POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Open-source Full-System simulation for CPU+FPGA platforms

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Alessio Spineto**

Student ID: 247581
Advisor: Prof. Serena Curzel
Academic Year: 2025-2026

Abstract

In the landscape of modern computing, Field-Programmable Gate Arrays (FPGAs) play an important role as a key hardware component for enabling High-Performance Computing (HPC). The goal of FPGAs is to accelerate data-intensive tasks or parts of applications that require energy-efficient processing. FPGAs are configurable integrated circuits that can be repeatedly programmed, enabling continuous functional development and improvements without the need to replace physical components. FPGA hardware is expensive, and the most high-performance boards are tied with long procurement times, which lead to delays in development. To overcome this problem and improve the design space exploration phase of development, the need for accurate hardware simulators is higher than ever.

This thesis provides an accurate and flexible CPU+FPGA system simulator that takes into consideration all the hardware limitations and software overheads that other tools fail to capture. These systems, which are composed not only of an FPGA but also of a Central Processing Unit (CPU) and a memory hierarchy, need to run not only the kernel, but also the Operating System (OS), and the host application.

To aid the development of hardware designs, this work also presents itself as a tool for design space exploration, integrating High-level synthesis (HLS) tools into the simulator's workflow. Through HLS, it is possible to automatically generate different implementations of the same behavioral design, but with different characteristics, for example, the number of memory interfaces. By simulating these implementations and measuring the performance, it is possible to improve a digital design before the need to physically deploy the FPGA kernel.

Keywords: FPGA, simulation, HLS, RTL, Xilinx, C++, gem5, Verilator

Abstract in lingua italiana

Nel panorama dell'informatica moderna, i Field-Programmable Gate Array (FPGA) svolgono un ruolo essenziale come componente hardware per l'High-Performance Computing (HPC). L'obiettivo degli FPGA è accelerare task e funzioni ad alta intensità di dati o porzioni di applicazioni che richiedono un'elaborazione a alta efficienza energetica. Gli FPGA sono circuiti integrati riconfigurabili che possono essere programmati ripetutamente, consentendo uno sviluppo hardware e un miglioramento funzionale continui senza la necessità di sostituire i componenti fisici.

L'hardware FPGA è costoso e le schede ad alte prestazioni sono spesso soggette a lunghi tempi di approvvigionamento, i quali portano a ritardi nello sviluppo. Per ovviare a questo problema e migliorare la fase di design space exploration durante lo sviluppo, la necessità di simulatori hardware accurati è oggi maggiore che mai.

Questa tesi fornisce un simulatore di sistema CPU+FPGA flessibile e accurato, che tiene in considerazione tutte le limitazioni hardware e gli overhead software che altri tool non sono in grado di simulare. Questi sistemi, composti non solo da un FPGA ma anche da una Central Processing Unit (CPU) e da una gerarchia di memoria, devono eseguire non solo il kernel, ma anche il Sistema Operativo (OS) e l'applicazione host.

Per migliorare lo sviluppo di progetti hardware, questo lavoro si presenta anche come uno strumento per la design space exploration, integrando acceleratori prodotti da High-Level Synthesis (HLS) nel workflow del simulatore. Attraverso l'HLS è possibile generare automaticamente diverse implementazioni dello stesso design, ma con caratteristiche differenti, ad esempio, il numero di interfacce di memoria. Simulando queste diverse implementazioni e misurandone le prestazioni, è possibile perfezionare un progetto hardware prima di dover effettuare il deploy fisico del kernel sull'FPGA.

Parole chiave: FPGA, simulazione, HLS, RTL, Xilinx, C++, gem5, Verilator

Contents

1 | Introduction

Field-Programmable Gate Arrays (FPGAs) are adaptable hardware devices configurable via Hardware Description Languages (HDLs), such as Verilog or VHDL, to deploy custom accelerators. They have emerged as a foundational component within heterogeneous High-Performance Computing (HPC) architectures. Their goal is to accelerate data-intensive tasks or parts of applications that require low-latency processing.

Similarly, Application-Specific Integrated Circuits (ASICs) offer an additional boost in performance and energy efficiency for specialized workloads. FPGAs and ASICs share the same foundational design phase: they are modeled, tested, and synthesized using the same HDL code. However, ASICs include additional control and testing steps before production.

In the current landscape of FPGA kernel simulation, two families of simulators emerge: Register-Transfer Level (RTL) simulators and Full-System (FS) simulators. The first focuses on simulating each digital signal of the hardware, but does not take into account the physical limitations and the software overheads, tied to memory transfers, of a physical board.

Full-System simulators, like gem5, while taking into account physical limitations and software overheads, typically do not simulate the complete model of external accelerators, simulating only the behavioral logic of the accelerator at a higher level of abstraction. While this approach provides a rapid execution and low simulation times, it completely sacrifices the cycle-accurate timing and resource utilization metrics necessary for precise hardware optimization. To not sacrifice either speed or accuracy, some simulators sacrifice flexibility by being able to simulate only application-specific hardware.

HDL designs can be either written manually or generated by means of High-level synthesis (HLS) tools, which simplify the process of developing FPGA-based accelerators; by translating C/C++ code into an HDL description, it is possible not only to generate complex hardware designs starting from a few lines of code, but also to rapidly and easily change the parameters of the design, like the number of parallel execution units or the memory interfaces.

This thesis presents itself as a tool to aid the development of physical hardware solutions by providing accurate simulations and a flexible simulator capable of simulating any hardware described through HDL code, in particular, HLS-generated designs. The objective of this thesis is to provide a simulator that takes into account the limitations and overheads usually measurable after the deployment of an FPGA design, while keeping the cycle-level accuracy of an RTL simulator.

Procuring high-performance FPGA boards can be not only very expensive but also time-consuming, which can slow down the development of an FPGA kernel. To overcome this problem and improve design space exploration, the need for accurate hardware simulators is higher than ever. While FPGAs remain reconfigurable, ASICs are permanently fixed in silicon once manufactured. Because committing to an ASIC tape-out is exorbitantly expensive and time-consuming, validating the underlying HDL code through accurate hardware simulators is critical to avoid costly design errors and accelerate development.

To achieve this goal, the gem5 simulator [21] [45] has been extended and improved to include the simulation of RTL designs. To achieve this, novel memory interfaces have been implemented, tested, and measured. The same HDL design can be evaluated on various boards to assess its performance, and an accelerator design can be implemented with different parameters to evaluate the impact of these parameters on the performance of the design. In this work, gem5 will be used in combination with the HLS tool Bambu [28] to demonstrate the framework's flexibility by testing it with different hardware designs generated by Bambu.

The code of this thesis is Open source and can be found on the Barcelona Super Computing Center's GitLab at: https://gitlab.bsc.es/glopez/gem5-rtl/-/tree/gem5-hls.

In summary, the primary contributions of this thesis are:

- Extended and improved the gem5+RTL framework [46], through the implementation of gem5 objects that sync the system memory with the accelerator's memory.
- The integration of the Bambu HLS tool into the workflow of gem5+RTL, to rapidly generate, simulate, and evaluate complex and parameterized HDL hardware designs.
- Designed and implemented different FPGA memory interfaces into gem5, to connect the Bambu-generated accelerators to the full simulated system.
- Conducted extensive benchmarking, evaluating multiple accelerator designs across various metrics, and various hardware implementations.

1.1. Outline

This thesis is structured as follows:

- **Chapter 2: Background** provides an overview of the background knowledge that underlies this thesis, covering fundamental concepts about FPGA simulators, both RTL-only and FS simulations, and the generation of HDL code through the use of HLS tools.
- **Chapter 3: Framework design** presents the motivation and design choices of this thesis. Focusing on what has been implemented, by introducing key concepts and mechanisms of the gem5 simulator, and how they are utilized to fulfill this thesis's goals.
- **Chapter 4: Implementation** walks through how the ideas described in the Framework design Chapter have been implemented, describing precisely how each component has been modeled and integrated with the existing gem5 ecosystem.
- **Chapter 5: Experiments** presents the experiments, and the relative results conducted to demonstrate how this thesis reached its goals.
- **Chapter 6: Related Work** reviews the current state-of-the-art FPGA simulators, focusing on how they contribute to the landscape of hardware simulations and comparing them with this work.
- **Chapter 7: Conclusions** summarizes the contributions of this work in the landscape of FPGA simulation, discusses the limitations of this work, and reflects on future research directions.

2 | Background

This chapter will provide an overview of the concepts useful to understand the contents of this thesis, covering the field of hardware simulation and other tools to ease the development of a hardware design.

2.1. RTL simulation

FPGA kernels are written in hardware description languages (HDLs), like VHDL and Verilog, which use Register-transfer level (RTL) abstraction to describe digital circuits. Register-transfer level (RTL) is a design abstraction that models a digital circuit in terms of the physical digital signals between hardware registers and the logical operations performed on those signals. HDL and RTL makes possible to describe and then synthesize complex digital circuits easily; the circuits described in this work are computing kernels, accelerated through the use of an FPGA or ASIC. Simulating RTL means simulating each digital signal present on a hardware design, the I/O, the clock, and every single signal between every single register present on the FPGA. An RTL simulation is cycle-accurate, meaning that the simulator evaluates the value each digital signal assumes at every clock cycle of the execution.

One of the most popular tools for RTL simulation, and the one used by this thesis, is Verilator [53]. Verilator is a compiler that converts HDL code, written in Verilog, into a C++ or SystemC library. After the compilation, the code can be executed. The compiled library contains not only the full RTL abstraction written in C++, but also various utilities that add coverage and waveform support, the latter of which is particularly useful to test and debug a kernel design. To simulate the kernel, the user must write a simple wrapper containing the `main()` function that can be executed. This process is extensively applied throughout this work, and Section 4.3.4 describes this process with more detail. Since Verilator simulates and evaluates every clock cycle, if the option is enabled, it is possible to generate and then examine a waveform file, which presents all the values assumed by each signal of the hardware implementation at each instant of the simulation. For the hardware developer, this kind of file is essential, since it helps to debug and understand

the behavior of all the components of the hardware design.

2.2. Full-System simulation

The physical hardware executing an FPGA kernel does not consist solely of the FPGA itself, but also includes other components, such as a processor executing binaries and memory containing the data (see Figure 2.1). The hardware never executes only the kernel, but has to run an operating system and other pieces of software. This introduces non-negligible overhead when executing a kernel; the loading of the memory is not instantaneous, but takes time, increasing execution latency. The hardware can also introduce physical limits; the most prominent is the size of the memory bus, which limits the amount of data transferable in one memory transaction.

Some system simulators ignore these overheads and limits by emulating syscalls and passing them to the host OS, hiding the real latencies. To have a precise simulation of an RTL design, the simulator has to take these limitations and latencies into consideration. A Full-System simulator is used to simulate bare-metal hardware on which a real operating system (OS) is executed. This includes support for interrupts, exceptions, privilege levels, I/O devices, etc.

These types of simulators can boot a complete OS and execute binaries running inside that OS, in this work, a headless Ubuntu distribution of the GNU/Linux operating system is simulated, together with the host binary used to execute the FPGA kernel.

The correct simulation of the host binary execution, is particularly important, since its purpose is not only to start the execution of the FPGA kernel, but also of loading to memory the input needed by the aforementioned kernel, and retrieving the output compute, this can introduce a lot of latencies, since FPGA's workloads can be massive and the load of the input data can take a lot of time relative to execution. Hence, having insights into how much overhead these memory operations introduce is essential to the development of high-performance hardware design, aiding the developer in optimizing hardware designs, by means of fine-tuning of the accelerator's parameters, like the number of cores or the number of memory interfaces.

2.3. gem5

gem5 [21] [45] is a flexible Full-System simulator derived from the merger of the M5 CPU simulator [22] and the GEMS memory system simulator [48]; it is widely used in computer architecture and computer system design research, its flexibility derives from the

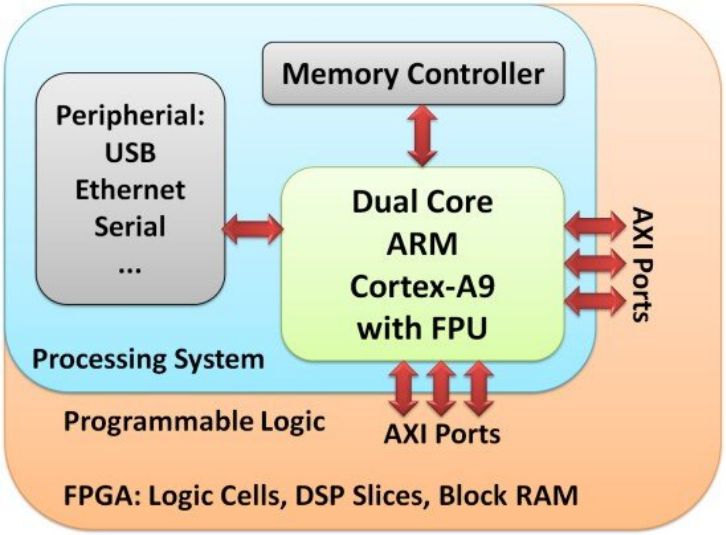

Figure 2.1: Architecture of Xilinx Zynq-7000 SoC [36].

modularity of its components: a user can change completely the hardware configuration of the simulation, changing one type of CPU with another, changing memory latencies or connecting and disconnecting components with a single line of code, all without the need to recompile the simulator. gem5 provides both Syscall-Emulation (SE) and Full-System (FS) modes; for the reasons mentioned before, in this thesis, only the FS simulation has been utilized.

2.3.1. Configuration script

gem5 is configured through configuration files written in Python, while the actual components, like the CPU, the Cache hierarchy, or the buses, are written in C++. The Python configuration and the modularity of the components enable a flexible and easy-to-configure simulation, which is particularly useful in design space exploration. The ease with which a user can change the hardware simulated is particularly useful, as it makes simulating different hardware solutions faster and easier. By modifying the configuration script, it is possible to change the simulated CPU, modify the parameters of a memory device, and even change the cache hierarchy without the need to recompile the simulator each time. An example of a gem5 configuration script is presented in Figure 2.2. This has proven to be useful when testing the same accelerator across multiple System-on-Chip (SoC) boards as described in Section 4.4; however, writing or adding a new component to the simulator still needs to recompile the simulator completely.

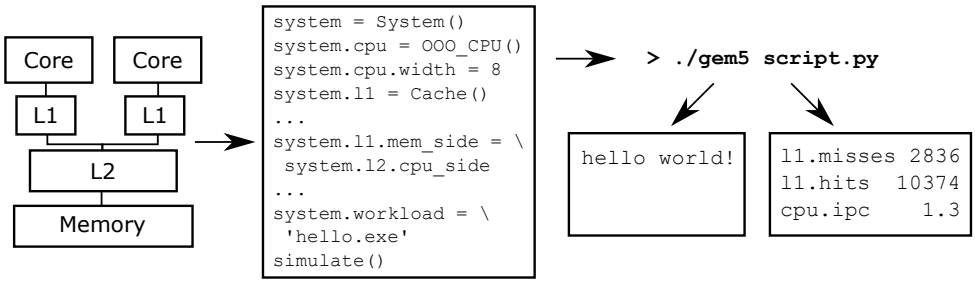

Figure 2.2: gem5 Python configuration example [45].

2.3.2. M5 syscalls

For OS simulation, gem5 provides various Linux-based options; these OS images contain binaries provided by the M5 CPU simulator, which execute custom syscalls, modeled by M5. To improve gem5's workflow, these modified Linux images execute various scripts after the canonical boot. The scripts, by executing M5 binaries, read the contents of a file that can be passed as arguments when starting the simulation. This is the bootloader script, which is typically used to either start the real workload or create a checkpoint. A checkpoint saves the current state of the simulation and enables the simulator to start from that point. For FPGA simulation, this is particularly useful since it enables the user to skip the uninteresting boot phase of the simulation. The M5 syscalls can be extended. As an example in gem5+RTL, two syscalls to start the kernel and wait for it to terminate were created; while in this work, to let the simulated CPU interact with the accelerator memory, two syscalls have been implemented: one to load data into memory, and another to retrieve data.

2.3.3. gem5 objects

In gem5, every component is an object, from the CPU to the Memory Hierarchy to the buses. Each object is an extension of the `SimpleObject` Python class, which makes it configurable using the configuration script. However, the Python class is only a shell for the real implementation; it provides the interfaces on which the object communicates with the rest of the system, and the parameters that define it. The real implementation is a C++ class, which inherits from the `SimpleObject` C++ class and contains the real logic of the component. gem5 provides various basic components that can be used or extended to create the simulation of a specific system. For example, in this thesis, the various basic classes for CPU cores have been modified and parameterized to correctly simulate the behaviour of specific CPU models.

2.3.4. GEMS Memory

As mentioned before, memory in gem5 is managed by GEMS. Since it is a gem5 object, the memory can be easily configured through the Python configuration script, and new models can be implemented using C++. To interact with the memory system, a component has to send a memory request, wrapped into a packet, and sent through a memory port. Said port eventually receives a response that gets forwarded to the object. Memory requests can be of various types. To perform specific actions in specific memory regions, for example, it is possible to invalidate a certain address inside the cache. For this thesis, the most important types of requests are read and write. gem5's memory doesn't rely on a single address, but each point in the memory has two addresses: a physical address, which corresponds to the memory of the simulated address, and a virtual one that belongs to the host machine. When implementing an object in gem5, it is usually not possible to get the physical address of something directly; however, using standard C++ pointers, it is possible to get the virtual address. It is possible to translate a virtual address into a physical one using translation tables, accessible through gem5's library. To correctly implement the latencies and measure timings, memory packets have to be sent through memory ports, which can be connected one-to-one, using the Python configuration file. It is possible to connect multiple ports using buses, called `XBar` in gem5.

2.3.5. Statistics

One of gem5's most prominent features is its ability to measure statistics. After a simulation, the M5 operation `dump_stats` is called, generating a plain text file containing useful statistics about the simulation. The measurements performed by gem5 encompass a wide range of statistics, including the time simulated, the power consumed by the hardware, and the latency of the memory. These statistics can be easily implemented and extended inside an object, using C++, to measure useful metrics, for example, the number of cycles that happened during the execution of a simulated kernel, or the average latency of the requests to the memory.

2.4. High-level synthesis

High-level synthesis is the process of generating an RTL implementation of a behavioral description [23]. HLS aids the development of hardware solutions by enabling a developer to automatically generate different versions of a hardware design, improving the design exploration stage of development. This is achieved through lexical analysis and parsing of

a function, along with any pragmas, written in C or C++; from this, the control flow graph and data flow graph are extracted. A scheduling step models the graphs into a finite state machine (FSM) following strict timing restrictions. The function to accelerate is now split into smaller, easily synthesizable blocks; these blocks are then bound to specific hardware resources, and the same is done for any memory allocation needed. Finally, the output HDL implementation is generated.

2.4.1. Bambu

In this work, the RTL implementations of the kernels simulated have been generated using the HLS tool Bambu [28]. Bambu is a free and open-source framework whose aim is to aid the development of hardware through high-level synthesis. Bambu takes as input a behavioral description of a kernel, written in annotated C/C++, and generates an RTL implementation of the kernel written in HDL. The framework also supports hardware synthesis and simulation through RTL simulators such as Verilator and Xilinx ISIM/XSIM.

2.4.2. Bambu's synthesis

Bambu's HLS flow, described in Figure 2.3, is split into three separate phases: front-end, middle-end, and back-end. The front-end consists of the parsing of the C/C++ code to accelerate with the GNU Compiler Collection (GCC) or Clang: GCC, by means of plugins, extracts its internal representation in Static Single Assignment form of the initial C/C++ code; similarly, the Intermediate Representation (IR) generated by Clang/LLVM is reduced, through the use of plugins, to the one used by Bambu. From the extracted IR, during the middle-end, further analyses are performed, and additional internal representations are built, such as Call Graph, Control Flow Graphs, Data Flow Graphs, and Program Dependence Graphs. To these additional representations, a set of analyses and transformations is applied (data flow analysis, loop recognition, dead code elimination, constant propagation), performing a further optimization. A relevant optimization done at this stage is the optimization of multiplications and divisions by a constant. These operations are typically transformed into operations that use only shifts and adds to improve area and timing.

Finally, during the back-end, the actual HLS process, described in Section 2.4, is performed.

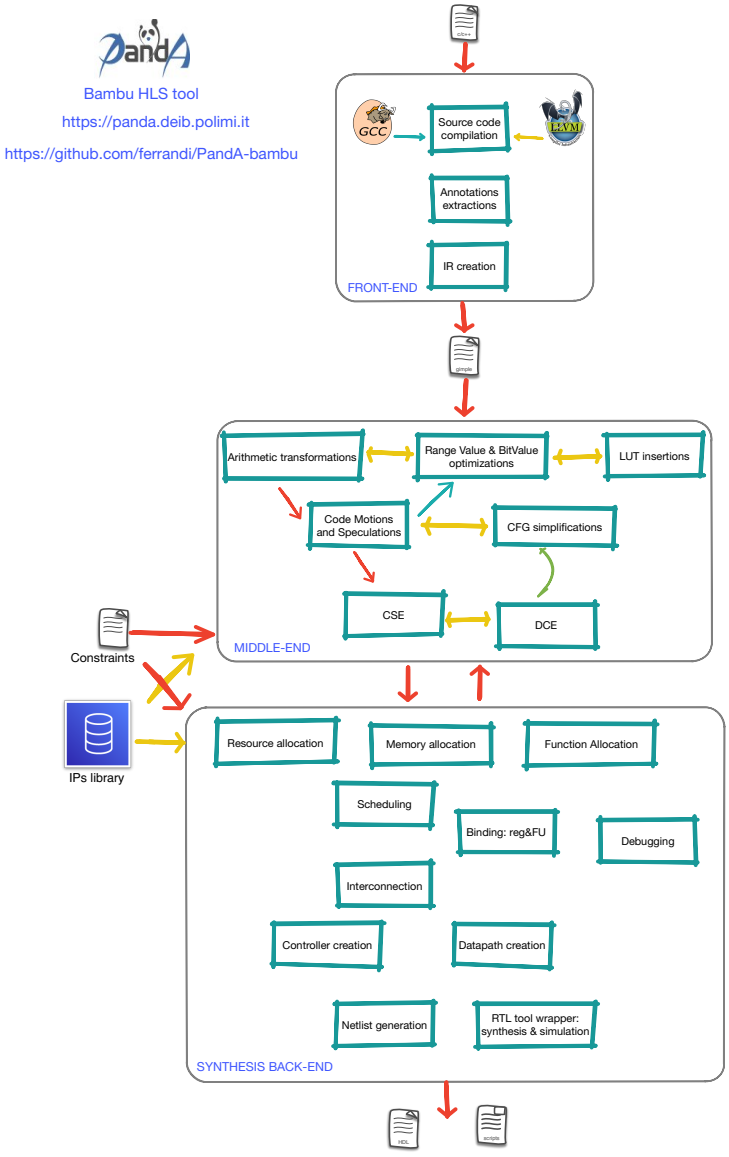

Figure 2.3: Bambu Compilation flow [28].

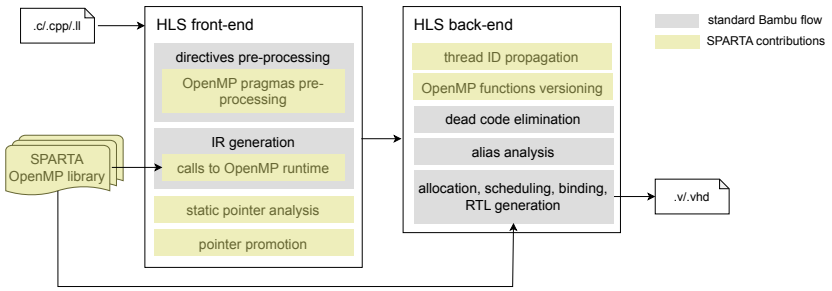

Figure 2.4: Enhanced SPARTA-Bambu HLS flow [34].

2.4.3. SPARTA

In the modern High-Performance Computing (HPC) landscape, the use of FPGA accelerators to address the data-intensive workloads of graph analysis is in constant growth. To improve the synthesis of these graph analysis kernels, Bambu has been extended with Synthesis of PARallel multi-threaded Accelerators (SPARTA) [34]. SPARTA improves the Bambu HLS flow by taking as input kernel specifications written in OpenMP and applying optimizations specific to parallel computations, as depicted in Figure 2.4. Accelerators generated with SPARTA can exploit both spatial and temporal parallelism, hiding the latency of external memory accesses through context switching (CS) across multiple execution threads.

2.5. Acronyms

The following table is a collection of the most important acronyms used throughout the thesis.

Table 2.1: List of Acronyms.

Acronym	Definition
General	
ASIC	Application-Specific Integrated Circuit
AWS	Amazon Web Services
CPU	Central Processing Unit
DMA	Direct Memory Access
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HDL	Hardware Description Language

Continued on next page

Table 2.1 – continued from previous page

Acronym	Definition
HLS	High-Level Synthesis
HPC	High-Performance Computing
IP	Intellectual Property
IR	Intermediate Representation
ISA	Instruction Set Architecture
ML	Machine Learning
OS	Operating System
RHBD	Radiation Hardened By Design
RISC-V	Reduced Instruction Set Computer-V
RTL	Register-Transfer Level
SoC	System on Chip
VCD	Value Change Dump
VHDL	VHSIC Hardware Description Language
PMU	Performance Monitoring Unit
CO	Cores
CS	Context Switches
FS	Full-System
SE	Syscall-Emulation
Memory	
DDR	Double Data Rate
HBM	High Bandwidth Memory
MSHR	Miss Status Holding Register
SPM	Scratchpad Memory
AXI	Advanced eXtensible Interface
CPU Microarchitecture	
ALU	Arithmetic Logic Unit
BTB	Branch Target Buffer
DTLB	Data Translation Lookaside Buffer
IEW	Issue, Execute, Writeback
IQ	Issue Queue
ITLB	Instruction Translation Lookaside Buffer
LQ	Load Queue
LSU	Load/Store Unit

Continued on next page

Table 2.1 – continued from previous page

Acronym	Definition
ROB	Reorder Buffer
SQ	Store Queue
TLB	Translation Lookaside Buffer
WB	Writeback
Software and Tools	
CUDA	Compute Unified Device Architecture
GCC	GNU Compiler Collection
LLVM	Low Level Virtual Machine
NVDLA	NVIDIA Deep Learning Accelerator
SST	Structural Simulation Toolkit
Benchmarks	
BFS	Breadth-First Search
CC	Connected Components
PRP	PageRank
TRC	Triangle Counting

3 | Framework design

This section aims to provide the details of the goals of this work and the methodology used to achieve such goals. The chapter will first introduce the motivations that led to the development of this thesis, specifying the strengths of this tool; then it shifts the focus to how the simulator works, what it does, its metrics collection processes, and how it evaluates the accelerators it simulates.

3.1. Motivations

The primary objective of this thesis is to provide a flexible and accurate simulator framework for FPGA-accelerated systems that utilize kernels generated through High-Level Synthesis (HLS) tools. To achieve this goal, the gem5+RTL framework [46] was extended by implementing multiple types of memory interfaces, making it possible to easily integrate new accelerators, generated by Bambu, into the simulator.

Most SoCs with integrated FPGA hardware mount ARM-based processors, so the gem5+RTL framework has been configured to model two ARM-based SoC Full-System environments, one for ARM64 and another for ARM32; the two ARM-based SoC Full-System environments include models of both hardware and software, like memory, CPU, and accelerator, but also operating system and guest applications to run the simulated kernel. These two environments enable the FS simulation of the boards used in this thesis, described in Section 4.4, but can be used to simulate any ARM-based FPGA-accelerated system.

In the current landscape of high-performance computing, procuring physical hardware or manufacturing ASICs is financially prohibitive and introduces significant logistical delays. Even before acquiring the target hardware, a developer can utilize this platform to explore various implementations of a design by evaluating the same kernel with different specifications, such as memory interfaces or thread counts. This work positions itself as a critical validation and exploration tool that improves the traditional development cycle. As shown in Figure 3.1, a traditional HLS workflow evaluates the performance only after the host application is developed and the design is physically deployed on the

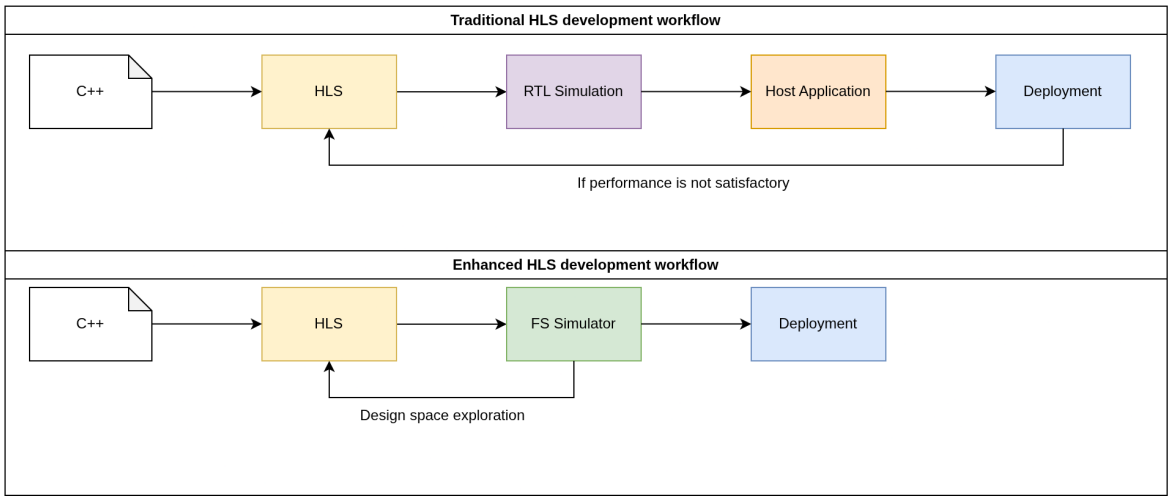

Figure 3.1: Traditional HLS design flow opposed to HLS design flow enhanced with the use of FS simulation

actual hardware. If the performance results are unsatisfactory, the developer must loop back to the HLS stage, tweak the design, and repeat the loop until the design performs as desired. The enhanced workflow introduces the FS simulator, which replaces the RTL-only simulator, enabling design space exploration without having to deploy the design. Paired with the ability of an HLS tool to automatically generate kernels with varying degrees of parallelization, this enables developers to iteratively test, measure, and optimize ASIC and FPGA designs efficiently, long before physical deployment is required.

3.2. gem5+RTL

As mentioned above, this thesis is an extension of the gem5+RTL framework. The original work already proposed an easy-to-use, accurate Full-System simulator; the main intent of this work is to extend its capabilities, in particular its flexibility, by implementing novel memory interfaces, making it possible to integrate more accelerators with this framework, each with different memory interfaces and different parameters. The original work only implemented two use cases: the NVDLA accelerator and a Performance Monitoring Unit (PMU), and while the original framework's NVDLA integration supports multiple NVDLA instances, it does not support multiple memory interfaces per instance.

3.2.1. Flexibility

The framework utilizes Verilator [53] to translate HDL code into a C++ library; for this reason, this framework supports any accelerator designed in Verilog or VHDL, the HDL

code can be written by hand, but also generated from an HLS suite such as Bambu [28] or Vitis HLS [8], the use of such tools is particularly useful since it makes it possible to generate various copies of the same accelerator, but with different parameters to enable optimization and design space exploration, in this work, the accelerators tested vary in the number of memory channels and number of threads, which, as anticipated in Section 2.4.3, are used to hide the latencies tied to memory requests.

3.2.2. Precision

The accelerators are simulated with cycle-level accuracy, meaning that each cycle of execution of the FPGA is simulated. This means that for both measuring and testing purposes, each accelerator signal is simulated and can be read from a VCD file generated by gem5+RTL.

Statistics such as simulated time or number of cycles executed are measured by gem5, and are reported in `stats.txt`, a human-readable file automatically generated by the simulator that can be used to check more general statistics like the aforementioned time simulated; for example to understand the performance of a design or of a specific hardware; however gem5's statistics are not limited to the overall performance of the simulation, `stats.txt` also contains more specific data, such as time spent reading and/or writing data, this can be used in the design space exploration phase of development to understand bottlenecks and the impact of the type of memory used. For example, while cosimulation in tools like Bambu allows developers to set an arbitrary, fixed number of cycles per memory request, real-world latencies depend on the specific memory technology and CPU frequency. Gem5 integrates these realistic hardware limitations, providing an accurate FS simulation.

3.3. Accelerator Wrapping

As mentioned before, the accelerator design written in HDL has to be translated into a C++ library; this is easily done and completely handled by Verilator [53], which generates a directory containing a C++ header file and the rest of the library. As mentioned in Chapter 2, the output of Verilator is a complete C++ model of the hardware; an example of this flow is given in Figure 3.2. To integrate the library into gem5, a wrapper is needed to command and check the signals exposed by the top interface of the library; these are mostly memory interface signals and control signals like reset, start, and the clock signal.

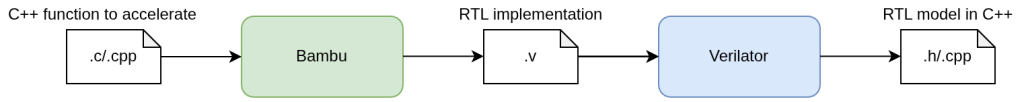

Figure 3.2: Generation of a C++ hardware model using Verilator and Bambu.

3.3.1. Object Implementation

To interact with the gem5 simulator, the accelerator also needs a proper gem5 object that handles requests and responses from the memory and the CPU. This object interacts with the Full-System using memory requests and can invoke the wrapper's functions directly; by accessing the wrapper's requests, the object can translate them into a gem5 memory request, then, after receiving the response, it gives it to the wrapper. Another function of the object is to synchronize the wrapper with the CPU clock, which gem5 provides.

3.3.2. Configuration

The RTL design cannot be configured by gem5, but all its characteristics, like the number of memory interfaces, are set once Bambu generates the hardware description and Verilator compiles the library. The object has to be configured to match the kernel's specification, like the number of interfaces and frequency. This is done through gem5's configuration scripts that are also used for device configuration.

3.3.3. Host Binary

To execute the kernels, a validation binary has to be cross-compiled and run on the simulated machine; the binaries share a small utility library and utilize gem5's syscalls to interact with the system and start the kernel or interact with the memory. Each binary takes data from a `graph.h` header file containing the dataset of a benchmark, vertices, edges, and a set of results. In the traditional HLS workflow, binaries to run the RTL implementation have to be developed to deploy the hardware design. In the Xilinx workflow, these binaries use the xrt library [7] to manage the memory and the execution during deployment. The host binaries of this thesis mimic the style and structure of the official AMD Vitis examples [59], each xrt call is substituted by a gem5 syscall, with the sole exception being the benchmark's data/input: in the original examples, data is read from binary files, but to reduce simulation times, the data is read offline and translated into C++ arrays that are included in the aforementioned `graph.h`.

3.3.4. Scripting and gem5 simulation

The execution of the host binaries, to start the accelerator, is taken care of by the OS simulator. As mentioned before, the operating systems simulated are modified versions of GNU/Linux provided with special binaries to execute m5 operations. Once the simulation is started and the OS is booted, a script is executed by the simulated OS that checks for a checkpoint; if it does not exist, it creates one and exits the simulation, using `m5_exit`. The checkpoint can be resumed, and now the script executed again finds the checkpoint and starts a validation script; the script then loads the host binary and executes it, effectively starting the simulation. During simulation, the input is first processed and loaded into the memory, using a polling method, the binary checks for the termination of the load process. The accelerator is started, and the same polling process is used to check the conclusion. After the accelerator finishes processing, the memory is retrieved, and the output is checked for correctness. The host binary then invokes `m5_exit`, which terminates the simulation, and the stats are dumped, together with a `.vcd` file containing all the values that each FPGA signal assumed during simulation. The process just described is summarized in Figure 3.3.

3.4. Measurements Methodologies

To evaluate a simulation, gem5 provides a system of statistics; the data is collected during execution and then saved in a plain text file. gem5's statistics already include important data about the full simulation of the system, but for this thesis, the measurement of new statistics had to be implemented, such as accelerator execution latency, number of cycles of the accelerator execution, number of memory requests, and total memory request latencies (see Table 3.1). After the simulation dumps all the statistics, they can be easily read by the user or parsed to automatically generate graphs containing all the relevant information.

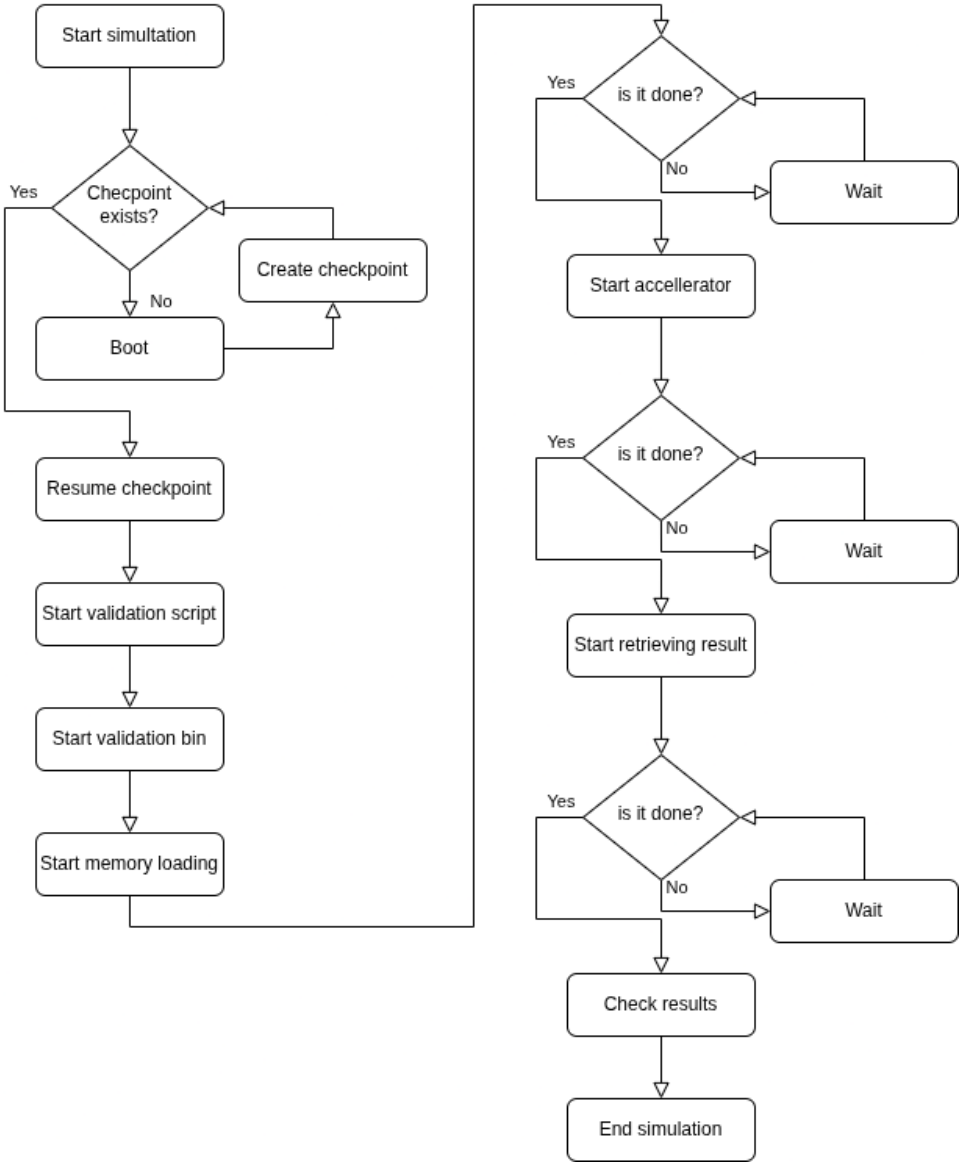

Figure 3.3: Full simulation flow.

Table 3.1: Most relevant statistics and measurements

Statistic	Description
Total system latency	The complete execution time of the system, including software overheads, OS interventions, data transfers, and the hardware execution itself.
Accelerator execution latency	The total latency taken by the hardware accelerator to complete execution, excluding software setup and OS overhead.
Number of cycles of accelerator execution	The total latency taken by the hardware accelerator, expressed in number of cycles.
Number of memory requests	The total count of read and write transactions requested by the accelerator during execution.
Total memory request latencies	The cumulative sum of the latencies experienced across all memory transactions throughout the hardware execution
Average memory latency	The average latency per memory request, calculated as the total memory request latency divided by the number of memory requests.

4 | Implementation

This chapter walks through the concrete implementation of the framework described in Chapter 3, starting from a complete overview of the hardware system architecture to simulate, down to the gem5 object implementations and the guest binaries simulated. The various sections will describe how this work is actually implemented, and also how each part contributes to achieving a flexible and accurate Full-System simulation.

4.1. System Architecture

To obtain a flexible framework, a basic, generic, and highly configurable system architecture is needed. gem5+RTL already provides a solid base; this framework aims to extend and improve it. This base architecture is a customized configuration called `fs_bigLITTLE_RTL.py`. In Gem5, the simulator configurations that describe the system to simulate are Python files, easy to configure and change, without the need for compilation. gem5's base configuration provides a system with a processor and a memory hierarchy; gem5+RTL attaches to the CPU an object representing the accelerator and connects it to a memory bus, so that it can interact with the memory, as depicted in Figure 4.1. What CPU and what memory to simulate can be configured by modifying some parameters in the file or by passing them as arguments when running gem5, for example "`./build/ARM/gem5.opt --cpu-type timing --big-cpus 1 --little-cpus 0 --last-cache-level 2 --caches`" runs a simulation where the CPU has a single performance core and a 2-level cache

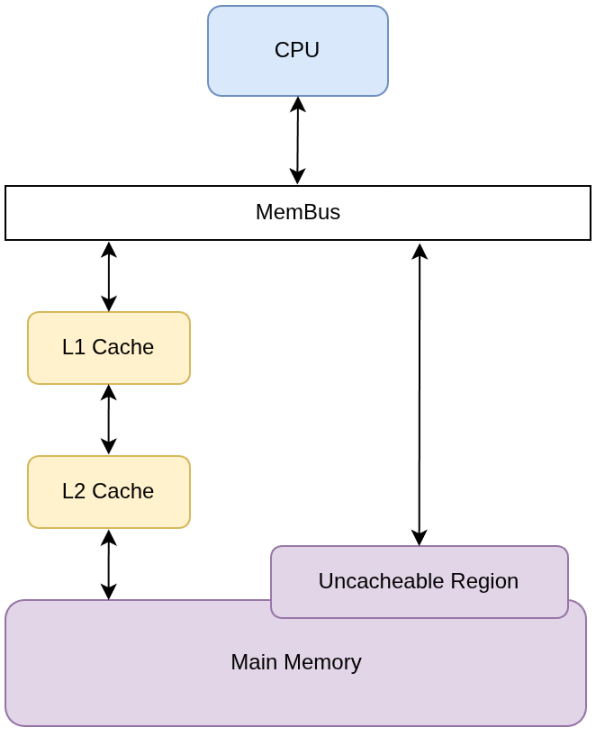

Figure 4.1: Simplified view of the CPU side memory system.

4.2. Memory utilities

Accelerators do not use the same memory as the rest of the system, but use a dedicated memory space, a Scratchpad memory (SPM). This memory is only used by the accelerator and only contains the input and output data of the FPGA, but this memory must first be loaded before kernel execution and then retrieved after the execution is complete; for these operations, two gem5 objects have been developed: memLoader and memRetriever. These two objects act as Direct Memory Access (DMA) devices that can be invoked by the CPU and proceed to store or get a region from the SPM. This section will focus mostly on memLoader, since both objects share the same structure and mechanisms.

4.2.1. Memory configuration

To correctly simulate these DMA-like memory transactions, the memory has to be correctly configured; the layout is composed of three regions, as represented in Figure 4.2: a "normal" cached region accessible by every component of the system, an uncacheable memory, where the input data exists before being loaded into the SPM, and a private

region, representing the SPM, accessible only by the accelerator, memLoader and memRetriever. The input data region is specifically configured as uncacheable because the data contained is transferred directly to the SPM in a single pass; since it is never reused, it should never be stored in the cache. The first two memory regions share the same characteristics, while the SPM has its own bandwidth and latencies. Moreover, the SPM region is split into individual banks, one for each memory channel of the accelerator.

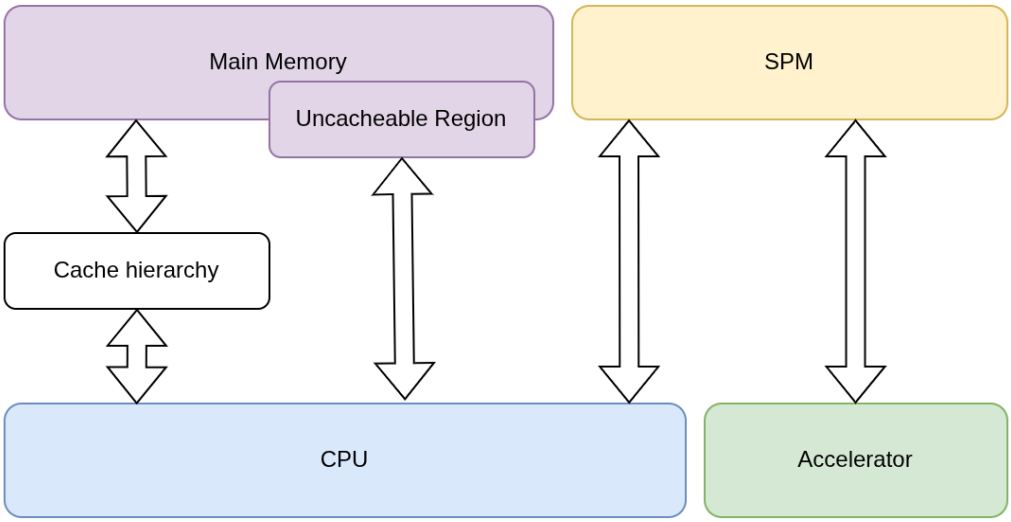

Figure 4.2: Memory layout of a generic system.

4.2.2. gem5 memory transactions

Before delving into the memLoader and memRetriever implementations, it is important to explain how memory transactions work inside gem5. gem5 memory system provides for the same memory two addresses, a physical one and a virtual one; the physical address refers to the simulated memory and a memory request is needed to either read or write to that address, the virtual one corresponds to the address of the host machine, it is useful for code implementation, write function and access data inside the gem5 object, but it's useless to get data on the simulated machine. When implementing the functionalities of a gem5 object, getting a physical address is not a trivial task, while getting a virtual one is very straightforward using C++ pointers; to get the physical address of an object while having only the virtual one, gem5 provides a `translateTiming` function that translates the virtual addresses into physical ones, this is particularly useful when passing input data to the accelerator. Once the physical address is obtained, it is possible to create a request to the memory. There are various request types in gem5, but for this thesis,

the only useful ones are `ReadReq` and `WriteReq`, which create a read and a write request, respectively, as shown in the official gem5 doxygen [31]. Both types of requests need as arguments a physical address and the size of the data, then if the request is a write, a pointer to the data to write has to be provided and set. A packet containing a copy of the request is then created, the write data is added to the packet if needed, and the request can finally reach the memory. A normal gem5 object can send a request directly to the memory and immediately receive a response; however, to measure latency and time spent, the packet can be sent by a memory port, which schedules when to send a packet, typically using a simple queue, and keeps track of the time passed between request and response. Once the request is sent through a memory port, the same port eventually receives a response, and the port then forwards it to the accelerator object itself, which receives the data, in case of a read, or a confirmation of the successful transfer, in case of a write.

4.2.3. memLoader

The purpose of memLoader is to take the data contained in the uncacheable region of the memory and load it into the SPM. To do so, the object needs to know the size of the data, the physical address from which to take the data, and the physical address of the destination. To get these parameters, the CPU has to send them to memLoader, using a memory request. To trigger this memory request, a new syscall is implemented and called by the CPU inside the simulated binary. The syscall takes as arguments the two addresses and the size mentioned before and forwards them to the CPU, which then makes a memory request to the memLoader object, which will respond only after the loading is done.

4.2.4. Initialization

To make memLoader more flexible, the destination address is optional. When the object receives the start request from the CPU, it gets its arguments and checks if the destination physical address is zero. In this case, the destination address will be the base address that is passed as a parameter in the configuration of the system; otherwise, it will be the address received. If only a single memory load/retrieve is performed, the first option makes it possible to modify the memory layout without the need to recompile the host binary, while the second option enables the user to store data in specific parts of the memory, which is particularly useful when dealing with multiple memory banks

4.2.5. Request forwarding

Once started, memLoader reads data from the uncacheable memory, it starts a read request, forwards it to the memory port, and when it receives a positive response, it saves the data read in an array, then it checks if the number of bytes read is equal to the size given by the CPU; if not, it starts a new request. The size of each request is determined by the size of the cache line, typically 64 bytes, which is configured by gem5; each request's size is equal to the maximum between the cache line size and the remaining bytes to read. After every read is completed, the same process starts, but to write the data to the destination address, the computation of the write requests' size is the same as that of the read requests. The full flow is summarized in Figure 4.4. Each request is not directly sent to the memory, but as mentioned, it is first sent to the memory port, and then the port sends it to the memory. This mechanism, represented in Figure 4.3, is meant to ensure correct measurement of the time needed to fulfill each memory request. This functionality can be enabled and disabled in the configuration files of the system.

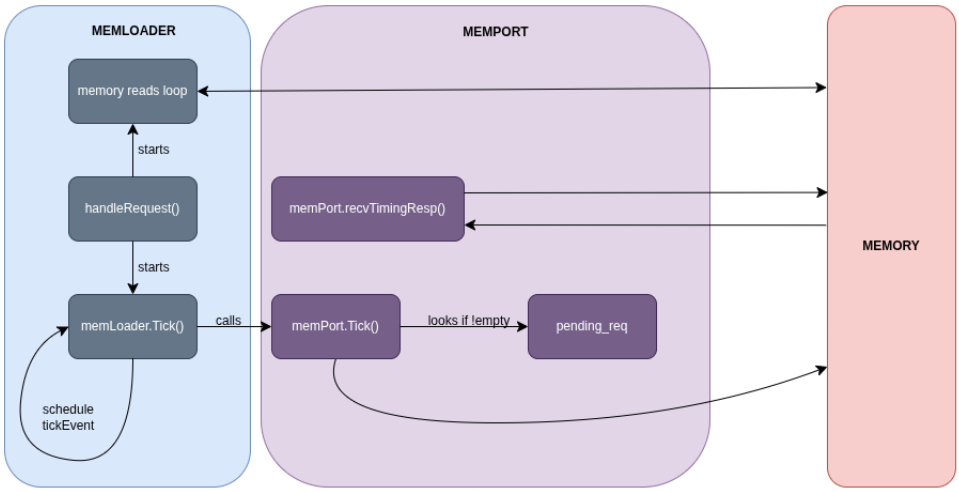

Figure 4.3: Flow of a memory request.

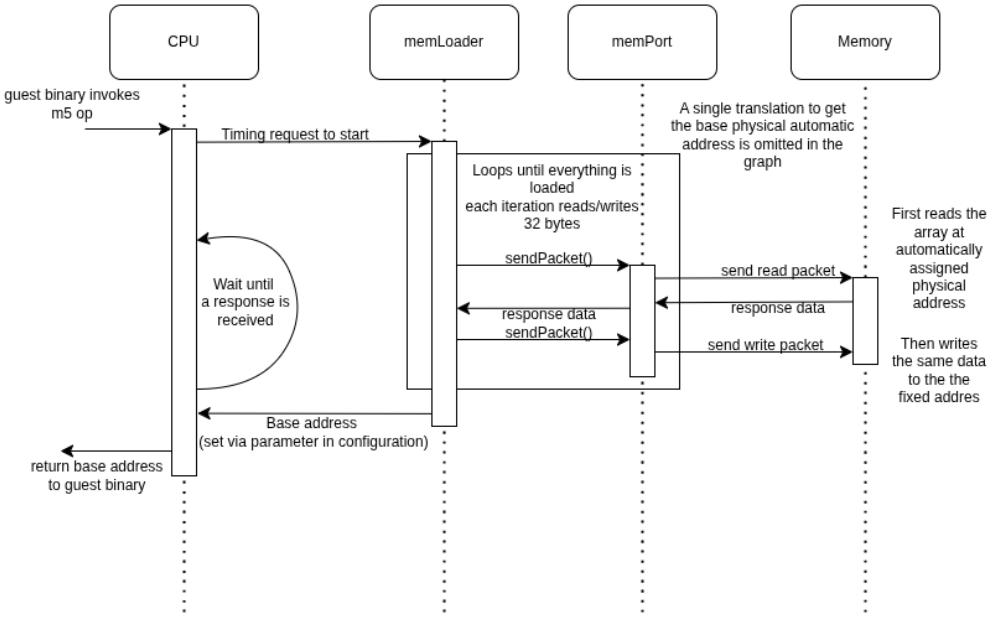

Figure 4.4: memLoader sequence diagram.

4.2.6. CPU response

Once every write is completed, the object sends a memory response to the CPU, containing a physical address, which corresponds to the physical address where memLoader wrote, so either the one passed by the CPU itself or the one configured. It is worth noting that memRetriever shares the same structures and functionalities, but instead of reading from the uncacheable memory and writing to the SPM, it does the contrary, reads from the SPM and writes to the uncacheable memory.

4.3. Accelerator modeling

The FPGA accelerator's implementation is split into two parts: a gem5 object and a Verilator wrapper; this section will go over the full implementation process of the hardware, focusing on the different types of accelerators modeled and the flexibility of the implementation itself.

As mentioned in Section 3.3, to simulate the accelerator, this work uses Verilator, which creates a C++ library, then the library is wrapped by a C++ object that directly interacts with the RTL signals of the accelerators exposed by Verilator. To make this wrapper

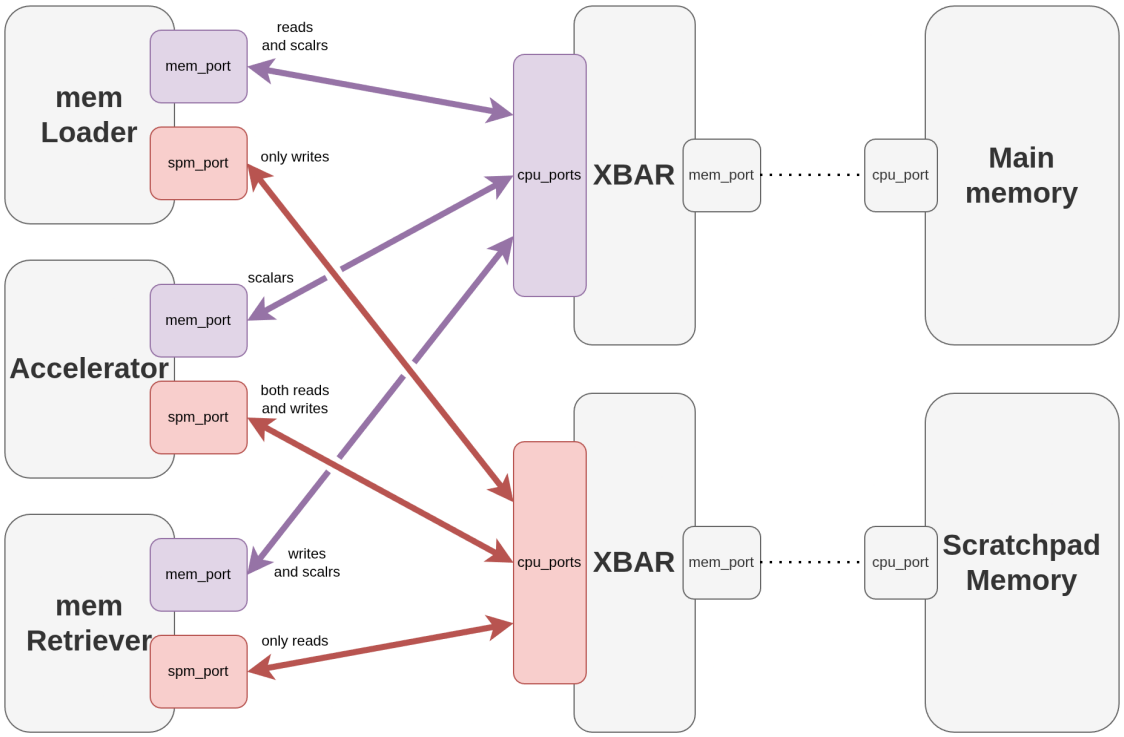

Figure 4.5: Accelerator, memLoader, and memRetriever connections.

interact with the rest of gem5, an object is created that can invoke the wrapper's functions, which enable it to start it, get its memory requests, and check if it has finished a computation.

The accelerator object has to be connected to the memory, but as mentioned before, it should only access the SPM, except for exchanging scalars with the CPU to start and stop the execution. In the system configuration, the accelerator and the memory utilities have been connected to two `XBar` objects, which enable multiple connections to the same object; one `XBar` is connected to the main memory and the other to the SPM. The layout just described is represented in Figure 4.5.

4.3.1. Object Implementation

To model the object, the official gem5 documentation was followed [32] [31]. The basic structure of a functioning RTL object was already present in the original work gem5+RTL, but for this thesis, it was modified and extended to model multiple types of accelerators. Unfortunately, for each type of accelerator, a new object has to be implemented. This is because different accelerators have different inputs, different interfaces, and different

memory layouts. If two accelerators share the same interface, input, and memory, then it is possible to use the same object. For this reason, the focus of this implementation is to make the integration of new accelerators as easy as possible.

Like the memory utilities, the CPU needs to be able to start the RTL object execution; gem5+RTL already provides a syscall that makes the processor send a memory request to the accelerator. The request contains the memory address of all the input necessary for the accelerator to start: the arguments of the accelerated kernel, a `start_port` signal, and a `base_address` representing the start of the SPM. The address received is a virtual one, so it first needs to be translated, once the translation is finished, a memory request to the physical address is created and sent to the memory, the size of the request is the sum of the sizes of the full input, a struct with all the data needed is made accessible by the wrapper, the size is computed through the use of C's standard library `sizeof()` function. Once the memory request gets a response, the input is passed to the accelerator, which resets and starts. The accelerator then uses gem5's event scheduling API to schedule a tick event.

4.3.2. Operational frequencies

In gem5, a tick is the fundamental unit of simulated time (1 picosecond by default), and each CPU clock cycle spans a number of ticks equal to the clock period. Every time the CPU completes a cycle, it triggers a clock event, and a gem5 object can schedule the call of a function on each such event to be synchronized to the rest of the system. It is also possible to schedule an execution after N CPU clock cycles, which makes it possible for the accelerator to run at a different frequency than the CPU. This mechanism is used in this work to account for the different working frequencies of the accelerators; as mentioned in Section 5.1.1, not every accelerator works at the same frequency, so the accelerator has a frequency multiplier parameter that can be set via system configuration and modifies the working frequency without the need to recompile the simulator. The factor N has to satisfy the following equation:

$$N = round(\frac{\text{CPU frequency}}{\text{FPGA frequency}}) - 1 \tag{4.1}$$

When configuring clock domains in gem5, the division factor (N) must be an integer. This restriction introduces an inherent approximation error for certain target speeds. For instance, with a base CPU clock of 1GHz, an exact rate of 150MHz is unachievable because it requires a fractional divider. Instead, the simulation must approximate this

target, yielding an effective speed of 142.857 MHz. Therefore, for the configurations depicted in Figure 5.1, with a CPU running at 1GHz, the factors used are: 9 for 100MHz, 6 for 150MHz, and 4 for 200MHz.

4.3.3. Object operations

After the wrapper is started, the accelerator object continuously receives a tick event, which calls the object's function `Tick()`; inside the function, the accelerator first checks if the wrapper has finished the computation, in that case it sends a memory response to the CPU with the output data, if present, and stops, otherwise it propagates the tick to the various memory ports of the object, one for each interface of the accelerator, the tick is then propagated also to the wrapper, then the request queues of the wrapper are checked, and in the end it reschedules a `Tick()` after the next N CPU clock cycles, where N is the aforementioned frequency parameter. If the wrapper requests a memory operation, it inserts it into a queue. When the object checks the queues, one for each interface, it makes a gem5 memory request for each of them in order. Since an accelerator might be oblivious of the underlying memory layout, before sending a request, its address might be modified to account for the memory layout of the system. The requests are propagated through a memory port that stores the current tick and the request, then sends the packet to the memory. When the packet is processed, and the memory returns a response, a memory port receives it, computes the number of ticks passed between sending the request and receiving the response, then forwards it to the accelerator, the latter adds the response to a response queue of the wrapper. The memory requests and responses are processed asynchronously, but the queues, both response and request, are checked synchronously during a tick.

If the goal is not to measure the performance of an hardware design, but just to validate it, to improve simulation times, it is possible, like for memLoader, to disable the timing measurement and the memory latencies by setting a flag inside the configuration, while the computation of the time elapsed is simply not executed, to remove memory latencies, the object uses gem5's `sendFunctional(packet)` which makes a functional, instant request to the memory, then calls `handleResponse(packet)` directly, bypassing completely the memory ports. The whole operation sequence is reported in Figure 4.6.

4.3.4. Wrapper Implementation

The implementation of the wrapper starts from the generation of the kernel library using Verilator, as described in Section 3.3. Verilator takes as input either a Verilog or VHDL

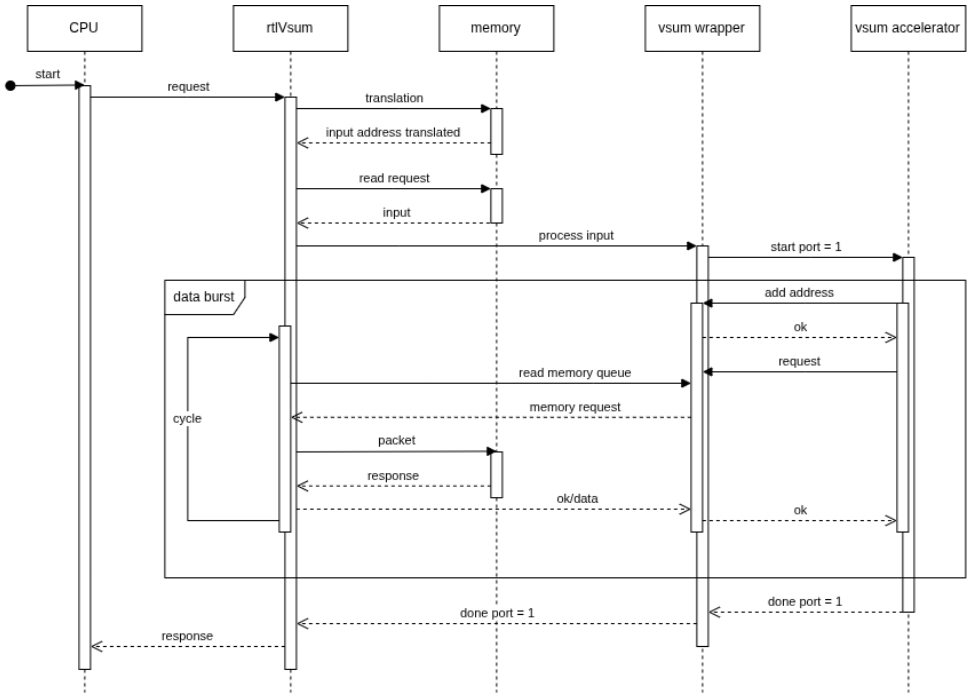

Figure 4.6: AXI interface sequence diagram.

file containing the kernel's code, and it then produces a full library that provides a C header file .h, which exposes all the physical signals of the accelerator, flagging them as input or output and giving them a size. The wrapper's method can modify input signals and read output signals directly.

The wrapper's purpose is to simulate the external signals that the accelerator would receive in a real execution. To do so, it provides various functions that the gem5 object can invoke to interact with the rest of the system. The object can give to the wrapper the input passed by the CPU, when the wrapper receives the input, it resets, driving the reset signal down to 0, does a clock cycle, by driving the clock signal to 0 then to 1, then starts the accelerator in two clock cycles, in the first it sets the signal representing the arguments of the kernel, typically by taking the values from the input, then the next cycle it drives the `start_port` signal to 1 for the duration of a single cycle, as represented in Figure 4.7.

After this initialization phase, the wrapper evaluates certain output signals to check for a memory request every cycle. This process, however, differs from interface to interface and is discussed in Section 4.3.5. Once the wrapper reads a memory request, it inserts it into a queue that the gem5 object can then access. A memory request is composed of a physical address, number of bytes, data in case of a write, and the ID of the interface in case of multiple interfaces; if the number of bytes of a request is bigger than the size of the

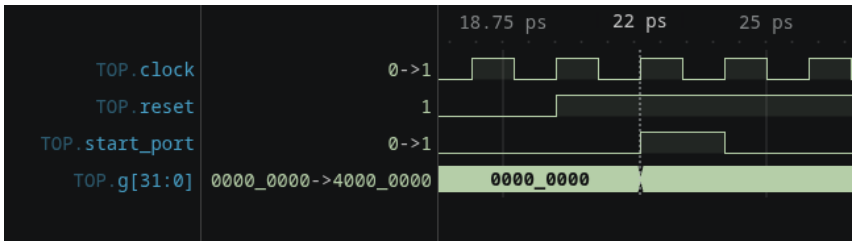

Figure 4.7: Signal traces at the start of the accelerator simulation.

cache line, the request is split into multiple requests. Each tick, the wrapper checks for entries in the response queue, which can be accessed by the object. If the queue contains a response, its data is then processed, and the FPGA signals are driven accordingly. Possible operations to do with a response include the forwarding of the data from a read request, or the driving of a specific signal in case of a successful request. This process, summarized in Figure 4.8, is repeated until the wrapper reads 1 on the `done_port` signal.

4.3.5. Memory interfaces

As mentioned before, accelerators can present various types of memory interfaces, to prove this thesis's flexibility, multiple AXI [11] interfaces have been implemented between the accelerator to the CPU (AXI-LITE with the CPU acting as master) and between the accelerator and the SPM (AXI4, with the accelerator acting as master): a basic slave AXI interface, a pair of master/slave AXI interfaces for controlling the accelerator, and multiple slaves AXI interfaces on a single accelerator, all three are represented in Figure 4.9.

Starting from the simplest of them, implementing a slave AXI interface for an accelerator means splitting memory requests into two stages, with a small third one for writes. The implementation of the protocol itself is all located in the function `eval()`, which mimics the function with the same name generated by Verilator; inside this method, five main checks are executed, one for each stage of the protocol: Read Address, Read Data, Write Address, Write Data, and Write Done.

As the name suggests, two of them are utilized for requesting a read, while the other three are for requesting a write. Specific output signals of the accelerator dictate which phase is currently executing, however to consider a phase "done" and proceed to the next both the accelerator and the memory interface must be ready, this is done through the check of two signals: the simulator's interface checks a `valid` signal from the accelerator, the same that indicates the phase, while the accelerator checks a `ready` signal, which is driven by the simulator. The first phase is either read address or write address, here the parameters

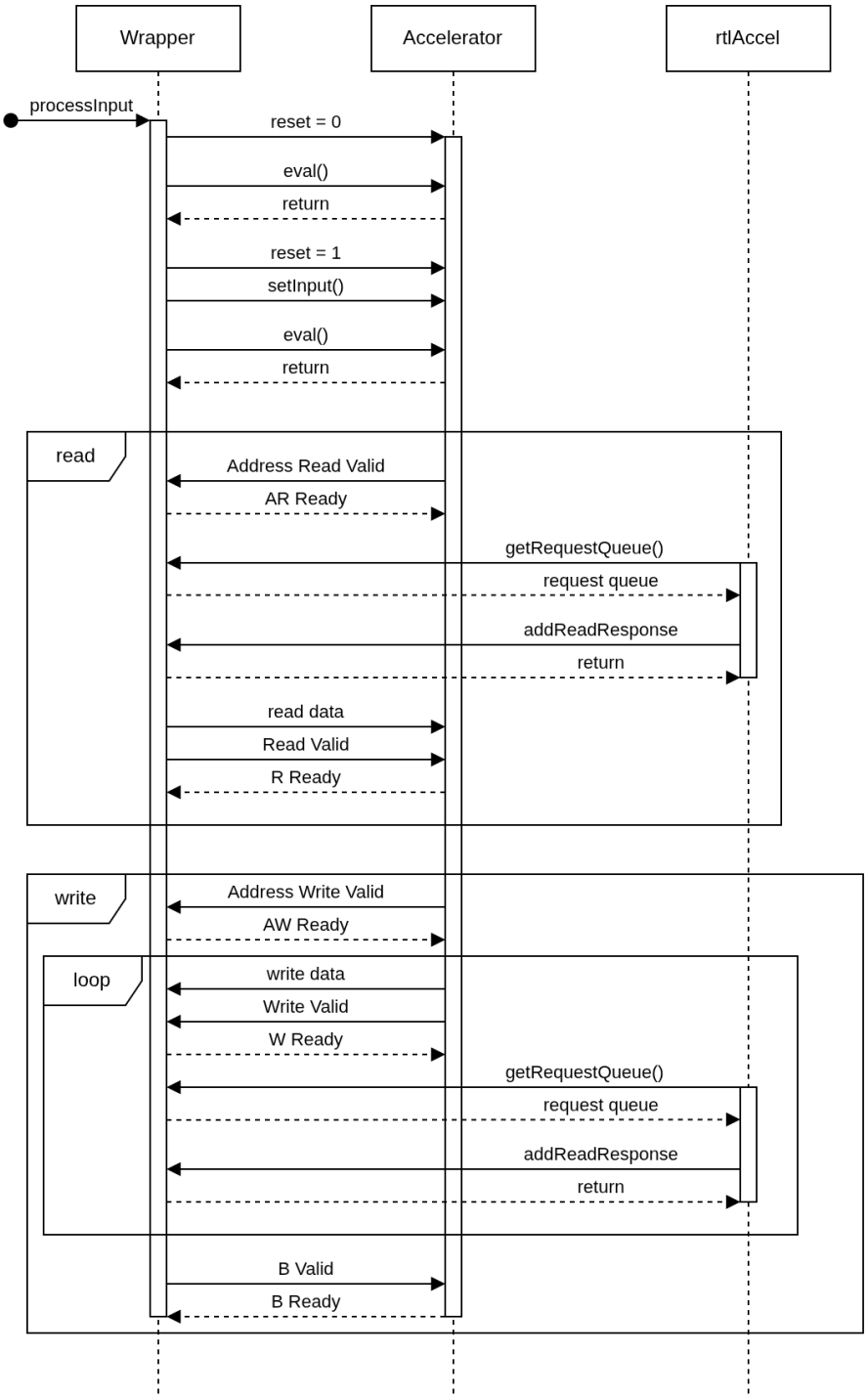

Figure 4.8: Wrapper execution diagram.

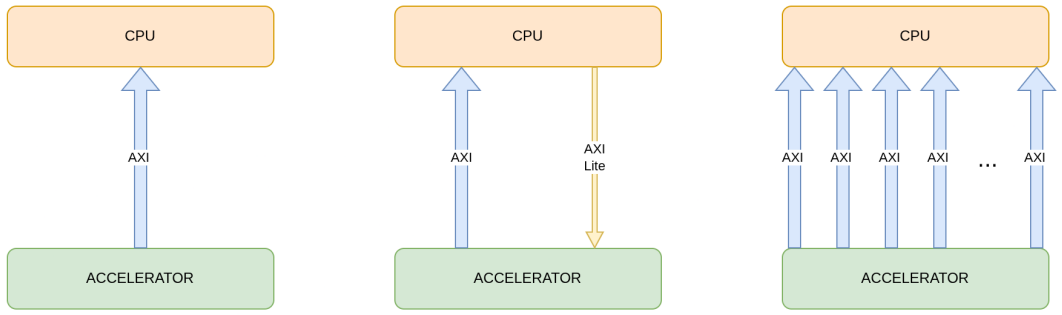

Figure 4.9: The three types of interface implemented, from left to right: single slave, pair master/slave, multiple slaves.

of the request are taken from the output signal of the kernel, the main parameters are size, length, type and address; then the accelerator goes to one of the data phases, which either expects a response with some data, in case of a read, or gives the interface the data to write; in case of a write, a final stage where the confirmation of a successful write occurs.

The parameters dictate to the simulator where and how to read or write from and to the memory: the address indicates the base of where in the memory the request occurs, the size indicates the number of bytes of an element, the length indicates the number of elements, so the total data is computed as $size * length$, and the type indicates the kind of request. There are three types of AXI requests: fixed, each element is read at the same address; incremental, each element is read one after another, meaning that each element is located at the base address plus the bytes already read; wrap, is similar to incremental, but once the address read goes over a threshold, it wraps back and the next element is read at base address. After a request is defined, it's passed to the wrapper, which may split it into multiple requests, then the accelerator object sends it to the memory system, the latter then responds, the object sends the response to the wrapper, which enqueues it, merges the data if needed, then the AXI interface receives the data and drives either the `data` signal, in case of a read, or the `valid` signal in case of a write.

The other two types of interface derive from this first implementation; the pair slave/-master interface is composed of a slave interface identical to the single slave interface, and a master simpler AXI interface. While the slave interface is a Full AXI interface, the master is a simpler AXI Lite interface, meaning that it can only read one element at a time and does not support multiple interfaces; aside from these differences, the processes

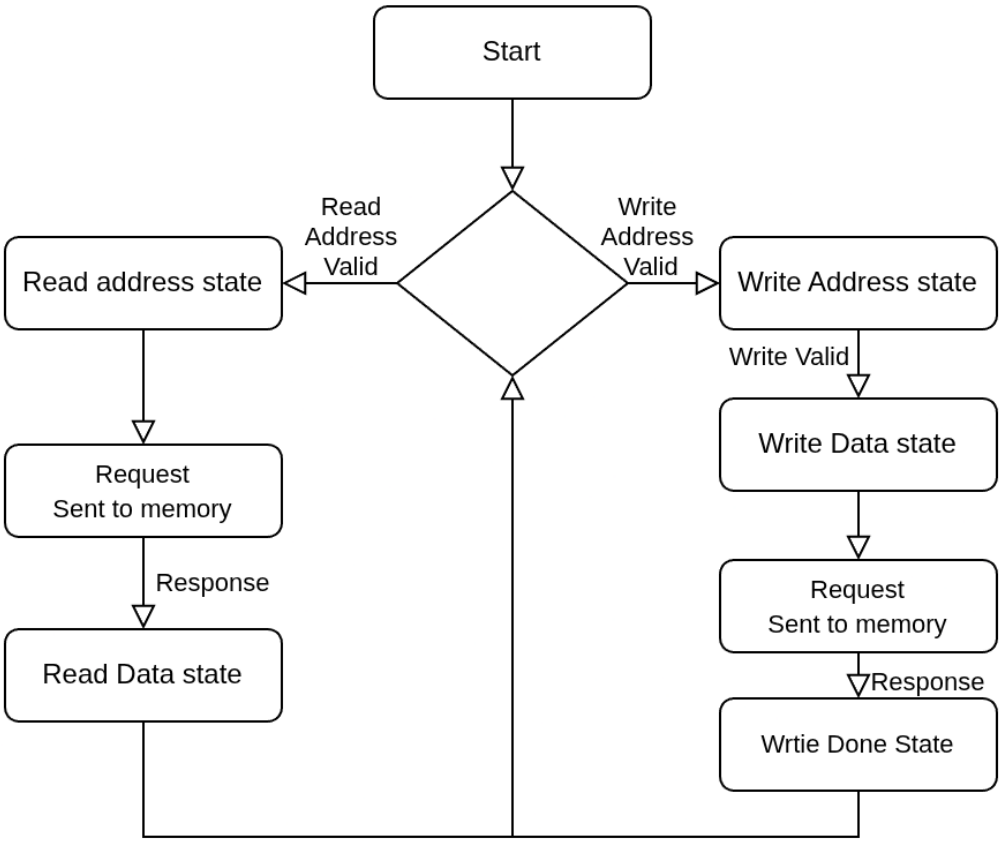

Figure 4.10: States of an AXI transaction.

are identical, in the accelerators implemented that use this pair interface, the slave is used for normal data transfer, while the master is used to control the accelerator, in particular to pass the arguments of the kernel and command the accelerator to start.

The last interface type modeled for this work, the multiple parallel slave AXI interfaces, consists of multiple copies of a Full AXI slave interface. Each interface needs its own evaluation step and its own memory port inside the accelerator object, the first is to assure that each interface works independently, for the same reason the wrapper provides its own request and its own response queues, the second is to assure a correct timing measurement: each request should go to a separate part of the memory, so that two packets from two different interfaces don't interfere with one another; so the SPM is split into banks one for each interface, all with the same parameters, this is done by modifying the system configuration and connecting all the banks to the same XBAR, in a fashion similar to the one represented in Figure 4.5. This last interface type is the one used by the accelerators in the upcoming experiments.

For the implementation of the memory itself, various proposals were considered, for example, implementing or adapting DRAM controllers for DDR and HBM, following [2] [35] [5]; however since the accelerators have multiple pseudochannels/interfaces, all accessing a private memory, the implementation adopted consists of multiple `SimpleMemory` objects, easy to configure with the boards' parameters.

4.3.6. Host Binaries

In this chapter, the implementation and functioning of various parts of the simulator are explored in depth. All the components disclosed share a single property: they all need to be started by the CPU through a syscall. This section will walk through the binaries simulated by gem5 and how they use CPU syscalls to activate various steps of the kernel simulation.

As seen in Section 4.1, the main memory is split in two parts, a "normal" cached region and an uncacheable region, in the host binaries this region needs to be flagged as such, the binaries allocate through the use of `mmap()` a region of a certain size, in the experiments 500MB, starting from a certain base address, in the experiments `0xC0000000`, flagged as `MAP_SHARED`. After an uncacheable region is allocated, it gets split into N banks, where N is the number of interfaces of the accelerator. From an external file `graph.h`, the input of the accelerator is taken and loaded into the newly allocated uncacheable regions. If N is greater than 0, the input has to be split into multiple banks. The accelerators generated by Bambu expect a specific memory layout in case of multiple memory interfaces: when

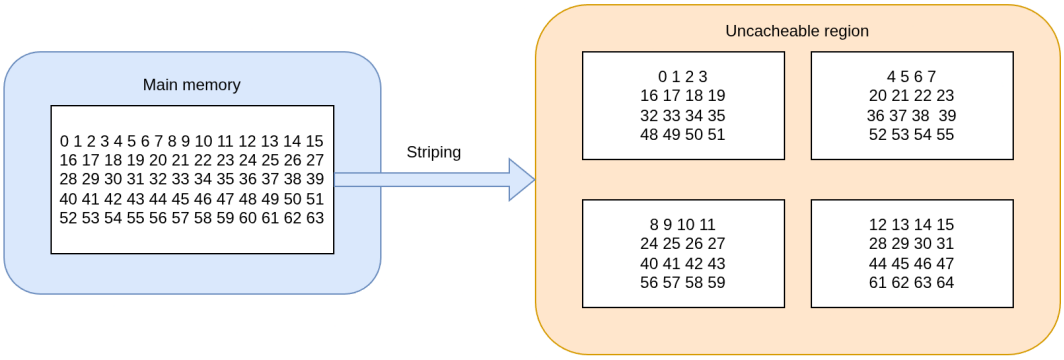

Figure 4.11: Striping of an array of 64 integers into 4, with chunk size equal to 16 Bytes (4 integers).

the data is loaded, it is split into chunks of a given size, then each chunk is loaded into a memory bank, in case of 2 banks/interfaces, the first chunk goes into the first bank, the second goes into the second bank, the third goes into the first bank, the fourth goes into the second, and so on, as represented in Figure 4.11. This process is called striping.

The addresses of the input are then loaded into a struct that will be sent, through a memory request, to the accelerator by the CPU.

The first syscall is then executed, the binaries invoke `m5_load_memory`, which takes as arguments the input base address, typically the uncachable memory base, the destination, and the size, then starts the memLoader utility; this process is repeated for each memory bank of the SPM. To check if the memory loading is terminated, the binaries implement a simple polling method, as represented in Figure 3.3.

Then the accelerator is started, the input struct is passed to the accelerator using the `m5_start_accel` syscall, the accelerator simulation starts, and through the same polling method, the syscall `m5_wait_accel` checks if the kernel execution terminated.

After the end of the kernel execution, in a similar fashion to the memory loading, mem-Retriever is called by the CPU by calling `m5_retrieve_memory`. After the memory is retrieved from all the banks, the memory is put back into a single region through an unstriping algorithm that reverses the striping.

The memory is then read and the results of the accelerator checked, the expected results

are stored in `graph.h` for all the kernels used in the experiments, it is worth noting that while the majority of the accelerators results are stored in memory, the TRC accelerator uses the return of the `m5_wait_accel` function since its result is a simple scalar value, the number of triangles.

4.4. Board modeling

To achieve an accurate simulation of a physical SoC, the computational cores and the memory devices must be accurately modeled. gem5 provides a highly customizable, easy-to-configure set of pre-configured objects that model CPU cores and memory technologies.

4.4.1. CPU cores

For this thesis, gem5's various models of CPU cores, based on the ARM architecture, will be used to represent the wide array of CPUs found in FPGA SoCs:

- `AtomicSimpleCPU`: A functional-only model used only to accelerate simulation during the initial phases of the simulation. By utilizing atomic memory accesses, it significantly accelerates certain parts of the simulation that are of no interest to this thesis, in particular, the boot of the operating system.
- `MinorCPU`: An in-order timing model employed here to represent "LITTLE" or efficiency-oriented cores, such as the Cortex-A53 and Cortex-R5.
- `DerivO3CPU`: An out-of-order execution model. In this study, the `DerivO3CPU` configuration is used to model high-performance "big" cores like the Cortex-A72 and the Cortex-A9.

Each core is modeled and configured in a Python file; each parameter of the various cores is configurable by modifying the corresponding file. To model a new core, instead of modifying an existing object, it's better to create a new class that inherits from the "base" core object and gives new values to the parameters of the parent class.

To prove the flexibility of this thesis, four SoCs were modeled and later tested:

- AMD Zynq™ 7000
- AMD Versal™ HBM Series VHK158
- AUP PYNQ-ZU
- NanoXplore NG-LARGE

4.4.2. Memory system

To accurately simulate a board, a correct core implementation is not enough; the memory hierarchy must also be modeled. In Gem5, modeling complex memory hierarchies can be configured through a Python script file, like the core configuration. In this thesis, various versions of the ready-to-go example `fs_bigLITTLE_RTL.py` [46] are used. This configuration is a modified version of `fs_bigLITTLE.py` that connects the accelerator object to the main system. In the configuration script, it's possible to configure the size and latencies of the main memory, configure the cache hierarchy, and connect other memories to the system. As described in Figure 4.1, the main memory is connected through a cache system to the CPU, but there exists an uncachable region where only the guest binary can write. This is done to accurately simulate memory latencies in these FPGA-accelerated systems, as physical boards typically feature faster and dedicated memory devices for the kernel [10].

To correctly model a SoC's memory system, the correct cache hierarchy must be declared and various parameters, for both the main memory and caches, have to be set, like latencies, dimensions, and bandwidth, as shown in Table 4.1

Table 4.1: Cache configuration parameters.

Parameter	Description
tag_latency	Cycles required to access and check the cache tag array.
data_latency	Cycles required to access the data array on a hit.
response_latency	The additional delay between a hit and the response being sent to the processor.
mshrs	Miss Status Holding Registers: The number of concurrent outstanding misses the cache can handle.
tgts_per_mshr	Targets per MSHR: The number of separate requests for the same block that can be merged into one MSHR.
size	Total storage capacity of the cache.
assoc	Associativity: The number of lines (ways) in each set (3-way set associative).
write_buffer	The number of slots for pending writes.

4.4.3. AMD Zynq™ 7000

Zynq 7000 SoC devices feature a single-core ARMCortex-A9 processor [3], which is a performance core featuring an out-of-order pipeline [12]. To correctly represent this core in gem5, it is possible to start from the `DerivO3CPU` and modify its parameters by creating a new ARM_Cortex_A9 class that inherits from `DerivO3CPU`. The core model parame-

ters, described in table 4.2, were established by Endo et al.[26]. These parameters were extracted from various manuals and from the ARMwebsite [18] [12] [14] [16].

Table 4.2: Main ARMCortex-A9 parameters.

Parameter	Value
BTB entries	4096
Return address stack entries	8
ITLB/DTLB entries	64 each
Issue width	2
gem5 effective execution stage depth (wbDepth)	8
Pipeline stages	8
Physical INT/FP registers	62/256
IQ entries	32
LSQ entries	8 each
ROB entries	40

The main difference between the core simulated in [26] and the `Zynq` is the clock speed, which can go up to 1GHz in our SoC. To simplify the implementation of the FPGA accelerator, it will be assumed that the core always runs at the top speed. This makes it significantly easier to implement the accelerator clock, since the CPU clock is an integer multiplier of the FPGA clock. On the memory side, the `Zynq 7000` inherits from the `Cortex-A9` a two-level cache system, with two L1 caches (one for instructions and one for data). The cache parameters, represented in Table 4.3, are taken from [26].

As for the memory model, the Zynq supports various DDR technologies. For this thesis, the memory will be modeled as a DMA-like `SimpleMemory` with DDR3-2133 parameters:

- Bandwidth: 17.0 GB/s
- Latency: 10 ns
- Latency variance: 3 ns

4.4.4. AMD Versal™ HBM Series

The AMD Versal™ HBM Series VHK158 is a high-performance board designed to keep up with the higher memory needs of compute-intensive or memory-bound applications. It is the most expensive board taken into consideration in this thesis, costing approximately $15,000 [3]. It features a dual-core ARMCortex-A72 processor, an out-of-order core built for performance. For this reason, the implementation is similar to the Cortex-A9 implementation, which starts from the `DerivO3CPU` Python class. The parameters for the correct representation of the cores were taken from the official ARMdeveloper website [13]

Parameter		Cortex-A9
L2	Size	512 KiB
	Associativity	8
	Latency	8
	MSHRs	11
	Write buffers	9
L1-I	Size	32 KiB
	Associativity	4
	Latency	1
	MSHRs	2
L1-D	Size	32 KiB
	Associativity	4
	Latency	1
	MSHRs	4
	Write buffers	16

Table 4.3: Cortex-A9 Cache parameters.

and from various articles [43] [37] and are summarized in Tables 4.4 and 4.5.

Table 4.4: Cortex-A72 main size parameters.

Parameter	Value
Load Queue (LQ) Entries	48
Store Queue (SQ) Entries	48
Issue Queue (IQ) Entries	92
Reorder Buffer (ROB) Entries	192
Dispatch Width	5
Issue Width	8
Writeback (WB) Width	8

The Cortex-A72 has a 2-level cache hierarchy, very similar to that of the `Cortex-A9`. The cache parameters are summarized in Table 4.6 and are taken from the official product page [3] and from the report "ARM's Cortex A72: aarch64 for the Masses" by Lam C. [43].

As the name suggests, the AMD Versal™ HBM Series uses a Gen2 HBM memory [39] [40] [38] [4] for its operations made with 2 stacks of 16GiB of memory (32 GiB in total) with a reported maximum bandwidth of 820 GB/s [3] [6]. However, this bandwidth is achieved only through the full use of both stacks and all the memory's pseudo-channels

Table 4.5: Cortex-A72 main delays.

Path	Delay (Cycles)
Decode → Fetch	1
Rename → Fetch	1
IEW → Fetch	1
Commit → Fetch	1
Rename → Decode	1
IEW → Decode	1
Commit → Decode	1
IEW → Rename	1
Commit → Rename	1
Commit → IEW	1

at the same time, which are 32; for this reason, in this thesis, it will be assumed that each pseudo-channel has a latency of 1-2 ns and a bandwidth of 27 GB/s.

Parameter		Cortex-A72
L2	Size	1 MiB
	Associativity	16
	Latency	22
	MSHRs	20
	Write buffers	16
L1-I	Size	48 KiB
	Associativity	3
	Latency	2
	MSHRs	2
L1-D	Size	32 KiB
	Associativity	2
	Latency	4
	MSHRs	6
	Write buffers	8

Table 4.6: Cortex-A72 Cache parameters.

4.4.5. AUP PYNQ-ZU

The AUP PYNQ-ZU is an open-source project from AMD that features a 64-bit Quadcore ARMCortex-A53 [55], an in-order pipeline processor, defined as a `"small"` or efficiency core. It's a less powerful core than the Cortex-A72 or the Cortex-A9, but it is more energy efficient. gem5 actually already has a core model that represents the Cortex-A53

accurately, called `MinorCPU`. Similar to the other cores, the characteristics to model the core were extracted from the ARMdeveloper website [15]. Therefore, the only parameters modeled were the cache hierarchy and the memory system. The cache characteristics, extracted from [42] and [15], are reported in Table 4.7.

Parameter		Cortex-A53
L2	Size	1MiB
	Associativity	16
	Latency	12
	MSHRs	16
	Write buffers	8
L1-I	Size	32 KiB
	Associativity	2
	Latency	2
	MSHRs	2
L1-D	Size	32 KiB
	Associativity	4
	Latency	3
	MSHRs	4
	Write buffers	4

Table 4.7: Cortex-A53 Cache parameters.

Memory side, the PYNQ supports DDR4-2400; therefore, the memory model has a bandwidth of 19 GB/s and a latency of 12ns.

4.4.6. NanoXplore NG-LARGE

The NanoXplore NG-LARGE is the second generation of Radiation Hardened By Design (RHBD) SRAM-based FPGA developed by NanoXplore [24]. It mounts an ARM Cortex-R5, a single-core, efficient, 32-bit CPU with an in-order pipeline. Being an in-order core, the Cortex-R5 is similar to the Cortex-A53; in fact, the core model is an extension of the `MinorCPU` Python class, and being 32-bit is not an issue, since gem5's models are suitable for both 32- and 64-bit ISAs. The core parameters, represented in Tables 4.9 and 4.8, were extracted from the ARMdeveloper website [17] and from the nanoxplore wiki [24]. Notably, the R5's cache hierarchy doesn't have an L2 cache. gem5's parameters enable the representation of any `last_cache_level` greater than 0, but the configuration `fs_bigLITTLE_RTL.py` doesn't connect the L1 caches to the accelerator, so the configuration was modified to connect the accelerator to the L1 caches if the `last_cache_level`

is equal to 1. With the connection fixed, the parametrization of the cache was identical to the previous boards; its parameters are summarized in Table 4.10. The NG-Large supports both DDR2 and DDR3 memory interfaces, so the memory model will be a `Simple Memory` with the bandwidth and latencies of a DDR3 memory, like in the Zynq board.

Table 4.8: Cortex-R5 main delay parameters.

Parameter	Value
fetch1ToFetch2ForwardDelay	1
fetch2ToDecodeForwardDelay	1
decodeToExecuteForwardDelay	2
Integer ALU latency	2
Integer Multiplier latency	2
Floating Point / NEON latency	2
Load/Store Unit (LSU) latency	2

Table 4.9: Cortex-R5 Functional Units.

Parameter	Operations	Count	Issue width
Integer ALU	IntAlu, IntAluOp	1	1
Integer Multiplier	IntMult, IntDiv	4	1
Floating Point / NEON	FloatAdd, FloatMult, FloatDiv	4	1
Load/Store Unit (LSU)	MemRead, MemWrite	2	1

Parameter		Cortex-R5
L1-I	Size	32 KiB
	Associativity	4
	Latency	2
	MSHRs	8
L1-D	Size	32 KiB
	Associativity	4
	Latency	2
	MSHRs	8

Table 4.10: Cortex-R5 Cache parameters.

5 | Experiments

This chapter presents the experiments conducted to evaluate the effectiveness of the proposed framework as a flexible tool that can precisely evaluate the performance of hardware designs while improving the design space exploration phase of development. The experiments, following the methodology presented in Chapter 3, aim to evaluate the performance of different hardware and different FPGA designs, following the process that a researcher could follow when developing and integrating algorithmic RTL implementations on specific hardware platforms. This chapter will walk through various batches of experiments and their results. First, by presenting the simulation setup for each batch, following with the experiments themselves, and then the presentation and discussion of the results obtained.

To demonstrate how this thesis aids the development of a hardware accelerator, multiple experiments have been conducted across multiple devices and simulating various kernel designs. The experiments aim to demonstrate the effectiveness of this thesis during the design exploration phase of development by measuring the performance of various RTL implementations of the same behavioral algorithm. As mentioned in Section 5.1.1, multiple versions of four algorithms, all with different parameters, have been synthesized by Bambu, the HDL code compiled into a C++ library by Verilator, and then simulated using this work on four different emulated hardware devices. Two batches of experiments have been performed: the first consisted of simulating all the different kernel implementations on a single board, to test how the number of cores and memory interfaces, influences the overall performance of the simulated accelerator, and the difference between the measurements of this thesis and the measurements of Bambu's cosimulation; the second batch simulated a single RTL implementation of a single algorithm, but on four different boards, which highlights how different hardware influences the performance of the complete system.

5.1. Benchmarks

This section is about understanding the benchmarks used in this work to test the simulator functionality and its accuracy. As previously mentioned, gem5+RTL is a Full-System simulator; this means that the benchmarks comprise a full OS simulation, the execution of a host binary, and the execution of the FPGA kernel. To simulate both 32- and 64-bit ARM ISAs, two operating systems are needed, both are headless versions of an Ubuntu image, both available at [33]. These images contain only the essential pieces of software to perform a complete boot of the operating system, execute a script that creates a checkpoint, to reduce simulation times, and then start the real simulation by running the host binaries.

5.1.1. Evaluated Benchmarks

The kernels chosen for the benchmark of this thesis are taken from the work of Gozzi et al.[34]. In this paper, they extend the Bambu HLS tool to integrate support for functions written in OpenMP into Bambu. This extended version of the Bambu HLS flow has been used in this work to implement four FPGA designs based on four highly parallel algorithms for graph evaluation/analysis taken from [19]. All benchmarks take a graph as input, which is structured into four data sets. Two sets represent the incoming and outgoing edges, while the other two contain the corresponding vertices connected to those specific edges. The algorithms used in this thesis were chosen for their computational diversity and their wide use in real-world scenarios. Their purpose is to evaluate graphs under various metrics. The algorithms chosen are:

- Connected Components (CC) labels all vertices by their connected component, and each connected component is assigned its own label.
- Triangle Counting (TRC) computes the total number of triangles in a graph. A triangle is defined to be three vertices directly connected; the set of vertices is unordered, meaning that the same three vertices are counted as only one triangle, no matter the order in which they are listed.
- PageRank Pull (PRP) computes the importance of each node in a graph based on the importance of the nodes connected to it. The version considered in this thesis is based on the work of Zhou et al.[60].
- Breadth-First Search (BFS) is a graph traversal algorithm, starting from a source vertex, BFS traverses all vertices at the current depth (distance from the source vertex) before moving onto the next depth. This thesis utilizes the implementation

Figure 5.1: Frequencies synthesized for the kernels used in the benchmarks.

of the algorithm developed by Betkaoui et al.[20].

For each kernel, Bambu automatically generated and synthesized 24 different versions. As described in Section 2.4.3, these versions differ in the number of threads or context switches, parameters derived from SPARTA to hide memory latencies. Context switches range from 1 to 8, while cores range from 1 to 32. Bigger and more complex accelerators run at slower frequencies than smaller and simpler ones. The maximum operative frequencies are obtained during the Place and Route (P&R) stage of synthesis and are reported in Figure 5.1. Configuration names are of the form Xco_Ycs, where X and Y correspond to the number of SPARTA cores or context switches of the synthesized kernel. The number of cores stands out as the main contributor to the limit of synthesized frequencies, except for the PRP accelerator, which always runs at the slowest frequency.

5.2. Experimental setup

The experiments were all conducted on a server running on an Intel® Xeon® Processor E5-2698; the processor has 16 cores, 32 threads, and its frequency was locked at 2.2 GHz. The server's OS was GNU/Linux Ubuntu 24.04.4 Long-term support (LTS) with Linux

kernel version 6.8.0, but the simulations were performed inside a Docker container, running the official Ubuntu 20.04 Docker image. Simulating Ubuntu 20.04 is essential to ensure the availability of the correct version of gem5+RTL's dependencies, since gem5+RTL is based on version 21.1.0.2 of the gem5 simulator. The versions of Verilator tested and functioning for gem5+RTL are versions v3.9 and v4. For the experiments, the version used is v4.220. To test how much the simulation times are impacted by the hardware on which it is run, a small batch of tests was conducted on a different machine running Cachy OS, also with Linux kernel 6.8.0, on an Intel® i7-12700K, 12 cores, 20 threads, and a maximum frequency of 5.0GHz. While the compilation times were similar, the simulations were significantly faster on the i7, because Gem5 is single-threaded, meaning that the extra cores of the server weren't utilized, and the higher speed of the i7 accelerated the simulation. The average frequency during simulation was around 4.7 GHz, and it reflects directly with the execution times, since a speed-up of up to 2× times, compared to the Xeon, with simulation times as low as 35 minutes, while the same benchmark took more than an hour on the server.

The experiments involving the kernels that implement CC, TRC, and PRP share the same benchmark input data, a randomly generated directed graph described by four arrays of integers, two for the vertices of the graph and another two for the edges connecting the vertices; each pair of arrays represents the inward and outward edges of a vertex. The number of vertices is 8192, while the number of edges is 32760. The input of the benchmarks is saved in a C header file to be included in the host binaries. The same file also contains the expected results of the benchmarks, two arrays for the CC and PRP algorithms, and a single scalar for TRC. The benchmark for the BFS implementation shares the same format, but the data is different; the number of vertices is 65536, the number of edges is 524288, and the expected output is an array of 7 integers.

5.3. Preliminary operations

A Verilog implementation of the four algorithms to simulate, CC, TRC, PRP, and BFS, has been generated by SPARTA-Bambu, following the flow described in Section 2.4.1. The HLS tool was used to generate 96 RTL implementations, 24 for each algorithm; all the implementations differ in the number of cores (CO) and context switches (CS). The number of cores ranges from 1 to 32, in powers of 2, so the accelerators can have 1, 2, 4, 8, 16, or 32 cores; the accelerators can have 1, 2, 4, or 8 context switches. The accelerators present each permutation of the pair CO-CS.

Each implementation was simulated and tested for correctness by Bambu through Verila-

tor; the simulation results will be compared to the simulation results of this thesis. Bambu also generated a report containing simulation, logic synthesis, and implementation results, containing the number of cycles simulated, the number of nanoseconds simulated, and, most importantly, the frequency synthesized. As mentioned in Section 5.1.1, the operational frequencies are essential for a correct simulation, since hardware constraints can limit the maximum actual frequency at which a FPGA design can run.

5.3.1. Scripting and automation

gem5's Python configuration makes creating scripts to run batches of simulations easy automatically: changing the simulation target consists only of replacing a file and modifying a couple of parameters in a file. To run all the experiments, a shell script has been developed. The script takes as arguments an algorithm and a directory, passed as arguments, and searches for a directory named with a configuration name within that directory. Once a directory is found, it searches for the name of an algorithm, CC, TRC, PRP, or BFS, and navigates inside the respective directory. Inside this directory, the HDL implementation of the kernel with parameters Xco_Ycs can be found. The files are moved inside the gem5 directory to be compiled by Verilator. The script then compiles gem5, sets the operational frequency of the accelerator by modifying a parameter inside the Python configuration script, and then starts the simulation itself.

At the end, each simulation produces a `stats.txt` file, containing all the statistics of the simulation; this file is copied and renamed in a separate directory, and the process starts again. This script is executed once for each algorithm, and iterates all 24 different implementations of the given kernel. It is also possible to iterate each execution for each board modeled in Section 4.4, but for this thesis, it hasn't been done; as discussed later, the simulations of the various boards will focus on a single RTL implementation of a single algorithm.

5.4. Evaluation of Data Processing Overheads

When developing a digital design, it is important, during the design exploration phase of development, to test various versions of a certain hardware implementation, to evaluate how the parallelization parameters of the design impact performance, to fine-tune the parameters of the design, or to decide which hardware to deploy the design on. The latency of the accelerator execution represents only a fraction of the total latency of the simulation. To provide a complete view and comparisons of all the latencies of each phase of the simulation, a batch of experiments has been conducted, comparing how much the

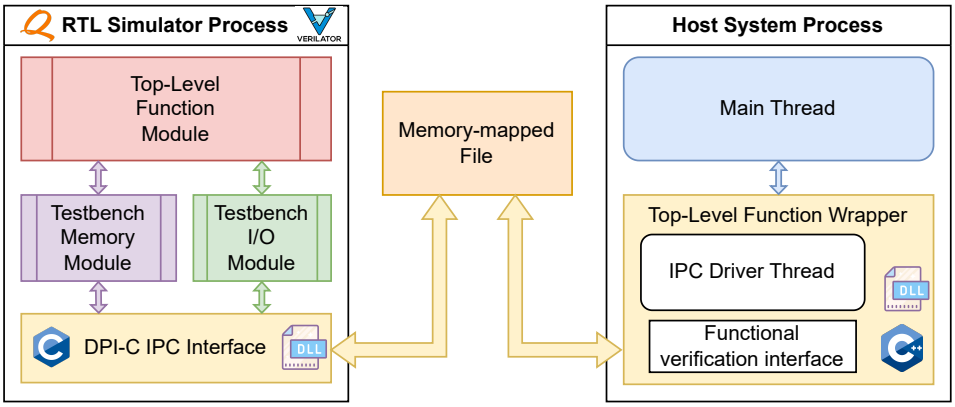

Figure 5.2: Overview of the proposed system-level HW/SW co-simulation framework [29].

various phases of the simulation impact the complete latency.

The results will also be compared to the latency measured by Bambu's co-simulation tool, which simulates the design by simultaneously running a host binary and the simulated hardware together, as presented in Figure 5.2.

The complete execution of an FPGA kernel, presented in Chapter 3 and summarized in Figures 3.3, 5.3, can be split into 5 phases: initialization of the data, memory loading, accelerator computations, memory retrieval, and result validation.

5.4.1. Designs Tested and Target Hardware

This batch of experiments was performed with four different RTL implementations of the four algorithms, all with the same configuration: 32 cores and 8 context switches. The 4 implementations were all evaluated on the same simulated board, which mounts HBM2 memory, an ARM Cortex-A72, and an AMD Alveo™ U55C [1]. The board presents the same components of the AMD Versal VHK158, but has a different FPGA. The decision to synthesize the RTL kernels targeting the AMD Alveo architecture, rather than the Versal, was necessitated by Bambu's lack of support for the Versal family; the Alveo provides a good approximation for the simulation.

The board's principal memory parameters are reported in Table 5.1; the most relevant are the memory latency and bandwidth. Because Bambu simulates only the RTL, it estimates a static memory latency of 20 clock cycles. This thesis implements a minimum latency of 10-12 ns, but since it simulates the entire system, this latency is not fixed; it is affected by what happens in the rest of the system

Parameter	Value
Latency	10-12 ns
Total Bandwidth	820 GB/s
Bandwidth per pseudo-channel	27 GB/s
Number of pseudochannels used	16

Table 5.1: AMD Versal™ HBM memory parameters.

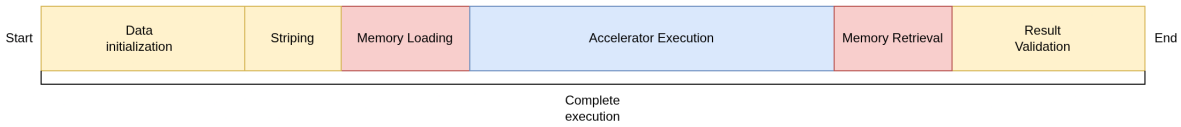

Figure 5.3: Latency partition of the execution of the host binary

5.4.2. Experiment results

The results of this experimental batch, detailed in Table 5.2, compare the total simulated kernel execution latency with the latencies of its individual phases. The table presents the total latency, the accelerator latency simulated by Bambu, the accelerator latency measured by this work, and the latencies tied to the memory operations to load the data onto the SPM. The reported latencies have been measured for all the 4 algorithms implementations. Since the CC, PRP, and TRC kernels share the same data, the memory operations have the same latency. The latencies of TRC for memory retrieving and data un-striping measured are 0, because the TRC kernel returns a scalar, so the retrieval of the memory is not necessary.

The results highlight how software overheads can decrease the kernel's performance; in some cases, the operations not related to the accelerator execution can represent up to 81.6% of the total latency. In these overheads, the handling of the data takes the spotlight: as described in Section 3.3.3 and Figure 4.11, the Bambu-generated kernels expect the data to be in a specific layout, so before loading, the input has to first be striped, this operation has to be executed by the CPU before transferring data to the SPM, meaning it suffers from high memory latencies. After the execution of the kernel, if the output is saved onto the memory, this is the case for CC and PRP; the memory also has to be un-striped to iterate the output and check the correctness of the computation. Although memory load and retrieval operations account for only 1-2% of the total latency in these experiments, with larger datasets, the latency of these operations increases, becoming non-negligible when measuring the performance of an accelerator design.

Table 5.2: Latency comparison across multiple kernels

Algorithm	Total Latency (s)	Bambu Acc. Latency(s)	gem5+RTL Acc.		Mem. Loading		Mem. Retrieving		Data Striping		Data Un-striping	
			Latency(s)	%	Latency(s)	%	Latency(s)	%	Latency(s)	%	Latency(s)	%
BFS	0.09871	0.01170	0.01815	18.4	0.00155	1.5	0.00172	1.7	0.00600	6.0	0.02559	26.0
CC	0.04158	0.00208	0.01279	30.7	0.00054	1.3	0.00059	1.4	0.00251	6.0	0.01052	25.2
PRP	0.04234	0.00087	0.01859	43.9	0.00054	1.3	0.00059	1.4	0.00251	5.9	0.01052	25.0
TRC	0.03030	0.00260	0.00736	24.3	0.00054	1.8	0.00000	0.0	0.00251	8.2	0.00000	0.0

5.5. Comparing RTL and Full-System simulator cycle counts

In this batch of experiments, 24 different RTL implementations of each graph algorithm presented in Section 5.1.1 have been simulated. These implementations, 96 in total, have not only been simulated using this work for a Full-System simulation, but also co-simulated by Bambu; comparing the two simulation results will give an insight into how the various graph algorithm implementations behave differently, with respect to their number of threads, number of contexts, and the hardware they are simulated on.

5.5.1. Designs Tested and Target Hardware

As mentioned before, each implementation differs in the number of cores and the number of contexts implemented in the RTL design. The 96 implementations were all evaluated on the same simulated board of the experiments presented in Section 5.4.

5.5.2. Experiment results

The results of this batch are presented by Figures 5.4, 5.5, 5.6, and 5.7. The graphs that show the measurements were split into each algorithm implementation and split between the implementation's number of cores, dividing the result into 8 graphs, 2 for each algorithm, of which the left represents the data of the implementations with fewer than 8 cores, while the graphs on the right represent the implementations with equal or more than 8 cores. On the Y axis is reported the number of cycles of the accelerator execution, while the X axis reports the number of cores of the simulated design. As depicted by the legend, the color of the lines and dots represents the number of context switches implemented in the RTL design. The dotted lines represent the trend of the number cycles measured by the RTL simulation executed by Bambu, while the solid lines represent the data collected by this work. In this batch, only the kernel execution's number of cycles is taken into consideration; however, this is only a part of the full execution time, and the rest will be the object of the following experiments.

In the CC implementations, configuration 1co_1cs, the Bambu simulation is 2.08× times slower than the execution simulated in this thesis; the same trend is present for the other configurations with a low number of cores. The situation changes when the implementation presents a large number of cores. With respect to the Bambu measurements, this thesis execution times slowed significantly. In the configuration 32co_1cs, this work measured the number of cycles executed during simulation, which is now 1.38× times larger than the RTL-only simulated ones; this trend becomes even more apparent when the number of context switches is increased, reaching a slowdown of up to 5.3× times. These results highlight how, in less parallel implementations, memory latency is the primary cause of execution slowdowns. Conversely, highly parallel designs effectively hide this latency through a high volume of threads and context switches.

It is also worth noting that these results also present a direct correlation between the speed of the accelerator execution and the number of cores and context switches of its implementation. In all the simulations, a higher number of cores meant fewer cycles executed, and the same happened with the increase of the number of context switches, with some exceptions, in cases with 16 and 32 cores. This highlights the need for simulation tools during design space exploration, as increasing parallelization to the maximum does not mean getting the fastest execution times possible. To get the best possible result, the fine-tuning of each parameter and testing how each parameter influences or is influenced by the others, is key to a correct design exploration and to the optimization of a hardware design.

When measuring only the cycle counts, the difference between the less parallelised and the more parallelised implementations is much more pronounced than the differences measured with the total latencies. This is because the system runs at a different frequency than the rest of the system, meaning that while waiting for a memory request to finish, the accelerator executes a lot of cycles where nothing happens, but since memory latency is not tied to a frequency, but expressed in nanoseconds, it means that the higher the frequency, the higher the number of "empty" cycles.

Therefore, relying only on the total latency of the simulation hides the true architectural efficiency of the hardware design. An accurate and correct design exploration needs both measurements.

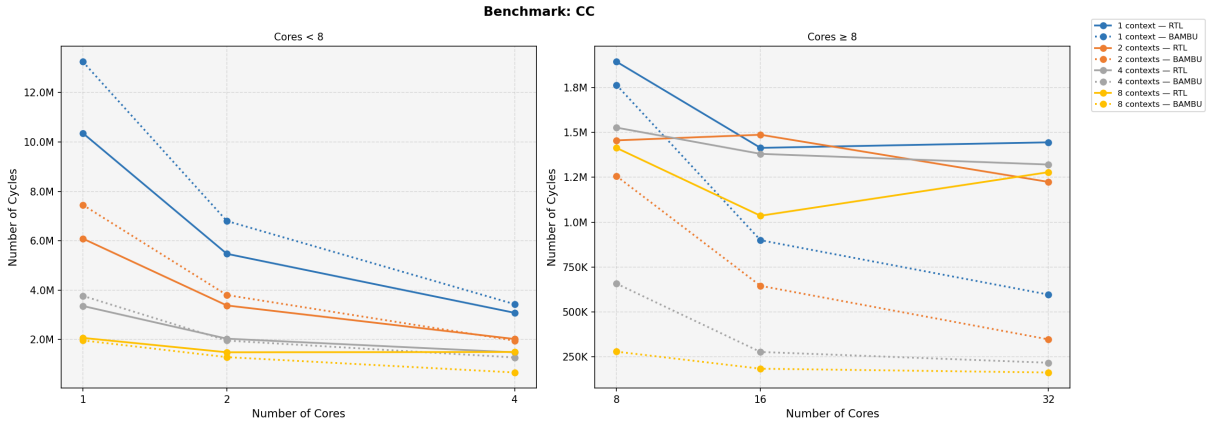

Figure 5.4: Number of cycles CC.

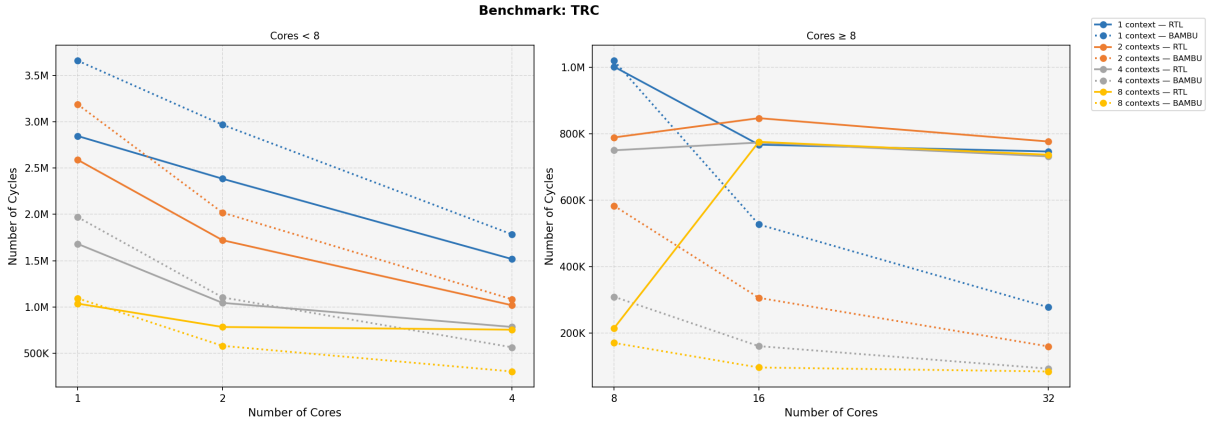

Figure 5.5: Number of cycles TRC.

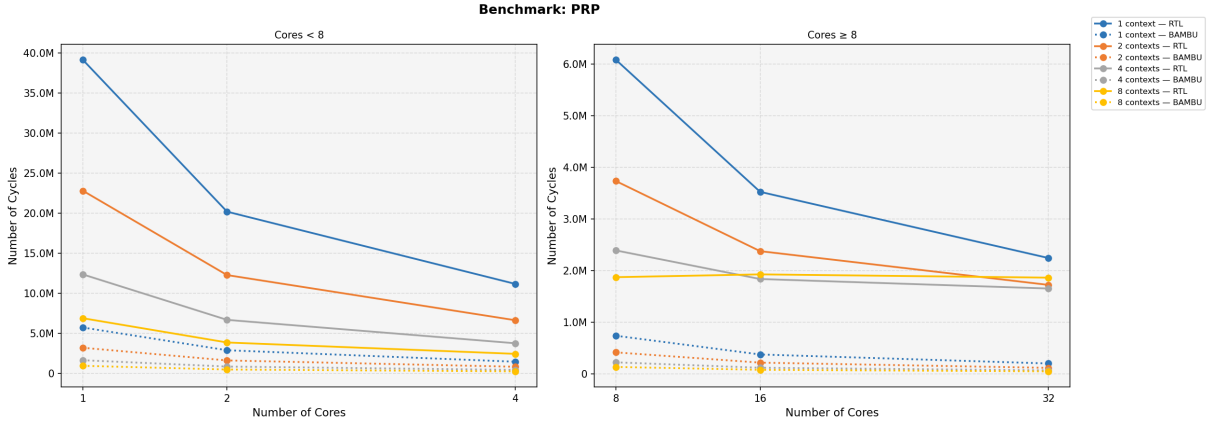

Figure 5.6: Number of cycles PRP.

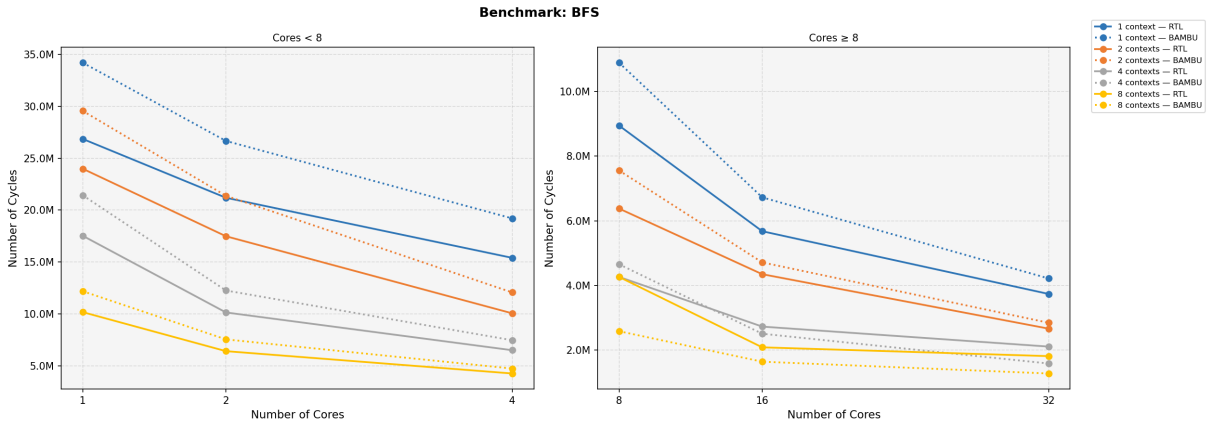

Figure 5.7: Number of cycles BFS.

5.6. Comparing RTL and Full-System simulator latencies

Having insights into the number of cycles that took a computation is important during the performance evaluation of a digital design; however, it does not suffice to have a precise and full view of the performance of the implementation. As mentioned before, the frequency at which the hardware operates is an essential piece of information that can change the outcome of an evaluation. Measuring the number of cycles does not take into consideration the operational frequencies of the accelerated designs; the simulated times can be very different for two simulations with the same number of cycles executed. Using the following formula:

$$Execution\ time = \frac{Number\ of\ cycles}{Frequency} \tag{5.1}$$

It becomes obvious how an accelerator running at double the frequency will take half the time to complete execution, if the number of executed cycles remains the same. This last example was an exaggeration, but as depicted in the next experiment's results, integrating the frequency can lead to different results.

This batch of experiments follows the same structure and processes as the previous one, 96 total simulations, 4 algorithms, and 24 configurations. The parameters taken into consideration remain the same, as well as their range of values that have been tested. The results of this batch will also be compared to the ones obtained by Bambu; the latter presents the data generated by the logic synthesis tools, which not only include the number of cycles, but also total latency and, most importantly, the operational frequency

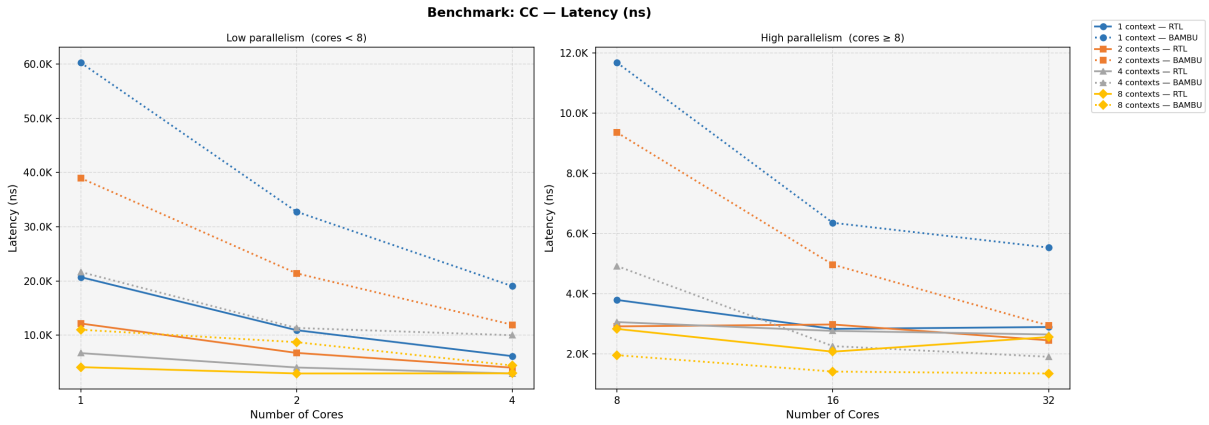

Figure 5.8: Execution latency CC.

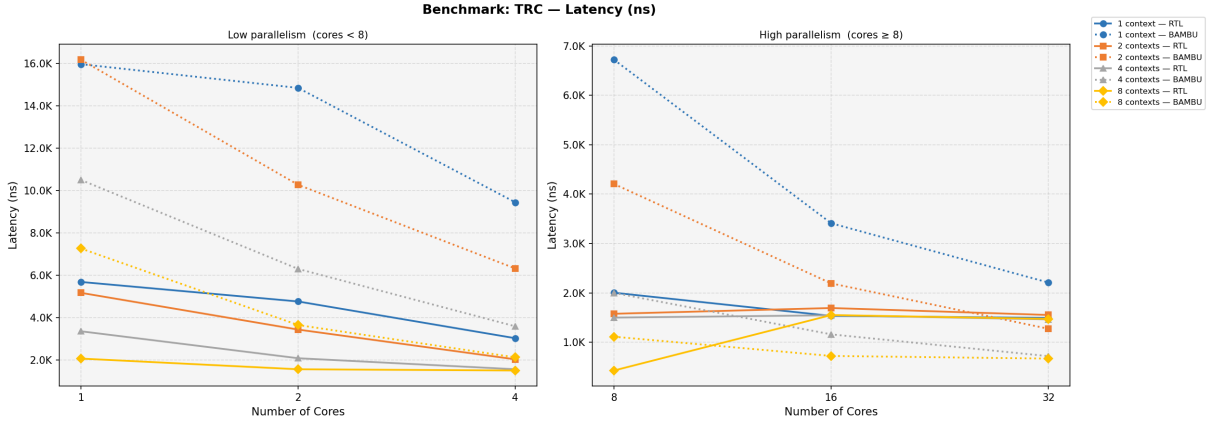

Figure 5.9: Execution latency TRC.

of the physical hardware. This last result is very important, since an FPGA can't run a kernel at any frequency, but it is limited by its resources. The synthesis step of the Bambu flow computes, after hardware allocation, also the maximum possible frequency at which a given RTL design can operate.

5.6.1. Designs Tested and Target Hardware

The setup is identical to the one used for the first experiment batch, with the same memory parameters described in Table 5.1. It is important to acknowledge that in both this and the batch before, each hardware implementation was associated with a frequency close to the one synthesized by Bambu, the same depicted in Section 5.1.1 in Figure 5.1.

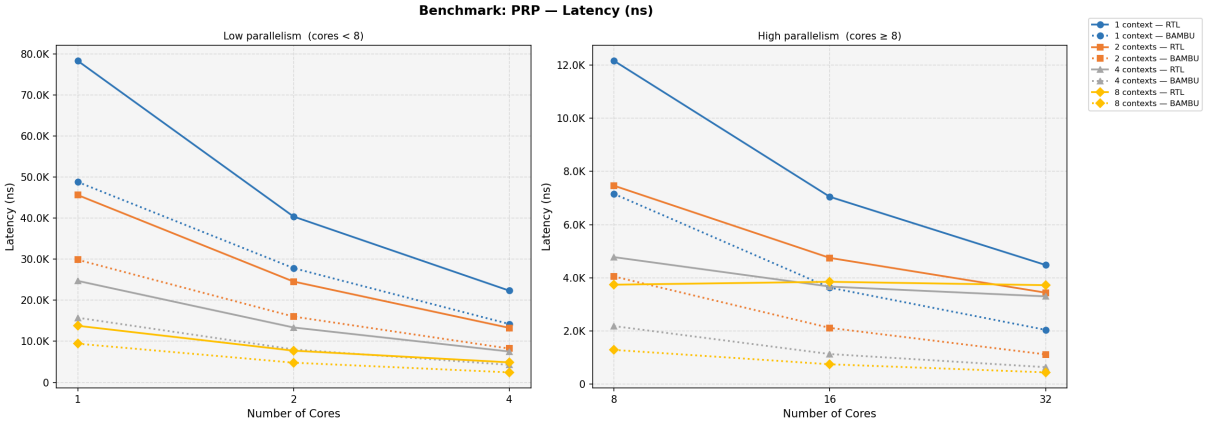

Figure 5.10: Execution latency PRP.

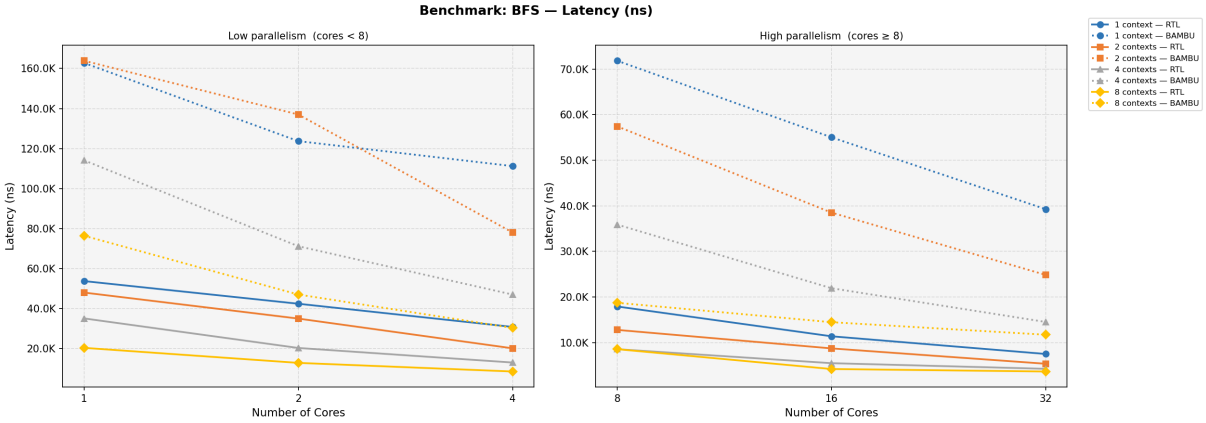

Figure 5.11: Execution latency BFS.

5.6.2. Experiment results

The results presented in Figures 5.8, 5.9, 5.10, and 5.11 are subdivided into 8 plots, the same as the previous batch, split by algorithm and number of cores, to improve the visualization of the data gathered. The total latency measured by Bambu was extracted from the same generated file, where the number of cycles was extracted together with the operational frequency. While this work's measurements derive from the simple computation described in Equation 5.1, by extracting from `stats.txt` the number of cycles and dividing by the frequency, the same process could have been done for Bambu's data, but the tool does it automatically. The frequencies used for this computation are the rounded ones, used during the simulation.

While in the previous batch, the RTL simulated execution eventually becomes faster than the FS execution, as the number of cores and context switches increases. In this batch, this effect is greatly reduced and never happens in the implementations of BFS and TRC, presented respectively in Figures 5.11 and 5.9. However, in all the graphs, it is possible to see how increasing the parallelism means results more similar between Bambu and this work, following the same trend as the previous batch, even if this work's simulated latency remains lower than Bambu's RTL-only cosimulation.

In the previous batch of experiments, PRP's data, represented in Figures 5.10, and 5.6, was an outlier between the other implementations. In this batch, Bambu's simulation still results in faster times than this work's simulations, but the gap between the measurements is much smaller; the difference in PRP's results between the two batches is the largest observed across all experiments. It is worth noting that PRP's synthesized frequencies are the lowest out of all the implementations; in fact, all the configurations have been simulated using the lowest frequency, 100MHz; for reference, some of the other implementations can run up to 200MHz, as represented in Figure 5.1 present in Chapter 3.

5.7. Evaluation of memory latencies across multiple SoCs

The previous batch of experiments aimed to validate how this thesis helps during design space exploration of a hardware design by presenting how various parameters and choices of the design influence performance. In this batch, the experiments will focus on only a singular design, but change the hardware simulated. This is to demonstrate how this work also aids in the choice of the actual hardware to procure to effectively run the hardware

design.

This batch is smaller than the previous one, only 4 simulations have been measured, one for each board modeled in Section 4.4: an AMD Zynq™ 7000, an AMD Versal™ HBM Series, an AUP PYNQ-ZU, and a NanoXplore NG-LARGE. The RTL design simulated is an implementation of the BFS algorithm, with 32 cores and 8 context switches. But for each board, a different implementation has been generated using Bambu, because each board has a different FPGA, with different functional units to allocate. Moreover, each board mounts a different memory, so having 16 memory interfaces in each implementation would not be realistic. So each implementation has a different number of interfaces, runs at a different frequency, and most importantly, is evaluated on different simulated hardware.

5.7.1. Designs Tested and Target Hardware

The four RTL designs were generated by Bambu, and as for the previous experiments, their relative operation frequencies have been measured. The four implementations share the same cores and the same context switches, to be precise, 32 cores and 8 context switches. Each board has its own CPU model and memory; the details of the implementation and parametrization of these boards' models are presented in Section 4.4. Despite having the same parallelisation parameters, each implementation is different, as mentioned before. To have a realistic simulation, the four kernels have a different number of memory channels: the Versal, with the most powerful memory, has 16 interfaces, the NG-LARGE has 8, while the Zynq and PYNQ-ZU both have 4 interfaces. These numbers, together with the operating frequencies associated with the implementations, are summarized in Table 5.3.

All the boards' parameters and components are configured inside gem5's configuration scripts, so when measuring, to change a board, it is possible to simulate another by simply changing the arguments of the command used to start gem5. In this particular context, however, since mostly all the accelerators have a different interface, it is still needed to recompile gem5 to change the accelerator object; this was skipped when simulating the Zynq and PYNQ boards, since they share the same number of interfaces.

board	number of interfaces	operational frequency
AMD Zynq™ 7000	4	150MHz
AMD Versal™ HBM Series	16	200MHz
AUP PYNQ-ZU	4	150MHz
NanoXplore NG-LARGE	8	50MHz

Table 5.3: RTL implementation parameters for BFS with 32 cores and 8 context switches for each board model.

5.7.2. Experiment results

The results, presented in Figure 5.12, show various latencies measured during the 4 simulations. For each board, the measurements taken into account are: the full execution time of the system after the OS boot was completed, full execution latency of the accelerator, time spent waiting for a response from memory, split into read and write, and the average latency of each memory request, again, split into read and write. For representation clarity, the graph y-axis is in logarithmic scale, and presents both seconds and nanoseconds values, because the average latencies are so small that they had to be represented in nanoseconds: in this benchmark, the accelerator performs approximately 1 million read requests, so the average latency of the requests is 1 millionth of the total read latency.

The best performing hardware in terms of memory latency is the AMD Versal: the total accelerator latency is 2.26× shorter than the second best accelerator latency. Both measurements share the same trends, so Versal's memory performs faster in both. But the board with the lowest total latency is not the Versal, but the Zynq 7000.

The Zynq 7000's memory performance is better than that of the PYNQ and NG-Large, both in total memory latency and average memory latency; however, the NG-Large accelerator times are better than both the Zynq's and PYNQ's times, due to having double the memory interfaces, but because of the slow frequency of the NanoXplore's FPGA, both the Zynq's and PYNQ's result in faster total execution times.

From the total latency, it is possible to understand the overall performance of the memory on the board, while the average latency of the single memory transaction gives insights into why the total performance is as measured and what exactly degrades the performance of the design. One measurement without the other can be useful, but taking the data from both and confronting them improves the understanding of the design and the hardware. For example, by taking into consideration only the total latency, the memory reads would result in the slower operation; in contrast, by looking only at the average latencies, the

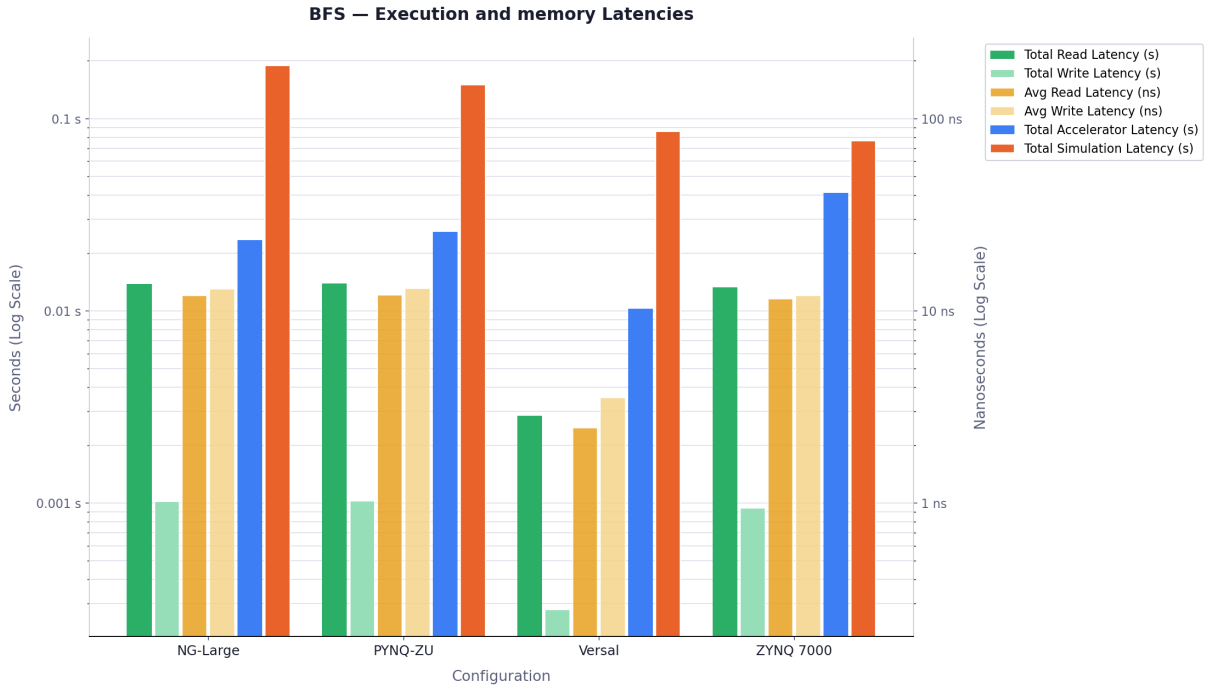

Figure 5.12: Latency comparison of the BFS algorithm implementations in various SoCs.

writes would result in the slowest operation. Ultimately, it is the sheer volume of read requests that drives up their overall latency. For certain algorithms, this volume cannot be determined a priori; the (PRP) algorithm, for example, must continuously iterate until it reaches convergence, so the number of requests is tied to the benchmark.

Memory measurements alone are not enough, as mentioned before, software overheads and physical limitations bring non-negligible execution slowdowns. As presented by the results, the physical limitations of the NG-Large SoC make it the slowest hardware taken into consideration in this thesis, and for the same reason, the Zynq's and PYNQ's accelerator times are slower, due to the low number of parallel memory interfaces. The Versal, which should be the best-performing board, falls behind the Zynq 7000's total execution time due to the overheads tied to memory synchronization.

5.8. Evaluation of different memory technologies on a single SoC

An FPGA could support more than a single type of memory; for example, the Zynq 7000 supports various models of DDR3 and DDR2 DRAM memories. For this reason, another batch of experiments has been conducted, testing the same BFS implementation, 32 cores, and 8 context switches, on a single board, the Zynq 7000, but each experiment presents

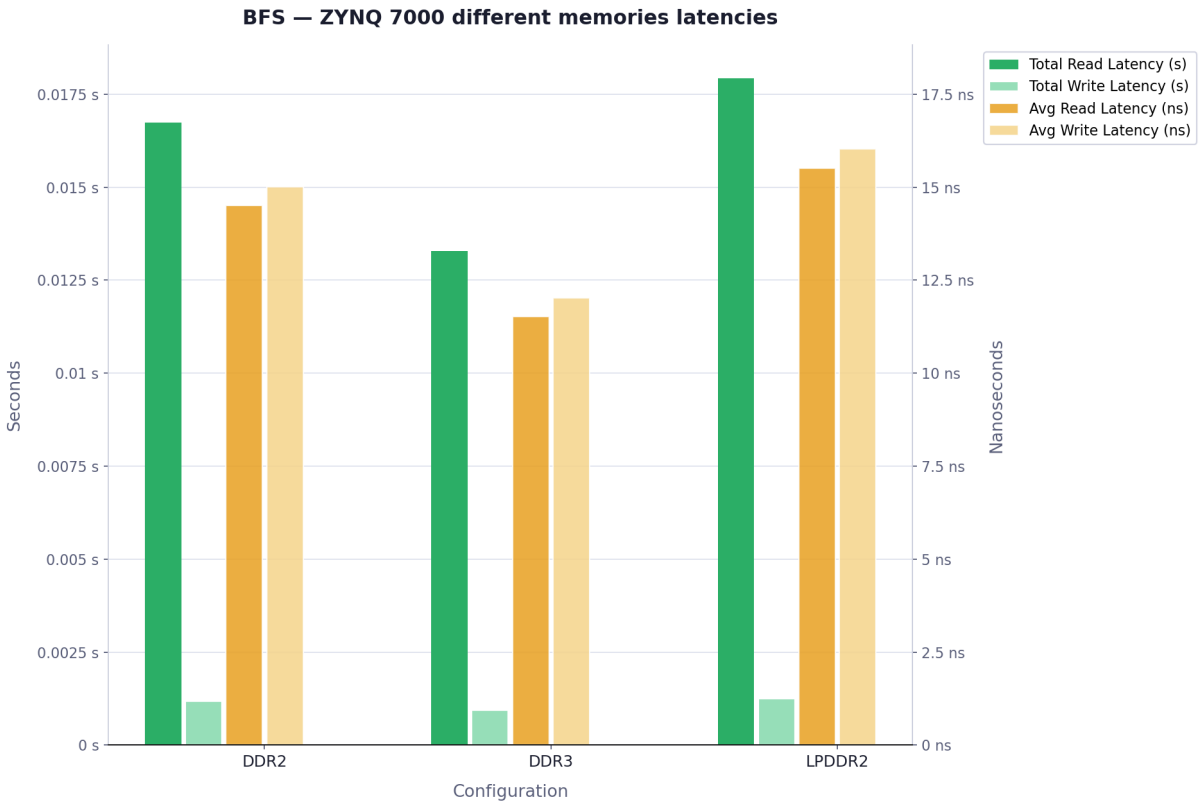

Figure 5.13: Latency comparison of different memory technologies attached to a Zynq 7000 with the BFS kernel.

a different memory technology: DDR3, DDR2, and Low Power DDR2.

The setup for this experiment is identical to the one from the previous batch, but instead of having 4 configurations for 4 boards, in this batch 3 configurations for a single board.

5.8.1. Experiment results

The results of these experiments are shown in Figure 5.13. The graph presents the measurements of the 3 simulations: total latency of memory requests, and average latency of memory requests, both quantities have been measured for both read and write memory requests.

The results present a predictable trend, for both reads and writes, the DDR3 DRAM results in the most performance, followed by the DDR2 and LPDDR2. These results might be predictable; however, they show the capability of this work as a tool to evaluate the performance, not only of the processor, but also of the memory system of a SoC.

6 | Related work

Simulating complex heterogeneous systems is not a trivial task; in the past, various simulator frameworks have been proposed, but no single framework is without limitations. Each framework has its requisites and its goals; some might care only about the RTL simulation, while others will focus on accuracy. gem5+RTL's goal is to provide a flexible and accurate Full-System simulation, trading off fast execution for cycle-level accuracy and the ability to run the framework on any machine, without the need for special hardware. This chapter reviews the principal frameworks for heterogeneous-system simulation and positions gem5+RTL among them. Each work is examined along two axes (whether it simulates RTL alone or the full system, and whether it depends on dedicated hardware), highlighting the trade-offs that distinguish it from this thesis.

6.1. FPGA simulation

gem5+RTL's accuracy derives from its ability to simulate RTL code, achieving cycle-level accuracy. In the academic world, many works can simulate and measure RTL accurately, but this entails trade-offs: most of these simulators ignore the rest of the system, both hardware and software, or require specialized hardware.

One of this work's downsides is the lack of speed in executing simulations; ERAS [49] aims to improve simulation speed and ease development. To increase simulation speed, ERAS compromised by not simulating the Full-System; it still simulates a computer architecture with CPU and a memory hierarchy using Structural Simulation Toolkit (SST) [50], but without considering the OS overheads, it loses on accuracy, even if the FPGA is RTL simulated. ERAS's ability to automate the integration of new IPs, together with a considerable speed-up over gem5+RTL in simulation times, makes it not only a valuable alternative to this work, but also a complementary tool, usable alongside this work in earlier development stages where speed matters more than evaluation accuracy.

Similarly, Nex+DSim [47] trades accuracy for speed, but approaching the problem in a fundamentally different manner, while ERAS simulates everything and accelerates the

simulation using SST's multi-thread support, Nex+DSim runs the execution on available hardware natively, simulating only any missing components and the critical part, the accelerator. By narrowing the simulation target and by omitting the OS, flexibility and accuracy are reduced, but simulation times are improved.

6.1.1. Hardware based frameworks

To overcome the speed limitations inherent to cycle-accurate RTL simulation, a class of simulators utilizes dedicated hardware to decrease execution time without sacrificing the accelerator simulation accuracy. This comes at the disadvantage of needing extra hardware to simulate an accelerator; two frameworks found a compromise to this problem in two different approaches. RAMP [58] is an accurate architecture simulator framework, specialized in parallel designs. It is as flexible as software simulators, but faster, while being less expensive than frameworks that need custom hardware, making it perfect to develop new hardware and systems. RAMP doesn't eliminate the need for external hardware, but proposes different solutions to ease development for various use cases. RAMP Red has hardware support for transactional memory, and RAMP Blue is a family of emulated machines that can run highly parallel applications or model a networked server cluster. Another prototype is RAMP White, which is a proof of concept to demonstrate the modularity of the framework by integrating components from both RAMP Red and Blue.

On the same premise of RAMP, Manticore [25] proposes a parallel computer to accelerate RTL simulation. Manticore uses a highly parallel execution model and a compiler-based scheduler to increase simulation speed dramatically, especially for the simulation of high-frequency accelerators. The hardware proposed consists of hundreds of small and simple cores to exploit fine-grained parallelism. With respect to Verilator, Manticore proved to be 4× times faster.

On the other hand, RTLFlow [44] proposes to run simulations not on FPGAs but on GPUs. The framework enables cycle-level accuracy of accelerator simulations while keeping a fast execution and eliminating the need for custom and/or costly hardware. The RTL code is translated directly to CUDA, providing support for any recent NVIDIA graphics card; this way, the end user can choose how much to spend on the simulator hardware while also decreasing development times, since graphics cards are easily available. RTLFlow avoids the considerable cost and procurement delays related to the acquisition of real hardware.

Table 6.1: FPGA simulation frameworks.

Work	Target	Extra HW	FS support
RAMP	Parallel multiprocessors	Yes, FPGA	Yes
Manticore	Arbitrary RTL	Yes	No
RTLFlow	Arbitrary RTL	Yes, GPU	No
ERAS	RTL IP blocks (CPU + memory via SST)	No	No
Nex+DSim	Accelerator (rest run natively)	No	Partial
gem5+RTL (this work)	Any FPGA accelerator	No	Yes

6.2. Full system simulation

One of this work's goals is to have a Full-System simulation, meaning that both pre-RTL and RTL code have to be simulated, together with the execution of the operating system. This is necessary to obtain accurate results and a correct design space exploration during the development of a system: the processor of a board may introduce unexpected limitations not accounted for in FPGA-only simulation, while simulating the OS provides the end user with useful insight into real overheads and slowdown introduced by it.

6.2.1. gem5 family

As the name suggests, this work is based on the gem5 simulator [21]. From this work, a lot of different simulators branched, creating simulators, similar to this thesis, that provide Full-System simulation of both FPGA+CPU hardware and operating system software. The most relevant are gem5-SALAM [51] and gem5-SALAMv2 [54], gem5-accel [57], gem5-nvdla [41], Rhea [61], and gem5-aladdin [52].

As mentioned before, all these works share the same base, the gem5 simulator, but each has a different approach and different goals. This work aims to have a cycle-accurate simulation and support direct RTL simulation; to achieve this, simulation times are considerably higher than the pre-RTL simulators, like gem5-SALAM and gem5-accel.

Close to this work, gem5-aladdin was born to create a cycle-accurate Full-System simulator, but despite its accuracy compared to real hardware, it assumes that all data has been pre-loaded into the local scratchpads, skipping the modeling of any interactions between accelerators and the rest of the system, failing to fully integrate with gem5 as opposed to this work, which includes full memory transaction models and measurements.

From this lack of system integration of the accelerator, gem5-SALAM was developed, providing complete memory modeling but focusing on speed instead of cycle-accuracy; by

simulating accelerators through LLVM, gem5-SALAM not only achieves greater speeds than RTL simulators like this thesis, but also greatly improves development speed, since adding or modifying an accelerator does not require recompiling the simulator, on the contrary, this work needs to fully compile gem5, whenever adding a new accelerator. Similar to gem5-SALAM, gem5-accel also focuses on fast development and simulation, it also provides Full-System support, and its purpose is not to substitute accurate RTL simulators, but rather to precede in development that kind of such tools by providing the user with an accurate simulator before committing to actually write RTL code.

While gem5+RTL aims to provide an accurate simulation for any FPGA accelerator, Rhea [61] narrows down the scope by simulating and validating more complex memory subsystems. Rhea uses the same stack gem5/Verilator to achieve an accurate Full-System simulation, inheriting the same time overheads as gem5+RTL. The difference between the two works stands in the scope of the simulation: gem5+RTL is very flexible, but lacks dedicated support for modeling or validating complex RTL cache-coherent memory subsystems. gem5-nvdla instead adopts a fundamentally different strategy, which is based on gem5+RTL, and in particular on its ability to simulate the NVDLA accelerator. gem5-nvdla provides the same cycle-level accuracy, being an RTL simulator, but sacrifices flexibility for speed: while the other gem5-based simulators can simulate any accelerator trading speed for accuracy or vice versa, this work focuses on only the NVDLA accelerator, improving execution times without losing on accuracy, trading flexibility for speed. The differences and trade-offs have been reported and summarized in Table 6.2.

Table 6.2: gem5 derived simulators.

Work	Target	FS support	RTL support	Fast
gem5-accel	Focus on ML accelerators	Yes	No	Yes
gem5-Aladdin	Any FPGA accelerator	Partial	No	Yes
Rhea	Cache coherent, memory subsystems	Yes	Yes	No
gem5-NVDLA	NVDLA deep-learning accelerator	Yes	Yes	Yes
gem5-SALAM	Any FPGA accelerator	Yes	No	Yes
gem5+RTL (this work)	Any FPGA accelerator	Yes	Yes	No

6.2.2. Other FS simulators

gem5 is, however, not the only system simulator used for heterogeneous systems; some simulators like Multi2Sim [56] even already support heterogeneous systems by extending the software with GPU simulation support. However, not many include heterogeneous CPU-FPGA system simulation; Heterosim [27] provided a solution, different from gem5,

to simulate accurately FPGA-accelerated systems. Heterosim is based on the aforementioned Multi2Sim and its ability to simulate CPU+GPU systems; it is very flexible and accurate, however, it is limited in the range of CPU architectures supported, as it can only simulate x86 ISAs. In contrast, gem5-based solutions can simulate x86, ARM and RISC-V depending on the version. In this thesis, the simulated architectures were ARM64 and ARM32.

To include the overheads and restrictions introduced by the processor and the operating system, but without losing speed and efficiency, some works include the use of real FPGA hardware alongside a conventional system. Works like Chimera [30] and Chipyard [9] use this approach to improve speed without losing accuracy. Chimera's simulation speed is high, but at the cost of needing to run the simulation on an FPGA. This speed-up is achieved through a synchronization mechanism between the FPGA board and the rest of the system, where the RTL is simulated on the accelerator, while the rest of the system is simulated on traditional hardware through the use of gem5. One of the goals for gem5+RTL is to be able to simulate, test, and do design exploration before having to even buy real hardware; for this reason, Chimera's approach can be impossible for certain use cases. In the middle between Chimera and gem5+RTL stands Chipyard: it needs dedicated hardware, like Chimera, but its workflow does not require a physical FPGA; instead, its scope is to be run on an AWS server with FPGA access, making simulation and testing more affordable than the Chimera approach.

Table 6.3: Core comparison of FPGA/accelerator simulation works.

Work	**Need for extra hardware**	**What it simulates**	**Full system simulation**
RAMP	Yes, FPGA	Parallel multiprocessors	Yes
Chimera	Yes, FPGA	RTL IP (decoupled from gem5)	Yes
Chipyard	Yes, FPGA	Custom RISC-V SoCs	Yes
RTLFlow	Yes, GPU	Arbitrary RTL	No
Manticore	Yes	Arbitrary RTL	No
ERAS	No	RTL IP blocks	No
NEX + DSim	No	Accelerators (rest run natively)	Partial
gem5-accel	No	ML accelerators	Yes
gem5-Aladdin	No	Fixed-function accelerators	Yes
gem5-NVDLA	No	NVDLA deep-learning accelerator	Yes
gem5-SALAM	No	Heterogeneous CPU+accelerator systems (LLVM-modeled)	Yes
HeteroSim	No	x86 multicore + FPGA	Yes
Rhea	No	Cache coherent, memory subsystems	Yes
gem5+RTL (this work)	No	Heterogeneous CPU+RTL systems	Yes

7 | Conclusions

In the current simulators landscape, gem5+RTL is the only heterogeneous-system simulator that combines support for arbitrary RTL accelerators with cycle-accurate, Full-System simulation and no dedicated hardware requirement. From the analysis presented in Chapter 6, it is not only evident how difficult it is to create a good simulator, but also that no framework works as a substitute for another; on the contrary, some of the works taken into consideration in this study can be easily used alongside this thesis to improve development quality and speed.

The key contributions of this work lie in the extension of the gem5+RTL framework through the implementation of memory utilities to simulate the synchronization of a system's SPM, the integration of Bambu's HLS flow for the development of FPGA kernels, and the development of new memory interfaces to connect the Bambu-generated accelerators to the simulated system.

The integration of the Bambu HLS tool enabled this thesis to be used as a design-space exploration tool. Bambu can generate automatically different versions of the same accelerator, but with different parameters, to explore the performance/resource trade-offs of the design. The effectiveness of this thesis as a design-space exploration tool was demonstrated through various experiments, which showed how different grades of parallelization of an RTL implementation can hide the latencies tied to memory accesses.

To simulate the accelerators generated by Bambu, multiple memory interfaces have been implemented, together with the associated Verilator wrapper and gem5 object to integrate the hardware design with the gem5 simulator, and consequently with the memory system and the CPU of the simulated hardware.

To accurately simulate an RTL implementation, it is necessary to consider the latencies associated with loading and retrieving the accelerator's memory. For this purpose, two gem5 objects have been implemented to execute the loading of the input data into the SPM of the accelerator and the retrieval of the output data, once the computation has been completed. The object has been implemented to enable gem5 to produce a human-readable report with the latencies of these memory processes.

In summary, this thesis demonstrated that it was a flexible and accurate tool for the development of complex, highly parallel, and memory-intensive hardware designs.

7.1. Future Work

Future work could focus on improving the framework by generalizing the accelerator objects and making it possible to change the accelerator to simulate directly on gem5's Python configuration script, eliminating the constant need to completely recompile the simulator every time a different kernel has been implemented. Implementing this feature would significantly decrease development times.

As highlighted by the experiments presented in Chapter 5, the striping and unstriping of the data passed to the SPM slow down the execution of the host binary considerably. Optimizing these steps of memory handling represents a possible future improvement of this work.

While the current simulations provide valuable insights, future work must prioritize empirical validation on actual hardware. In particular, replicating the experiments conducted in this thesis on physical hardware will be necessary to rigorously verify and improve the simulator's accuracy.

To improve the speed of the simulations, it could be possible to substitute the RTL C++ model generated with Verilator with the C model generated by the work of Fiorito et al.[29]. The latter proposes the generation of a model, written in C, from the RTL description of an accelerator, which is derived from the HLS IR of Bambu instead of the generated HDL. The execution of the C model generated has been proven to be much faster than other RTL simulators, while ensuring behavioral fidelity with respect to the final HDL design that Bambu generates from the same IR. This option would benefit the work of this thesis by keeping the cycle-level accuracy while significantly improving simulation times.

Bibliography

[1] AMD alveo™ u55c high performance compute card, . URL https://www.amd.com/en/products/accelerators/alveo/u55c/a-u55c-p00g-pq-g.html.

[2] Modeling HBM2 memory controller, . URL https://arch.cs.ucdavis.edu/memory/2022/06/01/hbm-gem5-workshop.html.

[3] AMD. AMD versal™ HBM series VHK158 evaluation kit. . URL https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vhk158.html.

[4] AMD. AMD zynq™ 7000 SoCs website, . URL https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html.

[5] AMD. HBM intro AMD, . URL https://docs.amd.com/r/en-US/Vitis-Tutorials-Vitis-Hardware-Acceleration/HBM-Overview.

[6] AMD. Versal hbm-series, . URL https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/hbm-series.html.

[7] AMD. Vitis acclelerator examples, . URL https://github.com/Xilinx/Vitis_Accel_Examples/tree/main.

[8] AMD. Vitis high-level synthesis user guide, . URL https://docs.amd.com/r/en-US/ug1399-vitis-hls/Introduction.

[9] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanovic, and B. Nikolic. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. 40(4):10–21. ISSN 0272-1732, 1937-4143. doi: 10.1109/MM.2020.2996616. URL https://ieeexplore.ieee.org/document/9099108/.

[10] B. Anuradha and C. Vivekanandan. Usage of scratchpad memory in embedded systems — state of art. In *2012 Third International Conference on Computing, Com-*

munication and Networking Technologies (ICCCNT'12), pages 1–5. doi: 10.1109/ICCCNT.2012.6396100. URL https://ieeexplore.ieee.org/document/6396100.

[11] ARM. AMBA AXI and ACE protocol specification AXI3, AXI4, and AXI4-lite ACE and ACE-lite, . URL https://developer.arm.com/documentation/ihi0022/latest/.

[12] ARM. The ARM cortex-a9 processors, .

[13] ARM. ARM cortex-a72 MPCore processor technical reference manual, . URL https://developer.arm.com/documentation/100095/0002/introduction/product-documentation-and-design-flow/documentation.

[14] ARM. ARM website. cortex-a9 processor. . URL https://developer.arm.com/Processors/Cortex-A9.

[15] ARM. Cortex-a53 developer documentation, . URL https://developer.arm.com/documentation/ddi0502/g?lang=en.

[16] ARM. Cortex-a9 technical reference manual, . URL https://developer.arm.com/.

[17] ARM. Cortex-r5 instruction fetch delay, . URL https://developer.arm.com/documentation/ddi0460/d/Cycle-Timings-and-Interlock-Behavior/About-cycle-timings-and-interlock-behavior/Instruction-execution-overview.

[18] ARM. Cortex-a series programmer's guide, . URL https://developer.arm.com/.

[19] S. Beamer, K. Asanović, and D. Patterson. The GAP benchmark suite. URL http://arxiv.org/abs/1508.03619.

[20] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 8–15. IEEE. ISBN 978-1-4673-2243-0 978-0-7695-4768-8. doi: 10.1109/ASAP.2012.30. URL https://ieeexplore.ieee.org/document/6341448/.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. 39(2):1–7, . ISSN 0163-5964. doi: 10.1145/2024716.2024718. URL https://dl.acm.org/doi/10.1145/2024716.2024718.

[22] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The m5

simulator: Modeling networked systems. 26(4):52–60, . ISSN 0272-1732. doi: 10.1109/MM.2006.82. URL http://ieeexplore.ieee.org/document/1677503/.

[23] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. 26(4):8–17. ISSN 0740-7475, 1558-1918. doi: 10.1109/MDT.2009.69. URL https://ieeexplore.ieee.org/document/5209958/.

[24] J. Daumal. NG-LARGE nanoxplore wiki. URL https://nanoxplore-wiki.atlassian.net/wiki/spaces/NAN/pages/11272201/NG-LARGE.

[25] M. Emami, S. Kashani, K. Kamahori, M. S. Pourghannad, R. Raj, and J. R. Larus. Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 219–237. ACM. ISBN 979-8-4007-0394-2. doi: 10.1145/3623278.3624750. URL https://dl.acm.org/doi/10.1145/3623278.3624750.

[26] F. A. Endo, D. Courousse, and H.-P. Charles. Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5. In *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 266–273. IEEE. ISBN 978-1-4799-3770-7. doi: 10.1109/SAMOS.2014.6893220. URL https://ieeexplore.ieee.org/document/6893220.

[27] L. Feng, H. Liang, S. Sinha, and W. Zhang. HeteroSim: A heterogeneous CPU-FPGA simulator. 16(1):38–41. ISSN 1556-6056. doi: 10.1109/LCA.2016.2615617. URL http://ieeexplore.ieee.org/document/7585071/.

[28] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. Invited: Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330. IEEE. ISBN 978-1-6654-3274-0. doi: 10.1109/DAC18074.2021.9586110. URL https://ieeexplore.ieee.org/document/9586110/.

[29] M. Fiorito, S. Curzel, and F. Ferrandi. Augmented co-simulation for fast functional and system-level verification of HLS accelerators. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. doi: 10.1109/ICCAD66269.2025.11240905. URL https://ieeexplore.ieee.org/document/11240905. ISSN: 1558-2434.

[30] C. Fu, Z. Wang, and J. Han. Chimera: A co-simulation framework combining with gem5 and FPGA platform for efficient verification. In *2024 34th Interna-*

tional Conference on Field-Programmable Logic and Applications (FPL), pages 133–139. IEEE. ISBN 979-8-3315-3007-5. doi: 10.1109/FPL64840.2024.00027. URL https://ieeexplore.ieee.org/document/10705631/.

[31] gem5 Community. Gem5 doxygen, . URL https://doxygen.gem5.org/release/current/index.html.

[32] gem5 Community. Gem5 tutorial, . URL https://www.gem5.org/documentation/learning_gem5/introduction/.

[33] gem5 Community. gem5 resouces, . URL https://resources.gem5.org/.

[34] G. Gozzi, M. Fiorito, S. Curzel, C. Barone, V. G. Castellana, M. Minutoli, A. Tumeo, and F. Ferrandi. SPARTA: High-level synthesis of parallel multi-threaded accelerators. 18(1):1–30. ISSN 1936-7406, 1936-7414. doi: 10.1145/3677035. URL https://dl.acm.org/doi/10.1145/3677035.

[35] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. N. Udipi. Simulating DRAM controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 201–210. IEEE. ISBN 978-1-4799-3606-9 978-1-4799-3604-5. doi: 10.1109/ISPASS.2014.6844484. URL http://ieeexplore.ieee.org/document/6844484/.

[36] Z. Horvat, V. Ilic, and M. Nikolic. Web server and QR decoder applications for xilinx FPGA boards. In *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, pages 148–151. IEEE. ISBN 978-1-5386-4927-5. doi: 10.1109/ZINC.2018.8448513. URL https://ieeexplore.ieee.org/document/8448513/.

[37] M. Humrick. ARM cortex-a72 architecture deep dive. URL https://www.tomshardware.com/reviews/arm-cortex-a72-architecture,4424.html.

[38] Joonyoung Kim and Younsu Kim. HBM: Memory solution for bandwidth-hungry processors. In *2014 IEEE Hot Chips 26 Symposium (HCS)*, pages 1–24. IEEE. ISBN 978-1-4673-8883-2. doi: 10.1109/HOTCHIPS.2014.7478812. URL http://ieeexplore.ieee.org/document/7478812/.

[39] H. Jun, J. Cho, K. Lee, H.-Y. Son, K. Kim, H. Jin, and K. Kim. HBM (high bandwidth memory) DRAM technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE. ISBN 978-1-5090-3274-7. doi: 10.1109/IMW.2017.7939084. URL http://ieeexplore.ieee.org/document/7939084/.

[40] K. Kim and M.-j. Park. Present and future, challenges of high bandwith memory (HBM). In *2024 IEEE International Memory Workshop (IMW)*, pages 1–4. IEEE.

ISBN 979-8-3503-0652-1. doi: 10.1109/IMW59701.2024.10536972. URL https://ieeexplore.ieee.org/document/10536972/.

[41] C. Lai and W. Zhang. gem5-NVDLA: A simulation framework for compiling, scheduling, and architecture evaluation on AI system-on-chips. 29(5):1–20. ISSN 1084-4309, 1557-7309. doi: 10.1145/3661997. URL https://dl.acm.org/doi/10.1145/3661997.

[42] C. Lam. ARM cortex-a53 tiny but important, . URL https://chipsandcheese.com/p/arms-cortex-a53-tiny-but-important.

[43] C. Lam. ARM cortex-a72 for the masses, . URL http://chipsandcheese.com/p/arms-cortex-a72-aarch64-for-the-masses.

[44] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang. From RTL to CUDA: A GPU acceleration flow for RTL simulation with batch stimulus. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–12. ACM. ISBN 978-1-4503-9733-9. doi: 10.1145/3545008.3545091. URL https://dl.acm.org/doi/10.1145/3545008.3545091.

[45] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, and Wood. The gem5 simulator: Version 20.0+. URL http://arxiv.org/abs/2007.03152.

[46] G. López-Paradís, A. Armejach, and M. Moretó. gem5 + rtl: A framework to enable RTL models inside a full-system simulator. In *50th International Conference on Parallel Processing*, pages 1–11. ACM. ISBN 978-1-4503-9068-2. doi: 10.1145/3472456.3472461. URL https://dl.acm.org/doi/10.1145/3472456.3472461.

[47] J. Ma, J. Kaufmann, E. Guandalino, R. Iyer, T. Bourgeat, and G. Candea. Fast end-to-end performance simulation of accelerated hardware-software stacks. In *Pro-*

ceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, pages 341–358. ACM. ISBN 979-8-4007-1870-0. doi: 10.1145/3731569.3764825. URL https://dl.acm.org/doi/10.1145/3731569.3764825.

[48] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. 33(4):92–99. ISSN 0163-5964. doi: 10.1145/1105734.1105747. URL https://dl.acm.org/doi/10.1145/1105734.1105747.

[49] S. Nema, S. K. Chunduru, C. Kodigal, G. Voskuilen, A. F. Rodrigues, S. Hemmert, B. Feinberg, H. Lee, A. Awad, and C. Hughes. ERAS: A flexible and scalable framework for seamless integration of RTL models with structural simulation toolkit. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 196–200. IEEE. ISBN 979-8-3503-0317-9. doi: 10.1109/IISWC59245.2023.00038. URL https://ieeexplore.ieee.org/document/10289370/.

[50] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. The structural simulation toolkit. 38(4):37–42. ISSN 0163-5999. doi: 10.1145/1964218.1964225. URL https://dl.acm.org/doi/10.1145/1964218.1964225.

[51] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi. gem5-SALAM: A system architecture for LLVM-based accelerator modeling. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 471–482. IEEE. ISBN 978-1-7281-7383-2. doi: 10.1109/MICRO50266.2020.00047. URL https://ieeexplore.ieee.org/document/9251937/.

[52] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks. Co-designing accelerators and SoC interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. doi: 10.1109/MICRO.2016.7783751. URL https://ieeexplore.ieee.org/abstract/document/7783751/?casa.

[53] W. Snyder. Verilator the fast free verilog simulator. URL https://www.veripool.org/verilator/.

[54] Z. Spencer, S. Rogers, J. Slycord, and H. Tabkhi. Expanding hardware accelerator system design space exploration with gem5-SALAMv2. 154:103211. ISSN 13837621. doi: 10.1016/j.sysarc.2024.103211. URL https://linkinghub.elsevier.com/retrieve/pii/S1383762124001486.

[55] TUL. AUP PYNQ-ZU. URL https://www.amd.com/en/corporate/university-program/aup-boards/pynq-zu.html.

[56] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2sim: a simulation framework for CPU-GPU computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370865. URL https://dl.acm.org/doi/10.1145/2370816.2370865.

[57] J. Vieira, N. Roma, G. Falcao, and P. Tomás. gem5-accel: A pre-RTL simulation toolchain for accelerator architecture validation. 23(1):1–4. ISSN 1556-6064. doi: 10.1109/LCA.2023.3329443. URL https://ieeexplore.ieee.org/document/10304264.

[58] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research accelerator for multiple processors. 27(2):46–57. ISSN 0272-1732. doi: 10.1109/MM.2007.39. URL http://ieeexplore.ieee.org/document/4287395/.

[59] Xilinx. Vitis-tutorials. URL https://github.com/Xilinx/Vitis-Tutorials/tree/7638918fe73e5a531ba75a76f18e76504939122c.

[60] S. Zhou, C. Chelmis, and V. K. Prasanna. Optimizing memory performance for FPGA implementation of pagerank. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE. ISBN 978-1-4673-9406-2. doi: 10.1109/ReConFig.2015.7393332. URL http://ieeexplore.ieee.org/document/7393332/.

[61] D. Zoni, A. Galimberti, and A. Guarisco. Rhea: a framework for fast design and validation of RTL cache-coherent memory subsystems. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE. ISBN 979-8-3315-1560-7. doi: 10.1109/ICCAD66269.2025.11240659. URL https://ieeexplore.ieee.org/document/11240659/.

List of Figures

List of Tables

Acknowledgements

Vorrei ringraziare la mia relatrice, la professoressa Serena Curzel, per l'opportunità di lavorare a questa ricerca e per avermi accompagnato durante questi lunghi mesi di lavoro. Un grazie anche a Guillem Lopez del BSC e a Giovanni Gozzi, che mi hanno aiutato con i problemi più tecnici e particolari.

Un grazie speciale va alla mia famiglia, in particolare a mia madre che mi ha sempre sostenuto fin dal liceo per permettermi di completare questo mio percorso accademico al Politecnico di Milano.

Ringrazio anche i miei amici e colleghi, sia a Isernia che a Milano, che durante questi anni mi hanno accompagnato con il loro sostegno, in particolare coloro con cui ho condiviso le ore di studio, in aula, in biblioteca e in patio, durante le lezioni, ma soprattutto durante i progetti.

:q!