is that it needs only the application source code or LLVM binary and does not require the user to extract any application-specific performance parameters by hand.

6.2 Proposed Architecture

In this section we present the general model for an integrated CPU-accelerator architecture that we investigate with our estimation and partitioning framework in this chapter. The focus of this work is on the interface of the architecture and its impact on the hardware/software partitioning. Therefore we intentionally leave the internal architecture of the reconfigurable accelerator relatively open to allow for flexible execution models adapted to the needs of different applications. The model conceptually covers FPGAs as fine-grained reconfigurable accelerators, as well as coarse-grained architectures, either implemented directly in hardware as introduced in Section 3.2.2 or as structurally programmable overlays on FPGAs as introduced in Section 3.2.3. We abstract the internal details of these architecture variations away in this work by using a general efficiency metric as presented in Section 6.3.1. With an explicitly spatial interpretation of compute resources, this estimation method does not cover instruction-programmable overlays. Instead, our model allows to map the customized datapaths of soft processors onto the reconfigurable overlay, whereas the remainder of the execution would remain on the GPP.

Our interface template is specifically designed to enable both such tight coupling of computations and more independent execution of entire loops or kernels on the accelerator. Subsection 3.2.2 introduces some proposed architectures that focus on only one of these two options. In our architecture concept, in order to achieve higher flexibility, the accelerator is embedded into the compute system through two interfaces, illustrated in yellow in Figure 6.1. First a direct low latency interface to an adjacent general-purpose CPU allows

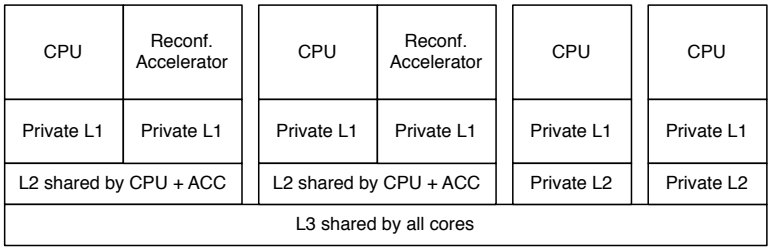

Figure 6.2: Possible integration of the proposed architecture into a multicore CPU.

fine grained interaction mainly on control level like activating a particular configuration of the accelerator, triggering its execution and synchronizing with its results. This may also include the exchange of small data entities like predicates, flags and small scalars from and to CPU registers. The second interface gives the accelerator access to the memory hierarchy independently from the CPU and can have different contact points to this hierarchy. In order to support virtual memory, the accelerator needs either a replicated memory management unit (MMU) or access to the CPU's MMU, possibly through the first interface element.

The dual interface simplifies the overall design, as the accelerator's memory interface may be tailored to the needs of the accelerator without compromising the CPU architecture. It comes at the possible cost of integrating shared caches, which we consider in our design space exploration in Section 6.4.2. The distinctive feature of this coupling with a dual interface is the support of a wide range of granularities of accelerated code parts, reaching from custom instructions, across kernel loops up to functions or threads. Although the proposed interface does not require a complete CPU redesign, it still allows for the acceleration of fine grained tasks and, thus, can also increase single-thread performance, an area where current CPU designs face diminishing returns.

The configuration in Figure 6.1 with private L1 caches for both CPU and FPGA and a shared L2 cache is one possible configuration of the memory hierarchy integration. It requires a coherency mechanism between the private caches. In our general interface template, the private caches on the accelerator side are optional and can either be omitted, which we investigate in this chapter, or replaced by other forms of local memory like scratchpad memory or buffers, which are not evaluated here. As a further degree of freedom, we vary the point, where private and shared caches are split, up to the extreme points where either no private or no shared caches exist.

There are various options to integrate our proposed architecture into a multicore system. Due to the proposed direct coupling of a CPU core and an accelerator, the most straightforward integration into a multicore system would be to pair individual CPU cores, either all or only a subset of the multicore CPU, each with a private accelerator. A possible configuration of such a multicore CPU with accelerators is illustrated in Figure 6.2. Depending on the accelerator's internal architecture, resource sharing between several accelerators on the same chip is an additional option. In our design space exploration in Section 6.4, we only

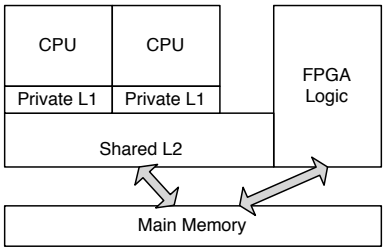

Figure 6.3: Illustration of Xilinx Zynq memory hierarchy.

evaluated single-threaded workloads and hence just investigate a single CPU-accelerator pair.

6.2.1 Relation to Existing Architectures

At the time of our original publications in [2, 3], the most closely related available systems had integrated FPGA accelerators in CPU sockets on dual or multi socket mainboards. Examples of such systems are the XtremeData In-Socket Accelerators [280], [279] with Altera FPGAs which communicate with the CPU via the Intel Front Side Bus or via HyperTransport respectively. Similarly, the Nallatech FPGA Socket Fillers [199] use the Intel Front Side Bus and contain Xilinx FPGAs. The Convey HC-1 targeted in Chapters 4 and 5 and illustrated in Figure 4.4 is another representative of this system class and has the distinctive feature of a coherent memory space realized with a sophisticated mix of software on the CPU side and hardware support on the host interface FPGA.

Meanwhile, first available products more directly exhibit individual aspects of our proposed architecture pattern. The Xilinx Zynq product line [214] and the Altera SoC product line [19] integrate a dual core ARM Cortex-A9 processor with FPGA fabric into a single SoC. These processors use out-of-order execution, speculation and superscalar execution and thus belong to the class of mobile processors that brought general-purpose computing characteristics into mobile devices, as outlined in Section 2.3.1. In Figure 6.3, we give a simplified illustration of the Zynq architecture in comparison to our architectural template. The FPGA logic has direct access to off-chip main memory through so-called high performance ports and to the shared L2 cache through the so-called Accelerator Coherency Port (ACP), and thus is similar to our hierarchy template without private L1 on the accelerator side. Additionally, not depicted in the figure, the Zynq architecture contains a scratchpad memory that is like the L2 cache accessible both by the CPU cores and the FPGA fabric. The FPGA of the Zynq has no fine-grained interface to a CPU core, but through its general purpose ports can for example trigger interrupts on the processor.

With the CAPI interface, more recently also the POWER8 architecture for high-end server processors provides hardware support for cache coherent accelerators [242, 246], which are however not integrated on the same chip. One supported accelerator class are PCIe attached FPGA boards. Figure 6.4 illustrates a simplified view on the architecture, where directly adjacent boxes are integrated on one chip and gray arrows indicate off-

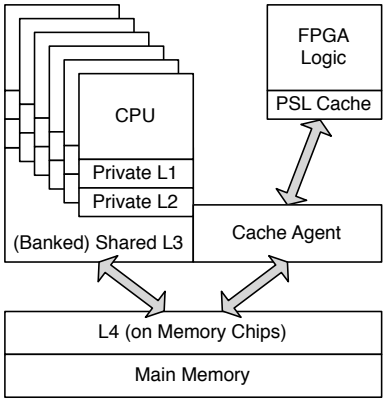

Figure 6.4: Illustration of POWER8 memory hierarchy with Coherent Accelerator Processor Interface (CAPI).

chip communication. On behalf of the external accelerator, a cache agent is attached to the common on-chip coherence and data interconnect. The much more complex memory hierarchy of this architecture compared to Zynq in particular also includes a private cache implemented in the FPGA's BRAMs.

While these architecture innovations focus on the integration of accelerators into memory hierarchies and don't exhibit the fine grained synchronization and register exchange of our first proposed interface component, there is also a project that can illustrate this aspect of our concept. The proposed and successfully prototyped VISC (no acronym) architecture [64] is purely a multicore processor architecture without reconfigurable accelerators. However, it proposes the close collaboration of several (in the first design two) CPU cores on a single task and software thread, just like in our architecture approach CPU and accelerator can jointly work on the same task and thread. Consequently, the designers of VISC focused on the tight coupling of a pair of cores and implemented a dual interface that enables the direct exchange of register values in a single cycle and also features a partitioned shared L1 cache. With a working prototype aiming at 2GHz clock frequency, this illustrates that the tight coupling proposed by our architecture is possible on the CPU side.

6.3 Method and Framework

In this section, we lay out the details of our performance estimation method and present our partitioning method, which is inspired by the work of Spacey et al. [241] (see also Section 6.5).

6.3.1 Estimation Model

In this section, we present the basic terms and definitions of our model as summarized in Table 6.1. The architecture-independent characterization of each benchmark depends

Table 6.1: Basic symbols of the model.

Instruction	I_k
Basic Block	B_k
CPU, Accelerator	`CPU`, `ACC`
Memory Hierarchy	$L1$, $L2$, $L3$, MEM
jth execution of instruction I_k	I_k^j

Table 6.2: Data obtained by profiling and static code analysis.

Execution count of instruction I_k, basic block B_k	$n(I_k)$, $n(B_k)$
Count of control flow from B_l to B_m	$n(B_l, B_m)$
Register value used or produced by B_l	$R_u(B_l)$, $R_w(B_l)$
Virtual cache level that executes a load/store	$V(I_k^j)$
Dependency pointer for a load/store	$P(I_k^j)$

on these symbols and gathers the values summarized in Table 6.2 by a combination of static code analysis and profiling with code instrumentation. A concrete incarnation of our architecture template is characterized by a set of architecture parameters, which are summarized in Table 6.3. The default parameters of our design space exploration are presented in Section 6.4 in Table 6.5. Based on the benchmark characterization and the architecture parameters, our framework computes a specific execution profile that adds up the values summarized in Table 6.4.

Like the tool flow presented in Chapter 5, our estimation framework is based on the LLVM compiler infrastructure [163]. The investigated software is compiled into LLVM assembly language, which is the intermediate code representation on which LLVM's analysis and optimization passes work. We model a program as a set of instructions $I = \{I_1, I_2, \ldots, I_{\texttt{nins}}\}$, and classify the instructions into memory dependent operations (e. g., load/store) and independent operations: $I_k \in \{\texttt{ld/st}, \texttt{op}\}$. The program structure groups the instructions into a set of basic blocks $B = \{B_1, B_2, \ldots, B_{\texttt{nblks}}\}$. The basic blocks form a control flow graph where an edge $B_l \rightarrow B_m$ denotes that block B_m might be executed directly after block B_l. The instructions of B_l use a set of register values $R_u(B_l)$ and produce a set of register values $R_w(B_l)$.

Table 6.3: Architecture parameters of the model.

Execution efficiency	$\epsilon(\texttt{CPU})$ and $\epsilon(\texttt{ACC})$
Communication latencies	λ_c, $\lambda_{r,\texttt{push}}$, $\lambda_{r,\texttt{pull}}$
Cache latencies	$\lambda_m(\texttt{L1}), \lambda_m(\texttt{L2}), \lambda_m(\texttt{L3})$, $\lambda_m(\texttt{MEM})$
Accelerator size	A

Table 6.4: Solution-specific data depending on architecture.

Mapping of basic block or instruction	$p(B_l)$, $p(I_k)$
Architectural cache level that executes a load/store	$v(I_k^j)$

The architecture comprises two processing units, the `CPU` and the accelerator (`ACC`), and a memory hierarchy consisting of `L1`, `L2` and optional `L3` caches, and main memory (`MEM`). The caches can be private for each processing unit, like the `L1` cache in Figure 6.1 or shared between both units like the `L2` cache in the same figure. We model the execution efficiencies $\epsilon(\texttt{CPU})$ and $\epsilon(\texttt{ACC})$ of the processing units through the average number of clock cycles spent per instruction (CPI), thus as inverse to the IPC metric mentioned in Chapter 2. The efficiency reflects on the one hand raw execution times of instructions and on the other hand parallel execution units and pipelining effects, which increase the throughput. We see this as a means to cover the various architectural opportunities discussed in Section 6.2, without making strict assumptions about the accelerators internal details. Thus, in contrast to the efficiency term introduced in Section 2.1.1, the execution efficiency ϵ in this model has no common denominator and is rather a consolidated performance metric. The model efficiencies can vary for different instruction classes, but lacking more accurate data for both processing units, in this work we use identical efficiency for all instructions except for two instruction classes. First, we assume that LLVM typecast instructions are implemented on the accelerator through wiring and thus are executed in zero execution time. Second, load/store instructions are also considered in a different way, because their execution time depends on the level in the memory hierarchy that they access. We describe the corresponding access latencies with $\lambda_m(\texttt{L1}), \lambda_m(\texttt{L2}), \lambda_m(\texttt{L3})$ and $\lambda_m(\texttt{MEM})$, expressed in clock cycles. When a processing unit requires data resident in the private cache of the other processing unit we include a latency penalty for writing back the data to the shared cache.

For communication between the CPU and the accelerator, we define λ_c as latency for transferring control between the processing units. This control latency also covers the efficiency losses that may occur, when the pipelining of instructions is reduced by control changes. Furthermore, we denote λ_r as latency for transferring a register value. Refining the register value transfer model, we foresee a push method with a low latency of $\lambda_{r,\texttt{push}}$ for actively sending a register value from one location to the other and a somewhat slower pull method with latency $\lambda_{r,\texttt{pull}}$ for requesting a register value from the other location and receiving it. Since the analysis of register dependencies is based on the code in LLVM intermediate representation, no register allocation has taken place, so all values are treated as available in an infinite register file after their first occurrence.

The partitioning process maps each basic block to either the CPU or the accelerator. We denote the mapping of block B_l as $p(B_l)$ with the two possible values $p(B_l) = \texttt{CPU}$ or $p(B_l) = \texttt{ACC}$. Obviously, the partitioning of basic blocks also implies a partitioning $p(I_k)$ of instructions I_k into $p(I_k) = \texttt{CPU}$ or $p(I_k) = \texttt{ACC}$, since $\forall k, l : I_k \in B_l \Rightarrow p(I_k) = p(B_l)$. At this time we do not assume a concurrent execution on both CPU and accelerator, so the

execution order of the program remains unchanged regardless of the mapping. One way to establish concurrent execution could be through thread level parallelism by adding multi-threading support to CPU and accelerator which does not impact the analysis presented here.

Our model limits the total number of instructions mapped to the accelerator and accounts for a unit area for each such instruction, thus modeling a structurally programmable accelerator without temporal reuse of functional units through time-multiplexing. The size A of the accelerator denoted in area units is part of the architecture parameters. An alternative model is to limit the size of each basic block mapped to the accelerator and to utilize dynamic reconfiguration between consecutive execution of different basic blocks on the accelerator. While such an approach would allow us to map an arbitrary number of instructions to the accelerator, it would also require a more involved modeling of the configuration process, including, for example, the number of configuration bits per instruction, the size and latency of the configuration memory and the required reconfiguration time. It would thus either significantly increase the design space to explore, or limit the model to a specific setup of an accelerator architecture.

Using the LLVM compiler infrastructure to instrument the benchmark codes and profile program executions, we determine the execution count for an instruction I_k as $n(I_k)$, for a basic block B_l as $n(B_l)$, and the number of control flows over an edge $B_l \rightarrow B_m$ of the control flow graph as $n(B_l, B_m)$. Furthermore, we denote the jth execution of instruction I_k as I_k^j and use this separation to determine the level accessed in the memory hierarchy for each load/store instruction as $v(I_k^j)$, with the possible values $\texttt{L1}, \texttt{L2}, \texttt{L3}$ and $\texttt{MEM}$.

These values $v(I_k^j)$ are determined in two steps. First, during the benchmark characterization phase we add a memory profiling pass to LLVM and perform a functional simulation of a deep virtual cache hierarchy consisting of many levels of inclusive, direct mapped caches with sizes increasing by a factor of two for every level. This simulation maps each memory access to a virtual cache level $V(I_k^j)$ and additionally annotates it with a dependency pointer $P(I_k^j)$, referring to the previous memory instruction that accessed the same cache line. In the performance estimation phase, the virtual cache hierarchies are collapsed to the physically existing caches, which hence need to have power-of-two size. Given a concrete mapping of each load/store instruction to CPU or accelerator, the dependency pointer allows to determine in which private or shared cache the first valid cache line for this memory access is located. Thus, we can compute the memory access time for each partitioning step without a repeated cache simulation, which would otherwise slow down the partitioning process significantly. At this point we also profit from the missing register allocation in that no register spills occur, which would change the memory access patterns for different mappings.

Finally, for each architecture-specific execution profile that includes a complete mapping of each basic block to a processing unit, we estimate the total program runtime as the sum of four components: the execution time t_e of instructions with only register operands, the memory access time t_m for load and store instructions, the time t_c for transferring control between successive basic blocks mapped to different processing units, and the time t_r for

exchanging register values between CPU and accelerator:

$$t = t_e + t_m + t_c + t_r$$

Execution time:

$$t_e = \sum_{k:I_k=\texttt{op}} n(I_k) \cdot \epsilon(p(I_k))$$

Memory access time:

$$t_m = \sum_{k:I_k=\texttt{ld/st}} \sum_{j=1}^{n(I_k)} \lambda_m(v(I_k^j))$$

Control transfer time:

$$t_c = \sum_{(l,m)} n(B_l, B_m) \cdot \lambda_c$$
$$\forall (l,m) : (B_l \rightarrow B_m) \wedge p(B_l) \neq p(B_m)$$

Register value transfer time:

$$t_r = \sum_{R} \min\left(n(B_l) \cdot \lambda_{r,\texttt{push}}, n(B_m) \cdot \lambda_{r,\texttt{pull}}\right)$$
$$\forall R : R \in R_w(B_l) \wedge R \in R_u(B_m) \wedge p(B_l) \neq p(B_m)$$

6.3.2 Partitioning Approach

We utilize a greedy partitioning algorithm that starts with all blocks at the CPU and iteratively moves partitioning objects *po* to the accelerator, as long as the cumulative area of all moved blocks fits the size of the accelerator. At each step, the partitioning object with the highest attractiveness, which is the ratio of estimated speedup to area requirements, is chosen. An overview of the partitioner is given in Algorithm 6.1. As presented in the previous section, we assume that each instruction in the mapped basic blocks requires one area unit on the accelerator.

Algorithm 6.1 multi-level partitioning

Input: BasicBlocks initialized with mapping to CPU
Input: A = size of total accelerator resources
Output: partitioned BasicBlocks

```
1: Next ← getBestPartitionObject(A)
2: while Next ≠ null do
3:    moveToAccelerator(Next)
4:    A ← A - computeArea(Next)
5:    Next ← getBestPartitionObject(A)
6: end while
```

Function 6.2 getBestPartitionObject

Input: A = size of currently available accelerator resources

```
1: Objects ← getMultiLevelPartitioningObjects()
2: BestObject ← null
3: BestAttractiveness ← 0
4: for all Object ∈ Objects do
5:    Attractiveness ← computeSpeedup(Object) / computeArea(Object)
6:    if Attractiveness > BestAttractiveness ∧ A > computeArea(Object) then
7:       BestAttractiveness ← Attractiveness
8:       BestObject ← Object
9:    end if
10: end for
11: return BestObject
```

As partitioning objects, we use not only all single basic blocks but additionally all loops (inner as well as nested loops) and all functions. Furthermore, whenever the partitioner moves part of a loop or function to the accelerator, the remaining basic blocks of the loop or function form another new partitioning object. In contrast to the partitioning method from Chapter 5 that focused solely on loops because of the accelerator architecture, we now include smaller and larger partitioning objects to cover different accelerator execution models. Hence, we denote this heuristic as *multi-level partitioning*. Beyond individual basic blocks, loops and functions, other beneficial partitioning objects may exist, like subsets of loops or functions, or like a pair of basic blocks with dependent memory accesses. However, we restricted the search space to the described categories, because the total number of all possible partitioning objects grows exponentially with the number of basic blocks. We have exemplarily validated the quality of this partitioning method with exact solutions obtained through integer linear programming. The results show that most solutions generated by the heuristic partitioner are identical or very close to the optimal solutions. Due to lengthy execution times of the exact method, the design space exploration in the following section builds upon results from the heuristic method.

6.4 Design Space Exploration

Based on this estimation method and framework, we now explore the potential and design space of our architecture model for the integration of CPUs with reconfigurable accelerators. All presented experiments are obtained with the partitioning approach introduced in the previous section and use the default parameters listed in Table 6.5, unless stated otherwise. The default memory hierarchy is a configuration with shared L2 and private L1 caches as depicted in Figure 6.1 and its size and latency properties mimic those of early dual core GPPs. The accelerator size is expressed by the number of LLVM instructions I_k that can be mapped to the reconfigurable hardware processing unit. The chosen accelerator efficiency models a rather modest speedup potential of up to 2x, because we don't limit the

Table 6.5: Default model parameters for design space exploration.

Description	Symbol	Value
Execution efficiencies	$\epsilon(\texttt{CPU})$	1.0 cycles
	$\epsilon(\texttt{ACC})$	0.5 cycles
Communication latencies	λ_c	2 cycles
	$\lambda_{r,push}$	1 cycles
	$\lambda_{r,pull}$	3 cycles
Cache latencies	$\lambda(\texttt{L1})$	3 cycles
	$\lambda(\texttt{L2})$	15 cycles
	$\lambda(\texttt{MEM})$	200 cycles
Cache sizes	L1	32KB
	L2	4MB
Accelerator size	A	128

accelerator model to extreme parallelism or customization that would justify much higher speedup potentials.

We investigate a set of 13 benchmarks that represent compute intense kernels from various application domains. Most of the benchmarks are taken from the MiBench suite [106] which contains six categories. The automotive, consumer, network and office domains are covered by the Susan, JPEG, Djikstra and Stringsearch applications respectively. From the telecommunications domain, we have included ADPCM, CRC32 and fast Fourier transform (FFT). From the security area, we use Blowfish and SHA from MiBench and add MD5 and AES. Furthermore we use the Whetstone benchmark and a SOR implementation representing the field of numerical analysis.

6.4.1 Speedups per benchmark

Figure 6.5 shows the speedup potential that our estimation method predicts for our architecture model for the presented benchmarks with an accelerator size of 128 as specified in Table 6.5 and with a large accelerator size of 2048. The speedups range from 1x to 1.78x at a size of 128 and from 1.28x to 1.91x for a size of 2048. As mentioned, the accelerator efficiency of the baseline model limits speedups to at most 2x. Code remaining on the CPU, communication overheads and memory limitations contribute to the observed speedups below this limit.

We note that for a number of benchmarks the default accelerator size of 128 already suffices, whereas the cryptographic benchmarks require a larger size to show notable speedups at all. A more detailed analysis of accelerator sizes follows in Subsection 6.4.3. Aside from the size requirements of the cryptographic benchmarks, their speedups are notably slower than those of fully custom cryptographic implementations on FPGAs. Both effects can

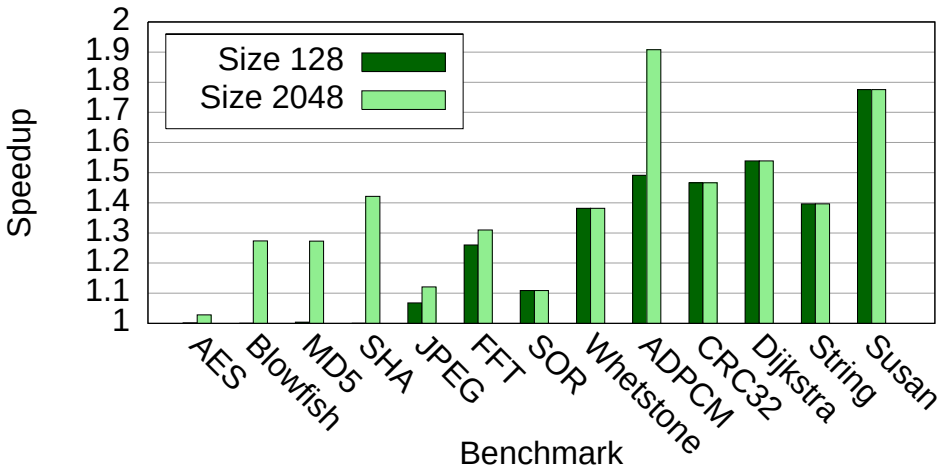

Figure 6.5: Speedups for 13 benchmarks with two different accelerator sizes. Further architecture parameters as specified in Table 6.5.

Table 6.6: Alternative parameters used for three level cache hierarchy.

Cache latencies	λ(L1)	4 cycles
	λ(L2)	11 cycles
	λ(L3)	39 cycles
	λ(MEM)	150 cycles
Cache sizes	L1	32KB
	L2	256KB
	L3	8MB

be attributed to the fact that our estimation method does not capture the bit level parallelism that is typically exploited to accelerate cryptographic functions on reconfigurable hardware.

6.4.2 Memory integration

We investigate the impact of different models for integrating the accelerator into the memory hierarchy. As starting points we use the two-level cache configuration specified in Table 6.5 and a three-level cache configuration that mimics more recent general purpose CPUs with smaller and faster L2 cache and an additional L3 cache as specified in Table 6.6. In both hierarchies we integrate the accelerator with five different design points: shared L1 data cache (*SL1*), shared L2 cache where both CPU and accelerator have private L1 caches (*SL2+*) (shown in Figure 6.1), shared L2 cache where only the CPU has a private L1 cache and the accelerator has no local memory at all (*SL2-*), shared main memory or shared L3 cache with private caches for both components (*SMM+* or *SL3+*), and shared

Table 6.7: Investigated cache configurations. Cache sizes follow Tables 6.5 and 6.6 respectively.

	two level cache hierarchy			three level cache hierarchy		
	shared	**CPU private**	**ACC private**	**shared**	**CPU private**	**ACC private**
SL1	L1 – MM	—	—	L1 – MM	—	—
SL2+	L2, MM	L1	L1	L2 – MM	L1	L1
SL2-	L2, MM	L1	—	L2 – MM	L1	—
SMM+	MM	L1, L2	L1, L2			
SMM-	MM	L1, L2	—			
SL3+				L3	L1, L2	L1, L2
SL3-				L3	L1, L2	—

main memory or shared L3 cache with private caches only for the CPU (*SMM-* or *SL3-*). We summarize these configurations in Table 6.7. They cover different aspects of available and announced architectures mentioned in Subsection 6.2.1. The *SL1* design points are very similar to the closely coupled VISC processor cores. Zynq and Altera SoC devices resemble a *SL2-* configuration and POWER8 with CAPI accelerator represents something between *SL3+* and *SL3-*, with a private cache on the FPGA side, but not symmetric to the private CPU caches.

For each of these configurations, we evaluate two parameter variants, one with the unaltered parameters from Tables 6.5 and 6.6, and the other one with an increased latency of the first shared cache by one cycle. This penalty is to reflect the increased complexity of a shared cache over a private one. Overheads for coherency that occur at private caches are not separately modeled. Figures 6.6 and 6.7 show the results averaged over all 13 benchmarks for the two-level and three-level cache hierarchies respectively. It turns out that even though the results of the different cache hierarchies show some differences, the big picture remains the same for both.

We note that without penalty, a shared L1 cache (*SL1*) delivers the best performance. However, this architectural design point turns out to be highly sensitive to latency penalties. Overall, the shared L2 cache with private L1 caches for both CPU and accelerator (*SL2+*) is a well-performing and also robust design point since it retains most of its speedup potential when applying the latency penalty to the shared L2 cache. Design points with private caches on both sides and shared main memory or shared L3 caches (*SMM+* and *SL3+*) show lower but still notable speedups and a similar robustness. When the proposed architecture gets integrated into a multicore CPU, *SL2+* might also be the best design for architectural integration, sharing the memory between an accelerator and its controlling CPU on L2 and between multiple CPU-accelerator pairs on L3, like illustrated in Figure 6.2.

The memory hierarchies where the accelerator does not have a private cache, exhibit a much lower performance, with similar values for the two- and three-level hierarchy and for *SL2-*, *SMM-* and *SL3-*. While this experiment underlines the necessity for the accelerator to have access to low latency memory, they do not prove that this memory actually needs

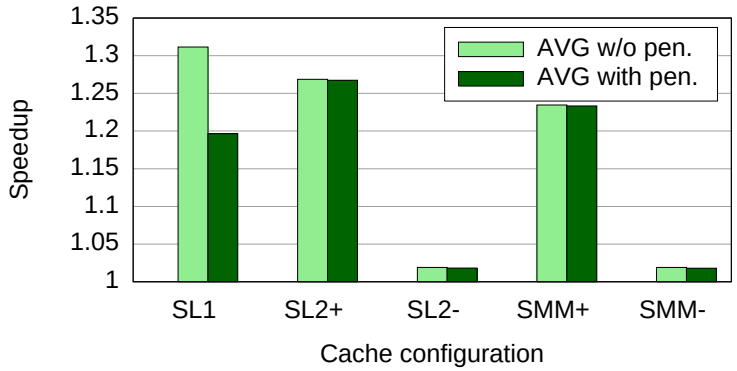

Figure 6.6: Speedups for different memory integrations into a *two level cache hierarchy*, with and without a 1 cycle latency penalty for shared caches, averaged over 13 benchmarks.

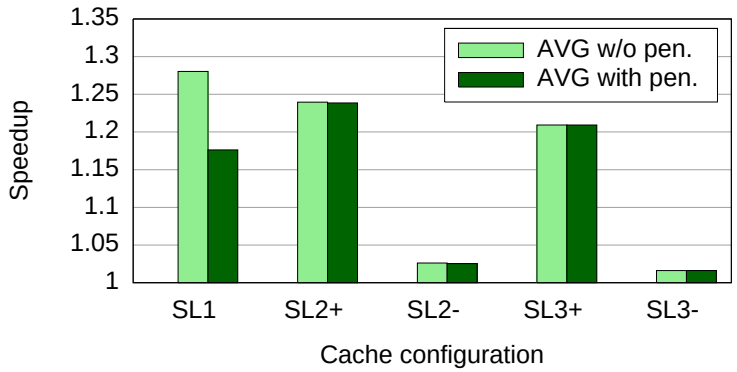

Figure 6.7: Speedups for different memory integrations into a *three level cache hierarchy*, with and without a 1 cycle latency penalty for shared caches, averaged over 13 benchmarks.

to be a cache, but explicitly managed scratchpad memory or BRAM resources might be a viable alternative. However, targeting explicitly managed local memory requires more development effort from application developers, so private caches on the accelerator side seem much more attractive for general-purpose adoption of reconfigurable accelerators. Our practical experiences with the Convey HC-1 platform, which lacks such a private accelerator cache underline this, as the kernels turned out to be highly sensitive to data reuse in registers. Thus, from this perspective, the POWER8 with CAPI architecture is a promising step in the right direction from the Zynq and Altera SoC architectures.

6.4.3 Accelerator Size

After our first overview of speedup numbers in Figure 6.5 already gave a glimpse at the impact of accelerator sizes, we now vary this parameter systematically. Figure 6.8 shows that increasing the size of the accelerator causes the average speedups to grow with steadily diminishing returns. Note that the x-axis is already exponentially scaled and the y-axis

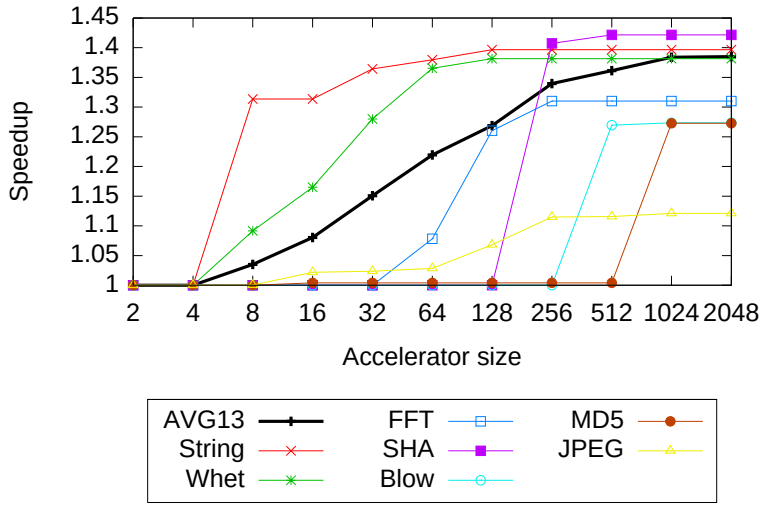

Figure 6.8: Speedups for different accelerator sizes, averaged over all 13 benchmarks and with selected single benchmarks.

is linearly scaled and still the graph of the average speedup becomes flatter and almost saturates at a size of 1024.

A more detailed investigation of selected individual benchmarks shows that the speedup does not grow smoothly with increasing accelerator size. Instead, most benchmarks show a phase transition behavior, that is, the speedup increases significantly when the accelerator size exceeds a certain threshold that allows mapping a beneficial selection of basic blocks to the accelerator. When further increasing the accelerator size, the speedups flatten out. The location of this phase transition threshold depends strongly on the application. We have selected some benchmarks to clearly illustrate this effect in Figure 6.8. While the Stringsearch benchmark already shows a phase transition between a size of 4 and 8, Whetstone profits in a range of 4 to 64 and FFT from 32 to 128. The cryptographic benchmarks SHA, Blowfish and MD5 show a very pronounced phase transition behavior when reaching an accelerator size of 128, 256, and 512 respectively.

This behavior is a problem when determining the size of an accelerator for general-purpose computing. On the one hand, it reflects a limitation of our estimation method that does not assess the potential for temporal reuse of accelerator resources. However, given the wide range of transition points observed, we don't expect this effect to entirely disappear with temporal reuse. Thus, on the other hand, it underlines the general importance of scalable methods to program reconfigurable accelerators, either with configuration parameters as demonstrated in Chapter 4, or by automated methods that work for different accelerator sizes.

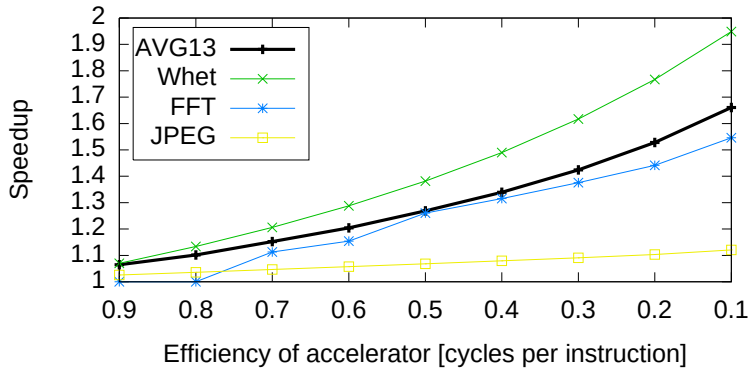

Figure 6.9: Speedups for different accelerator execution efficiencies, averaged over all 13 benchmarks and with selected single benchmarks.

6.4.4 Execution Efficiency

Next we investigate the effect of the execution efficiency of the accelerator. We vary this value between 0.9 and 0.1 cycles per instruction (CPI), while the efficiency of the CPU remains constant. For the majority of benchmarks, in Figure 6.9 exemplarily represented by Whetstone and JPEG, already for minor differences between the execution efficiencies of accelerator and CPU, a partitioning is found that generates speedups. For Whetstone and several other benchmarks, this partitioning remains to a large degree unchanged (not visible in the Figure), while the model parameters are changed towards a more efficient accelerator and yield almost linear savings in total execution time. Since the computation of speedups puts these savings in relation to the final execution time, the graphs grow more than linearly the larger speedups become. For other benchmarks, most pronouncedly JPEG, the resulting partitioning differs significantly for different accelerator efficiencies, nevertheless the resulting speedups after considering computational gains and communication overheads, grow quite regularly. The FFT benchmark shows an exception: here a certain difference in execution efficiencies is required, before the potential savings in computation time overcome the communication penalties and enable speedups at all. Therefore the speedup graph for FFT grows irregularly.

We conclude that for the majority of investigated benchmarks, already moderate differences in computation times overcome the communication latencies for the selected architecture parameters and continue to investigate this observation from another point of view in the next subsection.

6.4.5 Interface Latency

Figure 6.10 depicts the relative performance over the latency of the interface between CPU and accelerator. We apply one scaling factor to all three parameters of the interface latency, $\lambda_c, \lambda_{r,pull}$ and $\lambda_{r,push}$, and investigate both lower and higher latencies than the default settings of Table 6.5. Even though it might be architecturally infeasible to reach

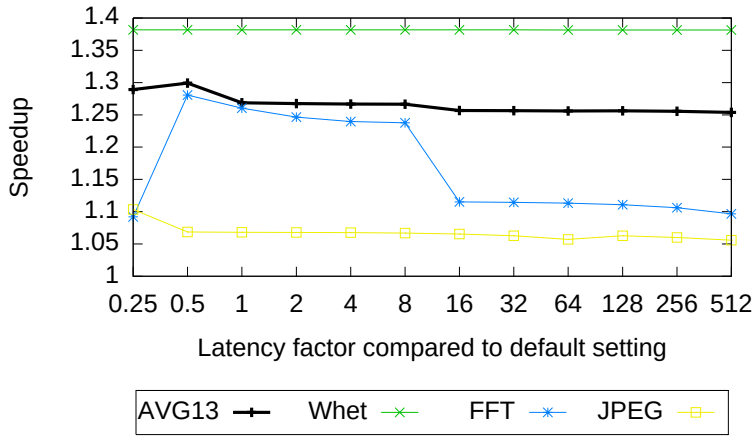

Figure 6.10: Speedups for different interface latencies, averaged over all 13 benchmarks and with selected single benchmarks.

the lowest studied latencies, the experiment provides insights into the nature of the design space.

A number of benchmarks, represented by Whetstone, has very stable partitioning results not only for varying execution efficiencies as discussed in the previous section, but also for varying interface latencies. This is enabled by program parts that can be moved to the accelerator with minimal communication requirements. Other benchmarks, again represented by JPEG, change their partitioning results with increasing latency factors, but still show relatively stable speedups. Again, FFT is most sensitive to increased latency factors and looses a significant portion of its speedups when the factor grows from 8 to 16. Yet overall the results are remarkably stable for long communication latencies. We conclude that our current selection of benchmarks do not significantly profit from fine-grained acceleration. This coincides with the discussed architecture trends with the exception of the VISC processors, which however promote concurrent execution on collaborating cores, whereas our model dedicatedly passes control between CPU and coprocessor.

In figure 6.10 we also observe two slightly anomalous results, which can be attributed to our greedy partitioning approach. For the extremely low latencies obtained with a factor of 0.25, the results for FFT are much worse than for the following latency factors. Also the JPEG result for factor 64 is a little bit worse than that for 128. In both cases, the partitioning results require a lot more communication than the ones for larger latencies. Clearly, the latter results would also have been superior for lower latencies, but are not found due to the greedy nature of our partitioning algorithm.

6.5 Related Work

Among known high level estimation methods, the approach of Spacey et al. [241] is most closely related to our work. There are, however, three important differences. First, in

contrast to the estimation method of Spacey et al. which models the memory subsystem with a single bandwidth parameter, our framework includes cache models and thus more realistically mimics relevant architectures. Second, their partitioning approach is limited to the basic block level, whereas for our experiments the multi level partitioning technique was required to obtain good partitioning results. Finally, whereas the system of Spacey et al. is x86 assembly based and can therefore be applied to x86 binary code, our framework leverages the LLVM infrastructure [163] which allows us to extend the framework towards automated code generation for various targets. A first step in this direction is presented in [1], whereas the work in Chapter 5 represents a more advanced evolution of this concept, yet with practical restrictions to a specific acceleration target.

The field of binary level partitioning has been investigated by Stitt et al. [244], who initially rely on analysis of block sizes and iteration counts of small kernel loops. In [245], they incorporate alias information to identify and group together regions of code that access the same memory locations, but without using a model for data flow and communication latencies.

The work of Holland et al. [126] provides a method for analyzing algorithms on a more abstract level in order to estimate how they can perform on CPU-accelerator architectures. In contrast to our work, their method requires user interaction for example to identify data elements and their communication patterns. It incorporates measured throughput values of existing architectures, which can increase accuracy compared to our latency based model, but limits the applicability for design space exploration of new architectures.

Henkel et al. [117] present a partitioning approach with partitioning objects with a flexible granularity ranging from one to many basic blocks. This approach is potentially more powerful than ours, but also requires more in-depth analysis both for the creation and for the selection of appropriate partitioning objects.

6.6 Chapter Conclusion

This chapter presents our high-level performance estimation method and framework for static and dynamic code analysis and multi-level hardware/software partitioning. The capability of our framework to provide fully automated estimation and partitioning results can be used for a systematic design space exploration that was previously hard to undertake for new architectures. We demonstrate this by applying our framework to a proposed class of CPU-accelerator architectures with a dual interface that aims to overcome the limitations of previously proposed architectures that were tailored to specific granularities of acceleration targets.

Our results indicate that the memory integration of CPU and accelerator is a very important design aspect. Within our model, either private caches on the accelerator side or a technically challenging shared L1 cache are required to achieve reasonable speedups. Various case studies have demonstrated that the not covered alternative of explicitly managing local memory can yield competitive performance, but it is much less productive for application designers. Compared to the high importance of memory integration, our results are

much less sensitive to the latency of the direct communication interface. In contrast to our earlier assumptions, the distinctive potential of fine-granular acceleration did not show up within our performance model for the set of analyzed benchmarks.

The recent industry trends for the integration of FPGAs with general-purpose processor systems show a remarkable correspondence. While earlier systems like the Convey HC-1 focused on the functional integration of memory spaces, the Xilinx Zynq and Altera SoC brought a much closer coupling with shared access to an L2 cache. More recently, POWER8 and CAPI take up the idea of a private cache on the FPGA side. On the other hand, none of these architectures took up the idea of a low-latency direct interface, out of technical infeasibility for Convey and POWER8, but for Xilinx Zynq and Altera SoC maybe also because of the limited potential like indicated by our experiments. With their acquisition of Altera in 2015, Intel placed a 17 billion dollar bet on FPGA acceleration. Although first server processor products with CPU and FPGA chips in the same package are announced, it remains to be seen which path towards further integration Intel will pursue.

Overall, our performance model makes many abstractions in order to provide the flexibility and speed that we aimed for. We have not been able to quantify the accuracy of the model or to demonstrate that the abstraction-induced inaccuracies somehow even out on average. However, qualitatively, concrete architecture trends seem to vindicate our results, which were unseen in this generality before.

The LLVM-based analysis and partitioning infrastructure used for the experiments in this chapter has been evolved into further offloading and analysis projects [1, 7, 11, 12] presented within this thesis (mostly Chapter 5) to different extends. As curious observation, while we found the mix of instrumentation and profiling necessary to extend static code analysis for application characterization in [11, 12], we demonstrated that by deferring concrete offloading decisions to application runtime, the same dynamics can often be covered more precisely and with minimal overhead without profiling [7, 11, 12].

CHAPTER 7

Conclusion

In this chapter, we briefly summarize the results of this thesis, before discussing next steps, future trends and research opportunities, first concretely around overlays and then with a broader scope.

7.1 Summary

In this thesis, we have investigated if and how FPGAs can play a bigger role in general-purpose computing. We have started with an overview on fundamental approaches to and trends in computing. From an architectural perspective, we see FPGAs with their ability to customize computations and parallelism for different workloads as promising candidates for several general-purpose computing markets.

We identified design productivity, including portability and scalability, as central challenge for more wide-spread adoption of FPGAs in these markets. We propose to supplement current industry efforts on OpenCL and application libraries with overlay architectures on FPGAs as targets for fast and automated compilation. However, for instruction-programmable overlays, neither the involved overheads nor the practicability as compilation target were sufficiently investigated.

In order to quantify such overheads, we extracted ten kernels from a stereo-matching application with high accuracy and general-purpose characteristics and ported them both to a vector coprocessor as overlay architecture and to fully customized FPGA implementations. We demonstrate that both targets enable speedups for the overall stereo-matching tasks, and compensate for hardware platform specific effects to quantify the overlay overhead as around 3x for a diverse set of kernels. On a detailed look, this number is much smaller for regular streaming kernels, and higher for kernels with different forms of customization potential.

The productivity of available compilation tools targeting this overlay did hardly justify such overheads. However, with our work we demonstrate that the knowledge of overlay

architecture can indeed enable fully automatic offloading processes with versatile and efficient code generation from software source code or binaries. In two excursions, we discuss that deferring offloading decisions to program runtime can often enable fast and precise decisions from the application perspective, whereas decisions on the system level can lead to overall better utilization of heterogeneous resources.

Furthermore, we have investigated the system integration of reconfigurable accelerators with GPPs. Based on an estimation method that overcomes the interdependency between architectures and application design, we explored the design space for such heterogeneous systems. Our proposed tighter integration into a common memory hierarchy is now reflected by actual products that recently emerged.

7.2 Outlook

Our outlook is subdivided into next steps and future directions in the area of overlay architectures on the one hand, and a broader view on future computing systems and their execution models. In our ongoing research in the area of OTF computing, we continue to focus on the conceptional questions around overlay architectures, without loosing touch to the dynamic evolution of industry developed hardware architectures and tools.

7.2.1 Towards a Library of Overlay Architectures and Tools

Our concrete work has demonstrated the feasibility of one type of instruction-programmable overlay architectures as a means of highly productive FPGA acceleration and complements similar related work for structurally-programmable overlays, in particular in the form of intermediate fabrics [63]. More overlays, like instruction-programmable manycore or GPU-like architectures, or like structurally-programmable architectures with decoupled functional units and local control, need to be understood on a comparable level with regard to overheads and compilation support. Also, it is not sufficiently clear to which degree different overlays complement each other for different workloads, or compete for a few particularly well-suited application patterns.

For a larger practical impact, such overlay research needs to be consolidated in two regards. Firstly, we need a library of different overlays synthesized for many FPGA types and ready-to-use with the according software interfaces in the relevant current systems with FPGA accelerators. Secondly, compilation tools targeting these overlays need to be systematically brought to production quality. Not all overlays and tools need to support the same set of source languages and program patterns, but for an application developer it needs to be either very clear which one to use, or he needs a common design entry point behind which the further selection of overlays and tools can be hidden. It remains an open research question, how many of the manifold application patterns that are generally suitable for FPGA acceleration can be efficiently covered with overlay architectures and corresponding tools.

When a first small library of overlays and compilation tools is established, further research needs to focus on scalability of overlay architectures and execution models, and on increasing efficiency through customization. While examples for successful customization have been demonstrated both for instruction-programmable and for structurally-programmable overlays, it is unclear how customization can be applied systematically, both with regard to the extraction of workload characteristics and with regard to common refinement methods for different overlay architectures. The generation of workload-customized overlay variants poses further questions concerning the prediction of customization effects and practical strategies for the local or centralized synthesis and deployment of designs.

In the longer run, an ecosystem around overlay libraries on FPGAs may not only complement OpenCL-based synthesis and application libraries, but provide further synergies. For example, when a specific kernel or application is running frequently on some overlay architecture, the next step after specialization of this architecture can be a fully custom design. Thus, the overlay would help to find the most promising workloads for application libraries on FPGAs and characterize the required degrees of flexibility that a custom design may need to exhibit. With regard to the other pillar, instruction-programmable overlays may serve as intermediate compilation target for OpenCL code, whenever the lengthy synthesis process is not acceptable, or when quickly changing kernels prefer a reusable configuration of the FPGA, like in our case-study in Chapter 6.

7.2.2 And Beyond

Beyond the productivity-driven research on overlay architectures on FPGAs, further research on architectures and execution models can contribute to reach new dimensions in absolute performance in HPC, in energy constrained performance mobile devices, and in highest efficiency for scaling workloads in cloud and OTF data centers. Particularly for the consolidated execution of diverse workloads in data centers, the interaction between heterogeneous scheduling decisions from a system perspective and application internal knowledge need to be further investigated.

Our work and most related work on offloading to accelerators leaves the high-level program structure intact. In order to harvest additional parallelism on thread and task levels, programming models for asynchronous task execution are actively researched in several computing domains. Heterogeneous architectures with reconfigurable accelerators can profit from such approaches on different granularity levels.

Concerning heterogeneity, we focused in this thesis on the system integration of FPGAs with CPUs. The most prominent driver for integrated heterogeneous systems have been GPUs. From a performance and efficiency perspective, the trends go towards ever more heterogeneity between general-purpose and differently specialized components. We believe that the wide success of systems with extreme heterogeneity depends on productive programming models and tools.

Acronyms

ACP Accelerator Coherency Port. 135

ADPCM Adaptive Differential Pulse Code Modulation. 142

AES Advanced Encryption Standard. 58, 59, 142

AGU address generation unit. 28

ALM adaptive logic module. 22

ALU arithmetic logic unit. 8, 12, 16, 17, 21, 28, 33, 58

API application programming interface. 38, 48, 49

ASIC application specific integrated circuit. 17, 22, 30, 33, 39, 41

ASIP application-specific instruction set processor. 17, 30, 38, 42, 59

BLAS Basic Linear Algebra Subprograms. 50

BLE basic logic element. 21

BRAM block RAM. 22, 23, 33, 53, 91, 106, 136, 145

CAPI Coherent Accelerator Processor Interface. xviii, 51, 135, 136, 145, 150

CFB configurable function block. 60

CGRA coarse-grained reconfigurable array. 57, 60–62

CISC complex instruction set computer. 9, 15

CLB configurable logic block. 22

CPI cycles per instruction. 138, 147

CPU central processing unit. xiii, xvii, 2–4, 8, 11, 12, 17, 18, 30, 39, 41, 45, 47, 49–52, 56, 58, 65, 93, 96, 103, 114, 124, 126, 127, 131–136, 138–142, 144, 147–150, 153

CRC collaborative research centre. 43

CRC32 cyclic redundancy check, 32-bit. 142

CUDA former acronym: Compute Unified Device Architecture. 49, 50

DDR double data rate. 22

DFG data flow graph. 60–62

DIMM dual inline memory module. 109

DLP data-level parallelism. 11, 12, 18, 29, 30, 44, 48–50, 58, 65, 82, 109

DSL domain-specific language. 61

DSP digital signal processor. 9, 13, 15, 22, 24, 33, 38, 51, 52, 61

ECU electronic control unit. 37

EPIC Explicitly Parallel Instruction Computing. 28

FDTD finite difference time domain. 40

FF flip-flop. 21, 91

FFT fast Fourier transform. 142, 146–148

FIFO first in, first out. 18–20

FPGA field programmable gate array. v, vi, xiii, 1–5, 7, 21–26, 32–35, 38–47, 49, 51–66, 73, 75, 77, 90, 104, 109–115, 126–128, 131–136, 142, 144, 150–153

FSM finite-state machine. 18

FU functional unit. 60, 61

GPIO general-purpose input-output pins. 22

GPP general-purpose processor. 2, 9, 13, 29, 30, 32–34, 38, 44, 45, 75, 111, 125, 131, 133, 141, 152

GPU graphics-processing unit. 2, 12, 13, 30–35, 39–41, 43–46, 49–56, 58, 63, 75, 77, 109, 127, 132, 152, 153

HDL hardware description language. 23, 52–54, 56

HLS high-level synthesis. 23, 52–55, 59, 63

HMC Hybrid Memory Cube. 34

HPC high-performance computing. 2, 5, 35, 36, 39, 40, 42, 44, 45, 48, 50, 52, 54, 59, 112, 153

HSA Heterogeneous System Architecture. 51, 132

ILP instruction-level parallelism. 10, 11, 18, 28–30, 48

IO input output. 22

IPC instructions per cycle. 10, 11, 17, 28, 138

IR intermediate representation. 113, 115

ISA instruction set architecture. 8–10, 15, 17, 28, 30, 42, 59, 111, 112

ISE instruction set extension. 60

IT information technology. 43

ITRS International Technology Roadmap for Semiconductors. 26

JPEG format defined by Joint Photographic Experts Group. 142, 147, 148

LAB logic array block. 22

LLP loop-level parallelism. 11, 18, 20, 21

LLVM compiler infrastructure; former acronym: Low Level Virtual Machine. 3, 113, 115, 131, 139, 149, 150

LUT lookup table. 21, 22, 24–26, 52, 91, 105

MAC multiply-accumulate. 22

MD5 Message-Digest Algorithm 5. 142, 146

MIMD multiple instruction, multiple data. 12, 49

MM-RPU Multimodal Reconfigurable Processing Unit. iii, 131

MMU memory management unit. 134

NRE non-recurring engineering. 7, 35, 39, 41, 42, 45, 60

NUMA non-uniform memory access. 51

OpenCL Open Compute Language. 2, 47, 49, 50, 53–57, 62, 63, 112, 128, 151, 153

OS operating system. 38, 51

OTF On-The-Fly. iii, 5, 43–45, 56, 111, 113, 115, 127, 128, 152, 153

PC personal computer. 35, 41, 42, 52, 66, 127

PCIe peripheral component interconnect express. 23, 109, 135

PE processing element. 60, 61

PLD programmable logic device. 21

RAM random access memory. 21, 22

RFU reconfigurable functional unit. 60

RISC reduced instruction set computer. xiii, 9, 14–16, 33, 60

RTL register-transfer level. 23

SaaS software as a service. 43

SDN software-defined networking. 39

SHA Secure Hash Algorithm. 142, 146

SIMD single instruction, multiple data. 11, 12, 27, 29–31, 38, 43, 44, 49–51, 58, 59, 93, 125

SIMT single instruction, multiple thread. 12, 49, 50, 58

SMP symmetric multiprocessor. 12

SMT simultaneous multithreading. 13

SoC System-on-Chip. 30, 32–34, 38, 42, 132, 135

SOR Successive Over-Relaxation. 142

TLP thread-level parallelism. 12, 18, 29, 48, 58

USB Universal Serial Bus. 23

VHDL Very high speed integrated circuit Hardware Description Language. 23, 52

VLIW very long instruction word. 10, 28, 30, 48, 58

WTA winner takes all. 67, 72

Author's publications

[1] Pablo Barrio, Carlos Carreras, Roberto Sierra, Tobias Kenter, and Christian Plessl. Turning control flow graphs into function calls: Code generation for heterogeneous architectures. In *Proc. Int. Conf. on High Performance Computing and Simulation (HPCS)*, pages 559 –565, July 2012.

[2] Tobias Kenter, Marco Platzner, Christian Plessl, and Michael Kauschke. Performance estimation for the exploration of CPU-accelerator architectures. In Omar Hammami and Sandra Larrabee, editors, *Proc. Workshop on Architectural Research Prototyping (WARP)*, June 2010.

[3] Tobias Kenter, Christian Plessl, Marco Platzner, and Michael Kauschke. Performance estimation framework for automated exploration of CPU-accelerator architectures. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 177–180. ACM, 2011.

[4] Tobias Kenter, Henning Schmitz, and Christian Plessl. Pragma based parallelization – trading hardware efficiency for ease of use? In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE Computer Society, December 2012.

[5] Tobias Kenter, Henning Schmitz, and Christian Plessl. Kernel-centric acceleration of high accuracy stereo-matching. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE Computer Society, December 2014.

[6] Tobias Kenter, Henning Schmitz, and Christian Plessl. Exploring trade-offs between specialized dataflow kernels and a reusable overlay in a stereo matching case study. *Int. Journal of Reconfigurable Computing (IJRC)*, page 24, 2015.

[7] Tobias Kenter, Gavin Vaz, and Christian Plessl. Partitioning and vectorizing binary applications for a reconfigurable vector computer. In *Proc. Int. Symp. on Reconfigurable Computing: Architectures, Tools, and Applications (ARC)*. Springer, April 2014.

[8] Achim Lösch, Tobias Beisel, Tobias Kenter, Christian Plessl, and Marco Platzner. Performance-centric scheduling with task migration for a heterogeneous compute node in the data center. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, 2016.

[9] Heinrich Riebler, Tobias Kenter, Christian Plessl, and Christoph Sorge. Reconstructing AES key schedules from decayed memory with FPGAs. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 222–229. IEEE Computer Society, April 2014.

[10] Heinrich Riebler, Tobias Kenter, Christoph Sorge, and Christian Plessl. FPGA-accelerated key search for cold-boot attacks against AES. In *Proc. Int. Conf. on Field Programmable Technology (ICFPT)*. IEEE Computer Society, December 2013.

[11] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. Deferring accelerator offloading decisions to application runtime. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE Computer Society, December 2014. Received Best Paper Award.

[12] Gavin Vaz, Heinrich Riebler, Tobias Kenter, and Christian Plessl. Potential and methods for embedding dynamic offloading decisions into application code. *Computers & Electrical Engineering*, 2016.

Bibliography

[13] Kanak Agarwal, Kevin Nowka, Harmander Deogun, and Dennis Sylvester. Power gating with multiple sleep modes. In *Proc. Int. Symp. on Quality Electronic Design (ISQED)*, pages 633–637. IEEE Computer Society, 2006.

[14] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. *VLSI Design*, (3):288–298, March 2004.

[15] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.

[16] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proc. ACM SIGPLAN Symp. on Compiler Construction*, pages 233–246, 1984.

[17] Altera. Altera Inc., form 10-k annual report 2012, February 2012.

[18] Altera. Altera Inc., form 10-k annual report 2015, February 2015.

[19] Altera. Product brochure, Altera's user customizable ARM-based SoC. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/br/br-soc-fpga.pdf`, September 2015.

[20] Altera. Stratix V device handbook, volume 1: Device interfaces and integration. https://www.altera.com/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf, January 2015.

[21] Hideharu Amano. A survey on dynamically reconfigurable processors. *IEICE Transactions on Communications*, (12):3179–3187, December 2006.

[22] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proc. ACM European Conference on Computer Systems (EuroSys)*, pages 295–308, 2013.

[23] Sebastian Anthony. Beyond silicon: IBM unveils world's first 7nm chip. `http://arstechnica.co.uk/gadgets/2015/07/ibm-unveils-industrys-first-7nm-chip-moving-beyond-silicon/`, July 2015.

[24] ARM Holdings plc. Annual report 2015: Strategic report. http://phx.corporate-ir.net/External.File?item=UGFyZW50SUQ9MzIzNjAzfENoaWxkSUQ9LTF8VHlwZT0z&t=1&cb=635913243929679950, March 2016.

[25] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, University of California at Berkeley, 2006.

[26] Kubilay Atasu, Laura Pozzi, and Paolo Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. *Int. Journal of Parallel Programming (IJPP)*, (6):411–428, 2003.

[27] Ali Azarian and João M.P. Cardoso. Pipelining data-dependent tasks in FPGA-based multicore architectures. *Microprocessors and Microsystems Journal*, pages 165 – 179, 2016.

[28] Jason D. Bakos. High-performance heterogeneous computing with the Convey HC-1. *Computing in Science and Engineering*, (6):80–87, November 2010.

[29] L. Barthe, L. V. Cargnini, P. Benoit, and L. Torres. The secretblaze: A configurable and cost-effective open-source soft-core processor. In *Proc. Int. Symp. on Parallel and Distributed Processing Workshops (IPDPSW)*, pages 310–313, May 2011.

[30] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP – a self-reconfigurable data processing architecture. *Journal of Supercomputing*, (2):167–184, September 2003.

[31] Antonio Carlos S. Beck, Mateus B. Rutzig, Georgi Gaydadjiev, and Luigi Carro. Transparent reconfigurable acceleration for heterogeneous embedded applications. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 1208–1213. European Design and Automation Association, 2008.

[32] Tobias Beisel, Tobias Wiersema, Christian Plessl, and André Brinkmann. Programming and scheduling model for supporting heterogeneous accelerators in Linux. In *Proc. Workshop on Computer Architecture and Operating System Co-design (CAOS)*, January 2012.

[33] Vaughn Betz and Jonathan Rose. FPGA routing architecture: Segmentation and buffering to optimize speed and density. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 59–68, 1999.

[34] J. Bispo, J.M.P. Cardoso, and J. Monteiro. Hardware pipelining of runtime-detected loops. In *Int. Symp. on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6, 2012.

[35] João Bispo, Nuno Paulino, João M. P. Cardoso, and João Canas Ferreira. Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units. *Int. Journal of Reconfigurable Computing (IJRC)*, 2013.

[36] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 101–113. ACM, 2008.

[37] Bruno Bougard, Bjorn De Sutter, Sebastien Rabou, David Novo, Osman Allam, Steven Dupont, and Liesbet Van der Perre. A coarse-grained array based baseband processor for 100Mbps+ software defined radio. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 716–721. ACM, 2008.

[38] Mathias Brandt. PC-markt fällt auf 8-jahres-tief. `https://de.statista.com/infografik/4238/weltweiter-pc-absatz/`, January 2016.

[39] Alexander Brant and Guy G. F. Lemieux. ZUMA: An open FPGA overlay architecture. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96. IEEE Computer Society, 2012.

[40] T.M. Brewer. Instruction set innovations for the Convey HC-1 computer. *IEEE Micro*, (2):70 –79, March-April 2010.

[41] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, (2):356–373, February 2007.

[42] F. Bruguier, P. Benoit, L. Torres, L. Barthe, M. Bourree, and V. Lomne. Cost-effective design strategies for securing embedded processors. *IEEE Transactions on Emerging Topics in Computing (TETC)*, (1):60–72, Jan 2016.

[43] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE Journal of Solid-State Circuits*, (11):1571–1580, Nov 2000.

[44] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for Haswell's restricted transactional memory. In *Proc. Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 187–200. ACM, 2014.

[45] Timothy J. Callahan, John R. Hauser, and John Wawrzynek. The Garp architecture and C compiler. *IEEE Computer*, (4):62–69, April 2000.

[46] D. Capalija and T.S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE Computer Society, September 2013.

[47] S. Di Carlo, A. Miele, P. Prinetto, and A. Trapanese. Microprocessor fault-tolerance via on-the-fly partial reconfiguration. In *Proc. IEEE European Test Symposium (ETS)*, pages 201–206, May 2010.

[48] J. E. Carrillo Esparza and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proc. 9th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 141–150. IEEE Computer Society, 2001.

[49] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (SCORE). In Reiner W. Hartenstein and Herbert Grünbacher, editors, *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 605–614. Springer Berlin Heidelberg, 2000.

[50] J. Castillo, P. Huerta, V. Lopez, and J. I. Martinez. A secure self-reconfiguring architecture based on open-source hardware. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 7–10, 2005.

[51] Ken Chapman. White paper: Xilinx FPGA families, multiplexer selection. `http://www.xilinx.com/support/documentation/white_papers/wp274.pdf`, February 2008.

[52] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel Core i7 turbo boost feature. In *Int. Symp. on Workload Characterization (IISWC)*, pages 188–197, October 2009.

[53] D. Chen and D. Singh. Invited paper: Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 5–12, Aug 2012.

[54] David Chinnery and Kurt Keutzer. *Closing the Power Gap Between ASIC & Custom: Tools and Techniques for Low Power Design.* Springer, 2005.

[55] David Chinnery and Kurt Keutzer. Closing the power gap between ASIC and custom: An ASIC perspective. In *Proc. Design Automation Conference (DAC)*, pages 275–280, 2005.

[56] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G.F. Lemieux. VEGAS: soft vector processor with scratchpad memory. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 15–24. ACM, 2011.

[57] W. W. S. Chu, R. G. Dimond, S. Perrott, S. P. Seng, and W. Luk. Customisable EPIC processor: Architecture and tools. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 1–6. IEEE Computer Society, 2004.

[58] E.S. Chung, P.A. Milder, J.C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proc. Int. Symp. on Microarchitecture (MICRO)*, pages 225–236, 2010.

[59] Jason Cong, Yiping Fan, Guoling Han, and Zhiru Zhang. Application-specific instruction generation for configurable processor architectures. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 183–189. ACM, 2004.

[60] Patrick Cooke, Lu Hao, and Greg Stitt. Finite-state-machine overlay architectures for fast FPGA compilation and application portability. *ACM Transactions on Embedded Computing Systems (TECS)*, (3):54:1–54:25, April 2015.

[61] J. Coole and G. Stitt. Fast, flexible high-level synthesis from OpenCL using reconfiguration contexts. *IEEE Micro*, (1):42–53, January 2014.

[62] J. Coole and G. Stitt. Adjustable-cost overlays for runtime compilation. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 21–24. IEEE Computer Society, May 2015.

[63] James Coole and Greg Stitt. Intermediate fabrics: virtual architectures for circuit portability and fast placement and routing. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, pages 13–22. ACM, 2010.

[64] Ian Cutress. Examining soft machines' architecture: An element of VISC to improving IPC. `http://www.anandtech.com/show/10025/examining-soft-machines-architecture-visc-ipc/2`, February 2016.

[65] T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D.P. Singh. From OpenCL to high-performance hardware on FPGAs. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE Computer Society, August 2012.

[66] William J. Dally, James Balfour, David Black-Shaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. *IEEE Computer*, (7):27–32, July 2008.

[67] F. de Dinechin, H. D. Nguyen, and B. Pasca. Pipelined FPGA adders. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 422–427, Aug 2010.

[68] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, (1):107–113, January 2008.

[69] André DeHon. Reconfigurable architectures for general-purpose computing. Technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1996.

[70] André DeHon. The density advantage of configurable computing. *IEEE Computer*, (4):41–49, April 2000.

[71] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, Leo Rideout, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Solid-State Circuits Society Newsletter*, (1):38–50, 2007. Reprinted from the IEEE Journal of Solid-State Ciruits, Vol. SC-9, October 1974, pp. 256-268.

[72] Sheng Di, Derrick Kondo, and Walfredo Cirne. Characterization and comparison of cloud versus grid workloads. In *Proc. IEEE Int. Conf. on Cluster Computing (CLUSTER)*, pages 230–238. IEEE Computer Society, 2012.

[73] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, (12):3088–3098, Dec 2014.

[74] H. C. Doan, H. Javaid, and S. Parameswaran. Flexible and scalable implementation of H.264/AVC encoder for multiple resolutions using ASIPs. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 1–6, March 2014.

[75] R.G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, (2):253–266, February 2010.

[76] Steven A. Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design & Test of Computers*, (5):375–386, May 2006.

[77] ESA. LEON: the space chip that europe built. `http://www.spacedaily.com/reports/LEON_the_space_chip_that_Europe_built_999.html`, January 2013.

[78] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 365–376. ACM, 2011.

[79] Jonathon Evans, Kyle Rupnow, and Katherine Compton. Reconfigurable functional units for scientific superscalar processors. In *Proc. Int. Conf. on Field Programmable Technology (ICFPT)*, pages 73–80. IEEE Computer Society, December 2007.

[80] Michael Feldman. OpenCL gains ground on CUDA. `http://www.hpcwire.com/2012/02/28/opencl_gains_ground_on_cuda/`, February 2012.

[81] R. Ferreira, J.G. Vendramini, L. Mucida, M.M. Pereira, and L. Carro. An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 195–204, October 2011.

[82] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers (TC)*, (9):948–960, September 1972.

[83] D.J. Frank, R.H. Dennard, E. Nowak, P.M. Solomon, Yuan Taur, and Hen-Sum Philip Wong. Device scaling limits of Si MOSFETs and their application dependencies. *Proceedings of the IEEE*, (3):259–288, Mar 2001.

[84] Haohuan Fu, William Osborne, Robert G. Clapp, Oskar Mencer, and Wayne Luk. Accelerating seismic computations using customized number representations on FPGAs. *EURASIP Journal on Embedded Systems*, pages 3:1–3:13, January 2009.

[85] Andrea Fusiello, Vito Roberto, and Emanuele Trucco. Efficient stereo with multiple windowing. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 858–863. IEEE Computer Society, 1997.

[86] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin. *High — Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, December 1992.

[87] Philip Garcia and Katherine Compton. A reconfigurable hardware interface for a modern computing system. In *Proc. 15th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 73–84. IEEE Computer Society, April 2007.

[88] Philip Garcia and Katherine Compton. Shared memory cache organizations for reconfigurable computing systems. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 239–242. IEEE Computer Society, 2009.

[89] Johan De Gelas. The Intel Xeon E5 v4 review: Testing Broadwell-EP with demanding server workloads. `http://anandtech.com/show/10158/the-intel-xeon-e5-v4-review`, March 2016.

[90] Heiner Giefers, Christian Plessl, and Jens Förstner. Accelerating finite difference time domain simulations with reconfigurable dataflow computers. *SIGARCH Computer Architecture News*, (5):65–70, June 2014.

[91] Ivan Godard. The mill: Split-stream encoding. *SIGARCH Computer Architecture News*, (5):1–5, June 2014.

[92] M. Golden, S. Hesley, A. Scherer, M. Crowley, S. C. Johnson, S. Meier, D. Meyer, J. D. Moench, S. Oberman, H. Partovi, F. Weber, S. White, T. Wood, and J. Yong. A seventh-generation x86 microprocessor. *IEEE Journal of Solid-State Circuits*, (11):1466–1477, November 1999.

[93] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: a reconfigurable architecture and compiler. *IEEE Computer*, (4):70–77, Apr 2000.

[94] Khosrow Golshan. *Physical Design Essentials: An ASIC Design Implementation Perspective*. Springer-Verlag New York, Inc., 2007.

[95] T. Good and M. Benaissa. Very small FPGA application-specific instruction processor for AES. *IEEE Transactions on Circuits and Systems I: Regular Papers*, (7):1477–1486, July 2006.

[96] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 137–147. ACM, 2003.

[97] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. Int. Conf. on Architectural support for programming languages and operating systems (ASPLOS)*, pages 151–162. ACM, 2006.

[98] V. Govindaraju, C. H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim. DySER: unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, (5):38–51, Sept 2012.

[99] Mariusz Grad and Christian Plessl. Woolcano: An architecture and tool flow for dynamic instruction set extension on Xilinx Virtex-4 FX. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, 2009.

[100] Mariusz Grad and Christian Plessl. On the feasibility and limitations of just-in-time instruction set extension for FPGA-based reconfigurable processors. *Int. Journal of Reconfigurable Computing (IJRC)*, 2012.

[101] David Grant, Chris Wang, and Guy G.F. Lemieux. A CAD framework for Malibu: An FPGA with time-multiplexed coarse-grained elements. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 123–132. ACM, 2011.

[102] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in LLVM. In *Proc. Int. Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2011.

[103] Shay Gueron. Advanced encryption standard (AES) instructions set. `http://softwarecommunity.intel.com/articles/eng/3788.htm`, 2008.

[104] Tim Güneysu and Christof Paar. Ultra high performance ECC over NIST primes on commercial FPGAs. In *Proc. Int. Conf. on Cryptographic Hardware and Embedded Systems (CHES)*, pages 62–78. Springer, 2008.

[105] Vaibhav Gupta, Debabrata Mohapatra, Sang Phill Park, Anand Raghunathan, and Kaushik Roy. IMPACT: Imprecise adders for low-power approximate computing. In *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED)*, pages 409–414. IEEE, 2011.

[106] 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 *Int. Symp. on Workload Characterization (IISWC)*, pages 3–14, Dec 2001.

[107] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, (4):287–317, December 1983.

[108] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 37–47. ACM, 2010.

[109] J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Proc. IEEE European Test Symposium (ETS)*, pages 1–6, May 2013.

[110] Markus Happe, Friedhelm Meyer auf der Heide, Peter Kling, Marco Platzner, and Christian Plessl. On-the-fly computing: A novel paradigm for individualized IT services. In *Proc. Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*. IEEE Computer Society, June 2013.

[111] David Harris and Sarah Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann Publishers, 2nd edition, 2012.

[112] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 642–649. IEEE, 2001.

[113] Reiner Hartenstein. The digital divide of computing. In *Proc. Int. Conf. on Computing Frontiers*, pages 357–362, 2004.

[114] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, (2):206–217, Feb 2004.

[115] Nicole Hemsoth. Genomics marks the next sequence for FPGAs. `http://www.nextplatform.com/2015/10/29/the-next-sequence-for-fpgas-is-in-genomics/`, October 2015.

[116] Nicole Hemsoth. Inside the GPU clusters that power Baidu's neural networks. `http://www.nextplatform.com/2015/12/11/inside-the-gpu-clusters-that-power-baidus-neural-networks/`, December 2015.

[117] Jörg Henkel and Rolf Ernst. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, (2):273–290, 2001.

[118] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[119] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, (4):1–17, September 2006.

[120] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED)*, pages 38–43, August 2007.

[121] M.C. Herbordt, T. Van Court, Yongfeng Gu, B. Sukhwani, A Conti, J. Model, and D. DiSabello. Achieving high performance with FPGA-based computing. *IEEE Computer*, (3):50–57, March 2007.

[122] A.J.G. Hey. Experience with MIMD message-passing systems: Towards general purpose parallel computing. In Pierre America, editor, *Parallel Database Systems*, pages 99–111. Springer, 1991.

[123] H. Hirschmüller and D. Scharstein. Evaluation of cost functions for stereo matching. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8. IEEE Computer Society, June 2007.

[124] Heiko Hirschmüller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, (2):328–341, February 2008.

[125] Heiko Hirschmüller and Daniel Scharstein. Evaluation of stereo matching costs on images with radiometric differences. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, (9):1582–1599, September 2009.

[126] Brian Holland, Karthik Nagarajan, and Alan D. George. RAT: RC amenability test for rapid performance prediction. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, (4):1–31, 2009.

[127] M. Horowitz. Computing's energy problem (and what we can do about it). In *IEEE Int. Solid-State Circuits Conf. Digest of Tech. Papers (ISSCC)*, pages 10–14, February 2014.

[128] HSA Foundation. HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG). `http://www.hsafoundation.com/?ddownload=4945`, July 2015.

[129] Paul Hsieh. 7th generation CPU comparisons. `http://www.azillionmonkeys.com/qed/cpujihad.shtml`, August 1999.

[130] Zhigang Hu, Alper Buyuktosunoglu, Viji Srinivasan, Victor Zyuban, Hans Jacobson, and Pradip Bose. Microarchitectural techniques for power gating of execution units. In *Proc. Int. Symp. on Low Power Electronics and Design (ISLPED)*, pages 32–37, 2004.

[131] Xuejue Huang, Wen-Chin Lee, C. Kuo, D. Hisamoto, Leland Chang, J. Kedzierski, E. Anderson, H. Takeuchi, Yang-Kyu Choi, K. Asano, V. Subramanian, Tsu-Jae King, J. Bokor, and Chenming Hu. Sub-50 nm p-channel FinFET. *IEEE Transactions on Electron Devices*, (5):880–886, May 2001.

[132] Richard Hughey. Parallel hardware for sequence comparison and alignment. *Bioinformatics*, (6):473–479, September 1996.

[133] Eddie Hung and Steven J.E. Wilton. Towards simulator-like observability for FPGAs: A virtual overlay network for trace-buffers. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 19–28. ACM, 2013.

[134] IDC. Prognose zum Absatz von Tablets, PCs und Smartphones weltweit von 2010 bis 2020 (in Millionen Stück). `http://de.statista.com/statistik/daten/studie/256337/umfrage/prognose-zum-weltweiten-absatz-von-tablets-pcs-und-smartphones/`, 2016.

[135] Amer Ihab. Introducing the video coding engine (VCE). `http://developer.amd.com/community/blog/2014/02/19/introducing-video-coding-engine-vce/`, February 2014.

[136] Intel. Intel 64 and IA-32 architectures optimization reference manual. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`, January 2016.

[137] International Roadmap Committee (IRC). The international technology roadmap for semiconductors (ITRS), 2011 edition. `http://www.semiconductors.org/main/2011_international_technology_roadmap_for_semiconductors_itrs/`, 2011.

[138] International Roadmap Committee (IRC). The international technology roadmap for semiconductors (ITRS), 2013 edition. `http://www.semiconductors.org/main/2013_international_technology_roadmap_for_semiconductors_itrs/`, 2013.

[139] International Roadmap Committee (IRC). The international technology roadmap for semiconductors (ITRS) 2.0, 2015 edition. `http://www.semiconductors.org/main/2015_international_technology_roadmap_for_semiconductors_itrs/`, 2016.

[140] C. Iseli and E. Sanchez. Spyder: a reconfigurable VLIW processor using FPGAs. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–24, Apr 1993.

[141] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. Mercury BLASTP: Accelerating protein sequence alignment. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, (2):9:1–9:44, June 2008.

[142] A.K. Jain, S.A. Fahmy, and D.L. Maskell. Efficient overlay architecture based on DSP blocks. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 25–28. IEEE Computer Society, May 2015.

[143] M. K. Jain, M. Balakrishnan, and A. Kumar. ASIP design methodologies: survey and issues. In *Proc. Int. Conf. on VLSI Design*, pages 76–81, 2001.

[144] Minxi Jin and T. Maruyama. A fast and high quality stereo matching algorithm on FPGA. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 507–510. IEEE Computer Society, August 2012.

[145] Minxi Jin and Tsutomu Maruyama. Fast and accurate stereo vision system on FPGA. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, (1):3:1–3:24, February 2014.

[146] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation (PLDI)*, pages 171–185, 1994.

[147] Chetana N Keltcher, Kevin J McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, (2):66–76, 2003.

[148] K. Kępa, R. Soni, and P. Athanas. Inferring custom architectures from OpenCL. In *Proc. Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 9–16, Sept 2015.

[149] S. Kestur, J. D. Davis, and O. Williams. BLAS comparison on FPGA, CPU and GPU. In *IEEE Computer Society Annual Symp. on VLSI (ISVLSI)*, pages 288–293, July 2010.

[150] Seonggun Kim and Hwansoo Han. Efficient SIMD code generation for irregular kernels. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 55–64. ACM, 2012.

[151] Jeffrey Kingyens and J. Gregory Steffan. The potential for a GPU-like overlay architecture for FPGAs. *Int. Journal of Reconfigurable Computing (IJRC)*, 2011.

[152] Laszlo B Kish. End of Moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, (3–4):144 – 149, 2002.

[153] Dmitrij Kissler, Frank Hannig, Alexey Kupriyanov, and Jürgen Teich. A dynamically reconfigurable weakly programmable processor array architecture template. In *Proc. Int. Workshop on Reconfigurable Communication Centric System-on-Chips (ReCoSoC)*, pages 31–37, 2006.

[154] Dirk Koch, Christian Beckhoff, and Guy G. F. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE Computer Society, 2013.

[155] Dirk Koch, Christian Beckhoff, and Jim Torresen. Zero logic overhead integration of partially reconfigurable modules. In *Int. Symp. on Integrated Circuits and Systems Design (SBCCI)*, pages 103–108. ACM, 2010.

[156] Dimitri Komatitsch, Gordon Erlebacher, Dominik Göddeke, and David Michéa. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster. *Journal of Computational Physics*, (20):7692 – 7714, 2010.

[157] P. Kristof, H. Yu, Z. Li, and X. Tian. Performance study of SIMD programming models on Intel multicore processors. In *Proc. Int. Symp. on Parallel and Distributed Processing Workshops (IPDPSW)*, pages 2423–2432, May 2012.

[158] Konstantinos Krommydas, Wu-chun Feng, Christos D. Antonopoulos, and Nikolaos Bellas. OpenDwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems*, pages 1–20, 2015.

[159] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction. In *Proc. Int. Symp. on Microarchitecture (MICRO)*, pages 81–92, December 2003.

[160] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA – a cost-optimized parallel code breaker. In *Proc. Int. Conf. on Cryptographic Hardware and Embedded Systems (CHES)*, pages 101–118. Springer, 2006.

[161] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 21–30. ACM, 2006.

[162] Petteri Laakso. Importance of programmability. `https://webcache.googleusercontent.com/search?q=cache:WCB_luCXgGoJ:https://www.vectorfabrics.com/de/blog/item/importance_of_programmability+&cd=1&hl=en&ct=clnk&gl=de`, June 2014.

[163] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, pages 75–86. IEEE Computer Society, March 2004.

[164] I. Lebedev, Shaoyi Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, Mingjie Lin, and J. Wawrzynek. MARC: a many-core approach to reconfigurable computing. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, pages 7–12, December 2010.

[165] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. Int. Symp. on Microarchitecture (MICRO)*, pages 330–335. IEEE Computer Society, 1997.

[166] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 451–460, 2010.

[167] Colin Yu Lin and Hayden Kwok-Hay Kwok-Hay So. Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture synthesis. *ACM SIGARCH Computer Architecture News*, (5):58–63, March 2012.

[168] Jason Luu, Jeff Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Norrudin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, (2):6:1–6:30, June 2014.

[169] Roman Lysecky, Kris Miller, Frank Vahid, and Kees Vissers. Firm-core virtual FPGA for just-in-time FPGA compilation. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, 2005.

[170] S. Ma, Z. Aklah, and D. Andrews. Run time interpretation for creating custom accelerators. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 900–905, March 2016.

[171] C. Mack. The multiple lives of Moore's law. *IEEE Spectrum*, (4):31–31, April 2015.

[172] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. An evaluation of vectorizing compilers. In *Proc. Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 372–382. IEEE Computer Society, 2011.

[173] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, (7):78–89, July 2012.

[174] S. Mattoccia. Fast locally consistent dense stereo on multicore. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 69–76. IEEE Computer Society, June 2010.

[175] Maxeler Technologies. MPC-C series. `https://www.maxeler.com/products/mpc-cseries/`.

[176] Maxeler Technologies. Programming MPC systems. Whitepaper, `https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf`, June 2013.

[177] Clive Maxfield. AutoESL acquisition a great move for Xilinx. `http://www.eetimes.com/author.asp?section_id=36&doc_id=1284904`, January 2011.

[178] Max Maxfield. With over 1 billion FPGAs sold, Lattice introduces MachXO3 family. `http://www.eetimes.com/document.asp?doc_id=1319597`, March 2013.

[179] Bingfeng Mei, F. J. Veredas, and B. Masschelein. Mapping an H.264/AVC decoder onto the ADRES reconfigurable architecture. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 622–625, Aug 2005.

[180] Bingfeng Mei, Serge Vernalde, Diederik Verkest, and Rudy Lauwereins. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 1–6. IEEE Computer Society, 2004.

[181] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 61–70. Springer, 2003.

[182] X. Mei, X. Sun, M. Zhou, S. Jiao, H. Wang, and X. Zhang. On building an accurate stereo matching system on graphics hardware. In *Proc. ICCV Workshop on GPU in Computer Vision Applications (GPUCV)*. IEEE, 2011.

[183] Francisco J Mesa-Martinez et al. SCOORE Santa Cruz out-of-order RISC engine, FPGA design issues. In *Proc. Workshop on Architectural Research Prototyping (WARP)*, pages 61–70, 2006.

[184] Björn Meyer, Jörn Schumacher, Christian Plessl, and Jens Förstner. Convey vector personalities – FPGA acceleration with an OpenMP-like programming effort? In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*. IEEE Computer Society, August 2012.

[185] Friedhelm Meyer auf der Heide and al. On-The-Fly computing, Individualisierte IT-Dienstleistungen in dynamischen Märkten, Finanzierungsantrag, December 2010.

[186] Friedhelm Meyer auf der Heide and al. On-The-Fly computing, Individualisierte IT-Dienstleistungen in dynamischen Märkten, Finanzierungsantrag, December 2014.

[187] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. Towards characterizing cloud backend workloads: Insights from Google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, (4):34–41, March 2010.

[188] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, (1):82–85, Jan 1998. Reprinted from Gordon E. Moore, "Cramming More Components onto Integrated Circuits," Electronics, pp. 114–117, April 19, 1965.

[189] Valentin Mena Morales, Pierre-Henri Horrein, Amer Baghdadi, Erik Hochapfel, and Sandrine Vaton. Energy-efficient FPGA implementation for binomial option pricing using OpenCL. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 208:1–208:6. European Design and Automation Association, 2014.

[190] Timothy Prickett Morgan. Looking ahead in the datacenter with intel. `http://www.nextplatform.com/2015/12/10/looking-ahead-in-the-datacenter-with-intel/`, December 2015.

[191] Timothy Prickett Morgan. Microsoft extends FPGA reach from Bing to deep learning. `http://www.nextplatform.com/2015/08/27/microsoft-extends-fpga-reach-from-bing-to-deep-learning/`, August 2015.

[192] Timothy Prickett Morgan. Traders bank on server switch hybrids for performance boost. `http://www.nextplatform.com/2015/04/06/traders-bank-on-server-switch-hybrids-for-performance-boost/`, April 2015.

[193] Timothy Prickett Morgan. The long future ahead for intel xeon processors. `http://www.nextplatform.com/2016/05/05/long-future-ahead-intel-xeon-processors/`, May 2016.

[194] Andreas Moshovos, Prithviraj Banerjee, Scott Hauck, and Zhi Alex Ye. CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 225–235. IEEE Computer Society, 2000.

[195] Ananya Muddukrishna, Peter A. Jonsson, Vladimir Vlassov, and Mats Brorsson. Locality-aware task scheduling and data distribution on numa systems. In Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller, editors, *Proc. Int. Workshop on OpenMP (IWOMP)*, pages 156–170. Springer Berlin Heidelberg, 2013.

[196] Aaftab Munshi. *The OpenCL Specification*. Khronos OpenCL Working Group, December 2008.

[197] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proc. Design Automation Conference (DAC)*, pages 174:1–174:9, 2013.

[198] Y. Nakamura and K. Hiraki. Highly fault-tolerant FPGA processor by degrading strategy. In *Proc. Pacific Rim Int. Symp. on Dependable Computing (PRDC)*, pages 75–78, Dec 2002.

[199] Nallatech. Intel Xeon FSB FPGA Socket Fillers, September 2010. http://www.nallatech.com/intel-xeon-fsb-fpga-socket-fillers.html.

[200] Viet Nhu Ngo. *Parallel loop transformation techniques for vector-based multiprocessor systems*. PhD thesis, 1995. UMI Order No. GAX94-33091.

[201] J. Nickolls and W.J. Dally. The GPU computing era. *Proc. Int. Symp. on Microarchitecture (MICRO)*, (2):56–69, March 2010.

[202] Dorit Nuzman and Ayal Zaks. Outer-loop vectorization: revisited for short SIMD architectures. In *Proc. Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 2–11. ACM, 2008.

[203] Stephen L. Olivier and Jan F. Prins. Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In Matthias S. Müller, Bronis R. de Supinski, and Barbara M. Chapman, editors, *Proc. Int. Workshop on OpenMP (IWOMP)*, pages 63–78. Springer Berlin Heidelberg, 2009.

[204] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

[205] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, (1):80–113, 2007.

[206] Karen Parnell. Could microprocessor obsolescence be history? 2003.

[207] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, (6):25–33, October 1980.

[208] David A. Patterson and Carlo H. Sequin. RISC I: A reduced instruction set VLSI computer. In *Proc. Int. Symp. on Computer Architecture (ISCA)*, pages 443–457. IEEE Computer Society Press, 1981.

[209] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, (4):42–50, August 1996.

[210] Christian Plessl and Marco Platzner. Zippy – a coarse-grained reconfigurable array with support for hardware virtualization. In *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 213–218. IEEE Computer Society, July 2005.

[211] Peter Poplavko, Twan Basten, and Jef van Meerbergen. Execution-time prediction for dynamic streaming applications with task-level parallelism. In *Proc. Euromicro Conf. on Digital System Design Architectures, Methods and Tools (DSD)*, pages 228–235. IEEE Computer Society, 2007.

[212] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/`, 2012.

[213] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, (3):10–22, May 2015.

[214] Vidya Rajagopalan, V Boppana, S Dutta, B Taylor, and R Wittig. Xilinx Zynq-7000 EPP – an extensible processing platform family. In *Hot Chips Symposium*, pages 1352–1357, 2011.

[215] A. M. Rashid, B. Kuhn, B. Arbab, and D. Kuck. PC design, use, and purchase relations. In *Int. Symp. on Workload Characterization (IISWC)*, pages 140–149, Oct 2015.

[216] M. Reshadi, B. Gorjiara, and D. Gajski. Utilizing horizontal and vertical parallelism with a no-instruction-set compiler for custom datapaths. In *Proc. Int. Conf. on Computer Design (ICCD)*, pages 69–74, October 2005.

[217] Mathieu Rosière, Jean-lou Desbarbieux, Nathalie Drach, and Franck Wajsbürt. An out-of-order superscalar processor on FPGA: The reorder buffer design. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 1549–1554. EDA Consortium, 2012.

[218] Hans-Peter Rosinger. Xilinx whitepaper: Connecting customized IP to the MicroBlaze soft processor using the fast simplex link (FSL) channel. `http://tec.icbuy.com/uploads/2010/8/3/xapp529.pdf`, May 2004.

[219] K. Rupp and S. Selberherr. The economic limit to Moore's law [point of view]. *Proceedings of the IEEE*, (3):351–353, March 2010.

[220] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter. Understanding portability of a high-level programming model on contemporary heterogeneous architectures. *IEEE Micro*, (4):48–58, July 2015.

[221] Randolf G Scarborough and Harwood G Kolsky. A vectorizing Fortran compiler. *IBM Journal of Research and Development*, (2):163–171, March 1986.

[222] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. Journal on Computer Vision*, (1-3):7–42, April 2002.

[223] Tamara Schmitz. The rise of serial memory and the future of DDR. Xilinx Whitepaper: UltraScale Devices, `http://www.xilinx.com/support/documentation/white_papers/wp456-DDR-serial-mem.pdf`, March 2015.

[224] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, (3):18:1–18:15, August 2008.

[225] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Proc. Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 29–40. IEEE, February 2002.

[226] A. Severance and G. G. F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES/ISSS)*, pages 1–10, Sept 2013.

[227] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. Soft vector processors with streaming pipelines. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 117–126. ACM, 2014.

[228] Aaron Severance and Guy Lemieux. VENICE: a compact vector processor for FPGA applications. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 245–252. IEEE Computer Society, 2012.

[229] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In José M. Laginha M. Palma, Michel Daydé, Osni Marques, and João Correia Lopes, editors, *Proc. Int. Meeting on High Performance Computing for Computational Science (VECPAR)*, pages 1–25. Springer Berlin Heidelberg, 2011.

[230] Yi Shan, Yuchen Hao, Wenqiang Wang, Yu Wang, Xu Chen, Huazhong Yang, and Wayne Luk. Hardware acceleration for an accurate stereo vision system using mini-census adaptive support region. *ACM Transactions on Embedded Computing Systems (TECS)*, (4s):132:1–132:24, April 2014.

[231] Y. S. Shao, B. Reagen, Gu-Yeon Wei, and D. Brooks. The aladdin approach to accelerator design and modeling. *IEEE Micro*, (3):58–70, May 2015.

[232] H. Sharangpani and H. Arora. Itanium processor microarchitecture. *IEEE Micro*, (5):24–43, September 2000.

[233] Anand Lal Shimpi. The Sandy Bridge review: Intel Core i7-2600K, i5-2500K and Core i3-2100 tested. `http://www.anandtech.com/show/4083/the-sandy-bridge-review-intel-core-i7-2600k-i5-2500k-core-i3-2100-tested`, January 2011.

[234] Sunil Shukla, Neil W. Bergmann, and Jürgen Becker. QUKU: A two-level reconfigurable architecture. In *Proc. IEEE Symp. on Emerging VLSI Technologies and Architectures (ISVLSI)*, pages 1–6. IEEE Computer Society, 2006.

[235] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, (7587):484–489, 01 2016.

[236] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers (TC)*, (5):465–481, May 2000.

[237] Scott Sirowy and Alessandro Forin. Where's the beef? Why FPGAs are so fast. Technical report, Microsoft Research, September 2008.

[238] T. Skotnicki, J.A. Hutchby, Tsu-Jae King, H.-S.P. Wong, and F. Boeuf. The end of CMOS scaling: toward the introduction of new materials and structural changes to improve MOSFET performance. *IEEE Circuits and Devices Mag.*, (1):16–26, Jan 2005.

[239] Ryan Smith. The NVIDIA GeForce GTX Titan X review. `http://www.anandtech.com/show/9059/the-nvidia-geforce-gtx-titan-x-review`, March 2015.

[240] Ryan Smith. HSA 1.1 specification launched: Extending HSA to more vendors & processor types. `http://www.anandtech.com/show/10387/hsa-11-specification-launched-multi-vendor-support`, May 2016.

[241] Simon A. Spacey, Wayne Luk, Paul H. J. Kelly, and Daniel Kuhn. Rapid design space visualization through hardware/software partitioning. In *Proc. Southern Programmable Logic Conference (SPL)*, pages 159–164. IEEE, April 2009.

[242] W. J. Starke, J. Stuecheli, D. M. Daly, J. S. Dodson, F. Auernhammer, P. M. Sagmeister, G. L. Guthrie, C. F. Marino, M. Siegel, and B. Blaner. The cache and memory subsystems of the IBM POWER8 processor. *IBM Journal of Research and Development*, (1):3:1–3:13, Jan 2015.

[243] Greg Stitt and Lu Hao. Virtual finite-state-machine architectures for fast compilation and portability. In *Proc. IEEE Int. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 91–94. IEEE Computer Society, 2013.

[244] Greg Stitt and Frank Vahid. Hardware/software partitioning of software binaries. In *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*, pages 164–170. IEEE, November 2002.

[245] Greg Stitt and Frank Vahid. A decompilation approach to partitioning software for microprocessor/FPGA platforms. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, pages 396–397. IEEE Computer Society, 2005.

[246] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. CAPI: a coherent accelerator processor interface. *IBM Journal of Research and Development*, (1):7:1–7:7, Jan 2015.

[247] B. Sukhwani and M. C. Herbordt. FPGA acceleration of rigid-molecule docking codes. *IET Computers & Digital Techniques*, (3):184–195, May 2010.

[248] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, (3), 2005.

[249] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, (7):54–62, September 2005.

[250] M. Suzuki, Y. Hasegawa, Y. Yamada, N. Kaneko, K. Deguchi, H. Amano, K. Anjo, M. Motomura, K. Wakabayashi, T. Toi, and T. Awashima. Stream applications on the dynamically reconfigurable processor. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 137–144, Dec 2004.

[251] Michael Bedford Taylor. Bitcoin and the age of bespoke silicon. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 16:1–16:10. IEEE Press, 2013.

[252] The Joint Task Force for Computing Curricula 2005. *Computing Curricula 2005, The Overview Report.* ACM, 2005.

[253] Stefan Tillich and Johann Großschädl. Instruction set extensions for efficient AES implementation on 32-bit processors. In *Proc. Int. Conf. on Cryptographic Hardware and Embedded Systems (CHES)*, pages 270–284. Springer, 2006.

[254] Beau Tippetts, Dah Jye Lee, Kirt Lillywhite, and James K. Archibald. Hardware-efficient design of real-time profile shape matching stereo vision algorithm on FPGA. *Int. Journal of Reconfigurable Computing (IJRC)*, 2014.

[255] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proc. Int. Symp. on Parallel and Distributed Processing (IPDPS)*, pages 1–12, May 2009.

[256] Tiffany Trader. Programmability matters. `http://www.hpcwire.com/2014/06/30/programmability-matters/`, June 2014.

[257] J. Vasiljevic, R. Wittig, P. Schumacher, J. Fifield, F. M. Vallina, H. Styles, and P. Chow. OpenCL library of stream memory components targeting FPGAs. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 104–111, December 2015.

[258] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers (TC)*, (11):1363–1375, Nov 2004.

[259] Olga Veksler. Fast variable window for stereo correspondence using integral images. In *Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 556–561. IEEE Computer Society, 2003.

[260] Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. MACACO: Modeling and analysis of circuits for approximate computing. In *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD)*, pages 667–673. IEEE, 2011.

[261] Francisco-Javier Veredas, M. Scheppler, W. Moffat, and Bingfeng Mei. Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 106–111, Aug 2005.

[262] Werner Vogels. Eventually consistent. *Communications of the ACM*, (1):40–44, January 2009.

[263] John von Neumann. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, (4):27–75, October 1993. Reprinted from John von Neumann, "First Draft of a Report on the EDVAC", Moore School of Electrical Engineering, University of Pennsylvania, June 1945.

[264] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proc. Int. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, pages 127–136, 2012.

[265] Wenqiang Wang, Jing Yan, Ningyi Xu, Yu Wang, and Feng-Hsiung Hsu. Real-time high-quality stereo vision system in FPGA. In *Proc. Int. Conf. on Field Programmable Technology (ICFPT)*, pages 358–361. IEEE Computer Society, December 2013.

[266] Matthew A. Watkins and David H. Albonesi. ReMAP: a reconfigurable heterogeneous multicore architecture. In *Proc. Int. Symp. on Microarchitecture (MICRO)*, pages 497–508. IEEE Computer Society, 2010.

[267] C. M. Weber, C. N. Berglund, and P. Gabella. Mask cost and profitability in photomask manufacturing: An empirical analysis. *IEEE Transactions on Semiconductor Manufacturing*, (4):465–474, November 2006.

[268] K. Wegner and O. Stankiewicz. Similarity measures for depth estimation. In *Proc. 3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-Con)*, pages 1–4. IEEE, May 2009.

[269] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *Proc. Int. Conf. on High Performance Computing (HIPC)*. IEEE, December 2014.

[270] Stephen Weston, James Spooner, Sébastien Racanière, and Oskar Mencer. Rapid computation of value and risk for derivatives portfolios. *Concurrency and Computation: Practice and Experience*, (8):880—894, June 2011.

[271] Wikipedia. Computer. https://en.wikipedia.org/wiki/Computer, 2015.

[272] S. Wong, T. van As, and G. Brown. ρ-VEX: A reconfigurable and extensible softcore VLIW processor. In *Proc. Int. Conf. on Field Programmable Technology (ICFPT)*, pages 369–372, Dec 2008.

[273] Nathan Woods. Integrating FPGAs in high-performance computing: The architecture and implementation perspective. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 132–132, 2007.

[274] Xilinx. Virtex-5 family overview, DS100 (v5.0). `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, February 2009.

[275] Xilinx. Virtex-6 family overview, DS150 (v2.4). `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`, January 2012.

[276] Xilinx. Xilinx Inc., form 10-k annual report 2012, May 2012.

[277] Xilinx. 7 Series FPGAs overview, DS180 (v1.16.1). `http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf`, December 2014.

[278] Xilinx. Xilinx Inc., form 10-k annual report 2015, May 2015.

[279] XtremeData. XD2000F FPGA In-socket Accelerator for AMD Socket F, September 2010. http://www.xtremedata.com/images/pdf/xd2000f_product_flyer.pdf.

[280] XtremeData. XD2000i FPGA In-Socket Accelerator for Intel FSB, September 2010. http://www.xtremedata.com/images/pdf/xd2000i_product_flyer.pdf.

[281] L. Yang, R. T. P. Lee, S. S. P. Rao, W. Tsai, and P. D. Ye. 10 nm nominal channel length MoS_2 FETs with EOT 2.5 nm and 0.52 mA/μm drain current. In *Device Research Conf. (DRC)*, pages 237–238, June 2015.

[282] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Fine-grain performance scaling of soft vector processors. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 97–106, 2009.

[283] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. ACM, 2013.

[284] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. Vector processing as a soft processor accelerator. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, (2):12:1–12:34, June 2009.

[285] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector processing as a soft-core CPU accelerator. In *Proc. Int. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 222–232. ACM, 2008.

[286] Ke Zhang, Jiangbo Lu, and Gauthier Lafruit. Cross-based local stereo matching using orthogonal integral images. *IEEE Transactions on Circuits and Systems for Video Technology (TCSVT)*, (7):1073–1079, July 2009.

[287] Peiheng Zhang, Guangming Tan, and Guang R. Gao. Implementation of the Smith-Waterman algorithm on a reconfigurable supercomputing platform. In *Proc. Int. Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA)*, pages 39–48, 2007.

[288] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 145–156. ACM, 2002.