FGPU:A Flexible Soft GPU Architecture for General Purpose Computing on FPGAs

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

vorgelegt von

Muhammed AL KADI

geboren in Edlib, Syrien

Bochum, 2017

Hauptbetreuer:	Prof. Dr.-Ing. Michael Hübner
Co-Betreuer:	Prof. João Cardoso

Abstract

Realizing embedded Graphics Processing Units (GPUs) on top of Field-Programmable Gate Arrays (FPGAs) would lead a new family of processors: it offers the efficiency and ease of programming of GPU architectures with the flexibility and reconfigurability of FPGA platforms. This dissertation describes the hardware and the tool flow of the FPGA-GPU (FGPU): a configurable, scalable and portable GPU designed specially for FPGAs. It is an open-source 32bit processor programmable in OpenCL. FGPU is designed to perform General Purpose Computing on GPUs (GPGPU) and it does not replicate any other architecture.

On a middle-sized Zynq System-On-Chip (SoC), up to 64 Processing Elements (PEs) can be realized at 250MHz. Over a benchmark of 20 applications, the biggest FGPU outperforms a hard ARM Cortex-A9 processor supported with a 128bit NEON vector engine running at 667MHz by a factor of 4x, on average. FGPU supports single precision floating-point arithmetic in hardware or as emulated instructions in software. In the case of hard floating-point support, it has on average 2.9x better throughput per area and 11x less energy consumption than any MicroBlaze-based homogeneous Multi-Processor System-On-Chip (MPSoC). In the case of emulated floating-point arithmetic, the average performance degradation does not exceed 4.5x. In addition, a dedicated compiler based on the LLVM framework has been designed. FGPU can be programmed and controlled through a PYNQ-based Application Programming Interface (API) using Python scripts.

We throw the light on the differences between the FGPU and the High-Level Synthesis (HLS) tool flows. Our experiments reveled that both approaches deliver similar performance. HLS has better area and energy efficiency if the task parameters are fixed at synthesis time. Otherwise, FGPU would outperforms HLS with more compact and simpler software implementations. At the end of this dissertation, our first attempts to realize a partially reconfigurable GPU, which changes its hardware at runtime, are described.

Zusammenfassung

Die Realisierung eingebetteter Grafikprozessoren (GPUs) auf Field-Programmable Gate Arrays (FPGAs) führt zu einer neuen Prozessorfamilie, die die Effizienz und eine einfache Programmierung einer GPU ermöglicht, aber die Flexibilität und die Rekonfigurierbarkeit einer FPGA-Plattform nutzt. Diese Dissertation beschreibt den Entwurf der Hardware sowie das Tool Flows der FPGA-GPU (FGPU): eine konfigurierbare, skalierbare und portierbare GPU, die für FPGAs speziell entwickelt wurde. Sie ist ein open-source 32bit Prozessor, der mit OpenCL programmierbar ist. FGPU ist für universelles Rechnen auf GPUs (GPGPU) gedacht und sie repliziert keine andere Architektur.

Auf einem mittelgroßen Zynq System-On-Chip (SoC) lassen sich bis zu 64 Verarbeitungselemente (PEs) in einer einzigen FGPU mit 250MHz realisieren. In einem Benchmark von 20 Applikationen übertrifft die größte FGPU, die wir implementieren konnten, einen hart verdrahteten ARM Cortex-A9 Prozessor unterstützt von einem 128bit NEON Vektor-Coprozessor im Durchschnitt um einen Faktor von 4x. FGPU unterstützt Gleitkomma-Arithmetik in einfacher Genauigkeit in Hardware oder als emulierte Instruktionen in Software. Im Falle einer harten Gleitkomma-Unterstützung bietet sie im Durchschnitt 2,9x besseren Durchsatz pro Fläche und 11x weniger Energieverbrauch als jedes MicroBlaze-basierte homogene Multi-Processor System-On-Chip (MPSoC) an. Im Falle einer emulierten Gleitkomma-Arithmetik überschreitet die durchschnittliche Leistungsverschlechterung 4,5x nicht. Darüber hinaus wurde ein dedizierter Compiler auf Basis der LLVM Entwicklungsumgebung entworfen. FGPU kann über eine PYNQ-basierte Schnittstelle mit Python-Skripten programmiert und gesteuert werden.

Wir beleuchten die Unterschiede zwischen den Ansätzen von FGPU- und High-Level Synthesis (HLS). Unsere Experimente zeigten, dass beide Lösungen ähnliche Performanz liefern. HLS hat eine bessere Flächen- und Energieeffizienz sofern die Aufgabenparameter zur Synthesezeit festgelegt sind. Ansonsten würde die FGPU HLS mit kompakteren und einfacheren Software-Implementierungen übertreffen. Am Ende dieser Dissertation sind unsere ersten Versuche beschrieben, um eine teilweise rekonfigurierbare GPU zu realisieren, deren Hardware sich zur Laufzeit anpassen lässt.

To my parents, my lovely Ola, Ahmad and Ayman

To my wonderful sisters, Ruba and Ola

In memoriam of my brothers, Mazen and Abdullah

Acknowledgments

Four years ago, FGPU was an idea in the thoughts of a single person. Going through the long journey of realizing its different components would not be possible without the unlimited support from Prof. Michael Huebner. First, I would like to thank him that he believed in the concept of FGPU and for the trust he put in me.

In order to get the chance to achieve this success, it was not only the time and effort I have spent, but also the help, love and patience from many others. My biggest thanks to my parents: it is all your own creation. I will stay in your debt for the whole of my life. A debt that can not be settled whatever I would do for you. Finishing this thesis, working in academia and raising a family would not be possible without my wife Ola. I miss the right words that describe my appreciation. The wonderful smiles of my both kids, Ahmad and Ayman, have contributed to this success as well. They helped me a lot to hold through the difficult moments.

Lots of things have changed in my life and my lovely country, Syria, during the work on this project. The war machine took many people away, whom I would like to express my appreciation. To those great people: I have done nothing in comparison to your sacrifice for our future.

Many thanks to all my colleagues for the nice moments we shared together, specially to Fynn Schwiegelshohn, Osvaldo Navarro, Philipp Wehner, Benedikt Janssen, Florian Kaestner, Thomás Grimm, Jens Rettkowski, André Werner and Florian Fricke. A Special thank goes to Jones Mori Da Silva for his effort in proofreading this thesis. To those who I forgot: your place in my heart and memories will not change. Finally, I would like to acknowledge the first-class assistance from Horst Gass, Maren Carevic and Linda Trogant. It is not common to work with people like you who never do mistakes.

Table of Contents

List of Figures

List of Tables

Glossary

ALU Arithmetic-Logic Unit

AU Arithmetic Unit

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction-set Processor

BRAM Block RAM

CAD Computer-Aided Design

CD Compute Density

CFG Control Flow Diagram

CPU Central Processing Unit

CRAM Code RAM

CU Compute Unit

CV Compute Vector

DAG Directed Acyclic Graph

DFG Data Flow Graph

DMA Direct Memory Access

DSL Domain Specific Language

DSP Digital Signal Processing

FF Flip-Flop

FIFO First-In First-Out

FGPU FPGA-GPU

FPGA Field-Programmable Gate Array

FSM Finite State Machine

FU Functional Unit

GDDR Graphics Double Data Rate

GFLOPS Giga Floating-Point Operations Per Second

GPGPU General Purpose Computing on GPUs

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High-Level Synthesis

HW Hardware

ISA Instruction Set Architecture

IC Integrated Circuit

ICAP Internal Reconfiguration Access Port

IP Intellectual Property

IR Intermediate Representation

MFLOPS Mega Floating-Point Operations Per Second

MPSoC Multi-Processor System-On-Chip

NoC Network-On-Chip

LLVM Low Level Virtual Machine

LRAM Link RAM

LUT Look-Up Table

NRE Non-Recurring Engineering

OS Operating System

PC Program Counter

PCAP Processor Configuration Access Port

PE Processing Element

PL Programmable Logic

PS Processing System

QoR Quality Of Results

RISC Reduced Instruction Set Computer

ROM Read-Only Memory

RTL Register Transfer Level

RTM Runtime Memory

SCGRA Soft Coarse-Grained Reconfigurable Array

SIMD Single-Instruction Multiple-Data

SIMT Single-Instruction Multiple-Threads

SM Streaming Multiprocessor

SMP Symmetric Multi-Processor

SoC System-On-Chip

SSA Static Single Assignemnt

SW Software

TFLOPS Tera Floating-Point Operations Per Second

VLIW Very Long Instruction Word

WCB Write Combine Buffer

WF Wavefront

WG Work-Group

Chapter 1

Introduction

In contrast to sequential software-based computing, FPGAs enable deploying application-specific and parallelized hardware implementations. They can potentially deliver better performance per Watt than multi-core processors [25]. In comparison to Application-Specific Integrated Circuits (ASICs), digital designs are not built directly on silicon, but synthesized by programming plenty of existing fine-grained hardware modules. The accompanied performance degradation and the extra power dissipation of the FPGA approach are traded with significant savings in the high Non-Recurring Engineering (NRE) development costs of ASICs [84].

Recent improvements in the manufacturing technology of Integrated Circuits (ICs) have affected FPGA-architectures slightly different than the processor-ones. Central Processing Units (CPUs) and Graphics Processing Units (GPUs) have basically replicated more cores. Nowadays, a single modern GPU can accommodate up to thousands of Arithmetic-Logic Units (ALUs) providing an accumulated compute power that exceeds 10 Tera Floating-Point Operations Per Second (TFLOPS) [70]. Special programming languages like OpenCL [52] have been developed and extended over the time to fulfill the demands of GPGPU. On the FPGA side, not only bigger circuits can be now realized, but also they became more heterogeneous. Next to the traditional FPGA logic, modern chips integrate multiple many-core processors and sophisticated Digital Signal Processing (DSP) blocks [97]. In addition, FPGAs have adopted High-Level Synthesis (HLS) as a part of their tool flow to compete against other solutions [8].

However, hitting the compute bound on modern devices is not possible in embedded applications [86]: while the hardware architecture is capable of processing thousands of floating-point numbers within a single clock cycle (see Figure 1.1), the device

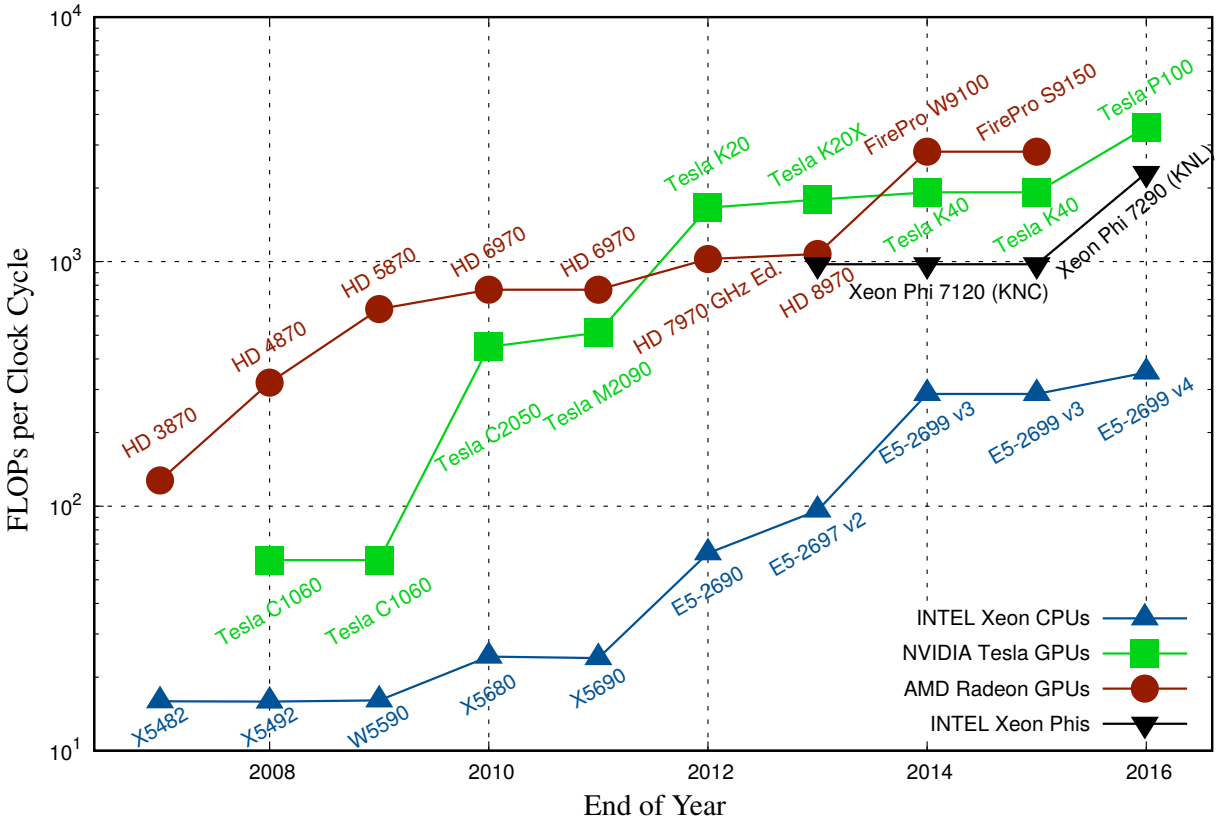


Figure 1.1: Theoretical peak double precision floating-point operations per clock cycle for many multi-core architectures [77]

IO and the external memory connection are not fast enough to hold the corresponding data traffic. In many applications, a portion of the compute capacity of an FPGA or a GPU may be sufficient to keep the memory controllers at maximum utilization. Therefore, sacrificing more chip area or a portion of the maximum operation frequency for more flexibility in the development process, e.g. by programming with HLS, is becoming more acceptable in many scenarios.

To simplify programming an FPGA, many overlay architectures have been developed, e.g. mesh of functional units [41] or vector processors [55]. These "soft" architectures can offer most of the FPGA advantages for many applications at a reduced development effort. In contrast to the hardened ones, which are implemented directly on silicon, soft overlays can be customized according to the application needs without going back to the factory. This dissertation describes a GPU-like processor that has been designed specially for FPGA-synthesis. It aims to investigate the efficiency of GPUs as an overlay architecture to perform general purpose computing on FPGAs.

1.1 Motivation

FPGA-Computer-Aided Design (CAD) has longer cycles than software development for multi-core architectures. Although HLS can significantly improve productivity, it may take hours to place and route a circuit. The excessive use of compiler directives to guide an HLS tool makes it remarkably harder than programming a processor with the same language like C++. Code implementations for processors are more compact and portable. Besides, HLS developers have to write their code in a way that reflects an efficient hardware structure.

To use FPGA-accelerators at the Operating System (OS)-level, application-specific drivers have to be developed and integrated into the kernel space. On the other hand, multi-core processors like GPUs can be controlled through standard APIs. The clear separation between Hardware (HW) and Software (SW) development reduces the programming effort: while GPU manufacturers offer the compilers, OS-drivers and debugging tools; software developers have only to focus on the corresponding API to implement their algorithms.

We propose using GPUs as overlays to offer the FPGA compute power in a flexible tool flow. Compared with many other overlay architectures, we will prove that the proposed tool flow is pareto-optimal regarding performance and flexibility (see Figure 1.2). It can execute the same OpenCL code that runs on real GPUs or standard simulators without any modifications. With soft GPUs, the developer does not need to program the individual processors of MPSoCs and manage the communication among them. Also, GPU architectures can natively scale better than vector processors. Nevertheless, it is possible to implement efficient GPU architectures on modern middle-sized FPGAs. When implemented as soft cores, the high power consumption of GPUs can be mitigated by application-specific optimizations, e.g. selective hardening of floating-point operations or Instruction Set Architecture (ISA)-subsetting.

1.2 Research Contributions

This dissertation describes an open-source GPU architecture that is:

- Designed specially for FPGAs.
- Available completely in RTL with its compiler and API.

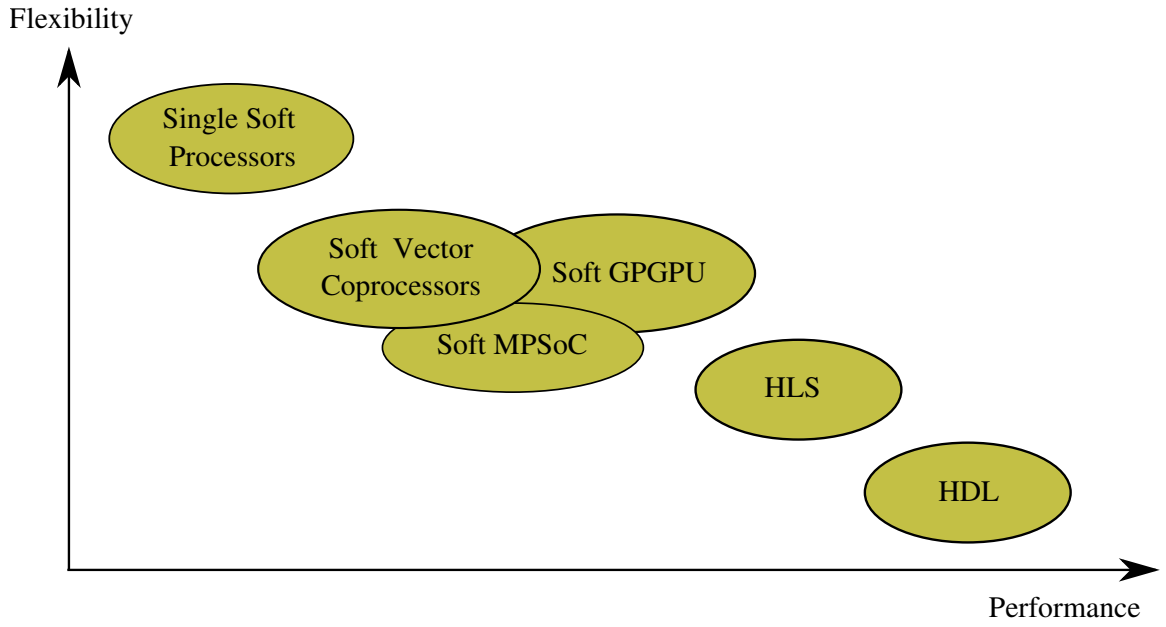


Figure 1.2: Overview of multiple soft processor architectures for computationally intensive applications: Flexibility vs. Performance

- Natively supports floating-point instructions

- Evaluated against many other solutions like HLS, soft and hard vector coprocessors and MPSoCs.

Because of their complexity, all successful GPU architectures have been designed by the industry. Most academic ones focused on replicating industrial GPUs, e.g. to reuse the existing compilers. This work provides researchers with an open-source and manufacturer-independent platform to realize and test new ideas about GPU architectures where no hardware components, or software blocks in the tool chain, are missing. This platform is highly configurable, scalable and portable. Besides, it has a realistic footprint that allows efficient implementations on middle-sized FPGAs.

Regarding throughput per area and energy efficiency, this work proves that soft GPUs are better than single soft processors or soft MPSoCs. Also, it states that soft GPUs can perform as good as HLS with reduced development effort. Finally, this work explores the possibility of realizing partially reconfigurable GPUs to improve their area efficiency, i.e. applying application-specific modifications at the hardware level at runtime.

1.3 List of Publications

1.3.1 Peer-reviewed Conference Proceedings

1. *Floating-Point Arithmetic using GPGPU on FPGAs [47]*: in the proceedings of the 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). This paper describes the organization of scratchpad memories, thread-level branching and floating-point logic.

2. *Integer Computations with Soft GPGPU on FPGAs [46]*: in the proceedings of the 2016 International Conference on Field Programmable Technology (FPT), **Best Paper Candidate**. The paper presents the ALU structure, design of atomic units, sub-integer operations and the OpenCL compiler.

3. *FGPU: An SIMT-Architecture for FPGAs [7]*: in the proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). Here the general hardware details of the proposed FGPU architecture have been described.

4. *Multi-FPGA reconfigurable system for accelerating MATLAB simulations [45]*: in the proceedings of the 2014 24th International Conference on Field Programmable Logic and Applications (FPL). This paper was the basis for the work described in Section 7.4.

5. *Dynamic and partial reconfiguration of Zynq 7000 under Linux [44]*: in the proceedings of the 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig). The paper deals with how to represent a reconfigurable hardware accelerator at the OS-level. Many of the principles described in Chapter 7 originates from this publication.

1.3.2 Peer-reviewed Journals

1. *GPU Overlays for General Purpose Computing on FPGAs:* in ACM Transactions on Reconfigurable Technology and Systems (TRETS), Volume 11 Issue 1, March 2018.

1.4 Supervised Theses

Some parts of this dissertation are based on the following Bachelor and Master theses written at the chair of Embedded Systems of Information Technology (ESIT) at the Ruhr-University of Bochum:

1. Patrick Rudolph. Modellierung einer rekonfigurierbaren Hardware unter Linux. Bachelor thesis, Ruhr-University of Bochum, 2013 [71].
2. Kevin Winge. Entwurf und Implementierung eines Simulationsmodells für eine eingebettete GPU. Bachelor thesis, Ruhr-University of Bochum, 2016 [87].
3. Volker Stegemann. Entwurf und Evaluation einer Multi-FPGA-Architektur zur Beschleunigung von Matlab-Simulationen. Master thesis, Ruhr-University of Bochum, 2014 [81].
4. Martin Tatara. Entwicklung eines Treibers für eine Soft-GPU unter Linux. Master thesis, Ruhr-University of Bochum, 2016. [82].
5. Florain Kriete. Evaluation verschiedener Multicore Architekturen auf FPGAs. Master thesis, Ruhr-University of Bochum, 2017 [54].

1.5 Thesis Organization

The thesis is organized as follows:

- *Chapter 1:* Introduction, motivation and contributions of this work
- *Chapter 2:* **Background** on FPGA and overlay architectures
- *Chapter 3:* **Hardware** design of FGPU
- *Chapter 4:* **Compiler** design of FGPU
- *Chapter 5:* **Tool chain** for FGPU
- *Chapter 6:* **Benchmarking** FGPU against a soft single processor, vector coprocessors and soft homogeneous MPSoCs.
- *Chapter 7:* **Comparing** FGPU-based computing with **HLS** and the benefits of the **partial reconfiguration** technology for the FGPU architecture

- *Appendix A:* **ISA** of FGPU
- *Appendix B:* **Control interface** of FGPU
- *Appendix C:* an FIR-filter in **HLS** or **FGPU**. This code example should help the reader to get a better understanding about the differences between programming in **HLS** or **FGPU**.

Chapter 2

Background and Related Work

Nowadays, FPGA developers have a wide spectrum of tools to realize their applications. This realization can be done at a low level using Hardware Description Languages (HDLs) or at a higher one using HLS. We will describe different approaches to program FPGAs. The common objective of all presented methods is to trade performance or chip area in shorter development time. Indeed, our work has been motivated by the same argument.

At the beginning of this chapter a quick overview about FPGA architectures is provided. This should help the reader to understand how a design should fit to the basic building blocks of an FPGA. Then, the concept of FPGA overlays is presented along with a brief survey of the existing architectures. In addition, we discuss the main pros and cons when using HLS or overlays. The last section describes how GPU architectures can be used to perform not only graphics but also general purpose computing.

2.1 FPGA Logic

Register Transfer Level (RTL) is a logical abstraction of digital hardware. It describes its modularity, functionality as well as the timing behaviour on a clock cycle basis. Hardware Description Languages (HDLs), e.g. VHDL and Verilog, are textual descriptions of digital circuits at the RTL-level.

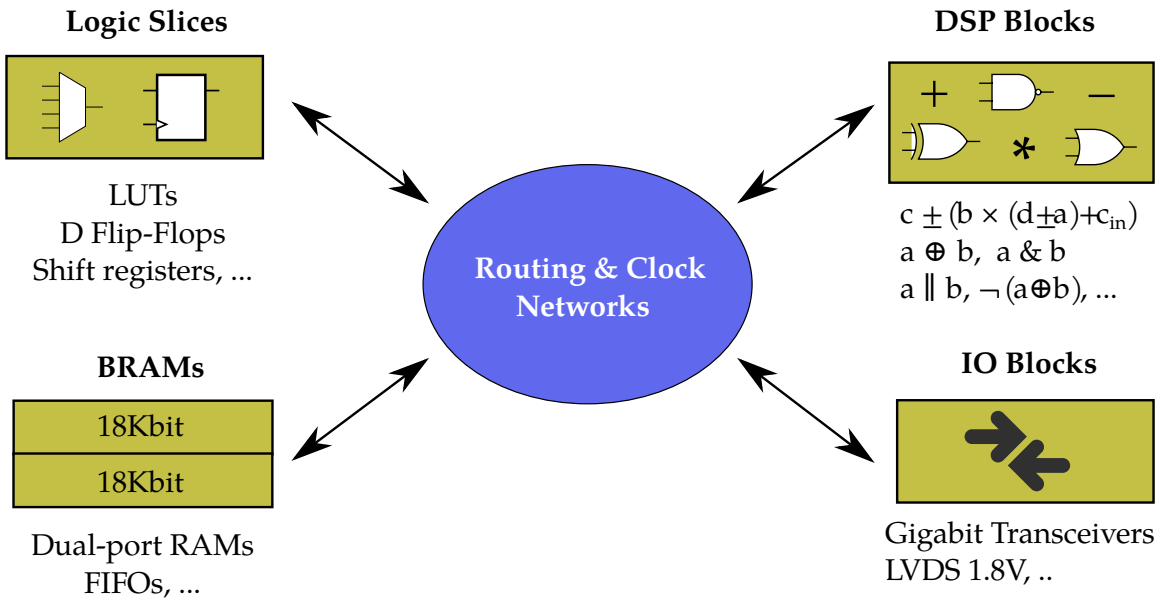


Figure 2.1: Main components of a 7 series FPGA form Xilinx

RTL is the entry level for physical realization of digital designs. When targeting ASICs, the circuit gets synthesized in logic gates. Then, the equivalent transistor realizations are placed on the available chip area. Finally, the required logical interconnections are routed on the chip, and the circuit can be sent to fabrication. On the other hand, an FPGA can be used as a flexible substrate: developers can implement their logic without going through the chip manufacturing process. FPGA-synthesis is done using more coarse-grained components than simple logic gates. The wiring infrastructure is fixed but configurable. In comparison to ASICs, FPGAs enable faster time to market, they have less development cost and are financially preferred in non-mass productions. Figure 2.1 illustrates the basic modules of a 7 series FPGA from Xilinx[1]:

- A *logic slice* comprises many Look-Up Tables (LUTs), D-Flip-Flops (FFs) and some local configurable interconnect. A LUT computes any logical function of up to 6 variables [93].
- A *DSP Block* can implement many arithmetic and logical operations, e.g. $a + b * c$ [94].
- *Block RAMs (BRAMs)* are used often as dual-ported memories or as a First-In First-Out (FIFO) storage with adjustable bitwidth and memory depth, e.g. 1Kx36 bit or 2Kx18 bit [95].

[1] The 7 series family are 28 nm devices introduced in 2010. Xilinx introduced later the UltraScale family on 20 nm technology in 2014.

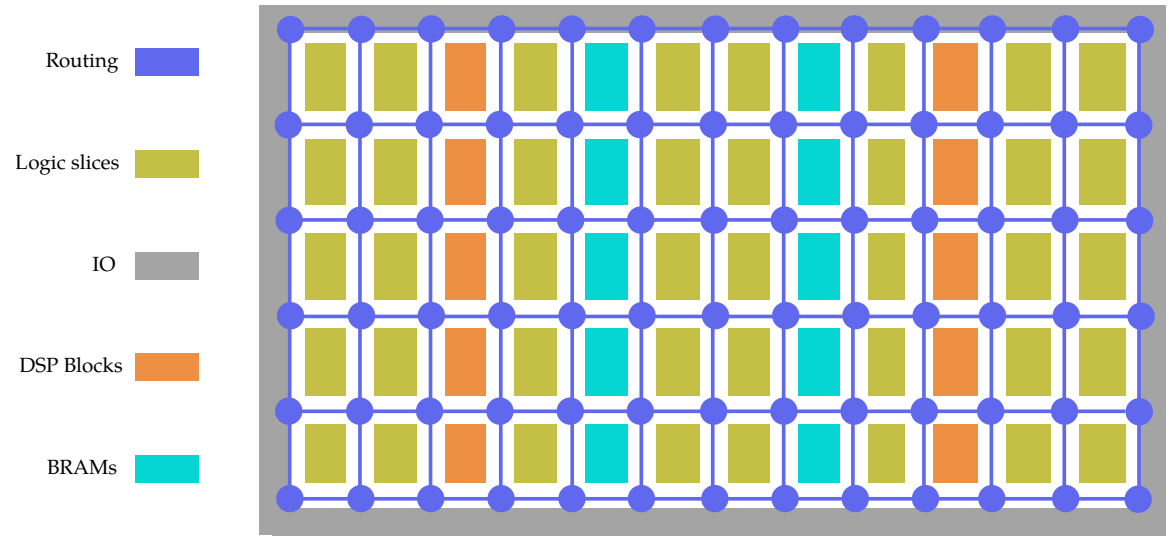


Figure 2.2: Column-based layout of FPGAs

- An IO block can be configured for many standards and voltage levels, e.g. LVDS 2.5V.

Figure 2.2 illustrates how these blocks are typically placed on an FPGA chip. The different components are distributed in a column-based manner. After synthesis, each resulting hardware block is mapped to a suitable physical instantiation. Then, the available interconnect logic is configured to route the design.

2.2 Overlay Architectures

During the implementation of a hardware design on an FPGA, the CAD tools have to solve multiple NP-hard problems: synthesis, technology mapping, placement and routing. As the integration capacities of modern FPGAs are still improving according to Moor's law [84], the conventional FPGA tool flow is becoming less attractive. While it takes a couple of seconds to compile, download and test a GPU kernel, for example, FPGA-CAD may need hours. Even minor modifications would imply repeating all implementation steps from the beginning.

Building another overlay layer on top of the FPGA substrate helps to shorten the development cycles further. Basically, it reproduces the FPGA compute power using a more coarsened architecture. Hence, mapping an application to an overlay is simpler but less efficient than a direct implementation on the FPGA fabric. Once deployed,

a single overlay is capable of executing multiple tasks with low reconfiguration overhead when the context is switched. Application-specific overlay customizations can often be automated by the CAD tools. Even manual adaptations are simpler and more intuitive than modifying the RTL- or even the HLS-code. Moreover, professionally designed overlays are highly optimized to fit to the FPGA-primitives and to operate at relatively high frequencies. Reaching this efficiency for each RTL implementation of the targeted tasks may be not affordable regarding development time.

An overlay can take many forms like an Application-Specific Instruction-set Processor (ASIP), a mesh of arithmetic units or a network of processors, an so on. Although their might be different ways to classify such a diverse family of architectures, we will try to cover the main scientific contributions in this field.

2.2.1 Soft Scalar Processors

Synthesizing a soft processor on FPGA fabric is a popular practice to implement control and non-time-critical tasks. Although a hard processor or microcontroller may consume less power and be faster, many developers prefer the soft version for the following reasons:

- Using a hard processor increases the chip count as well as the board complexity.
- Data exchange between a soft processor and other FPGA-modules remains on-chip. This is faster and less power consuming than off-chip communication.
- Future updates to the architecture of the soft processor are possible without significant development effort. On the other hand, changing a hard processor may imply significant hardware as well as software modifications.
- Soft processors are more suitable for design space exploration and can be customized to fit better to the application needs.
- Having the whole system on the FPGA fabric facilitates the development process including debugging cycles with a single set of CAD tools from the same vendor.

Although the throughput of a single scalar soft processor may be worse by three orders of magnitude than the FPGA compute bound, it illustrates the most flexible solution when a hundred Mega Floating-Point Operations Per Second (MFLOPS) are sufficient.

Soft Processors from FPGA Vendors

Most FPGA manufacturers provide official support for soft processors, e.g. Xilinx, Intel, Microsemi and Lattice. Some of them were developed specially for FPGAs, and they have never been implemented directly on silicon, e.g. NIOS II [9] from Intel, MicroBlaze [96] from Xilinx. Nowadays, most of them are 32-bit Reduced Instruction Set Computer (RISC) machines that can be customized for minimum area, maximum operation frequency, floating-point support, etc. In comparison to non FPGA-centric open-source processors, e.g. LEON3 and Amber, the ones developed by the FPGA vendors are proven to be better regarding performance and area-efficiency [43].

Soft Open-Source FPGA-Centric Processors

Modern FPGA fabric is getting more heterogeneous: there are many types of logic slices next to BRAMs and DSP blocks, each can be operated in multiple modes. An efficient soft processor design should fit onto these FPGA primitives.

Jan Gray explained many tricks on how to design a processor for FPGAs in [36]. Using the UltraScale Kintex KU040 FPGA, he could implement 400 RISC-V RV32I processors [35]. Another efficient architecture designed specially for the 7 series FPGAs from Xilinx is iDEA [24]. Its pipeline consists of 5 to 9 stages and its ISA is built on the DSP48E1 primitive. It can be operated at about double the frequency of a MicroBlaze, and it needs around half of its area.

Soft Multithreaded Processors

To improve the efficiency of a scalar processor when executing data-oriented tasks, multithreaded architectures can be advantageous. If several threads are used, and instant context switch is supported, the latency of a memory operation in one thread, for example, can be covered by arithmetic instructions from other threads.

CUSTARD is one the first multithreaded soft processors [31]. It uses a 4-stage pipeline. BRAMs are used to hold the register files of up to 4 threads. A context switch can be triggered periodically on a cycle-basis or as a result of runtime events like cache misses. Besides, Fort *et al.* [34] showed that significant area reductions can be achieved by using a 4-way multithreaded processor in comparison to 4 processors of the same type. The proposed architecture can save 40% of the occupied area with

a slight performance degradation. The Octavo processor family is designed for Intel FPGAs [57]. It uses 8-threads in a strict round-robin scheduling to fill a pipeline of 10 stages. Octavo is designed especially to operate at frequencies up to 550 MHz. It does not have a compiler support yet, but it outperforms a NIOS II/f processor and with a smaller area footprint [55].

Soft Superscalar Processors

Memory operations have different latencies, and they may cause pipelines stalls. A superscalar processor tries not to stall on these events by re-ordering the instructions before execution. It tracks the dependencies of a window of instructions and issues a ready instruction as soon as its dependencies are resolved.

Morpheo is a superscalar processor that uses a ReOrder buffer to manage the out-of-order execution. [76] describes an FPGA-friendly implementation of this buffer using BRAMs. However, the proposed processor can only track 16 instructions and it runs at a maximum frequency of 79 MHz on a Virtex5 XC5VLX330 FPGA. SPREX [5] is another FPGA-superscalar processor that operates at 133 MHz on a Stratix III. In comparison to the NIOS processor, the area overhead caused by adding the scheduling logic is about 19%, and the accompanied performance improvement is about 9%. Wong *et al.* explored the microarchitecture needed to implement the out-of-order scheduler [88]. The proposed design can track up to 40 instructions without frequency degradation and it runs at 240 MHz. An averaged 2x speedup is expected over normal execution.

Soft Very Long Instruction Word (VLIW) Processors

To improve the compute capabilities of a scalar ALU, its internal components can be used to execute multiple different instructions at the same time. For instance, logical and arithmetic operations are usually computed on different logic. If only one instruction is executed per clock cycle, either the arithmetic or the logical part has to stay idle. A well-designed compiler can extract data and control dependencies and combine multiple operations into a single VLIW-instruction according to the targeted ALU architecture (see Figure 2.3). Indeed, the main challenge when designing soft VLIW processors is the realization of the multi-ported register file on an FPGA. Therefore,

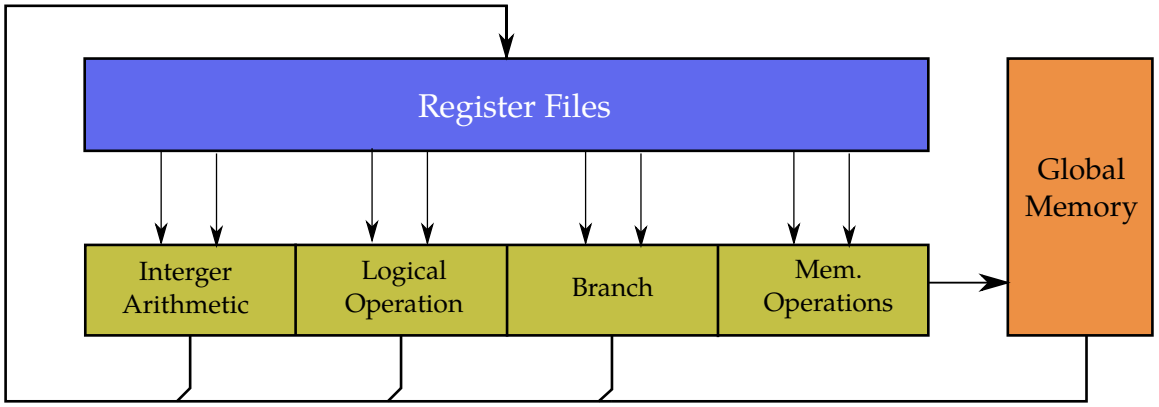


Figure 2.3: A typical VLIW processor

a successful and scalable realization should consider the physical mapping to FPGA primitives [73].

TILT is a soft VLIW processor that can be programmed in C code with an integrated memory fetcher [75]. It uses multiple deeply pipelined single-precision floating-point Functional Units (FUs) connected to a read and a write crossbar. A throughput of 41%-80% of equivalent HLS implementations is achievable with this architecture. On average, it takes only 38 seconds to program a new application onto an existing TILT processor.

VLIW-SCORE is a similar architecture but is supports double-precision floating-point arithmetic [48]. For simulating SPICE circuits, it achieves an average speedup of 2.6x over a sequential implementation running on an Intel Core i7-965 when a Virtex-6 LX760 FPGA is used.

2.2.2 Mesh Networks

A mesh overlay comprises a set of FUs connected via a Network-On-Chip (NoC) (see Figure 2.4). Over the last two decades, many researchers have investigated a plethora of different mesh architectures. Therefore, we will concentrate more on categorizing them and briefly explain the state-of-the-art architectures.

According to the utilized NoC, one may distinguish between:

- *Circuit-switched networks*: data is exchanged on routes that are computed offline and remain fixed at runtime [41].
- *Packet-switched networks*: paths are negotiated at runtime [50].

Figure 2.4: A simplified block diagram of a mesh overlay

The complexity of the utilized FUs may vary from simple arithmetic operations [23] to complete processors [61]. According to their types, one may distinguish between:

- *Heterogeneous overlays*: different types of FUs are used. They may be chosen statically according to the application needs [75] or customized dynamically using partial reconfiguration [45].

- *Homogeneous overlays*: the same FU is replicated but reconfigured later for each running application [42].

In addition, an overlay can be customized for an application by [41]:

- *spacial reconfiguration*: an FU executes the same arithmetic operation and the data is transferred over point-to-point dedicated connections.

- *time-multiplexing*: the functionality of compute and interconnect logic changes on a cycle basis to perform the required task, e.g. MPSoCs [32] or Soft Coarse-Grained Reconfigurable Arrays (SCGRAs) [61].

Coarse-grained Mesh Architectures

HLS tools have to solve the allocation, scheduling and binding problems to implement any design [22]. Using pre-synthesized coarse-grained architectures, only strongly reduced scheduling and binding problems have to be solved. Therefore, mapping designs on mesh overlays is expected to be faster but less efficient than HLS. However, individual publications have reported better performance for some overlays in comparison to HLS [42].

Jain *et al.* showed that using 2D mesh overlays can be up to 1200x faster than realizing circuits described in RTL on FPGA fabric [41]. Using partial FPGA reconfiguration technology [98], customizing an overlay for a new application can take 1000x less time than programming the whole FPGA. However, the DMA-based data transfer to an overlay is a major bottleneck [41]. On an *N*x*N* mesh, it is proven that even having 1 IO Node for each boundary-FU is not enough to exploit the whole compute potential of a mesh overlay [42].

Capalija *et al.* propose mapping DFGs (Data-Flow Graphs) to a mesh of FUs [23]. A realization of 18x16 mesh at 312MHz on a Stratix IV with floating-point support is reported. A peak theoretical throughput of 22 Giga Floating-Point Operations Per Second (GFLOPS) can be achieved. The suggested tool flow is capable of executing many instances of a Data Flow Graph (DFG) per clock cycle.

To improve the compute-to-interconnect resource utilization ratio, multiple primitive DSP blocks can be used to construct a single integer FU. The authors of [42] suggests using VPR [20] as a basis software to schedule operations, route data and generate the configuration bits of the mesh overlay. VPR is an academical tool widely used for placing, routing and generating bitstreams for FPGA fabric. In comparison to HLS, the investigated overlay has 40% better throughput, on average, but it needs 30-70x more FPGA area.

Soft Coarse-Grained Reconfigurable Arrays (SCGRAs)

Instead of using simple FUs to build a mesh overlay, relatively complex Processing Elements (PEs) can be deployed to execute a sequence of software instructions. For instance, a PE comprising an ALU with a instruction Read-Only Memory (ROM) has been used in [61]. To accelerate a compute intensive loop, its C/C++ code is first transferred into a DFG with user-specific unrolling factor. Then, the DFG gets scheduled on

a previously designed overlay, whose bitstream is already computed. Meanwhile, the contents of the ROMs are computed and the bitstream is modified accordingly. The proposed tool flow generates a driver automatically to control the SCGRA. A speedup between 6-9x is achieved in 4 benchmarks over the embedded ARM processor on the XC7Z020 SoC. However, an efficient use of the computing power of an SCGRA requires buffering all of the processed data locally in BRAMs. Also, scalability issues have been reported when increasing the size of the SCGRA to include more than 5x5 PEs.

Soft Multi-Processor System-On-Chip (MPSoC)

When general purpose processors are used as FUs, major simplifications in the overlay programming model, as well as the required CAD tools, can be achieved: workload distribution becomes as complex as scheduling multiple threads on multiple processing cores. For example, an MPSoC can operate as a Symmetric Multi-Processor (SMP) system where a Linux OS is running on top. The accumulated compute power of the constituent processors can be exploited by any running process [63].

MARC [59] is an MPSoC that comprises 48 MIPS-based processing cores implemented on a Virtex-5 FPGA. Thread scheduling is managed in hardware and the execution is based on the OpenCL model, like a normal GPU. However, the whole system is programmed with C. For the Bayesian network inference problem, a manually optimized RTL-implementation can outperform MARC by a factor of 3x but with a remarkable extra development effort.

OOGen is a design framework that offers an automatic generation of MPSoCs using an object-oriented programming language [32]. Each PE comprises a single MicroBlaze processor. The generated overlay can be programmed using a pthread-compliant microkernel (*HThreads*) or an OpenCL-like programming model (*HOpenCL*). Up to 32 PEs can be implemented on the XC7Z045 Zynq SoC using about 130K LUTs (85% utilization). Speedups up to 4x over the embedded ARM Cortex-A9 are reported.

In fact, we will prove later that using individual soft processors for general purpose computing is not a pareto-optimal solution in terms of energy consumption, performance and compute density.

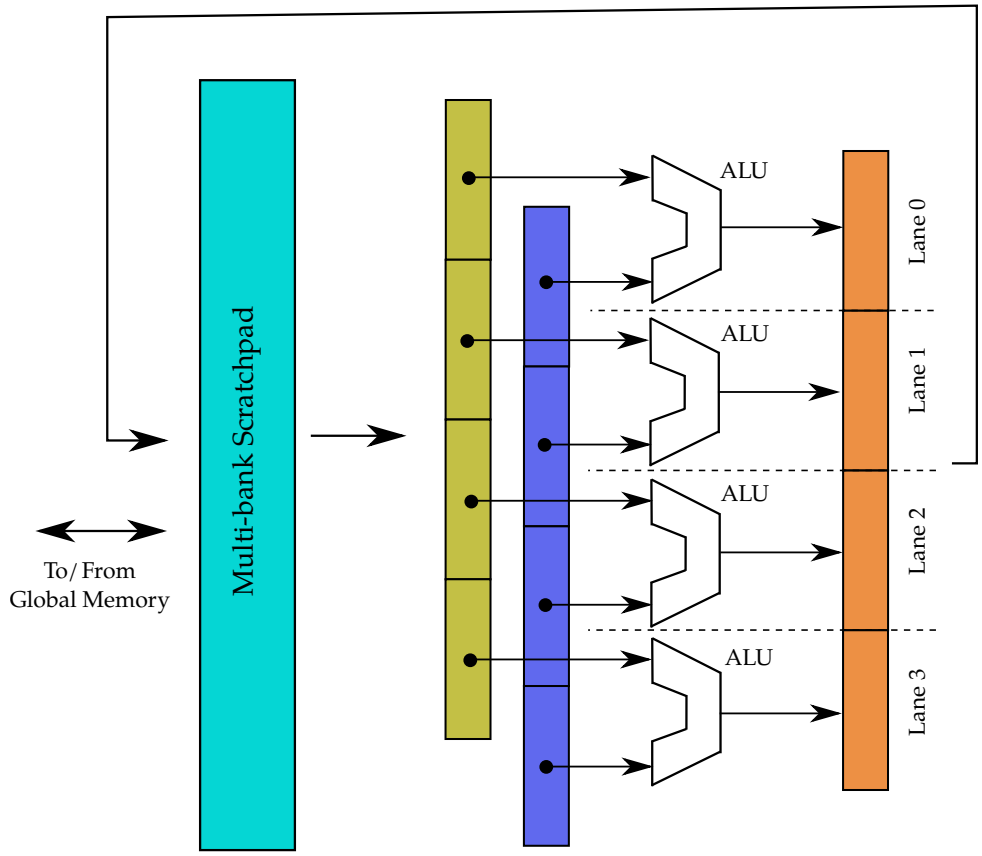


Figure 2.5: A simplified block diagram of a vector processor

2.2.3 Soft Vector Processors

To improve the throughput of a scalar processor, its ALU and register file may be replicated to build a vector processor. The resulting Single-Instruction Multiple-Data (SIMD) machine executes a single instruction simultaneously on multiple vector lanes (see Figure 2.5). The bitwidth of the vectorized data designates the architecture, e.g. the 128-bit NEON engine from ARM or the 512-bit SIMD of Intel Xeon Phi. Such architectures are usually built as coprocessors, i.e. another processor is required for control [2].

On FPGAs, using FFs to synthesize the register files is not a scalable solution. Instead, multi-ported scratchpad memories are realized with BRAMs and used to hold all registers. Data traffic to global memory is organized with Direct Memory Access (DMA) transfers to/from the scratchpad (see Figure 2.5). Efficient programming for vector coprocessors should be done manually using intrinsics. However, code vectorization can be automated by the compiler [49].

[2] *Vector processors* and *vector coprocessors* will be used interchangeably in the rest of this thesis

Integer Soft Vector Processors

VIPERS [105], VEGAS [26], VENICE [78] and MXP [79] are different generations of soft vector processors developed at the University of British Columbia, mentioned in chronological order. The recent architectures use fracturable ALUs: for example, a single 32-bit or four 8-bit add operations can be computed in a single cycle. The DMA transfers to global memory and the ALUs operate concurrently. Vector conditional instructions are also supported. The most recent version of MXP can accommodate up to 64 lanes and achieve speedups of up to 912x for matrix multiplication. In comparison to the NIOS II/f scalar processor, VENICE has up to 5.2X better performance per unit area.

The VESPA [104] architecture supports vector chaining which enables simultaneous execution of multiple vector instructions on different functional units, e.g. on an ALU or a memory unit. This technique can provide 15-45% performance improvement with an area overhead significantly less than doubling the number of lanes. Besides, heterogeneous lanes were investigated and found to reduce the area demands by 6-13% with little or no performance loss. By subsetting the instruction set, the performance per area of a soft vector processor can be improved by an average of 25% [103].

The most efficient vector processor we have found is the Octavo [55]. It runs at higher frequencies than all other soft vector processors: 525-370MHz for 1-32 lanes, respectively. Also, it has fewer area demands and better performance per area than MXP.

Floating-Point Vector Processors

In conventional vector coprocessors, the tight coupling of many ALUs with a single scratchpad leads to a remarkable reduction in the operating frequency when increasing the number of lanes [79] [56], or alternatively, when accommodating large ALUs equipped with floating-point hardware. Also, the wider the bitwidth of processed data is, the worse the performance will be. For instance, Kapre [49] shows how the speedup of a soft vector processor degrades on average by a factor of 2x over scalar execution for the same application when processing 32 bit data instead of 8 bit. Therefore, the good performance and area efficiency for fixed-point arithmetic on these architectures should not be expected to hold for the floating-point case.

FPVC is a vector processor with single precision floating-point support [51]. The reported implementation on a Virtex5 has 8 lanes and requires 26K LUTs. It achieves speedups of up to 5x over an on-chip PowerPC processor supported by a floating-point IP-core form Xilinx IP Library.

Although the original MXP architecture has no floating-point support, the authors of [80] propose adding application-specific custom pipes capable of half floating-point precision (16bit) to the original design. For the N-body problem, a speedup of about 7200x over a NIOS II/f processor with no floating-point hardware is reported. However, the complexity of manually designing these pipes and integrating them into the MXP is a very time-consuming task.

An automatic generation of application-specific SIMD extensions to an embedded processor is suggested in [37] . The number of vector elements can be parametrized and an automatic vectorization of the application code is described. With this flexible toolflow, a speedup as well as a power saving of up to 5x can be achieved over scalar execution. The biggest extension needs less than 55K LUTs on a Virtex5 device, but only addition and multiplication are supported.

Lu *et al.* [62] propose another vector coprocessor with floating-point (+,-,*) support. A version with 4 lanes was realized on a Virtex6 at 100MHz using 69K LUTs (46% utilization). The authors expect speedups of up to 1808x with 64 lanes over a single MicroBlaze. However, the feasibility of the required coprocessor has not been discussed.

2.2.4 Soft GPU Architectures

Due to the high needed FPGA area, most of the successful soft GPU architectures for GPGPU have been designed over the last decade. To the best of our knowledge, the first attempt was published in 2010 by Kingyens, *et al*. [53]. The proposed design supports the basic features of a GPU-like architecture, e.g. one-cycle context switching. However, it was not implemented completely in RTL.

The Guppy soft GPU was produced in 2012 [6]. It is based on LEON3 and can be programmed with CUDA. When performing matrix multiplication on up to 64x64 integer matrices, speedups of up to 3x over the LEON3 can be expected. Guppy operates at 70MHz and needed about 31K LUTs on Virtex5.

In 2013, Andryc, *et al.* presented another CUDA-programmable architecture based on the NVIDIA G80 named FlexGrip [13]. Its ISA supports 27 instructions without floating-point operations. On the Virtex-6 VLX240T, a single Streaming Multiprocessor (SM) with 8 PEs is realized at 100MHz. A MicroBlaze is used to control the system and hold the data to be processed in its cache. An average speedup of 12x is reported over MicroBlaze for a data size of 256. Power estimations indicate an average energy saving of 80%. When simulating bigger versions, speedups over 40x are achievable with 2 SMs each with 32 PEs. However, this needs an FPGA with about 413K LUTs [14].

Neko is a special implementation of the MIAOW [18] GPU presented in 2015. It is an open-source implementation of the AMD Southern Islands architecture, and it can be programmed with OpenCL. Neko does not own a memory controller and it needs an external processor to schedule the work-items on it. It accommodates 16 ALUs capable of performing fixed- and floating-point arithmetic. The developers reported a successful synthesis on the Xilinx Virtex7 XC7VX485T with 195,285 LUTs (64% utilization) but without any further details on the performance.

Nyami [21] is another open-source architecture for soft GPGPU published in 2015. It was deployed on a Cyclone IV EP4CE115F29C7 at 30MHz and consumed 81% of the device logic. Four 16-wide vector processors capable of floating- and fixed-point arithmetic are included. Nyami can be programmed with C/C++. Unfortunately, no time or power measurements are reported for the FPGA implementation.

Besides, we observed other attempts to implement GPUs partially or completely in RTL on FPGAs but for graphics and not for GPGPU. These projects were discontinued [30] [64]. Nowadays, performing GPGPU with OpenCL or CUDA on FPGAs seems to be the only motivation to realize soft GPUs.

2.3 High-Level Synthesis (HLS)

HLS enables the generation of parallelized RTL-implementations out of high-level software descriptions. It has a much shorter edit-compile-verify cycle than designing with HDL. Hence, HLS allows significant cuttings in the NRE development costs. Nowadays, many programming languages are supported as entry points for HLS. However, C/C++ and OpenCL are the most used ones in the FPGA world.

The Quality Of Results (QoR) of HLS tools has been improving over the last years. Many publications estimate the performance gap between HLS and manually and

highly-optimized HDL designs to be negligible [40][74]. Nowadays, a plethora of academic and commercial HLS tools are available. A recent survey than compared LegUp [22], Bambu [72] and DWARV [65] with a commercial HLS tool shows that not a single tool could provide the best results on all benchmarks [66]. Nevertheless, the application area of HLS is still smaller than the one of HDL. It is recommended to use it only for data processing tasks. For instance, doing cycle-accurate verification with HLS is still out of the focus of FPGA vendors.

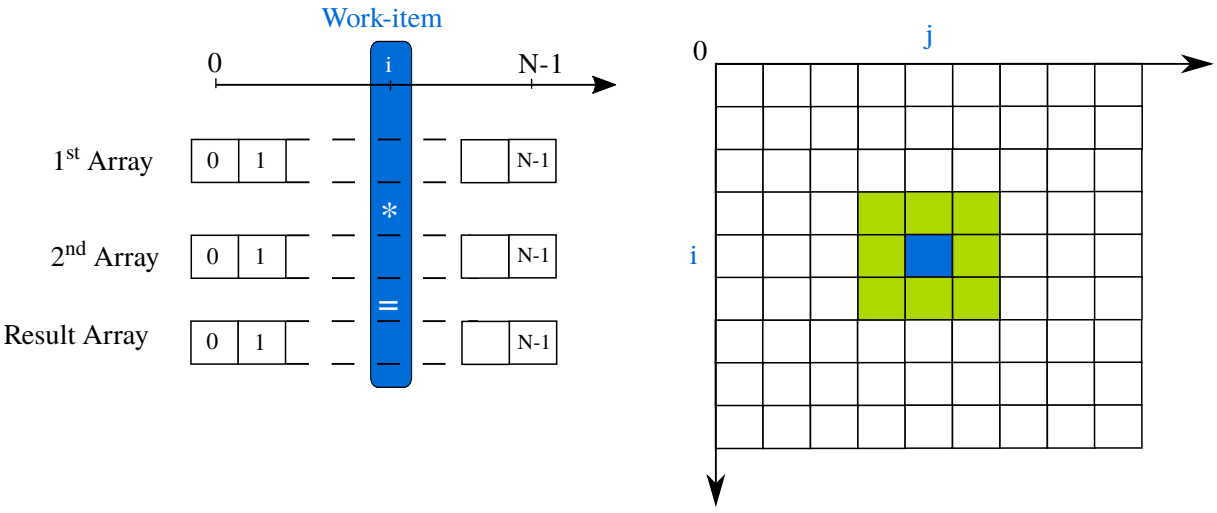
2.3.1 HLS vs. Overlays

When overlays are used, applications get mapped to predefined hardware architectures. On the other hand, HLS creates the required hardware resources according to the application requirements. Therefore, the performance, as well as the performance per area of overlays, are theoretically upper bounded by the ones of HLS designs. However, since the perfect HLS tool has not been developed yet, the opposite may hold in some individual cases. Moreover, if some task parameters are not fixed at compile time, generating efficient hardware designs as well as scheduling the required operations on them by HLS tools become more difficult. Under these conditions, an overlay with runtime scheduling may outperform HLS. A detailed evaluation of this case study is presented in Chapter 7.

2.4 General Purpose Computing on GPUs

In 2007, NVIDIA introduced CUDA as a programming language to use its GPUs for general purpose computations [68]. One year later, the Khronos group published the OpenCL specification [52] as the manufacturer-independent programming language for General Purpose Computing on GPUs (GPGPU). Since that time, application areas of GPUs have expanded remarkably: not only on top of graphic cards in personal computers, but also in high performance computing centers [4] and embedded systems [69].

Equipped with plenty of ALUs capable of single and double floating-point precision, GPUs can achieve remarkable accelerations over CPUs in many applications [33][60]. In the embedded domain, GPUs with a reduced power-consumption may be preferred



(a) 1D Index Sapce - vector multiplication (a) 2D Index Sapce - median image filter

Figure 2.6: Index space of the SIMT execution model when computing vector multiplication and a median image filter

for applications like deep learning [69] [39]. Nowadays, modern embedded SoCs tend to integrate hard GPUs next to CPUs [17] and FPGA-Programmable Logic (PL) [97].

The HW architecture that has been developed for this thesis, as well as its tool flow, are based on the platform, execution, memory and programming models of OpenCL [52]. Although a detailed description of the OpenCL standard is out of the scope of this thesis, the needed OpenCL terms will be briefly explained in the context of later sections.

2.4.1 Single-Instruction Multiple-Threads (SIMT)

SIMT is the execution model for GPGPU programming languages [67] [52]. It assumes that the same code will be executed in many work-items (threads) on multiple cores. To distribute the workload over the individual work-items, the concept of *Index Space* is introduced: each work-item is mapped to a tuple of coordinates in a 3D space called the *work-item global ID*. Figure 2.6.a illustrates an example when computing vector multiplication. The work-item whose index is *i* will multiply the i-th elements from the input arrays and write the result at the i-th entry of the result array. In this example, a 1D index space is sufficient. In Figure 2.6.b, a 2D index space is used for a 3x3 median image filter. One work-item is launched for each pixel in the input image: the one

whose global ID is (i, j) will read the corresponding pixel as well as all of the neighbouring ones from the input image, compute the median value and write the result at the pixel (i, j) in the output image.

Inter-thread Communication

OpenCL or CUDA offer two methods for inter-work-item communication:

- **Synchronization Barriers**: work-items are bundled in *work-groups* and then scheduled on the compute cores. When executing a sync instruction by any work-item, it waits until all other ones in the same work-group reach the sync barrier.
- **Atomic Operations**: they allow a work-item to read and modify a memory address with a single instruction without being interrupted by other work-items. The affected memory address can refer to a local memory inside a compute unit (local atomic) or to a global shared memory (global atomic).

OpenCL Built-in Functions

GPGPU programming languages are based on the sequential C/C++. Therefore, extra functionality is needed to manage the concurrent execution of multiple threads, e.g. to provide the running work-items with their global IDs in the index space. The OpenCL standard specifies a list of the needed functions (see section 6.12.1 of the OpenCL specifications [52]). The ISA of any GPU with GPGPU support should provide the corresponding functionality.

2.4.2 GPU Architectures

A GPU architecture can be simplified as illustrated in Figure 2.7. It hosts multiple Compute Units (CUs), each accommodates a set of PEs. The individual PEs are grouped in Compute Vectors (CVs) that can be considered as SIMD-machines. However, these vectors may support different ISAs. Caches and other on-chip memories are organized hierarchically on multiple levels. In fact, the decoupled CVs and the hierarchical structure enable scalable realizations and simpler programming of GPUs. For example, the NVIDIA Tesla P100 comprises 3584 PEs capable of single-precision floating-point arithmetic [70].

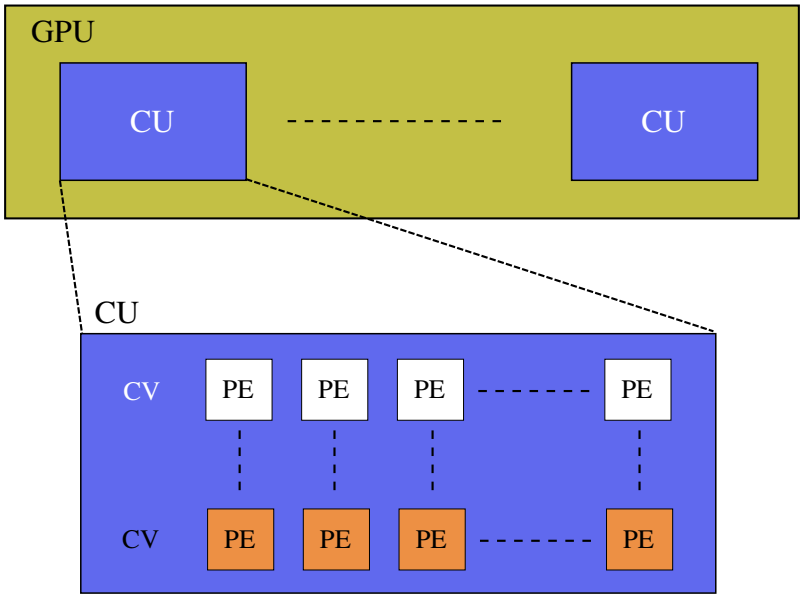


Figure 2.7: A simplified block diagram of a GPU architecture

The internal structure of a CU is more manufacturer-dependent. AMD uses 64 PEs distributed on 4 CVs as in the GCN architecture [12]. The same PE can execute integer, single- or double-precision floating-point operations with different latencies. NVIDIA prefers having a more heterogeneous mix consisting of 64 PEs for single-precision, 32 PEs for double-precision and 32 PEs for other types of operations as in the Pascal architecture [70].

To effectively use the computation capabilities of a CU, instruction scheduling should consider multiple work-items. For instance, while some of them are stalling at a memory load, arithmetic operations from others can be executed. Also, a better utilization of the deep floating-point pipelines can be achieved. For example, a single CU in the GCN architecture is designed to hold and switch instantly between the register files of 2560 work-items [12].

2.5 Summary

FPGAs enable the realization of high-speed digital circuits without going through the chip manufacturing process. To overcome the long development cycles of hardware development, overlays and HLS tools can be used. Both solutions enable the programming from high-level description languages like C/C++ and OpenCL. They

help to reduce the development effort and hence to shorten the time-to-market. HLS-CAD defines the hardware structure according to the task code. On the other hand, overlays are pre-implemented coarse-grained architectures on top of FPGAs, and can be programmed to perform the required functions. For example, an overlay can be a vector coprocessor, a mesh of functional units or a network of individual processors. When overlays are used, hardware and software development are well-separated procedures. Hence, the long cycles of placing, routing and generating bitstreams when inserting minor modifications or realizing new applications on an existing overlay can be avoided.

Because of their high area demands, GPUs are still not widely used as overlays on FPGAs. A successful and efficient realization of a soft GPU would help to program FPGAs in an attractive and easy-to-use GPU-like tool flow.

Chapter 3

FGPU: An Open-source Soft GPU for FPGAs

GPU architectures are relatively complex and most of them have been developed by international companies like NVIDIA, AMD or ARM. Even the open-source architectures are based on the industry ones [18][21]. In addition, GPUs have high area demands. An efficient implementation is only possible on modern middle-sized FPGAs. To the best of our knowledge, most publications that deal with soft GPU implementations for GPGPU report a successful synthesis on FPGAs but on low operation frequencies and without performance or power measurements. To the best of our knowledge, the most successful attempt to implement a soft GPU is reported in [13], where 8 PEs without floating-point support have been synthesized and tested on FPGA fabric. Therefore, we decided to develop our own FPGA-centric GPU which we called later FGPU or the FPGA-GPU.

This chapter explains the hardware details of the proposed architecture. The basic FGPU structure was presented first in FPGA'16 [7], The design of the fixed-point ALU, atomic units and sub-integer load and store operations was published in FPT'16 [46]. Later, thread-level divergence and floating-point support are described in ISVLSI'17 [47].

3.1 Architecture Overview

FGPU is a 32bit multi-core GPU-like processor based on the SIMT execution model of OpenCL (see Section 2.4.1). Figure 3.1 illustrates its main components. FGPU is

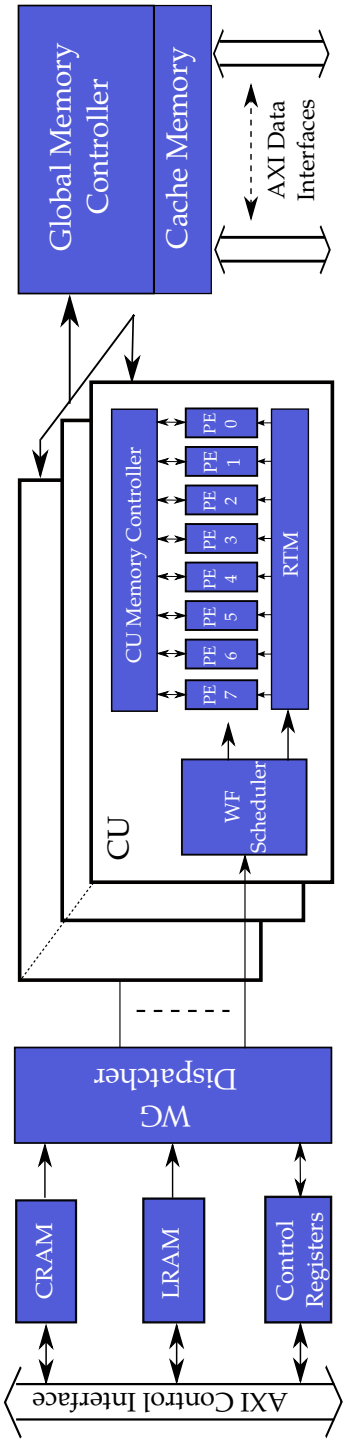


Figure 3.1: Block Diagram of the FGPU architecture

open-source and available completely in VHDL. It is designed to perform General Purpose Computing on GPUs (GPGPU), and it does not replicate any other existing architecture.

An FGPU can accommodate multiple Compute Units (CUs), each has a single SIMD machine of 8 identical PEs. A single CU can run up to 512 work-items and supports full thread-divergence, i.e. each work-item is allowed to take a different path in Control Flow Diagram (CFG). Each work-item owns exclusively 32 registers that can be extended using scratchpad memories. Currently, its ISA consists of 49 instructions including optional native support for floating-point arithmetic. Besides, it supports global atomic instructions that allows performing uninterruptible read-modify-write transactions on any address in the global memory. FGPU can perform integer arithmetic, sub-integer load and store instructions and single-precision soft or hard floating-point arithmetic[1]. FGPU is a deeply pipelined processor: executing a fixed-point arithmetic operation takes 21 clock cycles where only 5 are needed in a traditional MIPS-like architecture.

FGPU includes an central, direct-mapped, multi-ported and write-back cache system, which can serve multiple read/write requests simultaneously. In addition, it integrates multiple data movers that can parallelize the data traffic on up to four AXI4 interfaces. The whole architecture can be controlled on the hardware side through a single AXI4-lite interface. On the software side, only standard OpenCL-API procedures are needed.

FGPU is designed to be:

- *Customizable:* ISA sub-setting is supported on floating-point, atomic and sub-integer store instructions. The properties of cache and scratchpad memories are adjustable. Moreover, the number of IO interfaces can be set at 1, 2 or 4 to improve the bandwidth of the global memory connection.

- *Scalable:* a fixed-point FGPU with up to 64 PEs can be synthesized at 250MHz on Zynq FPGAs where more than 99% of the slices are occupied. If the floating-point logic is included, the maximum observed degradation in the operation frequency is less than 20%.

[1] Soft floating-point operations are performed by emulating the required functionality using fixed-point instructions. Hard floating-point ones are computed by optional hardware extensions to the PEs.

- *Portable:* No IP-cores or commercial libraries are used. However, a library of floating-point operations is needed if native support is desired.

To execute a task, equivalently called a *kernel*, its binary code has to be written in the Code RAM (CRAM). Other information, like parameters' values and the number of work-items to launch, has to be placed in the Link RAM (LRAM). When the execution starts, the WG dispatcher reads the LRAM content and schedules work-groups on the available CUs. A single work-group, which consists of up to 512 work-items, is broken internally inside a CU into a set of wavefronts, each holds up to 64 work-items that share a single Program Counter (PC). When an instruction from a wavefront is executed, the 8 available PEs repeat it over 8 clock cycles to cover all of the 64 member work-items. Each running wavefront is handled by a Wavefront (WF) controller that fetches instructions from CRAM, pauses the execution after a memory access has been submitted until the data gets served, performs branches, etc. Each work-item can use two types of memories to perform its computations:

- *Private Memory*: it comprises 32 registers each is 32 bits wide. They are not accessible from other work-items. Private memory is the closest memory to the ALUs as well as the fastest one. If more than 32 registers are required, an optional scratchpad can be synthesized to spill and fill the registers' contents.
- *Global Memory*: it is an off-chip memory that can be read or written from any work-item. Its size is limited to 4GB.

3.2 Control Interface

FGPU is controlled over an AXI4-lite bus [16]. Through this interface, three internal components can be written or read: CRAM, LRAM and the control registers. A detailed address map for the control interface is given in Appendix B.2. All read- and write-transactions via the control bus have to be word aligned.

3.2.1 Code RAM (CRAM)

CRAM is a local memory that holds the binary code to be executed. Its size can be customized from 1-16KiB, which corresponds to 256-4096 instructions. Since a fully utilized CU executes a new instruction every 8 clock cycle, a single CRAM block can

serve 8 CUs without causing any bottleneck. An additional CRAM block with identical content can be instantiated if more than 8 CUs have to be served. The program code of multiple kernels can coexist in the CRAM memory.

3.2.2 Link RAM (LRAM)

LRAM holds the information needed for kernel execution but not included the binary code, e.g. the number of work-items to be launched and the location of the first instruction of the binary code in CRAM. It is divided into 16 slots of 32 words, where each slot specifies how to execute a single kernel. Appendix B.1 describes the address map of LRAM in details. LRAM can be written through a standard OpenCL-API.

3.2.3 Control Registers

FGPU has 4 control registers. They serve the purpose of starting, controlling and monitoring kernel execution. A detailed description can be found in Appendix B.3.

3.3 Work-Group (WG) Dispatcher

The WG dispatcher gives the pulse to all other FGPU components. When the execution of a kernel is triggered, it begins with the initialization phase: it reads the content of the corresponding LRAM entry and initializes the Runtime Memorys (RTMs) located in the CUs. The initialization data are required later by the OpenCL-related assembly instructions (see Appendix A), e.g. when a work-item reads its global ID. In addition, the WG dispatcher computes which PEs have to be enabled or disabled while a work-group is being executed. Only if the work-group size is a multiple of the maximum wavefront size (64 work-items), all PEs may be enabled during execution[2].

After the initialization phase ends, which takes typically about 500 clock cycles, the WG dispatcher starts workload distribution. It iterates on the individual work-groups and assign them to be executed by the available CUs until all needed work-items have been launched. A CU can simultaneously manage the execution of up to 8 wavefronts. To successfully schedule a work-group consisting of n wavefronts, there should be no

[2] A 100% occupancy of all PEs is only possible when all wavefronts do not diverge at any conditional branch instruction, i.e. all member work-items either branch or do not branch, at the same time.

more than *8 - n* wavefronts already running on the corresponding CU. If all CUs are busy, the WG dispatcher keeps polling. As soon as some wavefronts retire and enough slots get available, another work-group gets assigned.

3.4 Compute Unit Design

Figure 3.2 illustrates the block diagram of a CU: the *WF scheduler* (on the left) is responsible of scheduling up to 8 wavefronts on a *Compute Vector (CV)* of 8 PEs (on the middle); memory operations on local scratchpads or global memory are managed by the *CU memory controller* (on the right); the RTM memory (on the bottom) is accessed when executing the OpenCL-related instructions.

3.4.1 Runtime Memory (RTM)

Work-item built-in functions are a part of the OpenCL standard [52]. The data they provide has to be computed at runtime, e.g. *get_global_id()* has to return the global ID of the calling work-item. RTM is a memory that holds the corresponding information. It is written by the WG dispatcher or the WF scheduler when a work-group or a wavefront gets scheduled on the CV, respectively. Later, it gets read when an OpenCL-related assembly instruction is executed.

At the beginning of kernel execution, the WG dispatcher needs less than 300 clock cycles to simultaneously initialize all RTMs in all CUs. When wavefronts get scheduled internally within a CU, the RTM content gets updated within three clock cycles. In addition, the storage demands of RTM is relatively low: a single BRAM block operating in the 512x72 simple dual-port mode [95] is sufficient. Although each CU owns an RTM instance and most of the RTM content is shared between all CUs, realizing a single shared block is not expected to be area-efficient. The accompanied area overhead of the scheduler needed to serve the simultaneous read requests from many CUs is much significant than the one of an extra BRAM block per CU. Besides, using a single RTM instance would create another bottleneck at the RTM read port.

Currently, the ISA of FGPU offers 5 basic instruction to implement the work-item functions of the OpenCL standard (see OpenCL-related instruction in Appendix A). The compiler can use them to synthesize the required functionality (see Section 4.2). However, these 5 instructions are not sufficient to fully support the OpenCL standard,

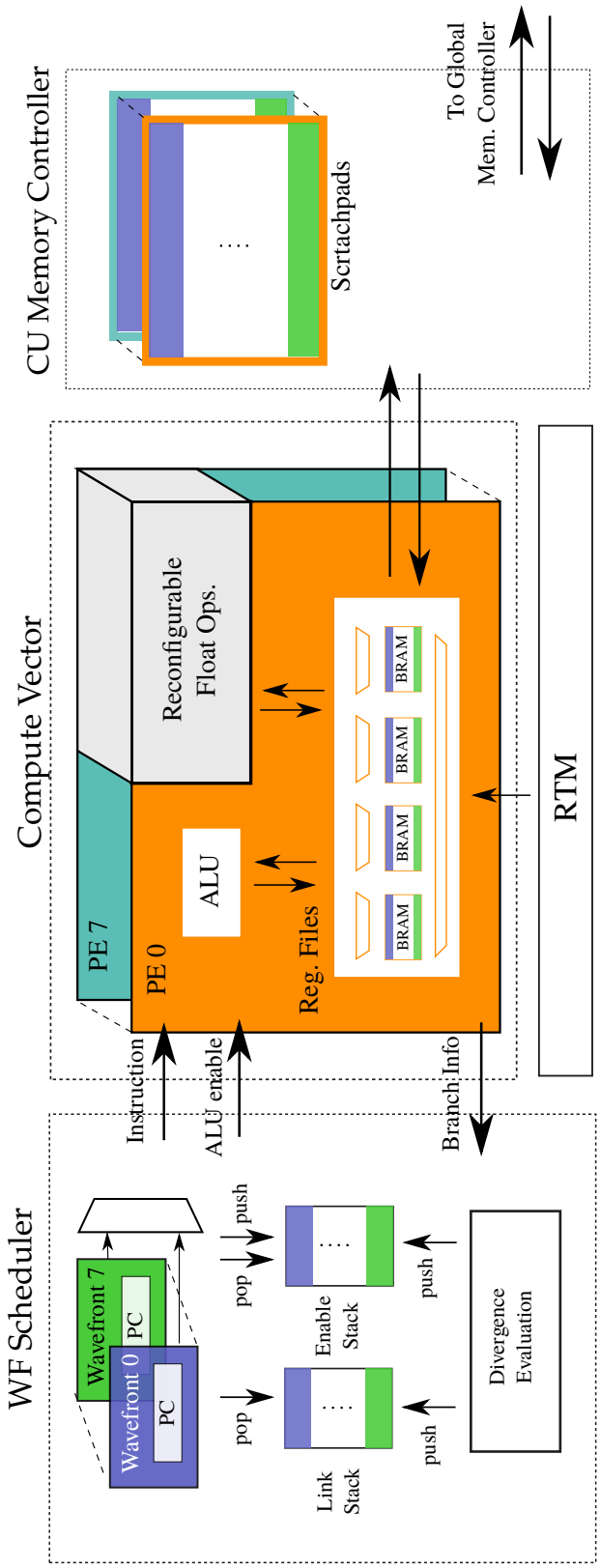


Figure 3.2: Block Diagram of a Compute Unit (CU)

but implementing the missing ones will neither change the proposed design nor increase the storage demands of RTM. Also, passing kernel parameters is achieved using RTM.

3.4.2 Wavefront (WF) Scheduler

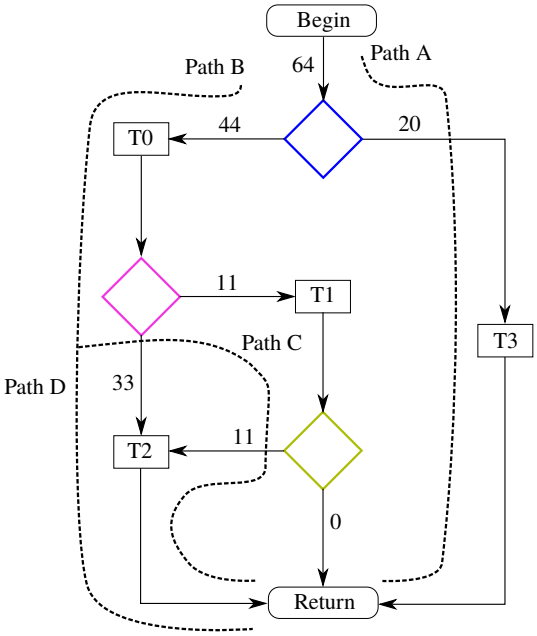
To save hardware resources, FGPU performs resource allocation and operation scheduling not for individual work-items but for complete wavefronts. For example, a single PC is used for the whole wavefront. The execution of a single instruction is repeated spatially over the 8 PEs and temporally over 8 clock cycles. Hence, to achieve maximum CV utilization, only one new instruction has to be fetched every 8 clock cycles.

The WF scheduler has to organize the execution of 8 wavefronts. Each is handled by a separate controller that manages instruction fetching, PC update, pausing the execution e.g. until a memory load is performed or the results of the previous instruction are written into the register files, etc. The WF controllers react with a central one which decides which wavefront is forwarded to execution. Because of their high latencies, wavefronts with memory or branch instructions are prioritized. Scheduling from multiple wavefronts helps to mitigate the delay of memory instructions and the deeply pipelined CVs.

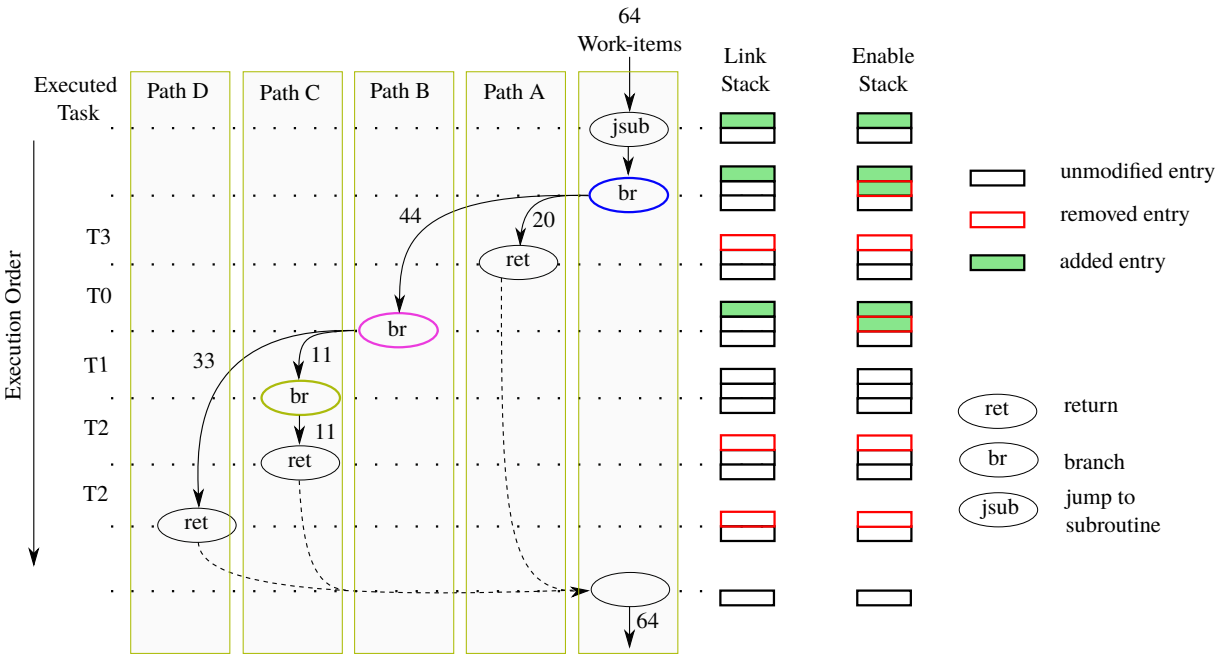
To simplify the scheduling logic, it is assumed that all floating-point operations have the same latency. The unified value is set at synthesis time to the maximum latency of any individual floating-point operation. Out-of-order execution of instructions from the same wavefront is not supported because the expected area overhead is relatively high. Indeed, the overall performance may be improved using other methods without degrading the throughput per area ratio, e.g. by increasing the number of realized CUs.

Thread-Level Divergence and Subroutine Calls

When the execution of a wavefront hits a conditional branch, some work-items may have to jump to a specific instruction while others do not. However, having a single and shared PC value allows taking only one of the two paths. For instance, when going through a for-loop, only the case where all work-items perform the same number of iterations may be tolerated. To enable full thread divergence, the CU scheduler use the *Link* and *Enable* stacks (see Figure 3.2).



(a) Control flow diagram



(b) Execution order

Figure 3.3: Handling branch instructions and subroutine calls

When branching into two different directions, the execution continues along one of the two paths. The address of the first instruction of the other path is pushed on the link stack. Besides, the top entry on the enable stack is removed and two bit masks are pushed instead, where the new entries define which work-items are enabled for each

path. When hitting a return instruction, the top entries of both stacks are popped out. Then, the execution continues along the path that was stored at the top. Similarly, handling subroutines is realized using the proposed stacks: the return address is pushed on the link stack on calling and popped out on return; the top entry of the enable stack is duplicated on calling and popped out on return as well.

For a better understanding, Fig. 3.3.a illustrates a sample CFG where different paths are taken by a full wavefront. The numbers next to the arrows represent how many work-items go along the corresponding path. Fig. 3.3.b shows how the CFG is executed on FGPU. When all work-items select the same path at a branch, the stacks remain unaffected (see path C in Fig. 3.3). Even if two paths merge later before a return instruction, the shared code will be executed twice (T2 shared by paths C and D).

Following the minority-path at first reduces the required maximum size of the link and enable stacks: for a full wavefront of 64 work-items, only $\log_2(64)$ entries are needed to manage the worst case scenario, even when each single work-item takes a different path. In addition, since a CU can host only up to 8 wavefronts, a single BRAM36E1 [95] memory block per stack is sufficient for physical realization.

3.4.3 Compute Vector (CV)

A CV accommodates 8 PEs with an ISA of 49 instructions listed in Appendix A. As illustrated in Figure 3.2, the floating-point logic is not implemented within the individual PEs. This is motivated by the partial reconfiguration approach explained in Section 7.4.

ALU Design

Modern FPGAs have natively DSP blocks capable of performing the operation $a*b\mp c$, where the bitwidth of a and b is greater 17 bits and the one of c is greater than 32 bits [97] [10]. The Arithmetic Unit (AU) design exploits these blocks to implement not only multiplication instructions, but also all types of shifts. To realize the operations listed in Table 3.1, only 4 DSP blocks are needed (see Figure 3.4). The instruction $R_s\ shl\ R_t$ is computed by multiplying 2^{R_t} with R_s and taking the value of the res_{low} port. Similarly, $R_s\ srl\ R_t$ is computed as $R_s\ *\ 2^{32-R_t}$ and captured at the res_{high} port. In case of arithmetic right shift, a sign correction has to be done:

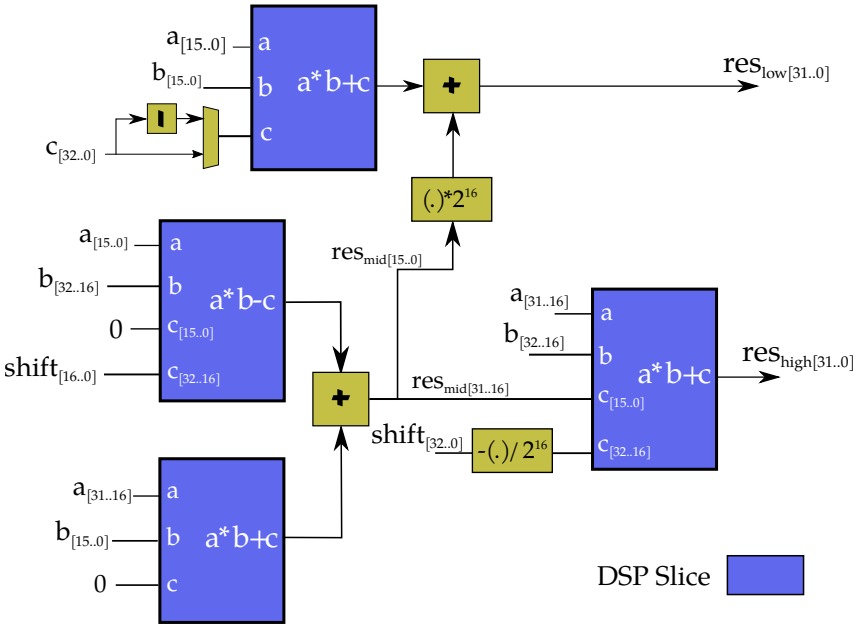


Figure 3.4: Functional block diagram of the AU of a PE

Table 3.1: Configuration modes of the fixed-point arithmetic unit shown in Figure 3.4

Op.	Inputs				Outputs	
	a	b	c	shift	res_{low}	res_{high}
ADD	R_s	1	R^t	0	R_s+R_t	-
SUB					R_s-R_t	
MACC		R_t	R_d		R_d+R_s*R_t	
MUL			0		R_s*R_t	
SHL		2^{R_t}			R_s shl R_t	
SRL		2^{32-R_t}			-	R_s srl R_t
SRA				$R_{s[31]}$*2^{32-R_t}		R_s sra R_t

$$R_s\ sra\ R_t\ =\ R_s\ srl\ R_t - R_{s[31]} * 2^{32-R_t}$$

Where $R_{s[31]}$ is the sign bit of R_s.

Actually, shift operations are executed in most processors using barrel shifters. If no DSP blocks were used, a 32bit barrel shifter would cost 32 instances of a 32:1 multiplexer. For instance, on Xilinx 7 series architecture, where a 16:1 multiplexer can be implemented with a single slice (4 LUTs) [93], a barrel shifter would cost more than 288 LUTs. The associated area overhead is much more than the one of the multiplier-based

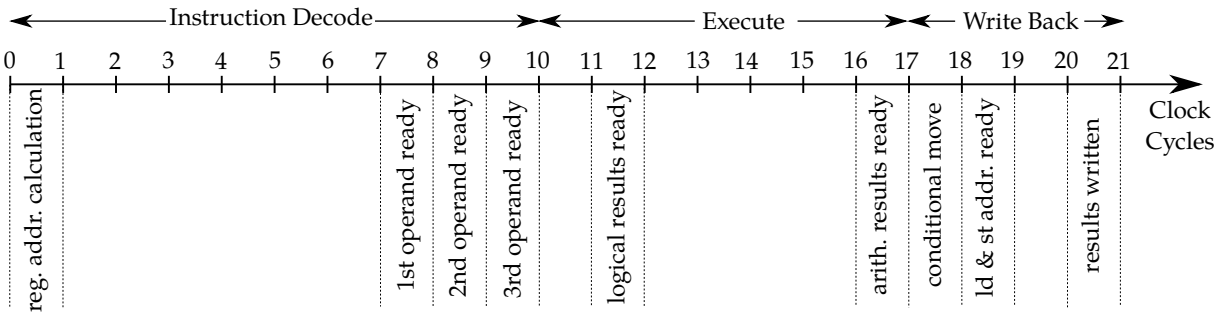


Figure 3.5: Pipeline of the fixed-point ALU inside a PE

realization. In addition, the AU illustrated in Figure 3.4 is generic and can be realized effectively on most modern FPGA architectures without using any manufacturer IP-cores or primitives. The proposed AU contains 7 pipeline stages.

The logic unit design is much simpler and it spans only over 2 pipeline stages: all supported logical instructions are implemented in the first stage and a multiplexer selects the result according to the operation code in the second.

FGPU Pipeline

The PEs are relatively deeply pipelined. After instruction fetching, up to 44 clock cycles may be needed until the results of a floating-point instruction are written into their target registers. Figure 3.5 illustrates the pipeline of an integer ALU. The excessive pipelining helps to achieve higher operating frequencies. The accompanied high latency may lead to sever performance degradation because of pipeline stalls but only in conventional processor architectures. In fact, the FGPU pipeline is hazard-free:

- *No data hazards*: interleaved execution from multiple wavefronts allows executing instructions from other wavefronts until the results of a certain instruction are written into the register files.

- *No control hazards*: branch, jump and return instructions are executed outside the pipeline by the WF scheduler. No PC stalls are expected.

- *No resource hazards*: the pipeline can execute any instruction from any type regardless of the ones that are still passing through the pipeline.

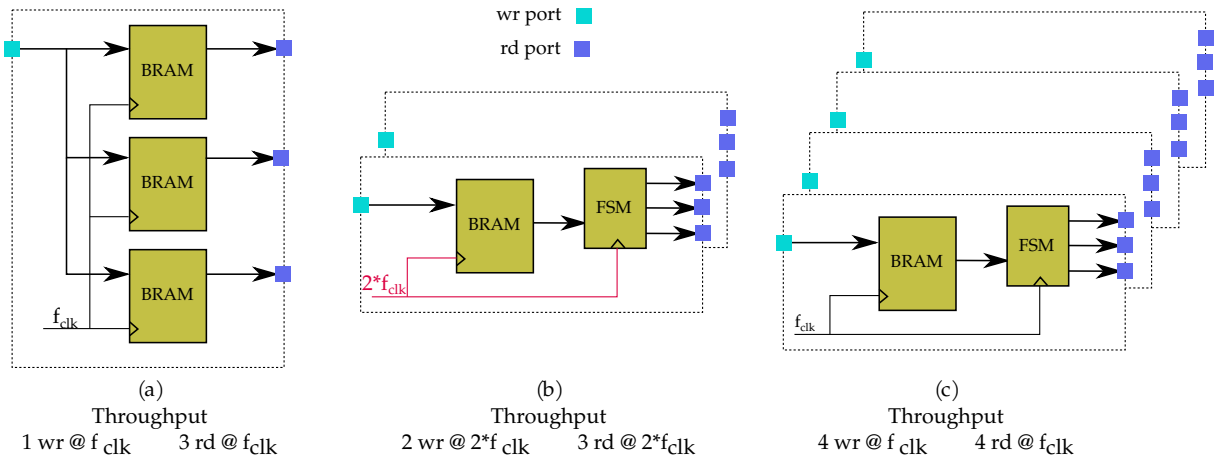


Figure 3.6: Possible implementations of the register files of a single PE using BRAMs

Register Files

Since the WF scheduler issues instructions from 8 wavefronts (512 work-items), a CV should be able to hold their register files on FPGA fabric at the same time and switch between them without any delay. A conventional soft processor architecture uses LUTs and FFs to realize the required registers. However, giving that each work-item owns 32 registers and considering only the storage demands, 8Ki LUTs are needed to hold 512x32 32bit registers per CU[3]. Indeed, this relatively high area overhead is practically unaffordable.

Instead, BRAMs can fulfil such high storage demands. For example, a single block of the type BRAM36E1 [95] can accommodate 1024 registers which correspond to the register files of 32 work-items, or a half wavefront. However, they are dual-ported and a register file needs at least 3 read ports and one write port to support the execution of any arithmetic instruction per clock cycle. Figure 3.6 illustrates three possible configurations that can be used to overcome this problem:

- Using three BRAMs in parallel as shown in Figure 3.6.a solves the bottleneck but trebles the storage demands. Each BRAM stores identical data. Moreover, this makes BRAMs the most critical FPGA resource.

- In Figure 3.6.b, the BRAMs are clocked at double the frequency of other FGPU parts and a Finite State Machine (FSM) is used to latch the read data. It takes 3

[3] We assumed that a LUT with 6-inputs can store 64 bits when used by the RAM32M primitive as 32x2 quad port distributed RAM [93]

clock cycles in the doubled frequency domain $3/(2*f_{clk})$, or two cycles in the normal frequency domain $2/f_{clk}$, to fetch 3 operands out of the read port of a single BRAM. By dividing the required register files on 2 BRAMs, the throughput of 3 read operands per clock cycle $1/f_{clk}$ can be achieved. However, implementing this combination has a negative impact on the maximum operation frequency. We have realized after placing and routing many FGPU cores that the critical paths are formed by signals that connect the normal and the doubled frequency domains.

- In Figure 3.6.c, we removed the doubled frequency domain and used 4 BRAM blocks, each has to provide 3 operands every 4 clock cycles. Although this approach require more BRAMs per PE, it scales better than the one of Figure 3.6.b and it does not waste BRAM space like in Figure 3.6.a[4].

Register files may be written by the results of fixed- or floating-point operations, logical instructions or memory loads from scratchpads or global memory. These operations may be issued from multiple wavefronts and have different or unexpected latencies. Therefore, the WF scheduler has to avoid simultaneous writing of the register files by multiple results from different instructions. To avoid any bottleneck at the write ports of the register files, FGPU uses time-division multiplexing when writing the four ports of the configurations (b) and (c) of Figure 3.6.

Hard Floating-Point with Partial Reconfiguration Support

The floating-point logic supports the 32bit single-precision format of the IEEE 754 standard. It comprises a configurable set of different instructions based on the floating-point IP core from Xilinx [92]. It is fully pipelined, i.e. a new result per clock cycle can be calculated. According to the application needs and the available FPGA area, the developer may selectively implement the desired floating-point operations before synthesis. For each implemented instruction, a hardware instance is created per PE. The same registers can be used for floating-point or fixed-point data. However, the floating-point logic is not realized within the ALU pipeline because of its high latency: it takes 28 clock cycles to compute a division or a square root. Moreover, all floating-point operations are implemented within a separate module inside a CU (see Figure 3.2).

[4] We used the configuration in Figure 3.6.b in the publications in FPGA'16 [7] and FPT'16 [46] but switched later to the one in Figure 3.6.c

This simplifies partially reconfiguring the implemented operations as described later in Section 7.4.

3.4.4 CU Memory Controller

FGPU uses the available ALU logic to calculate the target memory address when executing a load or a store operation. All transaction information including address, data and access type is pushed first into a FIFO buffer. An array of controllers, called *stations*, read this data. Each PE is mapped exclusively to a configurable number of stations. The concurrent processing of multiple memory requests issued from different work-items running on the same PE is very important to the overall performance. Empirical experiments revealed that synthesizing less than 4 stations per PE would lead definitely to a bottleneck.

Memory Operations on Global Memory

Read and write transactions on global memory are gathered from all stations and pushed into a FIFO buffer. Then, the individual requests are forwarded serially to the global memory controller over an AXI-stream interface. The serialization helps to simplify the required logic to prevent issuing identical read requests that access the same cache line. Indeed, this may often happen if the running work-items are reading the same region in the global memory.

In case of write transactions, stations can retire and process new requests directly after writing into the serialization FIFO. In case of a read transaction, the handling station has to listen to the read data. The cache has multiple banks and it serves several data words at once. Indeed, the number of banks at the cache read port should be equal to the number of CUs to avoid any bottleneck. If a reading station detects that the address of the severed cache line matches, the corresponding word is latched. Another controller collects the latched data from reading stations and writes them back into the register files.

Scratchpad Memory

Many algorithms need more than 32 registers to finish all required computations. Call stacks are needed to pass parameters to callees and for register spilling and filling [28]. However, holding call stacks in the global memory would overload its bus and narrow this bottleneck further. In addition, transactions on global memory have natively high latencies.

FGPU instantiates one or more scratchpad memory blocks per PE (see Figure 3.2) to hold call stacks. Load and store operations to scratchpads are realized using dedicated assembly instructions. Moreover, they have different addressing modes to the ones that operate on global memory[5]. They are executed directly on scratchpads without any scheduling logic. Each work-item is mapped to a fixed region in a scratchpad block. The allocated size can be specified at synthesis time according to the application needs. With a single BRAM36E1 [95] per PE, the call stack of each work-item can be extended by 16 entries. All operations on scratchpads are aligned to 32bits.

Sub-integer Memory Operations

Enabling 8- and 16bit loads requires right shifting the loaded word by a different number of bits. Besides, zero filling or sign extension are needed to fill the upper parts of loaded registers. To spare resources, we decided not to implement the shift operations in hardware but as extra instructions in software. The procedure of converting sub-integer loads to native instructions is handled by the compiler. The accompanied penalty in execution time is still affordable. Thanks to the plenty of PEs, the performance bottleneck is usually determined by how often the global memory is being accessed and the access pattern. Actually, cycle-accurate simulations for different fixed-point applications indicate that in case of 64 PEs, loaded with a total of 8Ki work-items, a CU spends on average 17%-83% of the execution time waiting for their

[5] Addressing a scratchpad location can be done using an immediate value and a base address. This is an effective approach when accessing call stack entries based on fixed offsets to the stack pointer. On the other hand, addressing a global memory address is done using register contents, e.g. *ld r1, r2[r3]*. This is more useful when iterating over array elements in for-loops.

memory requests to be served[6]. In case of 8 PEs, this range reduces to 10%-70%. Therefore, the extra instructions accompanied with each sub-integer load are not expected to significantly impair the performance.

In contrast to load implementations, we realized the sub-integer store instructions in hardware. Otherwise, a data word has to be read out to mask the data to be stored. The additional read operation would cause more traffic on the global memory bus, and it would narrow the bottleneck further. If not needed by the running application, sub-integer store support can be disabled at synthesis time to reduce the area footprint.

3.5 Global Memory Controller

FGPU has a central memory controller that can be connected to multiple external memories over a configurable number of 32- or 64bit AXI4 interfaces (see Figure 3.7). It serves multiple CUs and includes a shared direct-mapped data cache.

3.5.1 Cache Organization

The proposed cache design is true dual-ported, i.e. it has two ports that can serve read and write transactions. The bitwidth of both ports is identical and it spans over N 32bit data words. The first port is dedicated for serving transactions issued from the CUs and the other one for the data traffic to global memory. Cache size, line size and bitwidth of both ports can be configured[7]. Cache addresses are mapped directly to the global memory ones. To minimize the data traffic to global memory, a write-back strategy is used [38], i.e. written data get flushed into the global memory only when the corresponding cache line gets deallocated. Therefore, a *Byte Dirty* memory is used to mark which bytes are dirty within each cache line.

[6] A memory model that assumes a random delay distributed uniformly over the range [10,30] clock cycles before serving data has been used

[7] The terms *cache line* and *cache size* will be used interchangeably.

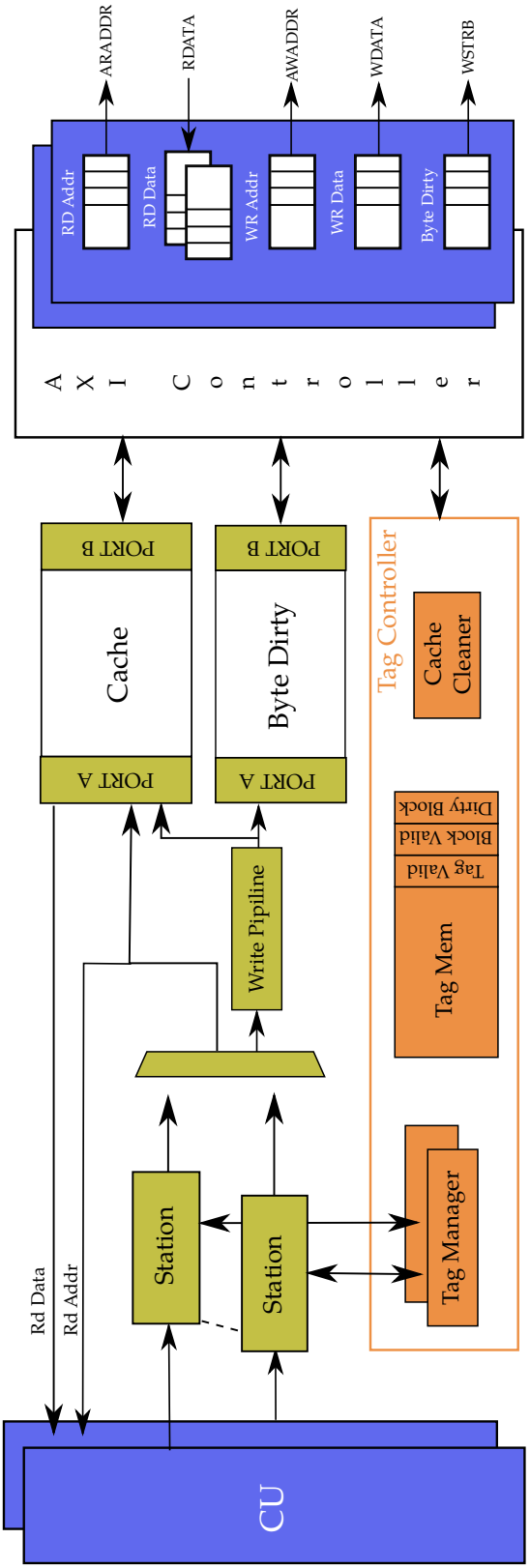


Figure 3.7: Block diagram of the global memory controller without atomics

3.5.2 AXI Controller

It connects the cache system to the global memory through multiple AXI4 interfaces. To overcome the high latency of loading from external memories, the AXI controller can initiate multiple read transactions with different IDs on each connected AXI bus. This allows serving the read data as soon as possible irrespectively of the submission order of the read transactions. All read and write data is transferred in bursts, where the burst size is as big as a cache line. In addition, FGPU integrates multiple FIFOs to temporally buffer data traffic to global memory. The number, as well as the storage capacities of these FIFOs, are enough to guarantee that no back-pressure will be practiced from FGPU on the global memory connection.

3.5.3 Tag Managers

A tag manager is responsible for managing the cache status. It decides when a cache line gets populated with what content and synchronizes the data traffic to global memory. In fact, FGPU uses multiple tag managers. They work collaboratively on editing the content on the following local memories:

- *Tag Valid*: tells whether a cache line has been already allocated.
- *Tag Memory*: stores the address mapping between cache lines and global memory.
- *Block Valid*: determines whether the allocated cache line has been populated with the corresponding content from global memory.
- *Dirty Block*: says if the populated cache line has been written by a CU but not yet flushed to global memory.

Each of these local memories has 4 read ports but a single write port to prevent race conditions, e.g. when multiple tag managers try to update the status of the same cache line. Having less than 4 read ports would create a bottleneck; and more than 4 would significantly increase the area overhead. In fact, these memories will get synthesized using *distributed memory*[8] where all of the corresponding FPGA primitives have a maximum of 4 read ports [93]. Besides, we have recognized that 1-2 tag managers are

[8] Distributed memories are realized by synthesis tools using LUTs, and they do not include any BRAM blocks. They are preferred when implementing small memory blocks or when more than 2 read/write ports are needed.

needed per CU to achieve the best performance. After the execution of all work-items finishes, the *Cache Cleaner* module can be activated to check the whole cache and to write back any dirty data it finds into the global memory (see Figure 3.7).

3.5.4 Handling Memory Requests

An incoming memory request generated by a CU gets received by a free *Station* as illustrated in Figure 3.7. Then, the station checks if the corresponding cache block has been already mapped to a physical location in the global memory (*Tag Valid Bit*), to which location it has been mapped (*Tag Address*) and if the cache block has been populated by data from global memory (*Block Valid Bit*):

- In case of a *Cache Hit*:
  - For write transactions: the write command enters the *Write Pipeline* and the dirty bit gets set for the updated block as well as the corresponding bits in the *Byte Dirty* memory.
  - For read transactions: if the *Block Valid* bit is set, the cache can be read safely. Otherwise, a *Tag Manager* gets signalized to populate the cache block with the corresponding content from global memory.
- In case of a *Cache Miss*: the corresponding station polls a tag manager to allocate the corresponding tag and waits until it is finished.
  - For write transactions: as soon as the tag memory gets updated accordingly, the write command enters the *Write Pipeline*. The corresponding bits in the *Byte Dirty* memory get set. The tag manager does not issue a read transaction to the global memory to fill the cache block. This will be done only for read commands.
  - For read transactions: the tag manager checks if the cache block is dirty. Only in this case, the dirty content will be written back into the global memory and the corresponding *Dirty Bit* gets cleared. The content of the *Byte Dirty* memory enables to selectively write the dirty parts of the cache block. After that, the global memory can be read, and the cache block gets populated with the requested data.

Multiple reads and writes can be executed simultaneously on the cache. Only the critical parts are serialized to avoid inconsistency between the cache system and the

global memory. For example, a single new tag can be allocated every clock cycle. Each outstanding request in the global memory controller is assigned an increased priority as more time is spent on waiting. Reading stations compare the value of the read address port of the cache with their own ones and retire when a match is detected. Since the cache read data port spans over multiple words, many stations can be served at once. If the tag value of a cache block gets modified, all outstanding stations waiting to read from the corresponding block will restart and go to the first step in serving their requests. To avoid race conditions when modifying tag values, every new tag is protected for a minimum time from being deallocated. The protection time should enable the corresponding outstanding stations to finish before they get restarted.

Write requests have to go through the *Write Pipeline*. This pipeline gives the necessary time to announce the address that is going to be written to other stations, check if any one is waiting to write the same cache entry, gather data and perform a combined write at once for as many stations as possible.

When a cache location is read and written by multiple CUs, the requests may be served in any order. Anyway, the OpenCL standard does not enforce constraints on synchronising the memory accesses outside the borders of a CU. Serving a read request may take at least 8 clock cycles. When a cache block has to be cleaned and new data has to be read from global memory, a latency of over 50 clock cycles is typical[9].

3.5.5 Atomic Operations

Atomic operations are optional extensions to FGPU and they are a part of the OpenCL standard [52]. They enable multiple work-items to access the same address and modify it while avoiding unintended overwriting. Figure 3.8 illustrates how atomics are implemented inside the global memory controller. The proposed modular design enables selective implementation of multiple atomic instructions at synthesis time according to the application needs. It targets only 32bit fixed-point operations.

Atomic are handled by the same stations used for normal loads and stores. On each clock cycle, a single atomic request can be executed. If the requested memory address is not processed previously by the corresponding atomic unit, e.g. max unit, a *populate* response is issued to the requesting station. This instructs it to read the address content out of the cache. If necessary, the requested cache line will be populated with content

[9] This number does not take into account the response latency of the external global memory.

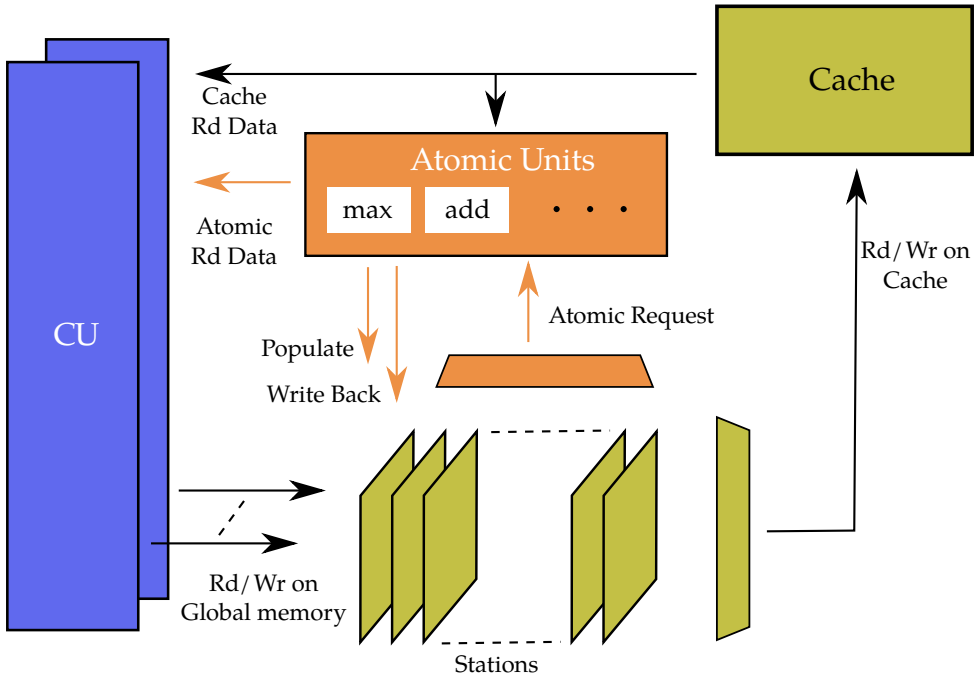


Figure 3.8: Atomic units inside the global memory controller

from global memory. In the meantime, the atomic unit listens to the data read out of the cache and captures it when the read address matches. Then, atomic requests can be served as long as they operate on the same address. On each executed operation, the old content of the corresponding atomic unit is served on a separate bus. According to the OpenCL standard, an atomic operation has to be interpreted as a read of the old address content and a write with the result of the requested operation.

With *write back* requests, atomic units can assign stations to write their dirty contents back into the cache. This can be done right after each executed atomic operation, but frequent write requests will keep many stations busy and hence affect the performance of the cache system. Therefore, each atomic unit has a timeout counter which issues a *write back* when the unit stays idle for an adjustable number of clock cycles.

An atomic unit works on a single address that is not allowed to be shared with other ones. The throughput of the proposed design can reach 1 operation per clock cycle, as long as all atomic operations of a specific type are operating on the same the address. Even if the address gets deallocated from the corresponding cache line, the atomic unit can proceed. The *write back* commands will reallocate the cache line if necessary. Hence, the peak throughput of the atomic units is achievable with ignorable impact on the cache performance.

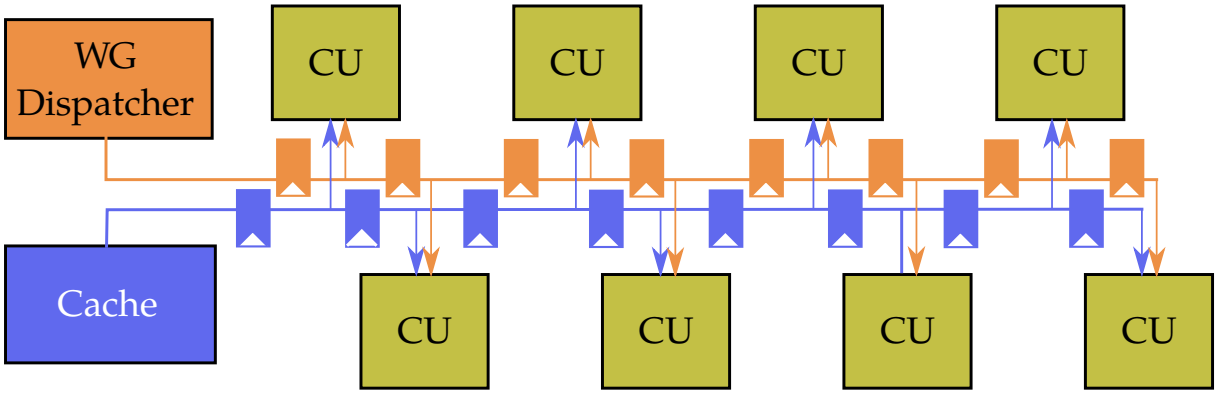


Figure 3.9: Connecting multiple CUs to shared global buses

3.6 Internal Interconnect

FGPU uses the AXI4 standard [16] to communicate with any other external modules. In addition, data communication between the CUs and the internal cache is AXI4-compatible. Instruction fetching is done by scheduling queries from all CUs on a single CRAM instance. When an instruction is served, all CUs listen to the corresponding bus and latch the instruction if needed by any running wavefront. The proposed design can not produce any bottleneck as long as no more than 8CUs are implemented[10]. We could not recognize any performance improvement when using multiple CRAMs.

In FGPU, all CUs are controlled by a single instance of the WG dispatcher. Data traffic to global memory is directed through a central cache system. Therefore, FGPU comprises many control and data buses that have to scan most of the occupied FPGA area. When many CUs are realized and most of the chip resources are consumed, these global buses will form the most critical paths after place and route. Moreover, the high bitwidth of some signals would intensify the problem, e.g. when the cache read port has 8 banks, i.e. 256 bits. To overcome this problem, FGPU uses multiple buffers along the global paths as illustrated in Figure 3.9. For each realized CU, one additional register is added on all critical paths. This allows scalable realizations of many CUs with an adequate reduction in the operating frequency. The downside is that communicating with the farthest CUs would take extra clock cycles. Anyway, we could recognize this effect only when running reduction algorithms on FGPU, where all CUs issue significantly more read than write transactions and the ratio of performed arithmetic operations to memory accesses is low.

[10] Each CU repeats the execution of any instruction over 8 clock cycles.

Table 3.2: Configurable and tested hardware parameters of the FGPU architecture

Module	Parameter	Tested Values
Top	# CUs	1, 2, 4, 8
	# AXI4 data interfaces	1, 2, 4
	Sub-integer store	true or false
	CRAM Size	4, 8, 16 KiB
	Global atomics	any combination
CV	Floating-point Ops.	any combination
CU Mem. Controller	Scratchpad size	0, 8, 16, 32 KiB
	# Stations	$8 * i,\ 2 \leq i \leq 8$
Global Mem. Controller	Cache Size	$4 * i$ KiB, $1 \leq i \leq 8$
	Cache line size	64, 128 Byte
	# Cache Read Banks	1, 2, 4, 8
	# Stations	32, 64
	# Tag Managers	1, 2, 4, 8, 16

3.7 Results

FGPU has been developed completely from scratch and implemented in VHDL-2002. Its current ISA includes 49 instructions and supports the SIMT execution model of OpenCL. However, a full support for the OpenCL standard is out of the scope of this thesis. We used the ZC706 board [100] from Xilinx with the z7045 Zynq SoC as a development platform. The following results are generated using Vivado v2016.2.

3.7.1 Flexibility

We have focused on offering a large space for design exploration with FGPU. Developers can customize the proposed architecture to fit to the available FPGA resources, application needs or the desired performance. Instruction subsetting is supported, e.g. disabling hard floating-point support, atomics or sub-integer store operations, etc. All desired features can be determined before synthesis in a VHDL-package file. Table 3.2 lists the adjustable hardware parameters that have been tested during the work on this thesis.

3.7.2 Operation Frequency and Scalability

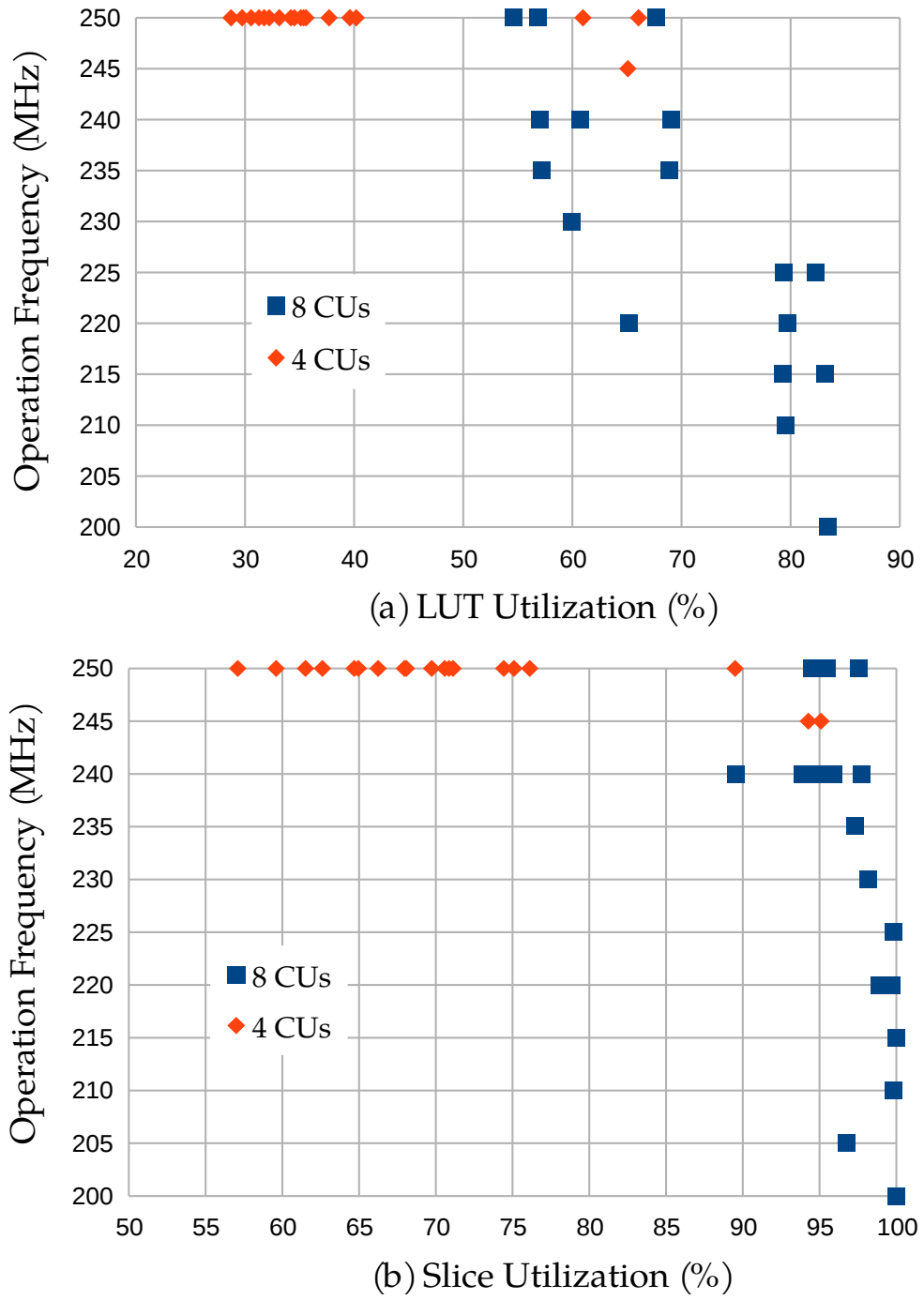


Figure 3.10: Operation frequency vs. LUT and slice utilization of many FGPU realizations

FGPU has relatively a high area overhead: when 8 CUs are implemented with typical settings, there are more than 800 FSMs operating simultaneously on the FPGA fabric. If 2 clock domains are used, where the register files operate at a doubled clock

frequency as other parts (see Figure 3.6.b on Page 41), FGPU can be implemented at 210/420 MHz for the normal and doubled clock domains, respectively. However, this frequency drops rapidly when more than 60% percent of the available LUTs are consumed. The most critical paths are the ones which cross between the clock domains.

FGPUs with single clock domain can be implemented at frequencies up to 250 MHz. This limit is enforced by the external AXI HP interfaces [101], to which FGPU is connected on Zynq. To achieve this frequency and to mitigate the degradation when implementing the bigger cores, we initiated many countermeasures, e.g.:

- Additional registers are automatically inserted on critical paths.
- The fanout of the most critical signals is limited explicitly using HDL-attributes.
- When describing BRAM or DSP primitives in the HDL code, the existing internal and output registers are targeted.

Figure 3.10 illustrates the relationship between the occupied area and the operation frequency of many FGPU cores, where 4 and 8 CUs are implemented and a single clock domain is used. To generate the data of this figure, we varied some of the parameters listed in Table 3.2, e.g. the supported floating-point operations. We used two metrics to estimate the occupied area: the utilization ratios of LUTs and slices. A single slice in the 7 series architecture includes 4 LUTs. Indeed, slices are the most critical FPGA resource when synthesizing FGPU. However, two designs may have the same slice utilization ratio but they consume different numbers of LUTs. In other words, many LUTs may stay unutilized while 100% of the available slices are occupied. Although some of the tested cores have more than 80% LUT utilization and up to 99.98% slice utilization, the accompanied frequency degradation does not exceed 20%. Only when synthesizing 8 CUs and activating many area-consuming features like hard floating-point support, the operation frequency degrades.

3.7.3 Portability

Although the FGPU design targets specific FPGA resources like DSP or byte-enabled true dual-port BRAM blocks, no Intellectual Property (IP)-cores have been used for that purpose[11]. However, a successful and efficient implementation depends primarily on

[11] Floating-point logic is realized using the corresponding IP-core from Xilinx. This is an optional feature and other libraries can be used instead.

Table 3.3: Area requirements for different configurations without hard floating-point support

	LUTs	**FFs**	**BRAMs**	**DSPs**
Available	219K	437K	545	900
8 CUs	119K (55%)	214K (49%)	437 (80%)	256 (28 %)
4 CUs	65K (30%)	112K (26%)	189 (35%)	128 (14 %)
2 CUs	41K (17%)	62K (14%)	105 (19%)	64 (7 %)
1 CUs	30K (14%)	39K (9%)	60 (10%)	32 (4 %)
MicroBlaze	7K(3.2%)	5.6K (1.3%)	27 (5.0%)	6 (0.7%)

the ability of the vendor's CAD tools to infer the targeted blocks from the VHDL code. Although FGPU has been synthesized and tested only for Zynq, it is absolutely possible to synthesize it on other FPGA families from the same vendor without modifying the RTL Code.

Only the floating-point units have to replaced when migrating to FPGAs from other vendors. Fanout limitations are included in the RTL code as signal attributes and they are vendor-dependent. They may have to to be adjusted when targeting other FPGAs to achieve better performance. Anyway, they are optional and they do not affect the portability at all. Since synthesis tools ignore uninterpretable VHDL attributes, having multiple vendor-dependent ones in the same design is possible.

3.7.4 Area

Efficient FGPU cores require a lot of FPGA resources. Moreover, floating-point operations are more area-consuming: a single floating-point divider (793 LUTs) is approximately as big as a complete integer PE (878 LUTs).

Table 3.3 gives an overview of the needed resources when FGPU is synthesized with 1, 2, 4, and 8 CUs without hard floating-point units. The last entry in this table shows the needed resources to implement a MicroBlaze processor with default settings for best performance. All designs are synthesized, placed and routed using default performance-oriented strategies in Vivado. We only considered FGPU cores with a single clock domain. Although BRAMs correspond to the highest utilization ratio, we only considered the number of LUTs to estimate the area demands in the rest of this

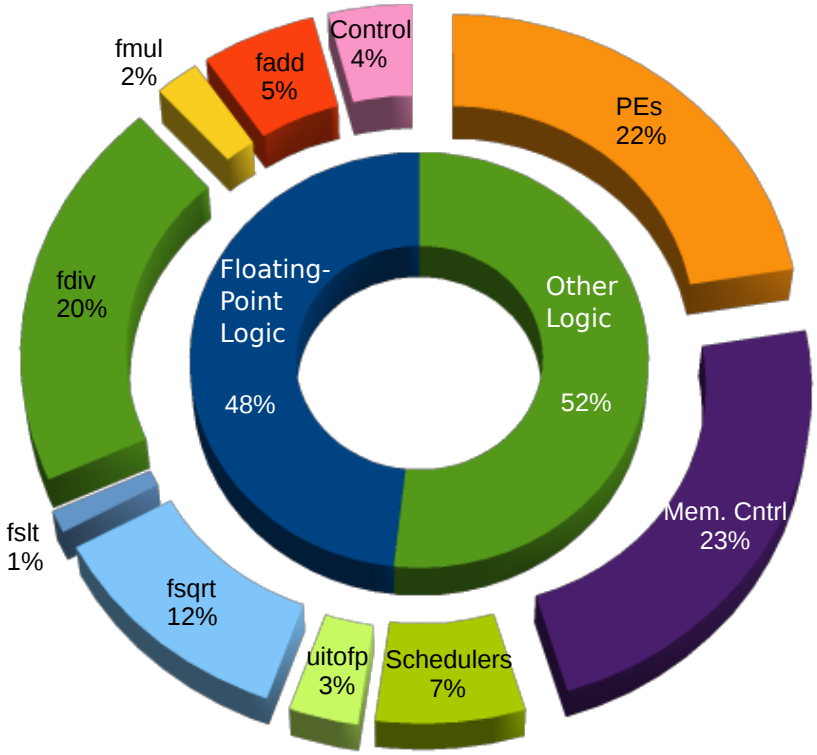


Figure 3.11: Relative resource utilization of the major FGPU modules. Four CUs are implemented with all supported floating-point operations

thesis. However, if more BRAMs are needed, the synthesis tool can use the available LUTs to realize them.

Fig. 3.11 illustrates the relative size of the main modules of an FGPU with 32 PEs and full hard floating-point support. The whole core consumes 125K LUTs. Each PE is mapped exclusively to its hard floating-point hardware, i.e. there are 32 floating-point dividers, multipliers, etc. Approximately, half of the required FPGA area is occupied by the floating-point hardware. When excluded, 44% of the rest is consumed by the memory controllers [12], 43% is used to implement the integer PEs and 7% is needed by scheduling logic (WG dispatcher and WF schedulers). In comparison to the reference MicroBlaze, the FGPU cores we used to generate the results of this thesis were approximately 6-26x bigger.

[12] This includes the global memory controller and the local CU ones

3.8 Summary

FGPU is a soft multi-core processor for general purpose computing on FPGAs. It is a configurable, scalable and portable 32bit GPU with floating-point support. FGPU is implemented completely in RTL and it does not replicate any existing architecture. Its ISA is designed to support the execution of OpenCL kernels. FGPU is a stand-alone processor and it includes its own directed-mapped data cache. The biggest FGPU implementation comprises 8 CUs (64 PEs) running at 250 MHz and it is capable of accommodating 4096 work-items on FPGA fabric and instantly switch between their register files. It supports ISA-subsetting on instructions like atomic and floating-point operations. FGPU supports full thread-divergence. Optional scratchpad memories can be synthesized to implement call stacks.

Chapter 4

Compiler Design

FGPU offers the compute power of FPGAs in the attractive execution model of OpenCL. Therefore, providing a compiler for the proposed tool flow is a main aspect of this thesis. Since a complete support for the OpenCL standard is not affordable, the main objective of the developed compiler has been set to successfully and effectively compile the benchmarks we targeted in Chapters 6 and 7 and to support all FGPU hardware features described in Chapter 3.

This chapter does not present a novel approach in the field of compiler technology. Nevertheless, the scientific contribution from the compiler-perspective is concentrated in handling the specialities of the proposed hardware architecture. Therefore, we will focus on describing the required modifications to the modules of a conventional compiler to support OpenCL-execution on FGPU. Most of the material in this chapter was published in FPT'16 [46].

4.1 Background

4.1.1 Compiler Design with LLVM

Low Level Virtual Machine (LLVM) is a framework for compiler design started at the University of Illinois at Urbana–Champaign [58]. It is open-source and programmed mainly in C++. An LLVM-based compiler comprises three major components:

Listing 4.1: LLVM IR code for the vector multiplication of Listing 4.3

```
1  define void @vec_mul(i32* nocapture readonly %in1, i32* nocapture readonly %in2, i32* nocapture
       %out) {
2  entry:
3    %0 = tail call i32 asm sideeffect "lid $0, $1", "=r,I,~{$1}"(i32 0)
4    %1 = tail call i32 asm sideeffect "wgoff $0, $1", "=r,I,~{$1}"(i32 0)
5    %add.i = add nsw i32 %1, %0
6    %arrayidx = getelementptr inbounds i32, i32* %in1, i32 %add.i
7    %2 = load i32, i32* %arrayidx, align 4
8    %arrayidx1 = getelementptr inbounds i32, i32* %in2, i32 %add.i
9    %3 = load i32, i32* %arrayidx1, align 4
10   %mul = mul nsw i32 %3, %2
11   %arrayidx2 = getelementptr inbounds i32, i32* %out, i32 %add.i
12   store i32 %mul, i32* %arrayidx2, align 4
13   ret void
14 }
```

- *A Frontend* that parses the program code into LLVM Intermediate Representation (IR).
- *An Optimizer* that applies many optimization transformations on the IR code.
- *A Backend* that transforms the IR into the ISA of the target machine.

The modular structure of LLVM simplifies compiler design. To support a new programming language, only a frontend has to be developed while other parts can be reused. Similarly, when a new architecture has to be supported, only a new backend is needed. Therefore, we decided to use LLVM to develop a compiler for FGPU since we need only to develop the backend part. Moreover, LLVM supports many programming languages including OpenCL as well as many processor architectures including AMD and NVIDIA GPUs.

4.1.2 Intermediate Representation (IR)

LLVM IR is a generic, high-level and RISC-like assembly language. It is independent of the programming language and the target machine. IR is produced by the frontend and consumed by the backend. Most of the optimization passes of LLVM run on the IR-code. Unlike machine ISA, IR is strongly-typed and it uses the Static Single Assignemnt (SSA) form, i.e. a variable is set only once. In addition, it supports structs,

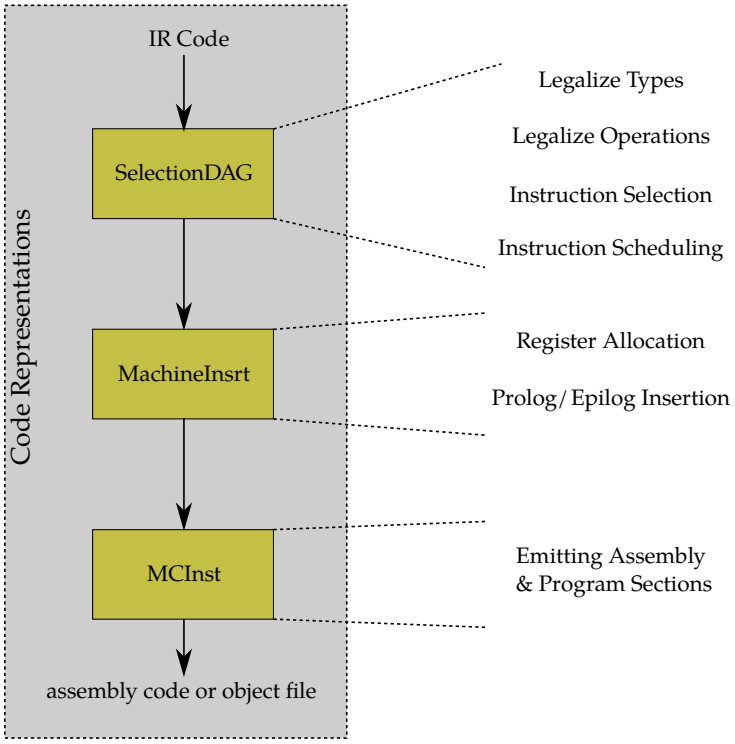


Figure 4.1: Code representations and main transformations in an LLVM-backend

arrays, SIMD instructions and inline assembly. Listing 4.1 illustrates a sample IR-code of an OpenCL kernel.

4.1.3 LLVM Backend Organization

Developing a backend in LLVM can be divided into describing:

- **The target architecture**: e.g. which operations are natively supported, their encoding, the number and the types of available registers, etc. LLVM offers a Domain Specific Language (DSL) called TableGen (.td) for that purpose.
- **The code generation algorithms**: e.g. calling conventions, constraints on register allocation, instruction legalization, etc.

As illustrated in Figure 4.1, a backend operates on three different code representations:

1. *SelectionDAG*: a Directed Acyclic Graph (DAG) generated from the IR code. The vertices are the IR operations and the edges represent the dependencies. LLVM

performs all required legalizations on this graph. For example, operations that are not supported by the target ISA get synthesized using other ones. At the last step, the graph gets destroyed, and its operations get scheduled to be executed in a specific order.

2. *MachineInstr*: an abstract sequential description of legal and untyped machine code generated when destructing the SelectionDAG. LLVM destroys the SSA form of the machine code and allocates the physical registers on this representations.

3. *MCInstr*: a low-level raw representation of the machine instructions at the object file level. Program sections and assembly labels are generated at this layer.

4.1.4 Relevant LLVM Sub-Projects

LLVM includes a collection of sub-projects that offer many reusable open-source tools [3].

clang

Clang is a compiler for C based languages including C++ and OpenCL. In comparison to GCC, it is faster, uses less memory and generates more expressive warning and error messages. In addition, it supports GCC-style inline assembly. Clang can generate machine code for many targets as well as IR code. Currently, it is a part of the official repositories of most Linux distributions.

LLVM Core

The LLVM core comprises an optimizer and a code generator. The optimizer part is independent of the source programming language and it operates on the IR-level. The code generator is generic and can be used as a basis to develop backends for many targets. In addition, it supports the generation of assembly and binary code formats. The official repository supports many target machines like ARM and MIPS.

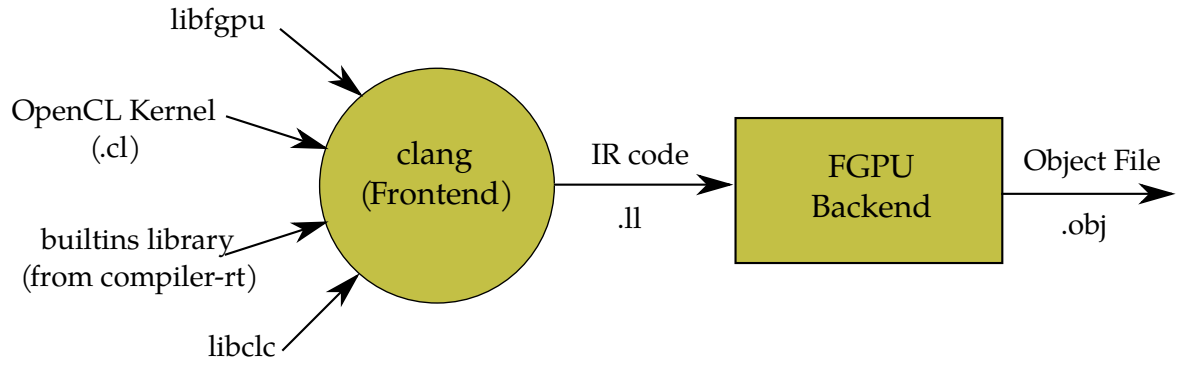


Figure 4.2: FGPU compiler structure

compiler-rt

This project provides low-level implementations for many IR operations in target-independent C or highly-optimized assembly. For example, the function `__floatunsisf` can be used to convert a fixed-point unsigned integer into a single-precision floating-point number if the target machine does not support this operation in hardware. In addition, it includes many runtime libraries for dynamic debugging, e.g. to detect memory leaks.

libclc

Libclc is an open-source implementation of the library requirements of OpenCL. For instance, it defines how to compute OpenCL math functions like `cos()` or `log()`. With this library, any C compiler can compile most of the OpenCL device code.

4.2 Compiler Structure

Figure 4.2 depicts the main components of the developed FGPU compiler. We used Clang as a frontend to convert the OpenCL code into LLVM IR assembly. A successful code compilation for FGPU requires 3 libraries:

1. **libfgpu**: an FGPU-specific library that defines the target-dependent implementations of the OpenCL built-in functions, e.g. `get_global_size()` and atomic operations. This library uses GCC-compliant inline assembly instructions to target the corresponding functionalities in the FGPU-ISA. Clang converts the inlined assembly to canonical IR instructions so that they can be processed later by the

Listing 4.2: FGPU implementation of the OpenCL function get_global_id()

```
1  inline unsigned get_global_id(const unsigned dim){
2      unsigned offset, local_ID;
3      // get the local ID of the calling work-item in dimension dim
4      __asm__ __volatile__("lid %0, %1"
5                           :"=r"(local_ID)
6                           :"I"(dim)
7                           );
8      // get the work-group ID offset of the calling work-item in dimension dim
9      __asm__ __volatile__("wgoff %0, %1"
10                          :"=r"(offset)
11                          :"I"(dim)
12                          );
13     return offset+local_ID;
14 }
```

backend. For instance, Listing 4.2 shows how to implement `get_global_size()` in FGPU-ISA. The corresponding IR instructions can be observed at the Lines 3-5 of Listing 4.1.

2. **builtins (from compiler-rt)**: a runtime library for emulated floating-point arithmetic that is compatible with IEEE-754.

3. **libclc**: currently, it is used only for defining the scalar and vector data types of OpenCL like `uint` or `char8`.

With the aforementioned libraries, the Clang compiler can convert a kernel code written in OpenCL into LLVM IR. Then, a special FGPU-backend translates it into an object file (.elf format) that can be used to program FGPU.

4.3 FGPU Backend

The FGPU backend has been developed based on both the MIPS backend provided in the official LLVM repository [3] and the Cpu0 backend [27]. In many aspects, the developed backend bears a certain resemblance to other official LLVM backends. In this section, we will concentrate only the issues that differentiate the FGPU backend from the other ones.

4.3.1 Instruction Selection

Legalization of Sub-Integer Loads: FGPU supports natively only 32bit load operations. Enabling 8- and 16bit loads requires right shifting the loaded word by 0, 8, 16 or 24 bits. The shift type can be arithmetic, e.g. in case of loading *char* or *short* types, or logical. After shifting, a sign extension or zero filling has to be done. For example, a load byte with sign extension is executed as [1]:

$$lb\ r2,\ r1 \equiv \begin{cases} lw\ r2,\ r1 \\ xor\ r3,\ r1,\ 31 \\ shl\ r3,\ r3,\ 3 \\ shl\ r2,\ r2,\ r3 \\ sra\ r2,\ r2,\ 24 \end{cases} \tag{4.1}$$

Similarly, other types of sub-integer loads can be performed as load word plus a maximum of 4 extra instructions. If some alignment information is available, further optimizations are possible. For example, in case of unsigned byte load, where the address is 4-bytes aligned, a single masking operation with the value 0xFF is enough. This often happens when loading vector sub-integer variables, e.g. *char4* where four signed bytes has to be read out and stored in 4 registers. The backend emits one load word command instead of four. This reduces the pressure on the memory controllers. In addition, because the alignments of the bytes to be loaded are known a priori, the four signed bytes can be stored in their target registers with 7 extra instructions instead of 16.

Legalization of Conditional Select Instructions: FGPU supports conditional set instructions in a MIPS-like flavour: only one type of conditional set is natively supported, namely *set if less than (slt)*. Four other variations of *slt* are offered to deal with unsigned numbers and immediate values: *sltu*, *slti* and *sltiu*. Two conditional move operations as supported: *move on zero (movz)* and *move on non-zero (movn)*. The backend realizes all variations of conditional select and set commands described in LLVM IR using the aforementioned instructions. For example, *set if less than or equal* is implemented as:

$$slte\ r3,\ r2,\ r1 \equiv \begin{cases} slt\ r4,\ r1,\ r2 \\ xor\ r2,\ r4,\ 1 \end{cases} \tag{4.2}$$

[1] The target register is given first in each assembly instruction (MIPS notation)

Listing 4.3: OpenCL kernel for vector multiplication

```
1 __kernel void vec_mul(__global int *in1, __global int *in2, __global int *out) {
2     int index = get_global_id(0);
3     out[index] = in1[index] * in2[index];
4 }
```

The same trick is applied for branches. Two branch instructions are supported natively: *branch if two registers are equal (beq) or if not equal (bne)*. More sophisticated ones, e.g. *branch if less than*, are expanded to a *beq* or *bne* with a conditional set. Indeed, the conditional set and move operations enable realizing simple if-else or ternary-based assignments without branching.

Conditional set on floating-point numbers is realized similarly. Only *floating-point set if less than (fslt)* can be supported in hardware. The developer may select one of two options to realize all types of other conditional instructions:

- If *fslt* is synthesized in hardware, the compiler will compute the desired conditional set instruction using a combination of logical ones and *fslt*.

- Otherwise, it calls the corresponding software implementation from a standard library.

Further optimizations: The ISA of FGPU has some instructions that can substitute multiple ones, e.g. *multiply and accumulate (MACC)* and the address calculation of load and store commands. These instructions are generated after the legalization pass by scanning against pre-defined matching patterns. For example, a *load word (LW)* instruction in the form $lw\ r_3,\ r_2[r_1]$ loads the register r_3 with the content of the address $r_2 + 4 * r_1$. This enables loading the *n*-th element from an integer or float array *a*, i.e. *a[n]*, with a single assembly instruction. In addition, the benefit of this technique can be observed in the FGPU assembly code of Listing 4.4 which has been compiled from Listing 4.3. Indeed, the OpenCL code at line 3 from Listing 4.3 is translated into 4 instructions at lines 7-10 in Listing 4.4. A MIPS compiler needs 8 instructions to perform this task.

Listing 4.4: FGPU machine code for the vector multiplication kernel of Listing 4.1 or Listing 4.3

```
1  lp    r2, 2          ; load 3rd paramter (output array)
2  lp    r3, 1          ; load 2nd paramter (2nd input array)
3  lp    r4, 0          ; load 1st paramter (1st input array)
4  lid   r5, 0          ; load local ID in dimension 0
5  wgoff r6, 0          ; load worg-group ID offset in dimension 0
6  add   r1, r6, r5     ; calculate work-item global ID
7  lw    r4, r4[r1]     ; load first operand
8  lw    r3, r3[r1]     ; load second operand
9  mul   r3, r3, r4     ; compute result
10 sw    r3, r2[r1]     ; store result
11 ret                  ; return
```

4.3.2 Call Conventions

Call stacks are used for passing call parameters and getting return results from callees. When the 32 private registers are not sufficient, call stacks are used for register spilling and filling. FGPU holds all call stacks in local scratchpads. They are accessed with different machine instructions than normal loads and stores on global memory. Frame pointers of running work-items are not held explicitly in physical registers. The stack pointer of a running work-item is the last register (R31) of its register file. We have used similar calling conventions to the ones of the MIPS processor: the first four parameters as well as the returned value are passed on registers (R1-R4); more parameters can be passed on the stack. This is expected to be faster and reduces the maximum required stack size in most cases.

However, when a work-item starts execution, it cannot get its parameters from a call stack as in most other architectures. Instead, the *load parameter (lp)* native command has be used to copy each parameter from the RTM memory into the corresponding register file. Determining how to get the call parameters of a function is decided upon the metadata at the IR level: if the function owns the `__kernel` qualifier, `lp` instructions will be used. Otherwise, stack operations will be generated.

Prologue and Epilogue: The prologue sequence is executed at the beginning a callee. In the FGPU backend, it allocates the needed stack space by decrementing the stack pointer and pushes the register values that have to be preserved across function calls on the stack [83]. Since linking a kernel function is different to linking other ones,

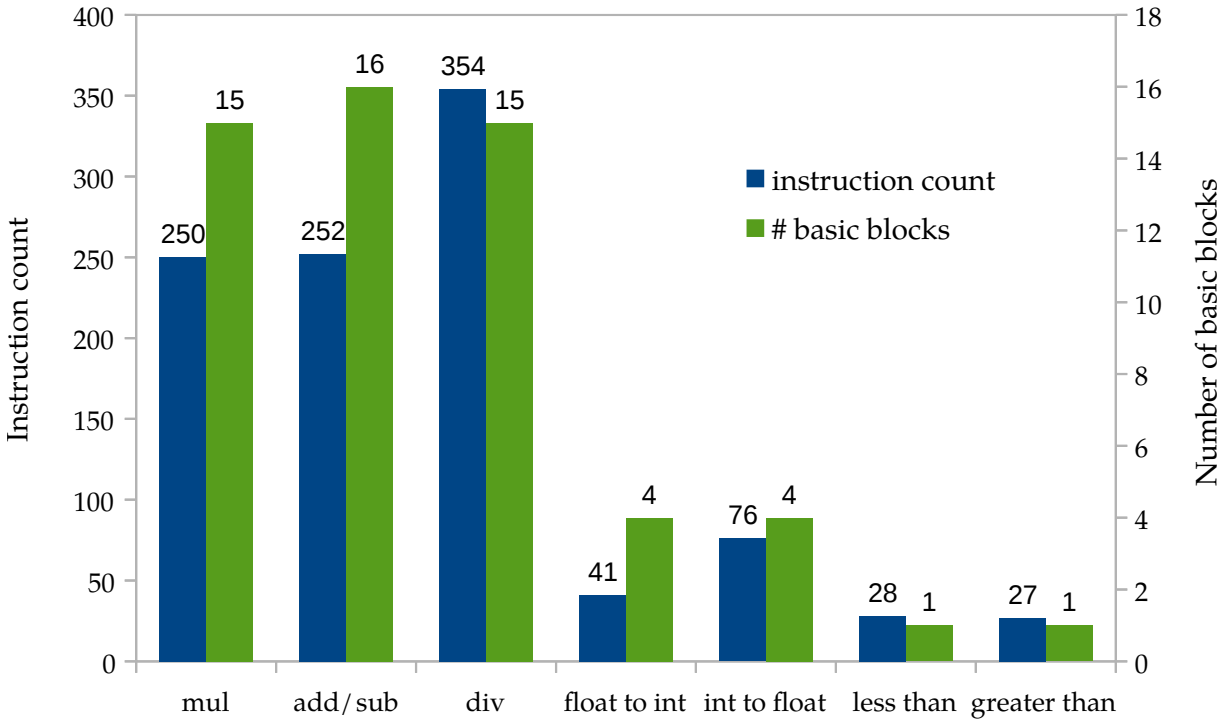


Figure 4.3: Instruction count and number of basic blocks for the emulated floating-point instructions on FGPU

the backend generates different prologue code for each case[2]. In fact, preserving register values is not necessary when a kernel starts execution. Therefore, this part of the prologue can be omitted from kernel calls.

The epilogue sequence increments the stack pointer and restores the callee-saved registers. Similarly, the restore part does not need to be executed when exiting kernel execution.

4.4 Emulated Floating-Point Arithmetic

We have used the builtins library from the compiler-rt LLVM sub-project as a basis for our implementation. Only single-precision IEEE-754 compatible floating-point arithmetic has been targeted. However, software implementations of floating-point operations use conditional code intensively. Therefore, we modified the existing implementations for faster execution on FGPU but without affecting their functionalities. For instance, the execution of if-else code blocks is delayed to later points if possible, or

[2] A kernel function represents the entry point for execution on FGPU. In other words, it can be called from outside FGPU. Non-kernel functions can be called from any code running on FGPU.

both if- and else-parts are executed if the estimated penalty is less that of a divergence point in the corresponding CFG.

If the required floating-point operation is supported in hardware, the compiler uses an assembly instruction to implement it. Otherwise, the corresponding standard function is called, e.g. *__mulsf3* for multiplication. Hard floating-point comparison instructions illustrate a special case: only the *set if less than (fslt)* variant is supported in hardware. The compiler realizes all other comparison types using *fslt* and some extra logical operations. Configuring the compiler to generate emulated or hard operations is done using an attribute when calling the compiler executable.

Figure 4.3 presents the instruction count and the number of basic blocks after compiling some basic floating-point operations. In contrast to normal processors, the penalty of emulated against hard floating-point in execution time is not expected to be proportional to the instruction count. This fact is confirmed by results of Section 6.4.4. Moreover, the excessive branching in the CFGs of emulated floating-point operations has the dominant effect in determining the performance penalty in comparison to hard floating-point ones.

4.5 Conclusion

An LLVM-based OpenCL compiler for FGPU has been developed. It uses `clang` as a frontend and a specially developed backend. It supports the basics of the OpenCL standard (version 1.1) and can be extended for full support without redesigning the presented parts. The same OpenCL code developed for normal GPUs can be compiled and executed on FGPU. The developed backend includes an inline assembler and a disassembler and it can be configured to use emulated or hard floating-point operations.

Chapter 5

Tool Flow

The thesis does not aim only to develop another efficient processor which no other tool can program, but also to encourage the scientific community to use soft GPGPU by providing the required CAD software. Since the underlying hardware architecture is designed specially for FPGA synthesis and it does not replicate any other GPU, developing its own tool flow has been done approximately from scratch. The first section of this chapter describes the functional and cycle-accurate simulation platforms we have developed and used for tuning many hardware parameters and testing and debugging more than 13000 VHDL code lines of the FGPU hardware. Indeed, FGPU has been developed according to the extreme programming principle in software development, i.e. a new feature is first defined, implemented and tested intensively before integrating the next one. Therefore, the focus will be the test and verification aspects of the developed cycle-accurate simulator. Later, two APIs that may be used to control and program FGPU from a Linux operating system are described.

5.1 Simulation Platforms

FGPU is an accelerator whose functionality can be configured in software. Although it can perform the assigned tasks without any support from external hardware, there should be another processor that controls FGPU. From this point of view, the OpenCL standard distinguishes between the host, usually a CPU, and the device, or equivalently FGPU. Hence, a task is performed collaboratively by executing its host

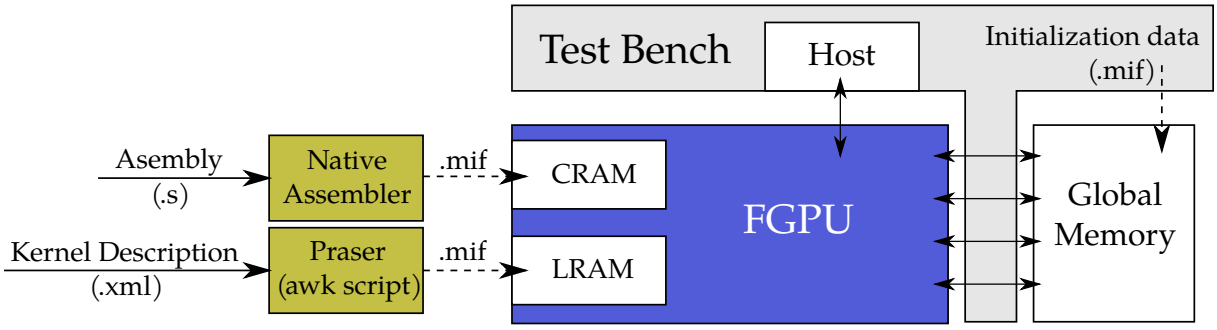


Figure 5.1: The main modules of the cycle-accurate simulator

code (in C/C++, Java, etc.) and device code (in OpenCL). Therefore, the simulation environment has not only to run a piece of binary code on the FGPU in an RTL-simulator, but also it has to emulate the host behaviour.

5.1.1 Cycle-accurate Simulation

In addition to the RTL code of FGPU, the following models and tools have been developed to simulate the execution of any kernel (see Figure 5.1):

- A cycle-accurate model of a multi-ported global memory.
- A host which configures and instructs FGPU.
- A native assembler that converts FGPU instructions from plain text into binary format.
- A parser to write the LRAM memory. The initialization data defines how to run kernels but can be modified at runtime by the host.

The global, as well as the LRAM and CRAM memories, get initialized before a simulation begins via automatically generated *.mif* files. Their binary contents are defined by user input given as human-readable assembly instructions (for CRAM), an XML file (for LRAM). In addition, some pre-given VHDL procedures can be used to generate specific test data to initialize the global memory. Moreover, the developed simulation platform offers some pre-made templates to facilitate the process of creating and running new tests in a cycle-accurate environment. They help to:

- Modify the content of the global memory with test patterns for multiple applications.

- Modify the LRAM content during simulation.
- Count and check the integrity of all read and write transaction to define the location and the time stamp of potential bugs.
- Collect statistical information to identify bottlenecks and optimize the hardware parameters, e.g. the occupancy of the compute vectors or the number of cache misses, etc.

The simulation platform has been written completely in VHDL using standard libraries, and it can be used in any simulator from any vendor[1] . It can manage the execution of up to 16 test kernels, mapped to the 16 LRAM-slots. Before the execution begins, the global memory gets initialized with relevant data and the expected results are computed. Then, a kernel is launched with the desired settings. During execution, the test bench checks the correctness of every write transaction by comparing against the precomputed golden data. When the execution finishes, the amount of written data is checked. The simulator reports any error it detects and breaks immediately. The test bench, which simulates the host behaviour, can modify the LRAM-content online, e.g. to scan multiple problem sizes or to change a parameter's value. This is required for some applications like bitonic sorting and FFT where the same piece of code has to be iteratively executed with different parameters. In the following subsections we will describe the tools that have been developed to facilitate the use of this simulation environment.

Native Assembler

Testing some special instruction combinations and applications written in manual assembly is often necessary to verify certain features of the FGPU hardware. However, writing in binary code is a very tedious task. We developed a lightweight native assembler which transfers programs written in human-readable assembly into binary coded instructions. It is a stand-alone program written in C++, and it supports the basic features of an assembler like using labels for branch instructions. It generates a binary program in a *.mif* file that can be used directly to initialize the CRAM at the

[1] When floating-point operations are supported in hardware, the simulation models of the corresponding IP cores from Xilinx are needed. However, the process of compiling and adding them to the simulation library has been automated using tcl scripts for ModelSim and QuestaSim simulators.

Listing 5.1: A sample LRAM slot of a matrix multiplication kernel described in XML

```
<kernel>
    <index>9</index>                          <!-- index of LRAM slot to be written (0 to 15)
        -->
    <nDim>2</nDim>                            <!-- number of dimension in index space (1 to 3)
        -->
    <start_addr>333</start_addr>              <!-- address of first instruction in CRAM --
        >
    <global_size0>10</global_size0>           <!-- size of index space in dimension 0 -->
    <global_size1>1024</global_size1>         <!-- size of index space in dimension 1-->
    <global_offset0>0</global_offset0>        <!-- offset of index space in dimension 0 -->
    <global_offset1>0</global_offset1>        <!-- offset of index space in dimension 1 -->
    <work_group_size0>10</work_group_size0>   <!-- size of a work-group in dimension 0 -->
    <work_group_size1>10</work_group_size1>   <!-- size of a work-group in dimension 1 -->
    <nParameters>3</nParameters>              <!-- number of kernel parameters -->
    <param0>0x10000000</param0>               <!-- value of parameter 0 -->
    <param1>0x10000000</param1>               <!-- value of parameter 1 -->
    <param2>0x10080000</param2>               <!-- value of parameter 2 -->
</kernel>
```

beginning of a simulation. During the last stages of this thesis, a more professional assembler has been developed within the LLVM environment.

XML as an API for Cycle-Accurate Simulations

LRAM content provides the information on how to run a kernel, e.g. the number of work-items to be launched. The standard OpenCL-API offers procedures to define this information, where each one may fill multiple entries in the LRAM (see Appendix B.1). Nevertheless, programming the standard API calls for each new application is more time-consuming than filling the LRAM-table. To avoid manually calculating the binary content of LRAM as well as using the standard API-calls, we developed an alternative XML-based format. An *awk* script parses the XML-input into a *.mif* file that initializes LRAM before a simulation begins. Each piece of information can be given in decimal or hexadecimal in a separate XML field as shown in Listing 5.1.

Global Memory Model

Integrating the global memory into the simulation test bench is not only necessary to check the correctness of the data produced by FGPU, but also to define potential bottlenecks and tune the performance parameters like the ones listed in Table 3.2 on Page 52. Therefore, not only a functional but also a realistic timing model for the global memory has been developed. It supports:

- Four 32- or 64-bit AXI4 interfaces: all ports operate simultaneously and they can be connected to the four FGPU data ports.
- AXI transaction identifiers: they are integers used to mark read or write transactions in the AXI4 protocol[2]. They help to multiplex multiple virtual AXI channels on a physical one. This feature is essential to improve the data throughput on AXI connections. Indeed, FGPU contains multiple modules, or *Tag Managers* as called in Section 3.5.3, that are authorized to issue data traffic commands. Each module uses an exclusive identifier. The AXI protocol allows serving read commands in any order as long as they have different identifiers. In addition, a Tag Manager is informed when its read or write transaction is performed by matching the read data identifier or the write response identifier with its own one, respectively.
- Burst transfer mode on both read and write channels: interleaved read transfers can be used, i.e. a read burst can be interrupted by other ones running on other virtual AXI4 channels and resumed later. Besides, read or write transactions can be configured to pause at random times for a configurable duration.
- Configurable response time: the response time on read, write or response channels can be fixed at specific values or configured to take a random delay in the range [x,y], where x and y can are configurable as well. This time corresponds to the inevitable physical delay when reading and writing data from global memory.

Many of the features listed above, e.g. interleaved read transactions or pausing burst transfers, simulate real and very common events that may happen during the execution on real hardware platforms. Indeed, if they have not been considered in our model, many bugs that had been detected in real tests could not have been reproduced in cycle-accurate simulations.

[2] On Zynq, a transaction identifier is 6 bits wide

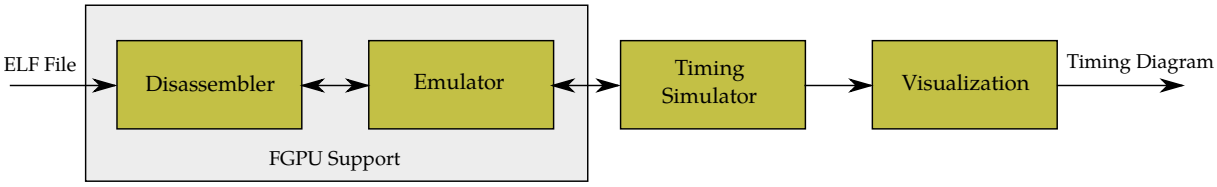


Figure 5.2: Simulation Paradigm of Multi2Sim

How to setup a cycle-accurate simulation

1. Define the hardware features of the FGPU core, e.g. number of CUs.
2. Compile the OpenCL code or use the native assembler to generate the binary code to be executed on FGPU. A *.mif* will be generated and it can be used to initialize CRAM.
3. Define linkage information in an XML-file (see Listing 5.1). A *.mif* will be generated and it can be used to initialize LRAM.
4. Use the pre-given VHDL methods to start and control FGPU during simulation.

5.1.2 Functional Simulation

Cycle-accurate simulations at the RTL-level are very time-consuming. For instance, it would take multiple minutes to simulate a single millisecond on an FGPU core with 8 CUs using a normal PC. Since many commercial and open-source simulators for multiple GPU architectures are already available, we decided to setup a model for FGPU in an existing open-source simulator instead of developing an FGPU-dedicated one.

A survey on the existing GPGPU simulators identified GPGPU-sim [15] and Multi2Sim [85] to be the best candidates. Because the latter has a better documentation and it supports many architectures, e.g. AMD Southern Islands and NVIDIA Fermi, we decided to use Multi2Sim as a basis for our functional simulator.

Multi2Sim Simulator

Multi2Sim supports the simulation of heterogeneous platforms consisting of a host (CPU) and a device (GPU), where the host may run an operating system and the device is controlled through a standard OpenCL-API. Hence, both the host and the device

programs have to compiled and executed interactively by the simulator. For simplicity, we used the "application-only" variant of Muti2Sim which allows the host and the device to communicate without the OS-layer in between.

To integrate a new architecture into Muti2Sim, the following tasks have to be done:

- **On the host side:** a special driver to communicate with the device is needed. It provides the required methods to execute the OpenCL-API calls on the host, e.g. launching a kernel on the device.

- **On the device side:** Figure 5.2 illustrates the main components of the Multi2Sim device simulator [19]. First, the binary code is loaded and converted by the *Disassembler* into an organized data structure that gets executed by the *Emulator*. The *Emulator* launches the work-items and modifies the state of global memory and the register files after executing each instruction. If the timing details are desired, it communicates with the *Timing Simulator* that tracks the state of the individual device parts like cache memory and pipeline stages. The visualization tool can be used to display all events during a simulation.

Since only a functional simulation with Muti2Sim is desired, no timing simulator has been developed. Our FGPU-Multi2Sim model includes the following components:

- **The Host driver:** to execute the API calls on the host side. Providing a complete support for the OpenCL-API can not be achieved within the time frame of this work. However, the implemented driver provides the needed methods to run many 1D and 2D kernels like FIR and matrix multiplication.

- **The Disassembler:** any FGPU instruction is 32 bits wide and it is stored in little endian. There are 8 formats for FGPU-instructions as shown in Figure 5.3. According to the 8 upper MSB bits, the instruction structure can be uniquely identified[3]. The disassembler takes an ELF object file as input. For the `.text` section, it iterates in all instructions and extracts their comprising fields. At the end, each instruction will be converted into an appropriate `struct` that can be processed by the emulator.

[3] The 8 MSBs comprises a family code (4 bits) and operation code (4 bits). There are 13 instruction families; each targets a specific hardware module, e.g. fixed-point ALU, global memory or atomic units. Each family comprises up to 16 instructions.

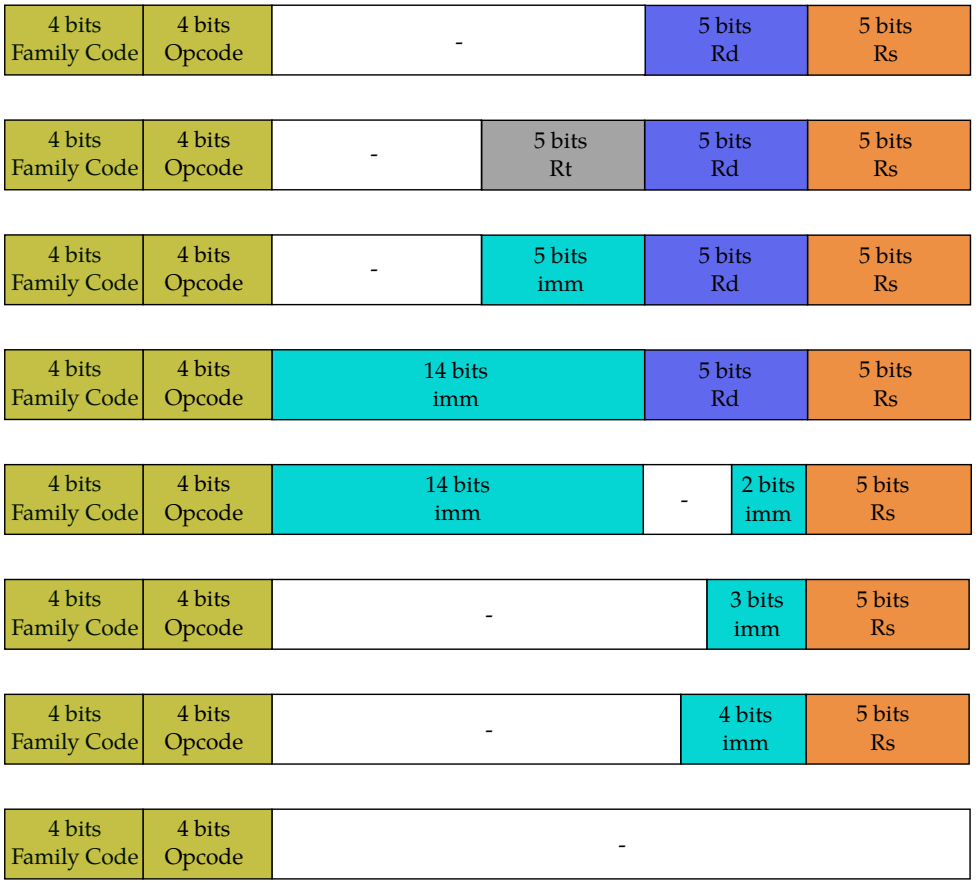


Figure 5.3: Bit fields of an FGPU instruction for the disassembler

- **The Emulator:** before the simulation starts, it creates the lists of work-groups, wavefronts, and work-items to be executed. Then, it processes these lists by simulating the instruction execution in each. The emulator keeps an image of the global memory and the register files of all work-items under processing. The order in which the work-items are executed may not correspond to the one on a real FGPU. However, the final states of the global memory in simulation and on real hardware must not differ.

How to setup a functional simulation

Assuming that the OpenCL code and the characteristics of the index space are already given:

1. Compile the OpenCL code has to generate the ELF file.

2. Write the host code to setup and control FGPU during the simulation in C. Only standard OpenCL-API functions are needed. In other words, the same host code that simulates or executes the kernel on any other GPU can be reused for FGPU with no modifications.

3. Start the simulation. The user can observe the contents of the global memory or the register files during and after the simulation ends.

Since no timing details are required, the functional simulator does not need any information about the number of available CUs, cache size or the number of data AXI interfaces.

5.2 Application Programming Interface (API)

FGPU has to be configured from an external component, usually another microprocessor. Modern SoCs from FPGA vendors are very convenient platforms to develop and test FGPU. They include a Processing System (PS), which can be used to control FGPU, next to Programmable Logic (PL). Mostly, the PS part comprises a multi-core ARM processor like a quad-core Cortex-A53 on Stratix 10 [10] or a dual-core Cortex-A9 on Zynq [101]. Hardware accelerators, like FGPU, are deployed on the PL and usually controlled by the PS. In many applications, system resources are managed by an OS running on the PS side. Therefore, an API is needed to control FGPU from any application running on the PS side.

In the terminology of the OpenCL standard, we consider the ARM processor to play the role of the *host*, and the FGPU the one of a *device*. To control FGPU, the required API has to:

1. Download the `.text` of the object file into the CRAM.

2. Implement the OpenCL-API methods to define the content of LRAM and control registers.

5.2.1 Linux Driver Support

Controlling FGPU from the PS side requires reading and writing fixed physical addresses. However, the privileges for such operations under an operating system

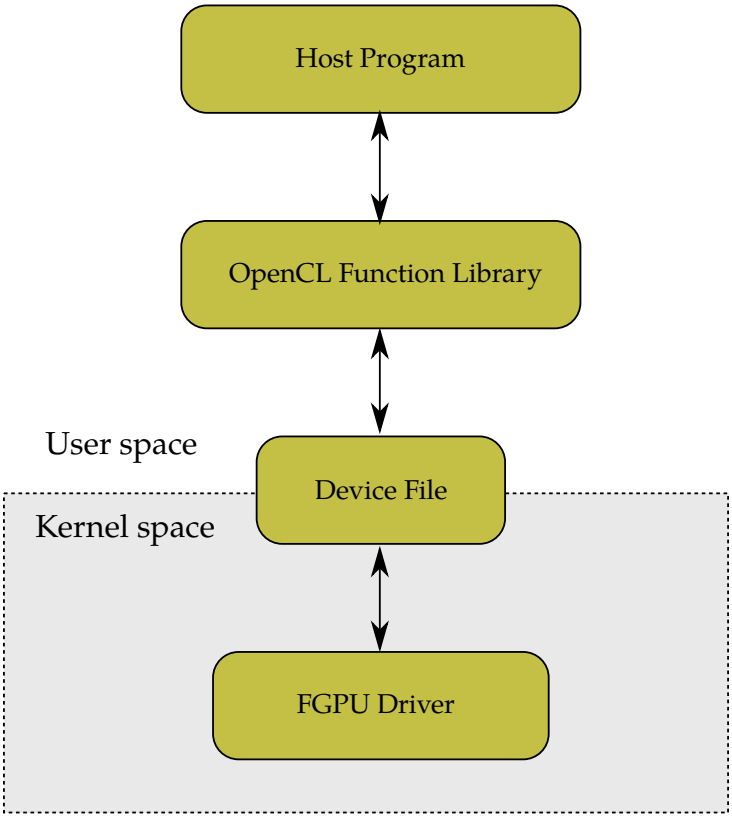


Figure 5.4: Programming FGPU through an OpenCL-standard API and a Linux driver

can not granted to any running piece of code. Under Linux, a process can be either launched in *kernel* or *user* modes. For security reasons, only the ones in the kernel mode can execute privileged instructions. The only way to offer privileged operations to normal users is to use system calls.

Each running process is mapped to a virtual address space, where all its memory objects are located. The real physical addresses of these objects remain unknown to the process itself and the mapping between virtual and physical spaces is managed by the OS. Moreover, a contiguous region in the virtual address space may be mapped to multiple physically separated ones. Indeed, writing a physical address from a running user application requires either:

- Using the `mmap` system call to map the corresponding physical address to a virtual one in the address space of the calling process.

- Developing an OS-driver that provides the required functionality through system calls.

The first approach is impractical because it requires root user privileges when running an application that uses FGPU. In addition, the current design of FGPU can serve only one process and hence it should be controlled at the OS-level through a single central module. Moreover, many methods of the OpenCL-API can not be implemented efficiently with `mmap`, e.g. allocating contiguous shared memory that will be accessed from both the FGPU hardware and applications running on the PS. Hence, we decided to follow the second approach and develop an FGPU-driver for Linux.

Figure 5.4 illustrates the main software modules needed to program FGPU via the developed driver. In the user space, the host application has only to use functions and data types of a standard OpenCL-API to communicate with FGPU. A library that partially implements the API has been developed: it translates the OpenCL calls issued by host applications into system calls and direct them to the driver through a char device. In Linux, this type of devices behaves like a pipe [29]. In the kernel space, the developed driver module has to be loaded in advance to serve the FGPU system calls. It maps the LRAM, CRAM and control registers of FGPU into its address space. In addition, it holds and manages the memory allocated by the host program, e.g. OpenCL buffers, and ensures cache coherency.

GPU manufacturers offer their compute machines as PCIe cards that can be plugged into servers or workstations. The global memory on these card is implemented as Graphics Double Data Rate (GDDR) memory, and it is physically separated from the host main memory. Therefore, the OpenCL standard offers the possibility to allocate memory on the host or on the device side. The main bottleneck for many GPGPU applications is the global memory bandwidth, especially when data has to be transferred over the PCIe-bus between the host and the device memories. On our Zynq platform, host and device memories are physically identical. If the host application allocates memory using the standard `clCreateBuffer` function, it can get a pointer to the allocated region and read/write it directly in user space. The same address range can be accessed and modified by FGPU. No copy operations within the global memory are needed, and hence the traditional bottleneck between the host and device memories does not exist.

The developed driver does not offer complete support for OpenCL-API (version 1.1). However, the current version is capable of running all the benchmarks covered in Chapter 6 and Chapter 7.

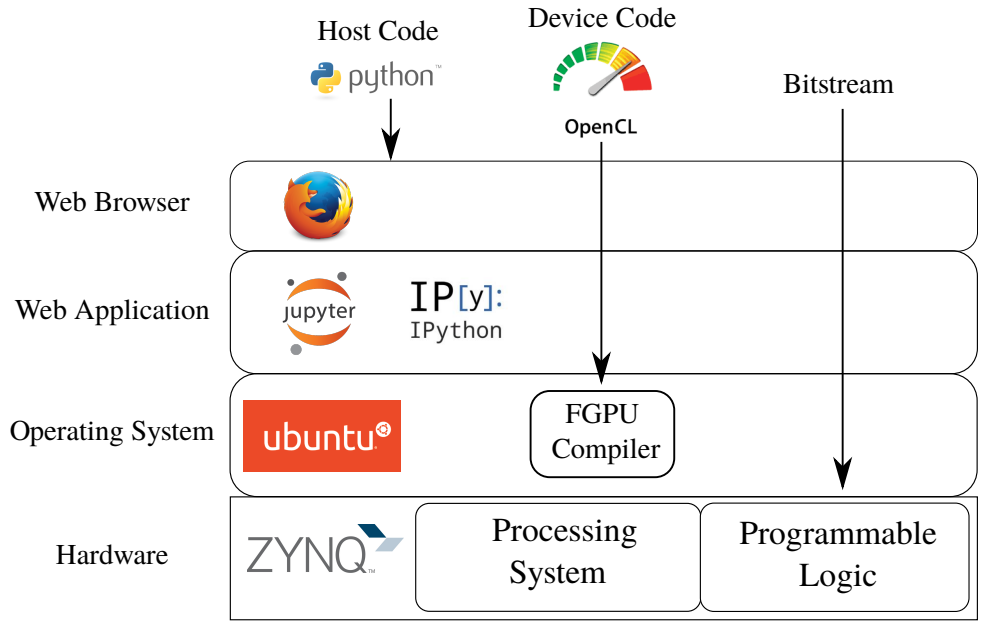


Figure 5.5: The FGPU-PYNQ tool flow

5.2.2 PYNQ-based Web Programming Interface

PYNQ is a recent open-source project from Xilinx to program Zynq SoCs with IPython (Interactive Python) [2]. The Python code runs in a Jupyter Notebook, which is a browser based interactive computing environment. To setup PYNQ, a Linux OS should run on the PS and the notebook server should be installed. System users can login to the notebook server via any web browser to create and run their applications. In addition, the PYNQ project contains many useful Python libraries, e.g. to program the PL with bitstreams and to allocate contiguous memory regions that can be shared between the PS and the accelerators that run on the PL. At the time of writing this thesis, only the PYNQ-Z1 board is officially supported. We have modified the source files to port PYNQ onto the ZC706 board which holds a bigger FPGA that we need to implement the largest and most efficient FGPU cores.

Figure 5.5 illustrates the suggested tool flow. Through the Jupyter web interface the developer can:

- **Program the host application.** It controls and configures FGPU. An open-source Python class has been developed for this purpose. It offers similar methods to the ones included by the standard OpenCL-API.

- **Program the FGPU bitstream.** The FGPU Python class offers a method to program the PL with any bitstream located within the Linux file system. Overwriting a previously programmed FGPU or programming a partial bitstream, e.g. to change the supported floating-point operations, is possible. This feature is realized through the `xdevcfg` driver as described in Section 7.4.1.

- **Write the OpenCL code.** The FGPU compiler has been installed under the Linux OS. Users can load their OpenCL code in the notebook webpage from a local file, edit it and then save the changes. The developed FGPU Python class offers methods to compile the OpenCL code and download the generated binary into the CRAM. The compilation gets done on-board without any communication with an external server.

The proposed solution does not affect the execution time on FGPU and it can easily visualize the processed data like images and videos. In addition, we believe that the flexibility of this tool flow exceeds the traditional one of normal GPUs. Moreover, developers need only to setup a web browser to use FGPU. On the other hand, PYNQ still has a serious security problem because the notebook server has to be launched with root privileges. A sample notebook is shown in Figure 5.6. It does exactly what the well-known `memcopy` C-function is capable of.

Kernel Code

```
%%writefile   OpenCL_code/copy.cl

#include"FGPUlib.c"

__kernelvoid copy_word(__global int *in, __global int *out) {
    int index = get_global_id(0);
    out[index] = in[index];
}
```

Create Objects

```
from FGPU import FGPU
from FGPU import xlnk

fgpu= FGPU()
mem=xlnk.xlnk() #This object is needed to manage memory operations
```

Program Hardware

```
fgpu.set_bitFile("bitstreams/V2_8CUs_250MHz.bit")
fgpu.download_bitstream()
```

Compile Kernel

```
fgpu.set_kernel_file("OpenCL_code/copy.cl")
fgpu.compile_kernel(True)
```

Allocate Memory

```
length = 256*1024 #length of source and destination buffers
src = mem.cma_alloc(length, data_type = "unsigned") #source buffer
dst = mem.cma_alloc(length, data_type = "unsigned") #destination buffer
mem.cma_stats()
```

Initialize Memory

```
#initialize the source and destination buffers
for  i  in  range(0, length):
    src[i] = i
    dst[i] = 0
```

Configure Kernel

```
#bind allocated memory to kernel parameters
fgpu.set_paramerter(0, src, mem)
fgpu.set_paramerter(1, dst, mem)
#setup index space
fgpu.set_num_dimensions(1)
fgpu.set_size(length)
fgpu.set_work_group_size(64)
fgpu.set_offset(0)
```

Execute on FGPU

```
#download the kernel's binary code and its settings to hardware
fgpu.download_kernel()
#execute and wait untill finish
execTime = fgpu.execute_kernel()
print("Execution time =", int(execTime*1000000), "us")
```

```
Execution time = 1359 us
```

Execute with memcopy

```
importtime
#allocate destination
dst2 = mem.cma_alloc(length, data_type = "unsigned")
#execute and measure time
start = time.time()
mem.cma_memcopy(dst2, src, length*4)
end = time.time()
print("Execution time =", int((end-start)*1000000), "us")
```

```
Execution time = 28879 us
```

Figure 5.6: A simple notebook example in Python that operates like a DMA. It copies a region of a given size within the global memory. The notebook executes this task w/o FGPU and measures the transfer time in each case.

Chapter 6

Benchmarking

The ultimate objective of any overlay architecture is to shift the development effort from the hardware to the software domain while trying to minimize the accompanied loss in area, throughput and power efficiency. FGPU is an another overlay whose performance is limited by the one of HDL designs. In comparison to other solutions, it offers a better separation between hardware customization and software development. This accelerates the developments process and enables more compact, portable and generic software implementations that can run on many FGPU cores. The standardized programming language and Application Programming Interface (API) enable other developers, e.g. GPU programmers, to get a faster start in the FGPU tool flow. Creating and programming efficient FGPU cores does not require more than a basic understanding of the programming language and the application demands.

In this chapter, we will compare the performance of FGPU to the ones of a soft processor (MicroBlaze), a soft vector coprocessor and a hard processor (ARM Cortex A-9) supported by a hard vector coprocessor (NEON). All these solutions can be programmed with a similar effort[1]. In addition, we will generalize our results to consider the case of soft homogeneous MPSoCs.

6.1 Experimental Platform

The development board ZC706 has been used to generate all the results of this thesis [100]. The Zynq-SoC z7045 is integrated on this board next to a 1GB SDRAM, which

[1] A *soft* processor runs of the FPGA logic while a *hard* one is realized directly on silicon.

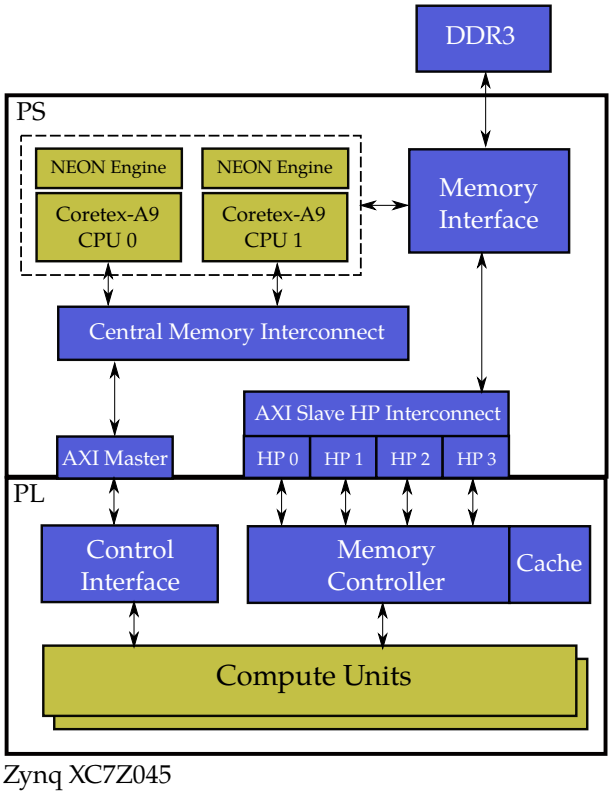


Figure 6.1: The hardware development platform

we consider as a global memory (see Figure 6.1). The data traffic to global memory takes place over the four AXI HP (High Performance) ports of the ARM. We have used different FGPU cores clocked at frequencies between 200-250MHz. A single MicroBlaze processor and the embedded ARM Cortex-A9 have been used to evaluate FGPU.

- The MicroBlaze is implemented in the PL. It is optimized for best performance with hard floating-point support and operates at 180MHz.

- The ARM is supported by the 128bit SIMD NEON engine capable of vector single-precision floating-point computation. The NEON engine is utilized through the automatic instruction vectorization done by the compiler. Both ARM and NEON are clocked at 667MHz.

The MicroBlaze and ARM processors are programmed in C++ by equivalent implementations to the OpenCL benchmarks running on FGPU. All implementations on all processors are compiled for best performance at the optimization level `-O3`. All reported time measurements include partial or complete cache flushing at the end of

execution. This applies for the FGPU cache as well as for the MicroBlaze and ARM ones. The cache content has been invalidated each time before the execution begins.

The on-chip ARM Cortex-A9 includes two cores. While the MicroBlaze or the FGPU is being evaluated, the first core is used to control the design under test while the second one is performing the power measurement. The Texas Instruments UCD90120A power-supply sequencer and monitor is used to measure voltage and current values of all supply rails of the PL of the Zynq chip. The inter-chip communication is realized using the PMBus protocol. Because it takes 1.7ms to sample a voltage/current pair, we have set the task size at a minimum of 256Ki during power measurements and repeated the task execution to take at least 500 samples for each measurement point. All reported power values are averaged over the whole amount of taken samples. The two ARM cores synchronize the measurement procedure with FGPU task execution over a semaphore located in the DDR memory.

We have fixed the number of PEs within a CU in all measurements at 8. The CRAM size is set at 16 or 32 KB. The realized LRAM can hold the settings of 16 kernels and hence its size is set at 4 KB. By setting the i^{th} bit in the start control register, the kernel at index i in LRAM will be launched. When it finishes, the corresponding bit in the finish register gets set. The execution time on FGPU is measured between setting the start register and reading the corresponding value from the finish register. Each time measurement has been repeated 10 times. Since it may take hours to execute a benchmark at the biggest investigated problem sizes, the execution is not repeated if 2 seconds have been elapsed. The reported execution times are the arithmetic mean values over all repetitions.

6.2 Benchmarks

Table 6.1 lists the benchmarks used for evaluation along with the types of the required floating-point operations. Most of these benchmarks are included in the standard OpenCL benchmarks for AMD GPUs [11]. Some are used for both fixed- and floating-point computations, e.g. *FIR* and *bitonic*. To avoid any confusion, any figure or table in the rest of this thesis deals exclusively either with floating-point arithmetic (published in ISVLSI'17 [47]) or with fixed-point (published in FPT'16 [46]). The

Table 6.1: Considered benchmarks with the required floating-point operations

Benchmark	Description	Floating-point operations
copymem	copy a region within the global memory	
vec_mul	vector multiplication of two arrays	*
vec_add	vector addition of two arrays	+
FIR (12 tap)	FIR filter with 12 taps	+,*
bitonic	bitonic sorting	compare
mat_mul	matrix multiplication	+, *
xcorr	sliding dot product of two arrays	+, *
LU-decompose	LU factorization of a square matrix	-,*, /
vec_add	vector addition of two arrays	+
FFT	radix-2 Fast Fourier Transform	+,-,*
parallel selection	sorting by reading the whole array then doing a single write per element	compare
sobel	3x3 Sobel filter for image edge detection (Amplitude only)	+,*,$\sqrt{\ }$,uitofp[1]
compass	3x3 compass edge detection on 8 directions (Amplitude and angle)	
median	3x3 median image filter	
sharpen	5x5 sharpen image filter	
max	finds the maximum number in an array	compare
ac_power	subtracts a value from an array and accumulates the squares of the results	-, +,*
div	divides the elements of an array by a given value	/
N-body	3D motion simulation of a group of objects under mutual gravity forces	+, -, *, /, $\sqrt{\ }$
Floyd-Warshall	finds the shortest paths between all pairs of vertices in a weighted graph	+, compare

[1] uitofp converts an unsigned integer into a single floating-point number

fixed-point results are generated with FGPU cores that use 2 clock domains while the floating-point ones are measured using FGPUs with a single clock domain[2].

The correctness and completeness of all processing outcomes of FGPU have always been checked after execution. This is done by executing the same task on the same input data on the ARM processor and comparing the results with the FGPU ones. The input data is randomly generated with the `rand()` C/C++ function. We have often noticed that merging the code of any two benchmarks and using switch-structures leads to measurable performance degradations. Therefore, each benchmark has been executed using a dedicated function. In addition, to achieve the best performance on FGPU, the work-group size is at any multiple of 64 when possible[3], i.e. an integer number of wavefronts. Using other values implies deactivating some work-items in some wavefronts. For example, the PEs will stay idle half of the time if the work-group size is set at 32. To compile the benchmarks for ARM and MicroBlaze, the compilers delivered with the Vivado Design Suit (v2015.4 and v2016.2) have been used. Indeed, the recorded execution times varied slightly for some benchmarks from one version to the other. In this chapter, we will consider these results as published in the aforementioned conferences.

6.3 Performance

6.3.1 Problem Size

To effectively fill the deep pipelines of FGPU, a minimum problem size is required to achieve the best efficiency. Figure 6.2 illustrates how the execution time changes when varying the problem size from 64 to 1Ki. The increase in execution time does not correspond to the benchmark complexity for problems smaller than 4Ki. In fact, this is the number of work-items that can be processed simultaneously by the FGPU core we used to plot Figure 6.2. Moreover, even for a problem size of 1, we could not record an execution time less than 3us. Most of this duration is spent on initializing the on-chip memories at the beginning and cleaning dirty cache content at the end of execution.

Hence, measuring the speedup when using FGPU over other architectures depends strongly on the problem size. Figure 6.3 illustrates this relationship for an FGPU with

[2] The main difference between the two variants is the organization of the register files. More information can be obtained from Section 3.4.3 on Page 41.

[3] The upper bound of the work-group size is 512

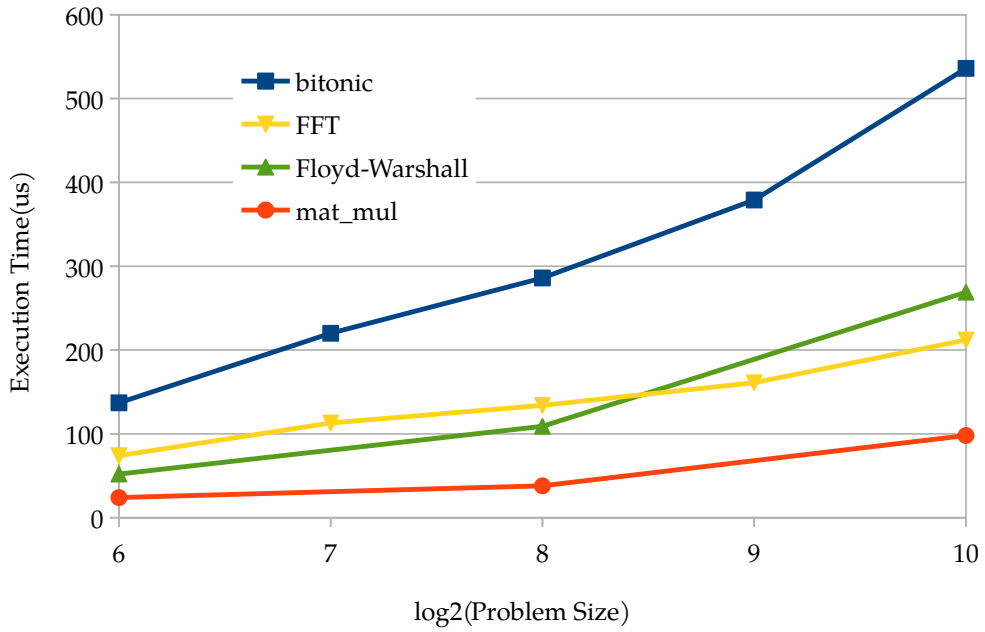


Figure 6.2: Execution time of selected benchmarks on 8CUs with increasing problem size and hard floating-point support

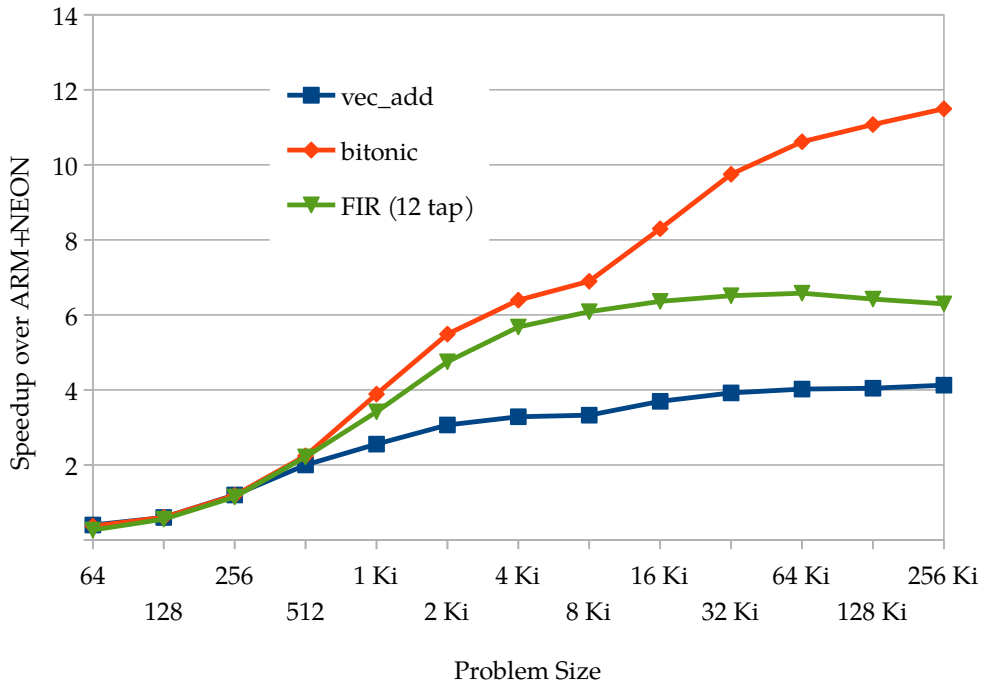


Figure 6.3: Wall clock time speedup for 8 CUs over ARM+NEON implementation for variable problem size and hard floating-point support

8 CUs. Overall, FGPU is as fast as the ARM+NEON combination at a problem size of 256. The more data has to be processed, the better FGPU performs. As a rule of

thumb, each CU needs at least 512 work-items to avoid being underutilized. In the following sections we will concentrate more on the averaged speedup values instead of minimum or maximum ones. The average is computed as the geometric mean on problem sizes from 1-256Ki.

6.3.2 Fixed-Point Arithmetic

Figure 6.4 depicts the wall clock speedup of FGPU over equivalent ARM+NEON implementations for the fixed-point benchmarks when 8CUs are synthesized. Overall, FGPU achieves a speedup of 3.9x. A maximum speedup of 13x was achieved when calculating the maximum of an array using atomic operations. The minimum speedup value of 1.1x was noticed when executing *xcorr*. In this benchmark, each work-item scans the whole input vectors to compute a single output value. Since the pace of execution in each running work-item differs and the FGPU cache is direct-mapped, the frequency of cache misses rises[4]. Similar problems with the memory access patterns can be observed for other benchmarks like *sharpen*. Improving the speedup in such situations requires increasing the cache associativity which would lead to remarkable area overheads.

Figure 6.5 shows the average speedup over the MicroBlaze for different number of CUs. The FGPU cores achieve average speedups of 10x-62x. The performance improves steadily as more CUs are implemented. All values are averaged over problem sizes form 1-256Ki. We often observed speedups which are greater than the number of existing PEs. For example, when having 8 CUs (64 PEs), 223x speedup in case of *mat_mul* and 148x in case of *parallel selection*. This is due to the efficient access patterns to the global memory generated by the FGPU. Its general memory controller can combine multiple read and write transactions from multiple work-items and shape burst transfers out of them. In addition, the multi-bank cache read port can serve up to 8 requests on each clock cycle.

The kernel *copymem* operates like a DMA and helps to measure the efficiency of the memory controllers. An FGPU with 8 CUs is capable of copying 1 MiB of data within the global memory at 1.0GB/s. The throughput of a DMA controller implemented in the PL and running at 100MHz with default settings does not exceed 400MB/s [91].

[4] The ARM Cortex-A9 has a relatively sophisticated 2 level cache system, where L1 and L2 are 4- and 8-way set-associative, respectively. Therefore, the relative performance gap between FGPU and ARM will shrink with increasing rate of cache misses.

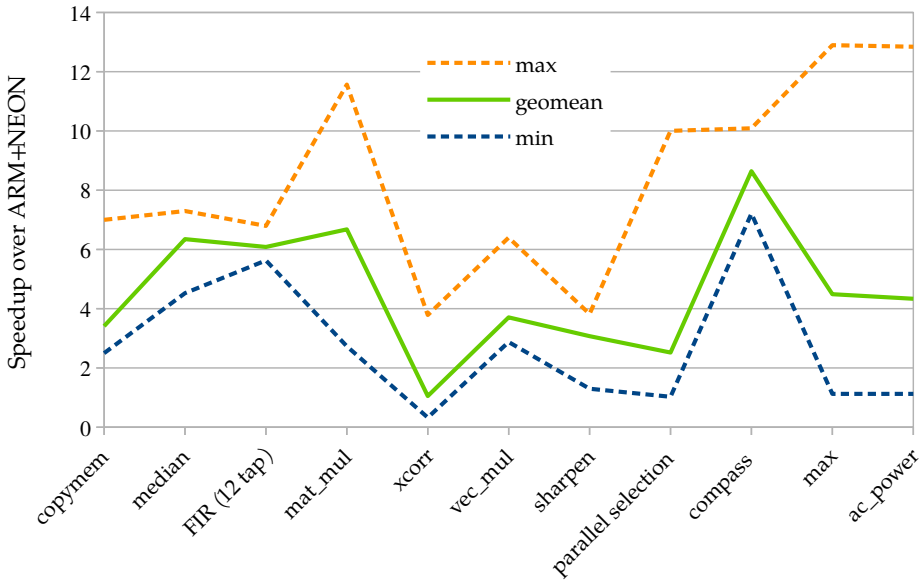


Figure 6.4: Wall clock speedup for 8 CUs over ARM+NEON on problem sizes ranging from 1-256Ki for fixed-point benchmarks

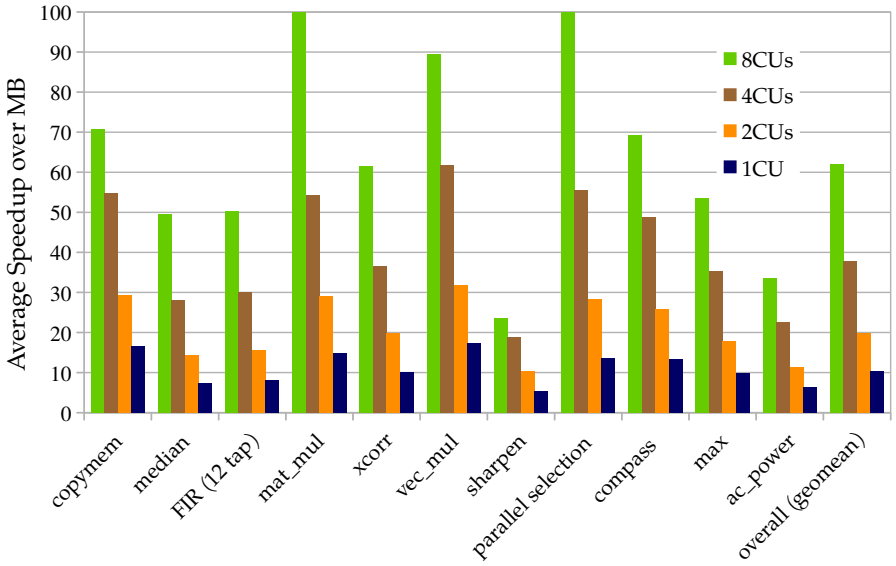


Figure 6.5: Wall clock speedup over MicroBlaze for different implementations averaged on problem sizes from 1-256Ki for fixed-point benchmarks

To evaluate FGPU against a soft vector processor, we considered MXP as a reference (see Section 2.2.3) [79]. The official realization for the development board ZC706 has been downloaded and used from the MXP repository. The corresponding MXP includes 32 lanes and runs at 100MHz. Unfortunately, we could neither regenerate the

bitstream to get the area and power estimations nor get this data from the MXP developers [1]. Therefore, a complete and meaningful comparison with FGPU was not possible. However, we modified the time measurements in the existing MXP benchmarks to include the delay of the memory transfers. Overall, we found that an FGPU with 8CUs is about 1.5-3x faster than MXP for the applications vector addition/multiplication, median and FIR filtering. These numbers have been measured when the bitwidth of processed data is 32bit and the problem size is greater than 1Ki.

6.3.3 Floating-Point Arithmetic

Figure 6.6 illustrates the range of recorded speedups for all floating-point benchmarks over the MicroBlaze. Overall, an average value of 60x is achieved. Speedups up to 480x for *FFT* and 280x for *N-body* are observed. The worst results are measured for reduction problems like *max* and *ac_power*. This is because of the strict synchronization constraints required for parallelizing these tasks since all work-items have to cooperate on computing a single value. In the fixed-point versions of such reduction problems, the performance can be remarkably enhanced using atomic operations. However, since a floating-point instruction requires multiple clock cycles to finish, atomic floating-point operations are not expected to be an efficient solution. Moreover, the OpenCL standard does not consider any atomic floating-point arithmetic.

A comparison to the ARM and NEON combination is shown in Figure 6.7. Overall, an averaged speedup of 4.2x is recorded over all benchmarks and problem sizes from 1Ki to 256Ki. The maximum value is 22.6x for the *N-body* motion simulation recorded at a problem size of 8Ki.

6.4 FGPU Synthesis Parameters

FGPU can be customized according to the application needs or to fit to the available FPGA area. Here we will investigate the effects of changing its main parameters.

6.4.1 Number of CUs

Figure 6.8 illustrates how the speedup of FGPU over MicroBlaze evolves when implementing more CUs. It is directly proportional to the number of CUs if no more than

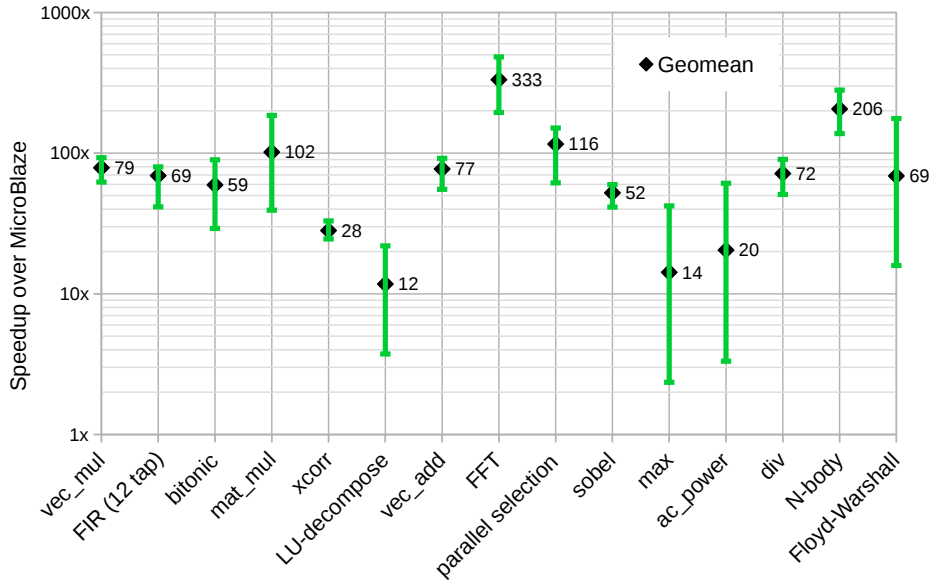


Figure 6.6: Average wall clock speedup over MicroBlaze and its variation when changing the problem size from 1Ki to 256Ki with hard floating-point support

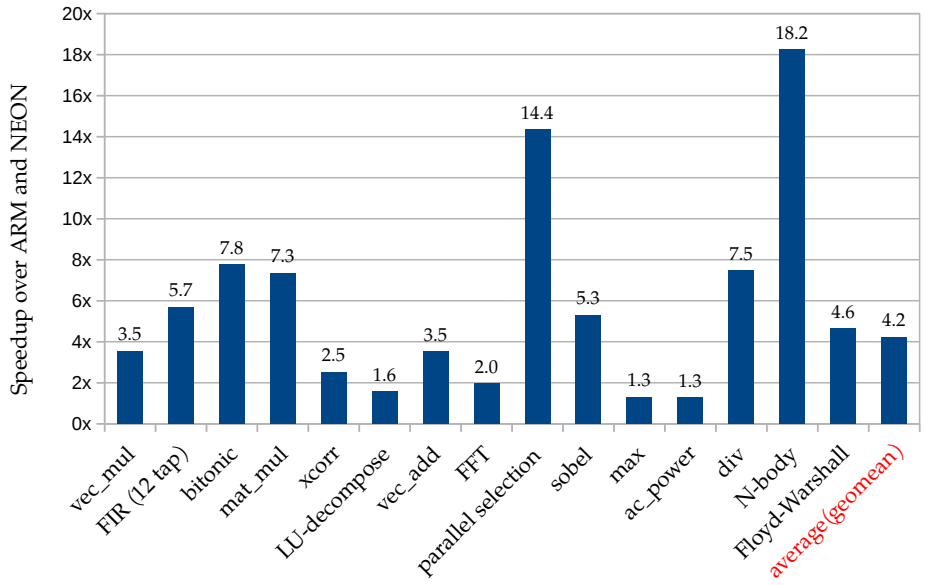


Figure 6.7: Average wall clock speedup over ARM+NEON when varying the problem size from 1Ki to 256Ki with hard floating-point support

4 CUs are synthesized. Because of the global memory bottleneck and since each benchmark accesses the global memory with a different pattern, this linear dependency may not hold for some benchmarks when more CUs are implemented. For example, by doubling the number of CUs from 4 to 8, the average speedup improves by a ratio of

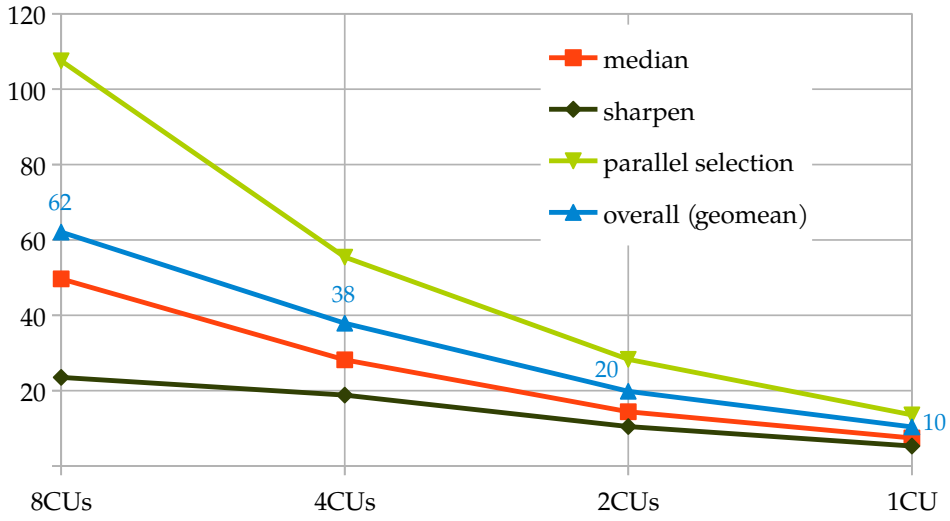


Figure 6.8: Wall clock speedup over MicroBlaze with different number of CUs averaged on problem sizes from 1 to 256Ki for fixed-point applications

24/19=1.26x for *sharpen* and a ratio of 108/55=1.96x for *parallel selection*. However, we noticed that using more CUs helps always to reduce the execution time for any of our benchmarks.

6.4.2 Sub-Integer Arithmetic

The 32bit integer benchmarks have been modified to operate on 8- and 16bit signed or unsigned integers. The internal computations inside the kernel remained unaffected on 32bit. The results are truncated to the corresponding sub-integer type at the end of execution. The modified benchmarks have been executed on the 8CUs variation of FGPU and on the MicroBlaze. Figure 6.9 shows the relative change in speedup in comparison to the 32bit integer case.

On average, the achieved speedups by GPGPU computing degraded by 11% and 28% for the 16- and 8bit benchmarks, respectively. This is due to the reduced efficiency of the local and global memory controllers in the soft GPU. In fact, a GPU architecture can generate more traffic on the global memory bus than any other single processor can do: while some work-items are waiting for their data to be served, others are doing their computations. As the bitwidth of the processed data reduces, issuing new memory requests gets less often and the latency of processing them becomes more

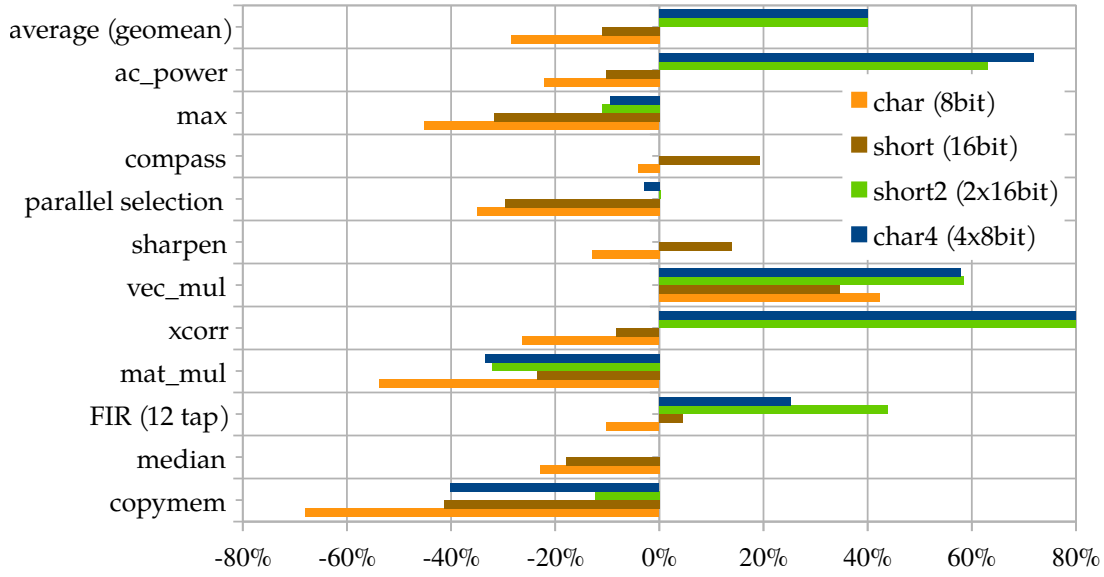


Figure 6.9: Relative change in average speedup of the sub-integer kernels in comparison to the integer ones. All speedups are computed over MicroBlaze

critical to the overall performance[5]. Hence, the performance gab between FGPU and MicroBlaze gets smaller for sub-integer computations.

An efficient workaround is using standard vector types in OpenCL [52]. For example, when a kernel processes a *char4* value instead of a *char* one, 4 pieces of data can be loaded with a single load operation. Therefore, the ratio of executed arithmetic operations to issued load instructions increases. The associated performance boost can be observed in Figure 6.9. However, it might be not not easy to apply this workaround in some applications, e.g. for image filters. On average, 40% improvement was achieved over the 32bit benchmark when *short2* or *char4* types have been used.

6.4.3 Atomics Performance

Using atomics may not only simplify programming in OpenCL but also improves the performance of GPGPU computing. Figure 6.10 illustrates the execution time of the kernel *max* with and without using atomics.

[5] On FGPU, serving a memory read request takes about 50 clock cycles in case of a cache hit and at least 150 in case of a cache miss

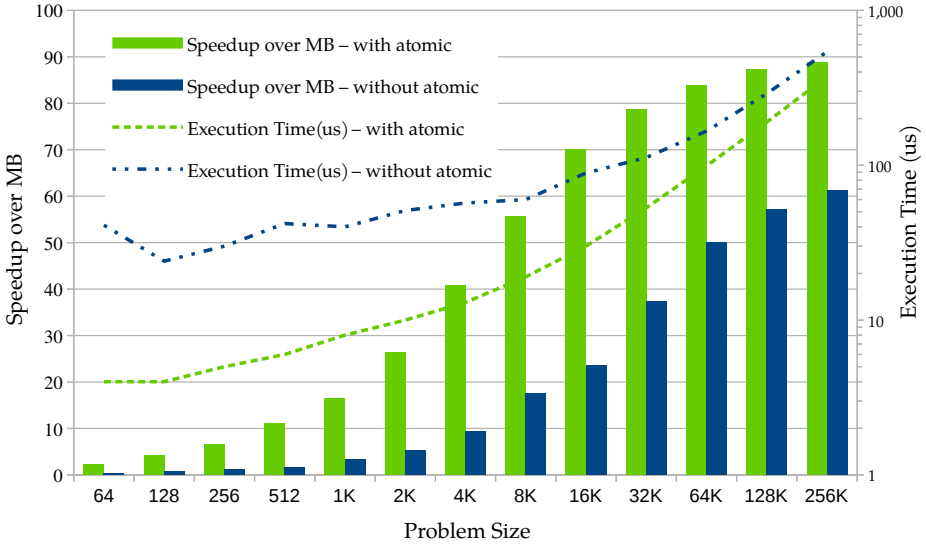


Figure 6.10: Execution time and wall clock speedup over MicroBlaze for the *max* kernel with and without using atomics for different problem sizes

- In the non-atomic version: each work-item computes the maximum of n array elements. The result is another array that is n times smaller than the input. This kernel is launched recursively until the length of the result array is less than or equal to 32. Finally, a single work-item reduces it into one value.

- In the atomic version: after a work-item computes the maximum of the n elements, it performs an atomic write on a fixed address in the global memory with the maximum of the local computed maximum and the old address content.

The value of the reduce factor n is determined empirically on each problem size for best execution time. The performance gap between the atomic and the non-atomic implementations starts at its maximum for the minimum problem size and reduces with increasing input data size (see Figure 6.10). In fact, when a kernel is launched on FGPU, about 2-3us must be spent on internal initialization. The recursive launch of the non-atomic version magnifies this delay. However, it becomes less critical as more time is spent on processing more data. Even at the biggest problem size, the atomic version is capable of saving 45% of the execution time.

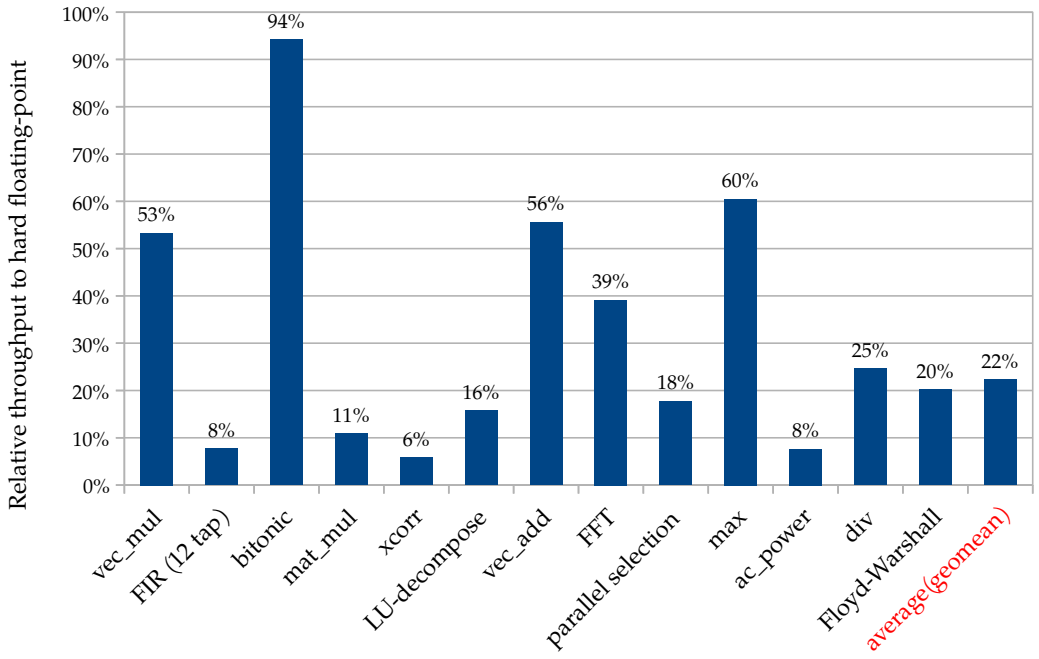


Figure 6.11: Relative throughput of processed data using soft floating-point implementations with respect to the hardened ones

6.4.4 Soft Floating-Point

Executing the required floating-point operations in software works irrespectively of the underlying hardware support. Therefore, significant savings in FPGA resources can be achieved: the floating-point logic may consume up to 48% of the FGPU area (see Figure 3.11 on Page 56). However, remarkable penalties in execution time have to be expected. Figure 6.11 shows the performance degradation of the soft approach in comparison to hard one. If only simple floating-point operations are required, e.g. compare, the degradation may be as small as 6% in *bitonic* sorting. When more complex operations are needed, 17x less throughput may be measured like for the *xcorr* benchmark. Actually, on a conventional processor architecture, the performance gap between soft and hard floating-point operations may be two orders of magnitude. On average, this gap is about 4.5x on FGPU.

6.5 Area and Power Efficiency

To evaluate the efficiency of the FGPU architecture, the following metrics have been used:

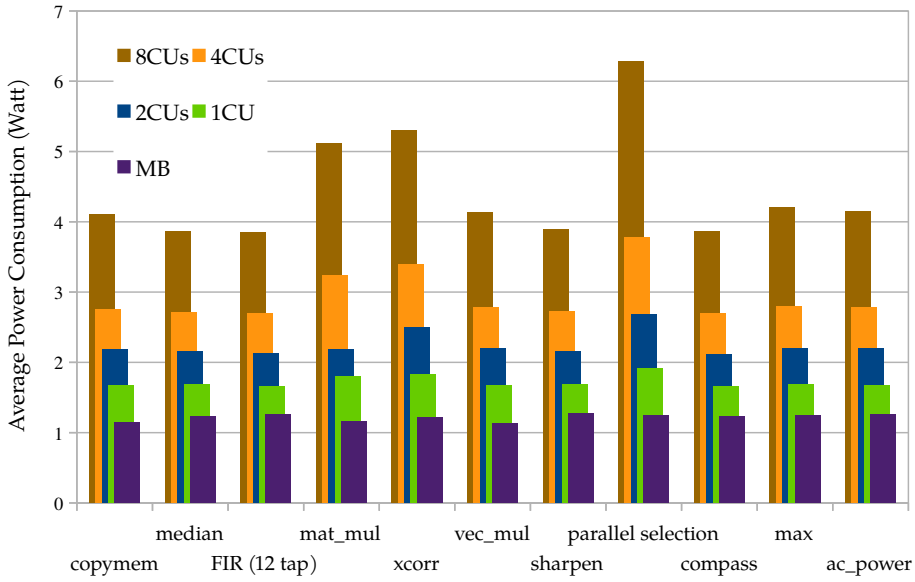


Figure 6.12: Power consumption for the MicroBlaze and multiple realizations of FGPU for some fixed-point benchmarks at a problem size of 256Ki

- **Processing Throughput:** the amount of written data, generated after processing, per time unit (MB/sec). This can be computed for each running benchmark. The averaged value is used to compare the performance of multiple processors quantitatively.

- **Compute Density (CD):** efficient FGPU cores require a lot of FPGA resources to be realized. To decide whether it is worthy using FGPU, we use the *CD* metric defined in [75]:

$$Compute\ Density = Processing\ Throughput\ (MB/sec)\ /\ Area\ (\#LUTs)$$

- **Energy Efficiency:** The energy consumed when executing a benchmark is the product of power consumption and execution time.

6.5.1 Fixed-Point Arithmetic

Figure 6.12 illustrates the power consumption for all fixed-point benchmarks of the MicroBlaze and multiple versions of FGPU. All values include about 0.9Watt that we recorded even if the PL logic is not programmed. On average, the FGPU cores needed 1.7-4.4Watt while the MicroBlaze consumed only 1.2Watt. Thanks to the measured

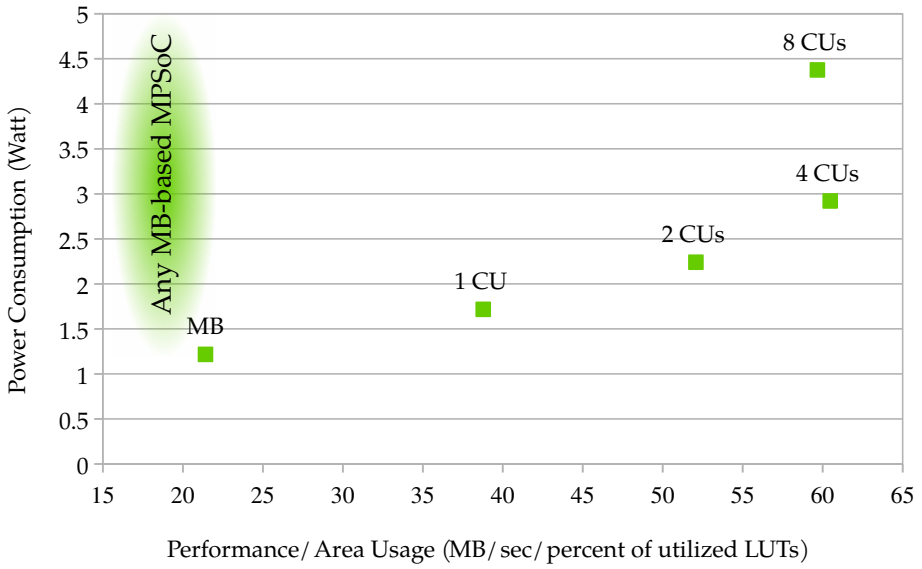


Figure 6.13: Average consumed power and throughput per FPGA area (CD) when executing the fixed-point benchmarks at a problem size of 256Ki using MicroBlaze and multiple FGPUs

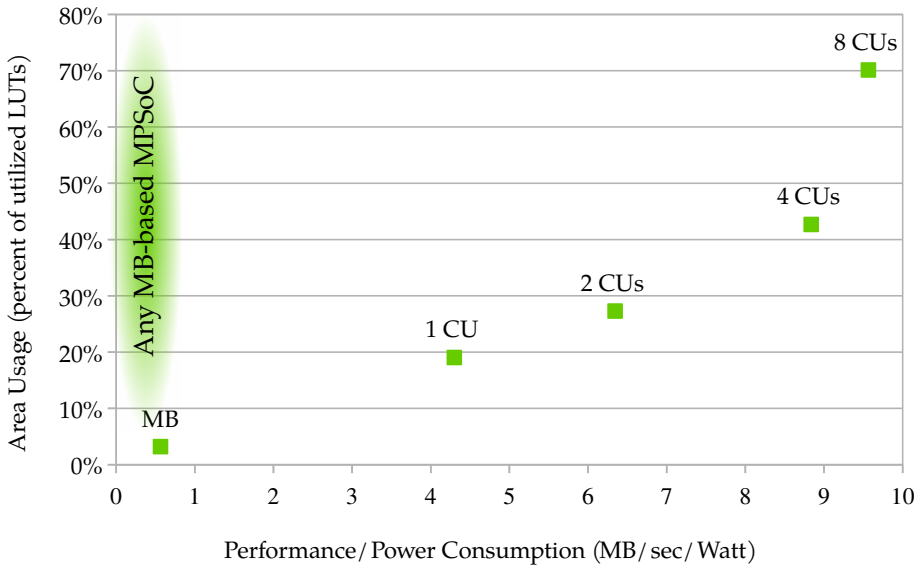


Figure 6.14: Occupied area and average throughput per power unit when executing the fixed-point benchmarks at a problem size of 256Ki using MicroBlaze and multiple FGPUs

speedups, the FGPU cores were able to save 87-94% from the energy consumed by the MicroBlaze when executing the same tasks (see Figure 6.14).

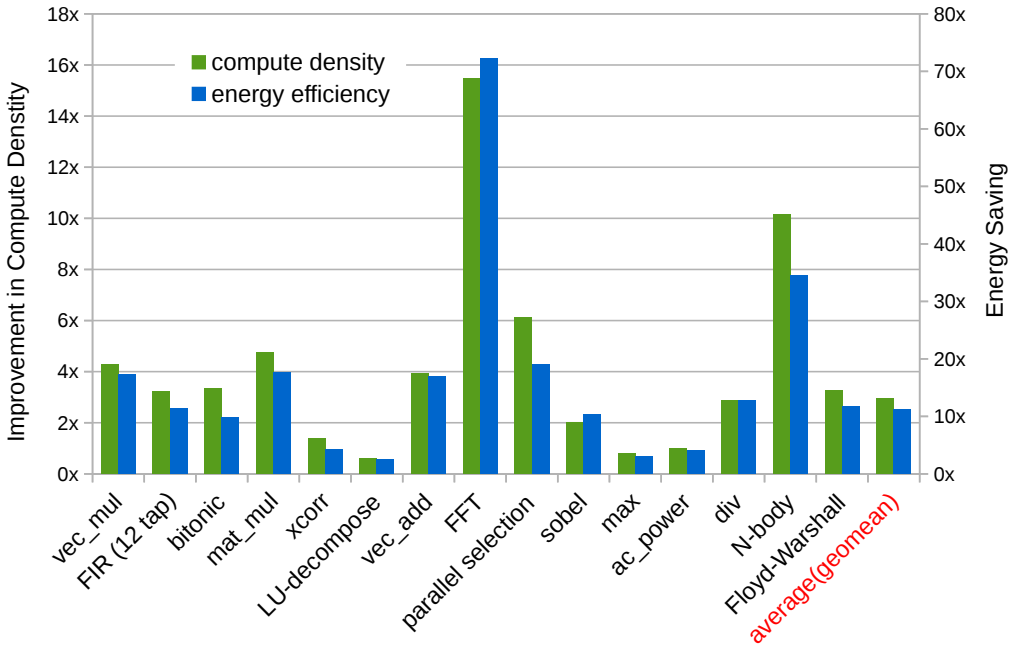


Figure 6.15: Improvement in compute density and power saving over MicroBlaze when using FGPU with hard floating-point support

Figure 6.13 and Figure 6.14 show the area overhead, power consumption and throughput of processed data for multiple FGPU cores and the MicroBlaze. The throughput as well as the power consumption are computed at a problem size of 256Ki and then averaged over all benchmarks. In comparison to the MicroBlaze (about 7K LUTs), the realized soft GPUs are 6-22x bigger. However, they are 7.6-17x more energy efficient, and they have 1.8-2.8x better compute density than the MicroBlaze. We have realized that increasing the number of CUs helps always to improve the energy efficiency. The best CD value is achievable when 4CUs are realized. A very slight degradation of about 1.3% was noticed when using 8CUs.

6.5.2 Floating-Point Arithmetic

About 124-180K LUTs are required to implement any of the FGPU cores with hard floating-point support. This corresponds to 57-82% of the available LUTs on the z7045 chip. In comparison to the reference MicroBlaze, the FGPU cores are approximately 18-26x bigger. The CD values have been calculated for FGPU and MicroBlaze for all benchmarks and problem sizes between 1-256Ki. Then, the average on problem size is calculated and the ratios of FGPU-values to the ones of the MicroBlaze are plotted

in Figure 6.15. On average, FGPU with hard floating-point support has 2.9x better compute density. For *FFT*, up to 15.4x better throughput per area is recorded. The MicroBlaze is only better for the *max* and *LU-decompose* benchmarks. To estimate the area overhead of FGPU, only the required floating-point operations have been implemented for each benchmark.

Using on-board power measurements, the utilized FGPUs consumed 4.9-7.5Watt over all benchmarks for soft or hard floating-point computation. On the other side, the MicroBlaze consumed a maximum of 1.24 Watt and on average 5.4x less power than FGPU. However, considering the measured speedups, FGPU with hard floating-point support needs 11.2x less energy than the MicroBlaze. In the worst case, 2.5x less energy was consumed by FGPU to compute *LU-decompose*. In addition, a strong correlation between CD and energy saving values has been observed as depicted in Figure 6.15.

6.6 A comparison with homogeneous MPSoCs

The compute power of an MPSoC or an FGPU can be simply extended by increasing the number of constituent processors or CUs, respectively. There is no special reason that makes MPSoCs slower or faster than FGPUs. However, if a time deadline has to be met when executing an application, it would be interesting to know which architecture needs less area, power or energy to fulfill this requirement.

In the best case scenario, the following statements holds for any MPSoC architecture that consists of N processors:

- The performance is directly proportional to N. Some degradation in the operation frequency, as well as bottlenecks in the NoC, can happen for the bigger N values.
- The power consumption increases linearly with N.
- When ignoring the area overhead of the underlying NoC, the area of an MPSoC equals N times the area of a single processor.

Hence, the throughput per area and the energy efficiency of an MPSoC can not exceed the corresponding values of a single processor. For example, if any MicroBlaze-based MPSoC is evaluated using our fixed-point benchmarks, the results would look like as plotted in Figure 6.13 and Figure 6.14.

To the best of our knowledge, MicroBlaze is one of the most efficient soft processors on Xilinx FPGAs. It has the highest operation frequency and the lowest area overhead in comparison to open-source processors [43]. Since FGPU has better CD and energy consumption values than a single MicroBlaze, one can conclude that using FGPU for general purpose computing is more area- and energy-efficient than using MPSoCs on FPGAs.

6.7 Summary

FGPU is a configurable and realistic soft GPU architecture. It offers an open-source and flexible platform for exploring the design space of GPUs. With soft GPGPU computing, better compute density and less energy consumption can be achieved in comparison to using soft general purpose processors. This applies for fixed-point as well as for floating-point computations. Although it may be possible to achieve a similar throughput using MPSoCs, soft GPUs need less area and they consume less energy. In addition, FGPU can outperform soft and hard vector coprocessors like NEON.

Chapter 7

Case Studies: Comparison with HLS and Partial Configuration

High-Level Synthesis (HLS) has been becoming an essential component in the tool portfolio of FPGA vendors. In contrast to HDL development, it is easier to use, and it has a much shorter edit-compile-verify cycle. For example, modern HLS tools offer many floating-point operations in multiple precisions and they are capable of generating pipelined RTL implementations for a pre-defined target clock frequency [89]. Most recent publications confirm that HLS can offer better performance than floating-point overlays [41] and better area efficiency as well [75]. Both FGPU and HLS can be programmed with OpenCL. We dedicated the first part of this chapter to discuss the pros and cons of the two approaches with a quantitative comparison.

The second part of the chapter explores the potential benefits of partial reconfiguration technology to reduce the area demands of soft GPU implementations. The proposed solution is based on replacing the floating point units at runtime according to the executed software instructions. Although an efficient implementation was not possible within the time frame of this thesis, some light will be thrown on the required design modifications as well as the main challenges accompanied with this technology.

7.1 Programming with HLS vs. FGPU

7.1.1 Coding Style, Code Size and Portability

Most HLS tools that target FPGAs use C/C++ or OpenCL[1] as a programming language. To describe parallelism and define dependencies in the kernel code, HLS programmers have to use `pragma` directives and special compiler attributes to unroll a loop or to characterize a bus interface, for example. These commands are complicated, vendor-specific and difficult to standardize. Much of them are optimization-oriented, i.e. they do not affect the design functionality; but they help to achieve significant performance improvements. For instance, Listing 7.1 shows the OpenCL code of an optimized 3x3 median image filter which is 74x faster than the simplest implementation, in which the blue code lines are ignored [102]. In addition, the performance of HLS outputs is very sensitive to the coding style. HLS code should be structured to reflect an efficient hardware realization.

On the other side, programming a GPU architecture is much simpler. The same programming guidelines for performance optimization apply for different architectures from different vendors. The OpenCL standard includes an API to program and control the GPU device. Moreover, software implementations for GPUs are much compact, portable and easier to understand and debug than HLS ones (see Listing 7.1).

7.1.2 Data Traffic Optimization

In many embedded applications, raw and processed data have to be stored in separate memory modules, e.g. SDRAM. This may shift the bottleneck from the required compute power to the bandwidth of the memory connection. Hence, for many compute-intensive but IO-bound tasks, local data caching and the resulting memory access pattern plays a major rule in determining the overall performance.

HLS users should be aware of how external data in global memory is being accessed. To generate efficient burst patterns, special functions like `memcpy` have to be used. To eliminate any redundant transfers, e.g. reading the same memory content multiple times, local buffers have to be created to serve repeated read operations. Additional local storage is required to realize Write Combine Buffers (WCBs) to fuse single

[1] OpenCL is based on C and its code can be compiled with any C compiler if the corresponding libraries are provided.

Listing 7.1: An optimized median image filter in OpenCL as suggested in SDAccel Development Environment Methodology Guide [102]

```
1  __kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
2  void median(__global const uint* input, __global uint* output, int width, int height) {
3    local uint line0[MAX_WIDTH], line1[MAX_WIDTH];
4    local uint line2[MAX_WIDTH];
5    for (int line = 0; line < height; line++) {
6      if (line == 0) {
7        async_work_group_copy(line0, input, width, 0);
8        async_work_group_copy(line1, input, width, 0);
9        async_work_group_copy(line2, input + width, width, 0);
10     }
11     else if (line < height-1) {
12       if (line % 3 == 0)
13         async_work_group_copy(line0, input+(line+1)*width, width, 0);
14       else if (line % 3 == 1)
15         async_work_group_copy(line1, input+(line+1)*width, width, 0);
16       else if (line % 3 == 2)
17         async_work_group_copy(line2, input+(line+1)*width, width, 0);
18     }
19     barrier(CLK_LOCAL_MEM_FENCE) ;
20     __attribute__((xcl_pipeline_loop))
21     for (int x=0; x < width; x++) {
22       ..
23       __attribute__((opencl_unroll_hint))
24       for (int channel = 0; channel < CHANNELS; channel++) {
25         result |= getMedian(channel, rgb);
26       }
27       lineres[x] = result;
28     }
29     async_work_group_copy(output + line*width, lineres, width, 0);
30     barrier(CLK_LOCAL_MEM_FENCE);
31   }
32 }
```

■ Code lines for Global/local memory management, pragmas or attributes (needed only for HLS)

■ Other code lines (needed for HLS or FGPU)

write transactions into burst ones. An example is shown in Listing 7.1, where 3 image lines (`line0-2`) are buffered locally and the function `async_work_group_copy` is used to read and write them.

When programming GPUs, a canonical memory model with a pre-given hierarchy is given: constant, global, local and private memories [52]. Although FGPU does not yet support the four types[2], it includes a sophisticated cache controller that can simultaneously serve up to 64 read/write requests. This controller creates burst read and write transactions to global memory out of the single requests and it can distribute data traffic on multiple interfaces. In contrast to HLS, where the tool defines when and how to generate the data transfers and which hardware has to be used at synthesis time; FGPU does this task at runtime and by dedicated hardware controllers. This reduces the programming effort as well as the code size for FGPU.

7.1.3 Allocation, Scheduling and binding

A high-level software description of a task defines only its functionality. For a hardware realization, HLS tools have to [22]

- **allocate** the required logic, i.e. the type and the amount of the needed hardware blocks have to be defined,
- **schedule** the required operations by assigning them to clock cycles and
- **bind** the required operations and the used variables to created blocks and storage resources, respectively.

HLS makes these steps at synthesis time where some optimizations can be done according to the user's constraints. For instance, if the maximum size of an image line is defined, HLS may allocate BRAMs and bind them to the cached image lines.

On the other side, the size and the compute capabilities of an FGPU core can be configured independently of the task code. Hence, the allocated hardware does not generally correspond to the best solution that suits the application needs. Moreover, the scheduling and binding problems are solved by dedicated hardware controllers at runtime. Therefore, soft GPU implementations are expected to be more area- and power-consuming than HLS ones.

[2] Currently, only private and global memories are supported. Local memory can be used to implement the call stacks of the running work-items but it can not be allocated directly by the programmer.

If the coding style of the targeted benchmark in HLS does not reflect an efficient hardware structure, soft GPU implementations can perform better regarding throughput as explained later in this chapter. Moreover, since the scheduling and binding problems have to solved offline in HLS, the quality of the generated hardware implementations may degrade if some parameters are not fixed at synthesis time, e.g. when implementing a filter with a varying number of tabs. This issue does not affect the efficiency of FGPU since the scheduling and binding steps are solved at runtime.

7.1.4 Meeting the desired area-performance trade-off

To control the size of the generated hardware block, HLS developers have to use compiler directives and attributes to guide their tools. These extra commands become a part of the task description. Usually, getting the desired performance requires repetitive and manual adjustments to these attributes according to the characteristics of the synthesized design.

When using FGPU, performance features and their area overhead can be estimated independently from the targeted applications. Moreover, the same software can run on many FGPU cores without modifications [3]. Table 7.1 summarizes the main differences between the HLS and the FGPU tool flows.

7.2 Fixed-Point Arithmetic

To estimate the performance gab between FGPU and HLS we reimplemented some of our benchmarks (listed in Table 6.1 on Page 88) using Vivado HLS (v2015.4) in OpenCL. These benchmarks include:

- Three image filters (*sharpen*, *median* and *compass*): built upon a highly optimized design example shown in Listing 7.1 given by the HLS tool provider [102].
- Vector multiplication (*vec_mul*): based on a pre-optimized example project delivered with Vivado HLS.
- An FIR filter (*fir*): with 10 taps. The multiply and accumulate operations are unrolled completely.

[3] The software may have to be recompiled when some features are not supported by the FGPU core, e.g. hard floating-point instructions. However, the task code remains untouched.

Table 7.1: Summary of the differences between programming for HLS or FGPU

	HLS	FGPU
resource allocation	at synthesis time	at synthesis time *
operation scheduling & binding	at synthesis time	at runtime
data traffic optimization	manual at design time	automatic at runtime
development effort	moderate	lower
area overhead	low	high
coding style	should reflect the hardware structure	should follow general guidelines
code portability	moderate	high
performance/area trade-off	defined using compiler directives and attributes	by configuring the core features

* If partial reconfiguration is supported, the hardware can be reallocated at runtime

The synthesized hardware include the needed logic to perform required data transfers. Figure 7.1 shows the execution time on FGPU with 8 CUs next to the one recorded with HLS for the selected applications. The input data was either a 512x512 image or an array of 256Ki 32bit elements. Both HLS and FGPU performs very closely to each other for all benchmarks except for *vec_mul*[4]. Although 4 out of 5 FGPU implementations are faster than the HLS ones, it might not be accurate to claim that FGPU is faster in general.

We picked the *fir* kernel for further investigation and implemented it with different settings in Vivado HLS. The outer for-loop, which iterates on the samples of the input signals, was unrolled with different factors. The inner loop, which iterates on the filer taps, is unrolled completely and the number of tabs was fixed before synthesis. Figure 7.2 depicts the measured throughput for many FGPU and HLS realizations against their power consumptions. For the HLS implementations with 0, 8, 16 and 32 unroll factors, the operating frequency was set at 250MHz. The throughput improves steadily with increasing unrolling. As we tried to unroll with 64, there were not enough DSP

[4] This is because the data flow optimization was not considered in the HLS example project which we used.

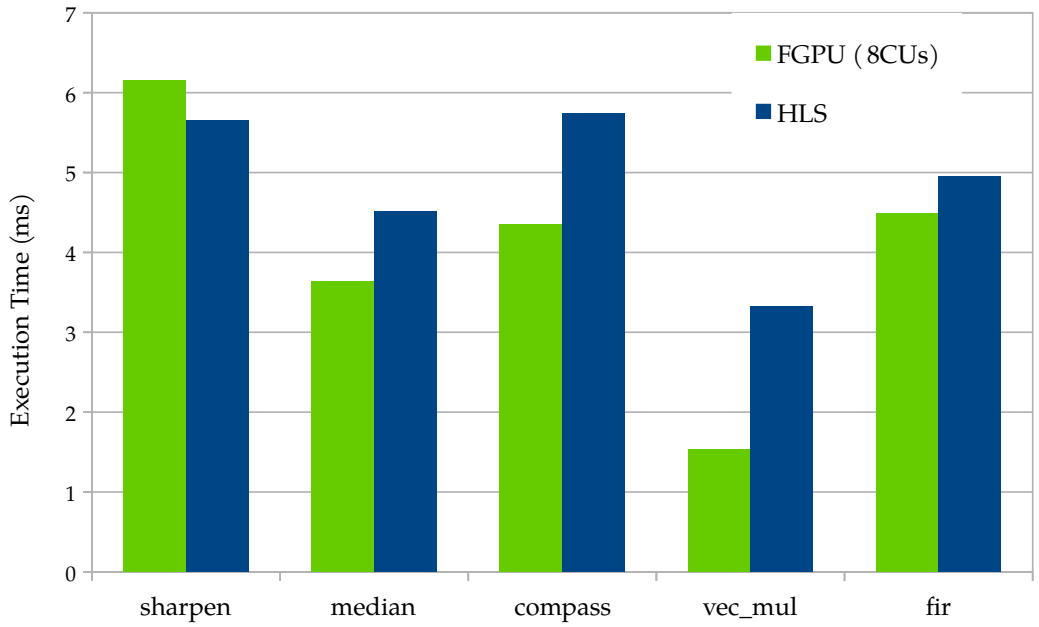


Figure 7.1: Execution time using FGPU and HLS for some fixed-point applications when processing 512x512 data words

blocks on the PL of the z7045 SoC. However, we could synthesize and run the 64 variation on 175MHz. The observed performance is even worse than the one we measured with smaller unrolling factors.

The power consumption and area overhead of all HLS-outputs are remarkably less than those of the FGPU cores. Nevertheless, the most performant HLS-implementation consumes 540 DSP blocks where the biggest FGPU needs only 256 ones. However, Figure 7.1 and Figure 7.2 show that even optimized HLS implementations cannot guarantee better performance than FGPU.

To achieve the best performance with HLS, all data transfers from global memory to local buffers are programmed and synchronized manually. Performance-limiting synchronization barriers have to be inserted after the transfer commands before doing any computations (see Lines 19 and 30 in Listing 7.1). Moreover, the temporal storage of multiple image lines has to be managed when computing the image filters. Otherwise, the data access pattern would be suboptimal, e.g. no burst access mode will be used. This increases remarkably the programming effort with HLS.

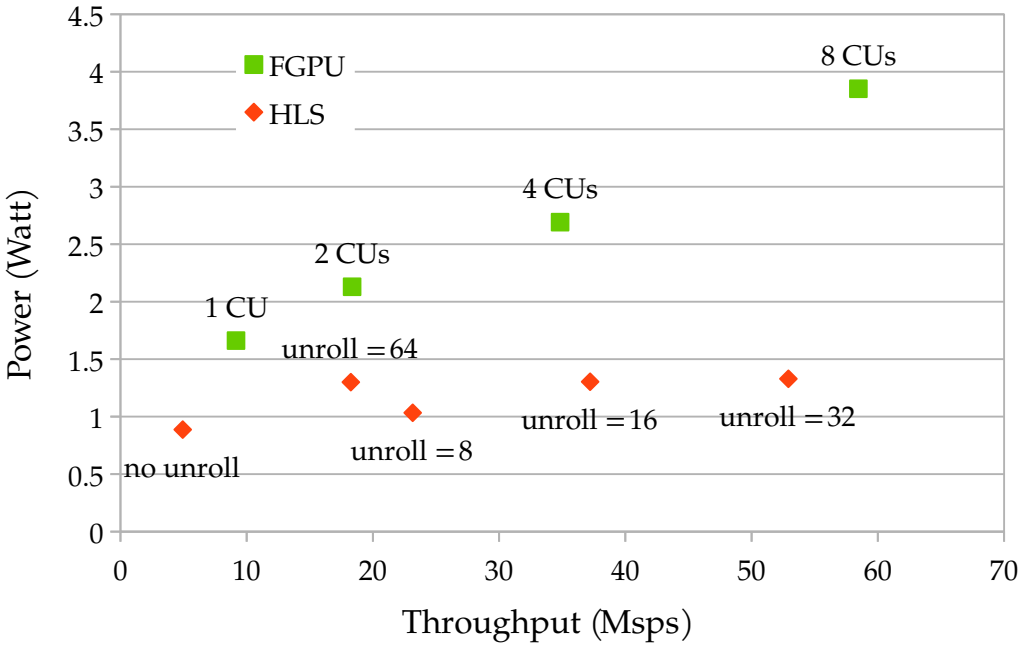


Figure 7.2: Power consumption and throughput of different soft GPUs and HLS realizations when computing an FIR of 10 taps

7.3 Floating-Point Arithmetic

We synthesized 4 of our benchmarks (listed in Table 6.1 on Page 88) using Vivado HLS (v2016.2): *FIR*, *N-body*, *bitonic* and *Floyd-Warshall*. In fact, efficient hardware realizations for applications like N-body motion simulation or bitonic sorting are quiet complex and out of the scope of this work. However, we have used the C-code of the ARM implementations described in Chapter 6 as a basis for our experiments with HLS. Moreover, we have synthesized the hardware with the same parameters of the ARM code, e.g. the problem size or the filter parameters were not known to HLS at synthesis time. The synthesized modules own the necessary logic to read and write the global memory without using external DMAs. Table 7.2 lists the needed number of LUTs for multiple implementations.

The columns Solution 1 to 3 use different implementation strategies in HLS:

- "Solution 1" refers to the case where we applied only some simple HLS optimizations like loop pipelining. Because the problem size is not specified and therefore

Table 7.2: Multiple HLS implementations of selected floating-point benchmarks with the execution time and required area

Benchmark	Solution 1		Solution 2		Solution 3	
	Time (ms)	#LUTs	Time (ms)	#LUTs	Time (ms)	#LUTs
FIR (12 Taps)	7.2	2.5 K	5.6	12 K	0.23	5 K
N-body	15301	6.2 K	1910	34 K		
Bitonic Sort	113	2.4 K	44	28 K		
Floyd-Warshall	638	2.4 K	130	9 K		

the loop bounds are unknown, HLS can only pipeline the inner loop [99]. Therefore, the required calculations are not effectively parallelized, and the synthesized hardware modules are relatively small.

- In "Solution 2", we rewrote the code supposing that the problem size is a multiple of 64. In other words, we transformed the inner loop into two ones, where the second loop has exactly 64 iterations. In this case, HLS can unroll the most inner loop with the fixed bound and pipeline the upper one next to it. Moreover, we used local data buffers to form burst-shaped traffic to global memory using the `memcpy` function.

- "Solution 3" is applied only on the FIR filter where all parameters are fixed at synthesis time, i.e. problem size and number of filter taps. In addition, all previously mentioned optimization techniques are used.

For a better understanding of the proposed strategies, the source code for the FIR benchmark is given for all solutions with the FGPU code in Appendix C. All implementations are placed and routed at frequencies from 187-250MHz. The measured execution times for a problem size of 16K (for Floyd-Warshall) or 8K (for all others) are reported in Table 7.2. We found that the implementations from the group "Solution 1" and "Solution 2" are 28-81x and 6.4-35x slower than the FGPU ones, respectively. FIR-Solution 3 is slightly slower than FGPU by a factor of 1.4x.

However, concluding that computing with FGPU is faster than with HLS would be inaccurate as mentioned before. Nevertheless, when flexibility is demanded, e.g. sorting arrays with variable lengths, simple hardware implementations of complex algorithms with HLS tools are not expected to perform as good as the FGPU ones. The main differences between the two approaches are the scheduling and binding parts:

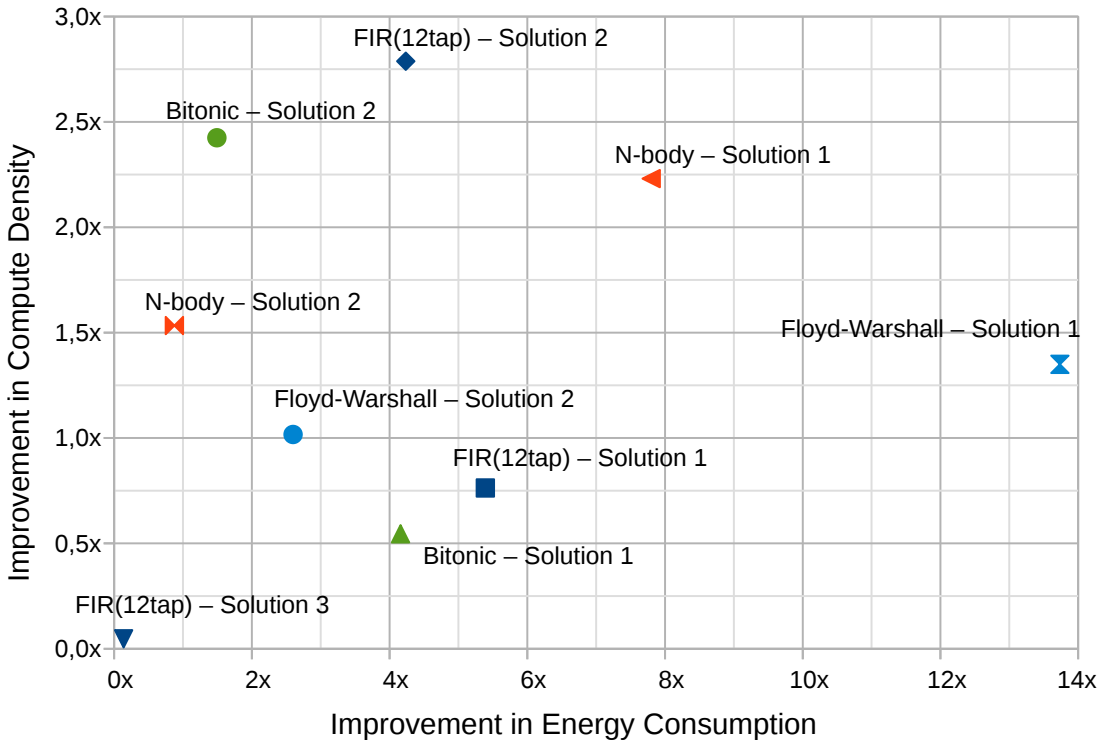


Figure 7.3: Improvement in compute density and power efficiency using FGPU over HLS

while HLS allocates the hardware and binds the required operations to the hardware blocks statically at synthesis time, FGPU solves this problem jointly with other ones like memory stalls at runtime using special hardware controllers.

Figure 7.3 depicts the improvement in compute density and energy saving when using FGPU over HLS. Indeed, except FIR-Solution 3, none of the HLS implementations is a pareto-optimal point. Only because all task parameters are fixed at synthesis time, Vivado HLS can synthesize FIR-Solution 3 very effectively: 7.2x better compute density and 22.2x better energy efficiency than FGPU are measured. However, efficiently programming an FIR filter in HLS is much simpler than the other considered benchmarks. FGPU proves better efficiency in the majority of the other solutions.

Moreover, the clear separation between the hardware and the software development with soft GPUs makes it more attractive: customizing a FGPU hardware implementation is done by selecting the desired features. It is much simpler than considering the many `pragma` directives to guide an HLS tool. While efficient coding for HLS should reflect a good hardware structure which may be not known to the developer, simple and compact software implementations are often enough to get the best performance out of an FGPU.

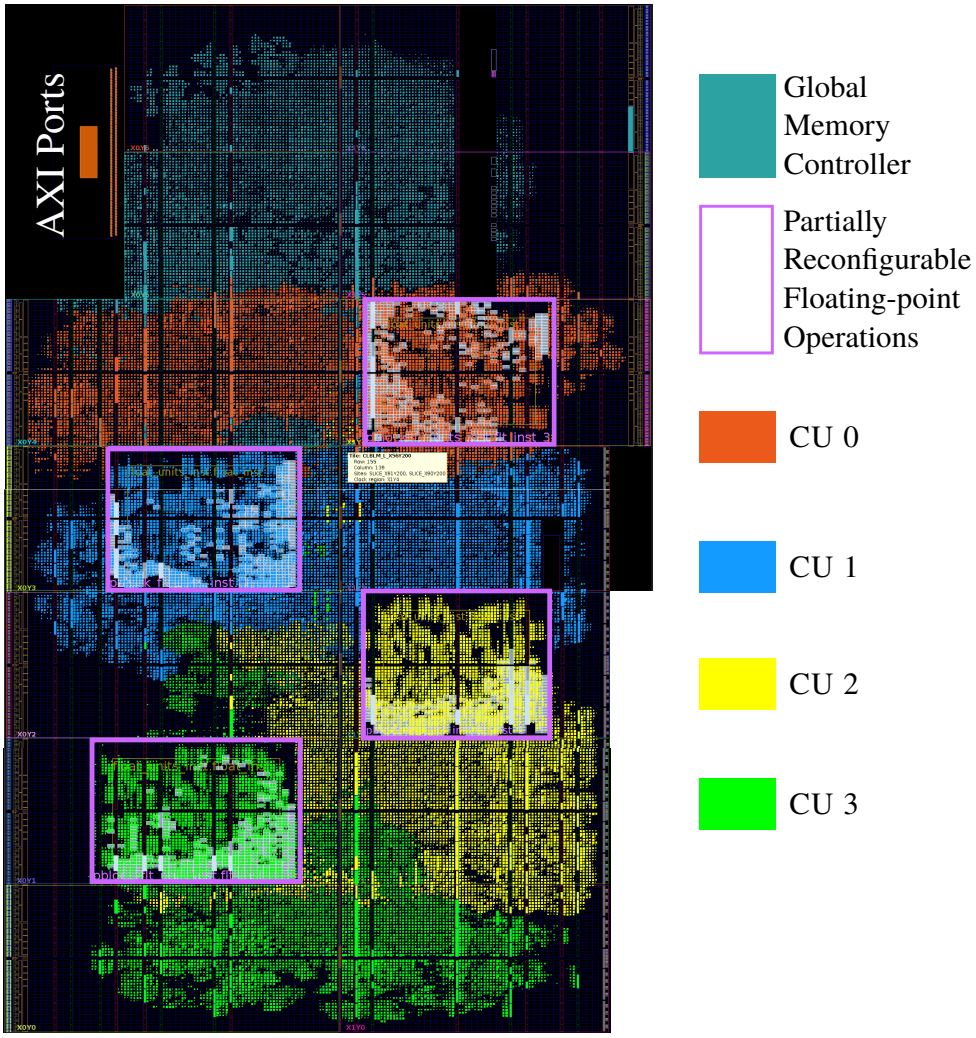


Figure 7.4: Layout of a placed and routed FGPU with 32PEs (4CUs) with partially reconfigurable floating-point operations

7.4 Partially Reconfigurable Floating-Point Operations

Partial reconfiguration enables modifying parts of a running FPGA system without disturbing others ones [98]. It helps to improve the area efficiency of a design by reprogramming unused logic with tasks that are currently required, or expected to be required later. In the FGPU case, if it is possible to exchange hard floating-point operations at runtime, significant area savings would be achieved.

On the other hand, partial reconfiguration has some drawbacks: the boarders of reconfigurable regions have to be manually defined and some drop in the maximum operation frequency should be expected. To mitigate the first problem, we gathered all floating-point-logic of a single CU within one module (see Figure 3.2 on Page 35).

Hence, only one reconfigurable region per CU (8PEs) has to be defined. Moreover, the floating-point logic is designed without any reset signal to minimize the expected frequency degradation.

We picked up an FGPU with 32PEs and tried many times to redraw the reconfigurable regions to get the minimum frequency degradation. Our best attempt is shown in Figure 7.4. For the initial place and route, only floating-point division is implemented. Then, the division logic is replaced with other floating-point operations. For each supported instruction, a partial bitstream per reconfigurable region is generated. The LUT utilization in all regions including the static one was about 60-70%. Despite the many IO connections of a reconfigurable region (773 signals), the design could be routed surprisingly at 250MHz.

7.4.1 Use Case 1: Offline Reconfiguration

If multiple tasks with different requirements on the floating-point native support have to be executed on FGPU, extremely high area demands can occur if all operations are realized simultaneously in hardware. Although the current tool flow of FGPU allows implementing some floating-point operations in hardware and emulating others in software, a more efficient implementation can be achieved if multiple FGPU cores are used to run the different tasks. However, exchanging the running FGPU core on the FPGA fabric is accompanied with significant time and power overheads. Since only the floating-point units have to be modified, partially reconfiguring an existing FGPU core is definitely more efficient than reprogramming the whole FPGA.

To enable this feature in the FGPU tool flow, no hardware extensions to the original architecture are required. However, the user should make the decision about which floating-point operations will be supported in hardware and configure the FGPU compiler accordingly to emulate the unsupported ones. In addition, the user API should offer the needed functionality to reconfigure an FGPU core before a task gets executed.

Partial Reconfiguration under Linux

Modifying the supported floating-point instructions of a deployed FGPU has to be triggered manually by the user's software. Therefore, providing the feature of partially reconfiguring an FGPU (on the PL) from an API (running on the PS) has to be offered.

On the hardware side, this is realized through the Processor Configuration Access Port (PCAP) interface. The ARM processor includes a device configuration unit (DevC) that enables configuring the PL at runtime. The partial bitstreams are transferred to DevC over the central ARM bus, where a dedicated DMA controller is used to accelerate this process. Thanks to the built-in AXI/PCAP bridge between DevC and PCAP, the AXI bus messages get converted to PCAP-compatible ones and vice-versa [90]. Receive/transmit FIFOs are used in both communication directions to move the reconfiguration data between the different PCAP and AXI clock domains. The maximum 3.2 Gbps reconfiguration speed of the PCAP is still achievable with this architecture, where the PCAP is clocked at its 100MHz maximum clock frequency [101].

On the software side, the reconfiguration process is triggered and managed through the `xdevcfg` driver that controls DevC. This driver enables performing a complete configuration or a partial reconfiguration for the programmable logic. It is built on a virtual file system, where the reconfiguration is done by writing the bitstream to some location in the Linux file system. This triggers a DMA transfer for the partial bitstream and blocks the operating system by polling, waiting for an event that indicates the end of the reconfiguration process. The partial bitstreams can be loaded from the Linux file system (stored on an SD card). Indeed, programming a complete or a partial bitstream of an FGPU core is done similarly in Linux[5].

Since a partial bitstream programs a single reconfigurable region and hence a single CU, N partial bitstreams should be downloaded when reconfiguring an FGPU with N CUs. Currently, it is not allowed that different CUs support different floating-point instructions.

7.4.2 Use Case 2: Online Reconfiguration

If a single task requires many floating-point operations where the accompanied area overhead is not affordable, the offline setup is not helpful. Therefore, exchanging the supported floating-point instructions in hardware at runtime might be the only solution to reduce the FGPU footprint. This can be controlled according to the executed assembly instructions. If a wavefront requires a floating-point operation, it stalls until a special reconfiguration controller finishes the programming of the corresponding

[5] Only a single flag in Linux virtual file system needs to be set or cleared when writing partial or complete bitstreams, respectively

partial bitstream. Meanwhile, other wavefronts may execute their integer instructions without being disturbed.

However, the offline reconfiguration strategy can be considered as a special case of the online one. This applies if the allocated floating-point logic is not replaced until the execution finishes. Instead of manually reconfiguring the partial bitstreams via the API in the offline approach, FGPU learns which instructions are needed in the online one. The time delay of the reconfiguration process has to be tolerated just at the first attempt to execute a kernel. In contrast to the offline reconfiguration strategy, the online one allows having different floating-point logic in different CUs at the same time. Table 7.3 gives an overview of the main differences between the two approaches.

Hardware Modifications for Enabling Online Partial Reconfiguration

Since the reconfiguration is triggered and managed by hardware, the Internal Reconfiguration Access Port (ICAP) interface is used for partial configuration. The occupied area by the floating-point logic is relatively large, and multiple regions in multiple CUs have to be reconfigured. Hence, an on-chip storage of the corresponding partial bitstreams is not affordable. The tasks of the required reconfiguration controller can be summarized as follows:

1. Supervising the execution of floating-point instructions in all CUs. If the required logic is not yet available, a reconfiguration process has to be started. When finished, the controller gives the green light to stalling instruction(s).

2. Fetching the partial bitstreams from an external memory over an AXI4 interface.

3. Programming the FPGA resources through the ICAP interface.

We have developed and described the reconfiguration controller which fulfils the criteria 2 and 3 [45]. However, the actual reconfiguration speed on current FPGAs is still far away from the accepted limit for our use case. We believe that using other FPGA architectures like multi-context FPGAs or future improvements in the partial reconfiguration technology may allow a very efficient implementation of the proposed concept. Further ideas are explained in the next chapter.

Table 7.3: Summary of the differences between the offline and online reconfiguration strategies

Feature	Offline Reconfiguration	Online Reconfiguration
reconfiguration granularity	the whole FGPU	CU-level
reconfiguration time	before execution	during execution
hardware modifications	-	required
API modifications	required	-
reconfiguration interface	PCAP	ICAP
kernel execution time	not affected	slower

7.5 Summary

The performance gap between FGPU and HLS is negligible for algorithms in fixed-point or floating-point. In contrast to HLS, FGPU can be programmed through a standard OpenCL interface. The same code which runs on other GPUs can be reused for FGPU without modifications. Because HLS creates the hardware according to the task code, better throughput per area and less energy consumption can be achieved than with FGPU. However, if some of the task parameters are not known at synthesis time, HLS can not parallelize the required computations effectively. Instruction scheduling and binding become less efficient. FGPU owns hardware controllers that solves these problems jointly at runtime with memory stalls.

Partial reconfiguration of the floating-point logic helps to reduce the area footprint of FGPU. An offline reconfiguration, i.e. before the execution starts, helps to reduce the FPGA programming time when executing multiple tasks with different demands on the hardened floating-point operations. An online reconfiguration, i.e. during execution, is still not efficient because of the high reconfiguration delay. Major modifications to the FGPU architecture are required to offer online reconfiguration efficiently.

Chapter 8

Conclusion and Future Work

8.1 Summary of Contributions

Chapter 3 describes the hardware architecture of FGPU: a highly pipelined multi-core processor designed especially for FPGAs. Its area overhead is realistic for modern middle-sized devices. In comparison to other soft GPU architectures, the remarkable advantage of FGPU is its scalability: to the best of our knowledge, there is no other soft GPU capable of running up to 4Ki Threads on 64 PEs at 250MHz or can perform similarly. Besides, FGPU is open-source, highly customizable and requires no other external logic.

Chapter 4 explains the specialties of the OpenCL compiler for the FGPU architecture. The proposed compiler is designed with the LLVM framework and its code is open-source as well. This chapter focuses on the main differences to a conventional compiler of a CPU architecture.

Chapter 5 suggests two tool flows to use FGPU in embedded systems. In addition to the standard OpenCL-API interface, a PYNQ-based environment is provided. This enables controlling and programming an FGPU core using Python code.

Chapter 6 evaluates FGPU against some other soft and hard processors. It proves that general purpose computing with soft GPUs is better regarding area and power efficiency than using single soft processors or homogeneous MPSoCs on the FPGA fabric. An FGPU with a single CU (8PEs) occupying about 14% of the FPGA logic of a z7045 and running at 250MHz can perform similarly to the 128bit NEON vector coprocessor running at 667MHz.

Chapter 7 compares FGPU to HLS. We have found that both approaches deliver similar performance. HLS-cores have a better area and energy efficiency if the code is structured in a way that reflects an efficient hardware architecture and if the task parameters are fixed at synthesis time. Otherwise, FGPU is proved to be a more efficient solution in many scenarios. Also, this chapter throws the light on the principle of partially reconfigurable soft GPUs, where the floating-point logic can be exchanged at runtime according to the application needs.

8.2 Future Work

In comparison to the well-established architectures of modern GPUs, FGPU still looks relatively simple. However, during the work on this thesis, a platform for implementing many new ideas has been created. In the following, we will describe some of the potential improvements to the FGPU architecture.

8.2.1 Asymmetric SIMD Machines

Currently, FGPU accommodates identical PEs. This coarsens the whole architecture and limits its configurability, e.g. supporting floating-point division in hardware where 4 CUs are implemented requires realizing 32 dividers. The accompanied area overhead doubles the footprint of FGPU. In addition, most of the computation logic remains underutilized at runtime: when executing any instruction, the logic for computing other ones in all PEs has to stay idle. Creating vectors of different ALUs would allow to simultaneously execute different instructions and hence to achieve better utilization.

8.2.2 Fine-grained Partial Reconfiguration

Currently, reconfiguring a new operation requires partially reconfiguring all CUs. Having heterogeneous SIMD vectors would reduce the granularity of the reconfigurable regions. This would push down the delay time of reprogramming the floating-point regions to more acceptable limits (see Section 7.4.2). If enabled, this feature would allow FGPU to execute a new operation as soon as it gets reconfigured into a single region in any CU.

8.2.3 Evolvable FGPU

FGPU developers have to carefully configure their cores according to the running applications to avoid unnecessary area overheads. Doing this task manually at synthesis time would limit the productivity of the FGPU tool flow. If asymmetric SIMD vectors are reconfigured at runtime according to the opcode of the instruction to be executed, the FGPU hardware would be able to extend or modify its ISA online without user intervention. This would allow the FGPU core to learn and to evolve as more time is being spent on execution.

8.2.4 Cache Organization

In fixed-point FGPUs, approximately half of the area overhead is caused by the memory and cache controllers. Their settings are very critical to the overall performance. Only direct-mapped cache systems with write-back strategy have been investigated. We realized in many cycle-accurate simulations that improving the cache properties would significantly improve the overall performance, e.g. increasing associativity. Analysing the performance boost and the accompanied area growth would help to improve the efficiency of FGPU.

8.2.5 Better Compliance to the OpenCL Standard

FGPU does not support yet synchronization barriers and many atomic operations. In addition, its compiler considers only the `.text` section of a compiled object filed, which contains the code to be executed. This prevents defining read-only data in the program code, for instance. On the API side, only the basic functions that we need to run our benchmarks are implemented. Better support for the OpenCL standard would help to expand the application area of FGPU.

8.3 Closing Remark

The personal motivation behind this thesis was providing an evidence that using networks of general purpose processors to execute data oriented tasks is not a pareto-optimal solution. GPU architectures can utilize the computing logic more efficiently

than normal CPUs. Despite of their higher power consumption, GPUs can be a better solution in terms of energy and area efficiency. Although this statement has been proved only with experiments on FPGAs, there is no reason that makes it invalid for architectures implemented directly on silicon. We do believe that a GPU which is aware of the reconfigurability of the underling FPGA logic would lead to a new family of GPU architectures. This may make FPGAs essential components of future GPU chips. I hope that FGPU will be one of the early steps in developing such architectures.

Appendix A

FGPU Instruction Set Architecture

Type	Inst.	Parameters	Description
Fixed-Point Arithmetic	add	rd, rs, rt	rd = rs + rt
	sub	rd, rs, rt	rd = rs - rt
	addi	rd, rs, $imm_{14}^{\pm}$	rd = rs + $imm_{14}^{\pm}$
	mul	rd, rs, rt	rd = rs * rt
	macc	rd, rs, rt	rd = rd + rs * rt
	li	rd, imm_{16}	rd = imm_{16}
	lui	rd, imm_{16}	$rd_{[31..16]}$ = imm_{16}
	slt	rd, $rs^{\pm}$, $rt^{\pm}$	rd = 1 if $rs^{\pm}$ < $rt^{\pm}$ else 0
	slti	rd, $rs^{\pm}$, $imm_{14}^{\pm}$	rd = 1 if $rs^{\pm}$ < $imm_{14}^{\pm}$ else 0
	sltu	rd, rs^{+}, rt^{+},	rd = 1 if rs^{+} < rt^{+} else 0
	sltiu	rd, rs^{+}, imm_{14}^{+}	rd = 1 if rs^{+} < imm_{14}^{+} else 0
Floating-Point	fadd	rd, rs, rt	rd = rs + rt
	fsub	rd, rs, rt	rd = rs - rt
	fmul	rd, rs, rt	rd = rs * rt
	fsqrt	rd, rs	rd = $\sqrt{rs}$
	fdiv	rd, rs, rt	rd = rs / rt
	fslt	rd, rs, rt	rd = 1 if rs < rt else 0
	uitofp	rd, rs	rd = rs
Logical	movz	rd, rs, rt	rd = rs if rt = 0
	movn	rd, rs, rt	rd = rs if rt != 0
	nor	rd, rs, rt	rd = rs nor rt
	xor	rd, rs, rt	rd = rs xor rt
	xori	rd, rs, imm_{14}	rd = rs xor imm_{14}
	or	rd, rs, rt	rd = rs or rt

Type	Inst.	Parameters	Description
Logical	ori	rd, rs, imm_{14}	rd = rs or imm_{14}
	and	rd, rs, rt	rd = rs and rt
	andi	rd, rs, imm_{14}	rd = rs and imm_{14}
	sll	rd, rs, rt	rd = rs « rt
	slli	rd, rs, imm_5	rd = rs « imm_5
	srl	rd, rs, rt	rd = rs » rt (logical)
	srli	rd, rs, imm_5	rd = rs » imm_5(logical)
	sra	rd, rs, rt	rd = rs » rt (arithmetic)
	srai	rd, rs, imm_5	rd = rs » imm_5(arithmetic)
Control	beq	rd, rs, $imm^{\pm}_{14}$	PC += (1+$imm^{\pm}_{14}$)«2 if rd = rs
	bne	rd, rs, $imm^{\pm}_{14}$	PC += (1+$imm^{\pm}_{14}$)«2 if rd != rs
	jsub	$imm^{\pm}_{14}$	PC += $imm^{\pm}_{14}$«2
	ret		return
	lp	rd, imm_4	rd = paramter Nr. imm_4
Memory	lw	rd, rs[rt]	rd = mem_{32}(rs + rt«2)
	lh	rd, rs[rt]	rd = mem_{32}(rs + rt«1)
	lb	rd, rs[rt]	rd = mem_{32}(rs + rt)
	sw	rd, rs[rt]	mem_{32}(rs + rt«2) = rd
	sh	rd, rs[rt]	mem_{16}(rs + rt«1) = rd
	sb	rd, rs[rt]	mem_8(rs + rt) = rd
	aadd	rd, rs[rt]	mem_{32}(rs+ rt«2) += rd
	amax	rd, rs[rt]	mem_{32}(rs+ rt«2)=max(rd, mem_{32}(rs + rt«2))
	llwi	rd, rs[imm^{+}_{14}]	rd = $scratchpad_{32}$((rs+imm^{+}_{14})«2)
	lswi	rd, rs[imm^{+}_{14}]	$scratchpad_{32}$((rs+imm^{+}_{14})«2) = rd
OpenCL-Related	lid	rd, dim	rd=work-item local ID in dimension *dim*
	wgoff	rd, dim	rd=work-gourp global ID offset in dimension *dim*
	wgid	rd, dim	rd=work-gourp ID in dimension *dim*
	size	rd, dim	rd=size of index space in dimension *dim*
	wgsize	rd, dim	rd=size of a work-group in dimension *dim*

Notes:

rd, rs, rt: registers holding 32bit numbers

$rd_{[31..16]}$: bits 31 to 16 from register rd

x^{+}: a fixed-point unsigned value

$x^{\pm}$: a fixed-point signed value
x_n: a fixed-point number on n bits
mem(x): global memory data at address x
scratchpad(x): scratchpad data at address x
dim : 0, 1 or 2

Appendix B

FGPU Control Interface

B.1 Link RAM (LRAM)

Table B.1: Address map of a single kernel descriptor in LRAM

	Bit fields					
Word Index	31:30	29:28	27:20	19:14	13:10	9:0
0	#WF-1 in WG		-		Address of first instruction	
1	Global size in D0					
2	Global size in D1					
3	Global size in D2					
4	Global offset in D0					
5	Global offset in D1					
6	Global offset in D2					
7	#dims-1	Size of WG in D2		Size of WG in D1	Size of WG in D0	
8	#WG-1 in D0					
9	#WG-1 in D1					
10	#WG-1 in D2					
11	#Parameters		-		WG size	
12..15	-					
16..31	Parameters array					

Shaded cells have to be defined by the user

Non-shaded cells can be computed according to user input by the driver

B.2 Address Map

The control interface of FGPU occupies an address range of 64KB, which is subdevided into three regions shown in Table B.2. Any access should be 32 bit alligned.

Table B.2: Address map for the FGPU control interface

Address range	Element	Size	#Address bits
0x0000 - 0x0800	LRAM	2KB	9 bits
0x4000 - 0x7FFF	CRAM	16KB	12 bits
0x8000 - 0x800F	Control Register	16B	2 bits

B.3 Control Registers

Table B.3: FGPU Control Registers

No.	Name	Access type	Description
0	status	R	set by FGPU when kernel execution is finished
1	start	W	start kernel execution
2	cache clean	RW	clean dirty cache content after the execution is finished
3	initialize	RW	initialize RTM content before execution starts

Supposing that the settings of the i-th kernel are stored at the i-th slot of the LRAM:

- The i-th kernel can be launched by setting the i-th bit of the *start register*.
- The i-th bit of the *status register* will be set when the execution is finished.
- The i-th bit of the *status register* will be cleared when the execution starts.
- If the i-th bit of the *clean cache register* is set, the dirty cache content will be flushed to global memory after the execution ends.
- If the i-th bit of the *initiate register* is set, the RTM memories will be initialized before the execution starts.

Appendix C

FIR implementations for FGPU and HLS

This appendix lists the source code for the FIR benchmarks used in Section 7.3. It should help the reader to understand the effect of fixing the parameters of an HLS core on the QoR of HLS outputs. In our case, these parameters are `filterLen` and `inputLen` which are the lengths of the filter and the array to be filtered, respectively.

C.1 HLS Implementations

Each of the following solutions includes two 32bit AXI4-master interfaces:

- An AXI4-lite one for controlling and configuring the core parameters.
- An AXI4-full one for data exchange.

All the loops in the following HLS solutions are pipelined. In case of nested loops, pipelining the outer loop implies unrolling the inner ones. If the bounds of the inner loops are not fixed at synthesis time, the automatic unrolling fails. In this case, only the most inner loop will be pipelined. Solution 1 and 2 assume unknown parameters' values while Solution 3 deals with fixed ones. The required tcl commands to define the I/O interfaces and pipeline the loops, or equivalently the compiler directives, are not included in the following code samples.

C.1.1 Solution 1

Listing C.1 shows our HLS implementation for lowest area overead. It has the lowest throughput among all other implementations but its compute density is better than FGPU and all other fully parameterized HLS implementations.

Listing C.1: FIR implementation in HLS -Solution 1

```
#define MAXIMUM_FILTER_LEN      32
void fir(float *input, float *output, float *coeffs, unsigned filterLen, unsigned inputLen) {
    float coeffs_buffered[MAXIMUM_FILTER_LEN];
    memcpy(coeffs_buffered, coeffs, sizeof(float)*filterLen);

    loop1: for(unsigned i = 0; i < inputLen; i++) {
        float acc = 0;
        loop2: for(unsigned j = 0; j < filterLen; j++) {
            acc += coeffs_buffered[j] * input[i+j];
        }
        output[i] = acc;
    }
}
```

C.1.2 Solution 2

Listing C.2 unrolls the outer loop with an adjustable factor, here fixed at 2^6 or 64 [1]. In comparison to Solution 1, Solution 2 occupies 4.7x more area but is has a 1.3x better throughput and hence a worse compute density.

Listing C.2: FIR implementation in HLS -Solution 2

```
#define MAXIMUM_FILTER_LEN      32
#define UNROLL_W                6
void fir(float *input, float *output, float *coeffs, unsigned filterLen, unsigned inputLen) {
    float coeffs_buffered[MAXIMUM_FILTER_LEN];
    memcpy(coeffs_buffered, coeffs, sizeof(float)*filterLen);

    loop1: for(unsigned i = 0; i < inputLen>>UNROLL_W; i++) {
        float input_buffered[(1<<UNROLL_W) + MAXIMUM_FILTER_LEN];
```

[1] Unrolling the same loop with a `pragma` directive increased the area overhead but it did not improve timing. Vivado HLS fails to pipeline the data accesses from the unrolled loop parts to the same AXI4 interface.

```
        memcpy(input_buffered, &input[i<<UNROLL_W], sizeof(float)*((1<<UNROLL_W)+filterLen
            ));
        float output_buffered[1<<UNROLL_W];
        loop2: for(unsigned k = 0; k < (1<<UNROLL_W); k++) {
            float acc = 0;
            loop3: for(unsigned j = 0; j < filterLen ; j++) {
                acc += coeffs_buffered[j] * input_buffered[k+j];
            }
            output_buffered[k] = acc;
        }
        memcpy(&output[i<<UNROLL_W], output_buffered, sizeof(float)<<UNROLL_W);
    }
}
```

C.1.3 Solution 3

Listing C.3 shows the code of Solution 3 where the filter and the input array lengths are already known to HLS. Although the FGPU implementation is about 1.4x faster than this solution, the synthesized HLS core is about 30x smaller than FGPU. This solution is better than all other HLS solutions in terms of performance and compute density.

Listing C.3: FIR implementation in HLS -Solution 3

```
void fir(float *input, float *output, float *coeffs) {
    const unsigned constFilterLen = 12;
    const unsigned constInputLen = 8*1024;

    float coeffs_buffered[constFilterLen];
    memcpy(coeffs_buffered, coeffs, sizeof(float)*constInputLen);

    float input_buffered[constInputLen + constFilterLen];
    float output_buffered[constInputLen];
    float acc;
    memcpy(input_buffered, input, sizeof(float)*(constInputLen+constFilterLen));
    loop1: for(unsigned i = 0; i < constInputLen; i++) {
        loop2:for(unsigned j = 0; j < constFilterLen; j++) {
            if(j == 0)
                acc = 0;
            acc += coeffs_buffered[j] * input_buffered[i+j];
        }
        output_buffered[i] = acc;
```

```
    }
    memcpy(output, output_buffered, sizeof(float)*constInputLen);
}
```

C.2 FGPU Implementation

Listing C.4 shows the OpenCL implementation for FGPU. No special compiler directives are needed. The code size is remarkably smaller than all HLS implantations. Moreover, the recorded throughput on an FGPU with 8 CUs is 40% better than the best HLS core while the filter and input array lengths are configurable. The high flexibility and good performance of FGPU are accompanied with high area overheads.

Listing C.4: FIR implementation for FGPU

```
__kernel void fir(__global float *in, __global float *coeff, __global float *out, int filter_len) {
  int index = get_global_id(0);
  int i = 0;
  float acc = 0;
  do{
    acc += in[index+i] * coeff[i];
    i++;
  } while(i != filter_len);
  out[index] = acc;
}
```

Bibliography

[1] MXP Repository on Github. `https://github.com/VectorBlox/mxp`. [Online; accessed 12-Feb-2017].

[2] PYNQ Project. http://www.pynq.io [Online; accessed 15-Jan-2017].

[3] The LLVM Comiler Webseite. `http://www.llvm.org`. [Online; accessed 12-Feb-2017].

[4] Top 500 Supercomputers. Available on `http://www.top500.org`. [Online; accessed 05-Jan-2017].

[5] K. Aasaraai and A. Moshovos. SPREX: A soft processor with Runahead execution. In *2012 International Conference on Reconfigurable Computing and FPGAs*, pages 1–7, Dec 2012.

[6] A. Al-Dujaili, F. Deragisch, A. Hagiescu, and Weng-Fai Wong. Guppy: A GPU-like soft-core processor. In *Field-Programmable Technology (FPT), 2012 International Conference on*, pages 57–60, Dec 2012.

[7] Muhammed Al Kadi, Benedikt Janssen, and Michael Huebner. FGPU: An SIMT-Architecture for FPGAs. FPGA '16, pages 254–263, New York, NY, USA, 2016. ACM.

[8] Altera Corp. Implementing FPGA Design with the OpenCL Standard. White Paper, 2013.

[9] Altera, Corp. Nios II Core Implementation Details(NII51015), 2015.

[10] Altera Corp. Stratix 10 Device Overview. Initial Release, Dec 2015.

[11] AMD, Inc. ADM Accelerated Parallel Processing SDK v3.0. http://developer.amd.com/amd-accelerated-parallel-processing-app-sdk/ [Online; accessed 15-Jan-2017].

[12] AMD, Inc. AMD Graphics Cores Next (GCN) Architecture. White Paper, 2012.

[13] K. Andryc, M. Merchant, and R. Tessier. FlexGrip: A soft GPGPU for FPGAs. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pages 230–237, Dec 2013.

[14] K. Andryc, T. Thomas, and R. Tessier. Soft GPGPUs for Embedded FPGAs: An Architectural Evaluation. In *Overlay Architectures for FPGAs (OLAF), 2016 Second Workshop on*, Feb 2016.

[15] A. Ariel, W. W. L. Fung, A. E. Turner, and T. M. Aamodt. Visualizing complex dynamics in many-core accelerator architectures. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, pages 164–174, March 2010.

[16] ARM, Limited. AMBA ® AXI™ and ACE™ Protocal Specification, (ID102711), 2011.

[17] ARM Limited. Take GPU Processing Power Beyond Graphics with Mali GPU Computing. White Paper, 2014.

[18] Raghuraman Balasubramanian, Vinay Gangadhar, Ziliang Guo, Chen-Han Ho, Cherin Joseph, Jaikrishnan Menon, Mario Paulo Drumond, Robin Paul, Sharath Prasad, Pradip Valathol, and Karthikeyan Sankaralingam. Enabling GPGPU Low-Level Hardware Explorations with MIAOW: An Open-Source RTL Implementation of a GPGPU. *ACM Trans. Archit. Code Optim.*, 12(2):21:21:1–21:21:25, June 2015.

[19] Chris et al. Barton. The Multi2Sim Simulation Framework, A CPU-GPU Model for Heterogeneous Computing (For Multi2Sim v. 4.2), 2013.

[20] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[21] J. Bush, P. Dexter, and T. N. Miller. Nyami: a synthesizable GPU architectural model for general-purpose and graphics-specific workloads. In *Performance*

Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on, pages 173–182, March 2015.

[22] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013.

[23] D. Capalija and T. S. Abdelrahman. A High-performance Overlay Architecture for Pipelined Execution of Data Flow Graphs. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sept 2013.

[24] Hui Yan Cheah, Fredrik Brosser, Suhaib A. Fahmy, and Douglas L. Maskell. The iDEA DSP Block-Based Soft Processor for FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 7(3):19:1–19:23, September 2014.

[25] D. Chen and D. Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 297–304, Jan 2013.

[26] Christopher H. Chou, Aaron Severance, Alex D. Brant, Zhiduo Liu, Saurabh Sant, and Guy G.F. Lemieux. VEGAS: Soft Vector Processor with Scratchpad Memory. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 15–24, New York, NY, USA, 2011. ACM.

[27] Chen Chung-Shu. *Tutorial: Creating an LLVM Backend for the Cpu0 Architecture (v3.7.1)*. 2015. Avaialble on `http://jonathan2251.github.io/lbd` [Online; accessed 14-Feb-2017].

[28] K. Cooper and L. Torczon. *Engineering a Compiler*. Elsevier Science, 2011.

[29] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[30] Diego Valverde. Theia: Ray Graphic Processing Unit. Available on `opencores.com/project,theia_gpu` [Online; accessed 4-Jul-2016].

[31] R. Dimond, O. Mencer, and W. Luk. CUSTARD - a customisable threaded FPGA soft processor and tools. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 1–6, Aug 2005.

[32] H. Ding, S. Ma, M. Huang, and D. Andrews. OOGen: An Automated Generation Tool for Custom MPSoC Architectures Based on Object-Oriented Programming Methods. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 233–240, May 2016.

[33] Qianqian Fang and David A. Boas. Monte Carlo Simulation of Photon Migration in 3D Turbid Media Accelerated by Graphics Processing Units. *Opt. Express*, 17(22):20178–20190, Oct 2009.

[34] Blair Fort, Davor Capalija, Zvonko G. Vranesic, and Stephen D. Brown. A Multithreaded Soft Processor for SoPC Area Reduction. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '06, pages 131–142, Washington, DC, USA, 2006. IEEE Computer Society.

[35] J. Gray. GRVI Phalanx: A Massively Parallel RISC-V FPGA Accelerator Accelerator. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–20, May 2016.

[36] Jan Gray. Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip. In *2000. [Online]. Available: http://www.fpgacpu.org/papers/soc-gr0040-paper.pdf*, 2000.

[37] Andrei Hagiescu and Weng-Fai Wong. Co-synthesis of FPGA-based Application-specific Floating Point Simd Accelerators. FPGA '11, pages 247–256, New York, NY, USA, 2011. ACM.

[38] J. Handy. *The Cache Memory Book*. Morgan Kaufmann Series in Comp. Academic Press, 1998.

[39] Gopalakrishna Hegde, Siddhartha, Nachiappan Ramasamy, and Nachiket Kapre. CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-based Platforms. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, pages 14:1–14:10, New York, NY, USA, 2016. ACM.

[40] E. Homsirikamol and K. Gaj. Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–8, Dec 2014.

[41] A. K. Jain, D. L. Maskell, and S. A. Fahmy. Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs? In *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, pages 586–593, Aug 2016.

[42] A. K. Jain, D. L. Maskell, and S. A. Fahmy. Throughput oriented FPGA overlays using DSP blocks. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1628–1633, March 2016.

[43] Rui Jia, Colin Yu Lin, Zhenhong Guo, Rui Chen, Fei Wang, Tongqiang Gao, and Haigang Yang. A survey of open source processors for FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sept 2014.

[44] M. A. Kadi, P. Rudolph, D. Gohringer, and M. Hubner. Dynamic and partial reconfiguration of Zynq 7000 under Linux. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 1–5, Dec 2013.

[45] M. Al Kadi, M. Ferger, V. Stegemann, and M. Hübner. Multi-FPGA reconfigurable system for accelerating MATLAB simulations. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2014.

[46] M. Al Kadi and M. Huebner. Integer computations with soft GPGPU on FPGAs. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 28–35, Dec 2016.

[47] M. Al Kadi, B. Janssen, and M. Huebner. Floating-Point Arithmetic Using GPGPU on FPGAs. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 134–139, July 2017.

[48] N. Kapre and A. DeHon. VLIW-SCORE: Beyond C for sequential control of SPICE FPGA acceleration. In *2011 International Conference on Field-Programmable Technology*, pages 1–9, Dec 2011.

[49] Nachiket Kapre. Optimizing Soft Vector Processing in FPGA-Based Embedded Systems. *ACM Trans. Reconfigurable Technol. Syst.*, 9(3):17:1–17:17, May 2016.

[50] Nachiket Kapre, Nikil Mehta, Raphael Rubin, Henry Barnor, Michael J Wilson, Michael Wrighton, Andre DeHon, et al. Packet switched vs. time multiplexed FPGA overlay networks. In *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on*, pages 205–216. IEEE, 2006.

[51] J. Kathiara and M. Leeser. An Autonomous Vector/Scalar Floating Point Coprocessor for FPGAs. In *FCCM'11*, pages 33–36, May 2011.

[52] Khronos Group. OpenCL 1.2 Specification, 2012.

[53] J. Kingyens and J. Gregory Steffan. A GPU-inspired soft processor for high-throughput acceleration. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[54] Florain Kriete. Evaluation verschiedener Multicore Architekturen auf FPGAs. Master's thesis, Ruhr-University of Bochum, 2017.

[55] C. E. LaForest. *High-Speed Soft-Processor Architecture for FPGA Overlays*. PhD thesis, University of Toronto, 2015.

[56] C. E. LaForest and J. G. Steffan. Maximizing speed and density of tiled FPGA overlays via partitioning. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 238–245, Dec 2013.

[57] Charles Eric LaForest and John Gregory Steffan. OCTAVO: An FPGA-centric Processor Family. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 219–228, New York, NY, USA, 2012. ACM.

[58] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004.

[59] I. Lebedev, Shaoyi Cheng, A. Doupnik, J. Martin, C. Fletcher, D. Burke, Mingjie Lin, and J. Wawrzynek. MARC: A Many-Core Approach to Reconfigurable Computing. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 7–12, Dec 2010.

[60] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the

100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[61] C. LIU. *QuickDough : a rapid FPGA loop accelerator design framework using soft coarse-grained reconfigurable array overlay*. PhD thesis, University of Hong Kong, Pokfulam, Hong Kong SAR. Retrieved from `http://dx.doi.org/10.5353/th_b5760939`, 2015.

[62] Yaojie Lu, Seyedamin Rooholamin, and Sotirios G. Ziavras. Vector Coprocessor Virtualization for Simultaneous Multithreading. *ACM Trans. Embed. Comput. Syst.*, 15(3):57:1–57:25, May 2016.

[63] Eric Matthews, Lesley Shannon, and Alexandra Fedorova. Shared Memory Multicore MicroBlaze System with SMP Linux Support. *ACM Trans. Reconfigurable Technol. Syst.*, 9(4):26:1–26:22, August 2016.

[64] T. Miller. OpenShader: Open Architecture GPU Simulator and Implementation. Available on sourceforge.net/projects/openshader [Online; accessed 4-Jul-2016].

[65] R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, Aug 2012.

[66] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[67] NVIDIA. CUDA C Programming Guide, PG-02829-001_v8.0, 2016.

[68] NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. White Paper, 2009.

[69] NVIDIA Corp. GPU-Based Deep Learning Inference: A Performance and Power Analysis. White Paper, 2015.

[70] NVIDIA Corp. NVIDIA Tesla P100. White Paper (WP-08019-001 v01.1), 2016.

[71] Patrick Rudolph. Modellierung einer rekonfigurierbaren Hardware unter Linux. Bachelor thesis, Ruhr-University of Bochum, 2013.

[72] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, Sept 2013.

[73] Madhura Purnaprajna and Paolo Ienne. Making Wide-issue VLIW Processors Viable on FPGAs. *ACM Trans. Archit. Code Optim.*, 8(4):33:1–33:16, January 2012.

[74] A. Qamar, F. Muslim, F. Gregoretti, L. Lavagno, and M. T. Lazarescu. High-level Synthesis for Semi-global Matching: Is the juice worth the squeeze? *IEEE Access*, PP(99):1–1, 2016.

[75] R. Rashid, J. G. Steffan, and V. Betz. Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS. In *FPT'14*, pages 20–27, Dec 2014.

[76] M. Rosière, J. l. Desbarbieux, N. Drach, and F. Wajsbürt. An out-of-order superscalar processor on FPGA: The ReOrder Buffer design. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1549–1554, March 2012.

[77] K. Rupp. CPU, GPU and MIC Hardware Characteristics over Time. Online blog of Karl Rupp, available on `https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/`. [Online, accessed: 24-Jan-2017].

[78] A. Severance and G. Lemieux. VENICE: A Compact Vector Processor for FPGA Applications. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 245–245, April 2012.

[79] A. Severance and G.G.F. Lemieux. Embedded Supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013.

[80] Aaron Severance, Joe Edwards, Hossein Omidian, and Guy Lemieux. Soft Vector Processors with Streaming Pipelines. pages 117–126, New York, NY, USA, 2014. ACM.

[81] Volker Stegemann. Entwurf und Evaluation einer Multi-FPGA-Architektur zur Beschleunigung von Matlab-Simulationen. Master's thesis, Ruhr-University of Bochum, 2014.

[82] Martin Tatara. Entwicklung eines Treibers für eine Soft-GPU unter Linux. Master's thesis, Ruhr-University of Bochum, 2016.

[83] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[84] S. M. Trimberger. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331, March 2015.

[85] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[86] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[87] Kevin Winge. Entwurf und Implementierung eines Simulationsmodells für eine eingebettete GPU. Bachelor thesis, Ruhr-University of Bochum, 2016.

[88] H. Wong, V. Betz, and J. Rose. High Performance Instruction Scheduling Circuits for Out-of-Order Soft Processors. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, May 2016.

[89] Xilinx, Inc. Floating-Point Design with Vivado HLS, Application Note(XAPP599 v1.0), 2012.

[90] Xilinx, Inc. Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 All Programmable SoC Device, Application Note(XAPP1159 v1), 2013.

[91] Xilinx, Inc. AXI DMA, LogiCORE IP Product Guide (PG021, v7.1), 2015.

[92] Xilinx, Inc. Floating-Point Operator v7.1, LogiCORE IP Product Guide (PG060), 2015.

[93] Xilinx, Inc. 7 Series FPGAs Configurable Logic Block v1.8, (UG474), 2016.

[94] Xilinx, Inc. 7 Series FPGAs DSP48E1 Slice v1.9, (UG479), 2016.

[95] Xilinx, Inc. 7 Series FPGAs Memory Resorces v1.12, (UG473), 2016.

[96] Xilinx, Inc. MicroBlaze Processor Reference Guide (UG984, v2016.2), 2016.

[97] Xilinx, Inc. UltraScale Architecture and Product Overview (v2.10), DS890, 2016.

[98] Xilinx, Inc. Vivado Design Suite User Guide, Partial Reconfiguration (UG909, v2016.2), 2016.

[99] Xilinx, Inc. Vivado Design Suite, User Guide (UG902 v2016.2), 2016.

[100] Xilinx, Inc. ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC , User Guide (UG954, v1.6), 2016.

[101] Xilinx, Inc. Zynq-7000 All Programmable SoC, Technical Reference Manual (UG585, v1.11), 2016.

[102] Xilinx, Inc. SDAccel Development Environment Methodology Guide, Performance Optimization (UG1207, v2.0) , 2016 Ch.7.

[103] P. Yiannacouras, J.G. Steffan, and J. Rose. Exploration and Customization of FPGA-Based Soft Processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):266–277, Feb 2007.

[104] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Fine-grain Performance Scaling of Soft Vector Processors. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 97–106, New York, NY, USA, 2009. ACM.

[105] Jason Yu, Christopher Eagleston, Christopher Han-Yu Chou, Maxime Perreault, and Guy Lemieux. Vector Processing As a Soft Processor Accelerator. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):12:1–12:34, June 2009.