Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Philip Gausaker

A Coarse-Grain Reconfigurable Accelerator for Rocket

Master's thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Magnus Jahre
June 2022

NTNU
Norwegian University of
Science and Technology

Philip Gausaker

A Coarse-Grain Reconfigurable Accelerator for Rocket

Master's thesis in Electronic Systems Design
Supervisor: Per Gunnar Kjeldsberg
Co-supervisor: Magnus Jahre
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU

Kunnskap for en bedre verden

Master thesis - A Coarse-Grain Reconfigurable Accelerator for Rocket

Philip Gausaker

June 2022

Problem Description

The end of Dennard scaling and the imminent end of Moore's Law means that computer architects can no longer rely on technology scaling to deliver performance improvement. Instead, computer architects are exploring to relax the general-purpose one-size-fits-all paradigm to improve performance and efficiency. More specifically, architects are proposing specialized compute units, commonly called accelerators, that are (much) more efficient for the typical computations of a carefully selected domain than a general-purpose processor. Unfortunately, it is currently unclear (i) which types of accelerators should be included in a system, and (ii) how they should be integrated.

The objective of this thesis is to work towards answering these questions. The first step is to identify applications that are likely to benefit from acceleration. The student should use the MachSuite profiles he generated during the autumn project as a starting point. The student should then integrate a Coarse-Grain Reconfigurable Architecture (CGRA) generated with CGRA-ME with the Rocket processor using the RoCC interface and generate and evaluate an accelerator for a suitable application. If time permits, the student should accelerate more applications and explain the root causes of performance differences.

Abstract

Pursuing ever faster and more power-efficient computer architecture is becoming increasingly difficult. This results from the imminent end of Moore's Law and the end of Dennard scaling. Computer architects are looking for alternative ways to achieve faster and more efficient computation. Some are looking at accelerators that are specialized for specific calculations. Accelerators are extremely good and efficient, but the downside is that they are idling when the calculation is in a not-suited domain.

Architects are exploring reconfigurable accelerators such as Coarse-Grain Reconfigurable Architecture (CGRA) to minimize the time these accelerators are idling. CGRA consists of an array of Processing Elements (PE) that are capable of executing word-level operations and can be reconfigured to minimize idling time. These accelerators can adapt and accelerate several parts of an application. This thesis integrates a CGRA, generated by CGRA-ME, into the Rocket Chip System on Chip (SoC). Rocket Chip is an SoC generator for producing RISC-V SoC. Rocket-ME is the control module that controls the CGRA accelerator and communicates to the CPU and L1 cache through the unique Rocket Custom Co-processor (RoCC) interface in Rocket Chip. Rocket-ME is implemented in three different versions with different capabilities.

Depending on the type of integration, the results show that most integrations are faster at accumulating an array than the CPU. Still, the overhead from the configuration of the CGRA makes the accelerator's total time slower than the time of the CPU. Nevertheless, For the longest benchmark run with parallel computation, we achieved a total speedup of 2.6× compared to the baseline. This benchmark was run with an accumulation length that let the CGRA's fast computation overcome its significant configuration overhead.

Sammendrag

Søken etter raskere og mer energi effektive data arkitekturer blir stadig vanskeligere. Dette er et resultat av slutten på Moores lov og Dennard skalering. Data arkitekter ser alternative veier for å oppnå raskere og mer energi effektiv kalkulering. Noen ser på akseleratorer som er spesialisert på spesifikke kalkuleringer. Akseleratorer er ekstremt gode og effektive, men ulempen er at de ikke blir brukt når komputeringen er i et ikke egnet domene.

Arkitekter utforsker re-konfigurerbare akseleratorer slik som Coarse-Grain Reconfigurable Architecture (CGRA) for å minimere tiden disse akseleratorene ikke blir brukt. CGRA består av et nett av Prosesserings Elementer (PE) som kan utføre ord-nivå operasjoner og kan bli re-konfigurert for å minimere tiden den ikke blir brukt. Disse akseleratorene kan tilpasse seg og akselerere flere deler av en applikasjon. Denne oppgaven integrerer en CGRA, generert av CGRA-ME, inn i Rocket Chip System on Chip (SoC). Rocket Chip er en SoC generator for å produsere RISC-V SoC. Rocket-ME er kontroll modulen som kontrollerer CGRA akseleratoren og kommuniserer med CPU og L1 cache gjennom det unike Rocket Custom Co-processor (RoCC) grensesnittet i Rocket Chip. Rocket-ME er implementert i tre forskjellige versjoner med ulike egenskaper.

Avhengig av type integrasjon så viser resultatet at de fleste av implementasjonene er raskere til å akkumulere en rekke enn CPU 'en. På grunn av konfigurasjons forsinkelse i CGRA'en er totaltiden for akseleratoren tregere enn den for CPU. Likevel, for den lengste benchmark kjøringen med parallell komputering, oppnådde vi en total hastighetsøkning på 2.6× sammenlignet med CPU. Denne benchmark'en hadde en lang nok akkumulerings lengde til å undertrykke den betydelige konfigurasjons tiden.

Preface

This master thesis is the conclusion of a two-year study in "Electronic Systems Design" at the Norwegian University of Science and Technology. The assignment was given by the "Faculty of Information Technology and Electrical Engineering" under the supervision of professor Magnus Jahre and professor Per Gunnar Kjeldsberg.

This thesis had a previous project in the subject TFE4590. This project covered some of the same topics mentioned here, and sections of this project have been used in this thesis.

I want to thank my friends and family for their support during my academic years. I also want to thank my supervisor Magnus Jahre for being patient and enthusiastic about the assignment and for guiding me in the right direction.

Contents

Figures

Tables

Acronyms

ADRES Architecture for Dynamically Reconfigurable Embedded System.

ASIC Application Specific Integrated Circuit.

BAR Berkeley Architecture Research.

BHT Branch History Table.

BTB Branch Target Buffer.

CGRA Coarse-Grain Reconfigurable Architecture.

DFG Data Flow Graph.

DLP Data Level Parallelism.

DRF Data Register File.

DSA Domain Specific Accelerators.

DySER Dynamically Specializing Execution Resources.

FPGA Field Programmable Gate Array.

FU Functional Units.

HTIF Host-TARGET Interface.

ILP Instruction Level Parallelism.

ISA Instruction Set Architecture.

LRF Local Register File.

MMIO Memory-Mapped Input Output.

PE Processing Elements.

RAS Return Address Stack.

RoCC Rocket Custom Co-processor.

RTL Register Transfer Level.

SHA Secure Hashing Algorithm.

SoC System on Chip.

XML Extensible Markup Language.

Code Listings

Chapter 1

Introduction

1.1 Motivation

Computer architects have long been able to increase the performance of computers by technology scaling and the desire to fulfill Moore's Law [1]. However, as the size of the transistor is scaled down, so are the supply voltage and the threshold value for the transistor. As the transistor is not a perfect switch, some current will leak when turned off. The current leakage increases exponentially as the threshold value is scaled down. This results in the current leakage being a substantial portion of the power consumption and consequently lowering the power efficiency [2]. This is one of the problems that have made many computer architects look for alternatives to scaling to pursue faster and more efficient computing. Specialized hardware is one way to achieve this. The first GPU released in the 1970s was better at computing video graphics than the CPU and opened a new dimension to video games. This GPU was used to drive the display of an arcade game [3]. Since then, the GPU has found several use cases, such as machine learning, crypto mining, gaming applications, etc., and is several orders of magnitude faster at computing video than the state-of-the-art CPU.

A bioinformatics accelerator, called Darwin, is up to 15.000× faster than a CPU at reference-based long-read assembly [4]. However, Darwin is highly specialized in one domain and is not particularly good at computing other domains. This is the case for many Domain Specific Accelerators (DSA). One could integrate several DSA which allows the acceleration of several domains, but the cost of doing so would be inherently expensive and takes up a lot of area and power resources. To overcome this problem, some researchers have taken inspiration from Field Programmable Gate Array (FPGA) and its reconfigurability. Instead of a fine-grained architecture like the FPGA, they came up with the Coarse-Grain Reconfigurable Architecture (CGRA). The CGRA is built up of several Functional Units (FU) capable of performing word-level operations. In contrast, the FPGA is built up of a large number of logic gates capable of performing bit-level operations. Being coarse-grained gives less flexibility on what it can calculate, but it also gives a much lower configuration time. The CGRA can be clocked at around

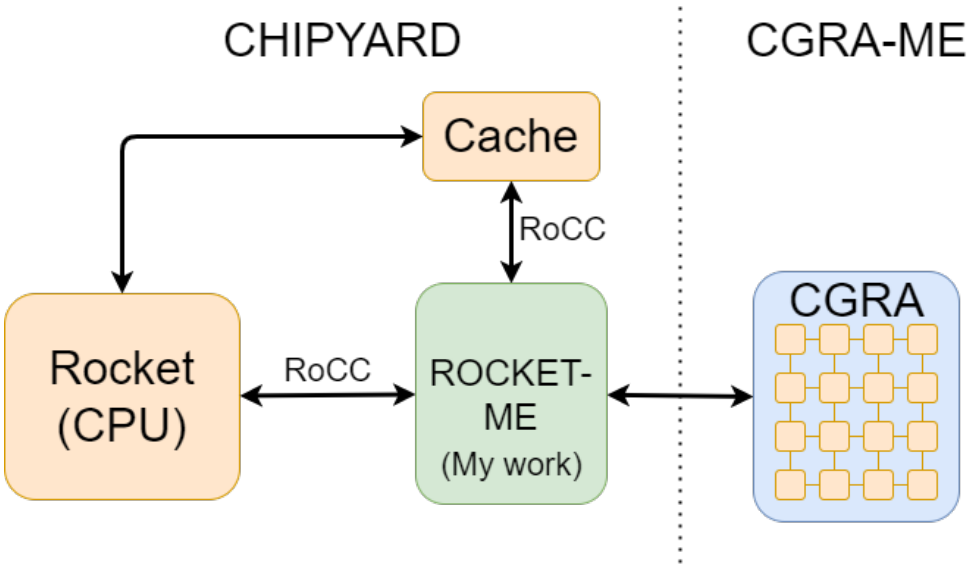


Figure 1.1: A high-level view of the system

1 GHz, whereas the FPGA typically operates at 200 MHz. This allows the CGRA to achieve higher throughput than the FPGA. The possibility of performing word-level operations and the higher clock frequency is two of the main benefits of the CGRA architecture. Being reconfigurable motivates its use as an accelerator as it can cover several domains.

CGRA's come in several different architectures and can be coupled to a processor in different ways. Some CGRA's are part of the execution stage of an architecture, and some are coupled as a co-processor that the CPU can offload certain regions onto to accelerate parts of an application. The RISC-V SoC generator Rocket Chip includes a unique interface called Rocket Custom Co-processor (RoCC) which enables the accelerator to communicate to the CPU and L1 cache and encourages the implementation of co-processors. The Rocket Chip generator is an open-source project which means using it in a project adds no extra licensing cost. This project will give experience with accelerator integration in an extensive system and will require insight into the working of the CGRA and the development environment for Rocket Chip.

Figure 1.1 represents a high-level view of the system. It shows the placement of the Rocket-ME module, which is the main contribution to the thesis, and the CGRA accelerator in relationship to the CPU and cache. The reason behind the figure is to give the reader an easier understanding of the work of this thesis. The Rocket and cache are produced in the Chipyard environment and connect to the Rocket-ME module via the RoCC interface. Rocket-ME is the interface and control module between the CGRA accelerator and the RoCC interface and was developed in the Chipyard environment. The CGRA is the only component produced by CGRA-ME in this figure.

1.2 Assignment Interpretation

The interpretation of the assignment can be different from person to person. It is therefore important to describe how we have interpreted the assignment. From reading the problem description, we have defined three main tasks. The tasks are:

1 Find applications that are likely to benefit from acceleration. The student should use the MachSuite profiles he generated during the autumn project as a starting point.
2 Connect a CGRA to the RoCC interface
3 If time permits, accelerate several applications with the CGRA.

Figure 1.1 shows the high-level result for Task 2. Connecting the CGRA to the RoCC interface can be divided into three smaller steps. Breaking the task down into several smaller steps may speed up the integration process and give a better understanding of the system. The smaller steps are:

2.1 Make a simple version of the interface that connects to the RoCC interface without the CGRA and do some simple operation.
2.2 Connect the interface to memory.
2.3 Connect the CGRA accelerator to the interface.

Task 1 was not addressable due to the difficulties of mapping the MachSuite benchmarks onto the architecture. CGRA-ME could not create a mapping for any of the MachSuite benchmarks as they consisted of several files and mostly contained branching in the for-loops. Having branches in the for-loop goes against the mapper limitation mentioned in Section 2.3.2. However, the task is open for interpretation, and the objective is not to accelerate the MachSuite benchmarks but to find an application that benefits from acceleration.

The primary goal of this thesis is to integrate the reconfigurable CGRA architecture into the Rocket Chip SoC and achieve a speedup for a given application with this implementation. This thesis focuses on performance gain and not energy efficiency or area usage. Task 1, 2, and 3 are what needs to be done to reach this goal.

1.3 Contributions

Task 1 is partly fulfilled as we have found an application that benefits from acceleration. The benchmark used to measure performance gain is the Sum benchmark and was included in the CGRA-ME environment. Sum accumulates an array and is simple enough to fit the small and larger coarse-grain architectures.

Task 2 and sub-tasks 2.1, 2.2, and 2.3 have been fulfilled with several accelerator implementations that utilize several architecture sizes. This thesis has integrated the Rocket-ME module, which attaches the CGRA accelerator to the CPU of Rocket Chip through the RoCC interface. The Rocket-ME module is developed by the author and is the main contribution to the thesis. The CGRA RTL is produced

by CGRA-ME and is slightly modified to fit the implementation.

We have successfully implemented three different architecture sizes and three different Rocket-ME modules. The total number of implementations is eight, as the smallest 2x2 architecture could not implement the parallel stream Rocket-ME implementation explained in Section 3.1.4. Chapter 5 presents the simulated results of the implementations. The plots show no total speedup for the benchmarks with a short array length due to a significant configuration overhead of the CGRA architecture. The result from the benchmark run with 9000 array elements shows that the best-performing implementation achieved a total speedup of 2.6× compared to the CPU. A common factor for the accelerators is that they are usually faster or as quick as the CPU at the computational stage, except for the "Buffer" implementations.

Task 3 in the assignment interpretation is somewhat fulfilled as the benchmark is made in 3 different versions, producing a different mapping in the CGRA architecture. One can argue for these being different applications.

1.4 Outline

The thesis is organized in the following way. The first chapter is this introduction, where we tell about the motivation behind the thesis and how the assignment is interpreted. Chapter 2 covers the necessary background and explains the development environment and simulation tools. Chapter 3 covers the implementation of the accelerator into the Rocket Chip SoC and some of the challenges of doing so. Chapter 4 cover the experimental setup. The simulated results are presented and discussed in Chapter 5. Chapter 6 covers the discussion part of the thesis, where the implementation and design choices are discussed. In Chapter 7 we conclude the results from the thesis and explain which implementation should be used in the integration of a CGRA into the Rocket Chip SoC. This chapter also contains some ideas that should be considered for further work.

Chapter 2

Background

This chapter covers the necessary background information needed to understand the implementation done in this thesis. The chapter is divided into several sections that cover different topics. First covered is the Chipyard environment and the different SoC generators included. The second section explains the Rocket Chip SoC generator. Section 2.1.2 covers the Rocket Core, which is the CPU used by default in the Rocket Chip SoC generator, and the special RoCC interface. How to create a new project in the Chipyard environment is explained in Section 2.1.3. The Verilator simulator is covered in Section 2.1.4 and an introduction to the RISC-V ISA is found in Section 2.1.5. A short explanation of Amdahl's law is found in Section 2.1.6. The accelerator used in the thesis is explained in Section 2.2 and some other CGRA architectures are explained in Section 2.2.1. Section 2.3 covers the CGRA-ME environment used to generate the CGRA RTL and configuration bitstream. An identification of the relevant background material was carried out in the project preceding this thesis [5]. Relevant information has been added as suited after the project end. The sections taken from the report are 2.1, 2.1.1 and 2.1.2.

2.1 Chipyard

Chipyard is a free open-source framework for developing Chisel-based System on Chip (SoC) [6]. Chisel is a hardware design language that adds hardware construction primitives to the Scala programming language. Chisel provides designers with a language that is highly parameterizable and encourages the reuse of circuit generators that produce synthesizable Verilog [7]. Chipyard was developed by Berkeley Architecture Research (BAR) group in the Electrical Engineering and Computer Sciences Department at the University of California [6]. It allows one to easily make a RISC-V SoC by using the power of the Rocket Chip SoC generator and other Berkeley projects. The Rocket Chip SoC generator is presented in Section 2.1.1. Chipyard was selected as the development framework because one can easily integrate custom accelerators and other peripherals into the SoC. It includes a useful set of toolchains and simulators. Chipyard contains several

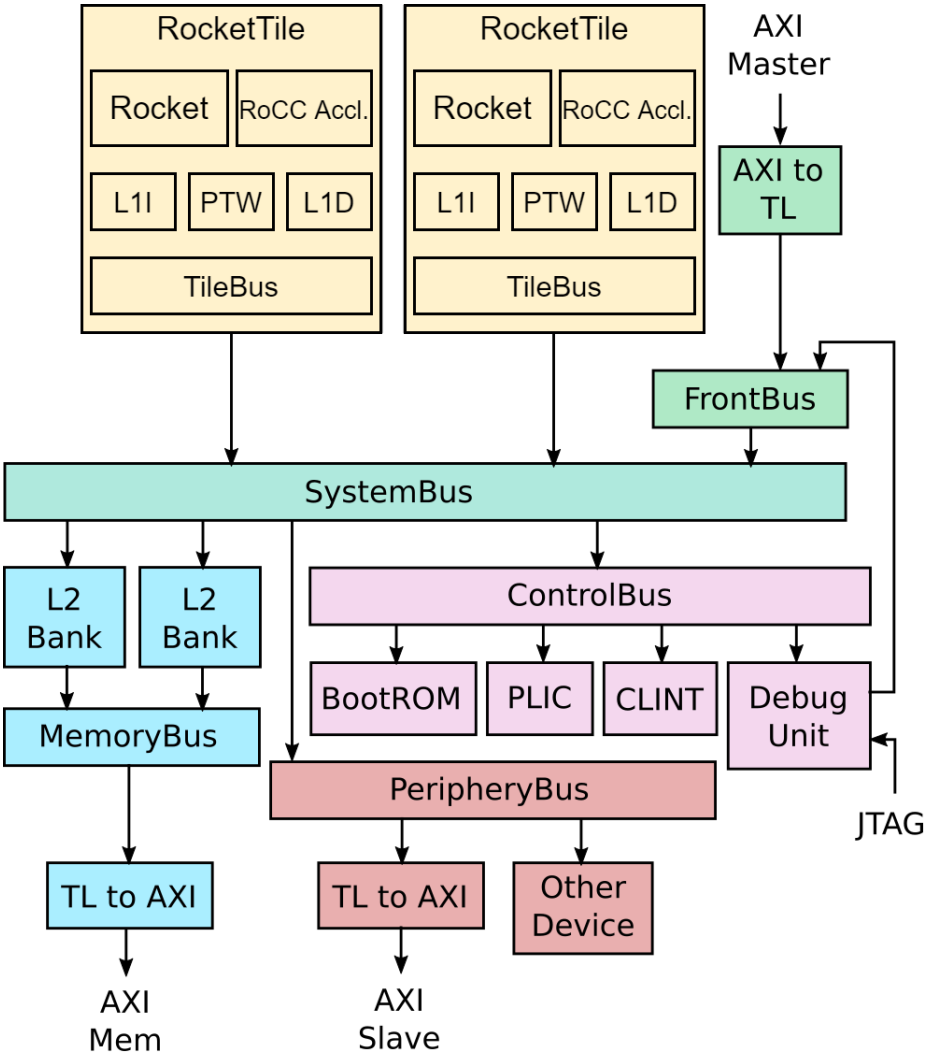


Figure 2.1: Illustration of Rocket Chip, inspired by [13]

RTL generators for CPU cores such as Rocket Core, Boom , and CVA6 [8]. It contains different memory systems and generators for accelerators such as SHA3 [9], Hwacha [10], Gemmini [11], and NVDLA [12]. Chipyard supports software RTL simulation, FPGA-accelerated simulation, and software workload generation for bare-metal and Linux-based systems.

2.1.1 Rocket Chip

Chipyard uses the Rocket Chip generator as default for producing a RISC-V SoC [13]. The Rocket Chip generator is an SoC generator developed at Berkeley and includes many parts of the SoC besides the CPU [8]. Rocket Chip uses Rocket Core, the in-order RISC-V CPU generator, as its CPU by default. Still, it can also be

configured to use the Berkeley Out-of-Order Machine (BOOM) generator or other custom CPU generator instead. An overview of Rocket Chip is shown in Figure 2.1.

The Rocket Chip CPU is contained inside a Rocket Tile. A Rocket Tile has a page-table walker, an L1 instruction cache, an L1 data cache, and a TileBus, which connects the back end of the caches to the system bus. This is also where a RoCC accelerator would be located. This allows the accelerator to easily communicate with the CPU and L1 Data cache through the RoCC. The RoCC interface is explained in Section 2.1.2. Further out, one can see the system bus that connects the Rocket Tile to other peripherals such as the L2 bank, control bus, and Peripery-Bus. The L2 cache is connected to the memory bus, which connects to the DRAM controller through a TileLink-to-AXI converter. The system has memory-mapped IO (MMIO) such as BootROM, Platform-Level Interrupt Controller (PLIC), Core-Local Interrupt (CLINT), and Debug unit through the system bus. The first stage bootloader and device tree are inside the BootROM. Linux uses the device tree to determine what other peripherals that are attached. Software interrupts, and timer interrupts for each CPU are controlled in the CLINT. The chip can be controlled externally through the debug unit, which can also pull data from memory or load instruction and data to memory. It is usually controlled through a standard JTAG protocol but can also be controlled through a custom DMI. The Pheripery-Bus exposes an AXI4 port but can also attach additional peripherals like the NIC and Block Device. The FrontBus can add DMA devices that read and write directly from the memory system. All these system components connect, making the Rocket Chip shown in Figure 2.1.

2.1.2 Rocket Core and Rocket Custom Co-processor (RoCC)

Rocket Core is an in-order scalar processor that provides a 5-stage pipeline and is the default CPU used by the Rocket Chip generator [14]. The Rocket Core is written in Chisel and implements the open-source RV64G and RV32G RISC-V instruction set. It includes an integer ALU and an optional FPU. It is provided with the accelerator interface RoCC. It also contains a memory management unit that supports non-blocking data cache, page-based virtual memory, and a front-end with branch prediction. A Branch Target Buffer (BTB), Branch History Table (BHT), and a Return Address Stack (RAS) provides the branch prediction, which is configurable.

Rocket Chip generator includes a special interface called Rocket Custom Co-processor (RoCC) interface[15]. This interface is a unique feature for the Rocket Chip SoC generator, making for an easy connection of co-processors into the SoC. This interface lets the accelerator communicate with the CPU and separately to the L1 data cache or the outer memory system via the system bus. This saves clock cycles as memory requests can be sent directly to the memory system without the interaction of the CPU. A visual representation of the RoCC interface is shown in Figure 2.2. The figure can be broken down into five parts. The first part is the

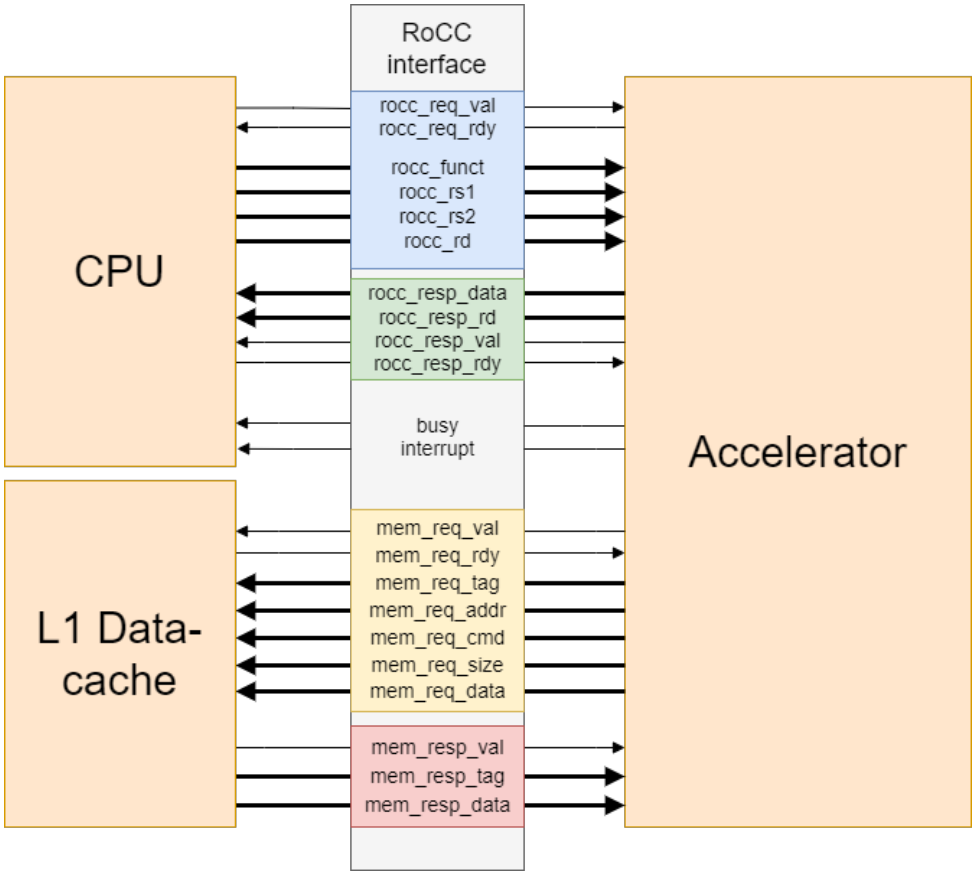


Figure 2.2: View of RoCC interface

commands sent from the CPU to the RoCC interface. The signals used for this are inside the blue box in the figure. The signals in the green box are used when the CPU expects a response from the accelerator. The signals in the gray box are used to tell the CPU that the accelerator is busy or to interrupt the normal execution in the CPU. The signals in the yellow box are used to make memory request, and the signals in the red box are the response line for the memory.

The Accelerator has a separate communication line to the L1 data cache. The L1 data cache work on a ready-valid basis. It has a request side, and a response side marked in yellow and red in Figure 2.2, respectively. The accelerator must set the request address, read command, tag, and data size when sending a memory request. The request is assumed to be received when the ready and valid signal is high at the same time. When the memory system has fetched the data, it assumes that the accelerator is listening and sends the data out on the memory response line while holding the data valid signal high with the appropriate tag for one clock cycle. The accelerator sets the memory address, write command, tag, data, and size when sending a write request. When the memory system has completed the write request, it will set the valid signal on the response line high for two cycles.

Custom RoCC command

To use the RoCC interface, one must include the RoCC opcodes in the program. The opcodes are defined in the "rocc.h" header file and give a set of 6 different opcodes. These instructions are non-standard ISA instructions reserved in the RISC-V ISA encoding space. The instructions have the following form:

customX, rd, rs1 rs2, funct

The X represents the accelerator instance with a number between 0 and 3. Each core can have up to 4 custom accelerator instances connected, controlled by custom instructions. The rd field is the register number of the destination register. The rs1 and rs2 field is the registers of the two 64-bit source registers. The funct field is a 7-bit integer used to distinguish different instructions sent to the accelerator. In the rocc.h file, there are different versions of this instruction depending on the number of source registers one wants to send and if there are expected values in return. If sending an instruction with a destination register, the CPU will busy-wait until the valid signal on the RoCC response line goes high. Therefore, it is crucial to understand the use of this function.

RoCC accelerators

There are several projects that utilizes the RoCC interface such as SHA3 [9], Gemmini [16], and Hwacha [10]. Hwacha is a vector processing unit that aims to be used in projects limited by their energy and power consumption. It connects to the BOOM or Rocket Chip through the RoCC interface and the main memory system through the system bus. It is included in the Chipyard framework as default and can be modified if necessary.

The Gemmini project is a RoCC accelerator that performs matrix multiplications. It Connects to the outer memory system by default to directly fetch and write data to the L2 memory. It includes two DMA engines, one for fetching data to Gemmini and one for writing the result back to memory. It is a part of the Chipyard ecosystem but can easily be edited to meet designers requirements for other projects.

Another project is the Secure Hashing Algorithm (SHA) 3 [17]. This project is included in the Chipyard framework and utilizes the RoCC interface. SHA3 is designed to communicate to the processor as simply as possible, uses only two RoCC instructions, and communicates directly to the L1 data cache. The only information the SHA3 accelerator needs from the processor is a pointer to the message, the address to store the resulting hash and the length of the message. When this is fed through the RoCC interface, the busy signal goes high, and the computation starts. The busy signal is set low again when the accelerator is done with the calculation.

2.1.3 Environment Setup

The Chipyard environment is used to integrate the accelerator into the Rocket Chip SoC. There are three basic steps for creating a new project with the RoCC interface [18].

1. Add the source code of the accelerator to a folder in "chipyard/generators". The heuristics of the folder has to be in the specific way: "chipyard/generators/<Your_project>/src/main/scala". If this heuristics is not present, the compiler won't find the files to create the Verilog design.
2. Create a new configuration that adds the accelerator to the system.
3. Add the accelerator project to the build system in 'build.sbt'. This tells the build system what to build and adds it to a top-level project.

After creating the project folder heuristic, one has to create a "mixin". A "mixin" tells the compiler what interface to couple the accelerator to. In this case, it is the RoCC interface. A "mixin" is shown in Code listing 2.1 and are located in the file <Your_project>.scala. It creates the class "With<Your_project>Accel" which extends the class "config". The mixin specifies that a Black box is used as a part of the architecture, and this is where the accelerator architecture is placed.

Code listing 2.1: Mixin configuration

```
class With<Your_roject>Accel extends Config ((site, here, up) =>{
    case <BlackBoxName>BlackBox => true
    case BuildRoCC      => up(BuildRoCC) ++ Seq(
    (p: Parameters) => {
      val Your_project = LazyModule.apply(
        new <Your_project>Accel(OpcodeSet.custom0)(p)
      )
      Your_project
    }
  )
})
```

The code in Code listing 2.2 is added to the build file to let the builder know where the project is located and the associated dependencies.

Code listing 2.2: Build file

```
lazy val <Your_project> = (project in file("generators/<Your_project>"))
  .dependsOn(rocketchip, chisel_testers)
  .settings(libraryDependencies ++= rocketLibDeps.value)
  .settings(libraryDependencies ++= chiselTestersLibDeps.value)
  .settings(commonSettings)
```

The source code for an implementation may consist of several files. It is common to have one file that takes care of the connections between the different modules, such as the connections to RoCC, and one file for the implementation's behavior. To let the compiler know where the project is, one must create a project class in the "RocketConfigs.scala" file. This file is located in "chipyard/generators/chipyard/src/main/scala/config/". This will be the top of the project, and depending on the configuration, it will look like the code in Code listing 2.3.

Code listing 2.3: Class creation

```
// DOC include start: <Your_project>Rocket
class <Your_project>RocketConfig extends Config(
  new <Your_project>.With<Your_project>Accel ++
  new freechips.rocketchip.subsystem.WithNBigCores(1) ++
  new chipyard.config.AbstractConfig)
// DOC include end: <Your_project>Rocket
```

From the lines in Code listing 2.3, one see that the configuration includes the class with the accelerator, a Rocket Chip with one big core, and the Chipyard abstract config.

2.1.4 Verilator

Verilator is an open-source Verilog simulator that can outperform many commercial simulators [19]. It is integrated into the Chipyard environment and supports ARM and RISC-V vendor IP Out-of-the-box. Verilator takes the Verilog code and compiles it into a fast optimized and optionally thread-partitioned model, which is in turn wrapped inside a C++/SystemC module. The performance result is a model that can execute over 10x faster than a standalone SystemC. By doing multithreading, one can achieve another 2-10x speedup. Another benefit of Verilator is that it performs cycle-exact simulations.

Verilator can run bare-metal programs by itself and is also capable of performing system calls with the help of a proxy kernel. The proxy kernel proxy the system calls to the host over a Host-TARGET Interface (HTIF) [20]. The system calls in an application are sent over to the host computer while the simulator waits for a response in a busy-waiting manner. The proxy kernel instructions are executed in supervisor mode and are easily distinguished from other instructions when looking at the program counter value.

When creating a Verilator simulator, one can choose to make a clean version that simulates without any extra information, or one can make a debug version that gives several options for retrieving data. The debug version enables the simulator to print out cycle-exact information such as program counter, instruction value, and read and write registers. It also allows the creation of a waveform diagram of the whole simulation.

2.1.5 RISC-V an Open ISA

RISC-V is an open standard Instruction Set Architecture (ISA) released in 2010. The start of the RISC-V instruction set was driven by Prof. Krste Asanovic and graduate students Yunsup Lee and Andrew Waterman as a part of the Parallel Computing Laboratory at UC Berkley [21]. Its motivation was to support research and education. It is now managed by the RISC-V Foundation, which is a non-profit corporation controlled by its members to drive the initial adoption of the RISC-V ISA. Today many processors can run RISC-V ISA, i.e., Rocket core, SiFive, etc. RISC-V is mainly used for research purposes but is increasingly drawing attention

Table 2.1: Table of standard extensions and the operations they provide

Prefix	Provide instructions for
I	integer computation, load/store and control flow instructions
M	multiplication and division
A	atomic instructions
F	single-precision floating point
D	double-precision floating point
G	Includes extension I,M,A,F and D
Q	quad-precision floating point
L	decimal floating point

from the industrial side. One example of this is the implementation of RISC-V in a data center used to accelerate AI workloads. This work was done by Esperanto Technologies [22]. Since RISC-V is open-source, there is no need for expensive compiler support, which motivates the development of RISC-V processors.

The RISC-V ISA is divided into integer bases with support for 32-bit and 64-bit as the most common and 128-bit as an option. The "base" integer instruction set consists of 47 instructions required to move memory, perform computations, and arithmetic.

The base integer ISA can be extended with one or more optional instruction-set extensions providing different functionality. These can be divided into two groups, standard and non-standard extensions. Standard extensions contain instructions one generally would expect to find in an ISA and should not conflict with other standard extensions. Non-standard extensions are more specialized instructions and may conflict with other standard or non-standard extensions [23].

The RISC-V Foundation believes that it is important to simplify the required portion of an ISA specification. They aim to keep the base and each standard extension constant over time, whereas other architectures will change to a new version with added instructions. Instead of making new versions of the whole ISA, RISC-V will add new instructions as part of optional extensions. Another reason to keep the ISA simple is to keep the complexity of the hardware needed for an implementation to a minimum.

The RISC-V Foundation assumes that there will be created many non-standard extensions for different purposes and that some of these may end up as standard extensions over time. Therefore, developers are encouraged to make new extensions if they can't find one that suits their needs.

Some of the most common and useful extensions are listed in Table 2.1. The most common standard extensions, "IMADF", are collected into a new standard extension indicated with "G". The resulting ISAs are then called RV32G for the 32-bit version and RV64G for the 64-bit version. The RoCC instructions used in this thesis are custom non-standard ISA instructions reserved in the RISC-V ISA encoding space.

RISC-V features four software privilege levels, more commonly called RISC-

V modes. These modes are Machine mode (M-mode), Hypervisor mode (H-mode), Supervisor mode (S-mode), and User mode (U-mode). Machine mode is the highest privilege mode and the only required mode. It has access to all hardware, control, and status registers and runs in physical memory only. It has no restrictions on the CPU. A RISC-V implementation may only include machine mode, but this will not protect against incorrect or malicious application code. Therefore, one can add more levels such as user mode and supervisor mode to make the system more secure.

User mode is usually added to protect the system from the application code. It is in this mode that the application gets executed. This is also the least privileged mode. Supervisor-mode adds protection between the user mode and the machine mode. At this level, the supervisor programs like Linux will execute. The last mode is hypervisor mode which is the second most privileged mode. This mode is used when doing hardware virtualization.

2.1.6 Amdahl's Law

Amdahl's law is a computational law presented in 1967 by computer scientist Gene Amdahl [24]. The law gives us an equation on the theoretical speedup when using multiple processors. By using the Equation 2.1 one can calculate the theoretical speedup before doing the actual parallelization and analyze whether the effort is worth it or not. To calculate the theoretical speedup, one need to split the execution time into two parts, the part that does not benefit from parallelization and the part that does benefit from parallelization.

$$S = \frac{T_{Serial}}{Parallelpart * T_{Serial}/p + (1 - Parallelpart) * T_{Serial}} \tag{2.1}$$

Let's say 90% of the application can benefit from parallelization and that 10% is inherently serial. If the serial, or total, run-time is T(serial) = 30 seconds, the parallelized part will be 0.9 * T(serial)/p = 27/p, where p is the number of cores available. The unparallelized part will then be 0.1 * T(serial) = 3. Combining these two times gives the overall parallel run-time: T(Parallel) = 27/p + 3. The theoretical speedup will now be:

$$S = \frac{T_{Serial}}{0.9 * T_{Serial}/p + 0.1 * T_{Serial}} = \frac{30}{27/p + 3} \tag{2.2}$$

If p is set to a very high number, the expression gets independent of 27/p, which equals 0. This shows that the highest possible speedup for this application is

$$S \leq \frac{30}{3} = 10.$$

Amdahl's law says that it is not possible to get a speedup of more than ten by using parallelization for this application. This is essentially the time it takes to execute the unparallelized part of the code. In short, Amdahl's law says that the

total execution time of an application can not be faster than the time it takes to execute the unparallelizable part of the application.

2.2 Coarse-Grain Reconfigurable Architecture

As we are at the end of Dennard's scaling and the imminent end of Moore's Law, computer architects are exploring alternative forms of achieving faster computation. Some explore new groundbreaking technology while others look past and look at some existing architectures with fresh eyes. Among these existing architectures are reconfigurable architectures [25]. The motivation behind the development of CGRA is the end of both Dennard's scaling and Moore's Law. To keep up performance scaling, architects have begun to explore reconfigurable systems where CGRAs seemingly have found a good balance between programmability and performance. CGRA's have got inspiration from FPGA, but where FPGA performs bit-level operation, CGRA's are capable of performing word- and sub-word operations. Where FPGA's can literately be configured to perform every operation imagine, we have restrictions on how we can configure a CGRA. We minimize the scope of programmability over faster configuration times. Therefore, we are bound to simplistic operations such as add, shift, multiply, etc.

On the programmability scale, the CGRA is placed between FPGA and ASIC [26]. CGRA contains large coarse-grained logic blocks with many inputs and outputs and can perform complex ALU-like functions. This is different from FPGA, which is fine-grained. The interconnect of CGRA is datapath-like, where entire busses of signals are routed together in tandem rather than individual logic signals. CGRAs are in the wind and are becoming increasingly used to realize Domain Specific Accelerators (DSA).

2.2.1 CGRA types

There are several different designs to CGRA which determine the performance, area, and power consumption. Ranging from the earliest KressArr [25] to the more modern and familiar ADRES. This subchapter will explain the ADRES[27], DySER[28], and TRIPS/EDGE[29] architecture and their difference. The coupling between CGRAs and the processors is often loose and exposed for communication overhead.

Architecture for Dynamically Reconfigurable Embedded System

The Architecture for Dynamically Reconfigurable Embedded System (ADRES) CGRA is a widespread architecture template for embedded architectures and is one of the most cited architectures [30]. It is an XML-based template architecture that allows the user to define arbitrary connections between the process elements (PE), while the most common configuration is to arrange the PEs in a mesh. Figure 2.3

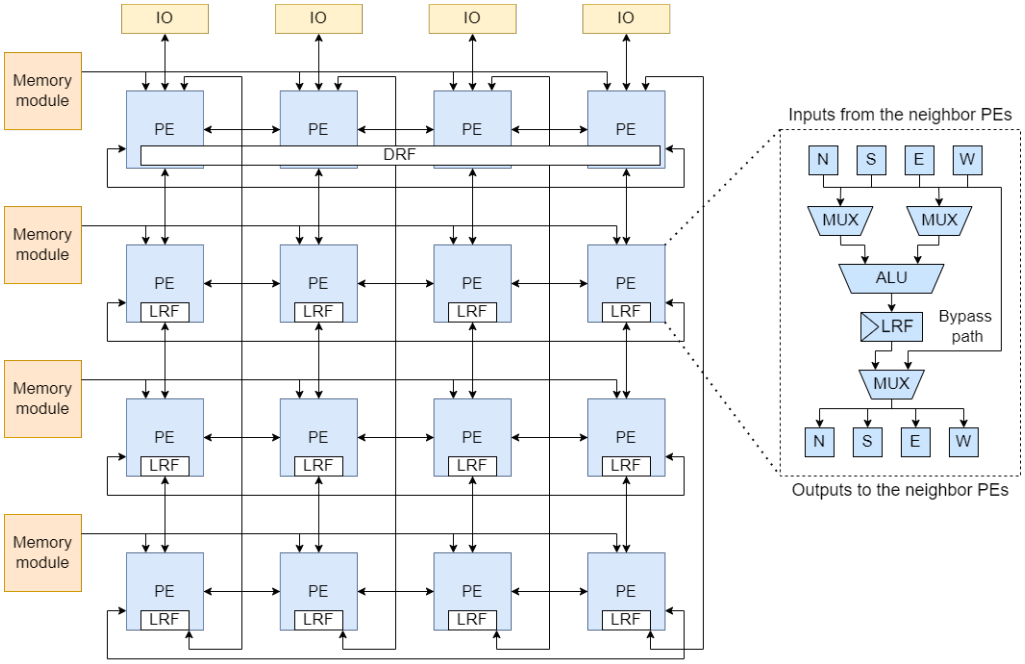


Figure 2.3: Display of the ADRES architecture, inspiration by [30]

shows a 4x4 ADRES CGRA. The figure shows only one of many possible configurations of the architecture. A 4x4 architecture has 4 columns and 4 rows of PE. The first row of PEs shares the same Data Register File (DRF), but PEs in the other rows have their own Local Register File (LRF). On the left-hand side of the figure, there are 4 memory modules. They are connected to one row each and are responsible for reading and writing to the memory. Each PE in the architecture is connected to its 4 North-South-East-West (NSEW) neighbors. They also connect to their diagonal neighbor, but this is not shown in Figure 2.3 for clarity. The PEs on the edges of the architecture is connected with their wrap-around neighbors (top/bottom, left/right). The first row in the architecture is unique as it can either be connected to the pipeline of a VLIW-processor or be a separate co-processor receiving inputs and outputs. A version of ADRES has been taped out on silicon, i.e., in the Samsung Reconfigurable Processor (SRP) [31].

A configuration bitstream is clocked through the architecture, which decides the behavior of the CGRA. The configuration bitstream is made by a mapper in compile-time and is loaded onto the CGRA architecture in run-time. All the PEs share the same configuration line and are connected in series. This means that the configuration bitstream size increases linearly with the number of PEs in the architecture.

The inner working of a PE is shown on the right-hand side of Figure 2.3. The two upper muxes take in values from neighboring PE and send them to the ALU, depending on the configuration. The ALU is capable of several operations such as add, multiply, subtract, shift, and logical operations. The computed result from the

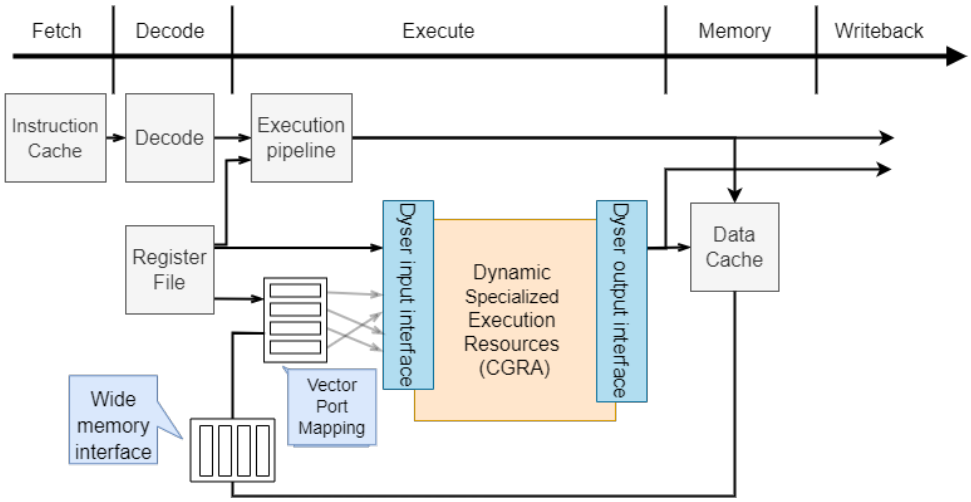


Figure 2.4: DySER, inspired by [28]

ALU is stored in the LRF. Another mux decides what value will be delivered to the neighboring PE. How the configuration bitstream that determines the behavior of the CGRA is created is described in Section 2.3

DySER

Dynamically Specializing Execution Resources, or DySER[28], is the name of a more tightly integrated CGRA. Instead of having the CGRA as a co-processor, they implement it as a part of the executing stage along the CPU. The main idea behind DySER is that programs execute in phases and that only a few of these phases contribute to most of a program's execution time. Dynamically creating specialized data paths for these phases can eliminate overhead and reduce energy consumption. They are also utilizing the processor's memory system by using its prefetcher, cache, and memory disambiguation to overcome load/store serialization bottlenecks in irregular code. The results from the report are that it outperforms an out-of-order (OOO) processor by 1.1x to 4.1x while simultaneously reducing the energy consumption by 9 percent [28].

The CGRA Architecture in DySER is much like the ADRES architecture, besides being more tightly integrated and having a much lower configuration time of 64 cycles. The compiler in DySER analyzes the c code and makes "mage-functions" executed on the CGRA. The CPU in DySER takes care of the code that the CGRA can't deal with and works as a store/load engine while the CGRA is computing. A figure of DySER is shown in Figure 2.4. The figure shows how DySER is integrated into the execution stage.

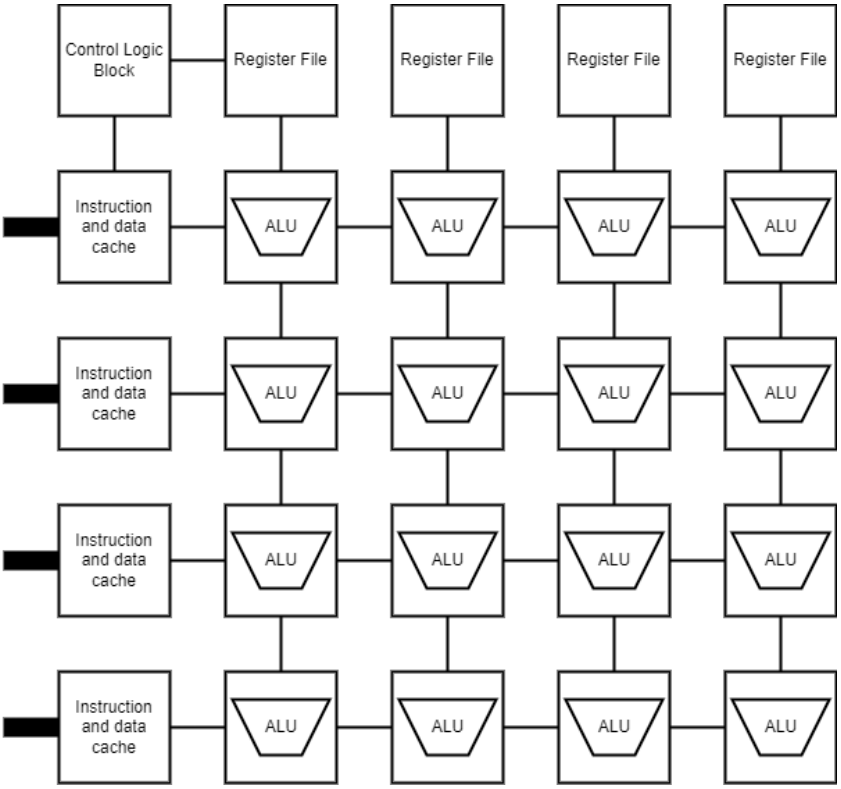


Figure 2.5: TRIPS/EDGE architecture, inspired by [25]

TRIPS/EDGE

The TRIPS/EDGE project was a long-running project where they moved away from the traditional way of exploiting instruction-level parallelism (ILP) in modern processors [25]. TRIPS/EDGE was used to replace the traditional superscalar Out-of-Order pipeline with a large CGRA array. The developer changed the structure so that instead of scheduling a single instruction, they scheduled entire blocks (CGRA configurations) on the processor. They did this by developing a new compiler. This allowed the TRIPS/EDGE architecture to execute up to 16 instructions at a single time.

As the technology reduced the size of transistors, wire delays became one of the main bottlenecks for the performance gain. The premise behind TRIPS/EDGE is to couple a memory unit close to the functional unit so that the wire latency would no longer be the limiting factor. The TRIPS/EDGE compiler examines the application for code that share information in a specific way and organize the instructions in blocks that can execute independently of other blocks. The data needed to execute these instruction blocks are stored in memory banks. These blocks can then be executed on the CGRA-like TRIPS/EDGE architecture shown in Figure 2.5. A prototype of the architecture was made and reached a performance of 500 gigaflops at a 500 MHz clock rate [29].

CGRA architecture comparison

The TRIPS/EDGE and DySER architecture is not template-based, meaning there is only one configuration. Still, the TRIPS/EDGE architecture can be expanded in width and depth, allowing for more ALU units and more advanced features.

All three projects implement the idea of a reconfigurable processing unit, but in different ways. One of the main differences between the ADRES, DySER, and TRIPS/EDGE architectures is how they deal with memory. TRIPS/EDGE have memory units close to the functional units, lowering the fetching latency. In contrast, ADRES has the option to integrate a memory unit in every row in the architecture. DySER uses the processor's memory unit, making it as fast as the CPU when it comes to fetching latency. ADRES has register files in each PE which only can store intermediate results. DySER does not implement such registers and only stores data when there are valid results on the output port of the architecture. The same accounts for TRIPS/EDGE. For a 4x4 configuration, TRIPS/EDGE can load or store up to 16 different data addresses, whereas the ADRES can only load or store four data addresses simultaneously. DySER is limited to the capabilities of the CPU's memory system.

2.3 CGRA-ME

CGRA-ME is an architectural and exploration open-source framework developed by the "Department of Electrical and Computer Engineering" at the University of Toronto [26]. An architect describes the CGRA architecture in an XML-based language, and this architecture is pared to CGRA-ME, which converts it into an in-memory representation. CGRA-ME includes some predefined CGRA architectures that are easy to utilize. CGRA-ME can map benchmarks onto the CGRA to assess the performance and area in different domains. CGRA-ME also includes automatic generation of Verilog RTL for the CGRA configuration. This makes it easier for architects to simulate the design before synthesizing it to ASIC or FPGA. Options can be added to the CGRA-ME mapper to make it produce a testbench with the configuration bitstream for the specified application and CGRA architecture.

2.3.1 Mapping

The CGRA needs to know what it should do precisely, and for that, it requires a configuration bitstream that can be loaded onto the CGRA architecture. CGRA-ME can create this configuration bitstream and mapping. When mapping an application, it has to include certain tags to let the mapper know what to map. This tag is called a loop tag and is placed at the start of a for-loop that one wants to map. It is possible to have several loop tags in the code, making the mapper produce several mappings. The source code will also have to meet the source code requirements listed in Section 2.3.2.

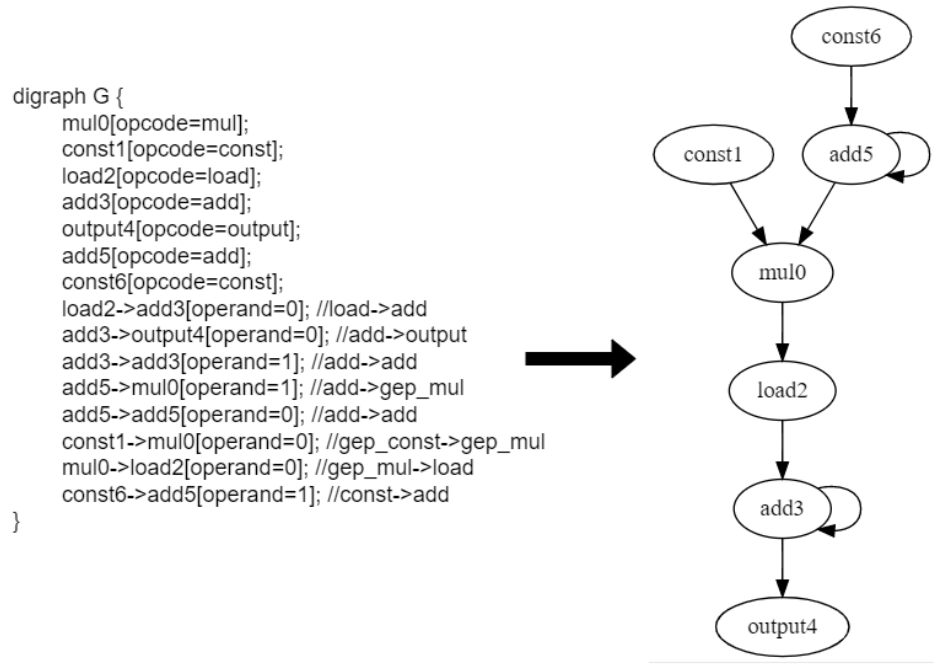


Figure 2.6: DFG code and visual representation of a benchmark produced by CGRA-ME

Invoking the CGRA-ME mapper will create a mapping specific to the application and the architecture. The mapper will produce a Data Flow Graph (DFG) representing the mapping. A DFG and a visual representation of it are shown in Figure 2.6. It also produces a Verilog testbench with the configuration bitstream. The DFG is a high-level description of the produces testbench. The mapper will try several different mappings before choosing the one that fits the architecture best. If the application is not suitable for mapping, the mapper will not produce any mapping. However, when the mapper produces an output, it will not care about constants and memory addresses. These will always be set to the default value of 0x80000001 in the testbench. Therefore, the constants must be set manually after the mapper has produced the testbench.

The mapper also produces a file called "stdout.txt" when doing a mapping. In this file, the mapper explicitly shows where the different nodes in the DFG are mapped to in the physical architecture.

Code listing 2.4: For-loop

```
for (int i = 0; i < N; i++) {
    //DFGLoop: loop
    sum += a[i];
}
```

Figure 2.6 is the DFG mapping representing the for-loop in Code listing 2.4. The figure on the right side of Figure 2.6 is the visual representation of the DFG code on the left-hand side of the figure.

The working of the DFG is as follows. In Figure 2.6 there are 2 constants, const1, and const6. Together with add5 and mul0, these constants are used to generate the address to load2. const6 goes into add5, where it gets added with itself. This result then goes into mul0, where it gets multiplied with the value in const1. This value goes into load2. load2 is connected to the memory module in the architecture, and the value sent into load2 is used as a memory address when making memory requests. The received data from memory is then sent to add3, where it is added with itself before it is sent to output4. add3 is here the accumulation that happens inside the for-loop.

2.3.2 Limitation

CGRA-ME cannot map every application onto the CGRA because of certain limitations in the compiler and architecture. The limitations can be divided into three groups. From the CGRA-ME documentation, these are [32]:
DFG Generation Limitation:

1. If there are nested loops, only the innermost loop will be accelerated. So, if you tagged the outer loop, it will not generate DFG.
2. You can only have one basic block within the loop body, meaning you can not have any branching in your loop body. (If statement, switch statement)
3. You must follow the User Guide to make sure all of your functions are properly inlined or not-inlined, otherwise, there is undefined behavior.
4. DFG generation has been having issues with structs, this has been noticed and will be fixed.

Mapper Limitation:

1. The mapper currently only generates the datapath portion of the mapping.
2. DFGs with cycle (caused by back-edges) that have a length larger than one will not be mapped correctly.
3. The mapper will not work correctly if the DFG provided has imbalanced path within.
4. There is behaviour differences between the ILP Mapper and the Simulated Annealing Mapper, the result from ILP should be the prefered one.
5. The Simulate Annealing Mapper needs a slower schedule for DFGs that have back-edges.

Source code Requirement:

1. This DFG generation software can only support single C source file with header files. Make sure every thing that you want to map to a single source file.
2. All the functions that contains within your loop of interest should be inlined.

You must add __attribute__((always_inline)) to the functions and put the definitions in the corresponding header file.

3. All the functions that contains your loop of interest should not be inlined. You must add __attribute__((noinline)) to the functions and put the definitions in the source file.

Chapter 3

Implementation

This chapter covers the work done to successfully implement the CGRA accelerator in 8 different Rocket-ME implementations into the Rocket Chip SoC generator. It also covers some of the challenges met during the implementation and how they were solved. Section 3.1 covers how the different versions of the implementation work. Section 3.2 covers how the CGRA mappings of the benchmark are created. The software and hardware flow of the system is covered in Section 3.3 and 3.4. The discussion part of the implementation is placed in Chapter 6.

3.1 Integrating the CGRA

Several code versions of the implementation have been made from the work of this thesis. The first successful code version of the implementation was done without the CGRA being connected to Rocket-ME. This version covers Task 2.1 and 2.2 in the assignment interpretation and had a simple goal. To receive three different variables and do a simple operation. Two of these were the memory addresses to source registers, and the last was the memory address to a destination register. The goal was to add the value of the two source registers and store the result in the destination register. This version talked directly to the CPU and memory system on two different lines. An illustration of this implementation is shown in Figure 3.1. When this version was successfully implemented, we moved over to integrating the CGRA into the code. To create the project in the Chipyard environment, we followed the approach described in Section 2.1.3.

The RTL for the CGRA accelerator is produced by the CGRA-ME environment and was integrated into the Rocket Chip SoC as a black box. By integrating it as a black box, the Chipyard system only knows the inputs and outputs of the CGRA design instead of the inner working. The Verilog files were generated based on the architecture described in an XML-based language in the CGRA-ME environment. While very useful, it also came with some challenges. Some of the naming for the different modules were erroneous and made compiler errors. These naming errors needed to be corrected before the CGRA design could be compiled and simulated.

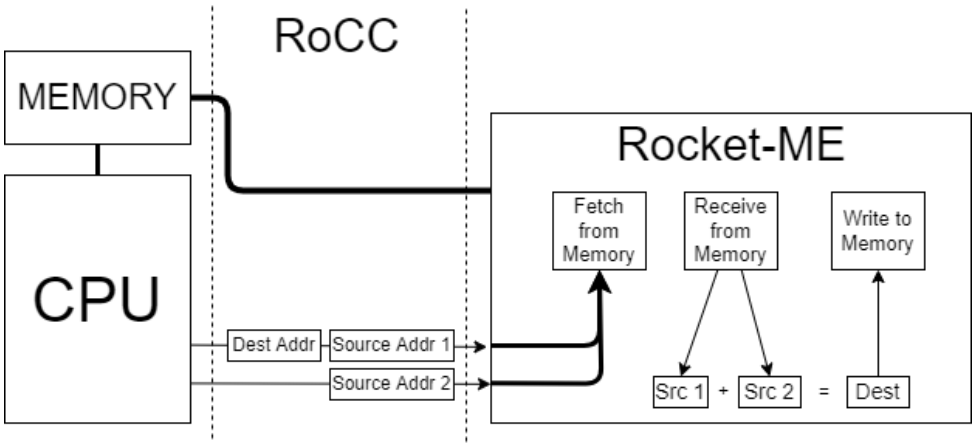


Figure 3.1: Illustration of the first implemented Rocket-ME.

The automatically generated RTL from CGRA-ME contained an empty memory module. This empty memory module is included to make it easier for the developer to implement their memory controller. In this design, the signals from the memory module are moved out to interface directly with Rocket-ME. By doing this, Rocket-ME will do the memory transactions for the CGRA accelerator.

Before integrating the CGRA into Chipyard, the design was hooked up to a testbench in Vivado[33] to see how it behaved. Running the testbench gave valuable insight into the working of the CGRA and provided an understanding of what to expect when operating it through Rocket Chip. However, while using the testbench, some strange behavior by the CGRA was noticed. The configuration bitstream went too quickly through parts of the CGRA, and the initial thought was that it was a timing problem, but after some debugging, the error was found. The CGRA-ME RTL generator had used blocking assignment instead of non-blocking assignment in the cells that held the configuration. Changing this to a non-blocking assignment fixed the problem, and the CGRA behaved as expected. When all the errors were sorted out, it was time to implement the CGRA in the Rocket Chip SoC.

Adding the Verilog files in chisel was a trivial experience. One has to define a class, add the inputs and outputs of the Verilog design in the class, and link them by using the function addResource() inside the class. This class was then added as a module in Rocket-ME. The code can be found in the Chipyard repository on GitHub [34].

The Verilog design for the 4x3 architecture initially had four bidirectional ports where the input/output of the CGRA is communicated. As Chisel is not good with bidirectional ports, extra ports were added to the design together with their own enable signal. The design where changed so that it has four input and four output ports together with enable/write signals. The outputs are used by most applications, while few are using the input ports. The direction of the ports is dependent on the mapping produced by CGRA-ME. This modification also accounts for the

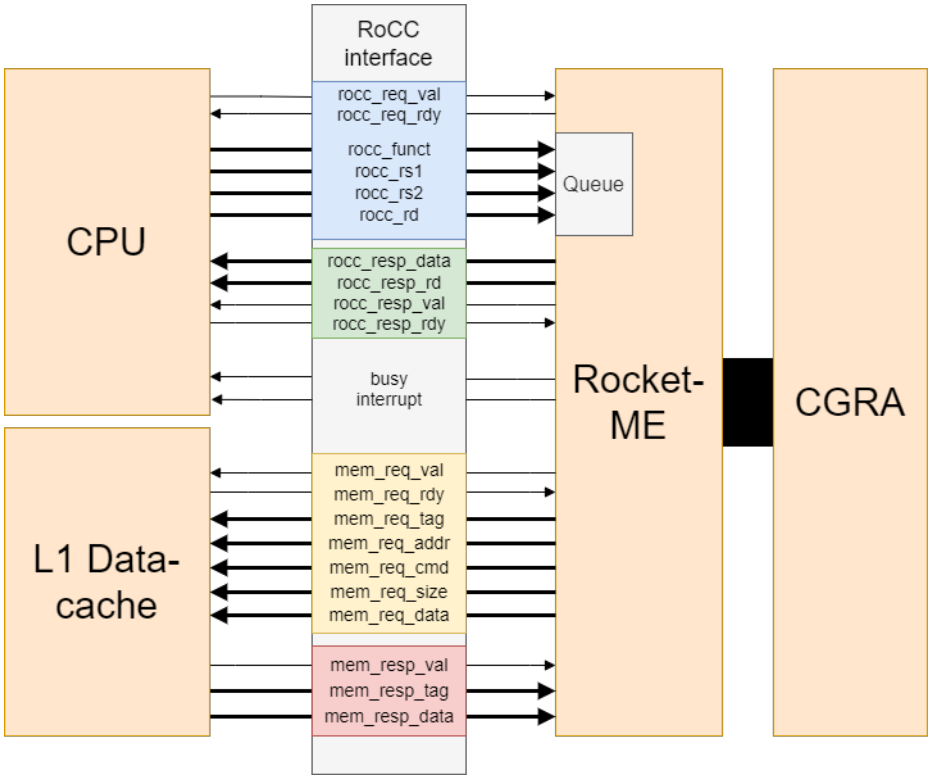


Figure 3.2: Overview of the RoCC interface with Rocket-ME

2x2 and 6x6 architecture sizes. However, the 2x2 and 6x6 architecture had 2 and 6 input/output ports, respectively.

3.1.1 Rocket-ME

Rocket-ME is the controller that sits between the RoCC interface and the accelerator and has the task of controlling the accelerator and communicating with the CPU and L1 cache. A figure of Rocket-ME and its placement relative to the CGRA and RoCC are shown in Figure 3.2. The CPU communicates with Rocket-ME through custom instructions sent via the RoCC interface. These instructions go into a queue in Rocket-ME and are handled when Rocket-ME tells the queue that it is ready. When programming the CGRA, the CPU sends the configuration bitstream 128 bits at a time over to Rocket-ME via the RoCC. Rocket-ME reverses the bitstream and puts it into a configuration array. When the whole configuration bitstream is sent over, Rocket-ME will start to configure the CGRA. Rocket-ME does also control the CGRA clock and the configuration clock. The configuration clock is used to clock the configuration bitstream through the CGRA architecture, and the CGRA clock is the clock for the data registers in the architecture. The configuration clock and CGRA clock start when the first bit in the bitstream is sent to the CGRA. When all the configuration bits are sent over, it stops the configur-

ation clock and the CGRA clock. Rocket-ME now waits for the input and output addresses and the compute length from the CPU. Rocket-ME is set up in a specific way, so when it receives the compute length, it also starts the computation. Therefore, the order in which the instructions are sent to Rocket-ME is important. An illustration of the flow of the system is shown in the setup part of Figure 3.6 and 3.7.

When Rocket-ME configures or monitors the CGRA, it sets the busy signal high, indicating operation. Since the RoCC interface works in a ready-valid basis, this typically means that the CPU needs to wait for Rocket-ME to be ready before it can send the custom instructions and do any more work. However, Rocket-ME implements a queue that handles the commands sent from the CPU. This means that instructions sent to Rocket-ME can be transmitted even though the busy signal is high. The instructions are now placed in the queue and sent to Rocket-ME when it is ready to receive the instruction. Implementing the queue is beneficial as it can hide some latency of the different stages.

Rocket-ME is made in 3 different versions and was constructed using the SHA3 accelerator as a RoCC- and memory implementation template. Using the SHA3 RoCC accelerator as our template made sense as we wanted an interface that was as simple as possible and needed a guide for the integration. The first version is the "buffer" version and is explained in Section 3.1.2. The second and third versions are streaming versions that have two different versions with small but significant differences. These are explained in Section 3.1.3 and 3.1.4. The reasoning behind the three different implementations is to see how the performance was affected by the Rocket-ME module.

3.1.2 Buffer

When the development of Rocket-ME started, we decided to make an interface with a buffer. This was because the CGRA generated memory addresses faster than Rocket-ME was able to fetch data from memory. A visual representation of the workflow of the implementation is shown in the "interaction with CGRA" part of Figure 3.6 and it works in the following way. First, Rocket-ME gets a set of addresses from the CGRA and stops the CGRA clock. It then fetches the data associated with the memory addresses from memory. When all the data is collected, it will start the CGRA clock and send the data elements one by one into the CGRA. This ensures correct execution in the CGRA and will keep the latency combined with fetching data to a minimum as it can have several memory requests in flight. However, as the length of the computation increases, the size of the architecture also increases as it needs available buffer slots for both the memory address and the memory data. The design is not very efficient since it will do the operations in sequence instead of in parallel.

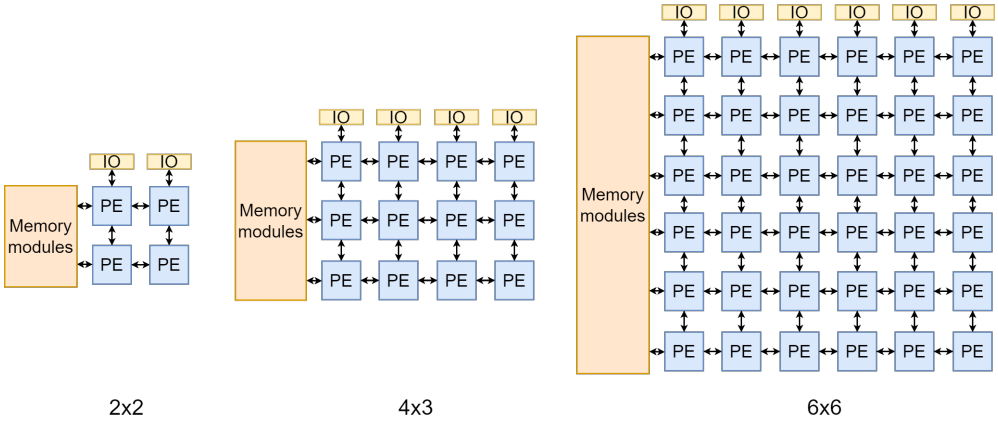


Figure 3.3: Display of the different CGRA sizes implemented

3.1.3 Streaming

After successfully implementing the buffer version, it was decided to make an improved version. This was to address the shortcomings of the prior version. This version would stop the CGRA every time a memory address was generated, for then to handle the request. This implementation does not benefit from having several memory requests in flight. However, this version requires much less physical area, and the compute length is not limited by the physical space available. It neither buffers the memory addresses and data, which is assumed to take a lot of extra clock cycles.

3.1.4 Parallel Streaming

After successfully implementing the Stream version, an enhanced version of the stream was also implemented. Instead of fetching only one value per memory address, it fetches two 16-bits words from a 32-bit memory address. The 4x3 parallel stream implements this with great efficiency. The 6x6 parallel stream takes it one step further by fetching two memory addresses and calculating three values. This allows the memory system to have several memory requests in flight which saves clock cycles.

Here, a trade-off is made for faster computing instead of computing on larger numbers. How the three implementations work in greater detail is explained in Section 3.4. The buffer and stream implementations have been implemented with 2x2, 4x3, and 6x6 CGRA architecture sizes. The 2x2 architecture was not able to map the benchmark for the parallel stream onto the architecture. Therefore, the Parallel stream has been implemented with the 4x3 and 6x6 CGRA architecture sizes. This makes a total of 8 different implementations. A visual representation of the differences in architecture size is shown in Figure 3.3.

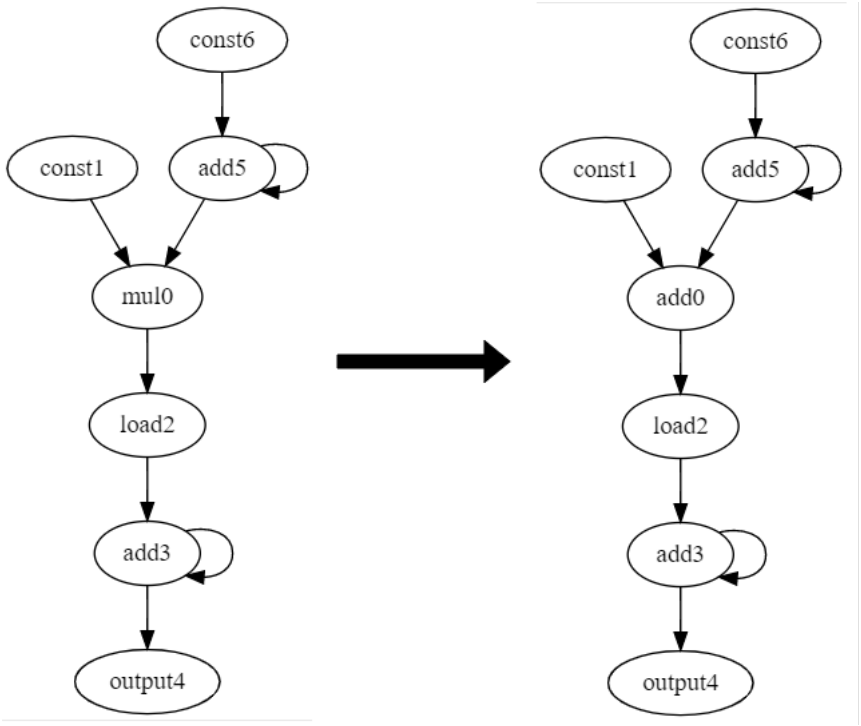


Figure 3.4: Visual display of DFG loop before and after changing the address generation

3.2 Mapping with CGRA-ME

As mentioned in Section 2.3.1, the mapper does not produce the right constants in the testbench. Therefore, we have to correct these constants manually. Consequently, we have to look into the configuration and determine which values and addresses we need. With the help of the DFG and the information in the "stdout.txt" file, we can quickly correct the configuration bitstream in the testbench with the correct constants.

The initial thought was to make the CGRA produce the full memory addresses to the variables we wanted to calculate. However, it was impossible to produce the correct memory address with the produced mapping. The mapping is shown on the left-hand side of Figure 3.4. The first address we wanted the CGRA to produce was "FFFFFAC0", which in itself is no problem to produce with the current mapping. The problem comes when we want the CGRA to produce a series of addresses spaced 4 byte apart starting from the address "FFFFFAC0".The solution was to change the multiplier "mul0" to an adder. This made it possible to keep the starting address in one constant and the spacing value in the other constant. The mapping would now produce the starting address and add an offset for every iteration. The new mapping is seen on the right-hand side of Figure 3.4.

However, this was a tedious process to do every time the mapper produced a

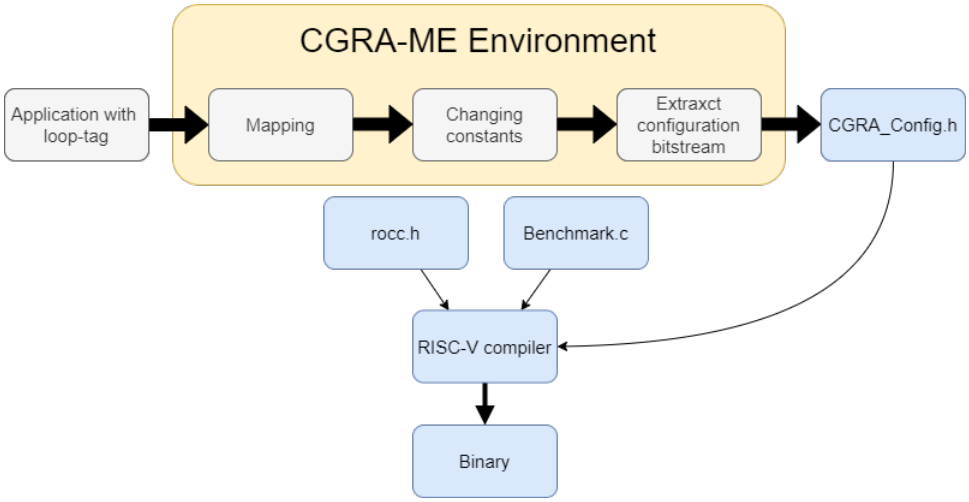


Figure 3.5: Software flow

mapping and relied on the first variable to always be placed in the same memory address. This is not always the case. A new way of producing the memory addresses was created. Instead of the CGRA making the full memory address, It is now making the offset from an input address. The CPU can then dynamically send in the memory address of the first address in the array and add the offset from the CGRA to produce the following memory address. This setup is used in all the implementations and can be achieved with both configurations in Figure 3.4. However, since the mapper creates the mapping on the left-hand side of the figure, this is the most used mapping.

A python script extracts the configuration bitstream from the testbench generated by CGRA-ME. The script was initially made by Lasse Eggen but has been exposed to several modifications since the start of the thesis. The script can be found in Appendix A. Support for every architecture size has been added, and some minor bugs have been fixed. Scripts for sending the configuration directly to the correct folder have been added to the script to ease the workflow and reduce the number of commands required. The script works by reading the benchmark file, removing all unnecessary information, and creating a string of all the bits in the testbench. This string is divided into strings of 64-bits and converted into an unsigned long int and appended to an array. The integer array is then written to a header file. This header file is sent to the correct folder depending on the architecture size and configuration.

3.3 Software Flow

Figure 3.5 gives a visual representation of the software flow in the system, starting from the application with a loop-tag to the complete binary. The first thing that needs to be done is to find the for-loop in the application that we want to

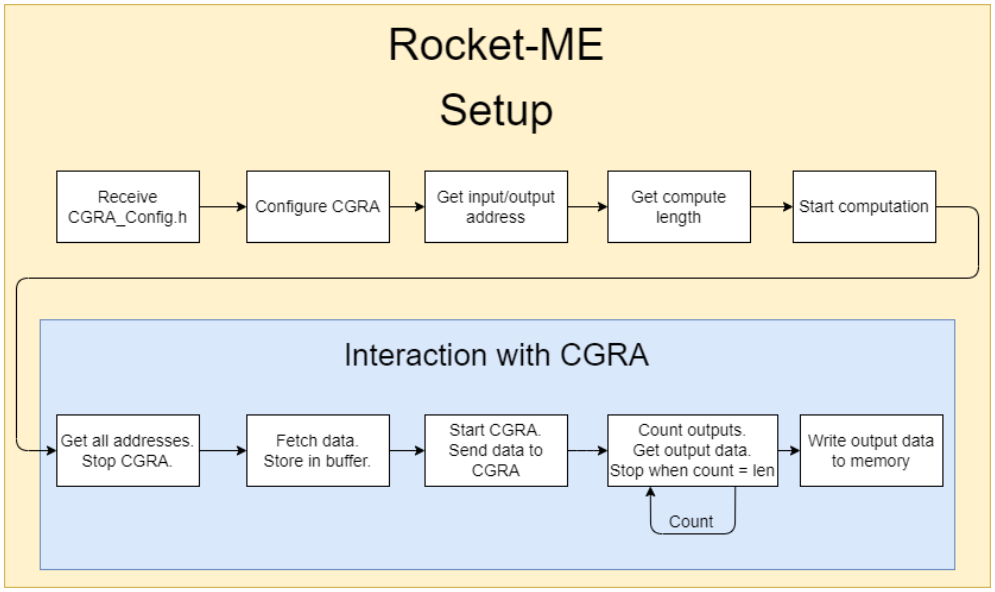


Figure 3.6: Hardware flow for Rocket-ME with the buffer implementation

accelerate. This loop needs to meet all the limitations mentioned in Section 2.3.2. The application with the loop tag is fed to the mapper, which produces a mapping if all the requirements are met. Since the mapper doesn't care about constants, we must manually go into the produced testbench and change the constants to the correct values. This process is aided by the "stdout.txt" file and the visual representation of the DFG. The configuration is extracted from the testbench with the help of a python script. This script removes all unnecessary information and converts the bitstream to an unsigned long int array. This array is then put in a header file and placed in the correct folder depending on the architecture.

Since we will run the mapped for-loop on the architecture, we need to replace the for-loop code in the application with a function for acceleration. This function is found in the file "Accelerate.c" [34]. This function will do the necessary setup and send the configuration bitstream, the input, output, and compute length to Rocket-ME. The necessary file that needs to be included in the benchmark code is "rocc.h" and the "CGRA_Config.h". These are then compiled into a binary with the RISC-V compiler.

3.4 Hardware Flow

There are two different hardware flows depending on the CGRA implementation. The hardware flows are visualized in Figure 3.6 and 3.7. The figures illustrate the flow in Rocket-ME and the interaction with the CGRA at the computation stage. The workflow is independent of the architecture size, and the setup stage is equal for the buffer and stream implementation.

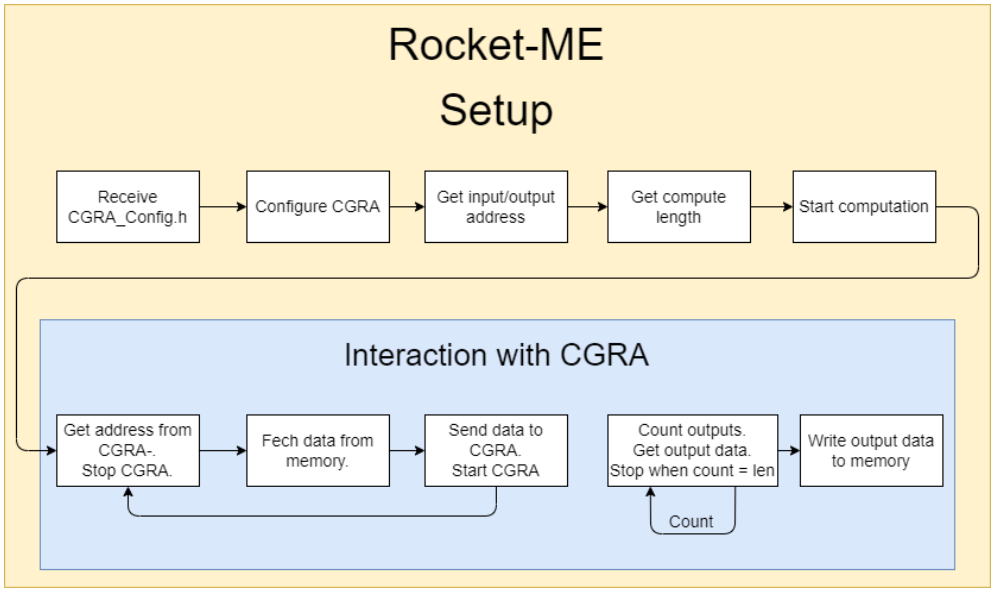


Figure 3.7: Hardware flow for Rocket-ME with the stream implementation

First, the CPU sends the CGRA configuration bitstream, created by CGRA-ME, to Rocket-ME. Rocket-ME then configures the architecture. When the configuration is done, Rocket-ME waits to receive the input address, output address, and compute length before it starts the computation on the CGRA.

The buffer implementation has the hardware flow shown in Figure 3.6. When starting the computation, Rocket-ME also starts the CGRA clock. The CGRA is now generating memory addresses that Rocket-ME stores in a buffer. When Rocket-ME has stored enough memory addresses, it stops the CGRA clock and starts sending multiple memory requests to the memory module. When all memory requests have been handled, and the corresponding data has been received, it starts the CGRA clock and feeds the architecture with data. Another circuit in Rocket-ME monitors the output from the CGRA and counts the number of outputs. When the number of outputs is equal to the compute length, the CGRA is stopped, and the output value is written in the memory location of the output address received from the CPU.

Figure 3.7 presents the hardware flow for the stream implementation. When starting the computation, Rocket-ME also starts the CGRA clock. When the CGRA has generated a memory address, Rocket-ME stops the CGRA clock and sends a memory request to the memory system. The fetched data is fed directly to the CGRA, and the clock is started again. When the CGRA again has produced a memory address, the process repeats. Like the buffered implementation, another circuit in Rocket-ME monitors the output from the CGRA and stops the CGRA clock when the number of outputs equals the compute length. The output is then written in the memory location of the output address received from the CPU.

Chapter 4

Experimental Setup

This chapter aims to cover the experimental setup of the thesis. The benchmark is covered in Section 4.1. Section 4.2 explain how the different architectures sizes affect the result and Section 4.3 covers how the various implementations of the Rocket-ME module affect the performance. How we analyzed the result is covered in Section 4.4.

4.1 Benchmarks

The benchmark Sum is used to measure the performance of the CGRA implementation against the CPU. Sum is a short C code that accumulates the sum of an array and is included as a benchmark in the CGRA-ME environment, among many other benchmarks. The benchmark is included in Appendix A. It is used because of its simplicity and ease of scaling as it works for the smallest 2x2 architecture and the more extensive 6x6 architecture. In our environment, Sum has a base version, with five additional other versions to accommodate the different implementations and architecture sizes. The main reason for the many versions is that the size and data in the included configuration bitstream vary depending on the architecture size and implementation. A table of the different versions is shown in Table 4.1.

Each version has a function that is called "array_gen". This function creates a pseudo-random array with a set length. This is the array the processors and co-processors will accumulate to test their performance. The 4x3 and 6x6 configuration in the Double stream implementation has modified this function to achieve a higher level of data-level parallelism in the computation. When creating the pseudo-random array for the 4x3 Double stream architecture, it places two 16-bit integer values into each memory address. This allows the architecture to ask for one memory address and receive two values. The same is done for the 6x6 Double stream architecture, but this architecture can calculate three values simultaneously. The solution was to place only one 16-bit integer value in every second memory address. By doing this the 6x6 architecture can have two memory requests in flight and calculate on three values. The initial thought was to place two 32-bits integer values into a 64-bits memory address, but it was impossible since

Table 4.1: Detailed display of which benchmarks that belongs to the different vizualizations in Figure 4.1

Implementation		Architecture Size	Representation in Figure 4.1
Serial	Base	-	a
	Buffer	2x2	
		4x3	
		6x6	
	Stream	2x2	
		4x3	
		6x6	
Parallel	Parallel Stream	4x3	b
		6x6	c
	Base Parallel	-	

Rocket-ME can only fetch 32-bits values from memory. The mentioned modification limits the calculations to 16-bits integer input values, which was a trade-off for achieving higher throughput. The other benchmarks have the same array_gen function, which only adds one 16-bit element to each memory address. However, the other implementations could calculate 32-bit integer values, but we chose to calculate only 16-bit values to make the benchmarks as similar as possible.

Each benchmark for the accelerator has a function called "send_config" which sends the configuration bitstream over to Rocket-ME through the RoCC interface. This function is a simple for-loop that sends the configuration bitstream for the architecture 128-bits at a time. Depending on the architecture size, the for-loop is iterated a set number of times. It is a concise and efficient function that can be seen for the 6x6 architecture in the benchmark code in Appendix A.

For the benchmarks running on the accelerator, it is essential to include memory fences. This ensures that the memory is in a "secure" and coherent state before Rocket-ME begins fetching data. If this is not done, Rocket-ME may risk race conditions which may cause the accelerator to fail or calculate with the wrong values. It is also essential to include a memory fence at the end of the accelerator computation. If the processor continues doing memory operations while the CGRA is running, it may end up with an inconsistent memory filled with errors. The fence notation can be seen in the benchmark code for the 6x6 architecture in Appendix A.

Figure 4.1 visualize the different versions of the benchmark. The Figure (a) on the left-hand side is the base version. This version fetches one value, adds it to the adder's value, and sends it to an output. The Figure (b) in the middle present the benchmark for the 4x3 parallel stream. This benchmark enables the fetching of two inputs at the same time. These inputs are added together and then accumulated. The accumulated value is then sent to the output. The right-hand side Figure (c) represents the benchmarks for the 6x6 parallel stream and the parallel baseline. This version takes in 3 values from 2 memory addresses, add them

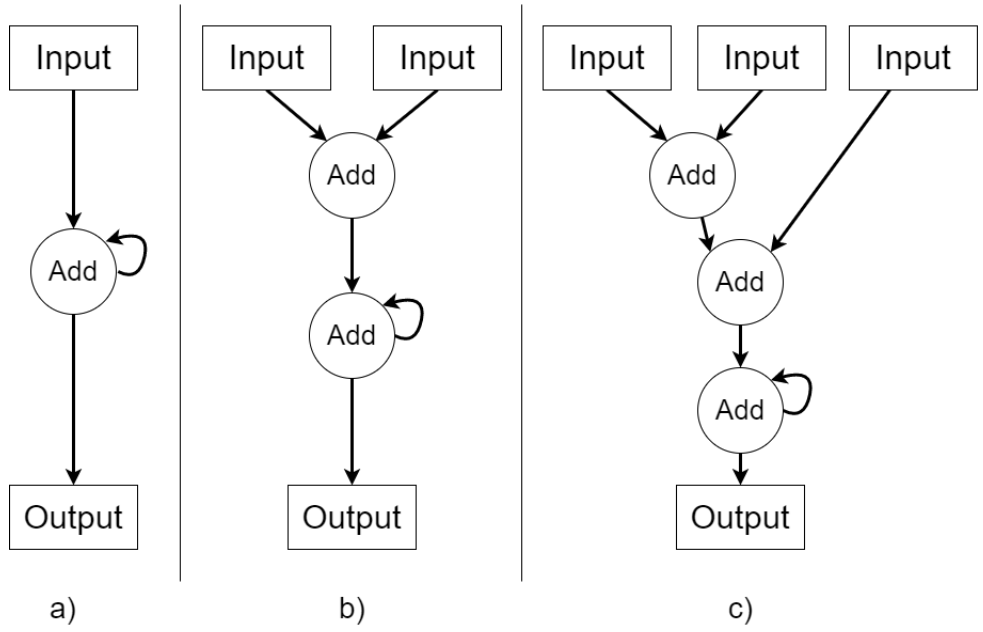


Figure 4.1: Visual representation of the benchmarks

together, and then accumulates the value. The accumulated value is then sent to the output. For every iteration, the data on the output will be an intermediate result. It is expected that the versions with higher data level parallelism (DLP) will perform better than the base benchmark. All the implementations run the base version except from the parallel baseline, parallel stream 4x3, and parallel stream 6x6. This is because the other implementations are not coded to handle DLP.

4.2 Architecture Size

The architecture size of the CGRA plays a considerable part in the performance of the integration. A CGRA with a larger architecture size can map larger and more complex applications, while a small architecture size can only map simpler applications. However, if no parallelization is done, a serial application that maps on the 2x2, 4x3, and 6x6 architecture will not benefit from the larger architecture sizes. The computation in a serialized application will take approximately the same time for the different architecture sizes. However, several PEs are not used in the larger architecture sizes, but these PEs still need to be configured. This contributes to a more significant overhead for the larger architectures and will affect the overall performance of the architecture.

The CGRA can significantly boost performance if exposed to data-level parallelism. The sum benchmark is not parallel by default, but by doing some minor modifications to the source code, it can achieve a higher level of parallelization. The sum benchmark can be mapped to all the architecture sizes of the serialized

buffer and stream implementations. However, parallelization demands more of the architecture. The sum benchmarks were modified to give a higher level of parallelization but could not map onto the 2x2 architecture. The 4x3 architecture had enough PE to create a mapping where it can receive two inputs into the architecture. The more extensive 6x6 architecture had enough PEs to receive three input values simultaneously, giving an even higher level of DLP compared to the 4x3. The parallelized architectures are compared to a parallelized version of sum that simultaneously accumulates three values.

4.3 Rocket-ME

The Rocket-ME module is implemented in three different versions implementing different functionality and handling of memory. The first version implemented is the buffer implementation. This is a serialized implementation that buffers the memory addresses and data into buffers before the actual accumulation of the array. This implementation does not benefit from the temporal storage of the fast cache in Rocket Chip and is expected to be the implementation with the poorest performance.

The second version, Stream, takes a different approach to memory address handling and is expected to be faster than the buffered version. This version is also dynamically scalable and does not require more physical space as the compute length increases. The parallelized stream implementation is expected to be the best performing implementation as it benefits from DLP and can have multiple memory requests in flight.

4.4 Measuring Execution Time

The results in Chapter 5 are obtained by analyzing the cycle-exact printout from Verilator and analyzing the waveform trace for the benchmarks. Verilator is used as our simulator because it performs cycle-exact simulations and is included in the Chipyard environment. Each benchmark has inserted "NOP" instructions to mark different parts of the code. The "NOP" instruction is an instruction that does nothing else than advance the program counter. The first "NOP" instruction marks the start of the application. The second "NOP" marks the beginning of the configuration stage. The third "NOP" marks the end of the configuration stage and the start of the computation stage. The last "NOP" is used to mark the end of the computation stage and the end of the application. The instructions can easily be found with a python script to separate and divide the printout into the wanted sections.

It is enough for the two Base benchmarks to analyze the cycle-exact printout from Verilator, but for the benchmarks involving the accelerator, we also need to analyze the waveform traces. When an instruction is sent to the accelerator, it first goes into a queue in Rocket-ME that the CPU is unaware of, and the sent command

is first received by Rocket-ME when it is not busy. Therefore, the cycles that the CPU sees and Verilator prints out are not accurate to the accelerator's actual cycles. We get an accurate cycle count for the configuration and computation stage by looking at the waveform diagram.

The base simulation is only run on the CPU and does not have an accelerator to configure, so it does not spend any time on configuration. In contrast, the configuration times for the different architecture sizes vary. The configuration times are a part of the presentation of the serial and parallel results in Figure 5.1 - 5.5. The size of the configuration bitstream for the different architecture sizes is seen in Figure 5.7a as well as the configuration time in Figure 5.7b.

Looking at the total time for each application would give an erroneous performance measure. This is because of the proxy kernel in Verilator, which busy-waits for a response from the host computer when doing system calls. The busy-wait period is different from simulation to simulation and will affect the result. Therefore, the waveform diagram is used to get an accurate cycle count. However, the cycle it takes for creating and sending the bitstream configuration to the accelerator is calculated from the cycle-exact printout from Verilator. This section does not contain any proxy kernel instructions and is assumed to show the correct number of cycles. We assume the only difference in time for the different benchmarks is made in the sections dealing with the accelerator. These sections are making and sending the configuration bitstream to the accelerator, and the time it takes to perform the accumulation of the array.

Chapter 5

Result

This chapter present and discuss the results obtained from simulating the implementations. Section 5.1 through 5.5 presents and discuss the achieved result for the different array lengths. The overall result discussion takes place in Section 5.6, and the configuration overhead is presented and discussed in Section 5.7.

The results from five different benchmark runs is seen in Figure 5.1 to 5.5. The y-axis shows the time of each benchmark relative to "Base", and the x-axis represents the different implementations. Each bar in the plots has three different sections. The blue section represents the computation time it takes to accumulate the array. For the Base and Base parallel benchmark, this computation takes place on the CPU. For the other benchmarks, this takes place on the CGRA. The measurement is taken from the start of the accumulation operation to the end of the accumulation operation. The orange section is the time spent by the CPU on sending the configuration bitstream over to the CGRA. That is, fetching the configuration bitstream from memory and sending it to Rocket-ME via the RoCC interface. The green section is the time spent driving the configuration bitstream through the CGRA architecture. The code config section is the only part executed on the CPU besides the Base and Base parallel benchmarks. Figure 5.1 to 5.5 is essentially divided into two parts. The left-hand side of the figures is the serial part, and the right-hand side of the figures is the parallel part. Each part has its baseline benchmark called Base and Base Parallel.

5.1 102 Array Elements

Figure 5.1 presents the result from the benchmark run with 102 array elements and shows no total speedup for any of the benchmarks. This is not unexpected as the overhead from configuration is quite significant. The smaller architectures have less configuration overhead since they have fewer elements in the architectures to configure. One can see that the time spent on code configuration and configuring the CGRA scales with the size of the architecture. Nevertheless, looking at only the serial computational part of the result, one can see a slightly faster

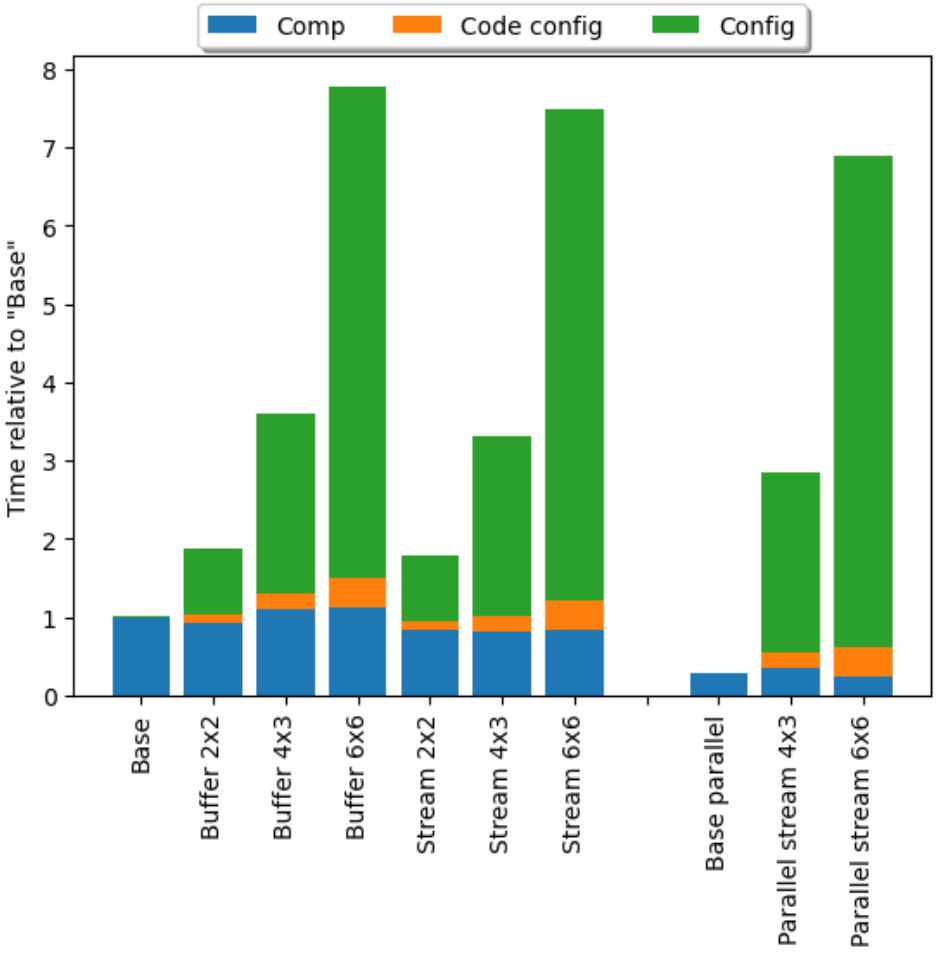


Figure 5.1: Result from benchmark run with 102 elements

computation for the CGRA architecture buffer 2x2 and all the stream implementations compared with Base. Seeing the buffer 2x2 implementation computing the accumulation faster than Base is surprising, especially as the buffered 4x3 and 6x6 architecture achieve the same performance. This implementation was expected to be slower than Base as it needs to buffer addresses and data before starting the computation. The stream implementation was expected to be as fast as Base as it performs the accumulation in somewhat the same manner. From Figure 5.1, one can see that the stream implementations performed as expected.

The parallel Base performs 3.5 times faster than the Base and is the baseline for the parallel computation. The parallel part of the result shows a slight speedup on the computational part for the parallel stream 6x6 implementation. This is not surprising as this implementation benefits from DLP. The baseline also benefits from loop unrolling but does not outperform the accelerator. However, the computation length is not long enough to overcome the significant overhead of

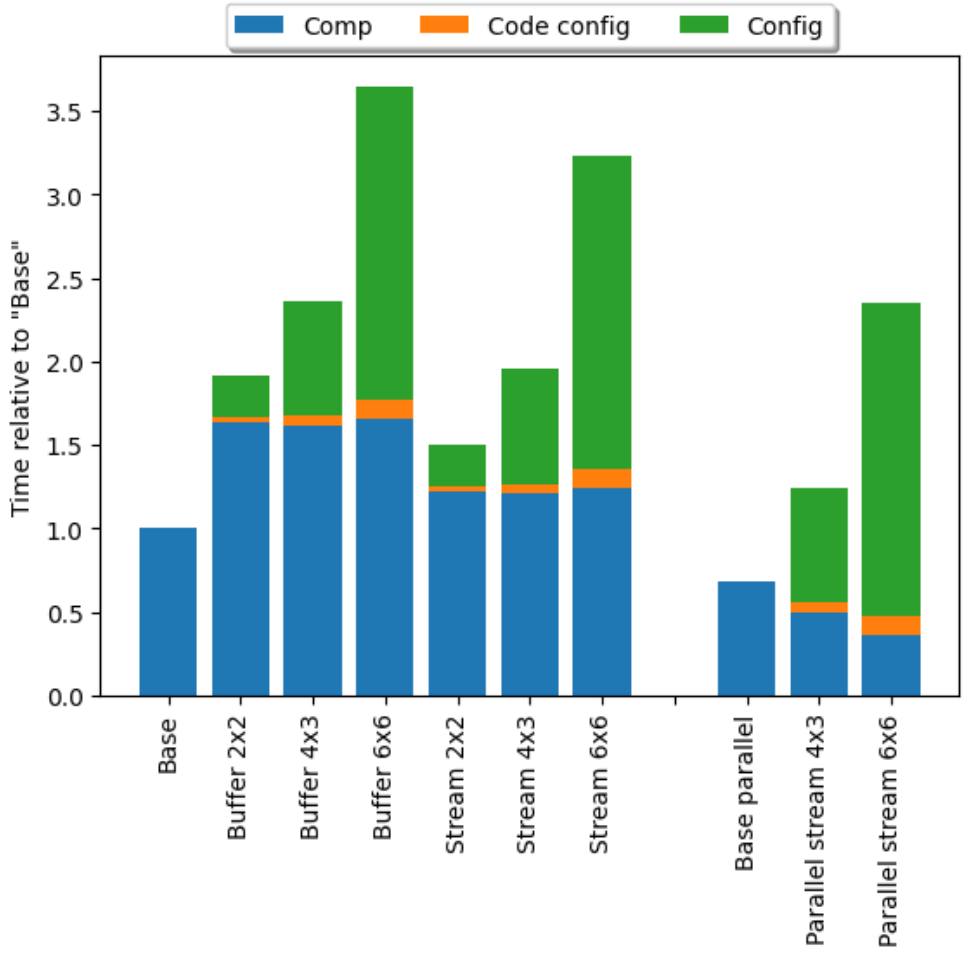


Figure 5.2: Result from benchmark run with 501 elements

configuration. The total time for the parallel stream 6x6 implementation is 23.8 times slower than the parallel baseline. The parallel stream 4x3 implementation also benefits from DLP, but as this implementation only fetches two values per iteration, it is not as effective as the 6x6 architecture. However, the total time for the parallel stream 4x3 implementation is 9.9 times slower than the parallel baseline, benefiting from a smaller configuration overhead.

5.2 501 Array Elements

Figure 5.2 presents the result from a benchmark run with an array length of 501 elements. The plot shows several things. For the serial benchmarks, there is no speedup. Unlike the result in Figure 5.1, the plot shows an increased difference in the computational part for the buffer and stream implementation. The performance of the Buffer implementation is the worst, and the computation time

and configuration time increases with the architecture size. The stream implementation is slightly faster than the buffer implementation but is also slower than Base. The expected result was that the stream implementation would be as fast as the baseline. The stream 2x2 implementation is the best performing accelerator for the serial computation and is 1.5 times slower than Base. The buffer 6x6 implementation is the worst performing and performs the accumulation 3.6 times slower than Base.

The parallel Base performs much better than the serial Base benchmark with a speedup of 1.5. This is less of a speedup compared to the result in Figure 5.1. The parallel stream 4x3 and 6x6 performs better than the baseline for the computational part with a speedup of 1.36 and 1.87, respectively. Still, the configuration overhead is so significant that the total time for the 4x3 and 6x6 architecture is 1.83 and 3.46 times slower than the parallel Base, respectively.

5.3 1002 Array Elements

Figure 5.3 presents the result from the benchmark run with an array length of 1002 elements. The plot presents no speedup for the serial benchmarks. The accelerated architectures do the accumulation significantly slower than Base. The best performing accelerated architecture is the stream 2x2, with a total time 1.45 times slower than the Base benchmark. Surprisingly the buffer 4x3 implementation is the worst performing on the computational part, with a computation time 1.74 times slower than Base. Still, due to the large overhead on the buffer 6x6 implementation, it is the worst overall performer, with a total time 2.78 times longer than Base. Comparing Figure 5.3 with Figure 5.1 and 5.2, one can see an emerging trend. As the array length increases, the time it takes the buffered version to complete the accumulation increases linearly compared to Base.

The buffered version benefits from having several memory requests in flight when it fetches the values from memory. This is the reason why one can see that the 2x2 buffer implementation is faster than Base in Figure 5.1. However, the performance benefit from having several memory requests in flight gets eaten up by the address and data buffering when the array length increases. Therefore, the time it takes for the buffer version to complete the accumulation increases linearly with the array length.

The parallel implementations have no total speedup for the accelerated architectures. Still, there is a slight speedup on the computational part of the parallel stream 4x3 performance of 1.08×. The 6x6 architecture performs even better with a speedup of 1.86×. However, these speedups get eaten up by the significant overhead from the configuration, and the 4x3 and 6x6 architecture end up using 1.48 and 2.02 times longer time than the baseline, respectively.

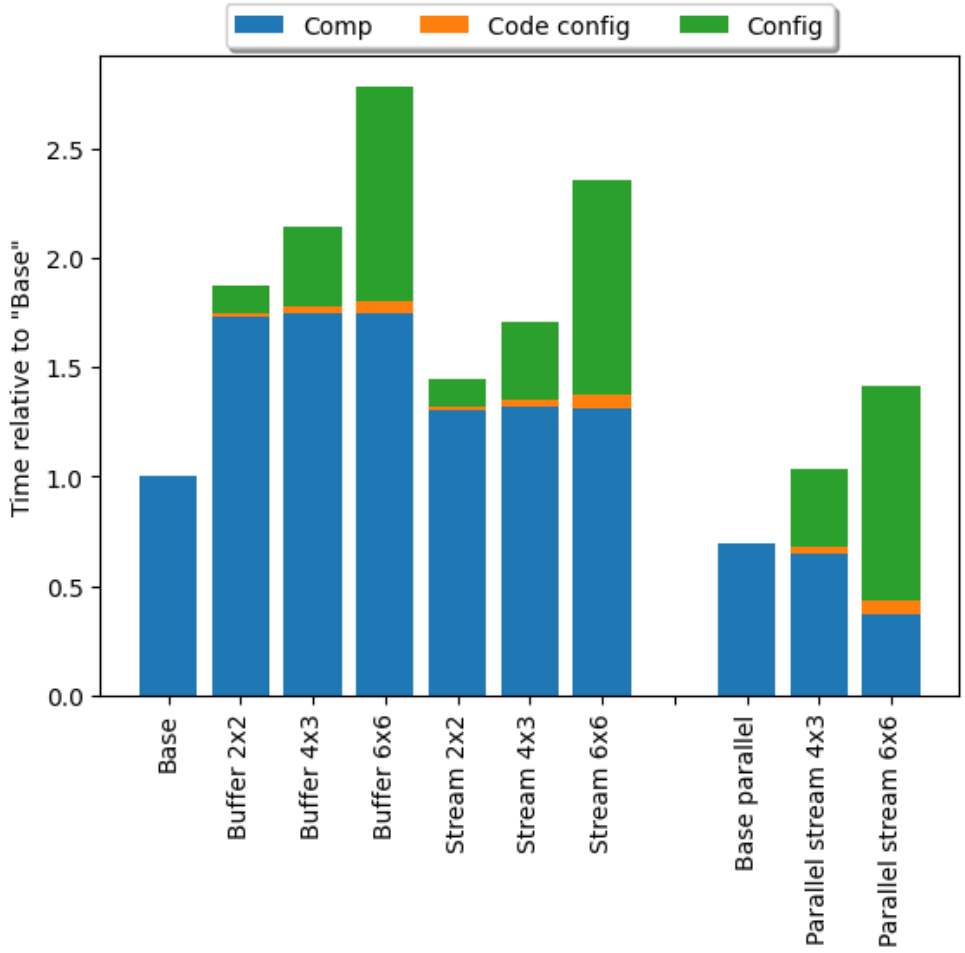


Figure 5.3: Result from benchmark run with 1002 elements

5.4 2001 Array Elements

Figure 5.4 presents the result from the benchmark run with an array length of 2001 elements. The plot shows a marginal speedup for the computational part for the stream 2x2, 4x3, and 6x6 implementations. The speedup for the 2x2, 4x3, and 6x6 architecture is approximately 1.04×,1.05×, and 1.06×, respectively. Still, the overhead from the configuration makes the total time for the architectures longer than Base. The total time for the stream 2x2, 4x3, and 6x6 is 1.02, 1.1, and 1.33 times longer than the baseline, respectively. The Buffer implementations suffer from poor performance, and the 2x2, 4x3, and 6x6 architecture uses 1.33, 1.41, and 1.64 times longer than the baseline, respectively.

The accelerated architectures for the parallel computation have achieved a speedup in both the computational part and the total time. The parallel stream 4x3 implementation achieved a speedup of 1.36× for the total time and 1.85×

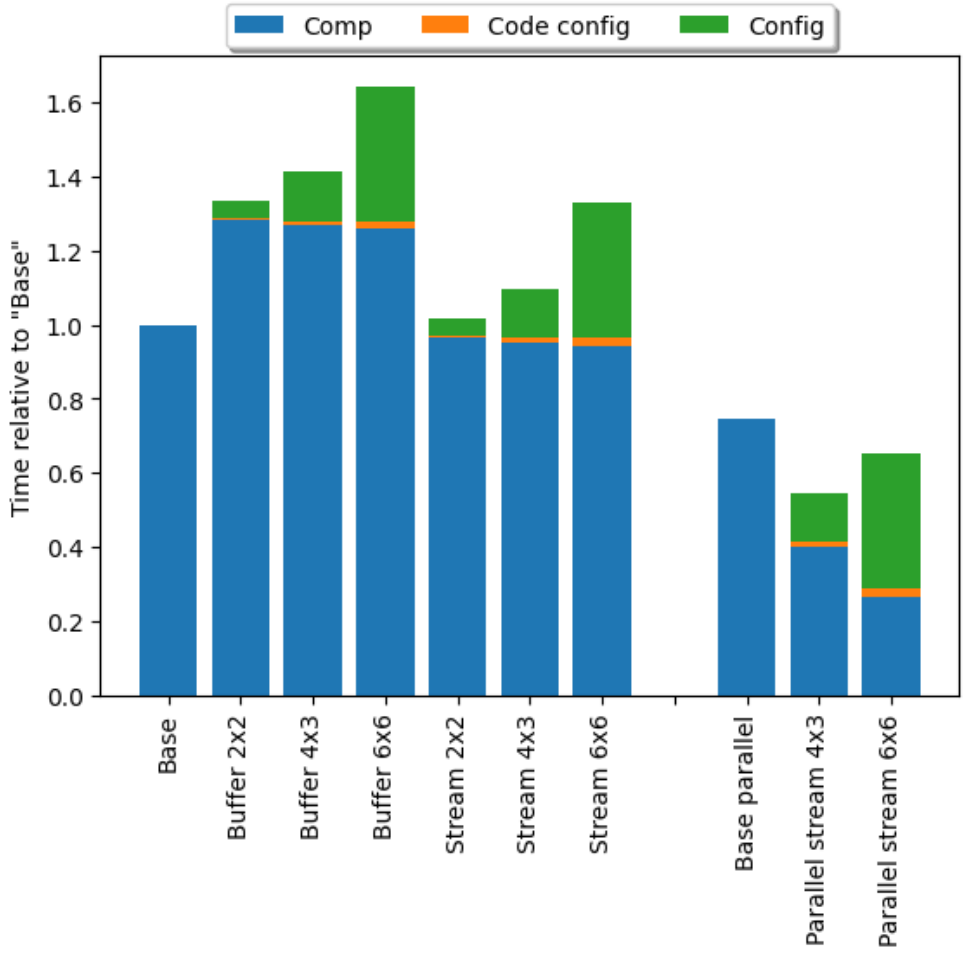


Figure 5.4: Result from benchmark run with 2001 elements

for the computational part. The parallel stream 6x6 implementation achieved a total speedup of 1.14× for the total time and 2.79× for the computational part. Even though the 6x6 architecture computes faster, it uses a longer total time than the 4x3 architecture. This is due to the significant configuration overhead of the larger architecture.

5.5 9000 Array Elements

Figure 5.5 present the benchmark run with 9000 array element. The plot does not include the Buffer implementation as this implementation could not implement the memory address and data buffer with an array length of 9000. The stream implementations are not achieving a speedup and are slower than the baseline. The plot shows that the computation time increases with the increased architecture size of the stream implementation. The differences are not that significant

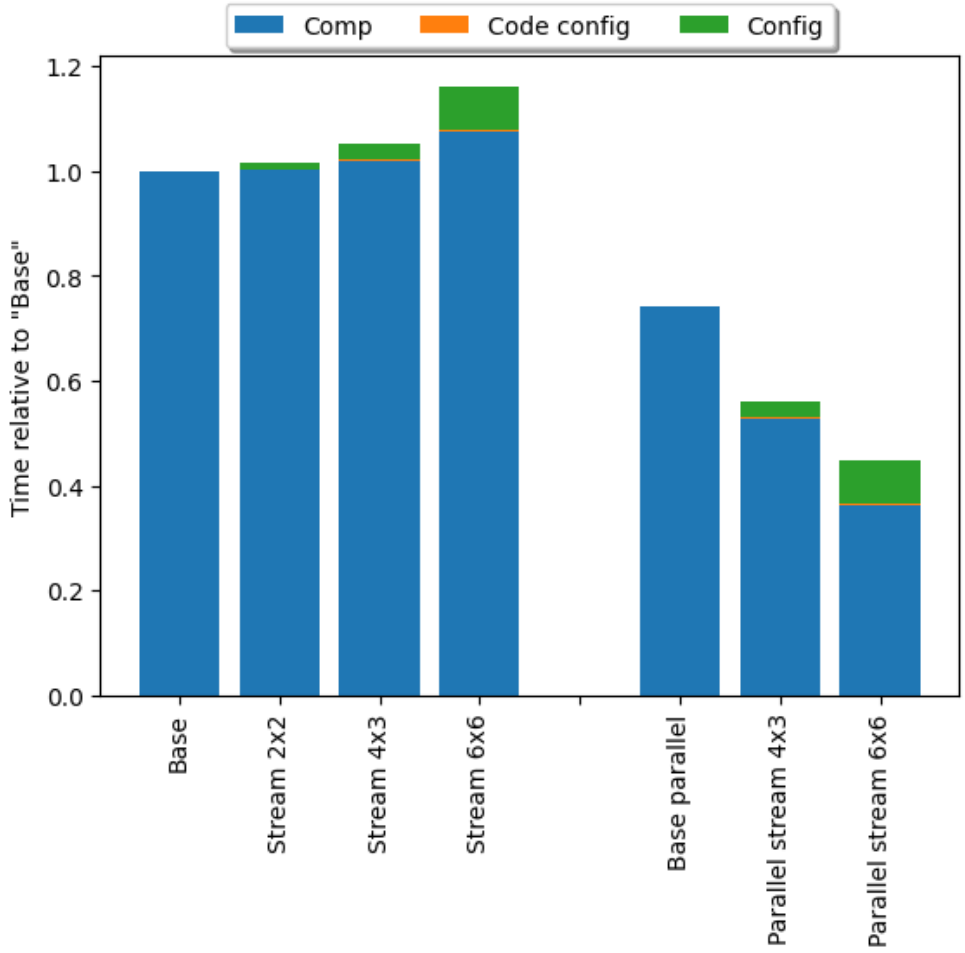


Figure 5.5: Result from benchmark run with 9000 elements

and are because of filling the pipeline in the CGRA architecture.

For the parallel computation, the Base parallel performs a speedup of 1.35× compared to the Base benchmark. The parallel stream 4x3 achieves a performance gain of 1.41× compared to Base parallel, and the Parallel stream 6x6, which is the best performer in this benchmark run, has a speedup of 2.05× compared to Base parallel.

5.6 Overall Result Discussion

The presented results show no speedup of the serial implementations' total time, independent of the array length. However, enabling the Rocket-ME module to handle DLP allows it to utilize the CGRA more efficiently. The parallel stream 6x6 implementation is always faster than the baseline at accumulating the array, but its overall performance is worse than the baseline due to the significant config-

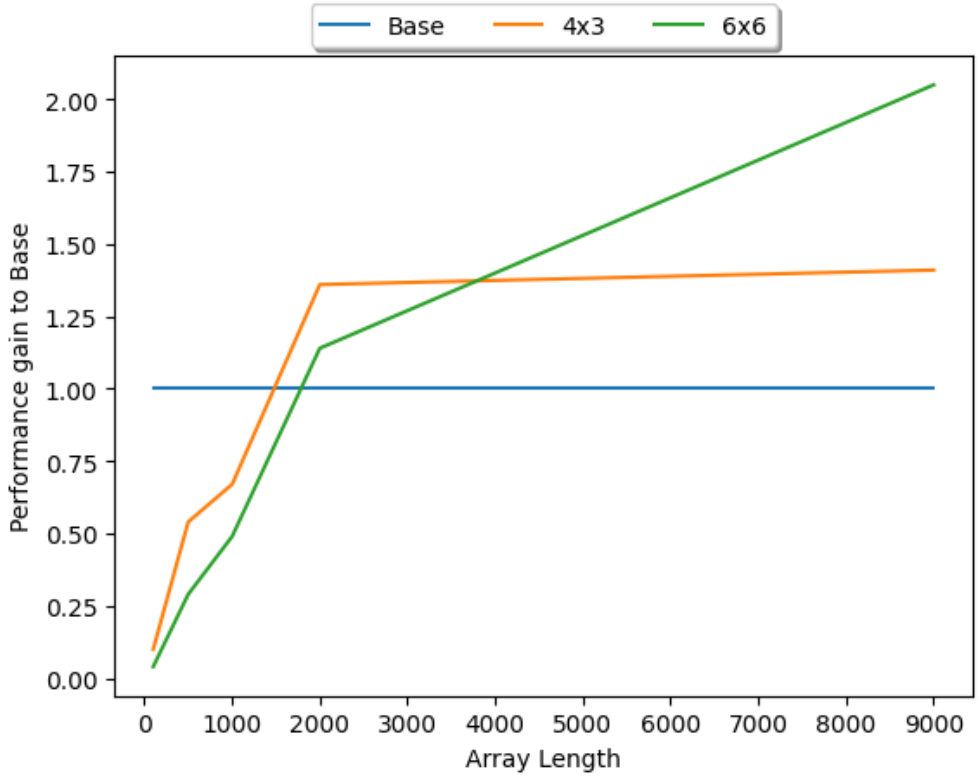


Figure 5.6: Plot of the performance gain relative to Base Parallel

uration overhead. That is, except for the benchmarks run with 2001 and 9000 array elements. This shows that the array length needs to be significantly long to overcome the configuration overhead of the larger architecture.

The parallel stream 4x3 implementation is the one that achieves the best speedup for the benchmark run with 2001 array elements. However, it does not perform the accumulation as fast as the parallel stream 6x6 implementation because of a lower level of DLP. But due to less configuration overhead, it outperforms the larger 6x6 architecture on the total time for this array length.

Nevertheless, increasing the array length to a significantly high number makes the parallel stream 6x6 implementation perform better than the 4x3 architecture. This is shown Figure 5.5 with the benchmark run with 9000 array elements. Comparing the result of parallel stream 4x3 and 6x6 reveals a trend as the array length increases. Increasing the array length reduces the impact of the configuration overhead, and therefore the larger and faster 6x6 architecture outperforms the smaller 4x3 architecture.

Figure 5.6 shows the performance gain relative to the Parallel Base for the Parallel stream 4x3 and 6x6. The plot shows how long the array length needs to be for the CGRA to overcome the significant configuration overhead. The plot shows that the Parallel stream 4x3 implementation will perform as good as the CPU when

the array length is approximately 1500 array elements. The Parallel stream 6x6 is as fast as the CPU at about 1800 array elements. An interesting observation here is that at around 3700 array elements, the 6x6 implementation performs as good as the 4x3 implementation. After this, the 6x6 is the best-performing implementation.

The benchmarks Base and Base Parallel have been used as our baseline when analyzing the results. They are essentially the same benchmark, with some minor changes. The Base iterates and accumulates one value every iteration in the computation stage. The Base Parallel iterates and accumulates three values every iteration. This makes the compiler do a loop-unrolling optimization, making the Base Parallel perform better than the Base. The reason for creating two base benchmarks is that it would be an invalid comparison if we had compared every result to the serialized Base version.

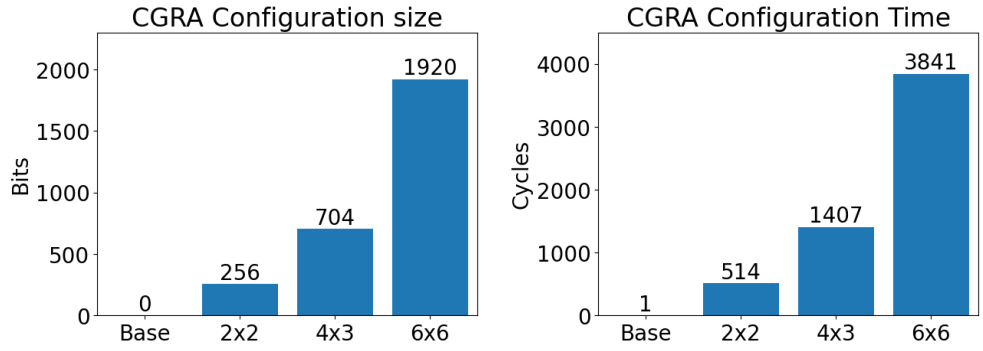
The results shown in Figure 5.1 to 5.5 presents the time used in 3 different parts of the simulation. These are the compute stage, configuration sending stage, and the CGRA configuration stage. It is assumed that all other parts of the application are run equally fast. We, therefore, concentrate on the part where the differences are. When we say there is a total speedup of 2, the accelerator is 2 times faster than the CPU at the particular compute stage in the application. The total speedup for the entire application is, in reality, smaller than 2.

The reason for doing it this way is the proxy kernel, which is utilized to assist with the application's system calls. The proxy kernel sends system calls to the host computer, which executes the system call. The time it takes to perform a system call may vary, affecting the total execution time of the application. The Computation stage, configuration sending stage, and CGRA configuration stage do not contain system calls and are therefore seen as cycle accurate and valid.

5.7 Configuration Overhead

Figure 5.7a present the size of the configuration bitstream for the different architecture sizes. Each processing element in the architecture adds around 52-54 bits to the size of the configuration bitstream. The Base does not have an accelerator and has 0 bits in configuration size. Figure 5.7b shows the configuration time for each architecture size. The result is obtained by counting cycles for each implementation's configuration stage in the waveform diagram. The configuration time for each architecture size is the same, independent of the implementation, since the only difference in the implementations is in the Rocket-ME module. This means that the buffer 2x2, stream 2x2, and parallel stream 2x2 have the same configuration time. The Base benchmark has one cycle of configuration time because of the inserted "NOP" instruction to mark the end of the configuration stage.

Comparing Figure 5.7a and 5.7b show that the configuration time is approximately twice the configuration bitstream size. This makes sense as the configuration clock runs at half the speed of the system clock.



(a) Configuration Size **(b)** Configuration times in cycles

Figure 5.7: Plots of configuration size and time

The configuration time is the main bottleneck when accumulating small array lengths. The benchmark result in Figure 5.4 and 5.5 are the only results that show a total speedup for any benchmark and are also the benchmark runs with the longest array lengths. Several of the benchmarks, such as the parallel stream 4x3 and 6x6 in Figure 5.2 show a speedup for the computational part compared to the baseline. Still, due to the configuration overhead, the CPU outperforms the accelerated architectures on the total time. However, the configuration overhead only applies for the first time the application start. If the application is doing multiple accumulations, the overhead of configuring the architecture would only apply to the first run. This means that the performance result will look different if doing several accumulations after one another.

Making the config clock run at the same speed as the system clock would make the CGRA configuration time twice as fast as shown in Figure 5.7b. This would positively affect the result. Several attempts to make the configuration clock run on the system clock have been made but have not successfully been implemented. This will be part of Further Work. The result presented in Figure 5.4 shows that the parallel stream 4x3 is the best performing implementation with this array length. However, cutting the configuration time in half would make the parallel stream 6x6 the preferred implementation. Nevertheless, the Parallel stream 6x6 implementation will be preferred if the application does several accumulations in a row or if the array is significantly long.

The benchmark is a metric for testing the implementation compared to the current processors. However, other applications will do more than just accumulate one array. Suppose the accumulation of an array is done in the middle of the application or a significant amount of cycles out in the application, it may be possible to hide the overhead combined with configuration.

Let's say the accumulation happens late in the application. We can then configure the CGRA architecture early in the application while the CPU does other parts of the application. The only overhead the CPU will exhibit is the overhead of sending the configuration bitstream to the CGRA. The significant overhead from

configuring the CGRA would now be hidden. This would improve the overall time for the accelerator significantly and reduce it down to the compute time.

Chapter 6

Discussion

This chapter aims to discuss the topics other than the results, which are discussed in Chapter 5. Section 6.1 explains the two options for integrating the accelerator into the Rocket Chip and how they differ. Section 6.2 covers the discussion about the benchmark and the challenges with the MachSuite benchmarks. Section 6.3 covers some of the challenges with CGRA-ME and its mapper. In Section 6.4 we discuss the design of the CGRA architecture. Section 6.5 explains how Amdahl's law sets the theoretical speedup for a CGRA architecture.

6.1 Way of Integration

There are two ways to integrate an accelerator into the Rocket Chip SoC. It can be connected as a Memory-Mapped Input Output (MMIO) or through the RoCC interface. Integrating the accelerator as an MMIO benefit from not needing a custom software compiler, and the communication is done through memory-mapped registers. However, if the accelerator changes something in the system that the CPU needs to know about, it needs to poll the memory register until the change happens. The communication is done through the memory bus, which several other modules also use. This can cause congestion and unwanted communication overhead if other modules use the memory bus simultaneously as the accelerator communication happens.

The second alternative is integrating the accelerator through the more tightly coupled RoCC interface. The RoCC interface is explained in Chapter 2.1 and is based on a ready-valid protocol. The communication to the accelerator happens on an isolated line that only the CPU and accelerator use. It also allows direct communication to the L1 cache. These features motivate the selection to use this interface when connecting the accelerator to the Rocket Chip SoC.

6.2 Benchmark

Initially, the thought was to use the MachSuite benchmarks to analyze the performance of the CGRA implementation. When reading about the CGRA-ME mapper, we quickly understood that it would be difficult to map the benchmarks on the CGRA architecture as the mapper has certain limitations mentioned in Section 2.3.2. One of the source code requirements for the mapper in CGRA-ME is that the DFG generation software can only support a single C source file with header files. The MachSuite benchmarks are built up from several C files, and this requirement limits the portion of code we would be able to accelerate. Nevertheless, all the source code requirements are assumed held for the MachSuite benchmarks.

Another challenge with the MachSuite benchmarks is that the mapper struggles with finding a loop to map. The central part of the for-loops in the benchmarks is using an if-else statement or some branching. This makes the mapper unable to make a mapping for the architecture as it goes against the second mapper limitation listed in Section 2.3.2. The mapper can only map basic blocks, limiting what it can accelerate. Some other CGRA mappers can implement some branch prediction, but this takes a lot of resources and is more common on larger architecture sizes [35].

After expediting several failed attempts at mapping the MachSuite benchmarks, the idea of using MachSuite as the performance measure was left behind, and the search for a more manageable and less complex benchmark started. The CGRA-ME environment includes some benchmarks guaranteed to work for the mapper. That is, they are guaranteed to work on the larger architecture sizes. From experience, very few of the included benchmarks were able to be mapped on the small 2x2 architecture. However, the mapper could map the benchmark Sum to the 2x2, 4x3, and 6x6 architecture. This is one of the main reasons for choosing the Sum benchmark as the performance measure. The Sum benchmark is not very compute-heavy, but it still is good enough to measure the performance of the implemented architectures. Using a more complex and compute-heavy benchmark would potentially increase the performance gain of the CGRA compared to the CPU. However, as the complexity increases, it may not be possible to map the benchmark to the smaller architecture.

6.3 CGRA-ME

The first implementation of CGRA in Rocket Chip involved a 4x4 CGRA architecture. This architecture size was scrapped as it was exposed to a bug in the CGRA-ME mapper. The mapper could map the for-loop onto the architecture, but the mapping contained a severe error. The problem was in the setup of the register files. In the 4x4 architecture, two functional units were writing to the same register. One register was sending decimal 4, and the other constantly sent 0. This made the accelerator produce the memory address 0 every iteration. Fixing this error would require severe insight into the mapper module. Changing one of the

functional unit's inputs in the configuration bitstream can cause the accelerator not to work correctly. This problem was avoided by instead implementing a 4x3 architecture. This architecture does not get any errors when mapping the Sum benchmark onto the architecture.

We did also experience some other bugs from CGRA-ME. While running some tests with the Sum benchmark, some unexpected behavior occurred. The address generation from the CGRA behaved differently for the different architecture sizes. The 6x6 architecture worked as expected with generating the correct offset for the input address, but the 4x3 architecture could not generate the correct values. It generated offset values randomly, indicating something was wrong with the mapping. The solution to this problem was to map the Multiply-accumulate (MAC) benchmark onto the 4x3 architecture and manually change the multipliers to adders. This made the CGRA behave and compute as expected.

From the mentioned errors in CGRA-ME, the confidence in the mapping environment decrease. There are many bugs in the software that need to be fixed in the future. There is also a lack of documentation on their website. Working with CGRA-ME was challenging as the lack of available documentation meant that we had to go through the vast amount of source code to find some of the files and functions we needed. This was time-consuming and frustrating. Nevertheless, the architecture and mapping were eventually successfully implemented into the Rocket Chip SoC.

As mentioned in Section 2.3.2, the CGRA-ME mapper has several limitations. This puts a limit on what the mapper is able to map onto the architecture. Including branch prediction would require a different mapper and require a significant amount of resources. We are therefore limited to basic blocks.

6.4 Design

The Gemmini project is mentioned in Section 2.1.2. This RoCC accelerator shares similarities with the accelerator implemented in the thesis, such as doing matrix multiplication and being a RoCC accelerator. However, the accelerator implemented in this thesis can do more operations than just multiplications. The Gemmini accelerator is aimed at a specific domain, while our accelerator is more general. However, if measuring performance for the Gemmini and CGRA architecture with the same application, the CGRA would most likely be outperformed by the more specialized Gemmini accelerator. One reason for this is that the Gemmini project implements two DMA engines for fetching and writing data. In contrast, the Rocket-ME module works as an unoptimized DMA, which fetches and writes data to the memory.

The initial idea for the project was to make the implementation dynamically adapt to different applications, but during the work of this thesis, we found the integration more challenging than first expected. Therefore, we decided to hard code the implementation to the Sum Benchmark. The challenge with a dynamic integration is knowing what channel the CGRA is pushing out the address offset

and which channels are the input. In a 2x2 architecture, there are two different channels that the address offset can be received through. The same accounts for which channels to send the fetched data and which channel the output is put on. This information is found in the configuration bitstream but is challenging to extract in runtime. When the architecture size increases, the number of channels also increases. By hard coding this into the implementation, we ensure correct execution and behavior for only one application.

The focus of this design has been compute time and not energy efficiency or area. However, other architectures have reported an increase in energy efficiency after integrating a CGRA [28][36][37]. Based on these reports, It is reasonable to assume that we might have increased the energy efficiency. This is only an assumption and is not measured in any way.

6.5 Amdahl's Law

The effect of parallelization can be seen in the result for the Parallel stream implementation. The Stream implementation does not perform any parallel computation, but the Parallel stream does. The Parallel stream 4x3 computes two values simultaneously, and the Parallel stream 6x6 computes three values simultaneously. The result shows that as the parallelism increases, so does the performance of the implementation. However, the result is concentrated around the parallel part of the benchmark. Therefore, the theoretical speedup achievable using Amdahl's law is the time it takes to load every variable in the array and the time it takes to add the data together. Instead of setting P as core, we set it as compute elements in the CGRA array. When this parameter gets significantly high, we see that the time it takes to accumulate over an array is the same as the time it takes to fetch the data and calculate one time.

Chapter 7

Conclusion

During this thesis, we have designed and developed an integration of a CGRA into the Rocket Chip SoC. To the best of our knowledge, this has never been done before. This is, therefore, a unique contribution to the movement for faster computing.

The results presented in Chapter 5 give us some clear indications on which implementation one would prefer to integrate. From the result shown, the parallel stream 6x6 is the best performing and the desired implementation. However, this is the case when the array length is over 3700 array elements. If the accumulated array is shorter than 3700 array elements, the 4x3 implementation will outperform the larger and faster 6x6 implementation due to the significant configuration overhead. Nevertheless, these results are based on a simple benchmark that only does one accumulation. If the application does several things, it may be possible to hide the configuration overhead of the larger 6x6 architecture. If this is the case, which it would be in a real-life application, the parallel stream 6x6 implementation will be the preferred implementation for all array lengths.

The parallel stream 6x6 implementation is the implementation that has the largest speedup at the computation stage, with a speedup of 2.8×. Still, it suffers from a significant configuration overhead and needs an array length of over 3700 array elements to be the best performer. However, If the overhead is eliminated in a way, there is no doubt that this is the correct implementation also for shorter arrays.

7.1 Further Work

Several optimizations could make the accelerator perform better. One can start by optimizing the commands sent to Rocket-ME. The commands sent over contain two 64-bit values, a function number, and an optional return variable. When configuring the CGRA, the CPU uses the full 64-bit of each source register, so no optimization is available here. However, we only utilize 40-bit of the 64-bit source register when sending the input address. To reduce the communication overhead, one could set the lower 40- bits as the input address and the highest 24- bits as the

calculation length. This would save the cost of transmitting only the computation length as the last transmission. By doing this, one would also be able to integrate more features since there is now one unused function call.

A more complex optimization would be to make the CGRA calculate and fetch data from memory simultaneously without stopping the CGRA while waiting for a response from memory. This should be possible since the memory system can have several memory requests in flight. Every CGRA architecture uses two clock cycles to produce a memory address, but the memory may only use one cycle to fetch data if the data is located in the cache. Such an implementation would increase the performance of the implantation but would require more advanced controlling of the CGRA and memory module.

Another complex implementation would be to make the CGRA do several different loops at the same time. This is more of a mapper problem than the integration, but the integration would also be more complex as we need to control memory and IO for the different loops. This will only be possible for the larger architecture sizes as it would need 2x the size it would require to map one loop.

A very reasonable optimization would be to reduce the configuration time of the CGRA. Each PE uses approximately 52-54 bits to be configured, depending on the size of the architecture. As we increase the size of the architecture, the cost of configuration also increases. This configuration time could be hidden by configuring the CGRA early in the code. Meanwhile, the CPU does other things, and when the CPU sends the start command to the Accelerator, it is configured and ready.

In this implementation, the configuration clock runs at 1/2 the speed of the system clock. Making the configuration clock run at the same speed as the system clock would reduce the time spent on the configuration to a half. Another suggestion is to configure the architecture in parallel. By configuring in parallel, one can configure each row or column in series. The time spent configuring the architecture would depend on how many PE there are in each row or column. However, this would require extensive knowledge of the CGRA architecture and is another topic.

Bibliography

[1] D. Burg and J. H. Ausubel, 'Moore's law revisited through intel chip density,' *PLOS ONE*, vol. 16, no. 8, pp. 1–18, Aug. 2021. DOI: 10.1371/journal.pone.0256245. [Online]. Available: https://doi.org/10.1371/journal.pone.0256245.

[2] S. Borkar and A. A. Chien, 'The future of microprocessors,' *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011, ISSN: 0001-0782. DOI: 10.1145/1941487.1941507. [Online]. Available: https://doi.org/10.1145/1941487.1941507.

[3] G. Singer, *The history of the modern graphics processors*, https://www.techspot.com/article/650-history-of-the-gpu/part-one/, 2021.

[4] W. J. Dally, Y. Turakhia and S. Han, 'Domain-specific hardware accelerators,' *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020, ISSN: 0001-0782. DOI: 10.1145/3361682. [Online]. Available: https://doi.org/10.1145/3361682.

[5] P. Gausaker, 'A time-proportional profiler for the rocket core,' Department of Electronic Systems (IES), NTNU – Norwegian University of Science and Technology, Project report in TFE4590, Dec. 2021.

[6] *Chipyard*, https://github.com/ucb-bar/chipyard.

[7] C. community, *Chisel/firrtl hardware compiler framework*, 2019. [Online]. Available: https://www.chisel-lang.org/.

[8] *Rocket chip*, https://chipyard.readthedocs.io/en/latest/Chipyard-Basics/Chipyard-Components.html, 2019.

[9] *Sha3 rocc accelerator*, https://chipyard.readthedocs.io/en/latest/Generators/SHA3.html, 2019.

[10] *Hwacha*, https://chipyard.readthedocs.io/en/latest/Generators/Hwacha.html, 2019.

[11] *Gemmini*, https://chipyard.readthedocs.io/en/latest/Generators/Gemmini.htmll, 2019.

[12] *Nvdla*, https://chipyard.readthedocs.io/en/latest/Generators/NVDLA.html, 2019.

[13] *Rocket chip*, https://chipyard.readthedocs.io/en/latest/Generators/Rocket-Chip.html, 2019.

[14] *Rocket core*, 2019. [Online]. Available: https://chipyard.readthedocs.io/en/latest/Generators/Rocket.html.

[15] *Rocket custom co-processor*, 2019. [Online]. Available: https://chipyard.readthedocs.io/en/latest/Customization/RoCC-or-MMIO.html.

[16] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic and Y. S. Shao, 'Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration,' in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[17] C. Schmidt and A. Izraelevitz, 'A fast parameterized sha3 accelerator,' EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-204, Oct. 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-204.html.

[18] A. Gonzalez, *Building custom risc-v socs in chipyard*, https://fires.im/micro19-slides-pdf/03_building_custom_socs.pdf, 2019.

[19] *Verilator*, https://www.veripool.org/verilator/.

[20] S. Karandikar, *Structure of the risc-v software stack*, https://riscv.org/wp-content/uploads/2015/01/riscv-software-stack-bootcamp-jan2015.pdf, 2015.

[21] R.-V. international, 'History of risc-v,' [Online]. Available: https://riscv.org/about/history/.

[22] E. Technologies, *Architecture that scale for ai ad non-ai workloads*, 2021. [Online]. Available: https://www.esperanto.ai/.

[23] A. Waterman and K. Asanovic, *The risc-v instruction set manual*, version 2.2, 2017. [Online]. Available: https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[24] P. S. Pacheco, 'Chapter 2 - parallel hardware and parallel software,' in *An Introduction to Parallel Programming*, P. S. Pacheco, Ed., Boston: Morgan Kaufmann, 2011, pp. 15–81, ISBN: 978-0-12-374260-5. DOI: https://doi.org/10.1016/B978-0-12-374260-5.00002-6.

[25] A. Podobas, K. Sano and S. Matsuoka, 'A survey on coarse-grained reconfigurable architectures from a performance perspective,' *IEEE Access*, vol. 8, pp. 146 719–146 743, 2020. DOI: 10.1109/ACCESS.2020.3012084.

[26] U. of Toronto, 'Cgra-me - modelling and exploration,' 2021. [Online]. Available: https://cgra-me.ece.utoronto.ca/.

[27] B. Mei, S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins, 'Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix,' *Lecture Notes in Computer Science*, vol. 2778, 2003. [Online]. Available: https://courses.cs.washington.edu/courses/cse591n/06au/papers/fpl_03_mei.pdf.

[28] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam and C. Kim, 'Dyser: Unifying functionality and parallelism specialization for energy-efficient computing,' *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012. DOI: 10.1109/MM.2012.51.

[29] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder and t. T. Team, 'Scaling to the end of silicon with edge architectures,' *Computer*, vol. 37, no. 7, pp. 44–55, Jun. 2004, ISSN: 0018-9162. DOI: 10.1109/MC.2004.65. [Online]. Available: https://doi.org/10.1109/MC.2004.65.

[30] X. Ling, T. Notsu and J. Anderson, 'An open-source framework for the generation of risc-v processor + cgra accelerator systems,' in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 35–42. DOI: 10.1109/DSD53832.2021.00015.

[31] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu and J. Kim, 'Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications,' *2012 International Conference on Field-Programmable Technology*, 2012. [Online]. Available: https://ieeexplore.ieee.org/document/6412157.

[32] U. of Toronto, *User guide*, 2021. [Online]. Available: https://cgra-me.ece.utoronto.ca/docs/userguide.html#dfg-generation-options-label.

[33] *Vivado*, 2022. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html.

[34] P. Gausaker, *Github master thesis*, 2022. [Online]. Available: https://github.com/philisg/Master-thesis.

[35] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin and S. J. Eggers, 'The wavescalar architecture,' *ACM Trans. Comput. Syst.*, vol. 25, no. 2, May 2007, ISSN: 0734-2071. DOI: 10.1145/1233307.1233308. [Online]. Available: https://doi.org/10.1145/1233307.1233308.

[36] T. K. Bandara, D. Wijerathne, T. Mitra and L.-S. Peh, 'Revamp: A systematic framework for heterogeneous cgra realization,' in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022, Lausanne, Switzerland: Association for Computing Machinery, 2022, pp. 918–932, ISBN: 9781450392051. DOI: 10.1145/3503222.3507772. [Online]. Available: https://doi.org/10.1145/3503222.3507772.

[37] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm and A. Shriraman, 'Needle: Leveraging program analysis to analyze and extract accelerators from whole programs,' in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 565–576. DOI: 10.1109/HPCA.2017.59.

Appendix A

Additional Material

```
#include <stdio.h>
#include <stdlib.h>
#include "rocc.h"

#define ConfigLength2x2 3
#define ConfigLength4x3 11
#define ConfigLength6x6 29

void send_config(int length){
    unsigned long int config1 = 0;
    unsigned long int config2 = 0;

    for(int k = 0; k < length; k=k+2){
        config1 = cgra_configuration[k];
        config2 = cgra_configuration[k+1];
        ROCC_INSTRUCTION_SS(0,config1, config2, 0);
    }
}

void Accelerate(int InputArray[], int Output, int ComputeLength, int size) {

    asm volatile ("fence");
    int length = 0;
    switch (size){
        case 2:
            length = ConfigLength2x2;
            break;
        case 4:
            length = ConfigLength4x3;
            break;
        case 6:
            length = ConfigLength6x6;
            break;

        default:
            length = 0;
            printf("Size not set, unable to send configuration to CGRA");
            break;
    }

    send_config(length);
```

```
    //************************************************************//
    //Declear array here.//
    //************************************************************//

    // Send the first input and output address
    ROCC_INSTRUCTION_SS(0,&InputArray,&Output,1);

    // Send the array length. This will also start the calculation
    ROCC_INSTRUCTION_SS(0,ComputeLength,0,3);

    asm volatile ("fence" ::: "memory");
```

```
/* Benchmark code for buffer/stream 6x6 implementation */
#include <stdio.h>
#include <stdlib.h>
#include "rocc.h"
#include "RoCC6x6Tester.h"

#define ComputeLength 1002

void send_config(){
    unsigned long int config1 = 0;
    unsigned long int config2 = 0;

    for(int k = 0; k < 29; k=k+2){
        config1 = cgra_configuration[k];
        config2 = cgra_configuration[k+1];
        ROCC_INSTRUCTION_SS(0,config1, config2, 0);
    }
    // printf("Config Sent!!\n");
}

void array_gen(int array[]){
    int arraysum = 0;
    for(int i = 0; i< ComputeLength; i++){
        array[i] = rand() % 65535;
        arraysum += array[i];
    }
    printf("Arraysum is: %d\n", arraysum);
}

int main () {

    asm ("nop"); //Marking start of program

    int array1[ComputeLength];

    array_gen(array1);

    int sum = 0;

    asm volatile ("fence");

    asm("nop"); //Marking start of configuration

    send_config();

    // Send the first input and output address
    ROCC_INSTRUCTION_SS(0,&array1,&sum,1);
```

```
    asm("nop"); //Marking the starting of computation

    // Send the array length. This need to be the same for array1 and array2 in
    // this configuration
    // This will also start the calculation
    ROCC_INSTRUCTION_SS(0,ComputeLength,0,3);

    asm("nop"); //Marking end of computatio

    // if not here, the Sum will not be available to CPU (Datarace)
    ROCC_INSTRUCTION_SS(0,&array1, &sum,1);

    asm volatile ("fence" ::: "memory");

    printf("Program Done! Sum is: %d\n", sum);
    return 0;
}
```

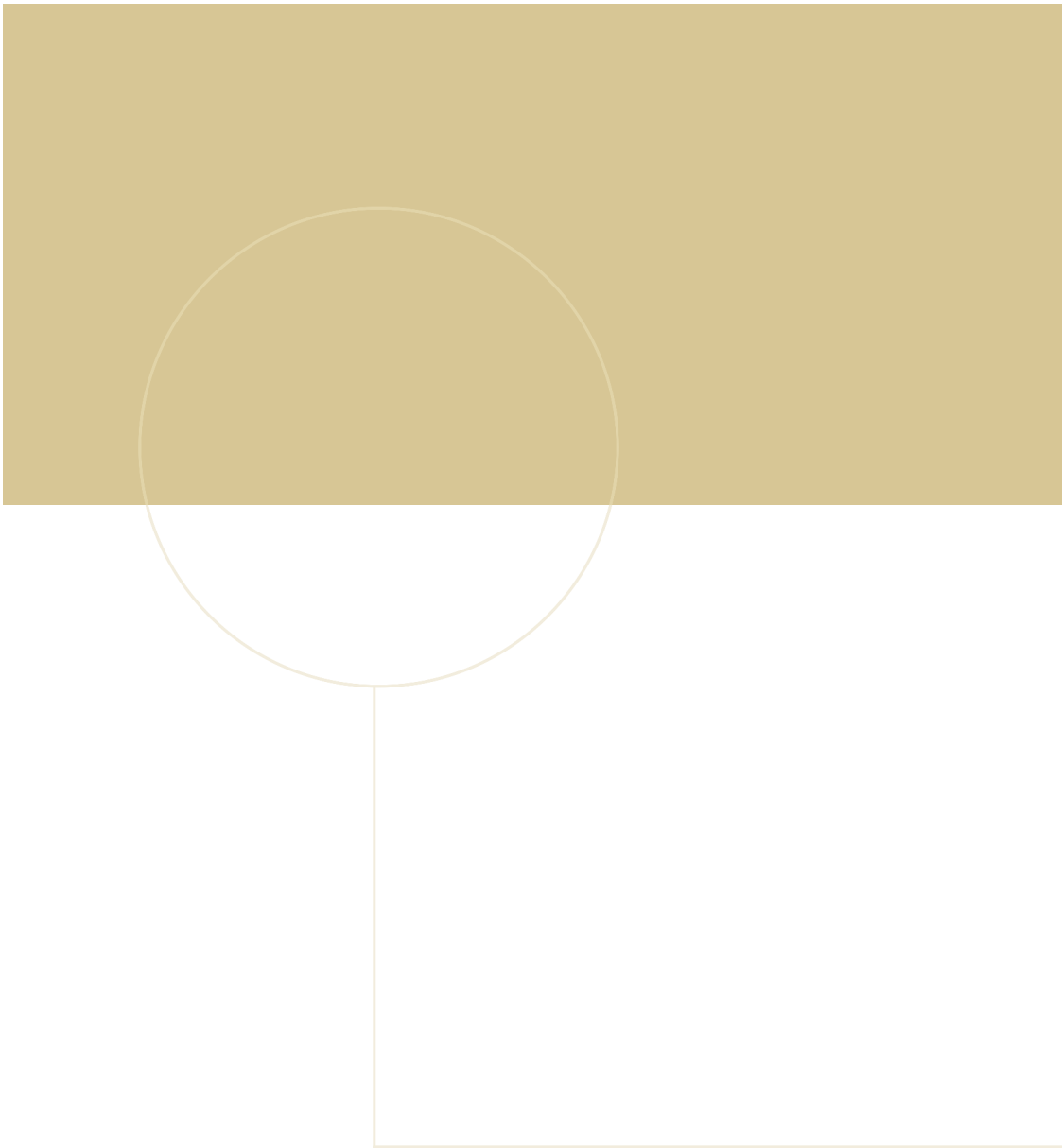
```
/* Benchmark code for Sum/baseline */
#include <stdio.h>
#include <stdlib.h>

#define ComputeLength 1002

void array_gen(int array[]){
    int arraysum = 0;
    for(int i = 0; i< ComputeLength; i++){
        array[i] = rand() % 65535;
        arraysum += array[i];
    }
    printf("Arraysum is: %d\n", arraysum);
}

int main () {
    asm ("nop"); //Marking start of program
    int array1[ComputeLength];

    array_gen(array1);

    int sum = 0;

    asm("nop"); //Marking start of configuration

    asm("nop"); //Marking the starting of computation

    for (int i = 0; i < ComputeLength; i++) {
        //DFGLoop: loop

        // Serial computing
        sum += array1[i];

        // Parallel computing
        // sum += array1[i] + array1[i+1] + array1[i+2];
    }

    asm("nop"); //Marking end of computation
    printf("Program Done! Sum is: %d\n", sum);
    return 0;
```

```
}
```

```
#!/usr/bin/python3
# Benchmark code for extracting configuration bitstream from testbench
import re
import subprocess
import sys

is2x2 = False #Are we fetching 2x2 architecture?
is4x3 = False #Are we fetching 4x3 architecture?
is6x6 = False #Are we fetching 6x6 architecture?
isDouble = False #Are we fetching the double stream architecture?

if(sys.argv[1]== "2x2"):
    is2x2 = True
elif(sys.argv[1] == "4x3"):
    is4x3 = True
elif(sys.argv[1] == "6x6"):
    is6x6 = True

try:
    if(sys.argv[2] == "double"):
        isDouble = True
except:
    isDouble = False

def extract_storage_register(filename: str) -> str:
    regex = r"storage\s*[^{]*{\s*([^}]+)\s*}"
    with open(filename) as f:
        bits = re.search(regex, f.read()).group(1)
    return bits

def remove_comments(text: str) -> str:
    return re.sub(r"\/\/.*", "", text)


def flatten_string(text: str) -> str:
    return re.sub(r"[\n\s]", "", text)


def bitstringify(text: str) -> str:
    return re.sub(r"1'b|,", "", text)


def dontcare_to_0(text: str) -> str:
    # Convert x (1'bx) to 0
    # I have no idea if this is OK or not.
    # It's comparable to DontCare, but more important for simulation (I think).
    return re.sub(r"x", "0", text)

def bitstring_to_int(bits: str) -> int:
    NrOfAdds = 64 - len(bits)%64
    if(len(bits)%64 > 0):
        for i in range(0,NrOfAdds):
            bits = "0"+bits
    LongInt =  {}
    for i in range(0,int((len(bits)/64))):
        temp = bits[(i*64):64+(i*64)]
        LongInt[i] = int(temp,2)
```

```
    return LongInt

def testbench_to_bytearray(filename: str) -> int:
    storage_register = extract_storage_register(filename)
    bits = remove_comments(storage_register)
    bits = flatten_string(bits)
    bits = bitstringify(bits)
    bits = dontcare_to_0(bits)
    return  bitstring_to_int(bits)

def to_carray_string(name: str, configuration_bytes: int) -> str:
    carray_string = f"unsigned long int {name}[CONFIGURATION_SIZE] = {{"
    for b in configuration_bytes:
        carray_string += f"{configuration_bytes[b]},"
    carray_string = carray_string[0:-1]
    carray_string += "};"
    return carray_string

def main() -> None:
    # To take from different benchmarks if we want to
    if(isDouble):
        if(is2x2):
            print("Taken from ownsum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/ownsum/testbench.v"
        elif(is4x3):
            print("Taken from ownsum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/ownsum/testbench.v"
        elif(is6x6):
            print("Taken from ownsum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/ownsum/testbench.v"
        else:
            print("Taken from ownsum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/ownsum/testbench.v"
    else:
        if(is2x2):
            print("Taken from sum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/sum/testbench.v"
        elif(is4x3):
            print("Taken from sum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/sum/testbench.v"
        elif(is6x6):
            print("Taken from sum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/sum/testbench.v"
        else:
            print("Taken from sum")
            verilog_file = "/lhome/philisg/cgra-me-ntnu-cgrame/cgra-me-ntnu-cgrame/
            cgra-me-1.0.1/benchmarks/microbench/sum/testbench.v"
    configuration_bytes = testbench_to_bytearray(verilog_file)
    configuration_size = len(configuration_bytes)
    c_configuration_size = f"#define CONFIGURATION_SIZE {configuration_size}\n"
    carray_string = to_carray_string(
        "cgra_configuration", configuration_bytes)
```

```
    if (is2x2):
        with open("cgrame-configuration2x2.h", "w") as f:
            f.write(c_configuration_size)
            f.write(carray_string)
            subprocess.run(["./MoveConfigFile.sh 2x2"], shell=True)
            print("2x2 config sent")
    elif(is4x3):
        with open("cgrame-configuration4x3.h", "w") as f:
            f.write(c_configuration_size)
            f.write(carray_string)
        if(isDouble):
            subprocess.run(["./MoveConfigFile.sh 4x3 Double"], shell=True)
            print("4x3 Double config sent")
        else:
            subprocess.run(["./MoveConfigFile.sh 4x3"], shell=True)
            print("4x3 config sent")
    elif (is6x6):
        with open("cgrame-configuration6x6.h", "w") as f:
            f.write(c_configuration_size)
            f.write(carray_string)
        if(isDouble):
            subprocess.run(["./MoveConfigFile.sh 6x6 Double"], shell=True)
            print("6x6 Double config sent")
        else:
            subprocess.run(["./MoveConfigFile.sh 6x6"], shell=True)
            print("6x6 config sent")

if __name__ == '__main__':
    main()
```



NTNU

Norwegian University of
Science and Technology