# Utilizing Heterogeneity in Manycore Architectures for Streaming Applications

Süleyman Savas

Supervisors:
Tomas Nordström
Zain Ul-Abdin

LICENTIATE THESIS | Halmstad University Dissertations no. 29

Utilizing Heterogeneity in Manycore Architectures for Streaming Applications
© Süleyman Savas
Halmstad University Dissertations no. 29
ISBN 978-91-87045-60-8 (printed)
ISBN 978-91-87045-61-5 (pdf)
Publisher: Halmstad University Press, 2017 | www.hh.se/hup
Printer: Media-Tryck, Lund

# Abstract

In the last decade, we have seen a transition from single-core to manycore in computer architectures due to performance requirements and limitations in power consumption and heat dissipation. The first manycores had homogeneous architectures consisting of a few identical cores. However, the applications, which are executed on these architectures, usually consist of several tasks requiring different hardware resources to be executed efficiently. Therefore, we believe that utilizing heterogeneity in manycores will increase the efficiency of the architectures in terms of performance and power consumption. However, development of heterogeneous architectures is more challenging and the transition from homogeneous to heterogeneous architectures will increase the difficulty of efficient software development due to the increased complexity of the architecture. In order to increase the efficiency of hardware and software development, new hardware design methods and software development tools are required.

This thesis studies manycore architectures in order to reveal possible uses of heterogeneity in manycores and facilitate choice of architecture for software and hardware developers. It defines a taxonomy for manycore architectures that is based on the levels of heterogeneity they contain and discusses benefits and drawbacks of these levels. Additionally, it evaluates several applications, a dataflow language (CAL), a source-to-source compilation framework (Cal2Many), and a commercial manycore architecture (Epiphany). The compilation framework takes implementations written in the dataflow language as input and generates code targetting different manycore platforms. Based on these evaluations, the thesis identifies the bottlenecks of the architecture. It finally presents a methodology for developing heterogeneoeus manycore architectures which target specific application domains.

Our studies show that using different types of cores in manycore architectures has the potential to increase the performance of streaming applications. If we add specialized hardware blocks to a core, the performance easily increases by 15x for the target application while the core size increases by 40-50% which can be optimized further. Other results prove that dataflow languages, together

with software development tools, decrease software development efforts significantly (25-50%) while having a small impact (2-17%) on the performance.

# Acknowledgements

I would like to express my appreciation to my supervisors Tomas Nordström and Zain-Ul-Abdin for their guidance. I would also like to thank to my support committee members Bertil Svensson and Thorstein Rögnvaldsson for their valuable comments on my research. I want to express my thanks to Stefan Byttner for his support as the study coordinator.

My colleagues Essayas, Sebastian, Amin and Erik deserve special thanks for all the discussions and suggestions. I would like to thank to all my other colleagues and Halmstad Research Students' Society (HRSS) members. Last but not least I want to thank to Merve for her support.

# List of Publications

This thesis summarizes the following publications.

**Paper I** Süleyman Savas, Essayas Gebrewahid, Zain Ul-Abdin, Tomas Nordström, and Mingkun Yang. "An evaluation of code generation of dataflow languages on manycore architectures", In *Proceedings of IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '16)*, pp. 1-9. IEEE, 20-22 Aug, 2014, Chongqing, China.

**Paper II** Süleyman Savas, Sebastian Raase, Essayas Gebrewahid, Zain Ul-Abdin, and Tomas Nordström. "Dataflow Implementation of QR Decomposition on a Manycore", In *Proceedings of the Fourth ACM International Workshop on Many-core Embedded Systems (MES '16)*, ACM, 18-22 June, 2016, Seoul, South Korea.

**Paper III** Süleyman Savas, Erik Hertz, Tomas Nordström, and Zain Ul-Abdin. "Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis", submitted to *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2017)*, Bochum, Germany, July 3-5, 2017.

**Paper IV** Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström. "Designing Domain Specific Heterogeneous Manycore Architectures Based on Building Blocks", submitted to *27th International Conference on Field-Programmable Logic and Applications (FPL 2017)*, Ghent, Belgium, September 4-8, 2017.

# Other Publications

**Paper A** Mingkun Yang, Suleyman Savas, Zain Ul-Abdin, and Tomas Nordström. "A communication library for mapping dataflow applications on manycore architectures." In *6th Swedish Multicore Computing Workshop, MCC-2013*, pp. 65-68. November 25-26, 2013, Halmstad, Sweden.

**Paper B** Xypolitidis, Benard, Rudin Shabani, Satej V. Khandeparkar, Zain Ul-Abdin, Tomas Nordström, and Süleyman Savas. "Towards Architectural Design Space Exploration for Heterogeneous Manycores." In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pp. 805-810. IEEE, February 17-19, 2016, Heraklion Crete, Greece.

**Paper C** Sebastian Raase, Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström. "Supporting Efficient Channel-Based Communication in a Mesh Network-on-Chip" Ninth Nordic Workshop on Multi-Core Computing (MCC 2016), November 29-30, 2016, Trondheim, Norway.

**Paper D** Sebastian Raase, Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström. "Hardware Support for Communication Channels in Tiled Manycore Systems". submitted to *27th International Conference on Field-Programmable Logic and Applications(FPL 2017)*, Ghent, Belgium, September 4-8, 2017.

# Contents


**1 Introduction** **1**
- 1.1 Research Questions . . . 4
- 1.2 Methodology . . . 5
- 1.3 Contributions . . . 6

**2 Background** **9**
- 2.1 Manycore Architectures . . . 9
- 2.2 Manycore Design . . . 11
  - 2.2.1 RISC-V and Rocket Chip . . . 11
  - 2.2.2 Rocket core and RoCC Interface . . . 12
- 2.3 Software Development for Manycores . . . 12
- 2.4 Dataflow Programming Model . . . 13
- 2.5 Streaming Applications . . . 15
  - 2.5.1 Two Dimensional Inverse Cosine Transform (IDCT2D) . 15
  - 2.5.2 QR Decomposition . . . 16
  - 2.5.3 SAR and Autofocus . . . 16

**3 Heterogeneity in Manycores** **17**
- 3.1 Levels of Heterogeneity . . . 18
- 3.2 Conclusion . . . 20

**4 Designing Domain Specific Heterogeneous Manycores** **21**
- 4.1 Application analysis and hot-spot identification . . . 22
- 4.2 Custom hardware development and integration . . . 23
- 4.3 Case Study - SAR Autofocus Criterion Computation . . . 24

**5 Summary of Papers** **27**
- 5.1 Evaluation of Software Development Tools for Manycores - Paper I . . . 27
- 5.2 Parallel QR Decomposition on Epiphany Architecture - Paper II 27
- 5.3 Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis- Paper III . . . 28

5.4 Designing Domain Specific Heterogeneous Manycore Architectures Based on Building Blocks - Paper IV . . . . . 29

**6 Conclusions and Future Work** **31**
6.1 Conclusions . . . . . 31
6.2 Future Work . . . . . 32

**References** **35**

**Paper I** **43**

**Paper II** **55**

**Paper III** **63**

**Paper IV** **71**

# List of Figures


1.1 Number of cores on a single die over the last 15 years . . . . . 2
1.2 An example streaming application . . . . . 4
1.3 Layers of the methodology together with covered aspects . . . . 5

2.1 Overview of Epiphany-V manycore architecture . . . . . 10
2.2 Connections between the core and the accelerator provided by the RoCC interface. Request and response connections are separated for a clearer illustration. . . . . 12
2.3 A sample streaming application - IDCT2D block of MPEG-4 Simple Profile decoder represented as interconnected actors (the triangles are input/output ports which can be connected to any source or sink) . . . . . 14

4.1 Design flow for developing domain specific heterogeneous manycore architectures. . . . . 22
4.2 Function-call tree of autofocus application. . . . . 24
4.3 Hierarchical structure of the cubic interpolation accelerator. . . 25
4.4 Overview of the tile including rocket core and cubic interpolation accelerator. . . . . 26

# List of Tables

3.1 Main levels of heterogeneity and example commercial architectures. . . . . 20

# Chapter 1
# Introduction

Microprocessors, based on a single central processing unit, has played a significant role in rapid performance increases and cost reductions in computer applications for more than three decades. This persistent push in performance improvement allowed many application areas to appear and it facilitated software to provide better user interfaces, more functionality, and generate more accurate and useful results while solving larger and more complex problems. Once the users became accustomed to these improvements, they expected even higher performances and more improvements. However, the hardware improvement has slowed down significantly since around 2003 due to power consumption and heat dissipation issues. We can understand this 'power wall' by looking at equation 1.1 describing how the power consumption depends on capacitance of the transistor, voltage level and clock frequency [70].

$$P = CV^2f \tag{1.1}$$

Power consumption increases linearly with respect to the frequency. In order to compensate the increase in frequency, voltage levels have been decreased. However, there are limitations on voltage scaling such as circuit delay and leakage energy. If the voltage level is decreased beyond a threshold, circuit delay increases and leakage energy dominates any reduction in switching energy [20, 43]. Therefore, power consumption and the resulting heat dissipation limit the increase of clock frequency.

Despite the limitations, the performance expectations kept increasing and today they still increase due to many reasons such as increased data size to be processed and computation requirements of applications. Applications of cyber-physical, signal processing, vision/image processing, machine learning, health monitoring systems together with modeling, simulation and many other scientific applications require massive amount of computation power. Some of these applications are targeted to be executed on embedded devices with limited power budget such as battery driven devices. Thus, they require not only high performance but also low power. In order to satisfy the requirements

of these applications, facilitate their advancement and push the technology forward, hardware improvements need to continue.

When the power wall was hit, it was not possible to increase the clock frequency due to power and thermal limits. Then the microprocessor manufacturers began to switch to architectures with multiple processing cores on the same chip in order to perform several tasks concurrently and continue increasing the hardware performance. The first architectures, consisting of 2-8 cores, were called as 'multicores' by the industry, then Asanovic et al. [3] introduced 'manycores' to represent architectures with hundreds of cores. Figure 1.1 [10] shows the changes in the number of cores on a single die.

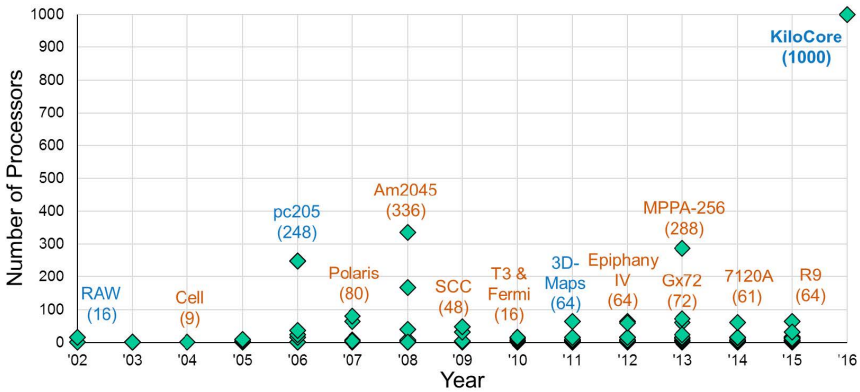


**Figure 1.1:** Number of cores on a single die over the last 15 years

The first multicores were produced by duplicating the processing cores which resulted in having identical cores on the same die, in other words homogeneous architectures. However, the tasks performed on these cores are not necessarily identical and each task might require different hardware resources to be executed efficiently. Therefore, we believe that in order to take the computer architectures one step further in performance and energy efficiency, heterogeneity, which combines different types of cores on the same die, needs to be utilized.

We define a heterogeneous computer architecture as an architecture with two or more cores which have different implementations or different complete instruction sets on the same chip. Heterogeneity in manycores comes in different forms. In simplest form, lightly heterogeneous architectures typically consist of processors using the same instruction set architecture (ISA) with different implementations [46] each resulting in different performance and power usage. On the other end, highly heterogeneous architectures consist of a large number of customized processing elements with different ISAs utilizing dark silicon [22] concept to reduce power consumption and manage thermal issues

[30]. Dark silicon is the part of the chip that cannot be powered on at nominal voltage levels due to thermal limitations. This particular thermal problems is due to the difference between transistor scaling and voltage scaling, which began around 2006 as Dennard's scaling broke down [19]. At small transistor sizes, leakage power starts dominating and heats the chip. Therefore, it is not possible to feed power to the entire chip without overheating. According to Esmailzadeh et al. [22], the amount of dark silicon on a chip might increase up to 50-80 % when transistor sizes reach 8nm. A similar concept is dynamic voltage and frequency scaling (DVFS) [53] however, in dark silicon a part of the chip is completely turned off whereas in DVFS the voltage and/or the frequency is scaled.

Heterogeneity has been adopted by both commercial and academic architectures. A simple form of heterogeneity can be seen in ARM's big.LITTLE architecture [31] that combines two cores with different sizes and different amount of hardware resources running the same ISA. A slightly more complex form of heterogeneity is found in Cell [42] and Myriad2 [7] architectures which combine two types of cores each having its own ISA. Additionally, there are several research architectures [50, 75, 51, 64] embracing heterogeneity. However, there is a lack of knowledge about the required levels of heterogeneity for different application domains and this problem requires a huge design space to be explored. This design space contains architectural design configurations regarding the types of cores, network on chip, and structure of the on-chip memory.

Even if the architectural design space gets explored and the most efficient architectures are designed for certain applications, the challenge of developing the software for these applications efficiently will remain. This challenge has increased after introduction of parallel architectures and it keeps increasing further as heterogeneity is introduced. The hardware resources require better management due to having different performance and power characteristics and they might even require different binaries due to supporting different instruction sets. We believe that new programming models, languages and software development tools will facilitate the development of efficient software for heterogeneous manycores. For instance, streaming applications with high performance requirement such as video coding/decoding, radar signal processing, and wireless communication can be implemented efficiently via dataflow programming model [54]. In this model, nodes of a block diagram are connected to one another to express the execution flow. They can execute in parallel and express parallelism as well. The nodes can consist of a single instruction or a sequence of instructions. Figure 1.2 presents a simple streaming application which consists of six nodes each performing a different task. The arrows represent the channels between the nodes and determine the direction of the dataflow.

Exploring the design space of heterogeneity in manycores is a very challenging task that requires significant amount of time, effort and knowledge.

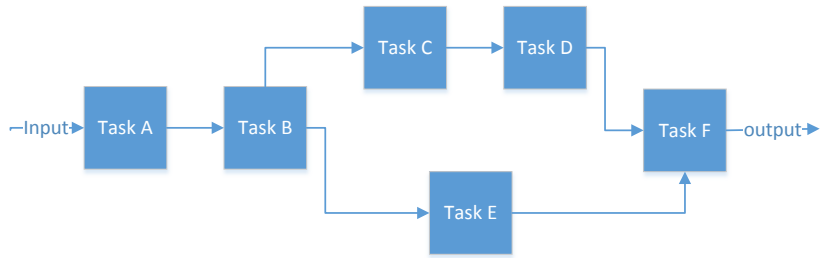


**Figure 1.2:** An example streaming application

Instead of looking for suitable heterogeneous architectures for applications, we propose to build the architectures based on the requirements of the applications. However, instead of supporting efficient execution of a single application, we aim to execute an entire domain of applications efficiently by designing the manycore architecture based on specialized building blocks. These blocks are custom hardware that execute the compute intensive parts of applications. The target architecture is a combination of many simple cores and building blocks each specialized for a different task to execute different applications efficiently. There can also be blocks which are shared between different applications.

RISC-V instruction set architecture (ISA) [73] is a good candidate for case studies while evaluating the proposed design approach due to supporting custom instructions which can be used for interacting with the accelerator. Additionally, rocket core [18], that executes RISC-V ISA, is generated via rocket chip [2] and supports an interface (rocket custom co-processor [RoCC]) for integrating custom hardware such as accelerators. Rocket chip is a RISC-V based system on chip (SoC) generator.

We have now seen the challenges of introducing heterogeneity in manycore architectures from both hardware and software design perspectives. We will now formulate the research questions related to the these challenges.

## 1.1 Research Questions

This thesis aims to cover both software and hardware aspects of developing and executing streaming applications on manycore architectures with the focus on exploiting heterogeneity of the architecture. Hence the research questions are divided into two groups. The first group of questions are about software development for manycores:

- What are the benefits of using dataflow programming model, streaming languages, and specialized software development tools while developing streaming applications?

- What are the typical characteristics of streaming applications which can cause bottlenecks in manycores?

The second group of questions are related to the hardware architecture and its development:

- How can we define different levels of heterogeneity and what are the benefits and drawbacks of these levels?
- How can we identify, develop, integrate and utilize specialized hardware blocks while designing domain specific heterogeneous manycores?

In the next section, we will present the methodology we used for finding the answers of these questions.

## 1.2 Methodology

We introduced two main problems which are the development of efficient manycore architectures, and difficulties of software development for these architectures. We believe that new programming models, new programming languages, and software development tools will facilitate the software development whereas utilizing heterogeneity will increase the efficiency of manycore architectures. Additionally, using a library of specialized hardware blocks will ease the development of heterogeneous manycore architectures.

Our approach consists of two layers as seen in Figure 1.3. The first layer focuses on software development for manycore architectures whereas the second layer focuses on manycore architecture development.

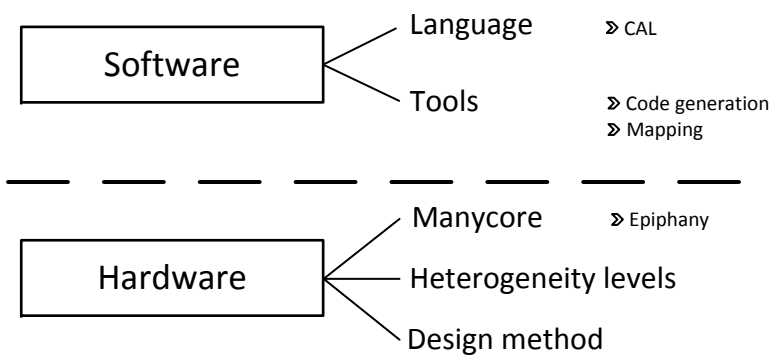


**Figure 1.3:** Layers of the methodology together with covered aspects

In the first layer, we evaluate a dataflow language and software development tools. For this purpose, we implemented several streaming applications in two different languages; the native language for a target architecture (C) and a dataflow language (CAL). The tools are used for generating code in the native language of the target architecture and mapping[1] the tasks on to the cores. These implementations are executed on the target architecture and the results are analyzed and compared.

In the second layer, we first evaluate the target manycore architecture by measuring different aspects such as the time spent on core-to-core communication and computation while executing the streaming applications. Then we focus on heterogeneity and formulate definitions for heterogeneity in manycores and its levels. Additionally, we discuss the advantages and drawbacks of the heterogeneity levels in terms of performance, power consumption, productivity and portability. Finally, we develop a method for designing domain specific heterogeneous manycore architectures. This method uses specialized hardware blocks as the building blocks of the manycores. It identifies hot-spots of target applications with static and dynamic analysis and covers integration and utilization of the blocks, which are developed based on the hot-spots. The design method is evaluated with a case study.

## 1.3 Contributions

This thesis focuses on streaming application domain due to its inherent support for parallelism. It introduces different forms of heterogeneity and evaluates software development tools which facilitate development of parallel applications. Additionally, the thesis points out the characteristics of streaming applications which can cause performance degradation. Finally it proposes a design methodology for developing domain specific heterogeneous manycore architectures.

The contributions of this thesis can be listed as:

- Definition of different heterogeneity levels and a taxonomy to classify manycore architectures.The aim is to provide guidance to hardware and software developers while choosing a manycore architecture by providing advantages and drawbacks of different heterogeneity levels.

- Implementation of parallel two dimensional inverse discrete cosine transform (2D-IDCT) on the Epiphany manycore architecture [57]. Evaluation

[1] In the target architecture, cost of library calls and copy operations dominate the core-to-core communication time and effect of the network-on-chip delay is not visible. Hence there is no significant difference between different mappings and therefore mapping results are not published.

of software development tools for manycore architectures. (Paper I)

- Implementation and evaluation of different versions of parallel QR decomposition [27] Evaluation of CAL actor language, software development tools and Epiphany architecture. (Paper II)

- Implementation and evaluation of custom hardware for single-precision floating-point division based on a novel method [34, 35] on an FPGA. (Paper III)

- A methodology for designing domain specific heterogeneous manycore architectures with specialized hardware blocks. Implementation of an accelerator based on these blocks to perform cubic interpolation on radar data. Integration of the accelerator to a RISC-V core. (Paper IV)

# Chapter 2
# Background

This chapter aims to help the reader to understand the thesis by presenting descriptions of the aspects used in the thesis, such as manycore architectures, environment used for the design method, dataflow programming model and the implemented streaming applications.

## 2.1 Manycore Architectures

Parallelism in computer architectures was introduced a long time ago, however, the form of parallelism we see today, having multiple cores on the same die, was first introduced in 2001 by IBM in POWER4 processor [65]. It became a trend after 2003 when the single core processors started to hit the power wall. The first parallel processors were built with a few identical fat cores. The number of processing cores on the same die kept increasing and nowadays it is up to hundreds.

With the number of cores increasing on the same chip, many new design options have emerged. The cores can be coupled tightly or loosely e.g. they may or may not share the caches/scratchpads. They can be placed in clusters and each cluster might have its own shared memory. Core-to-core communication methods may vary by using message passing, shared memory models or both. Network on chip might show variations in topology such as bus, ring, mesh, torus, crossbar etc. Manycore chips may have shared, distributed or hybrid memory models. These models might have variations such as centralized shared memory, distributed shared memory, partly centralized partly distributed shared memory, etc. Figure 2.1 [56] presents a manycore architecture which consists of 1024 identical cores with distributed memory. The structure of the manycore plays a significant role in the performance of executing parallel applications, however, it is not the only actor.

Manycore architectures are aimed to increase the performance of processors. However, performance improvement depends heavily on algorithms and implementations. In particular, the performance gain is limited by the fraction

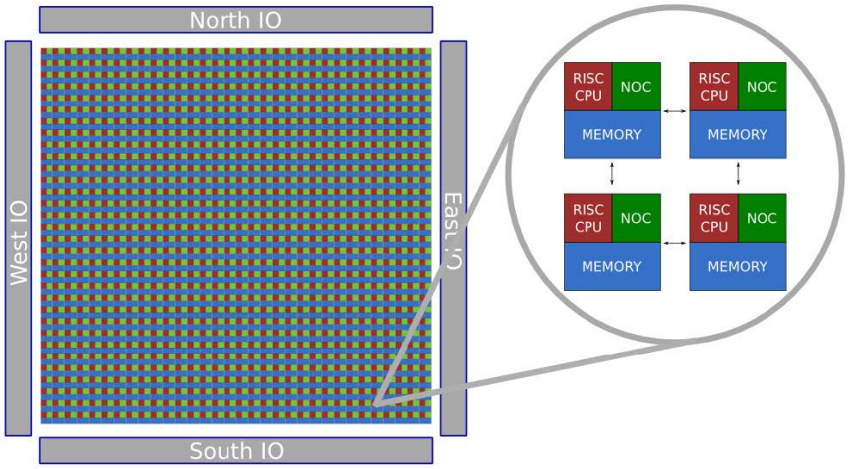


**Figure 2.1:** Overview of Epiphany-V manycore architecture

of the algorithm that can be executed on different cores in parallel. In other words, an algorithm can be as fast as its longest sequential fraction if sufficient number of cores are provided. This effect is described by Amdahl's law [36]. In the best case, the speed-up factor can be as big as the number of cores or even more if the problem is split enough to fit into the cache of each core, avoiding slower system memories. However, achieving a speed-up factor that is close to the number of cores is either extremely difficult or impossible. The developer needs to put excessive amount of time and effort in re-factoring and optimizing the algorithm/implementation. On the other hand, tailoring an application for a specific architecture makes it difficult to be executed on another architecture due to having different characteristics. In summary, manycore architectures improve the performance of processors [4] and decrease the power consumption [1], however, software productivity and portability decreases due to difficulties in parallel programming and differences in the characteristics of different architectures.

Despite the difficulties in development and programming, usage range for manycore architectures vary from radars [69] to video coders/decoders [41], vehicles [59], hand-held devices, network devices, robotics [8] and many other cyber-physical systems. Many commercial and research architectures have been developed. Some notable commercial multi / manycore processors are Xeon Phi [17], Cell [42], big.LITTLE [31], Ambric [14], Epiphany [57], Myriad2 [7], TILE-Gx [52] and Xtensa LX [28].

In order to execute the case studies implemented in this thesis, we have used Epiphany architecture from Adapteva due to providing high performance

while consuming low power and supporting floating point operations. Epiphany architecture consists of 2D array of identical RISC cores which support floating point operations. Each core has 32KB of local memory and access to a 32MB of external shared memory. The cores are connected to a mesh network which uses an XY algorithm for routing. Adapteva has released two different boards with 16 and 64 Epiphany cores. Even if Epiphany is a homogeneous architecture we can use it to explore the mapping of streaming applications onto manycore architectures and use it as a baseline when we introduce heterogeneity.

## 2.2 Manycore Design

There are several dimensions of manycore design such as structure of the cores, structure of the memory, network-on-chip, and connection to the outside world. We propose a design method, which initially focuses on the structure of the core and then on the interconnection.

We have used a case study to evaluate the proposed manycore design approach. Several building blocks and an accelerator have been developed in Chisel language [5]. The accelerator is integrated to a rocket core, which executes RISC-V ISA, through rocket custom co-processor (RoCC) interface and rocket chip generator is used for generating the integrated design in verilog. This chapter provides the background knowledge about the RISC-V environment used for the case study. This environment includes the RISC-V ISA, Chisel language, rocket chip system on chip generator, rocket core and RoCC interface.

### 2.2.1 RISC-V and Rocket Chip

RISC-V [73] is a new and open instruction set architecture designed at UC Berkeley to support research and education on computer architectures. The base ISA consists of approximately 40 integer instructions and there is space left to support extensions. Some of the standard extensions are *multiply and divide, atomics, single precision* and *double precision floating point* instructions. There is still opcode space left for non-standard extension to let the developers add new features to their processors. The ISA supports 32-bit, 64-bit and 128-bit addressing modes.

Rocket chip [2] is a SoC generator written in Chisel which is an open source hardware construction language based on Scala. The generator consists of many parameterized libraries and can construct different manycore models based on RISC-V ISA. In the generator there are six sub-components namely (I)core generator, (II)cache generator, (III)Rocket custom co-processor generator, (IV)tile generator, (V)tilelink generator and (VI)peripherals. Tiles can consist of a core, cache and an accelerator. So far there are two base types of cores implemented which are the rocket core and the Berkeley Out of Order (BOOM) core. The cache generator supports different sizes, associativities and

replacement policies whereas the tile generator supports different number of cores and accelerators.

### 2.2.2 Rocket core and RoCC Interface

Rocket core [18] is an in order, single-issue, scalar processor with a 5-stage pipeline. It executes the 64-bit RISC-V ISA and features an integer ALU and an optional FPU. The default configuration of this core includes first level instruction and data caches. It additionally provides an accelerator or co-processor interface called RoCC.

The RoCC interface provides connections to the core and memory. Custom instructions are used for communication with the accelerator through this interface. These instructions provide two source registers and a destination register together with some extra bit fields. A bit field in the instruction indicates if the rocket core requires an answer from the accelerator. Figure 2.2 gives an overview of the request and response connections between the core and accelerator.

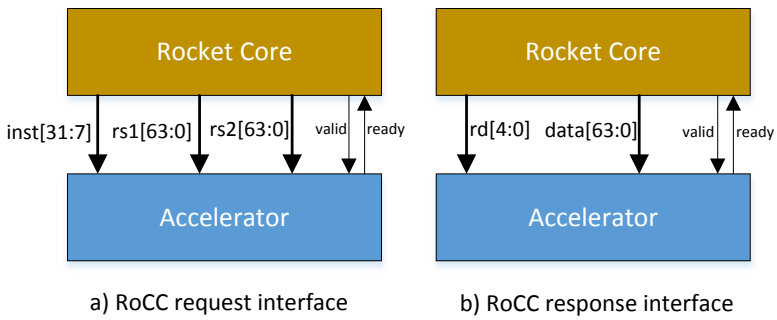


**Figure 2.2:** Connections between the core and the accelerator provided by the RoCC interface. Request and response connections are separated for a clearer illustration.

The interface provides an interrupt line and connections between the accelerator and the memory as well, however these connections are not necessarry in our implementations, thus they are not used.

## 2.3 Software Development for Manycores

Manycore architectures usually come with a low level programming language with some extensions or libraries. Developing efficient parallel applications in such languages requires good skills in parallel programming and detailed knowledge about the programming language, extensions and the underlying architecture. The difficulty of developing efficient applications increases as the

manycore architectures adopt heterogeneity. In homogeneous architectures, all cores can execute the same binary and since all the hardware resources are the same for every core, all tasks can be implemented in the same fashion. However, as the level of heterogeneity increases and new types of cores and new ISAs are introduced in the architecture, software development and execution becomes more complicated. The developer needs to generate different machine code for each type of core. To be able to run the same task on different cores, separate binaries are needed to be generated, distributed and stored. Having multiple binaries for the same task increases the memory requirement. If the architectures support dark silicon or similar methods, the developers need to take care of migrating the tasks from the cores which will be turned off. However, migrating tasks in heterogeneous architectures is not very easy since different cores might execute different instruction sets and have different processing resources. The migrated tasks may require modifications in case if they are not compatible with the new cores.

High level programming languages with support for parallelism require less knowledge and less effort for parallel application development. In order to reduce the complexity of development and execution of parallel applications, there is a need for languages with parallelism support and software development tools which facilitate parallelizing the application, generating code for different target architectures, mapping tasks onto the proper cores, etc.

We do not address all the software development and execution challenges in this thesis however, there has been some work done on the solution of some of these challenges. We will not survey through all the prior work, however we will mention a few of them. Bertozzi et al. [9] proposed a user-managed migration scheme based on code check-pointing and user-level middleware support to migrate tasks between cores. Ge et al. [24, 25] proposed a framework to migrate tasks between the neighboring cores in order to balance the load and manage the temperature. Kessler et al. [44] evaluates three different approaches to achieve portability and increased abstraction level for programming heterogeneous architectures. However there is still a lack of tools to automate the solutions and decrease the software development burden for both homogeneous and heterogeneous manycores.

In addition to programming languages and software development tools, programming model has a significant role in developing efficient software for manycore architectures. We have chosen dataflow programming model in order to implement our streaming applications. The next section provides the details on this programming model.

## 2.4 Dataflow Programming Model

Dataflow programming [54] is a programming paradigm that models a program as a directed graph of the data flowing between operations.

Streaming applications consist of various number of tasks which are performed on a flow of data. Dataflow programming model supports implementation of individual actors which are computational entities consisting of single or multiple instructions. The tasks can be directly mapped onto the actors and these actors can send the data to another actor to perform the next task. An actor executes its code when all the required input data is available. Multiple actors can execute simultaneously. These actors are connected through explicitly defined input and output ports. They can be locally synchronized by using blocking ports which makes the streaming application globally asynchronous but locally synchronous. The ports are connected via channels. Both Figure 1.2 and 2.3 present applications implemented with dataflow programming model.

A traditional application, that is implemented with dataflow programming model, is built as a series of actors connected in a specific order. For instance, in Figure 2.3 each box represents an actor whereas each arrow represents a unidirectional channel. The emphasis of this programming model is on the movement of the data that flows through these actors. There are several languages adopting this programming model such as Lucid [71], LUSTRE [32], StreamIt [66] and CAL actor language [21].

CAL is a modern dataflow language that is adopted by the MPEG Reconfigurable Video Coding (RVC) working group as part of their standardization efforts. We have used this programming language for implementing our streaming applications which will be described in the next section.

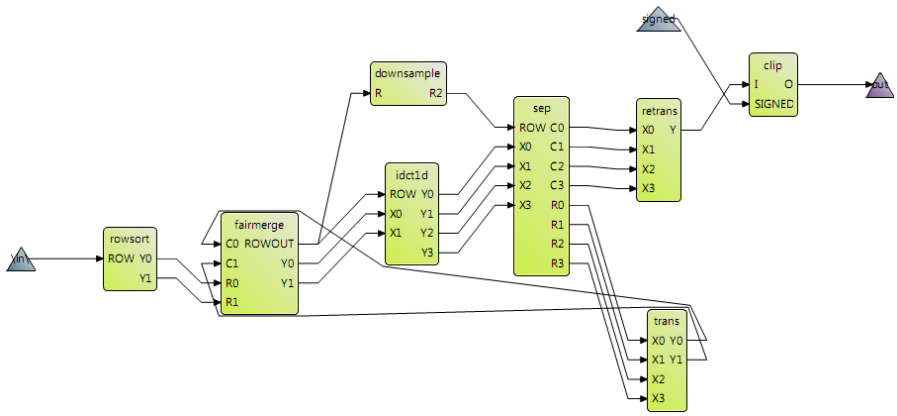


**Figure 2.3:** A sample streaming application - IDCT2D block of MPEG-4 Simple Profile decoder represented as interconnected actors (the triangles are input/output ports which can be connected to any source or sink)

## 2.5 Streaming Applications

Streaming applications, as illustrated in Figure 2.3 [60], consist of a chain of actors which perform some tasks on the input data. Some general characteristics of these applications are having a chain of processes (actors), data flowing through this chain and, synchronization of communicating cores by blocking. In addition to the general characteristics, individual actors, shown as boxes in the figure, might have different characteristics and require different hardware resources. For instance, the only task that is performed within the 'downsample' actor in Figure 2.3 is forwarding every second input to the output port which does not require any computing power. However, the 'idct1d' actor consists of several arithmetic and logic operations and hence require arithmetic and logic units. Similarly, other actors might require other hardware resources such as vector operators, floating point units, high performance addressing units, larger memory banks, etc. Due to this inherent heterogeneity, streaming applications and heterogeneous architectures become a natural match.

The most important characteristics of the applications implemented in this thesis is the large amount of core-to-core communication. Input data usually goes through the whole chain of cores which means it gets sent and received many times depending on the number of the cores. Therefore, the amount of core-to-core communication increases rapidly if the input data size increases. If the core-to-core communication is slow on an architecture, the overhead of communication might dominate the execution time of the applications and reduce efficiency. In order to execute streaming applications efficiently, the architecture needs to handle communication efficiently.

This thesis covers several streaming applications such as two-dimensional IDCT, different approaches for QR decomposition and autofocus criterion computation in SAR systems.

### 2.5.1 Two Dimensional Inverse Cosine Transform (IDCT2D)

Two dimensional IDCT, that is implemented as 15 actors and mapped onto 15 cores in Paper II, is a part of MPEG-4 simple profile decoder [60]. This decoder requires high computation power due to the capabilities and complexities of encoders in order to compress high quality videos efficiently. The implementation consists of many actors with different characteristics e.g. some actors only shuffle data in a matrix, some controls memory accesses and some performs different types of computations. When identical cores, as in the Epiphany architecture, are used the cores which perform duplicating or shuffling the data usually waits for the other cores which perform heavier duties. Therefore some cores get underutilized and wasted.

### 2.5.2 QR Decomposition

QR decomposition (QRD) [27] is another application that we have studied. It is used in several applications as a replacement of matrix inversions to avoid precision loss and reduce the number of operations. The operation is decomposition of a matrix into an upper triangular matrix R and an orthogonal matrix Q. We have implemented three basic approaches in dataflow programming model. The implemented actors are mostly identical, therefore executing them specifically on a heterogeneous architecture would not provide any specific benefits. This operation is a part of the task chain in massive MIMO application [58]. Paper II covers different methods of performing this operation.

### 2.5.3 SAR and Autofocus

Synthetic aperture radar systems [63] are usually mounted on moving platforms and uses the motion of the platform over a target region to create two or 3-dimensional high resolution images of landscapes. The path of the movement is not perfectly linear, however, this can be compensated with additional processing. The compensations are typically based on positioning information from GPS[68]. In cases where this data is insufficient, autofocus criterion can be used. There are different methods for calculating this criterion [16, 33]. One method tries to find the flight path compensation that results in the best possible match between two images of the contributing subapertures. This requires several flight path compensations to be tested. The matching is checked regarding a selected focus criterion. The criterion calculations, including interpolations and correlations, are performed many times. Paper IV proposes building blocks and an accelerator to perform these calculations efficiently.

As a conclusion, the streaming applications, which we developed, require high performance and low power. They consist of a chain of tasks with different characteristics and requirements. These tasks can be executed with different efficiencies on different types of cores. Therefore they are suitable candidates to be executed on heterogeneous architectures. However, each application would probably require different form of heterogeneity to be executed efficiently.

# Chapter 3
# Heterogeneity in Manycores

Heterogeneity in the context of computer systems usually means combination of processors with different instruction sets. However, there are different levels of heterogeneity and each level has advantages and disadvantages. One typical usage of heterogeneity is when a normal CPU is combined with a digital signal processor (DSP), specialized on fast multiply and add operations found in most signal processing applications. Thus, heterogeneity has for a long time been used as a way to accelerate computations in multiprocessors. Now, in the era of manycore architectures, we visit heterogeneity again in order to accelerate computations and reduce power consumption.

In parallel applications, especially in streaming domain, each core might perform a different task, and each task might require different computation, communication and memory resources for efficient execution. Our case studies on Epiphany architecture, which is a homogeneous architecture, have revealed that some cores might become a bottleneck due to executing tasks with specialized or higher requirements than the other tasks. Additionally, a sequential task, that is supposed to be executed on one core, might be too large to fit into the core. Dividing the sequential task on several cores and dealing with data dependencies would more likely harm the performance. All these facts imply that homogeneity is not the best solution and we need different types of cores to execute different parts of applications which means there is a need for introducing heterogeneity in manycores. Moreover, other research groups [47, 11, 51] have shown that heterogeneous architectures are more efficient than homogeneous architectures in power consumption and performance when implemented on the same die size.

Besides the power and performance advantages, heterogeneous architectures have drawbacks. These drawbacks can be divided into two aspects namely productivity (engineering efficiency) and portability (task/application migration both on-chip and off-chip). In order to make use of all resources in a heterogeneous architecture, one needs to develop different code for different cores unless there are software tools which generate specific code for each core.

This requires knowledge and effort and decreases the productivity of the developer.

In case of under-utilization of the resources such as cores, the tasks can be migrated from one core to another and the unutilized core can be shut down to save power. However, migrating tasks in heterogeneous architectures is a challenging duty. The cores might execute different ISAs which would require storing different binaries for the same task to support the migration. In some cases the new core might not support the operations performed by the task which might result in not migrating the task.

Another problem is the lack of knowledge about heterogeneity requirements of applications. This problem requires defining different forms of heterogeneity, profiling the applications, and exploring architectural design space of heterogeneous manycores. In order to address the first requirement, we will define different levels of heterogeneity in the next section.

## 3.1 Levels of Heterogeneity

In recent years we have seen the introduction of new forms of heterogeneity as a means to improving energy efficiency. One such example is the big.LITTLE concept from ARM, where two processors with the same ISA have different implementations in order to allow for a trade-off between performance and power-consumption. In a way the big.LITTLE concept can be seen as a "mild" form of heterogeneity. There are many other forms (or as we will call them, 'levels') of heterogeneity. They range from lightly heterogeneous architectures with processors using the same ISA but with different implementations [46], each resulting in different performance and power usage, all the way to very heterogeneous architectures with a large number of customized processing elements with different ISAs utilizing dark silicon concepts to reduce power consumption and manage thermal issues [30]. With these different forms of heterogeneity we find it natural to introduce the concept of *Levels of Heterogeneity*.

We base heterogeneity levels on the number of complete ISAs executed by the architectures. The level is equal to the number of ISAs with one exception. Homogeneous architectures execute one ISA, however, we consider them as Level 0 and leave Level 1 for the heterogeneous architectures with one ISA. If a core in a Level 1 architecture supports more instructions than the other cores, the heterogeneity level becomes Level 1Aug to reflect that some cores execute augmented version of the base ISA. This applies to the other levels as well.

In our definition we focus on the heterogeneity of the cores, and even if one can foresee architectures with heterogeneous memory models, this aspect has not been included. Based on this description we can identify the following levels of heterogeneity:

- HetLevel0: Level 0 contains homogeneous architectures.
- HetLevel1: Level 1 contains architectures with a single ISA but multiple implementations (IMPs).
- HetLevel1Aug: Level 1Aug contains the architectures where all the cores share a single base ISA but have different instruction augmentations.
- HetLevel2: Level 2 contains the architectures which contain two different ISAs (and therefore at least two distinct IMPs).
- HetLevel2Aug: Level 2Aug contains architectures where the cores execute two base ISAs and have different instruction augmentations.
- HetLevelN: Level N contains the architectures which contain N different ISAs.
- HetLevelNAug: Level NAug contains the architectures where the cores execute N base ISAs and have different instruction augmentations.

At Level 0 we have all the classical homogeneous parallel computers where all cores are identical. These multicore computers are today found in everything from smart phones to high-end desktop computers. As on-chip versions of symmetric multiprocessing (SMP) systems, which have decades of research behind them, these multicore architectures are well understood and have a relatively straightforward programming model.

In Level 1 every core executes the same instruction set. However, the implementation is not the same for all cores. Each core consists of different functional units and control logic while achieving different performances and energy efficiencies on the same application due to having different performance and power characteristics. Additionally, the cores might have differences in the length of pipelines, execution type (superscalar, in order, out of order), structure of functional units (parallel units) etc.

The simplest implementation would be having two different cores such as the big.LITTLE [31] architecture of ARM. However, there is no limit on the number of different cores. In our definition, any architecture with cores which differ in implementation but run the same complete ISA is a Level 1 architecture. There are many similarities between the HetLevel1 architectures like big.LITTLE and HetLevel0 architectures using DVFS and they can potentially be treated in the same operating system framework.

In some cases, an architecture might have different types of cores executing the same instruction set however some of the cores might support additional instructions due to having additional functional units such as floating point units or digital signal processing extensions. When an application does not need these extensions, it is executed on the cores without the extensions to consume less power. In such architectures, we consider the ISA of the core,

which has additional units, as an augmented ISA, and the heterogeneity level of such architectures as Level 1aug.

Level2 heterogeneous manycores consists of cores which use two different instruction set architectures. The implementations of the cores are different by default due to executing different ISAs. There are many architectures like the Epiphany which has a host core and an array of identical accelerator cores residing on different chips and executing different ISAs. This kind of architectures look like Level 2 architectures however, we do not include them in our definition due to having different types of cores on different chips. These architectures can become Level 2 architectures by moving the host core and the accelerator cores on to the same chip.

HetLevelN covers the levels above two without any ISA augmentation whereas HetLevelNAug covers the levels above two with augmented ISAs. There currently are no commercial architectures with three or more complete ISAs. However, we believe there will be such architectures in the future. Especially mobile platforms would gain from the energy efficiency of these architectures.

Table 3.1 shows the main levels of the heterogeneity together with example commercial architectures. Second and third columns are presenting the number of instruction set architectures and number of different implementations respectively.

**Table 3.1:** Main levels of heterogeneity and example commercial architectures.

| | **#ISA** | **#IMP** | **Example** |
|---|---|---|---|
| **Level 0** | 1 | 1 | Epiphany, |
| **Level 1** | 1 | 2+ | big.LITTLE, TILE-Gx |
| **Level 1Aug** | 1+Aug | 2+ | Xtensa LX, Ambric |
| **Level 2** | 2 | 2+ | Cell, Myriad2 |
| **Level N** | N | N+ | |

## 3.2 Conclusion

We defined heterogeneity in manycores and classified the heterogeneous architectures. We presented advantages and drawbacks of these architectures based on their levels. We conclude that heterogeneous architectures have the potential of achieving higher performance and lower power consumption when compared to homogeneous architectures. However, the advantages do not come for free. There are drawbacks in terms of productivity and portability which we believe can be solved (up to some extent) by developing software development tools. Additionally, there is still a huge design space to be explored to make best use of heterogeneity.

# Chapter 4
# Designing Domain Specific Heterogeneous Manycores

Specialized single core processors and heterogeneous manycores have been studied and proven to be more efficient than general purpose processors [47, 11, 51]. However, introducing heterogeneity in manycores increases the complexity of the architecture. Hence they are more difficult to develop and program when compared to homogeneous architectures.

In this chapter we try to develop a methodology to ease and facilitate development of heterogeneous manycore architectures. There have been several other works which try to facilitate development of manycore architectures by exploring the design space [15, 49, 74, 45, 62]. However, the design space for manycore architectures is huge, especially when heterogeneity is involved. Exploring all heterogeneity levels to find the most suitable architecture for target applications would require significant amount of time and effort. Instead of exploring the design space, we propose to design the architectures based on the requirements of the applications within a target domain. There have been processors designed for specific applications [61, 37, 29], however we target a more flexible architecture which can cover a domain with many application specific cores. Moreover, the method that we propose is generic and can be used for development of different architectures targeting different application domains. We have divided this method into 4 main steps namely, software development, analysis, hardware development, and manycore generation as seen in Figure 4.1.

The aim is to analyze applications within a domain, develop specialized hardware blocks to execute compute intensive parts of these applications, create tiles by integrating these blocks to simple cores and generate a manycore by placing these tiles on the same die. When an application is executed on this manycore processor, only the necessary cores would be utilized to save power.

The architectures developed with this method would be classified as architectures with augmented ISAs.The level of the heterogeneity would still

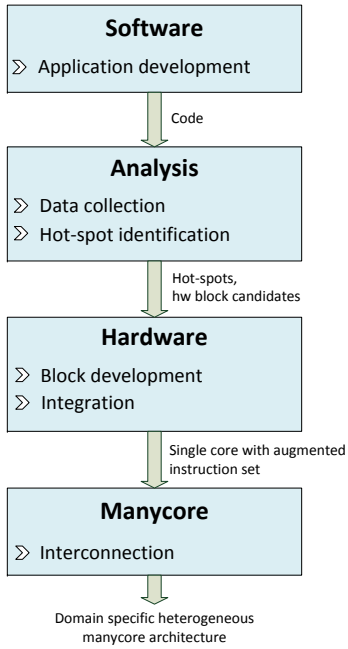


**Figure 4.1:** Design flow for developing domain specific heterogeneous manycore architectures.

depend on the number of ISAs. The heterogeneity level of the target manycore in Paper IV is HetLevel1Aug.

Automating the steps of the method will allow generation of libraries of specialized hardware blocks for different application domains. These libraries can be used for generation of application or domain specific heterogeneous architectures.

The first step, software development, is covered in section 2.3 and the last step, interconnection of the cores, is not covered in this thesis. However, it is within the future plans. The next sections provide details of analysis and hardware development steps.

## 4.1 Application analysis and hot-spot identification

Application analysis can be done statically or dynamically. Static analysis is usually done on the code before and after compilation to realize the types of operations and required instructions. However, it is more common to use dy-

namic analysis which is usually done by using hardware performance counters. Typical dynamic characteristics of applications are execution time, number of instruction fetches, cache miss rate, number of function calls and branch mis-prediction rate.

There are a number of tools and methods developed for analyzing applications at different abstraction levels [72, 6, 13, 39, 12]. These levels vary between static single assignment (SSA) form and high level dataflow languages. We try to keep the required information for identifying the hot-spots at minimal in order to make it easy to collect the data and analyze it. For the presented case study, gnu profiler (gprof) [23] and valgrind [55] tools provide enough information for identifying the hot-spots and estimate the potential speed-up. However, as the design method gets more mature, different tools might be necessary to collect different data. Gnu profiler provides number of function calls, total execution time of the application and the amount of time spent in each function among other information. This information is enough for identifying the hot-spots, however, to estimate the potential speed-up and make the final decision, it is necessary to combine them with the information provided by valgrind which is number of instruction fetches per-function.

The application that is studied in paper IV has a static schedule and the analysis information is independent of the inputs. However, dynamic profiling information might change with different input sets based on the conditions of different execution paths of the application. In such cases, either more complicated analysis methods must be used or more analysis information must be gathered with different input sets to estimate the average characteristics.

## 4.2 Custom hardware development and integration

Developing specialized hardware blocks usually seems to be straight forward. However, the integration step must be taken into account while developing these blocks. The interface, processing core, ISA, verification tools are some of the aspects which affect the hardware development.

The environment that is used for the initial version of the design methodology is the rocket chip environment. This environment includes a cycle accurate emulator, verilog generator and a rocket core implementation that executes RISC-V ISA and has an interface for custom hardware.

There are tools to automate the custom hardware development [40, 67, 48]. However, in order to integrate the custom hardware blocks to the target core and verify the functionality, development needs to be done manually as there are currently no tools to generate Chisel code.

Integration of the hardware blocks can be done through the rocket custom co-processor (RoCC) interface that is provided by the rocket core. This interface allows the core to communicate with the hardware block by executing custom instructions supported by the RISC-V ISA. The custom instruction needs to be inlined as assembly due to lack of compiler support. Number of in-

tegrated hardware blocks might change based on the requirement of the target application. Current interface (RoCC) supports integration of up to 4 different blocks.

## 4.3 Case Study - SAR Autofocus Criterion Computation

The application is developed in C language and profiled with gnuprof and valgrind. The most frequently called functions are identified with gnuprof. Additionally, the hot-spots are identified based on the time spent in each function. Figure 4.2 presents the functions of the application while highlighting the most frequently called ones. These are also the most time consuming and compute intensive parts.

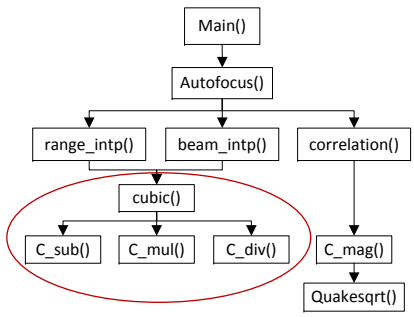


**Figure 4.2:** Function-call tree of autofocus application.

The compute intensive part of the application consists of cubic interpolation. This computation consists of 34 complex operations and each complex operation consists of several floating-point operations. These operations are developed as hardware blocks in Chisel. These blocks are used for building the accelerator with a hierarchical structure. In the bottom layer are the floating-point operation blocks such as subtraction and multiplication, on top of them are the complex operations. A group of complex operations forms a pixel calculation component and 6 of these components create the accelerator. Figure 4.3 illustrates this hierarchical structure.

The accelerator is integrated to a rocket core that executes the RISC-V instruction set. The integration is done through the RoCC interface. Figure 4.4 presents the processing tile that includes the rocket core, the accelerator and an overview of the RoCC interface.

Rocket core forwards the custom instructions to the RoCC interface together with the source registers in the instruction. Therefore, the custom in-

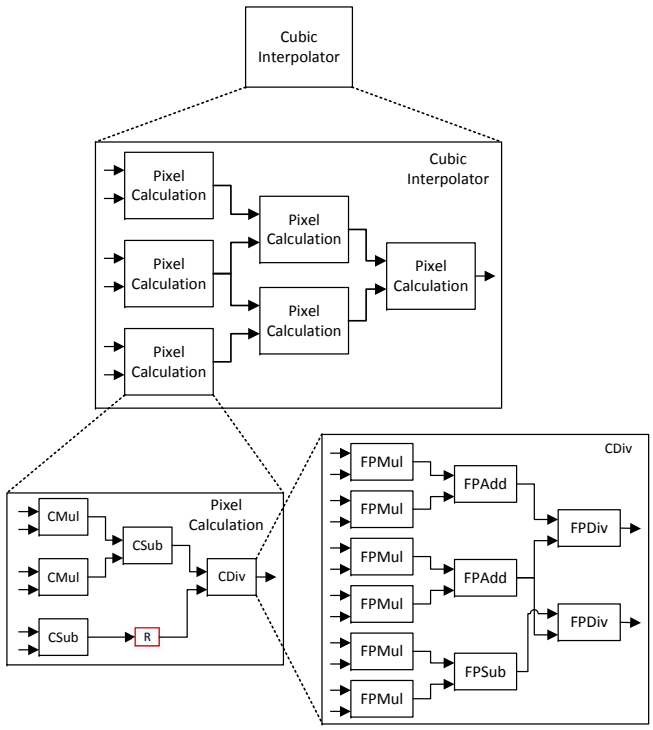


**Figure 4.3:** Hierarchical structure of the cubic interpolation accelerator.

structions need to be executed to trigger the accelerator. These instructions require to be inlined as assembly code. RISC-V instructions support two source registers which can be 64-bits long. We have combined 2 32-bit floating-point inputs in 1 register in order to forward 4 inputs with 1 instruction. The accelerator requires 16 inputs to perform the cubic interpolation, hence 4 instructions are required to be executed to forward all the inputs. This implies extra 3 clock cycles and a small interface between the RoCC interface and the accelerator to store the data of first 3 instructions and forward them together with the data of the 4th instruction.

After integration of the accelerator, the cubic interpolation is executed 15 times faster. The area of the tile increases by approximately 50%. However, there is room for improving both the performance and size.

The presented implementation (in Paper IV) is fully flattened. However the whole cubic interpolation can be performed by utilizing only one pixel calculation component with a loop. This would harm the performance however

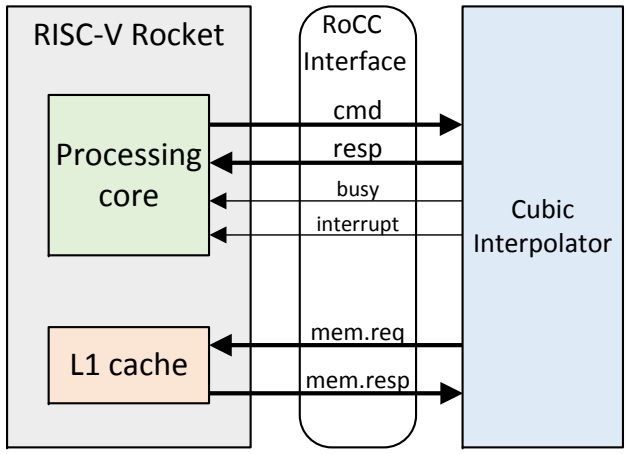


**Figure 4.4:** Overview of the tile including rocket core and cubic interpolation accelerator.

the area usage would decrease significantly. The area usage could be decreased further by using single components for the complex operations in the pixel calculation or by using single components for the floating-point operations in the complex operations. This implies a trade-off between area and performance. Once the hardware blocks are developed, one can estimate the performance and area usage of different implementations and make a decision based on the requirements.

# Chapter 5
# Summary of Papers

## 5.1 Evaluation of Software Development Tools for Manycores - Paper I

Developing applications for manycore architectures is a challenging task. The developer must have advance knowledge about the architecture and the language that is supported by the architecture. These languages are usually low level and include architecture specific extensions. In order to decrease development complexity and knowledge requirement, high level languages and code generation tools are developed. CAL actor language is a high level language aimed for the usage of parallel application development and Cal2Many [26] is a compilation framework which takes CAL code as input and generates compilable code for several different manycore platforms.

The contribution of this paper is the evaluation of the CAL actor language together with the Cal2Many compilation framework. Based on the evaluation and feedback, the compilation framework has been optimized. We developed two dimensional inverse discrete cosine transform in CAL and C languages targeting the Epiphany architecture. Cal2Many framework is used for generating C code by using the CAL code. The results (number of source line of code and execution duration) of hand-written and generated implementations with different memory and communication configurations are compared to each other and presented in the paper. The final results of the paper shows that the generated code runs approximately 30% slower while having 78%less source line of code.

## 5.2 Parallel QR Decomposition on Epiphany Architecture - Paper II

This paper presents development and comparison of three different parallel approaches of QR decomposition written both in CAL and C languages and executed on Epiphany architecture. Additionally, an evaluation of the Cal2Many

code generation framework is performed by comparing CAL and C versions of the approaches.

QR decomposition has many useful applications such as replacing matrix inversions to avoid precision loss and reduce number of operations, being a part of the solution to the linear least squares problem and being the basis of an eigenvalue algorithm (the QR algorithm).

The approaches are implemented with dataflow programming model and a custom communication library is used for communication between processing cores. The implementations are tested with different matrix sizes and varying memory and communication configurations. The results show that different algorithms show different performances with different configurations. Additionally, source line of code and footprint size differ from algorithm to algorithm.

## 5.3 Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis- Paper III

Computations on floating-point numbers are performed in countless areas. However implementing efficient floating-point arithmetics in hardware is a challenging task. Among the operations such as addition, subtraction, multiplication, and division, the latter is clearly the most challenging. It usually requires more area and more clock cycles.

In this paper, we implement a novel floating-point division hardware. This hardware performs an inversion on the divisor and then multiplies the result with the dividend. The inversion method is a combination of Parabolic Synthesis and Second Degree Interpolation [35, 34]. This hardware performs division on IEEE-754 single-precision floating-point numbers [38].

Pipelined and non-pipelined versions of the hardware is synthesized on Xilinx Ultrascale FPGA and compared to several other implementations which adapt different methods. The results show that the proposed implementation needs less resources than the comparable implementations which leads to shorter delay. In case of throughput, the proposed implementation outperforms most of the other works except one implementation on an Altera Stratix-V FPGA. This is due to the difference in the size of the multipliers used in the DSP slices. Stratix-V provides larger multipliers which would change the results of the proposed implementation significantly.

The proposed implementation provides high performance with low resource usage. Therefore, it is suitable for high performance embedded systems executing intensive computations such as complex division and cubic interpolation which are presented in the next paper (Paper IV).

## 5.4 Designing Domain Specific Heterogeneous Manycore Architectures Based on Building Blocks - Paper IV

This paper presents the first iteration of a method for designing domain specific heterogeneous architectures. The steps of the presented method can be summarized as:

- Analyze applications within a domain and identify hot-spots (compute intensive parts).
- Develop custom hardware for the operations in the hot-spots. Combine the blocks to create accelerators to execute these hot-spots.
- Integrate many accelerators to many simple cores.
- Connect the cores with a network-on-chip.

The first three steps covered in the paper with a case study using a SAR Autofocus application. The application performs intense floating-point operations for cubic interpolation computations which is identified as the hot-spot of the application. Several building blocks are developed to perform the operations and an accelerator is built by combining different building blocks. The accelerator performs the entire cubic interpolation on four input pixels.

The accelerator is integrated to a RISC-V core through the RoCC interface. The rocket chip generator is used for generating an RTL implementation that is synthesized on a Xilinx Ultrascale FPGA. Resource usage and timing results are obtained from the Xilinx tools whereas the performance results are obtained via a cycle accurate emulator provided with the rocket chip generator.

The results show that the cubic interpolation can be executed 15x faster if the accelerator is utilized, despite the drawbacks in the integration between the rocket core and the accelerator. The accelerator is fully pipelined and can produce a result per-cycle. However the interface cannot feed the required (16 floating-point) input data in a clock cycle due to having only two source registers in the RISC-V instructions. Maximum input rate is 4 input data per-cycle (2 single precision floating-point numbers in each 64-bit source register). Even with this input rate, the potential speed up for throughput is above 100x, however this would require usage of interrupts and some extra investigation. The area usage increases significantly however there is room for improvement. For instance, the size could be decreased to approximately 1/6 by using only a pixel calculation component, which can be seen in Fig 4.3, for the whole computation.

The processing tile that consists of the rocket core and the accelerator proves to be more efficient than the rocket core itself while executing the cubic interpolation. A manycore architecture with such tiles with different

accelerators targeting applications within a domain would provide efficient execution of the entire domain. Advantage of targeting a specific domain is that several applications might share the same specialized blocks. This would allow reuse of the blocks and decrease the development time and area usage. The implementation in this paper is fully flattened, however, this is not always the best solution.

# Chapter 6
# Conclusions and Future Work

## 6.1 Conclusions

Streaming applications consist of different tasks which require different hardware resources to be executed efficiently. However, a majority of current manycore architectures are homogeneous - built with cores which have identical resources. Therefore, we propose utilizing heterogeneity in manycore architectures in order to increase their efficiency in terms of performance and power consumption. However, when heterogeneity is introduced, developing software and porting it to different architectures or even to different cores in the same architecture become more challenging. In order to facilitate software development and increase code portability we propose utilization of new software development tools such as source to source compilers and mapping generators. Another challenge is development of efficient heterogeneous manycore architectures due to the complexity of their structure. We propose a design method to reduce the difficulty of heterogeneous manycore development.

In order to evaluate our proposals, we revealed advantages and drawbacks of heterogeneity in manycore architectures by identifying and discussing different levels of it. We described a new taxonomy for manycore architectures that is based on these levels. Additionally, for revealing the challenges of software development for manycores, we developed several streaming applications. In order to facilitate the software development, we used a dataflow language and utilized a source to source compilation framework. We executed the applications on a commercial manycore architecture, evaluated both the framework and the target manycore architecture revealing some of the drawbacks of the architecture and the compilation framework. In addition to the challenges of software development, we target the challenges of hardware development as well by suggeting and exploring a methodology for designing domain specific heterogeneous manycore architectures. The methodology is based on augmentation of simple cores with specialized hardware blocks. We developed several hardware blocks for complex and floating-point arithmetics. We created an

accelerator based on these blocks to execute compute intensive parts of an application and integrated it to a RISC core.

We have realized that usage of software development tools, which increase the abstraction level, and domain specific languages leads to reduced development effort (25-50%) and increased software portability with a small penalty on the performance (2-17 %). It additionally decreases the level of knowledge required about the target architectures. However, the tools usually cover only homogeneous architectures, thus there is still a need for progress in these tools, languages and programming models in order to exploit heterogeneity in manycores. On the hardware side, our floating-point division block, which is based on a novel method, has outperformed the other division hardwares either in performance or resource usage. We have achieved 15x better performance while performing cubic interpolation by integrating custom hardware to the RISC core. The custom hardware increases the cores size by ~50% with room for both performance and area improvements.

Our results show that heterogeneity has the potential to increase the performance of manycore architectures whereas software development tools increase the productivity of the developer and portability of the code. The design method eases the development of heterogeneous architectures by augmenting simple existing cores and avoid design space exploration to find suitable architectures for applications or domains. There can be a requirement for design space exploration for the area-performance trade-off of the accelerator, however, we believe that this design space is significantly smaller than the design space of manycore architectures. Additionally, in a usual case, an application will not utilize all the cores and specialzed blocks. This could facilitate usage of power saving techniques by revealing the components which could be shut down while executing an application.

## 6.2 Future Work

The accelerator that is developed with smaller blocks in paper IV has room for performance and area optimization. It can be shrunk significantly without any penalty on current performance and the performance can be increased by using interrupts instead of halting the processor until the accelerator returns the result. The accelerator is built hierarchically with several layers. These layers can be structured in different ways to test different implementations which would give varying area and performance results.

The proposed manycore design methodology is tested partially in paper IV. There is a need for larger application, more hardware blocks, more cores and a network-on-chip to test the last step of the method which is the interconnection of cores.

The tasks in each completed step of the design method are performed manually. This demands great amount of time, effort and knowledge. However, we plan to automate each step and finally combine all these steps in a framework.

For the automation of each step, we aim to utilize the existing tools as much as possible to decrease the required effort, however there will be a need for new tools for tasks such as generating hardware based on the analysis data while considering integration, connecting the cores with each other etc.

# References


[1] Hartwig Anzt, Vincent Heuveline, José I Aliaga, Maribel Castillo, Juan C Fernandez, Rafael Mayo, and Enrique S Quintana-Orti. Analysis and optimization of power consumption in the iterative solution of sparse linear systems on multi-core and many-core platforms. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–6. IEEE, 2011.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. 2016.

[3] 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, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[4] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[6] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society, 1996.

[7] Brendan Barry, Cormac Brick, Fergal Connor, David Donohoe, David Moloney, Richard Richmond, Martin O'Riordan, and Vasile Toma.

Always-on vision processing unit for mobile applications. *IEEE Micro*, (2):56–66, 2015.

[8] Ron Bekkerman, Mikhail Bilenko, and John Langford. *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press, 2011.

[9] Stefano Bertozzi, Andrea Acquaviva, Davide Bertozzi, and Antonio Poggiali. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 15–20. European Design and Automation Association, 2006.

[10] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32 nm 1000-processor array. In *IEEE HotChips Symposium on High-Performance Chips*, August 2016.

[11] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik, and Olaf O Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[12] Simone Casale Brunei, Marco Mattavelli, and Jörn W Janneck. Turnus: a design exploration framework for dataflow system design. In *International Symposium on Circuits and Systems (ISCAS)*, pages 654–654. IEEE, 2013.

[13] Simone Casale Brunet, Marco Mattavelli, and Jorn W Janneck. Profiling of dataflow programs using post mortem causation traces. In *Workshop on Signal Processing Systems (SiPS)*, pages 220–225. IEEE, 2012.

[14] Michael Butts, Brad Budlong, Paul Wasson, and Ed White. Reconfigurable work farms on a massively parallel processor array. In *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on*, pages 206–215. IEEE, 2008.

[15] Horia Calborean and Lucian Vinţan. An automatic design space exploration framework for multicore architecture optimizations. In *9th Roedunet International Conference (RoEduNet)*, pages 202–207. IEEE, 2010.

[16] Terry M Calloway and Gregory W Donohoe. Subaperture autofocus for synthetic aperture radar. *IEEE Transactions on Aerospace and Electronic Systems*, 30(2):617–621, 1994.

[17] George Chrysos. Intel xeon phi coprocessor-the architecture. *Intel Whitepaper*, 2014.

[18] Rocket Core. Avaliable online [https://github.com/ucb-bar/rocket], 2017.

[19] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[20] Ronald G Dreslinski, Michael Wieckowski, David Blaauw, Dennis Sylvester, and Trevor Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, 2010.

[21] J. Eker and J. W. Janneck. CAL language report specification of the CAL actor language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003.

[22] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *IEEE M MICRO*, 32(3):122–134, 2012.

[23] Jay Fenlason and Richard Stallman. The gnu profiler. *World Wide Web, URL: http://www. cs. utah. edu/dept/old/texinfo/as/gprof_toc. html*, 2005.

[24] Yang Ge, Parth Malani, and Qinru Qiu. Distributed task migration for thermal management in many-core systems. In *Proceedings of the 47th Design Automation Conference*, pages 579–584. ACM, 2010.

[25] Yang Ge, Qinru Qiu, and Qing Wu. A multi-agent framework for thermal aware task migration in many-core systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(10):1758–1771, 2012.

[26] Essayas Gebrewahid, Mingkun Yang, Gustav Cedersjo, Zain Ul Abdin, Veronica Gaspes, Jörn W Janneck, and Bertil Svensson. Realizing efficient execution of dataflow actors on manycores. In *12th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 321–328. IEEE, 2014.

[27] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[28] R. E. Gonzalez. Xtensa: a configurable and extensible processor. 20(2):60–70, 2000.

[29] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 137–147. ACM, 2003.

[30] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *IEEE M MICRO*, 31(2):86–95, 2011.

[31] Peter Greenhalgh. Big.little processing with ARM cortex-A15 and cortex-A7. White paper, ARM, September 2011.

[32] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[33] Hans Hellsten, Patrik Dammert, and Anders Åhlander. Autofocus in fast factorized backprojection for processing of SAR images when geometry parameters are unknown. In *Radar Conference, 2010 IEEE*, pages 603–608. IEEE, 2010.

[34] Erik Hertz. *Methodologies for Approximation of Unary Functions and Their Implementation in Hardware*. PhD thesis, Halmstad University Press, 2016.

[35] Erik Hertz, Bertil Svensson, and Peter Nilsson. Combining the parabolic synthesis methodology with second-degree interpolation. *Microprocessors and Microsystems*, 42:142–155, 2016.

[36] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, (7):33–38, 2008.

[37] Andreas Hoffmann, Oliver Schliebusch, Achim Nohl, Gunnar Braun, Oliver Wahlen, and Heinrich Meyr. A methodology for the design of application specific instruction set processors (asip) using the machine description language lisa. In *International Conference on Computer Aided Design, ICCAD*, pages 625–630. IEEE/ACM, 2001.

[38] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[39] Jorn W Janneck, Ian D Miller, and Dave B Parlour. Profiling dataflow programs. In *International Conference on Multimedia and Expo*, pages 1065–1068. IEEE, 2008.

[40] Jörn W Janneck, Ian D Miller, David B Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 63(2):241–249, 2011.

[41] De Saint Jorre, Damien Jack, Claudio Alberti, Marco Mattavelli, and Simone Casale Brunet. Porting an mpeg-hevc decoder to a low-power many-core platform. In *Asilomar Conference on Signals, Systems, and Computers*, number EPFL-CONF-189720, 2013.

[42] M.N Kahle, J.A .and Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4):589–604, 2005.

[43] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-threshold voltage (ntv) design: opportunities and challenges. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1153–1158. ACM, 2012.

[44] C. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J.L. Traff, and S. Pllana. Programmability and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1403 –1408, march 2012.

[45] Michel A Kinsy, Michael Pellauer, and Srinivas Devadas. Heracles: A tool for fast rtl-based design space exploration of multicore processors. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 125–134. ACM, 2013.

[46] 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 *Proceedings of 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-36*, pages 81 – 92, dec. 2003.

[47] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 64 – 75, june 2004.

[48] David Lau, Orion Pritchard, and Philippe Molson. Automated generation of hardware accelerators with direct memory access from ansi/iso standard c functions. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 45–56. IEEE, 2006.

[49] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-42*, pages 469–480. IEEE/ACM, 2009.

[50] Dake Liu, A. Karlsson, J. Sohl, Jian Wang, M. Petersson, and Wenbiao Zhou. epuma embedded parallel dsp processor with unique memory access. In *Proceedings of 8th Int Information Conference on Communications and Signal Processing (ICICS)*, pages 1–5, 2011.

[51] Shaoshan Liu, Won W Ro, Chen Liu, Alfredo C Salas, Christophe Cérin, and Jean-Luc Gaudiot. Eha: The extremely heterogeneous architecture. In *Proceedings of the International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN 2012), San Marcos, Texas*, 2012.

[52] Matthew Mattina. Architecture and performance of the tile-gx processor family. Whitepaper, Tilera Corporation, 2014.

[53] Sparsh Mittal. A survey of techniques for improving energy efficiency in embedded computing systems. *International Journal of Computer Aided Engineering and Technology*, 6(4):440–459, 2014.

[54] Walid A Najjar, Edward A Lee, and Guang R Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13):1907–1929, 1999.

[55] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[56] Andreas Olofsson. Epiphany-v: A 1024 processor 64-bit risc system-on-chip. Technical report, Adapteva Inc, 2016.

[57] Andreas Olofsson, Tomas Nordstrom, and Zain Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *48th Asilomar Conference on Signals, Systems and Computers*, 2014.

[58] Arogyaswami J Paulraj, Dhananjay A Gore, Rohit U Nabar, and Helmut Bölcskei. An overview of mimo communications-a key to gigabit wireless. *Proceedings of the IEEE*, 92(2):198–218, 2004.

[59] Tiana A Rakotovao, Diego P Puschini, Julien Mottin, Lukas Rummelhard, Amaury Negre, and Christian Laugier. Intelligent vehicle perception: Toward the integration on embedded many-core. In *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures*, pages 7–12. ACM, 2015.

[60] Suleyman Savas. Implementation and evaluation of mpeg-4 simple profile decoder on a massively parallel processor array. Master's thesis, Halmstad University, 2011.

[61] Deepti Shrimal and Manoj Kumar Jain. Instruction customization: A challenge in asip realization. *International Journal of Computer Applications*, 98(15), 2014.

[62] Cristina Silvano, William Fornaciari, and Eugenio Villar. *Multi-objective design space exploration of multiprocessor SoC architectures.* Springer, 2014.

[63] Mehrdad Soumekh. *Synthetic aperture radar signal processing*, volume 7. New York: Wiley, 1999.

[64] S. Swanson and M. B. Taylor. Greendroid: Exploring the next evolution in smartphone application processors. 49(4):112–119, 2011.

[65] Joel M Tendler, J Steve Dodson, JS Fields, Hung Le, and Balaram Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[66] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[67] Jelena Trajkovic, Samar Abdi, Gabriela Nicolescu, and Daniel D Gajski. Automated generation of custom processor core from c code. *Journal of Electrical and Computer Engineering*, 2012:7, 2012.

[68] Zain Ul-Abdin, Anders Ahlander, and Bertil Svensson. Energy-efficient synthetic-aperture radar processing on a manycore architecture. In *42nd International Conference on Parallel Processing (ICPP)*, pages 330–338. IEEE, 2013.

[69] Zain Ul-Abdin and Mingkun Yang. A radar signal processing case study for dataflow programming of manycores. *Journal of Signal Processing Systems*, pages 1–14, 2015.

[70] Srinivasa R Vemuru and Norman Scheinberg. Short-circuit power dissipation estimation for cmos logic gates. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 41(11):762–765, 1994.

[71] William W Wadge and Edward A Ashcroft. *Lucid, the dataflow programming language.* 1985.

[72] Chao Wang, Xi Li, Huizhen Zhang, Aili Wang, and Xuehai Zhou. Hot spots profiling and dataflow analysis in custom dataflow computing soft-processors. *Journal of Systems and Software*, 125:427–438, 2017.

[73] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The risc-v instruction set manual, volume i: Base user-level isa. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[74] Benard Xypolitidis, Rudin Shabani, Satej V Khandeparkar, Zain Ul-Abdin, Süleyman Savas, and Tomas Nordström. Towards architectural design space exploration for heterogeneous manycores. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 805–810. IEEE, 2016.

[75] Miao Zhang and Dexian Zhang. A heterogeneous multi-core architecture with stability and high performance. In *International Conference on Computer Application and System Modeling (ICCASM)*, volume 5, pages V5–164 –V5–168, oct. 2010.

# Appendix A

## Paper I
## An evaluation of code generation of dataflow languages on manycore architectures

Süleyman Savas, Essayas Gebrewahid, Zain Ul-Abdin, Tomas Nordström, and Mingkun Yang

*20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '14)*, IEEE, 20-22 Aug, 2014, Chongqing, China.

# An Evaluation of Code Generation of Dataflow Languages on Manycore Architectures

Süleyman Savas, Essayas Gebrewahid, Zain Ul-Abdin, Tomas Nordström and Mingkun Yang

Centre for Research on Embedded Systems Halmstad University, Halmstad, Sweden

{suleyman.savas, essayas.gebrewahid, zain-ul-abdin, tomas.nordstrom}@hh.se, minyan09@student.hh.se

***Abstract*—Today computer architectures are shifting from single core to manycores due to several reasons such as performance demands, power and heat limitations. However, shifting to manycores results in additional complexities, especially with regard to efficient development of applications. Hence there is a need to raise the abstraction level of development techniques for the manycores while exposing the inherent parallelism in the applications. One promising class of programming languages is dataflow languages and in this paper we evaluate and optimize the code generation for one such language, CAL. We have also developed a communication library to support the inter-core communication. The code generation can target multiple architectures, but the results presented in this paper is focused on Adapteva's many core architecture Epiphany. We use the two-dimensional inverse discrete cosine transform (2D-IDCT) as our benchmark and compare our code generation from CAL with a hand-written implementation developed in C. Several optimizations in the code generation as well as in the communication library are described, and we have observed that the most critical optimization is reducing the number of external memory accesses. Combining all optimizations we have been able to reduce the difference in execution time between auto-generated and hand-written implementations from a factor of 4.3x down to a factor of only 1.3x.**

***Index Terms*—Manycore, Dataflow Languages, code generation, Actor Machine, 2D-IDCT, Epiphany, evaluation**

## I. Introduction

In the last decade we have seen a transition from single core processors to multicore and manycore architectures, we furthermore see a transition to a distributed memory model where each core contains part of the common memory. These transitions have emphasized the need for programming languages that better express inherent parallelism available in the targeted applications. As a result, new languages based on parallel and concurrent programming paradigms have emerged [1], and the dataflow programming model [2], [3], [4] seems to be a particularly good candidate for the big class of streaming applications where data is processed as a continuous stream. There is however a lack of compilation frameworks, supporting dataflow languages, that generate code for the latest generation of manycore architectures.

In this paper we will evaluate and optimize a compilation framework targeting Adapteva's manycore architecture Epiphany [5]. This architecture is an example of the next generation of manycore architectures that have been developed in order to address the power and thermal dissipation limitations of traditional single core processors. In the Epiphany architecture the memory is distributed among all cores and a shared address space allows all cores to directly access data located in other cores. However, in contrast to traditional multicores there is no cache memory and no direct hardware support for memory coherence.

The dataflow language used in this work is the CAL actor language (CAL) [4], [6], which is a modern dataflow language that is adopted by the MPEG Reconfigurable Video Coding (RVC) [7] working group as part of their standardization efforts. In [8] the CAL compilation framework evaluated and optimized in this paper is described. Our code generator uses a simple machine model, called *actor machine* [9], to model actors in a dataflow application. We have also introduced an *action execution intermediate representation* [8] to support program portability and we currently support generating sequential C, parallel (manycore) C, and aJava/aStruct languages. These intermediate representations are further introduced in section IV. For the parallel C backend the code generator utilizes an in-house developed communication library [10] to support the communication between actors.

In order to evaluate various optimization methods for our CAL code generator we will use a CAL implementation of the Two-Dimensional Inverse Discrete Cosine Transform (2D-IDCT) as a case study. We will compare the performance of our automatically generated code (from CAL) with a hand-written implementation programmed in C. From this comparison a number of optimization opportunities has been found and we will be able to evaluate the benefit of the various optimizations implemented.

## II. Related Works

CAL compilers have already been targeting a variety of platforms, including single-core processors, multicore processors, and programmable hardware. The Cal2C compiler [11] targets the single-core processors by generating sequential C-code. The Open-RVC CAL Compiler (ORCC) [12] generates multi-threaded C-code that can execute on a multicore processor using dedicated run-time system libraries. Similarly the d2c [13] compiler produces C-code that makes use of POSIX threads to execute on multicore processors.

In this paper, we present and evaluate a code generator for manycore architectures that use *actor machine* [9] and *action execution* intermediate representations (IRs) together with a backend for the target architecture and a custom communication library. Executing a CAL actor includes choosing an action that satisfy the required conditions and firing the action. In other CAL code generators [11] [12], all required conditions of the actions are tested at once, thus common conditions among actions will be tested more than once. However in our work, the schedule generated by the *Actor machine* stores the values of the conditions and avoids a second test on the conditions that are common.

In order to increase the portability of our compilation tool we have used an imperative IR called *action execution intermediate representation*(AEIR) which distinguishes computation and communication parts of the CAL applications. Hence it becomes easier to port the application to another architecture just by replacing the communication part with architecture specific communication operations. In addition, our compilation tool generates separate code for each actor instance so that it could be executed on individual processing cores. Therefore we do not require any run-time system support for concurrent execution of actors, in contrast to ORCC and d2c.

## III. Background

### A. Epiphany Architecture

Adapteva's manycore architecture Epiphany [5] is a two-dimensional array of processing cores connected by a mesh network-on-chip. Each core has a floating-point RISC CPU, a direct memory access (DMA) engine, memory banks and a network interface for communication between processing cores. An overview of the Epiphany architecture can be seen in Figure 1.

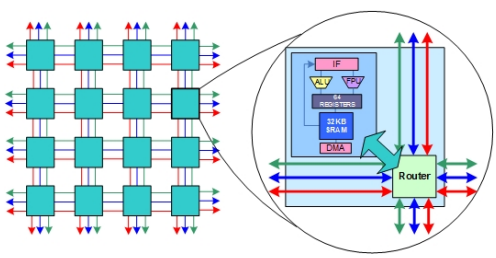


Fig. 1. Epiphany architecture overview.

In the Epiphany architecture each core is a superscalar, floating-point, RISC CPU that can execute two floating point operations and a 64-bit memory load operation on every clock cycle. The cores are organized in a 2D mesh topology with only nearest-neighbor connections. Each core contains a network interface, a multi-channel DMA engine, a multicore address decoder, and a network-monitor. The on-chip node-to-node communication latencies are 1.5 clock cycles per routing hop, with zero startup overhead. The network consists of three parallel networks which are used individually for writing on-chip, writing off-chip, and all read requests, respectively. Due to the differences between the networks, writes are approximately 16 times faster than reads for on-chip transactions. The transactions are done by using dimension-order routing (X-Y routing), which means that the data first travels along the row and then along the column. The DMA engine is able to generate a double-word transaction on every clock cycle and has its own dedicated 64-bit port to the local memory. The Epiphany architecture uses a distributed memory model with a single, flat address space. Each core has its own aliased, local memory range which has a size of 32kB. The local memory of each core is accessible globally from any other core by using the globally addressable IDs. However, even though all the internal memory of each core is mapped to the global address space, the cost (latency) of accessing them is not uniform as it depends on the number of hops and contention in the mesh network.

### B. Dataflow Programming

Dataflow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations or actors that operate on the dataflow. Dataflow programming emphasizes the movement of data and models programs as a series of connected actors. Each actor only communicates through explicitly defined input and output connectors and functions like a black box. As soon as all of its inputs become valid, an actor runs asynchronous from all other actors. Thus, dataflow languages are inherently parallel and can work well in large, decentralized systems. Dataflow programming is especially suited for streaming applications, where data is processed as a continuous stream, as, e.g., video, radar, or base station signal processing.

A number of dataflow languages and techniques were studied [2], [3], and one of the interesting modern dataflow languages is CAL Actor Language which has recently been used in the standardization of MPEG Reconfigurable Video Coding (RVC).

### C. CAL Actor Language

A CAL dataflow program consists of stateful operators, called actors, that transform input streams of data objects (tokens) into output streams. The actors are connected by FIFO channels and they consist of code blocks called actions. These actions transform the input data into output data, usually with the state of the actor changed.

An application consists of a network of actors. When executed, an actor consumes the tokens on the input ports and produces new tokens on the output ports. Following example shows an actor which has one input port, two output ports and two actions.

```
actor Split() int I1 ==> P, N:
  action I1:[a] ==> P:[a]
    guard a >= 0 end
  action I1:[a] ==> N:[a]
    guard a < 0 end
```

```
end
```

The first line declares the actor name, followed by a list of parameters (which is empty, in this case) and the declaration of the input (I1) and output ports (P and N). The second line defines an action [14] and the third line defines a guard statement. In many cases there are additional conditions which need to be satisfied for an action to fire. These conditions can be specified using the guard statements, as seen in the Split actor. The schedule keyword in CAL permits to determine the transitions between the different states. Each state transition consists of three parts: the original state, a list of action tags, and the following state. For instance,

```
init (readT) --> waitA;
```

`init` is the original state, `readT` is an action tag and `waitA` is the following state. In this example, if `readT` action is executed within the `init` state, the actor will go to the next state which is `waitA`.

CAL has a network language (NL) in order to configure the structure of the application. The structure is built by instantiating the actors, defining channels and connecting the actors via these channels. Following example demonstrates a network with one input port and one output port. In the network, one processor is instantiated and it is connected to the input and output ports of the network.

```
network idct X0 ==> Y0
entities
    scale = Scale()
structure
    X0 --> scale.X0
    scale.Y0 --> Y0
end
```

## IV. Code Generation

The code generator starts from high-level CAL implementation and generates a native implementation. Figure 2 shows the CAL compilation framework. In CAL, actors execute by firing of actions that satisfy all the required conditions like availability of tokens, value of the tokens and internal state of the actor. In code generation, the code generator has to consider the ordering of these conditions and the number of the tests performed before an action is fired. To schedule the testing of firing conditions of an action we have used a simple actor model called *actor machine* (AM) [9]. In addition, we also have used an *Action Execution Intermediate Representation* (AEIR) that bring us closer to imperative languages [8]. As shown in Figure 2, in our compilation process each CAL actor is translated to an AM that is then translated to AEIR. Finally, the AEIR is used by three backends that generate target specific code. Currently we have three backends: a uniprocessor backend that generates sequential C code for a single general purpose processor, an Epiphany backend that generates parallel C code for Epiphany, and Ambric Backend that generates aJava and aStruct for Ambric massively-parallel processor array [15].

### A. Intermediate Representation

*1) Actor Machine:* AM is a controller that provide an explicit order to test all the firing conditions of all actions in an actor. AM is made up of a set of states that memorize the values of all conditions in an actor. Each state has a set of AM instructions that leads to a destination states once applied on the state. These AM instructions can be

- a *test* to perform a test on a guard or input port for availability of token,
- an *exec* for an execution of an action that satisfy all the firing conditions, or
- a *wait* to change information about absence of tokens to unknown, so that a test on an input port can be performed after a while.

Cedersjö and Janneck [16] have presented and examined several ways of transforming an actor to an actor machine . The code generator used in this paper translates CAL actors to actor machines which memorize all the conditions and have at most one AM instrucion per state.

*2) Action Execution Intermediate Representation:* AEIR is a data structure that is applicable to generate code for imperative languages like C, C++ and java. It has constructs for expressions, statements, function declarations and function calls. Translation of AM to AEIR deals with two main tasks. The first task is the translation of CAL constructs to imperative constructs. This includes CAL actions, variable declarations, functions, statements and expressions. The second task is the implementation of the AM that is the translation of the AM to a sequential action scheduler. The states and the instructions of the AM are translated to programing language constructs: the states are translated to unique labels or if-statement on state variable, *test* nodes to if-statements, *exec* to function call and *wait* to a call for a target specific operation that pause the execution of a process. The action scheduler is kept as a separate imperative function in the AEIR. In a prior work [8], we have presented the use of AEIR to generate sequential and parallel C code. Section V-B presents three different parallel C codes generated for Epiphany architecture.

### B. Mapping CAL actors

There is no well-defined semantics for NL to guide an implementation. In our work we have used Kahn Process Networks (KPN) [17] for Ambric and extended Dataflow Process Networks (DPN) [18] to generate sequential C code for a general purpose CPU and parallel C code for Epiphany.

Ambrics proprietary tool comes with communication API and mapping and scheduling protocols that support KPN. Thus, for Ambric we have adapted the code generation in accordance with the existing support for KPN. Ambric code generation generate `aJava` object and `aStruct` code for each actor and top level design file for the application. The `aJava` source code is compiled into executable code by using Ambrics compiler.

For DPN we have implemented a communication API that use bounded buffers to connect output ports to input ports:

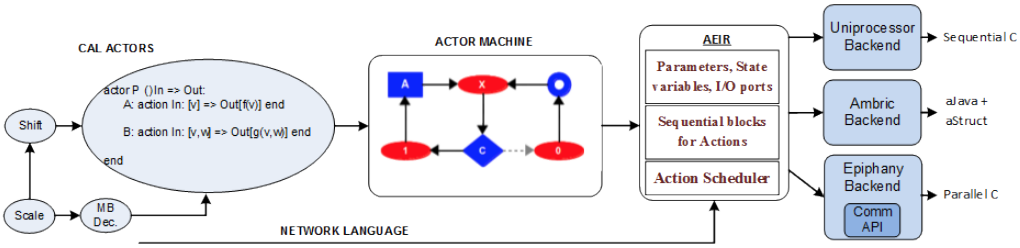


Fig. 2. CAL compilation framework.

these are blocking for writing when the buffer is full, but allows peeking without blocking. If there are multiple actors on single core, writing on full buffer blocks the running of the current actor and gives control to the other actors. The flattened network is used to generate a round-robin scheduler for the sequential C code and a host code for Epiphany that map the actors on separate processors.

*1) The Dataflow Communication Library:* All the communication that is done between the actors is done through FIFO buffers, thus making this functionality a key component for the compilation of the applications developed in CAL onto a manycore architecture. We suggest to implement this functionality as a dataflow communication library, with only five basic functions. In addition to the traditional functions of `write` and `read` from the FIFO buffer we have added functions such as `connect` which logically connects two actors, `disconnect` which logically disconnects the actors, and finally a function `end_of_transmission` that flushes the buffer and indicates that there are no further tokens to be sent.

When implementing these buffers on the Epiphany architecture, two special features of this architecture need to be considered. First one is the speed difference between read and write transactions (as mentioned earlier, writes are faster). The second one is the potential use of DMA to speed up memory transfer and allowing the processor to do processing in parallel to the memory transfer.

We have investigated three ways to implement the FIFO buffering. The first implementation is a 'one-end-buffer' which places the buffer inside the input port in the destination core. (Putting the buffer on the source core would result in reading from a remote core instead of writing to a remote core and thus a tenfold slowdown.) The communication overhead resides completely in the sender.

If we want to use DMA, we need to have a buffer on both the sender and receiver side. In the second implementation ('two-end-buffer') each core performs read and write transactions on its local memory and then uses DMA to transfer the data. This transfer is performed when both sides are ready, which requires that the sender's buffer is full and the receiver's buffer is empty. Even though we are using DMA for data transfer, the processor will be busy waiting for the DMA to finish. This is obviously not very efficient, and this method should be seen as a transition to our third method.

To allow the DMA to work in parallel with the processing core, we have implemented a 'double-two-end-buffer' method, which introduces two "ping-pong" buffers on each side of the communication channel. This allows the cores to work on one local buffer while the data from the other buffer is transferred to the other core by means of DMA.

If the token production rate on the sending actor is equivalent to the token consumption rate on the receiving actor, it is expected that the 'double-two-end-buffer' method should be the most efficient. On the other hand, if there is a big imbalance in the production/consumption rate, all three buffering methods will suffer from blocking, after the buffer gets full.

We have implemented the broadcast capability in all three implementations of the communication API. This mechanism is used when the output channel of an actor needs to be connected to multiple input channels of the other actors. Hence, the same output data can be written to multiple input ports and actors. In all implementations, the synchronization between sender and receiver is achieved by using flags belonging to the buffers. These flags indicate the availability of the data or the empty space in the buffers. These flags are kept in the local memories of the cores due to being polled continuously in a busy waiting loop, during the read and write operations.

The `write` and `read` function calls are designed to be *asynchronous* (non-blocking calls) by default, however, they will be blocking if the buffer is full or empty respectively. The functions `end_of_transmission`, `connect`, and `disconnect` calls will always be blocking. More details of the communication library are described in a different work [19].

## V. Implementation

As our case study, we use the two-dimensional inverse discrete cosine transform (2D-IDCT), which is a component of MPEG video decoders. The CAL implementation of the 2D-IDCT is used as input to the code generator and as a reference implementation for the hand-written code. This implementation consists of 15 actors and these actors are mapped one-on-one to the Epiphany architecture using 15 out of 16 available cores. This implementation of the 2D-IDCT uses two one-dimensional inverse discrete cosine transforms after each

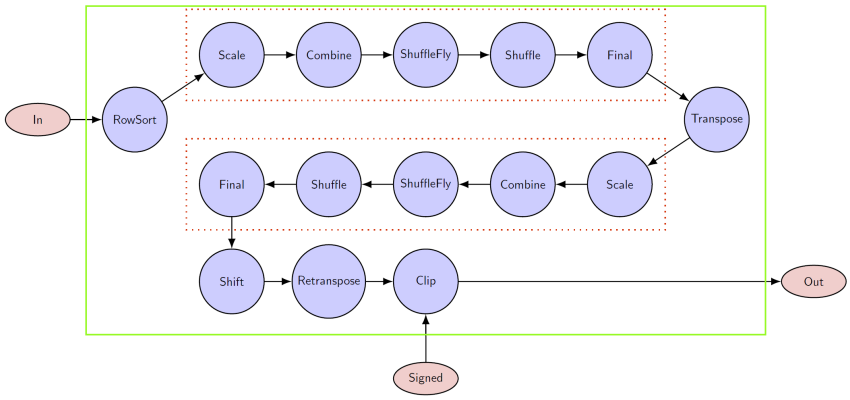


Fig. 3. The dataflow diagram of 2D-IDCT algorithm. The connections between actors, in blue, is done with 4 communication channels, while the external input, in red (In, Signed and Out), is done over a single channel.

other, with all actors connected in a pipeline fashion. One dimensional IDCTs are highlighted with the dotted rectangles in Figure 3.

Both hand-written and automatically generated implementations map the actors onto the cores by using the serpentine layout which can be seen in Figure 4. This layout takes into account the physical layout of the cores on the chip and the routing method of the architecture (X-Y routing in this case), so that consecutive actors are mapped into neighboring cores.

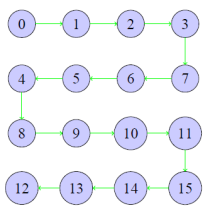


Fig. 4. Serpentine layout to map 2D-IDCT to cores on the chip. RowSort is mapped onto core 0, Clip is mapped onto the core13 whereas core 12 is not used.

### A. Hand-written Implementation

The hand-written implementation of 2D-IDCT is based on the reference CAL code. The implementation includes an embedded communication mechanism. This communication mechanism is similar to the 'one-end-buffer' implementation of the communication library presented in the previous section. However, the buffer size is 1 which means an actor can send only one data token at a time to a target actor. Due to the write operation being cheaper than the read operation on the NoC of the Epiphany architecture, the write function of the communication mechanism writes the data to the memory of the remote core and the read function of this mechanism reads the data from the local memory of the core. The read and the write operations are both blocking. The communicating cores use the flags of the ports in order to inform each other about the availability of data elements or empty space in the buffer. Two actors had input availability control for all input channels in the guard states. Therefore, these actors have been modified further to be able to combine multiple communication channels into 1 channel. There is no broadcasting functionality. Majority of the actors have 1/1 input/output ratio however there are two actors which have 1/2 input/output ratio and two actors which have 2/1 input/output ratio. This information is used in order to run each actor in a loop until they produce the expected number of outputs.

### B. Automatically Generated Implementations

The results from the initial and three different optimizations of our compilation framework can be found in Table I. The initial implementation, which is called as 'Uninlined' in Table I, is intended to be structurally similar to the original CAL actors. Actions are translated to two functions: *action_body* for the body of the action and a separate function, *action_guard*, to evaluate the conditions of the guard of the action. For proper initialization the variable declaration of the actions are pushed for both functions. In the action scheduler AMs *test* instruction are translated to function call to the *action_guard* and *test_input* functions, and *exec* instructions are translated to a call for *action_body*.

The second implementation, called as 'Inlined G' in Table I, optimizes the initial one by using a pass that inline all *action_guard* functions. Beside function inlining, the pass also analyses the scope of the variables and remove unused variable initializations.

The third implementation, 'Inlined G & B' in Table I, inlines both *action_guard* and *action_body* function calls. Like the second pass, third pass also use variable scope information

to eliminate unnecessary variable initializations. Thus, instead of copying the entire action variable initializations, only those variables which are used by the guard or the body of the action are initialized.

All generated implementations use the custom communication library. The parameters of the communication library such as buffer sizes, communication mechanism version and mapping layout are controlled by the code generator however for this evaluation they are kept fixed at the best configuration found after exploring the design space.

### *C. Differences Between Hand-written and Generated Implementations*

To be able to understand the reasons behind the performance differences and evaluate the code generation, we can note some important differences between the automatically generated and the hand-written implementations. These differences are:

- State machines, in the generated code, often include high number of branches and states.
- Actions are generated as separate functions whereas in the hand-written code, they are combined in one function. Function calls add overhead (due to dealing with the stack) and decrease the performance.
- Different communication mechanisms are used. The mechanism, that is used in the generated code, is meant to be generic, e.g broadcasting is supported which is not necessary for the particular application used as the case study. Hence the generated code includes communication mechanism overhead when compared to the hand-written code.
- Several further optimizations are implemented in the hand-written code, such as reduction of channel numbers. In the CAL implementation, actors are connected to each other with 4 channels. These channels are combined into one channel in the hand-written code in order to decrease network setup overhead and simplify the usage of channels.

## VI. Results and Discussion

We tested both of the implementations on the Epiphany III microchip running at 600 MHz. Among the different mechanisms of the communication library, we got the best results with the 'one-end-buffer' mechanism. Hence, we use the results of this mechanism for the discussions and the evaluation. In the test cases, we read the input data elements from the external memory and write the output elements back to the same external memory.

As can be seen in the last row of Table I the execution of the hand-written implementation takes 0.14 ms when the input size is 64 data elements (a block) and 93.6 ms when the input size is 64k data elements. These are the results which we aim to achieve with the automatically generated implementation.

Initially, the execution of the auto-generated implementation took around 1.2 ms for 64 data elements and around 405 ms for the 64k data elements. These results are given in the first row of Table I. The hand-written implementation out-performed the auto-generated implementation by 8.8x for 64 data elements and by 4.3x for 64k data elements. After further analysis, it is realized that the main bottleneck for both implementations is the external memory access. Most of the execution time is spent on reading the data from the external memory. In addition to the reading, writing to the external memory consumes some time as well. In the hand-written implementation, the external memory access for each input and output data is at a minimal level. However, in the auto-generated implementation, there were more external memory accesses per input and output elements due to both the communication library and the code generation.

In the auto-generated implementation, the variables, which were residing in the external memory, are moved to the local memory of the cores. This significantly decreased the execution time from 405 ms to 154 ms for 64k data elements, which resulted in a throughput increase by 63%. The performance of the auto-generated implementation got as close as 1.6x to the performance of the hand-written implementation. This optimization is referred as memory optimization in Table I.

TABLE I
EXECUTION TIMES AND CYCLE NUMBERS OF THE IMPLEMENTATIONS, USING SERPENTINE LAYOUT AND EXTERNAL MEMORY ACCESS (BUF SIZE: 256 FOR THE AUTO-GENERATED IMPS). (MEM OPT: MEMORY OPTIMIZATION G: ACTION_GUARD, B: ACTION_BODY, K: THOUSAND, M: MILLION)

| | **64 data elements** | **64k data elements** |
|---|---|---|
| **No mem opt + Uninlined** | 1.206ms (724k cyc) | 405.6ms (243M cyc) |
| **Mem opt + Uninlined** | 0.389ms (233k cyc) | 153.9ms (92M cyc) |
| **Mem opt + Inlined G** | 0.387ms (232k cyc) | 130.6ms (78M cyc) |
| **Mem opt + Inlined G&B** | 0.386ms (231k cyc) | 125.2ms (75M cyc) |
| **Manual** | 0.136ms (81k cyc) | 93.6ms (56M cyc) |

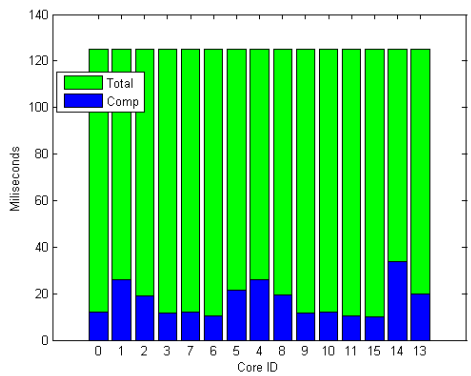


Fig. 5. Total execution times for the auto-generated implementation together with the times spent for the computation.

So far, the optimizations were done only on the communication library. Then, as the first optimization to the code generation, we inlined the functions which were the

correspondence of the guard conditions. The results, when this optimization is applied, can be seen in the third ('Mem opt + Inlined G') row of the Table I. As the second optimization on the code generator, we inlined the functions which corresponded to the action bodies. This optimization provided us a new auto-generated implementation and the results of this implementation corresponds to the fourth ('Mem opt + Inlined G & B') row of the Table I. By applying these optimizations to the code generation, we decreased the execution time from 154 ms to 125 ms for 64k data elements. This decrease in the execution time corresponds to 18% throughput increase. Using this result, the hand-written implementation shows 1.3x better performance. The difference is reduced further from 1.6x to 1.3x.

After the optimizations on the communication library, apart from the input and output data elements, no data element is kept in the external memory. There are a few data elements which reside in the shared memory and used for synchronization between the host and core0, however, they can be ignored since they are used only once before the cores start running. Even if the only access to the external memory is the access to input and output data elements we find that this external access is dominating the execution time. Figure 5 shows the total execution times of individual cores together with the time spent for the computation. The difference between the total time and the actual computation time gives the time that is spent on the communication and waiting for I/O operations due to the slow external memory accesses. In Figure 5 we can see that for most of the cores around 90% of this time (the difference of the total and the computation times) is spent on reading the input data elements from the external memory. The external memory accesses are performed by only the first and last cores. For 64k elements, the first core performs 64k external memory reads whereas the last core performs 1k external memory reads and 64k external memory writes. In the hand-written implementation, the first core runs for approximately 56 million cycles and 51.9 million of these cycles are spent for reading the input from the external memory. The situation is more or less the same in the auto-generated implementation. Due to the slow read operations in the first core, the other cores wait for the input data and this waiting time is the main contributor of the difference between the total and the computation times. The cost of the write operations, in the last core, which is far less than the cost of the read operations, are mostly hidden by the cost of the read operations. The cost of the communication between the cores and the computation times are mostly hidden by the cost of the external memory accesses. In summary, in this application, the cores are mostly blocked while reading the input data elements and they are blocked for a shorter time while writing the output data elements.

An additional aspect for comparison between the hand-written and the auto-generated implementation is the code size. In Table II the machine code size of each actor/core is given for both hand-written and auto-generated implementations. (The auto-generated implementation, which includes memory optimization and inlined action_guards & action_bodies, is used.) These code sizes are obtained by running the 'size' command in linux command line. The .elf files, which are produced by the Epiphany linker, are used as input parameters to the 'size' command. The auto-generated implementation has 71% more machine code when compared to the hand-written implementation.

TABLE II
MACHINE CODE SIZE (IN BYTES) OF EACH CORE FOR HAND-WRITTEN AND AUTO-GENERATED IMPLEMENTATIONS

| Cores / Actors | Hand-written | Auto-generated |
|---|---|---|
| **host** | 8,636 | 8,077 |
| **core0 / RowSort** | 5,488 | 9,020 |
| **core1 / Scale** | 5,556 | 9,568 |
| **core2 / Combine** | 5,012 | 9,372 |
| **core3 / ShuffleFly** | 5,080 | 8,888 |
| **core4 / Shuffle** | 5,556 | 9,452 |
| **core5 / Final** | 4,952 | 9,060 |
| **core6 / Transpose** | 5,708 | 10,628 |
| **core7 / Scale** | 5,244 | 9,568 |
| **core8 / Combine** | 5,008 | 9,376 |
| **core9 / ShuffleFly** | 4,920 | 8,892 |
| **core10 / Shuffle** | 5,080 | 9,456 |
| **core11 / Final** | 4,952 | 9,064 |
| **core12 / Shift** | 4,952 | 9,044 |
| **core13 / Retranspose** | 7,160 | 10,564 |
| **core14 / Clip** | 5,044 | 10,236 |
| **Total** | 87,712 | 150,265 |

Another interesting aspect is the difference in development effort writing the 2D-IDCT application in C and writing it in CAL. In Table III the number of source lines of code (SLOC) for each actor for the hand-written (native - C) implementation and the reference CAL implementation is compared. We see that in total, 495 SLOC are needed to implement the 2D-IDCT application in CAL while more than four times as many (2229 SLOC) is needed for the C implementation. This clearly indicates the expressiveness of the CAL language and the usefulness of the CAL compilation tool as described in this paper.

## VII. FUTURE WORK

For further evaluation of the code generation and the custom communication library, a more complex application such as the entire MPEG-4 decoder implementation can be both automatically generated and manually written. This work might need combining and splitting the actors. Hence the compilation framework may need to be extended. In addition to the evaluation of the code generation, the mapping approach proposed by Mirza et al. [20] can be tested with this application.

The performance of both hand-written and auto-generated implementations might be increased by further optimizations. We observed that the main bottleneck for both of the implementations is the read operations performed on the external memory. An optimization could be using the write operations

TABLE III
SOURCE LINE OF CODE OF EACH ACTOR FOR HAND-WRITTEN AND CAL IMPLEMENTATIONS. (THE NUMBER OF CODE LINES ARE THE SAME FOR THE DIFFERENT INSTANCES OF THE SAME ACTORS.)

| Actors + libs | Hand-written | CAL |
|---|---|---|
| **RowSort** | 190 | 57 |
| **Scale** | 142 | 26 |
| **Combine** | 148 | 48 |
| **ShuffleFly** | 116 | 22 |
| **Shuffle** | 143 | 41 |
| **Final** | 113 | 10 |
| **Transpose** | 214 | 71 |
| **Shift** | 109 | 9 |
| **Retranspose** | 203 | 54 |
| **Clip** | 175 | 45 |
| **Network + comm** | 333 | 112 |
| **Host** | 343 | – |
| **Total** | 2229 | 495 |

instead of read operations. Such as, instead of the Epiphany core(s) reading the input data from the external memory, the host can write the data to the Epiphany core's memory. The synchronization between the host and the Epiphany core can be obtained by using the blocking write operations. Hence no read operations would be used. Another optimization for the communication library would be customization of the functionality. E.g the operations such as broadcasting can be turned on and off for different applications in order not to add overhead when the operation is not needed. A radical optimization can be replacing the communication mechanism with a more efficient one which can make better use of the specific features of the architecture.

In our test application, each actor is connected only to the neighbor actors. Hence the communication is not very complicated, and choosing the best fitting layout is not a complex task. However, with larger applications when the communication patterns get more complicated, choosing the mapping layout, quickly becomes a very complex task. For e.g individual actors might be connected to several other actors. Hence the mapping method would need to take network usage, communication frequency between actors etc. into account while searching for the best fitting layout. Mirza et al. [20] propose a solution to the mapping, path selection and router configuration problems. They refer that their approach is capable of exploring a range of solutions while letting the designer to adjust the importance of various design parameters.

As an extension to the compilation framework we used in our work, new backends can be implemented and integrated to the compilation tools in order to target other manycore architectures and the code generation can be tested on these architectures.

## VIII. CONCLUSIONS

Manycore architectures are emerging to meet the performance demands of high-performance embedded applications and overcome the power dissipation constraints of the existing technologies. A similar trend is visible in programming languages in the form of dataflow languages that are naturally suitable for streaming applications. This paper deals with the evaluation of a compilation framework along with a custom communication library that takes CAL actor language (CAL) code as input and generates parallel C code targeting Epiphany manycore architecture. As a case study we have used a CAL implementation of a 2D-IDCT application and compare the automatically generated code from the proposed tool-chain with a hand-optimized native-C implementation.

The preliminary results reveal that the hand-written implementation has 4.3x better throughput performance with respect to the auto-generated implementation. After memory access optimizations on the communication library, the auto-generated implementation gained 63% throughput increase. With further optimizations on the code generation, the throughput of the auto-generated implementation was further improved by 18%. To sum up, we are able to decrease the difference in execution time between the hand-written and the auto-generated implementations from a factor of 4.3x to 1.3x. In terms of the development effort by considering source lines of code metric, we observe that the CAL based approach requires 4.5x less lines of source code when compared to the hand-written implementation.

To conclude we are able to achieve competitive results of execution time with respect to the hand-written implementation and the use of high-level language approach leads to reduced development effort. We also foresee that our compilation methodology will result in focusing on optimizing the tool-chain to produce efficient implementations rather than manual optimizations performed on each application.

## ACKNOWLEDGEMENTS

The authors would like to thank Adapteva Inc. for giving access to their software development suite and hardware board. This research is part of the CERES research program funded by the Knowledge Foundation and HiPEC project funded by Swedish Foundation for Strategic Research (SSF).

## REFERENCES

[1] Z. Ul-Abdin and B. Svensson, "Occam-pi for programming of massively parallel reconfigurable architectures," *International Journal of Reconfigurable Computing*, vol. 2012, no. 504815, p. 17, 2012.

[2] E. A. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[3] B. Bhattacharya and S. S. Bhattacaryya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.

[4] J. Eker and J. W. Janneck, "CAL language report specification of the CAL actor language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M03/48, 2003.

[5] Epiphany, "Epiphany architecture reference G3, rev 3.12.12.18," Adapteva, Tech. Rep., 2013.

[6] J. Eker and J. W. Janneck, "Dataflow programming in CAL – balancing expressiveness, analyzability, and implementability," in *Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), 2012*. IEEE, 2012, pp. 1120–1124.

[7] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, "Overview of the MPEG reconfigurable video coding framework," *Journal of Signal Processing Systems, Springer*, 2009.

[8] E. Gebrewahid, M. Yang, G. Cedersjö, Z. Ul-Abdin, J. W. Janneck, V. Gaspes, and B. Svensson, "Realizing efficient execution of dataflow actors on manycores," in *International Conference on Embedded and Ubiquitous Computing, IEEE*, 2014.

[9] J. Janneck, "A machine model for dataflow actors and its applications," in *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*. IEEE, 2011, pp. 756–760.

[10] M. Yang, S. Savas, Z. Ul-Abdin, and T. Nordström, "A communication library for mapping dataflow applications on manycore architectures," in *Sixth Swedish Workshop on Mutlicore Computing*, 2013.

[11] G. Roquier, M. Wipliez, M. Raulet, J.-F. Nezan, and O. Déforges, "Software synthesis of CAL actors for the MPEG reconfigurable video coding framework," in *15th IEEE International Conference on Image Processing, 2008. ICIP 2008.* IEEE, 2008, pp. 1408–1411.

[12] ORCC, "Open RVC-CAL compiler," http://orcc.sourceforge.net/, Accessed: 3 Aug 2013, 2013. [Online]. Available: http://orcc.sourceforge.net/

[13] D2C, "CAL ARM compiler," http://sourceforge.net/projects/opendf/, Accessed: 3 Aug 2013, 2013. [Online]. Available: http://sourceforge.net/projects/opendf/

[14] J. W. Janneck, "Tokens? what tokens? - a gentle introduction to dataflow programming," Programming Solutions Group, Xilinx Inc., Tech. Rep., 2007.

[15] A. Jones and M. Butts, "Teraflops hardware: A new massively-parallel mimd computing fabric ic," in *Proceedings of IEEE Hot Chips Symposium*, 2006.

[16] G. Cedersjö and J. W. Janneck, "Toward efficient execution of dataflow actors," in *Signals, Systems and Computers (ASILOMAR), 2012 Conference Record of the Forty Sixth Asilomar Conference on*. IEEE, 2012, pp. 1465–1469.

[17] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," *In Information Processing 77: Proceedings of the IFIP Congress*, 1977.

[18] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[19] M. Yang, "CAL code generator for epiphany architecture," Master thesis report 1401, Halmstad University, 2014.

[20] U. M. Mirza, F. Gruian, and K. Kuchcinski, "Design space exploration for streaming applications on multiprocessors with guaranteed service noc," in *NoCArc '13 Proceedings of the Sixth International Workshop on Network on Chip Architectures*, 2013.

# Appendix B

## Paper II
## **Dataflow Implementation of QR Decomposition on a Manycore**

Süleyman Savas, Sebastian Raase, Essayas Gebrewahid, Zain Ul-Abdin, and Tomas Nordström

*Fourth ACM International Workshop on Many-core Embedded Systems (MES '16)*, IEEE, 18-22 June, 2016, Seoul, South Korea.

# Dataflow Implementation of QR Decomposition on a Manycore

Süleyman Savas, Sebastian Raase, Essayas Gebrewahid,
Zain Ul-Abdin and Tomas Nordström
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
Suleyman.Savas@hh.se, Sebastian.Raase@hh.se, Essayas.Gebrewahid@hh.se,
Zain-Ul-Abdin@hh.se, Tomas.Nordstrom@hh.se

## ABSTRACT

While parallel computer architectures have become mainstream, application development on them is still challenging. There is a need for new tools, languages and programming models. Additionally, there is a lack of knowledge about the performance of parallel approaches of basic but important operations, such as the QR decomposition of a matrix, on current commercial manycore architectures.

This paper evaluates a high level dataflow language (CAL), a source-to-source compiler (Cal2Many) and three QR decomposition algorithms (Givens Rotations, Householder and Gram-Schmidt). The algorithms are implemented both in CAL and hand-optimized C languages, executed on Adapteva's Epiphany manycore architecture and evaluated with respect to performance, scalability and development effort.

The performance of the CAL (generated C) implementations gets as good as 2% slower than the hand-written versions. They require an average of 25% fewer lines of source code without significantly increasing the binary size. Development effort is reduced and debugging is significantly simplified. The implementations executed on Epiphany cores outperform the GNU scientific library on the host ARM processor of the Parallella board by up to 30x.

## 1. INTRODUCTION

Computer architectures are moving towards manycores for reasons such as performance and energy efficiency. The required parallelism to use these architectures efficiently requires new software tools, languages and programming models to abstract the differences between different architectures away. This reduces required knowledge about the architectures and their specific programming language extensions. Even with tools, writing efficient parallel applications is challenging and there is a lack of knowledge on performance of common applications such as QR decomposition when executed on manycores.

QR decomposition (QRD) [8] is one of the major factorizations in linear algebra. It is well known to be numerically stable and has many useful applications such as replacing matrix inversions to avoid precision loss and reduce the number of operations, being a part of the solution to the linear least squares problem and being the basis of an eigenvalue algorithm (the QR algorithm).

In this paper, we evaluate Cal2Many [6] source-to-source compiler, which translates CAL [5] code to native code for multiple manycore architectures. As a case study, we implemented three QRD algorithms (Givens Rotations, Householder and Gram-Schmidt) both in CAL and in native C for Adapteva's Epiphany [11] architecture. All implementations use our own communications library [13]. We used the Parallella platform to evaluate our implementations in terms of performance, development effort and scalability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*MES '16, June 19 2016, Seoul, Republic of Korea*

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4262-9/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2934495.2934499

## 2. BACKGROUND

### 2.1 CAL Actor Language

The CAL actor language is a dataflow language consisting of actors and channels. Actors are stateful operators which execute code blocks (actions), take inputs and produce outputs usually with changing the state of the actor. The channels are used to connect the actors to each other. Therefore, interaction among actors happens only via input and output ports. CAL actors take a step by 'firing' actions that satisfy all the required conditions. These conditions depend on the value and the number of input tokens, and on the actor's internal state. The actors are instantiated and connected to each other via Network Language (NL) included in CAL.

### 2.2 Cal2Many

The Cal2Many compilation framework contains two intermediate representations (IRs): Actor Machines (AM) [10] and Action Execution IR (AEIR) [6]. Each actor is first translated to an AM, which describes how to schedule execution of actions. To execute AM, its constructs have to be transformed to a different programming language constructs, which have different implementations in different programming languages on different platforms. To stay language -agnostic and get closer to a sequential action scheduler, *AEIR* is introduced.Epiphany backend generates C code using our custom communications library and generates channels and mapping of actor instances by using the NL.

### 2.3 QR Decomposition

QR decomposition is decomposition of a matrix into an upper triangular matrix $R$ and an orthogonal matrix $Q$. The

equation of a QRD for a square matrix $A$ is simply $A = QR$. The matrix $A$ does not necessarily need to be square. The equation for an $m$ x $n$ matrix, where $m \geq n$, is as follows:

$$A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1.$$

We have implemented three QRD algorithms (Givens Rotations, Householder and Gram-Schmidt) in both CAL and native C for the Epiphany architecture.

## 2.4 Epiphany Architecture

Adapteva's manycore architecture is a two-dimensional array of cores connected by a mesh network-on-chip [11]. It operates in a shared, flat 32-bit address space. The network-on-chip is made of three meshes called *rMesh*, *cMesh* and *xMesh*. The former is used exclusively for read requests, while the latter two carry write transactions destined for on-chip and off-chip, respectively. The mesh uses a static XY routing algorithm.

Each core contains a single-precision floating-point RISC CPU, 32KB of local memory, a two-channel DMA engine and a mesh interface. Two event timers allow cycle accurate measurements of different events.

We have used the Parallella-16 platform, which contains a 16-core Epiphany-III running at 600 MHz in addition to a dual-core ARM Cortex-A9 host processor running at 667 MHz.

## 2.5 Communication Library

We have implemented a custom communications library for the Epiphany architecture [13], which is used in all implementations. It is centered around token-based, unidirectional communication channels. These channels support blocking *read*, *write* and non-blocking *peek* operations and allow checking the current number of immediately read- or writable tokens for block-free operations.

A global table describes all channels in the system. Local data structures and buffers are allocated at run-time by each core. Channels are implemented as ring buffers and it is possible to use the event timers to gauge blocking overhead.

# 3. RELATED WORKS

There are many approaches for QRD, however, they focus either on architectures or scheduling and use only one algorithm. We have executed three algorithms in two programming languages and compared them to each other.

Buttari et al. [3] present a QRD method where they run a sequence of small tasks on square blocks of data and use a dynamic scheduling mechanism to assign tasks to computational units. We execute small tasks in a dataflow fashion.

Hadri et al. [9] present a QRD method for shared-memory multicore architectures and modified an existing algorithm to perform panel factorization in parallel. They aim tall or small square matrices whereas we aim square matrices.

Agullo et al. [2] implement a three step QRD on a multicore CPU which is enhanced with multiple GPU cores. They divide the decomposition into a sequence of tasks as we have done. Then they schedule these tasks on to individual computational units. Their approach suffers if the number of CPUs and GPUs are not the same.

While the CAL2C compiler [15] generates sequential C code, the Open-RVC CAL Compiler (ORCC) [12] and d2c [4] compilers generate multi-threaded C code, but require dedicated run-time system libraries. Our compilation framework generates separate C code for each actor instance to be executed on individual cores and does not require any run-time system support.

# 4. IMPLEMENTATIONS

All implementations use the same library to implement communication between the Epiphany cores, which is used by the Cal2Many as well. However, the communication characteristics differ naturally between algorithms and, to a lesser extent, between the hand-written (C) and the generated (CAL) implementations of the same algorithm.

## 4.1 Givens Rotations

The Givens Rotations (GR) algorithm applies a set of unitary rotation $G$ matrices to the data matrix $A$. In each step, one of the sub-diagonal values of the matrix $A$ is turned into zero, forming the $R$ matrix. The multiplication of all rotation matrices forms the orthogonal $Q$ matrix.

We implemented Givens Rotations (GR) with a modified Gentleman-Kung systolic array [7] [16] using 1, 5 and 12 cores individually. In parallel versions, two cores are used for distributing and collecting inputs and outputs, the rest of the cores are used for computations.

The implementations consist of 4 types of units named as cells. *Boundary* and *inner* cells perform the computation while *splitter* and *joiner* cells distribute and collect data. Figure 1 gives the layout of the systolic array mapped on the 4x4 core matrix of the Epiphany architecture. The inputs are read row by row. Each row is divided into four pieces and distributed to the first row of 4x4 Epiphany core matrix. Token size is defined as the size of each piece and the cores send and receive one token at a time for communication. Apart from the input elements, c, s and final r values are communicated.

Each cell calculates one $r$ value. For $4 \times 4$ matrix, each cell is mapped onto an Epiphany core. However for $16 \times 16$ matrix, 16 *boundary* and 120 *inner* cells are required. Therefore the implementations are modified to combine a number of cells in one single core. Each core has to store the calculated $r$ values which becomes a problem when large matrices are used. For $512 \times 512$ input matrix, an inner core consists of $128 \times 128$ *inner* cells which results in storing a $128 \times 128$ $r$ matrix. Since r is a *float*, the required memory is 4 x 128 x 128 = 65536 bytes. Due to the local memory limitation, the largest matrix size that can be decomposed with Givens Rotations method is $256 \times 256$. The implementations can scale when the number of cores increases, i.e. with 64 cores the implementations can decompose a $1024 \times 1024$ matrix.

## 4.2 Householder Transformation

The Householder (HH) algorithm describes a reflection of a vector across a hyperplane containing the origin [18].

In our implementation, the Epiphany cores are connected as a one-dimensional chain of processing elements. Each core handles an equal amount of matrix columns and runs the same program. The communication is wave-like and next-neighbor only.

First, the input matrix is shifted into the processing chain, column by column, until it is fully distributed among the cores. Then, the last core in the chain computes a reflection vector $w$ for each of its columns, updates them, and sends the vector towards the beginning of the chain, forming a

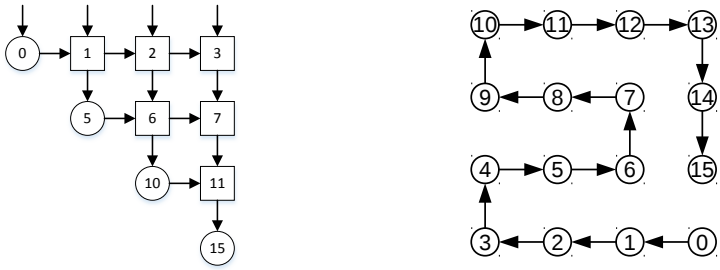


**Figure 1: Mapping layout of all implementations. Givens rotations on the left, Householder and Gram-Schmidt on the right. On left, circles are boundary cells, squares are inner cells. On right, each core executes the same code. No shape relation across layouts.**

communication wave each. All previous cores forward these vectors and update their own columns. After these waves have been forwarded by the penultimate core, it – after updating its own columns – computes its own reflection vectors and sends them along, forming new waves. When a core has sent its last reflection vector, it will form a final set of waves containing its result columns.

### 4.3 Gram-Schmidt

The Gram-Schmidt (GS) algorithm produces the upper-triangular matrix $R$ row-by-row and the orthogonal matrix $Q$ as a set of column vectors $q$ from the columns of the data matrix $A$ in a sequence of steps. In each step, we pick a column $a$ of the matrix $A$. The dot product of this column with itself is calculated. Then, the square root of the result is taken to generate an element of matrix $R$. This element is later used to normalize the column $a$ to produce a column of matrix $Q$. Then the column of matrix $A$ is updated by subtracting a multiple of vector $q$ with a value from matrix $R$ to produce an orthogonalized vector that is then used to compute the next columns of matrix $R$.

Both CAL and C implementations work as a chain of processes that work on a certain number of columns, depending on the number of processes and the matrix size. All processes perform the steps required to compute a column of matrix $R$. In the first step, each process stores its local columns and pushes the remaining columns into the chain. In the second step, the processes read and forward the previous orthogonalized vectors after they use the vectors to produce the elements of matrix R and to update their local columns. In the third step, the processes compute and forward the local, orthogonalized vectors using their updated local columns. In the final step, the processes forward the elements of matrix $R$. The implementation can scale up to any number of processors $num_p$ and any $m$ x $n$ matrix, where n is a multiple of $num_p$.

## 5. RESULTS & DISCUSSION

Our implementations are executed on the Parallella board and tested with different number of cores and different sized square matrices.

### 5.1 Performance Analysis

There are several aspects such as memory usage, input size and number of cores which affect the performances. Therefore we executed the implementations with different configurations and Table 1 presents the number of execution cycles while using external and internal memories to store input and output matrices. Additionally, last two columns give the number of cores used and the total code size in bytes.

As a reference point, we measured the execution time of a QRD implementation, that is a part of GNU Scientific Library [1], on the ARM processor of Parallella board. The library implementation uses Householder approach with double precision, whereas the Epiphany architecture supports only single precision. Decomposition of a $128 \times 128$ matrix on a single ARM core takes 90 milliseconds whereas on the Epiphany cores it takes 6.9, 4.1 and 2.9 milliseconds (shown in Table 1) for hand-written parallel GR, GS and HH implementations respectively. GR implementation is the only one that can decompose $256 \times 256$ matrices. While decomposing $128 \times 128$ matrices, it outperforms the library by 13x, however, with $256 \times 256$ matrices this number increases to 23x. When the matrix sizes increase, parallel implementations perform better due to increased computation/communication ratio as a result of communication patterns used in the implementations.

We implemented a message passing mechanism for inter-core communication as a library. If messages (or as we call them, 'tokens') are passed very frequently, the overhead of communication dominates the execution time due to library calls and copy operations. The communication patterns we use in our implementations keep the number of message passes constant regardless the matrix size. If the matrix sizes increase, instead of increasing the number of messages, the size is increased. By keeping the number constant, we avoid extra overhead of calling library functions. The communication cost still increases due to increased size of data that needs to be copied. However, it does not increase as fast as the cost of computation. Hence its effect on overall execution time decreases.

Figure 2 shows performance of each hand-written algorithm decomposing a $64 \times 64$ input matrix that is stored in the internal memory. We chose this matrix size because

| | External mem | Local mem | SLoC | #cores | Footprint |
|---|---|---|---|---|---|
| **GR Hand-written** | 9.51 M (15.8 ms) | 4.16 M (6.9 ms) | 647 | 12 | 71 k |
| **GR CAL** | 10.45 M (17.4 ms) | 6.11 M (10.1 ms) | 400 | 12 | 72 k |
| **HH Hand-written** | 10.45 M (17.4 ms) | 1.76 M (2.9 ms) | 219 | 16 | 225 k |
| **HH CAL** | 10.69 M (17.8 ms) | 2.00 M (3.3 ms) | 170 | 16 | 223 k |
| **GS Hand-written** | 11.17 M (18.6 ms) | 2.47 M (4.1 ms) | 188 | 16 | 179 k |
| **GS CAL** | 11.40 M (19 ms) | 2.70 M (4.5 ms) | 160 | 16 | 193 k |

**Table 1: Execution times (in clock cycles and milliseconds), source lines of code, number of used cores and code size in bytes, $128 \times 128$ matrix. GR = Givens Rotations, HH = Householder, GS = Gram-Schmidt**

it is the biggest size that is supported by all implementations with different core numbers (except for the single core implementation of Gram-Schmidt algorithm due to memory limitations). Gram-Schmidt implementation can decompose $64 \times 64$ matrix by using 2, 4, 8 and 16 cores and achieve 4x speed-up by going from 2 core to 16 cores. Householder implementation can decompose on 1, 2, 4, 8 and 16 cores and achieve 5.2x speed-up going from single core to 16 cores. Givens Rotations implementation decomposes the same matrix on 1, 3 and 10 computational cores due to it's structure and achieves 3.4x speed-up while going from single core to 10 cores. When decomposing small sized matrices such as $64 \times 64$, communication overhead plays a significant role in the execution time and decreases the speed-up. However, as the matrix size increases, the effect of communication decreases.

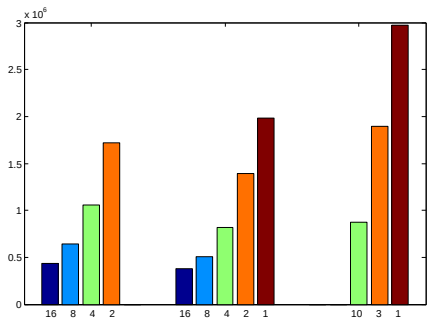


**Figure 2: Execution cycles for GS, HH and GR respectively with different number of cores and 64x64 input matrix. X axis represent number of cores and Y axis represents number of clock cycles.**

In our previous works [17, 14] we have experienced that the external memory access can be a bottleneck in the Epiphany architecture due to the slow link between the memory and the processing cores. Therefore we tested the implementations with and without using the external memory for storing the input matrix. We observed that when the input size increases, which means increased computation, the influence of external memory decreases however, it is still a bottleneck. Table 1 shows that using external memory to store the input matrix slows down the execution by 56% to 83% depending on the implementation. Givens Rotations seems to be the least influenced algorithm due to overlap between memory reads and computation.

An interesting point is the comparison of C and CAL implementations. Looking at Table 1, one can see that there is not much difference between hand-written and CAL implementations while using external memory. Even while using internal memory, the difference increases only for the GR implementation due to a slightly more complicated structure compared to the other implementations such as different communication pattern or having different number of channels depending on the position of the core. These small details increase the number of actions which are converted into functions in C. Overhead of repetitive calls to these functions increases the execution time. The main reasons of slow down for the generated code is having a scheduler and function calls. In the hand-written code, the actions are combined in the main function and there is no scheduler. Therefore there is neither function call overheads nor switch statements to arrange the order of these functions.

## 5.2 Productivity Analysis

In addition to performance, we compare the CAL and C implementations in terms of development effort, which is difficult to measure. It should be noted that our approaches to QRD have been implemented by three developers, who have more knowledge and experience with C rather than CAL. Nonetheless, the CAL implementations required about 25% less source code, while the binary code size stays approximately the same. The numbers shown in Table 1 do not include the additional source code for the ARM host or the communication library, since it is similar in all cases and used by the Cal2Many code generator as well.

In each case, about half of the development time was spent on understanding both the problem and the algorithm, which is independent of the choice of programming language. The actual implementation times for each algorithm varied. One of the developers had no prior experience with CAL and required approximately 20% more time for the CAL version, while the other developers required slightly more time for their C implementations. While this is by no means a hard measure, it provides an idea on the complexity of CAL. More importantly, the CAL implementations are completely oblivious of the underlying hardware and are easily portable, while the C implementations are quite restricted to the Epiphany system architecture. This higher level of abstraction also reduced the debugging effort, which is extremely cumbersome in low-level parallel programming.

## 6. CONCLUSIONS

Parallel implementations show up to 30x better performance in terms of execution time when compared to the library implementation. However, the Givens Rotations implementation shows that with bigger matrices the speed-up increases. Since the implementations are scalable, we believe that with larger local memory or larger number of cores the implementations can decompose bigger matrices and achieve higher speed-ups.

While using the external memory, Givens Rotations is slightly better than the other implementations, however, Householder method outperforms the others when local memory is used. Givens Rotations has higher amount of computation and more overlap between memory reads and computation. High computation amount increases the execution time when local memory is used. However, when external memory is used, due to the overlap, it shows the best performance. One should keep in mind that the number of cores is smaller for Givens Rotations implementation. In case of development complexity, Table 1 shows that implementing Givens Rotation requires more coding whereas the other implementations have similar code sizes.

As an average result of the three implementations, the generated code runs 4.3% slower than the hand-written code while using the external memory. When the internal memory is used, the slowdown is around 17% whereas the average source lines of code that is needed for the CAL implementations is 25% smaller. When the required knowledge level and development time and complexity is taken into account, the slowdown seems reasonable. It is easier to develop and debug parallel applications in CAL rather than in low level languages provided by manycore developers. The custom communication library reduces the burden however, it does not help with debugging and requires debugging itself.

In the future, Parallella board with 64 cores can be used for gaining higher speed-ups, and in order to decrease the effect of communication even further, direct memory access feature of Epiphany architecture can be analyzed.

## Acknowledgment

This work was supported in part by the ESCHER Project funded by the Swedish Knowledge Foundation, the HiPEC project funded by the Swedish Foundation for Strategic Research, and the ELLIIT Strategic Research Initiative funded by the Swedish Government.

## 7. REFERENCES

[1] Gnu scientific library [online accessed] http://www.gnu.org/software/gsl/, March 2016.

[2] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 932 –943, may 2011.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[4] D2C. CAL ARM compiler. http://sourceforge.net/projects/opendf/, Accessed: 3 Aug 2013, 2013.

[5] J. Eker and J. W. Janneck. Dataflow programming in CAL – balancing expressiveness, analyzability, and implementability. In *Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), 2012*, pages 1120–1124. IEEE, 2012.

[6] E. Gebrewahid, M. Yang, G. Cedersjo, Z. Ul-Abdin, V. Gaspes, J. W. Janneck, and B. Svensson. Realizing efficient execution of dataflow actors on manycores. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 321–328. IEEE, 2014.

[7] W. M. Gentleman and H. Kung. Matrix triangularization by systolic arrays. In *25th Annual Technical Symposium*, pages 19–26. International Society for Optics and Photonics, 1982.

[8] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[9] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tile qr factorization with parallel panel processing for multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[10] J. W. Janneck. Actor machines - a machine model for dataflow actors and its applications. *Department of Computer Science, Lund University, Tech. Rep. LTH*, pages 96–2011, 2011.

[11] A. Olofsson, T. Nordström, and Z. Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *48th Asilomar Conference on Signals, Systems and Computers*, 2014.

[12] ORCC. Open RVC-CAL compiler. http://orcc.sourceforge.net/, Accessed: 3 Aug 2013.

[13] S. Raase. A dataflow communications library for adapteva's epiphany. Technical report, Halmstad University, 2015.

[14] S. Raase and T. Nordström. On the use of a many-core processor for computational fluid dynamics simulations. *Procedia Computer Science*, 51:1403–1412, 2015.

[15] G. Roquier, M. Wipliez, M. Raulet, J.-F. Nezan, and O. Déforges. Software synthesis of CAL actors for the MPEG reconfigurable video coding framework. In *15th IEEE International Conference on Image Processing, 2008. ICIP 2008.*, pages 1408–1411. IEEE, 2008.

[16] S. Savas. Linear algebra for array signal processing on a massively parallel dataflow architecture. Technical Report ID: 2082/3452, Halmstad University, 2008.

[17] S. Savas, E. Gebrewahid, Z. Ul-Abdin, T. Nordström, and M. Yang. An evaluation of code generation of dataflow languages on manycore architectures. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–9, 2014.

[18] B. Zhou, R. P. Brent, et al. Parallel implementation of qrd algorithms on the fujitsu ap1000. *17th Australian Computer Science Conference, Christchurch, New Zealand, January*, 1994.

# Appendix C

## Paper III
## **Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis**

Süleyman Savas, Erik Hertz, Tomas Nordström, and Zain Ul-Abdin

submitted to *IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2017)*, Bochum, Germany, July 3-5, 2017.

# Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis

Süleyman Savas, Erik Hertz, Tomas Nordström, Zain Ul-Abdin
School of Information Technology, Halmstad University, Halmstad, Sweden

{suleyman.savas, erik.hertz, tomas.nordstrom, zain-ul-abdin}@hh.se

***Abstract*—This paper proposes a novel method for performing division on floating-point numbers represented in IEEE-754 single-precision (binary32) format. The method is based on an inverter, implemented as a combination of Parabolic Synthesis and second-degree interpolation, followed by a multiplier. It is implemented with and without pipeline stages individually and synthesized while targeting a Xilinx Ultrascale FPGA.**

**The implementations show better resource usage and latency results when compared to other implementations based on different methods. In case of throughput, the proposed method outperforms most of the other works, however, some Altera FPGAs achieve higher clock rate due to the differences in the DSP slice multiplier design.**

**Due to the small size, low latency and high throughput, the presented floating-point division unit is suitable for high performance embedded systems and can be integrated into accelerators or be used as a stand-alone accelerator.**

## I. Introduction

Floating-point division is a crucial arithmetic operation required by a vast number of applications including signal processing. For instance, the advance image creating sensors in synthetic aperture radar systems perform complex calculations on huge sets of data in real-time and these calculations include interpolation and correlation calculations, which consist of significant amount of floating-point operations [1], [2]. However, it is quite challenging to implement efficient floating-point arithmetics in hardware. Division itself is the most challenging basic operation to implement among the others such as addition, multiplication and subtraction. It requires larger area and usually achieves relatively lower performance.

In this paper, we implement a single-precision division method that performs division on floating-point numbers represented in IEEE-754 standard [3]. The implementations are synthesized and executed on a field-programmable gate array (FPGA) for validation and evaluation.

The proposed method consists of two main steps. The first step is the inversion of the divisor and the second step is the multiplication of inverted divisor and the dividend. In the inversion step Harmonized Parabolic Synthesis is used as the approximation method, which is a combination of Parabolic Synthesis and Second-Degree Interpolation [4], [5]. When compared to other methods, Parabolic Synthesis methodology converges faster and entails faster computation and smaller chip area, which in turn leads to lower power consumption. In the Harmonized Parabolic Synthesis methodology the Second-Degree Interpolation achieves the required accuracy by using intervals. The accuracy increases with the number of intervals and for single-precision floating-point inversion 64 intervals are sufficient.

The division hardware that is implemented in this paper is a part of an accelerator that performs cubic interpolation on single-precision floating-point numbers. The accelerator is implemented in Chisel language [6] to be integrated to a RISC-V [7] core via Rocketchip [8] system on chip generator.

The accelerator and the division hardware will be used as basic blocks for building domain-specific heterogeneous manycore architectures. Based on the requirement of applications, these custom blocks or other similar blocks will be integrated to simple cores. Many of these cores together with many custom blocks will form efficient heterogeneous many-core architectures targeting specific application domains. If an application does not use all of the cores with custom blocks, they can be shut-down by utilizing dark-silicon concept[9].

The rest of this paper is structured as follows: In section II we provide background knowledge on single-precision floating-point representation and the floating-point division algorithm that we have used. Section III discusses related work. Section IV presents the approach that we have used for implementing our algorithm. The implementation on an FPGA board is explained in details in Section V. Section VI provides the results which are compared to related work in section VII. Section VIII concludes the paper with conclusions and future work.

## II. Background

In this section, we introduce the FPGA, floating-point number representation and the method used for performing division.

### A. FPGA Features

The target platform in this paper is Virtex UltraScale XCVU095-2FFVA2104E FPGA from Xilinx. This FPGA is developed with 20 nm technology and consists of 537600 look-up tables (LUTs), 1075200 flip-flops (FFs), 1728 block RAMs and 768 DSP slices. Block rams are dual port with the size of 36 Kb and can be configured as dual 18 Kb blocks. The DSP slices include $27 \times 18$ two's complement multiplier.

### B. Binary32 - Single-precision floating-point numbers

Even though there are many possible floating-point number formats, the leading format by far is the IEEE Standard

for Floating-Point Arithmetic (IEEE-754) [3]. This standard defines the representation of floating-point numbers and the arithmetic operations. In this paper, we will focus on binary32 numbers more commonly known as single-precision floating-point numbers.

The IEEE 754 standard specifies a binary32 (single-precision floating-point) number as having:

- Sign bit $s$: 1 bit
- Exponent width $e$: 8 bits
- Significand (mantissa) $m$: 24 bits (23 explicitly stored)

as illustrated in Figure 1.

Fig. 1. Single-Precision (binary32) floating-point representation in IEEE-754 format

The sign bit determines the sign of the number. The exponent is an 8-bit signed integer and biased by $+127$. The true significand consists of 23 visible bits to the right of the decimal-point and 1 invisible leading bit to the left of decimal-point which is 1 unless the exponent is 0. The real value is calculated with the following formula:

$$(-1)^s \times (1+m) \times 2^{e-127}. \quad (1)$$

The exponent ranges from $-126$ to 127 because $-127$ (all zeros) and 128 (all ones) are reserved and indicate special cases. This format has the range of $\pm 3.4 \cdot 10^{38}$.

### C. Floating-point division

An overview of division algorithms can be found in [10]. According to the author's taxonomy, division algorithms can be divided into five classes: digit recurrence, functional iteration, very high radix, table look-up, and variable latency. These algorithms will differ in overall latency and area requirements. The algorithm we use, is a table look-up algorithm with an auxiliary function for decreasing the table size.

We implement the division $R = X/Y$ as an inversion of $Y$ ($T = 1/Y$) followed by a multiplication of $X$ ($R = X \cdot T$), as shown in Figure 2.

The floating-point multiplier [11] uses on-board DSP slices to multiply the significands and adder slices (carry logics) to add the exponents. The inverter utilizes the efficient Harmonized Parabolic Synthesis method [4], [5]. The Parabolic Synthesis method is founded on a multiplicative synthesis of factors, each of them a second-order function. The more factors that are used, the higher is the accuracy of the approximation.

## III. Related Works

There has been extensive research on implementing floating-point operations in hardware. The research has been focused

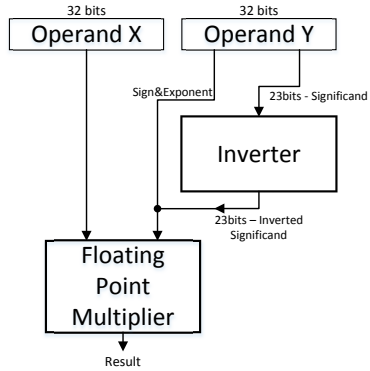


Fig. 2. Overview of the division method. X = dividend, Y = divisor

on three aspects namely performance, area, and error characteristic. In the rest of this section, we first present approximation methods for calculating the inversion and then present the prior works about implementing division with single-precision.

There are several methodologies such as CORDIC [12], [13], Newton-Raphson [14], [15] and polynomial approximation [16], [17] for calculating the inverse of a number. However, these methods are additive, whereas the method used in these paper, Parabolic Synthesis, is multiplicative. This means that the approximation converges faster, which results in faster and smaller implementation.

With the introduction of the Parabolic Synthesis methodology, the following improvements were accomplished compared to CORDIC. First, due to a highly parallel architecture, a significant reduction of the propagation delay was achieved, which also leads to a significant reduction of the power consumption. Second, the Parabolic Synthesis methodology allows full control of the characteristics and distribution of the error, which opens an opportunity to use shorter word lengths and thereby gain area, speed and power.

Singh and Sasamal [18] implement single-precision division based on Newton-Raphson algorithm using subtractor and multiplier, which is designed using Vedic multiplication technique [19]. They use a Spartan 6 FPGA and require quite high amount of hardware resources when compared to the proposed method. Leeser and Wang [20] implement floating-point division with variable precision on a Xilinx Virtex-II FPGA. The division is based on look-up tables and taylor series expansion by Hung et al. [21], which uses a 12.5KB look-up table and two multiplications. Regardless of the FPGA, the memory requirement is more than the proposed implementations. It is difficult to compare the resource utilization as the underlying architecture is different between Virtex-II and Ultrascale series, however, they seem to use more resources than the proposed implementations.

Prashanth et al. [22], Pasca [23], and Detrey and De Dinechin [24] implement different floating-point division algorithms on Altera FPGAs. Even though it is very difficult to

do a comparison between these works and the proposed work in terms of the utilization and timing results, one can still see the clear difference in Table I.

Prashanth et al. [22] design a single-precision floating-point ALU including a non-restoring division block consisting of shifters, adders, and subtractors. In contrast to proposed implementations, they do not use DSPs or block RAMs and have quite high cycle count requirement. Pasca [23] presents both single-precision and double precision division architectures based on Newton-Raphson and piece-wise polynomial approximation methods. His work focuses on correct rounding, which comes with a cost of several extra operations whereas rounding has no cost in the proposed implementations. Additionally, memory requirement is more than the requirement of the proposed implementations. Detrey and De Dinechin [24] compute the mantissa as the quotient of the mantissas of the two operands by using a radix 4 SRT algorithm [25]. The exponent is the difference between the two exponents plus the bias. Their work does not utilize any block RAMs or DSPs, however, the requirements for the other resources are significantly higher than the method implemented in this paper.

The methodology, proposed in this paper, is based on an inversion block and a floating-point multiplication block. The novelty lies in the inversion block, which uses the Harmonized Parabolic Synthesis method. This block uses three look-up tables, 4 integer multipliers and 3 integer adders, all with different word lengths. The look-up tables consist of 64 words with 27, 17 and 12 bits of word length respectively. These are significantly smaller compared to other table look-up methods for the same accuracy.

## IV. Methodology

The proposed method consists of two main steps as presented in Figure 2. The first step is the inversion of the divisor and the second step is the multiplication of the inverted divisor and the dividend. Both divisor and dividend are in IEEE-754 single-precision floating-point (binary32) format.

In the first step, as seen in Figure 3, the sign bit, the exponent bits, and the significand bits of the divisor are extracted. The significand bits are forwarded to the inverter where the Harmonized Parabolic Synthesis (approximation) is performed. In this block, the coefficients of the synthesis method are stored in look-up tables.

In parallel to the inversion, sign of the exponent is inverted while taking the bias into account as follows:
removing the bias:

$$e' = e - 127 \tag{2}$$

inverting the sign and adding the bias:

$$e'' = 127 - e' \tag{3}$$

when equations 2 and 3 are combined:

$$e'' = 254 - e \tag{4}$$

The exponent is further decreased by 1 unless the result of the inverter is 0.

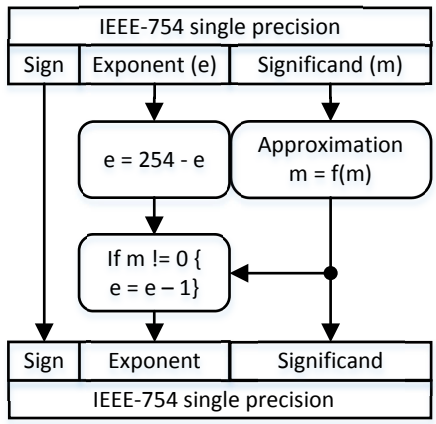


Fig. 3. Block diagram of the inversion algorithm

Sign bit, exponent with inverted sign, and the inverted significand are combined to form a floating-point number. This number is fed as the first input to a floating-point multiplier, which performs the second step of our method. The dividend is forwarded directly to the multiplier as the second input (shown in Figure 2).

In the multiplier, the significands are multiplied and adjusted, the exponents are added together and adjusted based on the resulting significand. The multiplication result of the significands is rounded with different rounding methods such as 'round toward zero', 'round to $+\infty$', and 'round to nearest (ties to even)'. 'Round to zero' provides the smallest max error and best error distribution when compared to other methods.

The inverter has been tested for every possible input. The max error is $1.18358967 \times 10^{-7}$ ($\approx 2^{-23.01}$) which is smaller than the machine epsilon ($\epsilon$) (upper bound for the error) that is commonly defined as $2^{-23}$ (by ISO C, Matlab, etc) for the single-precision floating-point format.

The division result, which is the output of the floating-point multiplication block, is tested for $2^{23}(8,388,608)$ inputs. The rounding errors for the tested inputs are smaller than 1 $ULP$ (unit in the last place) [26], that is used as a measure of accuracy in numeric calculations and calculated as follows:

$$ULP(x) = \epsilon \times 2^{e'} \tag{5}$$

where $x$ is the result in the single-precision floating-point format, $\epsilon$ is machine epsilon and $e'$ is the (unbiased) exponent of $x$.

## V. Implementation Details

The division hardware is implemented in Chisel language and Verilog code is generated. In the generated code, the coefficients were produced via constant wires and multiplexers. The code is modified to reside the coefficients in a block ram.

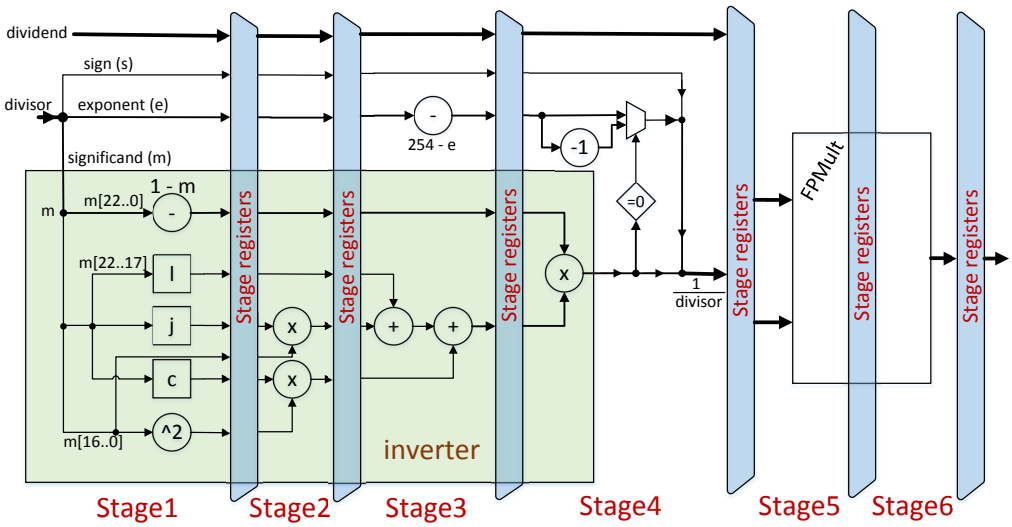


Fig. 4. Overview of the 6 stage pipelined division hardware with detailed view of the inversion component

Synthesis is performed by Xilinx tools. The target platform is Xilinx Virtex UltraScale XCVU095-2FFVA2104E FPGA.

Two different implementations of the floating-point division hardware are presented in this paper. The first implementation consists of a single stage, whereas in the second implementation, the inverter is divided into 4 stages and the floating-point multiplier is divided into 2 stages to increase the clock frequency. Consequently the dividend, sign, and exponent bits of the divisor are delayed for four cycles before being fed to the floating-point multiplier. When the inverter and the multiplier are combined the total number of stages in the second implementation becomes 6.

The stages and other details of the hardware implementation of the first step are given in Figure 4. As shown in this figure, two 8-bit subtractors, a single 23-bit equation logic and a single multiplexer are used for computing the exponent of the inverted divisor. These components correspond to the flow of the exponent in Figure 3.

The significand bits of the divisor are connected to the input of the inverter. As shown in Figure 4, the inverter block consists of three look-up tables for storing the coefficients, four integer multipliers, two adders, and a single subtractor. Each look-up table stores 64 words, however, the length of the words change due to the changes in number of bits for storing the coefficients. The word lengths for $l$, $j$, and $c$ coefficients are 27, 17, and 12, respectively. When synthesized, these look-up tables are stored in one block ram. The most significant 6 bits of the significand are used for addressing these look-up tables. The same address is fed to each table. Rest of the bits are fed to two multipliers one of which is actually a squarer.

Each integer multiplication unit in the inverter utilizes one DSP slice on the FPGA except the last unit, which requires two DSP slices due to having large inputs. The input and output sizes of the multipliers are $17 \times 12 \rightarrow 17$, $17 \times 17 \rightarrow 18$, $17 \times 17 \rightarrow 17$, and $23 \times 25 \rightarrow 24$. The subtractor and the two adders utilize 6, 6, and 7 carry logics, respectively.

The floating-point multiplier utilizes two DSP slices for multiplying the significands and an 8-bit adder to sum the exponents. Additionally, two 8-bit subtractors are used for adjusting the result exponent.

## VI. Results

Two different implementation of the division hardware with different number of stages have been evaluated. The implementations have different results in terms of clock frequency and resource usage. These results are presented in the first two rows of Table I.

The first implementation computes the division in one stage and takes 15.2 ns. Thus, the clock frequency becomes 65 MHz. In the second implementation the inverter is divided into 4 stages whereas the floating point multiplier is divided into 2.

$Stage4$, that is shown in Figure 4, has the longest latency with 4.3 ns. The multiplier, in this stage, utilizes 2 DSP slices and causes a latency around 4 ns. The next stage consists of another multiplier utilizing 2 DSP slices for the multiplication of the mantissas and has a similar latency. Since it is not possible to divide these multiplications, the clock period is chosen as the latency of the $Stage4$ which is 4.3 ns and the stages are not divided further. As a result, the clock frequency is 232 MHz and the total latency is $4.3 \times 6 = 25.8$ ns.

TABLE I
COMPARISON OF DIFFERENT IMPLEMENTATIONS OF SINGLE-PRECISION FLOATING-POINT DIVISION

| Published | LUT | FF | DSP | BRAM | Freq(MHz) | Period(ns) | Cycles | Latency(ns) | Platform |
|---|---|---|---|---|---|---|---|---|---|
| Proposed Imp 1 | 79 | 32 | 7 | 1 | 65 | 15.2 | 1 | 15.2 | Xilinx Virtex Ultrascale |
| Proposed Imp 2 | 183 | 257 | 7 | 1 | 232 | 4.3 | 6 | 25.8 | Xilinx Virtex Ultrascale |
| Singh-2016[18] | 10019 | 408 | - | - | N/A | N/A | N/A | N/A | Xilinx Spartan 6 SP605 |
| Prashanth-2012[22] | 762 | 201 | - | - | 94 | 11 | 50 | 550 | Altera Stratix-III |
| Pasca-2012[23] | 426 | 408 | 4 | 2 M20K | 400 | 2.5 | 15 | 37.5 | Altera Stratix-V |
| Detrey-2007[24] | 1210 | 1308 | - | - | 233 | 4.2 | 16 | 67.2 | Altera Stratix-V |
| Leeser-2005[20] | 335 slices x (2LUT, 2FF) | | 8 | 7 | 110 | 9 | 14 | 126 | Xilinx Virtex-II |

The implementations use a pipelined approach and produce one result at each cycle. This means max throughput (number of floating point divisions per second) of each implementation is equal to its clock frequency.

Resource utilization of the whole implementation is given in Table I whereas Table II shows the utilization of each individual component for both implementations. The first implementation uses 32 registers to hold the result of the division. In the second implementation, the resource usage increases due to the stages. Some of the stage registers are moved out of the inverter block during optimization by Xilinx tools and they are included in the leaf cell registers in Table II. Most of these registers are used for delaying the dividend that needs to wait for the inverter result.

TABLE II
UTILIZATION RESULTS OF PROPOSED IMPLEMENTATIONS ON A XILINX VIRTEX ULTRASCALE FPGA

| Implementation | Module | LUT | FF | DSP | BRAM |
|---|---|---|---|---|---|
| Imp 1 (1 stage) | Inverter | 40 | - | 5 | 1 |
| | FPMult | 36 | - | 2 | - |
| | Leaf cells | 3 | 32 | - | - |
| Imp 2 (6 stages) | Inverter | 107 | 101 | 5 | 1 |
| | FPMult | 45 | 10 | 2 | - |
| | Leaf cells | 31 | 146 | - | - |
| | Available | 537600 | 1075200 | 768 | 1728 |

## VII. DISCUSSION

Table I compares the utilization and timing results of our implementations with five prior works. The implementations can be compared based on two aspects; resource usage and timing. However, comparing implementations running on different FPGAs is very difficult, if not impossible. Hence we will mention only comparable results and try to give advantages and disadvantages of our implementations in comparison to the others.

The main advantage of our implementations is the low resource requirement and consequently small area usage. The implemented inverter design requires 3 look-up tables with $64$ words in each, which sums up to $448$ bytes and utilizes a single block RAM on the target FPGA. On the other hand, the method adopted by Leeser and Wang [20], which is another table look-up method, requires a look-up table with the size of $12.5$ KB. Similarly, Pasca[23] uses look-up tables to store approximation results and targets an Altera FPGA. The implementation requires two M20K memory blocks, each of which consists of $12480$ configurable bits. Another important resource is the DSP slices. Leeser and Wang [20] utilizes $8$ DSP slices within a Virtex-II, which has $18 \times 18$ multipliers. The DSP utilization of the proposed implementations would remain as 7 with the same sized DSPs. The required number of slices is $4$ for Pasca [23]. However, these DSP slices (on Altera FPGA) support $25 \times 25$ multiplication size, whereas the DSP slices on Virtex Ultrascale support $27 \times 18$. With larger sized DSP slices, the DSP requirement of the proposed implementations can be reduced to $5$. Consequently, due to solving the bottlenecks in $Stage4$ and $Stage5$, the clock frequency can be increased significantly.

Within the implementations, which do not use look-up tables, the implementation of Prashanth [22] utilizes the least number of resources. However, it suffers from the long latency.

Comparing the timing results would hardly lead to any meaningful conclusion due to having different platforms for different implementations. Hence we present the timing results in Table I without going into any comparison to give a hint of the performance of our implementations on a high-end FPGA.

## VIII. CONCLUSION

This paper presents a novel method for single-precision floating-point division based on an inverter, which utilizes a Harmonized Parabolic Synthesis method, and a floating-point multiplier. The method is used in two different implementations. These implementations perform the divisions on floating-point numbers represented in IEEE-754 binary32 format. In the current state, exceptions and denormalized numbers are not taken into account. However, the error performance of the inverter is validated by exhaustive testing. The maximum error of the division is found to be under 1 $ULP$ (unit in the last place) after testing with random inputs and some corner cases.

The two implementations differ in the number of pipeline stages used and consequently provide different results in terms of latency, throughput and resource usage. Low resource requirement and high throughput make this plotting-point division unit suitable for high performance embedded systems. It can built into an existing CPU floating-point unit or be used as a stand alone accelerator.

Future work includes handling the exceptions and denormalized numbers, completing the cubic interpolation accelerator, which adapts the 6 stage division design, and integrating it to a RISC-V core. Additionally, we plan to implement some other basic operations such as square root and inverse square root with Parabolic Synthesis method, make them IEEE-754 compliant and use them as basic building blocks while building manycore architectures.

## REFERENCES


[1] Z. Ul-Abdin, A. Ahlander, and B. Svensson, "Energy-efficient synthetic-aperture radar processing on a manycore architecture," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 330–338.

[2] Z.Ul-Abdin, A. Ahlander, and B. Svensson, "Real-time radar signal processing on massively parallel processor arrays," in *2013 Asilomar Conference on Signals, Systems and Computers*, Nov 2013, pp. 1810–1814.

[3] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*. New York, NY, USA: IEEE, Aug. 2008.

[4] E. Hertz, B. Svensson, and P. Nilsson, "Combining the parabolic synthesis methodology with second-degree interpolation," *Microprocessors and Microsystems*, vol. 42, pp. 142–155, 2016.

[5] E. Hertz, "Methodologies for approximation of unary functions and their implementation in hardware," Ph.D. dissertation, Halmstad University Press, 2016.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[7] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[8] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," 2016.

[9] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 365–376.

[10] S. F. Obermann and M. J. Flynn, "Division algorithms and implementations," *IEEE Transactions on computers*, vol. 46, no. 8, pp. 833–854, 1997.

[11] H. Z. Mao. (2017) Floating point chisel modules. [Online]. Available: https://github.com/zhemao/chisel-float

[12] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.

[13] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 1998, pp. 191–200.

[14] I. Newton, "Methodus fluxionum et serierum infinitarum," 1664-1671.

[15] J. Raphson, "Analysis aequationum universalis," 1690.

[16] J.-M. Muller, *Elementary functions*. Springer, 2006.

[17] P. Behrooz, "Computer arithmetic: Algorithms and hardware designs," *Oxford University Press*, vol. 19, pp. 512 583–512 585, 2000.

[18] N. Singh and T. N. Sasamal, "Design and synthesis of single precision floating point division based on newton-raphson algorithm on fpga," in *MATEC Web of Conferences*, vol. 57. EDP Sciences, 2016.

[19] J. Swami, S. B. Krisna, and T. Maharaja, "Vedic mathematics or sixteen simple mathematical formulae from the veda, delhi (1965)," *Motilal Banarsidas, Varanasi, India*, 1986.

[20] M. Leeser and X. Wang, "Variable precision floating point division and square root," DTIC Document, Tech. Rep., 2005.

[21] P. Hung, H. Fahmy, O. Mencer, and M. J. Flynn, "Fast division algorithm with a small lookup table," in *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*, vol. 2. IEEE, 1999, pp. 1465–1468.

[22] B. Prashanth, P. A. Kumar, and G. Sreenivasulu, "Design & implementation of floating point alu on a fpga processor," in *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*. IEEE, 2012, pp. 772–776.

[23] B. Pasca, "Correctly rounded floating-point division for dsp-enabled fpgas," in *22nd International Conference on Field Programmable Logic and Applications (FPL), 2012*. IEEE, 2012, pp. 249–254.

[24] J. Detrey and F. De Dinechin, "A tool for unbiased comparison between logarithmic and floating-point arithmetic," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 49, no. 1, pp. 161–175, 2007.

[25] M. D. Ercegovac and T. Lang, *Division and square root: digit-recurrence algorithms and implementations*. Kluwer Academic Publishers, 1994.

[26] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

# Appendix D

## Paper IV
## Designing Domain Specific Heterogeneous Manycore Architectures Based on Building Blocks

Süleyman Savas, Zain Ul-Abdin, and Tomas Nordström

submitted to *27th International Conference on Field-Programmable Logic and Applications (FPL 2017)*, Ghent, Belgium, September 4-8, 2017.

# Designing Domain Specific Heterogeneous Manycore Architectures Based on Building Blocks

Süleyman Savas, Zain Ul-Abdin and Tomas Nordström

School of Information Technology, Halmstad University, Halmstad, Sweden
{suleyman.savas, zain-ul-abdin, tomas.nordstrom}@hh.se

***Abstract*—Performance and power requirements has pushed computer architectures from single core to manycores. These requirements now continue pushing the manycores with identical cores (homogeneous) to manycores with specialized cores (heterogeneous). However designing heterogeneous manycores is a challenging task due to the complexity of the architectures.**

**We propose an approach for designing domain specific heterogeneous manycore architectures based on building blocks. These blocks are defined as the common computations of the applications within a domain. The objective is to generate heterogeneous architectures by integrating many of these blocks to many simple cores and connect the cores with a network-on-chip. The proposed approach aims to ease the design of heterogeneous manycore architectures and facilitate usage of dark silicon concept.**

**As a case study, we develop an accelerator based on several building blocks, integrate it to a RISC core and synthesize on a Xilinx Ultrascale FPGA. The results show that executing a hot-spot of an application on an accelerator based on building blocks increases the performance by 15x, with room for further improvement. The area usage increases as well, however there are potential optimizations to reduce the area usage.**

## I. Introduction

Processor architectures shifted from single cores to multi/manycores due to performance, power consumption and thermal limits. The first multicores were produced by duplicating the processing cores which resulted in having identical cores on the same die, in other words homogeneous architectures. However, the tasks performed on these cores are not necessarily identical and each task might require different hardware resources to be executed efficiently. Therefore, we believe that in order to take the computer architectures one step further in performance and energy efficiency, heterogeneity, which combines different types of cores on the same die, needs to be utilized. Additionally, these architectures should adapt the dark silicon[1] concept due to power and heat limitations. However, designing heterogeneous architectures is a very challenging task due to the complexity of these architectures.

In this paper, we propose an approach for designing high performance, energy efficient, domain specific heterogeneous manycore architectures based on basic building blocks. We define these blocks as the compute intensive parts (hot-spots) of the applications within a domain such as signal processing. The hot-spots might consist of one or many blocks. These blocks are custom hardware and integrated to simple cores which execute the control flow of the applications. The cores delegate the compute intensive (hot-spot) tasks to these blocks, using them as accelerators.

The proposed architectures consist of many simple cores with many blocks targeting specific application domains. A number of cores and blocks might be sufficient for efficient execution of an application within the target domain. Hence, power budget of the chip can be used to power these utilized cores and blocks whereas the idle ones stay dark. This allows the architecture to execute different applications efficiently with a limited power budget. The cost is extra area usage, however, area comes for free as the transistor size decreases.

The design approach can be summarized as identifying the building blocks within an application domain, creating a hardware library of these blocks and generating heterogeneous manycore architectures by integrating the blocks to simple cores. In this design, the cores act as controllers and use the blocks and accelerators for computations.

As a case study, we profile an Autofocus application which is a key component of synthetic aperture radar systems [2]. The chosen method performs cubic interpolations based on Neville's algorithm [3] for calculating autofocus criterion. We identify the hot-spot of the application and implement it as an accelerator by combining several building blocks. The accelerator is integrated to a core (rocket core) which is based on RISC-V open-source instruction set architecture (ISA) [4]. The core and the accelerator are generated via rocket chip system on chip generator [5]. The generated design is synthesized via Xilinx tools while targeting a Xilinx Ultrascale FPGA. The hardware description language is chosen as Chisel [6] which is the language used for the rocket chip environment.

## II. Background

In this section we present the application and the accelerator development environment used in the case study.

### A. Synthetic Aperture Radar & Autofocus

A Synthetic Aperture Radar (SAR), is a coherent mostly airborne side-looking radar system that utilizes the flight path of the platform to simulate an extremely large antenna or aperture electronically. This radar system generates high-resolution remote sensing imagery. In reality the flight path is not perfectly linear. Therefore, GPS data is used for compensation. However, if this data is insufficient autofocus criterion

calculation can be used. There are different autofocus methods. The one used in this study is maximization of correlation of image data [2].

### B. RISC-V Environment

RISC-V is an open instruction set architecture based on reduced instruction set computing (RISC) principles. It is originally developed at University of California (UC), Berkeley.

Rocket core is an in-order scalar processors based on RISC-V ISA. It features 5-stage pipeline and has an integer ALU, an optional FPU and L1 data and instruction caches. This core supports up to 4 accelerators via an interface called rocket custom co-processor (RoCC) which can be seen in Figure 1. Custom instructions can be added to RISC-V ISA and they can be forwarded to the accelerators through the RoCC interface (represented as cmd in the figure). Depending on a bit field in the custom instruction, the core might halt until it receives a response from the accelerator. Rocket core can be generated via rocket chip generator which is a Scala program that invokes Chisel compiler in order to emit RTL describing a complete system-on-chip (SoC). This generator allows the developers to configure the generated core by changing some parameters including cache sizes, FPU usage, number of cores, accelerator usage, etc. Rocket chip environment includes cycle-accurate Verilator [7] for simulations.

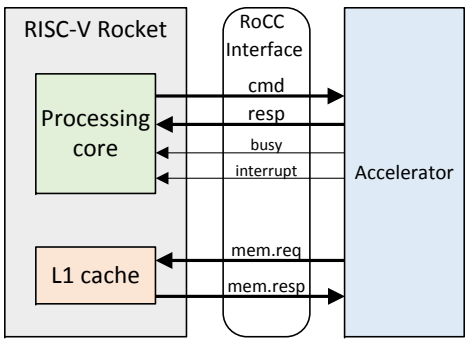


Fig. 1. Simplified view of RoCC interface

Chisel is an open-source hardware description language developed at UC Berkeley. It is embedded in Scala language. Rocket chip (RISC-V) tools can take Chisel code and generate synthesizable verilog code.

## III. Methodology

The methodology for designing domain specific heterogeneous manycore architectures consist of 6 steps as shown in Figure 2. The first 5 steps are covered in this paper and the last step will be covered once we have a larger library of blocks and accelerators and a larger application. Step 4 (accelerator development) can be skipped if only a single block is required.

We use traditional methods such as counting number of instruction fetches and function calls and measuring execution

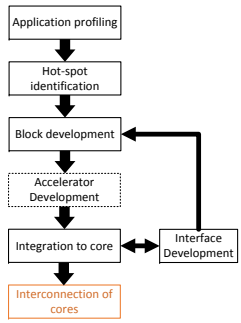


Fig. 2. Design flow of a single core in heterogeneous manycores

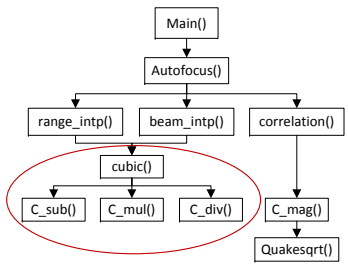


Fig. 3. Function-tree of Autofocus application

time of functions for dynamically analyzing the Autofocus application. We profile the application at function level due to having quite small functions. However, one can choose to profile at basic block level (during compilation) which requires more effort due to interaction with the compiler. There are many other works which profile the applications at basic block level to find out the most frequently executed instruction sequences [8], [9]. However we keep the profiling at a higher level to keep it simple.

In order to gain profiling information, Valgrind [10] and gnu profiler (gprof) tools are used. The application needs to be compiled with specific flags (-pg for gprof, -g for valgrind) to gain the necessary information. Valgrind provides number of total instruction fetches per-function which can be used for analysis of potential speed-up however this is not enough to determine a hot-spot. gprof provides number of function calls, total execution time of the application and the amount of time spent in each function. Since the Autofocus application has a static schedule, which means the input data does not change the order of functions or instructions, the time spent in each function does not change. Thus we use this information to identify the hot-spots as the most frequently executed functions. For instance, according to the analysis information, the highlighted functions in Figure 3 consume 96% of the total execution time. Detailed analysis of execution times spent in each function can be seen in Figure 4. $C_mul, C_sub$ and $C_div$ functions are called within the $cubic$ function as can be seen in Figure 3.

Additionally, we use the number of fetched instruction per-

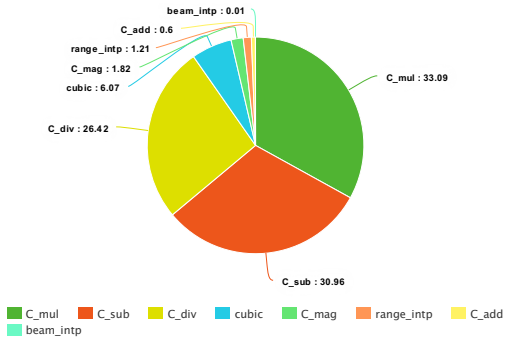


Fig. 4. Percentage of execution times spent in each function of the Autofocus application

Fig. 5. Overview of custom instruction to call the accelerator

function together with number of function calls to estimate the potential speed-up. Once the hot-spots are identified, one can proceed to the next step which is developing the necessary blocks to execute these compute intensive tasks.

We design the operations in the hot-spot as building blocks and connect these blocks to form an accelerator to execute the entire hot-spot. Integration to the core is done with the RoCC interface. This interface supports transfer of two 64-bit registers at a cycle. If the accelerator requires more input data to fire, another interface might be required to store the input data until the last inputs arrive. The accelerator returns the result through the same interface.

The core executes a custom instruction to forward the necessary data to the interface and fire the accelerator. The bit fields of the instruction can be seen in Figure 5. $xs1$ and $xs2$ indicates the usage of source registers whereas $xd$ indicates if the accelerator will write a result in the destination register. If this bit is set, the core will wait until the accelerator finishes the computation. The instruction call needs to be inlined as assembly code to the application. A sample call can be performed as follows:

```
register uint64_t rd_  asm ("x" # 10);
register uint64_t rs1_ asm ("x" # 11)
= (uint64_t) rs1;
register uint64_t rs2_ asm ("x" # 12)
= (uint64_t) rs2;
asm volatile (
  ".word " 0b0001011        |
  (10    << ( 7) )          |
  (0x7   << ( 7+5 ))        |
  (11    <<  ( 7+5+3 ))     |
  (12    <<  ( 7+5+3+5 ))   |
  (funct << ( 7+5+3+5+5 )) |
  "\n\t"
  : "=r" (rd_)
  : [_rs1] "r" (rs1_),
  [_rs2] "r" (rs2_));
rd = rd_;
```

The above code uses the registers 10, 11 and 12 with the custom instruction and sets the $xd, xs1$ and $xs2$ fields. The values in $rs1$ and $rs2$ are forwarded to the accelerator and the result will be written into $rd$.

Each core can be connected to a number of different accelerators (max 4 with RoCC interface, however this can be increased with addition of another interface between the RoCC and the accelerators) depending on the requirement.

The last step of the design approach is the interconnection of the cores which consist of different blocks and accelerators. Figure 6 gives an overview of how the final architecture might look like. In the figure there are 16 cores, each with different number and type of accelerators. The numbers are arbitrary in the example case, however they should be determined based on the requirements of the target application domain.

The final architecture includes many cores with different blocks and accelerators to support efficient execution of all applications within a domain. In most cases, an application does not require every block which results in idle cores. These idle cores can be shut down to save power.

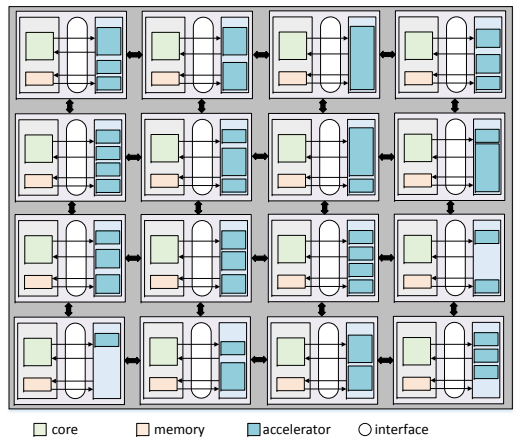


Fig. 6. Overview of a potential domain specific heterogeneous architecture

The methodology steps covered in this paper are performed mostly manually, however we plan to automate every step either by using the existing tools or developing new ones.

## IV. Related Works

Accelerator development has been extensively studied and several different accelerators have been developed [11], [12], [13], [14], [15], [16], [17]. However, cubic interpolation has not been accelerated and more importantly, these works do not target developing manycore architectures but only single accelerators.

Sano et al. [18], Tanabe et al. [19] and Schurz et al. [20] develop FPGA based manycore architectures. However, they do not propose any generic method for developing these architectures.

Clark et al. [21] automates custom instruction generation and Koeplinger et al. [22] generates accelerators for FPGAs automatically. These works inspire the automation of first two steps of our methodology, however, do not cover the idea of designing domain specific heterogeneous manycore architectures.

We present the first draft of a generic methodology to design domain specific heterogeneous manycore architecture based on building blocks and accelerators.

## V. Implementation

The blocks and the accelerator are developed in Chisel language. The accelerator is integrated to a rocket core through RoCC interface. Verilog code for the accelerator is generated and synthesis is performed by Xilinx tools. The target platform is Xilinx VCU108 evaluation kit including Virtex UltraScale XCVU095-2FFVA2104E FPGA. Additionally, a rocket core with and without the accelerator are synthesized individually.

The Autofocus implementation takes $6x6$ pixel block as input and applies complex-number calculations on these blocks. The hot-spot of the application is the computation of cubic interpolation. This computation is a combination of complex multiplication, complex subtraction and complex division as seen in Figure 3. Each of these operations is implemented as a building block. Figure 7 presents the structure of complex division block which consists of floating-point operations. This block is represented as $CDiv$ in Figure 8 which shows the structure of the accelerator that executes the cubic interpolation.

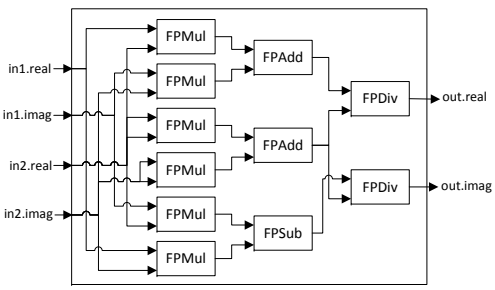


Fig. 7. Building block that performs division on complex numbers

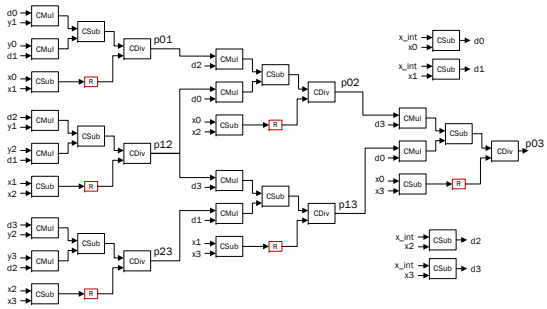


Fig. 8. Cubic interpolation accelerator based on building blocks (Small red boxes are delay registers

Cubic interpolation operation requires 4 pixels as input. Each pixel consists of 2 complex numbers namely position and value. Each complex number has 2 floating-point values, one for the real part and one for the imaginary part. Total number of inputs to the accelerator is 16 floating-point numbers represented in IEEE-754 single precision (binary32) format [23]. These inputs are named as $x$ and $y$ in Figure 8. The RoCC interface supports transfer of two 64-bit values from core to the accelerator with each custom instruction. Therefore, we combine real and imaginary values of each complex number and send 4 floating-point numbers to the accelerator with each instruction. It takes 4 instructions to send all input data.

We have implemented an interface between RoCC interface and the accelerator to handle the synchronization of the input data. This interface stores the input data until it receives the instruction with the last 4 data. Then all 16 input data is fed to the accelerator and the accelerator is fired. It takes 52 cycles for the accelerator to produce the results.

The accelerator is designed with a pipelined structure and supports 1 set of results (one complex or two floating-point numbers) per-cycle. However due to the limitations in the data transfer, it can be fired at every 4th cycle (ignoring the cache misses).

Table I presents the number of (blocks) complex operations executed by the accelerator whereas Table II shows the number of floating-point operations performed by each block and the accelerator. In total, 140 floating-point operations are performed for each set of inputs. While computing floating-point multiplication and divisions, on-chip DSP blocks are utilized.

TABLE I
Number of complex operations in the accelerator

| | **Accelerator** |
|---|---|
| **CMul** | 12 |
| **CDiv** | 6 |
| **CSub** | 16 |

TABLE II
NUMBER OF FLOATING-POINT OPERATIONS IN BLOCKS AND THE ACCELERATOR

| | Accelerator | CMul | CSub | CDiv |
|---|---|---|---|---|
| **FPMul** | 84 | 4 | - | 6 |
| **FPDiv** | 12 | - | - | 2 |
| **FPSub** | 50 | 1 | 2 | 1 |
| **FPAdd** | 24 | 1 | - | 2 |

## VI. RESULTS

This section presents three different results namely performance, resource usage and max clock frequency.

Performance results, which can be seen in Table III, are gained through running the cubic interpolation on the cycle accurate emulator provided with the rocket chip generator. Hardware counter is used to get the cycle counts. The interpolation is executed first on the rocket core without the accelerator and then on the rocket core with the accelerator.

The accelerator produces the result in 52 cycles. However, due to the pipelined structure, one result can be produced at every cycle if enough inputs are provided. Since RoCC interface can provide max 4 floating-point inputs per-cycle, the throughput of the accelerator decreases to 1 result per-4-cycle. Furthermore, the rocket core halts (does not send any input data to the accelerator) until the accelerator produces the result. Hence, the throughput of the accelerator decreases to 1 result per-52-cycles. The cycle count for the integrated design in Table III includes 8 additional mov instructions for copying the data to the registers to be sent to the accelerator. While measuring the these results the caches are pre-filled, hence there are no cache misses. Despite the drawbacks in the integration, the cubic interpolation is computed 15x faster when the accelerator is utilized. Theoretically, the accelerator can produce 1 result per-cycle which makes it 925x faster than the processing core in the long run. However this requires reading at least 8 registers in the same cycle which is not possible with the RISC-V ISA. It takes 4 cycles to read 8 source registers, hence if the core does not halt until the accelerator returns the result, the throughput can be increased to 1 result per-4-cycles.

TABLE III
EXECUTION TIMES IN TERMS OF CLOCK CYCLES

| | Cubic interpolation |
|---|---|
| **Rocket Core** | 925 |
| **Rocket Core + Accelerator** | 60 |

Resource usage results, given in Table IV, are provided by Xilinx synthesis tools. First the rocket core and the accelerator are synthesized individually. Then the integrated design (core + accelerator) is synthesized. The results for the integrated design in Table IV include the resource usage of the interfaces.

The accelerator consists of 6 coarse grained blocks which compute the $p$ values as given in Figure 8. The whole computation can be performed in one of these blocks to reduce the resource utilization. This will cause a decrease in throughput, however, the throughput is already decreased by the interface and there would be no further decrease.

The increase in the size of the integrated design causes the critical path of the processor to increase insignificantly. The integration causes the max clock frequency of the processor to decrease from 60MHz to 57Mhz whereas the accelerator achieves 126MHz when it is synthesized individually. These results can be seen in Table V.

TABLE IV
RESOURCE USAGE RESULTS. LAST ROW SHOWS THE RESOURCE USAGE OF THE ACCELERATOR OVER THE INTEGRATED DESIGN

| | LUT | LUTRAM | FF | DSP | BRAM |
|---|---|---|---|---|---|
| **Rocket Core** | 40990 | 710 | 16512 | 24 | 12 |
| **Accelerator** | 24707 | 464 | 14471 | 252 | 3 |
| **Integrated** | 69429 | 1832 | 36722 | 276 | 16 |
| **Accelerator %** | 34% | 25% | 39% | 91% | 18% |

TABLE V
TIMING RESULTS

| Design | Max clock frequency |
|---|---|
| **Rocket Core** | 60 MHz |
| **Accelerator** | 126 MHz |
| **Integrated** | 57 MHz |

## VII. CONCLUSIONS

We proposed an approach for designing domain specific heterogeneous manycore architectures based on building blocks and accelerators. Several building blocks and an accelerator based on these blocks are developed and integrated to a core as a case study.

The proposed approach aims to ease the development of heterogeneous manycore architectures by providing a library of building blocks for different application domains. The target architecture is a domain specific manycore which adapts dark silicon technology to deal with the thermal and power limitations. An architecture based on blocks facilitates usage of this technology by enabling or disabling certain blocks.

The case study shows that utilizing specialized custom hardware increases the performance of the processing core significantly with a cost in the area usage. However, there is room for optimizing the size of the accelerator without harming the performance. Additionally, area cost decreases as the transistor size decreases.

There are a number of tasks to be performed in order to evaluate the whole design approach. More applications need to be analyzed to define the building blocks of a domain. These blocks must be integrated to several cores to build a manycore architecture targeting the specific domain. The cores should be minimalistic in case of resource usage.

The manycore architectures are slowly going into the direction of specialized cores and we believe that the proposed approach will facilitate the design of new architectures.

## VIII. Acknowledgement

We would like to thank Schuyler Eldridge from IBM and Howard Mao from UC Berkeley for their help with the RoCC interface.

## References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE M MICRO*, vol. 32, no. 3, pp. 122–134, 2012.

[2] Z. Ul-Abdin, A. Ahlander, and B. Svensson, "Energy-efficient synthetic-aperture radar processing on a manycore architecture," in *2013 42nd International Conference on Parallel Processing*, Oct 2013, pp. 330–338.

[3] E. H. Neville, *Iterative interpolation*. St. Joseph's IS Press, 1934.

[4] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[5] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," 2016.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 1216–1225.

[7] W. Snyder, D. Galbi, and P. Wasson. (2017) Verilator, verilog hdl simulator. [Online]. Available: https://www.veripool.org/wiki/verilator

[8] K. Atasu, R. G. Dimond, O. Mencer, W. Luk, C. Özturan, and G. Dündar, "Optimizing instruction-set extensible processors under data bandwidth constraints," in *Proceedings of the conference on Design, automation and test in Europe*. EDA Consortium, 2007, pp. 588–593.

[9] M. Haaß, L. Bauer, and J. Henkel, "Automatic custom instruction identification in memory streaming algorithms," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, 2014, p. 6.

[10] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[11] M. Peemen, A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*. IEEE, 2013, pp. 13–19.

[12] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 225–236.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012, pp. 449–460.

[14] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3. ACM, 2010, pp. 37–47.

[15] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 36–47.

[16] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 24–35.

[17] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1. ACM, 2010, pp. 205–218.

[18] K. Sano, Y. Hatsuda, and S. Yamamoto, "Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*. IEEE, 2011, pp. 234–241.

[19] S. Tanabe, T. Nagashima, and Y. Yamaguchi, "A study of an fpga based flexible simd processor," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 4, pp. 86–89, 2011.

[20] F. Schurz and D. Fey, "A programmable parallel processor architecture in fpgas for image processing sensors," in *Proceedings of the integrated design and process technology (IDPT07) Conference*. Citeseer, 2007, pp. 30–35.

[21] N. T. Clark, H. Zhong, and S. A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1258–1270, 2005.

[22] D. Koeplinger, C. Delimitrou, R. Prabhakar, C. Kozyrakis, Y. Zhang, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 115–127.

[23] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic*, Aug. 2008.