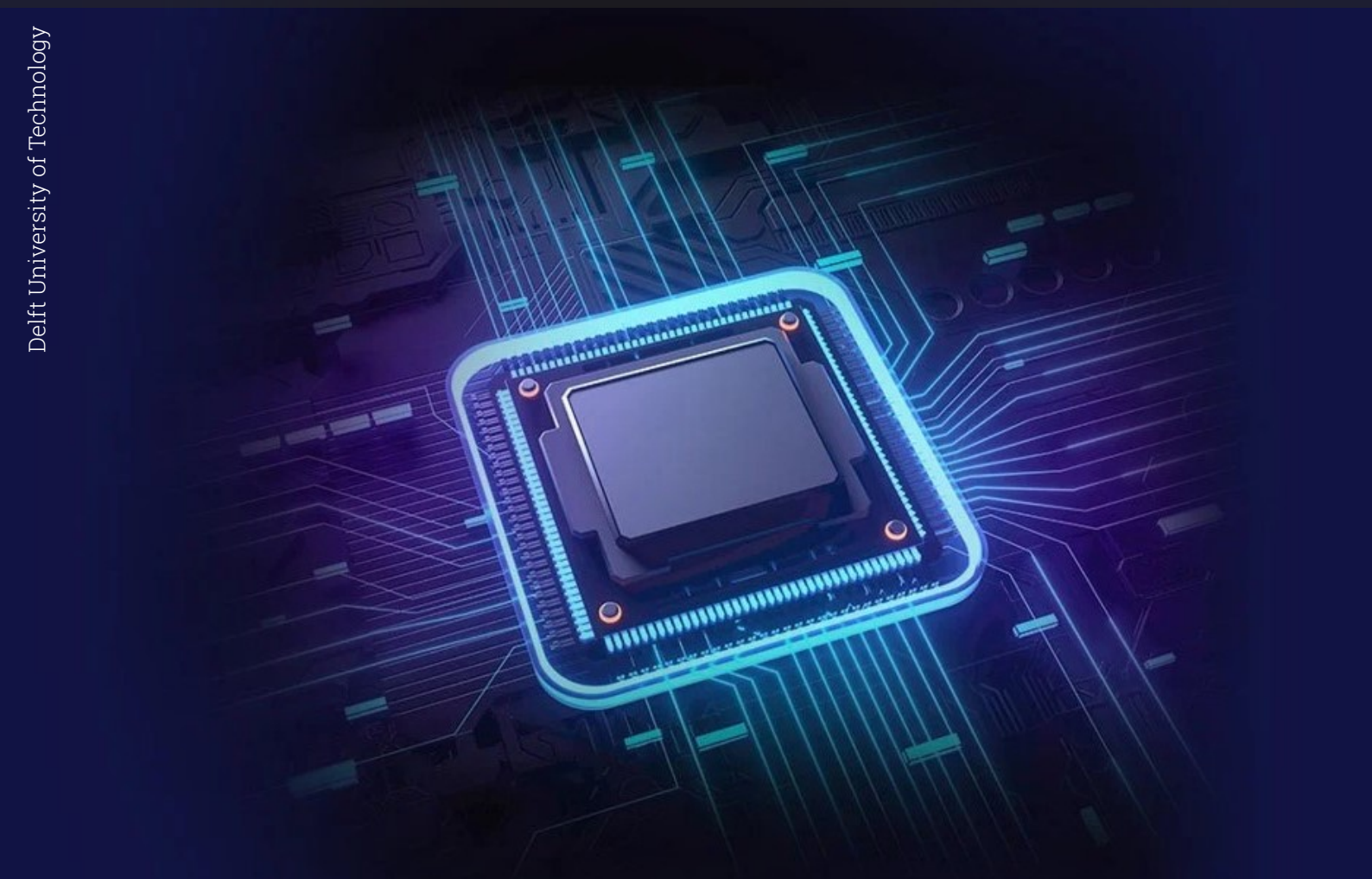
Building an Event-Driven Timing Simulator for Embedded Hybrid GPU-AI Accelerator

Dimitra Karatza

Delft University of Technology



TU Delft

Building an Event-Driven Timing Simulator for Embedded Hybrid GPU-AI Accelerator

by

Dimitra Karatza

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on September 15, 2023.

Student number: 5624525
Project duration: December 1, 2022 – August 1, 2023
Thesis committee:

Prof. S. Wong,	TU Delft, supervisor
Prof. A. Katsifodimos,	TU Delft
Dr. G. Keramidas,	Think Silicon S.A., an Applied Materials Company

This thesis is confidential and cannot be made public until September 15, 2025.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.
Cover: Think Silicon | ULTRA-LOW POWER GPUs (Modified)

TU Delft

Preface

I would like to express my heartfelt gratitude to all those who have contributed to the completion of this thesis.

First and foremost, I am deeply grateful to my supervisor and committee member, Professor Stephan Wong, for their invaluable guidance, unwavering support, and profound insights throughout this research journey. Their expertise and mentorship have played an instrumental role in shaping and refining this work.

I would also like to extend my sincere thanks to Professor Asterios Katsifodimos for their valuable feedback and constructive criticism as a member of my thesis committee. Their input has significantly enhanced the quality and depth of this thesis.

My heartfelt appreciation goes out to the continuous support and guidance of my mentor, Dr Georgios Keramidas, who not only provided expertise but also served as a member of my thesis committee. Their practical insights and unwavering dedication have been a constant source of inspiration and have immensely enriched this research.

My time at Think Silicon was enriched by my colleagues' collaborative spirit, stimulating discussions, and invaluable contributions. Their diverse perspectives and collective knowledge have greatly influenced the outcome of this work.

I would also like to acknowledge the unwavering support and resources provided by Delft University of Technology and the Faculty of Electrical Engineering, Mathematics, and Computer Science. Their commitment to academic excellence has been pivotal in the successful completion of this thesis.

I am deeply indebted to HELLENiQ ENERGY for awarding me the Proud of Youth Scholarship. Their generous support and investment in my education have been instrumental in enabling me to pursue this research and bring this thesis to fruition.

I am also grateful to my friends, whose continuous support and encouragement have made this academic journey more enjoyable and inspiring.

Lastly, but most importantly, I want to express my profound gratitude to my family for their unconditional love, unwavering understanding, and constant encouragement. Their unwavering support has been the bedrock of my motivation and success.

To all those who have contributed, directly or indirectly, to the successful completion of this thesis, I extend my sincerest gratitude. Thank you for being an integral part of this significant milestone in my academic journey.

Dimitra Karatza
Delft, September 2023

Summary

The current trend towards the integration of artificial intelligence (AI) and graphics processing unit (GPU) technologies has resulted in the development of embedded hybrid GPU-AI accelerators, which offer high computational power and energy efficiency. One of the key challenges in designing such accelerators is ensuring their timing correctness, as any timing violation may lead to system failure and incorrect results. To address this challenge, timing simulators have been proposed as a promising solution, as they enable accurate and efficient timing analysis of these complex systems. Nevertheless, such simulators have their limitations: detailed ones, such as cycle-accurate simulators, have high accuracy but exhibit high overhead, while faster approaches, like event-driven simulators, usually cannot achieve high accuracy levels. Therefore, the question arises: How can we implement a timing simulator to achieve a balanced trade-off between accuracy and execution time?

In this context, we introduce NEOX-V, a cutting-edge RISC-V based GPU Processor optimized for GPGPU and AI workloads. However, the current NEOX-V product lacks timing information in its simulator. This has prompted us to use it as a case study to bridge the gap between accuracy and execution time in timing simulators. This is achieved through the implementation of a new feature: a timing simulator that utilizes event-driven modeling.

To assess the proposed simulator's accuracy and effectiveness, we employ a comprehensive validation framework, using diverse workloads and configurations, from simple micro-benchmarks to intricate AI tests. The results demonstrate that the timing simulator achieves an accuracy error below 8% when compared to the RTL equivalent for all applications, with a marginal increase in actual simulation time of only 0.7%. It is worth noting that the timing simulator's utility extends beyond predicting execution time; it also plays a crucial role in verifying the existing design and uncovering its limitations.

Overall, this thesis makes a significant contribution to the field of computer architecture by providing a powerful tool for the design, development, and evaluation of an embedded hybrid GPU-AI accelerator called NEOX-V. It is our hope that this work will inspire further research and development in this exciting and rapidly evolving field.

Contents

List of Figures

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
API	Application Programming Interface
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DLA	Deep Learning Accelerator
DMA	Direct Memory Access
FIFO	First In First Out
FP	Floating Point
GFX	Graphics
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HMI	Human-Machine Interface
IoT	Internet of Things
ISA	Instruction Set Architecture
MCU	Microcontroller Unit
ML	Machine Learning
MM	Matrix Multiplication
OEQ	Ongoing Event Queue
PEQ	Pending Event Queue
RTL	Register-Transfer Level
RTOS	Real-Time Operating System
SIMD	Single Instruction Multiple Data
SPRAM	Scratch Pad Random Access Memory
TMU	Tile Management Unit
TRS	Thread Ready Status

Glossary

Term	Definition
ARM	ARM is a family of reduced instruction set computer (RISC) instruction set architectures for computer processors, configured for various environments.
CUDA	It is a parallel computing platform and programming model developed by NVIDIA for general computing on GPUs.
FPGA	Field Programmable Gate Arrays (FPGA) are semiconductor devices designed to be configured after manufacturing. They provide flexibility, customizability, performance optimization and accelerated processing for a wide range of applications.
GCC	The GNU Compiler Collection (GCC) is an optimizing compiler produced by the GNU Project supporting various programming languages, hardware architectures and operating systems.

Term	Definition
LLVM	The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.
MOESI	A full cache coherency protocol. Each cache line in a processor's cache can be in one of the following states: Modified (M), Owned (O), Exclusive (E), Shared (S), Invalid (I).
NEOX-V	A RISC-V based GPU Processor optimized for AI and GPGPU applications. Also referred to as accelerator or GPU.
OpenCL	The Open Computing Language (OpenCL) is a framework used to write programs that execute across heterogeneous platforms such as CPUs, GPUs or FPGAs.
OpenGL	The Open Graphics Library (OpenGL) is a cross-language, cross-platform API used for rendering 2D and 3D vector graphics.
RISC-V	An open standard instruction set architecture based on established reduced instruction set computer principles.
RTL	A design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals. It is used in hardware description languages (HDLs) like Verilog and VHDL to create high-level representations of a circuit, from which lower-level representations and ultimately actual wiring can be derived.

Symbols

Symbol	Definition	Unit
e	Error of the timing model simulation compared to the RTL version	%
rtl_cycles	Execution time simulated by the RTL version	clock cycles (cc)
sim_cycles	Execution time simulated by the timing model	clock cycles (cc)

1 Introduction

1.1. Motivation

Moving from the mobile world to the rising IoT/Wearable era, requirements for performance and power are changing rapidly. Most IoT devices need to perform complex tasks such as sensing, communication, information processing, calculation, and the release of control commands. Adding AI and Machine Learning (ML) to the edge is the most promising way to scale these capabilities, thus moving the data processing close to the source of the data. Therefore, it becomes evident that such devices require high quality graphics displays, Artificial Intelligence (AI) based applications, as well as long battery life.

Typically, smart connected devices are powered by microcontrollers (MCUs). MCUs are simple, programmable, and fully integrated systems typically used for embedded control and, in general, consume little power while being relatively inexpensive. Systems focusing on MCU hardware are based on the so-called cloud-first architecture. In this architecture, every edge device is connected directly to the internet, then provisioned and managed through a cloud service. The problem with powering smart IoT display-based devices and wearables by MCUs is that MCUs typically are not equipped with a Graphics Processing Unit (GPU) leading to low quality human-machine interfaces (HMI) and poor user experience. Moreover, due to their limited processing capabilities, MCUs face difficulties in handling the computational demands of AI and delivering fast and efficient results.

Since MCUs are reaching their limits due to the demands arising by new applications, more advanced hardware, such as dedicated AI accelerators, is required. To perform such tasks, Think Silicon proposes a new GPU architecture called NEOX-V. NEOX-V is transferring ultra-low-power GPU technology from Graphics to AI. It is a multicore and multithreaded GPU based on RISC-V Instruction Set Architecture (ISA) with Graphics/AI extensions. The common base ISA between the CPU and the GPU enables new programming paradigms and dynamic load balancing techniques. Its architecture is heterogeneous meaning that the accelerator operates as a slave device within a master-slave system. To devise informed architectural decisions and end up with a balanced design in terms of performance-power-area (PPA), the company is developing a high-level (at C/C++ level) functional simulator. The current version of the simulator focuses on modeling the functional behavior of NEOX-V, without considering the execution time. However, timing information is important for developing designs with an optimal combination of performance, area, and power consumption.

1.2. Existing methods

RTL simulation is intractable for design space exploration, as it requires significant time and effort to create the design and evaluate it. On the other hand, architecture simulators offer a quick and efficient mean to explore the design space or evaluate the performance of an existing design. They do so by modeling the behavior and performance of a computing system (CPU, GPU, memory) at various levels of abstraction. There are different kinds of architecture simulators based on the detail of the simulation:

- Functional simulators: Their focus is to perform the same function as the targeted design. They are used to validate the correctness of a design rather than evaluate its performance.

- Timing simulators: These are functional simulators which also provide timing information. They predict the execution time in number of cycles, and, therefore, they can be utilized for performance evaluation.

Two widely used types of timing simulators are cycle-accurate simulators [1] and event-driven [2] simulators. Cycle-accurate simulators model a microarchitecture on a cycle-by-cycle basis, thus achieving very high accuracy. However, this level of detail often comes at the cost of increased simulation time due to the intricacies involved. Event-driven simulators, in contrast, focus on examining the most essential events and interactions in the system. By monitoring specific events that significantly affect performance, such as cache misses or branch mispredictions, event-driven simulators can achieve faster simulation times but lower accuracy compared to cycle-accurate simulators.

A recent development involves machine learning techniques which use available data and/or reinforcement learning algorithms to accurately identify patterns and correlations and finally predict the system's behavior and performance. Nonetheless, machine learning techniques require the presence of large amounts of data and extensive training and cannot always be reliable since they do not inherently model the underlying architecture.

Therefore, it is evident that all these different approaches offer various levels of modeling accuracy due to their different levels of complexity.

1.3. Research question

The selection of the simulation method is typically influenced by the desired level of accuracy and the impact on simulation time overhead. These factors are related, as a more detailed simulation method, such as cycle-accurate simulators, usually achieves higher accuracy at the cost of increasing the simulation time. To this effect, faster simulators, like event-driven simulators, are unable to achieve high accuracy levels as they lack simulation details. Considering the challenges posed by the trade-off between accuracy and simulation time, it becomes evident that developing an architectural simulator with timing characteristics is far from a trivial task that requires extended research. Therefore, the following research question arises:

How can we incorporate timing characteristics into a functional simulator to achieve a balanced trade-off between accuracy and execution time?

1.4. Requirements & Goals

To address this research question, the focus will be on developing a timing simulator specifically designed for NEOX-V. The acceptable requirements for the proposed solution are:

- The timing simulator's accuracy is defined as having a relative error of 20% compared to the RTL version in the generated results.
- The timing simulator should introduce an overhead of no more than 10% when added to the functional simulator.
- The timing simulator should be easily enabled or disabled.
- The proposed solution should be flexible and extensible to allow for easy configuration and adaptation to future changes and enhancements.

To meet these requirements the following goals are set:

- Attain a comprehensive understanding of the existing functional simulator, including its underlying principles, architecture, and functionalities.
- Define the different computational and memory components that require modeling.
- Enhance the existing simulator to provide a detailed understanding of the performance of the system, including its latency and throughput. This information will facilitate system performance optimization.
- Enrich the simulator to predict the execution time in number of cycles required for program execution.

1.5. Methodology

To achieve the aforementioned goals, the following methodology will be employed:

- Exploration of existing simulation methods to gain a comprehensive understanding of the current implementation of the simulator.
- Development of diverse micro-kernel benchmarks that incorporate a mix of instructions in relation to the number of threads. This enables a deeper understanding of the underlying hardware components and their interactions, as well as easier extraction of the latency and throughput metrics.
- Implementation of event-based modeling, which represents the system as a series of discrete events and assesses their impact on performance in terms of simulation cycles. This methodology aids in identifying the components that require modeling and understanding their behavior.
- Continuous validation against the RTL code. This will include the micro-kernel benchmarks mentioned earlier, real-time applications, and the development of an automatic test case generation tool. This extensive approach aims to cover both common and edge cases, enabling thorough testing and assessment of the simulator's accuracy and performance.

1.6. Thesis organization

The organization of this thesis is as follows. Chapter 2 includes all the required information that is needed to deeply understand the implementation and the results of the simulator. It describes the related work as well as the architecture of the NEOX-V accelerator. Chapter 3 presents the method that was followed, including all the implementation details, while Chapter 4 introduces the validation framework used to evaluate the proposed timing simulator. Chapter 5 delves into the extensive presentation and analysis of the results obtained from the validation process. Chapter 6 includes a summary of the current thesis; it highlights the main contributions and describes the possible future work that can enrich the current implementation.

2

Related Work and Background

In the previous chapter, an extensive explanation was provided regarding the motivation driving this research, the research question, the goals, and the chosen methodology. It has become evident that the increasing demands of the IoT landscape necessitate more advanced embedded GPUs and AI accelerators. However, simulating such hardware presents a non-trivial challenge that requires extensive research. The specific challenge lies in extending a functional simulator with timing characteristics, utilizing an event-driven technique that surpasses the limitations of traditional event-driven simulators. With this in mind, the focus now shifts towards gathering the essential information needed to effectively address the research question.

This chapter presents the related work and background information that is essential for comprehending the subject matter, offering a viable solution to the problem statement, and successfully achieving the objectives outlined in the thesis. Section 2.1 includes an overview of architecture simulators, analytical models, and machine learning techniques. Furthermore, Section 2.2 presents a detailed explanation of the NEOX-V accelerator, emphasizing its key characteristics and architecture. Additionally, it includes information about the current implementation details of NEOX-V as a product.

2.1. Simulation techniques

Over the years, GPU designers and researchers have been using multiple techniques to verify functional correctness, evaluate performance, and explore the design space. In this way, they can identify bottlenecks, perform optimizations and draw informed decisions about the design in a reasonable time frame. These techniques typically include detailed architectural models such as event-driven or cycle-accurate simulators, analytical models or even machine learning methods. This chapter explores the most relavant and state-of-the-art simulation techniques that provide insight into building an event-driven timing simulator for an embedded hybrid GPU-AI accelerator.

2.1.1. Architecture simulators

Architecture simulators are essential tools for modeling the behavior of computer systems, simulating the execution of instructions, and interactions among hardware components. In this context, we will present some popular simulators, focusing on the most relevant ones.

SniperSim [3] is a high-speed multi-core CPU simulator that supports timing simulations for multi-program workloads and multi-threaded, shared-memory applications. It employs statistical simulation methods and does not provide cycle-accurate simulations. While it offers valuable insights into CPU performance, it is not suitable for GPU or AI applications.

GPGPU-Sim [4], on the other hand, offers a detailed simulation model for contemporary GPUs running CUDA and/or OpenCL workloads. It provides cycle-accurate simulation but is customized for high-end GPUs, primarily targeting general-purpose GPU (GPGPU) applications. This means that it lacks support for graphics processing and specialized AI processing with dedicated components like scratchpads, limiting its applicability in certain scenarios.

For graphics-specific simulations, there are simulators like Attila [5] and Teapot [6], both of which support only the OpenGL API. While these simulators are useful for analyzing graphics processing, they do not cover other aspects of modern computer systems, which include AI workloads.

SimpleScalar [7] is a versatile simulator that models a virtual computer system with a CPU, cache, and memory hierarchy. Its simple design and support for different levels of simulation detail make it an ideal choice for understanding how timing simulators work.

Gem5 [8] [9] is another architecture simulator known for its versatility and modular approach, capable of simulating various hardware architectures, including CPUs, GPUs, and memory systems. It provides cycle-accurate simulation and offers a high level of detail for in-depth analysis.

Given their extensive capabilities, our analysis will focus on the latter two simulators, Simplescalar [7] and Gem5 [8] [9]. These simulators not only offer valuable insights into the design and performance evaluation of computer systems, but they can also serve as a source of inspiration for the development of a timing simulator for an embedded hybrid GPU-AI accelerator like NEOX-V.

SimpleScalar

SimpleScalar [7] is an architecture simulator which models the behavior of a computing system with CPU, cache, and memory hierarchy. It is designed to simulate the execution of real programs on modern processors and to track the microarchitecture state for each cycle. For this purpose, the tool set includes different simulators ranging from a fast functional simulator to a detailed timing simulator which models an out-of-order processor with branch prediction, speculative execution, caches, and external memory.

The out-of-order processor timing simulator supports out-of-order issue and execution and performs a detailed modeling of all the stages of the pipeline, and the associated structures such as instruction queue, register renaming unit, and reorder buffer. Figure 2.1 shows the pipeline stages, which are:

- Fetch: the instruction is fetched from the memory
- Dispatch: the instructions are decoded and RUU and LSQ resources are allocated
- Issue: the instructions which are ready to execute from a previous cycle are issued to the functional units
- Writeback: the results of the instruction are available and dependencies between non-issued instructions are checked
- Commit: the results of the executed instructions are committed to the architectural register file

The stages of the pipeline are traversed in reverse order (from the last to the first one) to eliminate relaxation problems that is to provide correct inter-stage latch synchronization. Hence, the main loop of the simulator is structured as follows:

```
ruu_init();
for (;;) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}
```

In more detail, the simulator incorporates a Load/Store queue (LSQ) and a Register Update Unit (RUU) to to accurately model the various stages of the system.

The LSQ saves the state of the load and store operations in program order. The available states of the load/store accesses are:

- issued: The address computation is complete and the memory access is in progress
- completed: The memory access has completed and the stored value is available
- squashed: The memory access was squashed which means that this entry should be ignored

It is important to note that when a store instruction is executed and its address is known, the value stored in memory can be bypassed by the LSQ immediately to a load instruction in the same cycle, effectively allowing the store to complete with zero time delay.

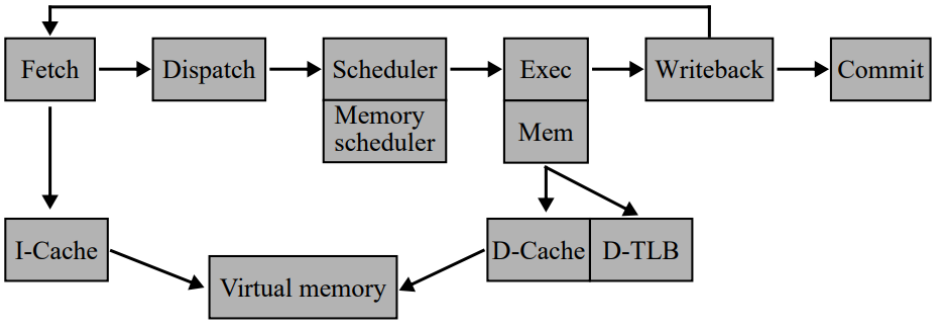


Figure 2.1: Pipeline stages of Simple Scalar [7]

The RUU entries hold the results of registers and wait for all operands to be ready. When this is done, the instruction is sent to the functional units. Instructions are added to the RUU in program order and the results are stored in its buffers. When an entry in the RUU becomes the oldest (and is ready), the corresponding instruction and its value are transferred, in program order, to the architectural register file.

Modeling of the pipeline stages includes the following process:

- At first, the instruction fetch unit -if not blocked- fetches as many instructions as allowed by a single branch prediction and cache line access, without overflowing the queue present between the fetch and dispatch stages.
- The instructions waiting in this queue are decoded, then resources in RUU and LSQ (for load/store operations) are allocated and the dependencies are updated accordingly in the dispatch stage.
- The next stage will try to issue instructions from the ready queue to the functional units. The ready queue contains instructions which have all register dependencies resolved. If a functional unit is available in this cycle to perform the execution, the instruction is issued and a writeback event is scheduled. Then the result is written into the LSQ, while the actual store into the memory system happens in the commit stage.
- During the writeback, the results from the functional units are written back to the RUU. At this point, the dependencies of the completed instructions are also checked to identify dependent instructions whose register operands are ready and can thus the ready instructions can be inserted in the ready queue.
- Finally, the results from the oldest completed entries of RUU and LSQ are committed to the register file and in case of a store operation the data cache is updated with the data from the LSQ.

While SimpleScalar is a very accurate and flexible simulator, it exhibits certain limitations. SimpleScalar supports features such as non-blocking caches, speculative execution, and branch prediction; thus, it may not be the most suitable solution for architectures that lack these specific characteristics. Moreover, the lack of thread-level parallelism limits SimpleScalar's ability to accurately evaluate the performance of multi-threaded architectures, while its limited instruction set support further restricts its applicability.

Gem5

Gem5 [8] [9] is a cycle-level computer architecture simulator widely used in academic research and industry. It is built upon an event-driven simulation engine and incorporates an extensive and customizable modeling of system components such as CPU cores (out-of-order designs, in-order designs, and others), a detailed DRAM model, on-chip interconnects, coherent caches, I/O devices, and many others.

The simulator can operate in two modes:

- System-call Emulation (SE) mode: Avoids the simulation of devices or operating system (OS) by emulating most system-level services.

- Full-System (FS) mode: Executes both user-level and kernel-level instructions and simulates the complete system including devises and OS.

As seen in Figure 2.2 (1), Gem5 supports various processor architectures, including Arm, GPU ISAs, MIPS, Power, RISC-V, SPARC, and x86. Most of these ISAs have some level of full system support. The simulator's modularity enables the seamless integration of these ISAs with the generic CPU models as the CPU models are designed to be ISA-agnostic (Figure 2.2 (2)). Four CPU models are supported by the simulator:

- Simple (the "simple" CPU): A faster simulation of the system that does not require the same fidelity as a real device.
- In-order (the "minor" CPU): High-fidelity simulation of an in-order processor. This model is slower, but can provide more realistic results.
- Out-of-order (the "O3" CPU): High-fidelity simulation of an out-of-order processor. Likewise, this model is slower, but can provide more realistic results.
- KVM (the "model" CPU): This model is used for fast-forwarding the simulation to the region of interest. After allowing the Linux booting process to quickly boot the system, researchers and developers can switch to the desired detailed CPU to run the benchmark.

  By leveraging the Kernel Virtual Machine (KVM) API in Linux, the simulation can bypass the usual software-based simulation (if the ISA of the application running on Gem5 complies with the ISA of the host) and take advantage of the underlying hardware virtualization support available in modern processors. This approach enables Gem5 to achieve near-native performance for the simulated binaries, as the execution is handled directly by the host processor. However, it does not model the timing of execution or memory request.

To facilitate the connection of various compute, memory, and I/O device models, the simulator employs a modular port interface (Figure 2.2 (3)), enabling seamless integration between different system designs.

Furthermore, it includes two different cache systems:

- Ruby (Figure 2.2 (4)), which models a variety of cache coherence protocols with high fidelity.
- "Classic" caches (Figure 2.2 (5)), which models a fast and easily configurable memory system, without the same cache coherence fidelity and flexibility. This cache system includes a single hard-coded hierarchical MOESI coherence protocol [10].

The memory system simulation is completed by a detailed configurable event-driven DRAM model, which supports different DRAM controllers as seen in Figure 2.2 (6).

Apart from the CPU models, Gem5 includes a cycle-level compute-based GPU (Figure 2.2 (7)) which is based on AMD's Graphics Core Next (GCN) architecture [11], [12]. It does not support graphics applications, but since the GPU model has a modular ISA similar to its CPU model, it can be extended to support other GPU ISAs.

Supporting I/O and various devices is a crucial aspect of comprehensive system simulation. Gem5 addresses this by providing compatibility with numerous system-specific but also system-agnostic devices (Figure 2.2 (8)) like disk controllers, PCI components, Ethernet controllers, and more.

Finally, it is worth mentioning that it has been seamlessly integrated with other computer architecture simulator systems, allowing users to leverage Gem5's features even when working with models from different simulator systems (Figure 2.2 (9)) such as the IEEE standard SystemC API [13].

Overall, Gem5 is a highly customizable and flexible simulation platform that enables users to explore various configurations, parameters, and levels of detailed modeling. However, like any software tool, it it exhibits certain inherent limitations. Gem5's comprehensive nature and intricate design make it a sophisticated simulator, necessitating users to invest time and effort to become proficient in its utilization. Moreover, conducting computationally intensive simulations and handling memory overhead can pose challenges, particularly for large-scale scenarios. While it offers detailed modeling for most components, there may be certain unsupported features. For instance, not all ISAs have complete full system support, while also the provided GPU model does not support graphics applications. Moreover, it is important to note that certain features of Gem5, such as the KVM-based model or the SE mode, prioritize execution speed over timing accuracy. Consequently, they may not precisely replicate the

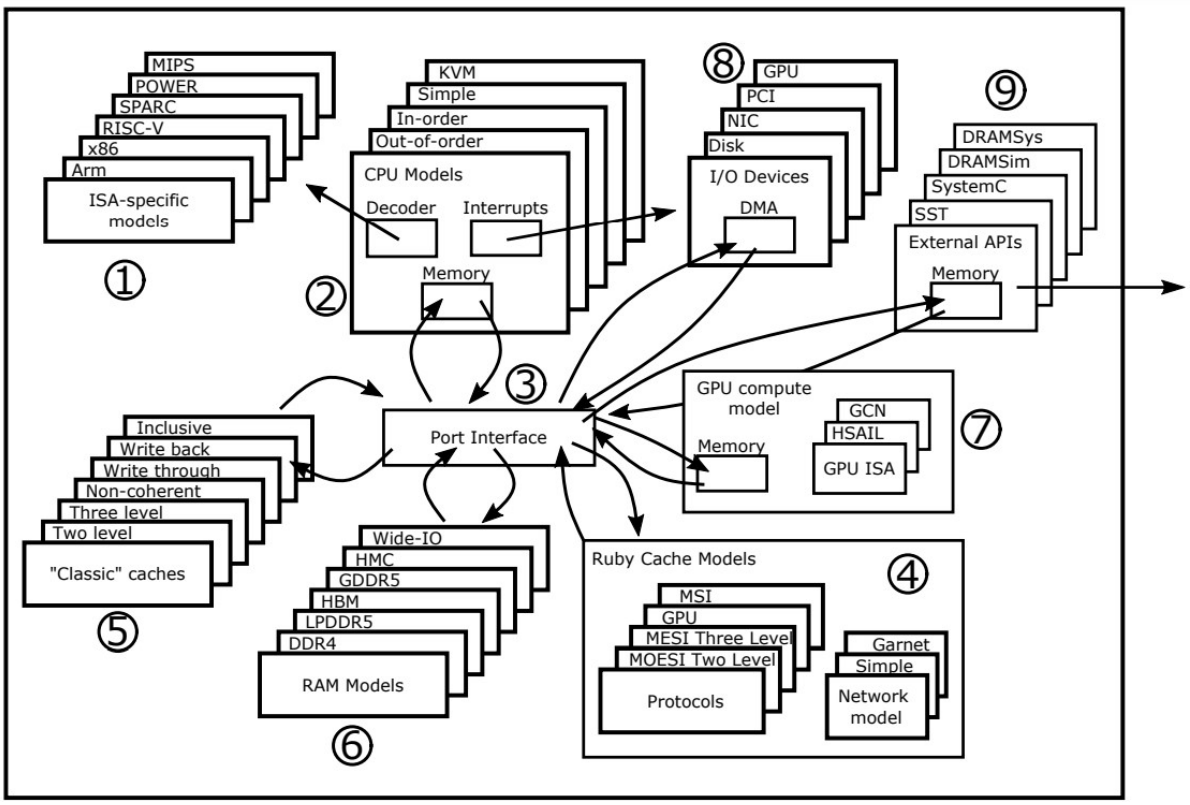


Figure 2.2: The architecture of Gem5 [9].

timing aspects of execution and memory accesses, limiting their suitability for meticulous performance analysis and timing-sensitive research.

2.1.2. Analytical models

In contrast to architecture simulators, analytical models have the advantage of involving less computational overhead since they don't perform explicit system simulation. Instead, analytical models employ mathematical or statistical techniques to derive performance equations or formulas based on simplified assumptions. While analytical models provide valuable analytical insights and predictions, they may sacrifice a certain level of accuracy and realism compared to the more detailed simulations offered by architecture simulators.

Mechanistic performance model

In 2009, a mechanistic model [14] was proposed for modern superscalar out-of-order processors, which considered their key microarchitectural features to achieve performance prediction. This detailed performance model uses interval analysis to break down the execution time into intervals disrupted by miss events. After analysing the type of miss events and predicting the execution time for each interval using a set of performance equations, the overall processor performance can be estimated by aggregating these individual predictions. The model has an average difference of 7% compared to simulation for a 4-wide superscalar out-of-order processor and is accurate across the processor design space.

Interval analysis is based on the premise that when a well-balanced superscalar out-of-order processor is free from miss events such as cache misses and branch mispredictions, it should be able to seamlessly move instructions through its pipelines, buffers, and functional units. However, this smooth streaming of instructions is interrupted by miss events, which can be used to split the execution time into intervals. These intervals are the primary entity used to form a mechanistic model, meaning that the modeling of the processor is based on the mechanisms that affect its performance. This analysis provides, thus, useful insight to the factors that impact processor performance, such as instruction-level parallelism, cache latency, branch prediction accuracy, instruction reordering, register renaming, and

speculative execution.

The mechanistic model is then used to explore resource scaling in out-of-order balanced processors as well as overprovisioned designs. A processor design is considered to be balanced if the resources are enough to achieve a specific throughput of instructions per cycle when miss events do not occur. The resources should not, however, exceed the required ones to achieve that, meaning that in the case of resource reduction, the processor should face performance degradation.

The mechanistic model was used to apply different pipeline configurations to explore how the pipeline depth, width, and their relationship affects a processor's performance. The resource scaling exploration showed the fundamentally greater affect that branch mispredictions have to a processor's performance compared to cache misses. The authors point out the implication a branch predictor has to a processor design due to the branch resolution time increasing with depth because of pipelining overhead.

Simple BSP-based Model

Another way to predict execution time in GPUs is using the Bulk Synchronous Parallel (BSP) model [15]. The BSP model is one of the most fine-grained models for parallel computing which treats the communications and computations as abstractions of a parallel system.

The model focuses on the prediction of the execution time of a kernel function and splits this execution time in two phases: communication and computation. In the computation phase, the model estimates the time required to execute the computational tasks, while in the communication phase the time required for data transfer between different processing elements is being estimated. The BSP model also takes into account the synchronization phase, where the model estimates the time required for the threads to synchronize after completing their tasks, but the authors concluded that this time is negligible and thus does not affect the execution time.

Using this abstraction the authors model the effect of optimizations and the heterogeneity properties of GPUs in a parameter λ of the equation 2.1 where T_k expresses the execution time of a kernel function, t is the number of threads, $Comp$ the computational cost, while $Comm_{GM}$ and $Comm_{SM}$ are the communication cost of accesses to the global and shared memory respectively. R is the clock rate, P the number of cores available in the GPU while the parameter models the effects of application optimizations, such as divergence, shared bank conflicts and coalesced global memory accesses. It is calculated as the ratio between the predicted execution time with the actual measured execution time of the application.

$$T_k = \frac{t * (Comp + Comm_{GM} + Comm_{SM})}{R * P * \lambda} \tag{2.1}$$

The number of cycles that each thread consumed for its computations defines the computational cost, while the communication cost is dominated by the load and store accesses to the global memory and shared memory. This cost is calculated using equations 2.2 and 2.3, where g_{SM}, g_{GM}, g_{L1}, g_{L2} represent the latency in communication over shared, global, L1 cache and L2 cache memory, respectively. ld_0 and st_0 represent the total number of load and store operations by all threads in the shared memory, while ld_1 and st_1 represent the accesses to the global memory. The values of these variables are determined by analysing the source code and by profiling the application execution.

$$Comm_{SM} = (ld_0 + st_0) * g_{SM} \tag{2.2}$$

$$Comm_{GM} = (ld_1 + st_1 - L1 - L2) * g_{GM} + L1 * g_{L1} + L2 * g_{L2} \tag{2.3}$$

The authors evaluate their model on two different GPU applications, including Matrix Multiplication and Maximum Subarray Problem using different boards. For most of the cases, the predictions were within 0.8 to 1.2 times the measured execution times, indicating that the model is able to provide generalized predictions across various problem sizes and GPU configurations.

2.1.3. Machine Learning techniques

In recent years, machine learning techniques have gained immense popularity for their ability to predict a wide range of outcomes. In line with this trend, the BSP-based model was also subjected to

comparison with various machine learning techniques to accurately predict the execution time of GPU programs [16].

The analytical modeling approach presented before involves manually constructing a mathematical model of the program's execution time based on its input parameters and the hardware architecture. The machine learning techniques are Linear Regression, Support Vector Machine and Random Forest. This approach involves training a model on a set of input-output pairs, where the inputs are the program's input parameters and the output is the execution time. The authors use a neural network model with multiple hidden layers to predict execution times.

The authors conclude that although the analytical model is able to give more accurate predictions, it necessitates computations to be performed for each application. On the contrary, machine learning may be less accurate but it is considered a more generalized approach that does not require any detailed knowledge of the hardware specifications, or application code. The predictions of the execution time of 9 different kernels over 9 different GPUs had accuracy between 0.8 and 1.2 for the analytical model while the accuracy of the machine learning approach varied between 0.5 and 1.5. The authors, thus, conclude that machine learning can be a valuable complement to analytical modeling in predicting GPU execution time, especially when dealing with complex relationships between input parameters and execution time.

2.2. The NEOX-V Accelerator

In this thesis, NEOX-V will serve as a case study to delve into the intricacies of architecture simulation and analysis. As an embedded hybrid GPU-AI accelerator, NEOX-V presents a valuable opportunity to develop an event-driven timing simulator. Through a detailed examination of NEOX-V, we aim to enhance our understanding of its capabilities and provide valuable contributions to the field of architecture simulation.

NEOX-V is a scalable, multi-core and multi-threaded embedded GPU designed by Think Silicon S.A., an Applied Materials company. It is based on the RISC-V RV64C open instruction set architecture; RISC-V is a new potential alternative for ARM in many new IoT/Edge applications (expand this). Its field of use varies from graphics rendering to GPGPU applications and Deep Neural Network workloads. AI inference and higher graphics performance is required for medium/large full HD display devices or connected endpoints or edge devices. Some examples are wearables, smart homes and factories, security, augmented reality, entertainment or automotive.

2.2.1. Main characteristics

The following characteristics of NEOX-V make it a suitable solution for both AI and Graphics workloads:

- Scalable Design: Multi-core and Multi-threaded 1-64 cores targeting from embedded market to high performance solutions
- Leverage RISC-V ecosystem and Tooling (GCC/LLVM)
- AI Inference: Neox AI SDK/TensorFlow Lite/MCU
- Graphics Rendering: Think Silicon graphics Libraries
- Small Compact Design with ultra low area and gate count
- Extensibility with user defined instructions

Figure 2.3 shows how a system using NEOX-V can be implemented, while the configuration of the accelerator can be seen in table 2.2.1.

SoC characteristics	
Operating System	RTOS/Linux
CPU	ARM A/M Series (or similar)
Memory	Internal or external
Clock Frequency	200-800MHz
Typical nodes	22-40nm

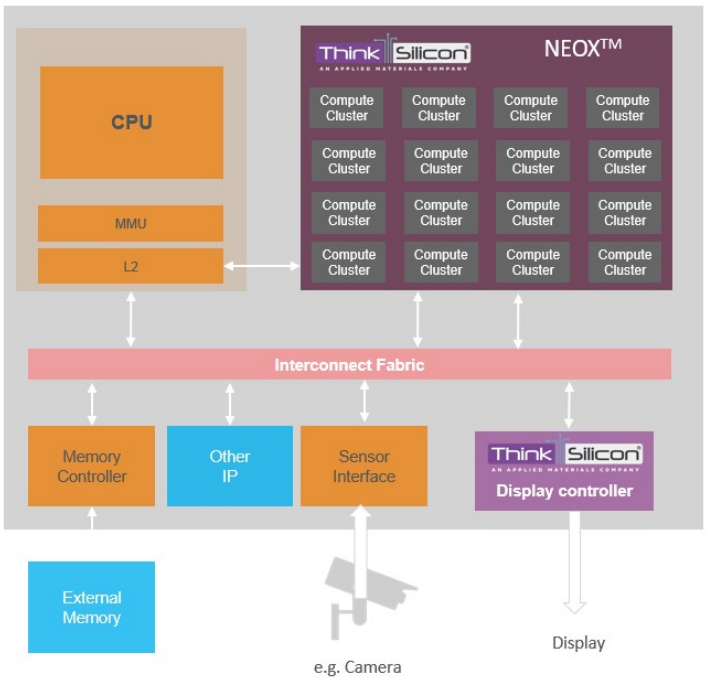


Figure 2.3: Overview of a system using the NEOX-V accelerator [17]

2.2.2. Architecture

NEOX-V is architected as a highly configurable IP core with custom instructions, configurable number of cores, configurable memory system, thread count and application aware thread scheduling policies. NEOX-V includes AI-specific ISA extensions, SIMD Vector in variable length data types including 8-bit and optionally Graphics ISA Extensions/Co-processors: Unified Shader Architecture, Tile Based Rendering, Color/Vertex, Vector Support and contains dedicated hardware modules, such as rasterizer, texture unit, tile management unit and texture caches. The architecture can be seen in Figure 2.4.

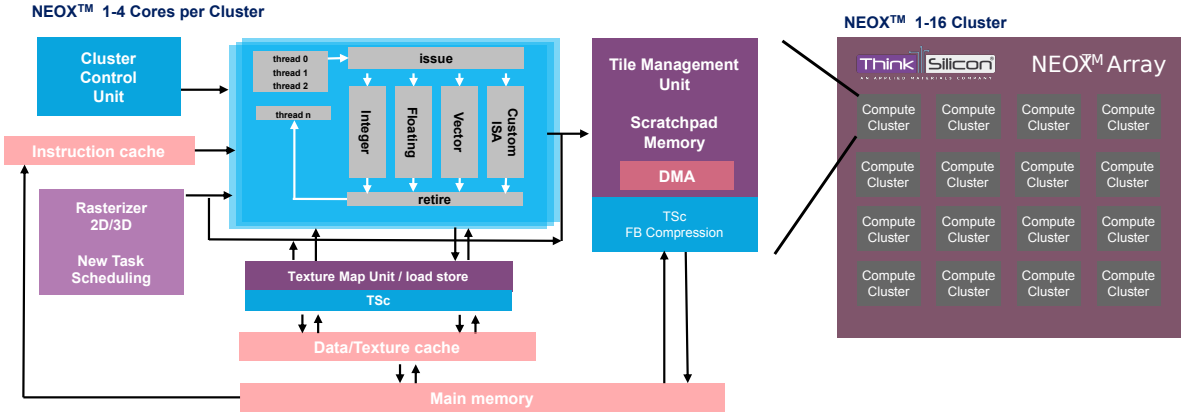


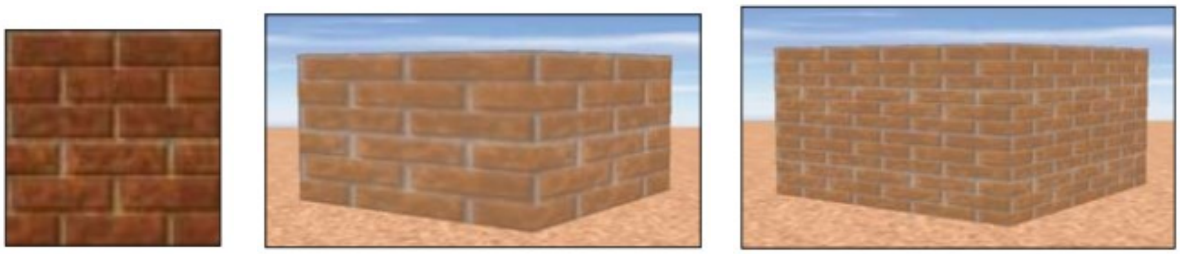
Figure 2.4: Architecture of the NEOX-V accelerator [17] (Modified)

As illustrated in Figure 2.4, the Neox Array is composed of multiple clusters, with each capable of supporting 1-4 cores. In addition to the multiple cores, the design can also accommodate multiple Texture Map Units. The main memory is unique to the entire array, while each cluster has its own Cluster Control Unit. The remaining components are shared among the cores and Texture Map Units. The components presented in purple color are intended for Graphics (GFX) use, while the blue ones are used for AI applications. It is important to note that while the GFX components are explained for clarity in the analysis that follows, they will not be further utilized in this thesis. Instead, our primary

focus will be on targeting AI applications within the NEOX-V architecture.

Figure 2.4 depicts the following components of the NEOX-V array:

- Core(s): The cores are responsible for issuing a new instruction, executing it and retiring the instruction. Depending on the number of generated threads, multiple threads can execute the same instruction on different data, improving thus performance.
- Cluster Control Unit: The Cluster Control Unit is responsible for coordinating the operation of the multiple cores within the cluster. This includes synchronization and communication between the cores, proper scheduling of the tasks and management of memory operations as well as error handling.
- Memory: The memory components of NEOX-V include a Main Memory, an Instruction Cache (ICache), Data Cache (DCache) and a Scratchpad memory (SPRAM). Scratchpad, which is part of the Tile Management Unit, is an on-chip high-speed memory directly connected to the CPU core that is used to temporarily store data for rapid retrieval. It provides an alternative to the Data Cache and is generally used to simplify caching logic and ensure smooth functioning of a unit in a system with multiple cores, particularly in embedded systems, by avoiding main memory contention [18]. The inclusion of a scratchpad in the Tile Management Unit offers advantages for implementing DMA. This allows for the efficient transfer of data between the main memory and scratchpad by bypassing the processing elements of the Tile Management Unit.
- Tile Management Unit: In computer graphics, the Tile Management Unit is responsible for performing a method called tiling, which is used to apply specific kind of texture to surfaces. If that texture imitates a material with a consistent look and no obvious points of discontinuity like for example sand, asphalt, or bricks, the texture image is replicated as required to cover the surface. In such cases, the texture is usually a small sample image, either generated or photographed, of the material itself, which has been designed in such way to guarantee a flawless fit between adjoining tiles. An example is being illustrated in the Figure 2.5. It is apparent that applying the brick pattern (2.5a) to each face of a rectangular prism without tiling creates an adequate image (2.5b). However, by using tiling the perceived number of brick rows can be increased, resulting in an image (2.5c) that more accurately represents a tall fortress [19].
- Rasterizer: In computer graphics, the rasterizer is used to convert an image from vector graphics format into a raster image [19] as seen in Figure 2.6. Since the rasterizer is responsible for generating the new image's pixels, it is also responsible for spawning the threads which will work on smaller parts of the original image to create the raster image. Hence, the generation of new threads and the workload balancing takes place in the rasterizer component.
- Texture Map Unit: The Texture Map Unit is a component in a GPU that applies textures to 3D objects [19]. The purpose of texture mapping is to add detail and complexity to the surface of an object by overlaying a 2D image (the texture) onto its 3D geometry as seen in Figure 2.7.



(a) Square image of a brick pattern

(b) Result of stretching one copy of the brick texture onto each wall.

(c) Result of tiling multiple copies of the brick texture onto each wall

Figure 2.5: Applying tiling to a wall surface to represent a tall brick fortress [19].

The configuration of NEOX-V considered for this thesis project is illustrated in Figure 2.8. It encompasses several components, including an execute stage with multithreaded pipeline, a register file, a decode unit, a thread scheduler, an instruction cache, a data cache, and a scratchpad. There are separate pipelines dedicated to handling different instruction types including vector, floating-point (FP), integer, branch, or load/store instructions.

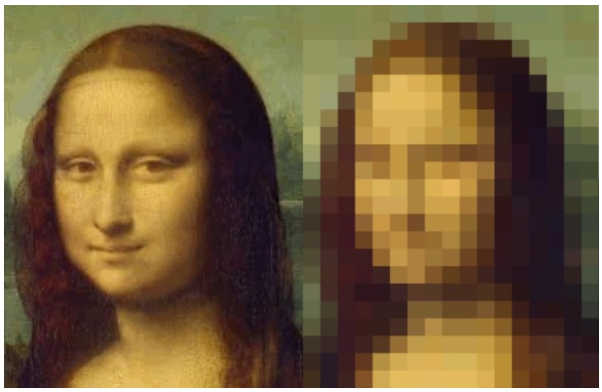


Figure 2.6: Rasterization of the famous painting of Mona Lisa. On the left is the original painting (vector image) and on the left is the rasterized image [20]

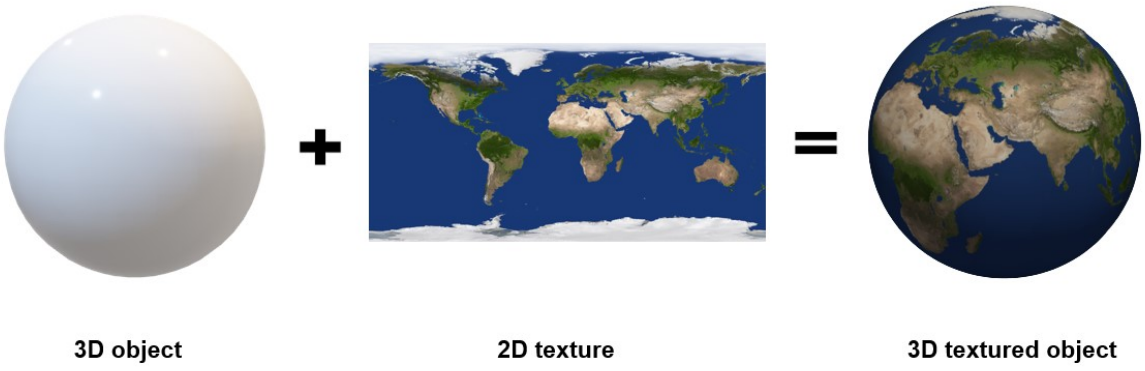


Figure 2.7: Texture Mapping: the process of applying a texture onto an object.

NEOX-V employs an advanced multithreaded pipeline to streamline its operations. The thread scheduler monitors the status of each thread, enabling it to effectively issue instructions from a pool of threads that are ready to execute. One notable advantage of this pipeline design is its lightweight nature, eliminating the inclusion of complex components such as hazards, interlocks, branch prediction units, or feedback paths.

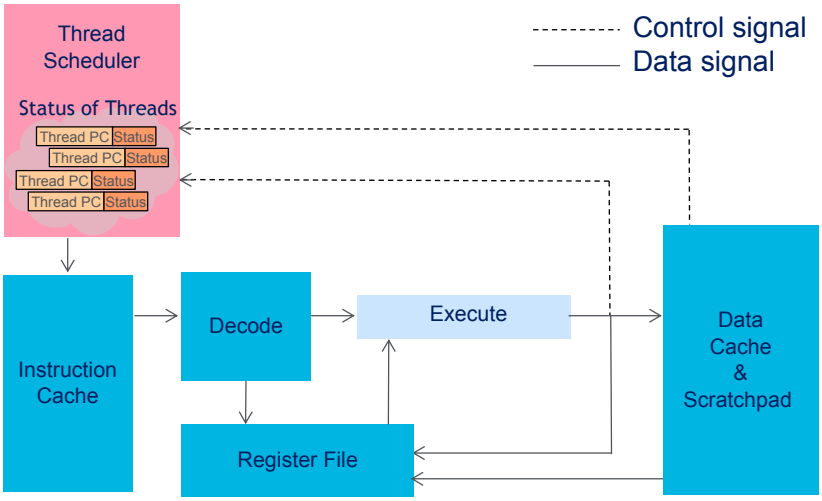


Figure 2.8: The architecture of NEOX-V considered in this thesis.

Multithreading is the concurrent execution of multiple threads that share system resources such as the CPU, memory, and I/O devices. It provides improved performance by effectively utilizing available resources and exploiting parallelism in tasks that can be executed simultaneously. The thread scheduler, based on scheduling policies like round-robin, FIFO, or others, determines the order and timing of thread execution. Figure 2.9 illustrates an example of a system utilizing four threads to leverage parallelism. The specific ordering of the threads depends on various factors, including the scheduling policy, thread dependencies, resource availability, and external events. For instance, a thread may need to wait for data to become available or encounter a resource contention issue that affects its execution order.

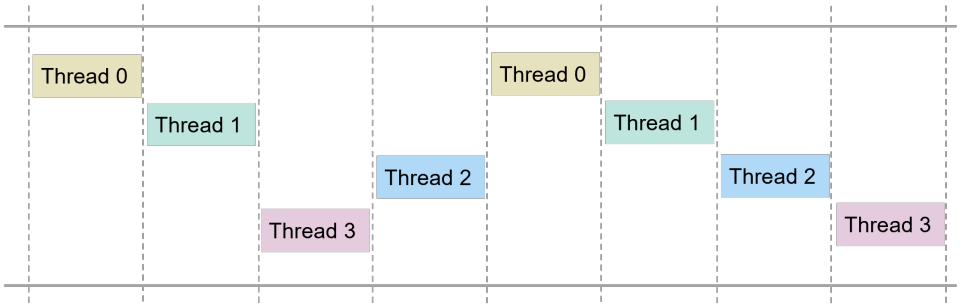


Figure 2.9: Multithreading using 4 different threads.

The integration of multithreading within NEOX-V brings forth a host of benefits, with a primary focus on maximizing efficiency by optimizing resource utilization. Particularly in memory-intensive applications, the utilization of multithreading ensures that the pipeline operates at its peak performance, achieving a compute utilization rate of up to 100%.

2.2.3. NEOX-V as a Product

The current implementation of the GPU includes the RTL code (Verilog), an SDK for graphics and AI applications as well as a high-level functional simulator in C/C++, used to devise informed architectural decisions and end up with a balanced design in terms of performance, power and area. This simulation

is currently limited to the functional level, which means that it does not offer any timing information regarding the execution at either the core or memory system level.

The current release of the product contains 1 core and 32 threads. However, the company plans to build a multi-core and a multi-cluster version of NEOX-V with an optimized memory hierarchy (consisting of both scratchpad and cache memories) exploring at the same time various ISA level extensions as well as thread scheduling and code parallelization techniques.

2.2.4. Challenges of NEOX-V simulation

Developing a timing simulator for an embedded hybrid GPU-AI accelerator, such as NEOX-V, is far from a trivial task due to several key factors. The system's inherent complexity requires modeling and synchronizing multiple computational and memory components. Furthermore, the highly multi-threaded architecture of Neox-V introduces additional challenges as multiple threads execute on the same pipeline in a pseudo round-robin fashion. The timing model must accurately consider this concurrency and scheduling mechanism. Building a timing simulator with a balanced trade-off between accuracy and execution time is another critical aspect of the simulation process. Achieving a small error percentage between the timing model and the RTL equivalent implementation is far from a trivial task, especially when it is crucial to maintain low execution times for the system. Finally, the task is compounded by the dynamic nature of the accelerator's development. The ongoing iterations and bug fixes in the RTL code pose challenges that require adaptability and flexibility in the simulation process. Analyzing and incorporating these changes into the simulation framework is crucial to providing accurate timing predictions and enabling efficient debugging and optimization.

2.3. Conclusion

Engineers and researchers have been continually developing various state-of-the-art techniques to effectively model a GPU's functionality and predict its execution time. These techniques range from simple and fast simulators to heavyweight cycle-accurate ones or even machine learning methods. SimpleScalar [7] is a simple but powerful architecture simulator that models the microarchitecture of a processor while also providing timing information. It exhibits certain limitations, though, such as limited ISA support, flexibility, or parallelism due to its simplistic nature. On the other hand, Gem5 [8, 9] is a complex cycle-accurate simulator that offers detailed modeling and exploration of various configurations. Despite its powerful nature, its complexity makes it a heavyweight simulator with a long execution time. When execution time is prioritized, its accuracy decreases. Often, analytical models are preferred over architectural simulators to perform GPU modeling. A mechanistic performance model [14] proposes that effective modeling can be achieved if the execution time is split into intervals disrupted by miss events. It focuses on modeling the events that degrade a GPU's performance. On the contrary, a simple BSP-based model [15] splits the execution time into two phases: communication and computation. It then uses mathematical models to predict the execution time based on events that happen during these two phases.

While these techniques are commonly employed, it is important to acknowledge their limitations. Some of them have a theoretical and generic nature, disregarding the complexities of complex memory systems, multi-threaded or multi-core architectures, and ISA-specific configurations. On the other hand, the more detailed ones can be overwhelming, requiring significant effort and expertise to effectively utilize the tool. Additionally, their complexity introduces significant computational overhead and simulation runtime, impacting time-sensitive projects or situations that demand faster iterations.

Considering these limitations of the existing simulators, NEOX-V stands out as a promising option for simulation due to its unique features and capabilities. NEOX-V is an ultra-low-power RISC-V based GPU processor optimized for both GPGPU and AI workloads, which serves as a case study for the current thesis. It is multi-core and multi-threaded, with a configurable number of cores serving a wide range between power, performance, and functional integration with different levels of the SoC platform. Multithreading hides long latency delays from the external memory controller, maintaining high computation throughput for the entire system. NEOX-V caters to a range of domains, including AI, IoT/Edge, and performance media processing, targeting both consumer and industrial markets.

Its architecture incorporates AI-specific ISA extensions and SIMD vectors of varying lengths, including 8-bit data types. Additionally, it optionally includes Graphics ISA Extensions for Unified Shader Ar-

chitecture, Tile Based Rendering, Color/Vertex processing, and Vector Support. Moreover, the system comprises dedicated hardware modules such as a rasterizer, texture unit, tile management unit, and texture caches. The current implementation of NEOX-V includes the RTL code, the SDK for graphics and AI applications, and a high-level functional simulator in C/C++. At its current state, this simulation is constrained to the functional level, implying that it does not provide timing information pertaining to the program execution. As a result, NEOX-V offers an ideal opportunity for exploring how we can incorporate timing characteristics into a functional simulator to achieve a balanced trade-off between accuracy and execution time.

However, building a timing simulator for NEOX-V is a challenging task. Some of these challenges arise from its highly multi-threaded architecture and the need to synchronize multiple computational and memory components. Moreover, the requirement to strike a balanced trade-off between accuracy and execution time adds another layer of complexity. Achieving a small error percentage between the timing model and RTL implementation is difficult, given the importance of maintaining low execution times. Additionally, the dynamic nature of the accelerator's development requires adaptability and flexibility in the simulation process to incorporate ongoing iterations and bug fixes.

After delving into the relevant literature and foundational knowledge required to understand the research question, the focus now shifts to the implementation details. The subsequent chapter will demonstrate how these theoretical concepts are translated into a tangible implementation, overcoming the challenges presented. This will allow us to further advance towards building an event-driven timing simulator with high accuracy and low execution time.

3

Implementation

In the preceding chapter, we provided an extensive exploration of the necessary background information and related work to establish a solid foundation for the subsequent discussions. This included considering the relevant literature on architecture simulators, analytical models, and even ML techniques, as well as understanding the NEOX-V specific details. With the required background knowledge established, we can now delve into the implementation details that will answer the research question.

This chapter outlines the process of transforming the theoretical concepts and design specifications of developing a timing simulator into a functional software tool. Section 3.1 presents the requirements of the timing simulator, while Section 3.2 includes the definition of some keywords for better comprehension of the implementation details that follow. Furthermore, Section 3.3 presents the high-level overview of the timing simulator and its interface with the rest of the system, while Section 3.4 illustrates its internal structure. Section 3.5 describes the implementation of the scheduler to facilitate the timing simulator. Finally, Section 3.6 presents the importance of using the timing model not only for simulation purposes but also as feedback to identify any weak points of the existing design. By delving into the details of the implementation, this chapter aims to provide a comprehensive understanding of the simulator's construction, paving the way for evaluating its performance and effectiveness in subsequent chapters.

3.1. Requirements

Having well-defined requirements for a simulator is crucial for its successful development, deployment, and acceptance by users. The following requirements enable a systematic and structured approach to building our timing simulator:

- Accuracy: The relative error of the clock cycle estimation should be smaller than 20% compared to the RTL equivalent.
- Overhead: The execution time of the simulation should not increase more than 10%.
- Flexibility: The timing simulator should be easily configured with different parameters such as cache configuration, latencies etc.
- Extensibility: The timing simulator should be implemented in a modular way to allow easy addition of new features.
- User activation: The timing simulator should be easily enabled or disabled without interrupting the overall functionality of the system.

3.2. Dictionary

As the following sections will encompass technical information regarding the implementation of the simulator, we provide a dictionary at this point to define essential terms and concepts for better comprehension.

- **Instruction:** A basic unit of operation that a processing unit can understand and execute.

- **Kernel Function or Kernel Code:** A function that is specifically designed to be executed on GPU cores. A kernel function consists of multiple instructions optimized for GPU computing, facilitating parallel processing to accelerate specific computational tasks.
- **Pipeline:** A technique used to break down the execution of instructions into a series of sequential stages, with each stage performing a specific operation on the instruction. In this way, multiple instructions are overlapped during execution. The number of instructions that can be executed in parallel in a pipeline depends on the number of stages - also called pipeline length.
- **Thread:** The smallest sequence of programmed instructions that can be managed independently by a scheduler. In NEOX-V, each thread takes one instance of the kernel function (multiple instructions) and executes them on different data. It is similar to the CUDA or OpenCL programming model, but the difference is that each thread of NEOX-V has its own program counter. As a result, each thread can run independently of other threads, meaning that each thread can run a different instruction without the need for thread synchronization.
- **Event:** Any identifiable occurrence that has significance for the system simulation. In event-driven programming, such events can be user input or message passing. For the timing simulator of NEOX-V, the execution of an instruction is split into different events which can be core, cache, or other events. Each type of event targets the modeling of a different component of NEOX-V. More information is provided in Section 3.4.1.
- **Thread Ready Status (TRS):** This status indicates the clock cycle in which a thread will have completed executing an instruction and is available to run a new one. Being available does not mean that it will be chosen by the thread scheduler to execute a new instruction. It only means that it has finished execution. More information is provided in Section 3.4.2.
- **Cache flush:** It is the process of removing the contents of the cache memory, after writing back to the main memory. Cache flushes are commonly used when there is a need to free up space in the cache.
- **Cache invalidate:** It is an operation that marks the cached data as outdated without writing it back to the main memory. When data is modified in main memory, cache invalidation is used to mark the corresponding cached copy as invalid. The next time the processor tries to access the invalidated data, it will be forced to fetch the latest data from the main memory instead of using the outdated cached copy.

3.3. High-level overview

Prior to delving into the intricacies of the timing simulator, it is imperative to establish its high-level overview for a comprehensive understanding of its position within the system architecture. This includes elucidating the inputs and outputs, the interconnections with other system components, and the specific information dependencies it has on other integral elements of the system. Only with this foundational comprehension can we proceed to investigate the internal structure of our timing simulator.

Figure 3.1 depicts the high-level overview of the simulator including the functional and the timing part. The figure shows the parameters used to configure the two simulators and the inputs and outputs for each simulator.

3.3.1. Simulation Parameters

The following parameters are used to configure the simulators and can be different for each simulation:

- **Number of cores**: The number of cores for the NEOX-V simulation
- **Number of threads**: The number of threads that run on each core of NEOX-V
- **Memory configuration**: Sizes for Instruction and Data caches, Scratchpad, and DMA
- **Instruction latencies**: The clock cycles required for the different parts of an instruction execution as derived from the RTL (Register-Transfer Level) design after running multiple benchmarks. The different latencies include:
  - pipeline length: the clock cycles needed for an instruction to traverse the pipeline
  - load instruction: the extra clock cycles a load instruction requires
  - store instruction: the extra clock cycles a store instruction requires

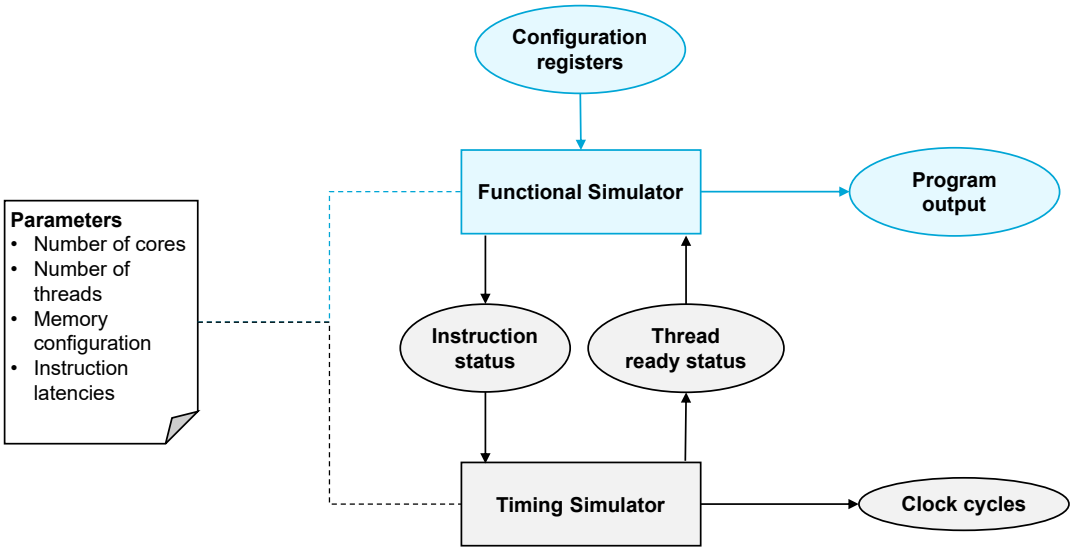


Figure 3.1: Block diagram with a high-level system overview of the timing model. The timing model takes as input the instruction status and outputs the clock cycles. It also gives as input to the functional simulator the ready status of each thread. The light blue components indicate correlation with the functional simulator, while the gray ones indicate correlation with the timing simulator.

  - hit cache access: the extra clock cycles a hit cache access requires
  - miss cache access: the extra clock cycles a miss cache access requires

Section 3.4 contains further information about how these latencies are used.

3.3.2. Functional Simulator

The functional simulator is already part of the NEOX-V product and serves as the interface with the proposed timing simulator. Responsible for modeling the functionality of the system, the functional simulator does not account for any timing characteristics. It has the following interface with the rest of the system:

- **Inputs:** configuration registers, thread ready status
- **Outputs:** program output, instruction status

It takes as input the configuration registers, which include information like the kernel function (the pointer to the first instruction of the program to be executed), the arguments of the kernel function, and the number of threads to run this kernel. Using this information, the functional simulator models the functionality of NEOX-V. It then gives the program output, which includes the traces from the program execution (print messages), statistics (number of instructions, cache hit/miss rates etc.) and register or memory values.

The interaction between the functional simulator and the timing simulator occurs through the instruction status and the thread ready status, both of which are explained below.

3.3.3. Timing Simulator

The timing simulator is the proposal of this thesis and is added as an extension to the functional simulator to model the execution time of a program. It has the following interface with the rest of the system:

- **Input:** instruction status
- **Outputs:** thread ready status, clock cycles

Once the functional simulator executes an instruction, it provides the timing simulator with the instruction status, which includes the following information about a kernel's instruction:

- Thread ID: The thread used to execute an instruction
- Instruction cache access: If fetching this instruction caused an instruction cache miss or hit and in which address. Also, if the type of instruction was a load, flush, or invalidate.
- Data cache access: If this instruction caused a data cache access and in which address. Also, if this access was miss or hit and if the type of instruction was a load, store, flush, or invalidate.
- Scratchpad: If this instruction accessed the scratchpad memory.
- DMA: If this instruction needed DMA and relevant information such as the DMA access size.

With this information, the timing simulator calculates the clock cycle at which the thread executing this instruction will complete its execution. This is the thread ready status which is then passed back to the functional simulator as input. The thread scheduler, which is a component of the functional simulator, relies on this information to manage the execution of multiple threads efficiently. This is further explained in 3.5.

The iterative process of the functional simulator providing the instruction status to the timing simulator and the timing simulator providing the thread ready status to the functional simulator continues until there are no more kernel instructions left to process. At that point, the timing simulator outputs the total clock cycles taken to complete the entire simulation. Figure 3.1 illustrates the repetitive cycle of operation between the functional and timing parts of the simulation through the instruction status and the thread ready status exchange.

Despite that the timing simulator gives input to the functional simulator, the timing simulator can be selectively enabled or disabled as required. This means that it does not directly impact the time required for each instruction to traverse the pipeline. Instead, it assumes that instructions move through the pipeline instantaneously, without any inherent time delay, similar to what the mechanistic model suggests [14]. The only dependence of the functional simulator on the timing simulator is through the thread ready status, which can be disabled if needed, affecting only the scheduling. There is more information provided in section 3.5 about how this is achieved.

3.4. Internal structure

Having meticulously examined the high-level overview of the timing simulator and its intricate connections with the functional simulator, along with a detailed understanding of their inputs, outputs, and parameters, we can now delve into the implementation details. This section analyzes the internal structure of the proposed timing simulator. The timing simulator primarily focuses on calculating the execution time of a program using event-based modeling. By estimating the clock cycles needed for execution, it provides a detailed assessment of the execution time without explicitly considering the individual traversal time for each instruction within the pipeline. Next, we will see how this is achieved.

Figure 3.2 depicts the timing simulator's internal structure in detail. The main components are an event generator, which creates the different events, and an event manager, which handles these events using two queues.

3.4.1. Event generator

The modeling of the various NEOX-V components is accomplished using these different types of events. As a result, a direct one-to-one correspondence exists between the main NEOX-V components and the events of the timing simulator. The event generator analyzes the input of the timing model, which is the current instruction status. This input is given in the form of a structure. Upon analyzing the given information, the event generator generates distinct types of events depending on the NEOX-V components utilized by the instruction:

- Core event
- Instruction cache event
- Data cache event
- Scratchpad event
- DMA event

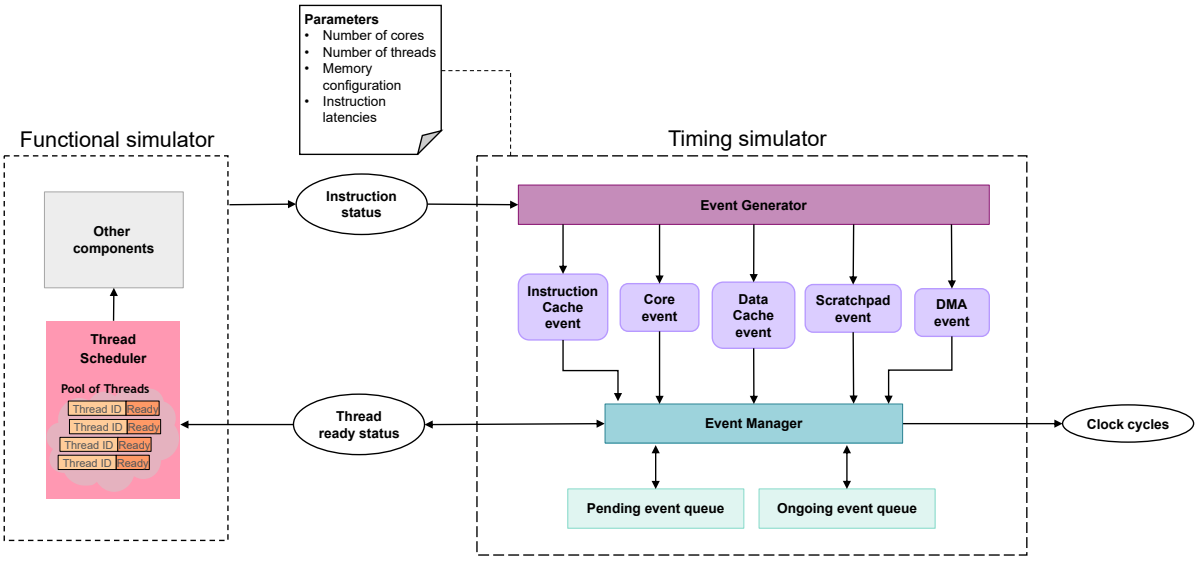


Figure 3.2: The internal structure of the timing simulator and its interface with the functional simulator. Different colors indicate different components, while the same colors are used for components with similar functionality. The main entity of the timing simulator are the events, depicted with rounded corners, while all the square components are responsible for handling these events.

As their name implies, the different types of events target a specific component of NEOX-V. For example, the core events represent the part of the instruction that utilizes the core, while the instruction cache events represent the part of the instruction that utilizes the instruction cache, etc. It is evident based on the simulated instruction, different components are utilized. Therefore, it is not mandatory that the event generator create all these events on every iteration. Depending on the input, it is possible that a data cache event is created and not a scratchpad event. For example, if the instruction status indicates that the scratchpad was not utilized, the scratchpad event is not created.

Core event

This event is related to the operation of the NEOX-V core(s). At the moment, the NEOX-V functional simulator supports only one core, so the timing simulator also has the same configuration. The core event contains two primary pieces of information:

- **thread ID**: the thread used to execute this event. This is extracted from the input of the simulator.
- **delay**: the clock cycles this event needs for completion. This is equal to the pipeline length, which is given as a parameter to the timing simulator.

Instruction cache event

This event represents the instruction cache access needed to fetch an instruction. It includes the following information:

- **thread ID**: the thread used to execute this event. This is extracted from the input of the simulator.
- **access type**: if the access to the instruction cache is a load, an invalidation, or a flush
- **cache hit/miss**: if the access is a hit or miss
- **address**: which address is requested
- **delay**: the clock cycles this event needs for completion.

The first three pieces of information can be easily extracted from the instruction status given as input to the timing simulator. The delay is calculated by adding the latency of the access type to the latency of the cache hit or miss, as provided by the parameters. For example, if the access type is a load with a cache hit, the delay will be different from a flush or a cache miss.

Data cache event

Similar information as in the instruction cache applies here. The only difference is that the cache access type can also be a store. Therefore, the following information is provided:

- **thread ID**: the thread used to execute this event. This is extracted from the input of the simulator.
- **access type**: if the access to the instruction cache is a load, a store, an invalidation, or a flush
- **cache hit/miss**: if the access is a hit or miss
- **address**: which address is requested
- **delay**: the clock cycles this event needs for completion.

Scratchpad event

The scratchpad is a small, fast memory that is local to the Tile Management Unit (TMU). It provides a fast, low-latency memory that the processing elements -that have access to it- can use to store data that is frequently accessed or modified during computations.

Accessing the scratchpad has a constant cost, which is significantly lower compared to accessing other memory components. Therefore, this event has an attribute called **delay** which equals the clock cycles required for using the scratchpad.

DMA event

Having a scratchpad in the Tile Management Unit makes it beneficial to implement DMA, which enables the transfer of data between the main memory and scratchpad by bypassing the processing elements of the Tile Management Unit.

A DMA event has an attribute called **delay** to represent the cost of this transfer. To calculate the cost of DMA, several factors need to be taken into account:

- DMA access size: The total amount of data that needs to be transferred by the DMA. This is typically measured in bytes.
- DMA burst size: The amount of data transferred by the DMA in a single burst. This is typically measured in bytes.
- Transfer latency: The cost of performing a single DMA write or read transfer due to latency. This cost is measured in clock cycles (cc).
- DMA write acknowledge time: In case the DMA operation is a write back to the main memory, the system needs 1 extra cycle to acknowledge that the writeback is complete.
- DMA bus width: The width of the bus used by the DMA to transfer data. This is usually measured in bits.
- Cost of AXI: The cost of accessing the AXI bus, which is used by the DMA to transfer data between the main memory and the scratchpad.

The DMA access size is extracted from the input of the timing simulator while the rest information is given by the parameters used to configure the timing simulator. Taking the above factors into consideration the cost of DMA can be calculated as follows:

1. Determine the number of bursts required to transfer the entire DMA access size:

$$\text{Number of bursts} = \frac{\text{DMA access size}}{\text{DMA burst size}} \tag{3.1}$$

2. Calculate the DMA transfer time for a single burst:

$$\text{DMA burst transfer} = \frac{\text{DMA burst size}}{\text{DMA bus width}} \tag{3.2}$$

3. Calculate the total cost of transferring all bursts:

$$\text{Total cost} = \text{Number of bursts} * \text{DMA burst transfer} + \text{Transfer latency} \tag{3.3}$$

   The total cost is measured in clock cycles (cc). The transfer latency is only added to the total cost for the first transfer, since subsequent transfers are pipelined, meaning that the transfers occur in a continuous and overlapping sequence. Therefore, the latency for each transfer, is hidden and does not add to the total cost. The pipelining of requests allows multiple transfers to occur concurrently, which reduces the overall latency and improves the efficiency of the transfer.

4. Add the cost of accessing the AXI bus to the total cost.
5. Add the DMA write acknowledge time to the total cost.

This calculation provides the total cost in clock cycles required to perform the DMA transfer, and the result is saved to the **delay** attribute.

3.4.2. Event manager

Once the events are created, they are provided as input to the event manager. The event manager is responsible for 1) updating the thread ready status and 2) calculating the clock cycles.

Clock Cycles (cc)

The Clock Cycles (cc) are used to keep track of the progress of the simulation. It represents how many clock cycles have passed since the start of the simulation. At the first iteration of the simulation, it is equal to 0. As the simulation progresses, it is updated using the thread ready status, as explained below.

Thread Ready Status (TRS)

The Thread Ready Status (TRS) represents the clock cycle in which each thread will have completed executing an instruction. This information is stored in a vector of size equal to the number of threads. So, thread_ready_status[0] represents the clock cycle in which thread 0 will be ready, while thread_ready_status[1] when thread 1 will be ready, etc. In each iteration of the simulation, the event manager updates the thread ready status of the thread ID given as part of the timing simulator input.

To update the thread ready status, the event manager considers the delays from all the events. This is calculated by adding the delays of all the events to the current value of the clock cycle. The current value of the clock cycle is considered to be the request cycle, which is the cycle in which the instruction started to be executed. The formula is the following:

$$\begin{aligned} \text{thread\_ready\_status} = \text{clock\_cycles} \\ + \text{core\_event.delay} + \text{icache\_event.delay} \\ + \text{dcache\_event.delay} + \text{scratchpad\_event.delay} \\ + \text{dma\_event.delay} \end{aligned} \tag{3.4}$$

If an event is not present in the current instruction, then it is simply not included in the formula. For example, if this instruction utilized scratchpad and not data cache, then the data cache event is not created and therefore it is emitted from this calculation.

Once the value of the thread ready field is updated, the event manager also updates the clock cycles with the same value. This process is repeated for each thread that executes an instruction. Hence, each time a thread starts processing an instruction, the relevant information about the instruction status is given as input to the timing simulator. The timing simulator calculates the thread ready status of the current thread and updates the clock cycles accordingly. In the last iteration, the clock cycles will represent the clock cycle in which the last thread of the simulation completes the execution of its instruction, denoting the end of program execution. This final result is the output of the timing simulator.

Pending & Ongoing queues

It is important to note that cache events require special treatment. The event manager utilizes two queues to handle events that involve blocking components of NEOX-V. By ”blocking,” we refer to any component capable of handling only one event at a time. Caches in NEOX-V are examples of such components since they cannot simultaneously serve two or more accesses.

- Ongoing Event Queue (OEQ): An event that is currently served by the system enters this queue. Once the event is done with its execution, it is removed from this queue.
- Pending Event Queue (PEQ): All events that cannot be executed instantly enter this queue. These can be instruction cache events or data cache events that cannot be executed because the corresponding cache is busy.

With these two queues, the serving of events is managed as follows. Once a cache event is present, it tries to enter the OEQ. If this queue is not full, it enters the queue and is considered currently being served. In that case, the **delay** variable is not modified. The event exits the OEQ - so, it is considered as served- when the value of clock cycles is larger than the thread ready status of the thread used to serve this event. This means that the time has passed and the thread serving this event is done with its execution since its thread ready status is larger than the current clock cycle. If the OEQ is full, the event is placed at the back of the PEQ. This means that it will have to wait until all the previously arrived events are served. The **delay** value is based on the delay inherited by all its preceding events. Finally, when the event is the first in the queue and when the OEQ is empty, it enters the OEQ and it is considered currently being served.

To further clarify how the event manager handles the cache events, Figure 3.3 shows two flowcharts, one for managing the queues and one for managing a new cache event. Each time the event manager gets a new cache event, it first updates the queues (Figure 3.3a) and then handles the event (Figure 3.3).

Figure 3.3a) shows how the queues are being updated. The event manager constantly checks if the OEQ is full. If the OEQ is not full, the event manager inserts the first event from the PEQ to the OEQ and repeats this process. If the OEQ is full, it checks the next event (in the first iteration this is the first event of OEQ) to see if the current value of the clock cycle is larger than the TSR of the thread serving this event. If the clock cycle is larger, then the event manager removed the event from OEQ, since it is served, and continues checking the OEQ. If the clock cycle is not larger, then this event is not yet served and the check ends.

Figure 3.3b) shows how the event manager handles a new cache event. It first checks whether the OEQ is full. If it is full, the cache event is inserted into the PEQ and its delay is increased by the delay of the last event that entered the PEQ. This means that if there are any preceding events, they have to be served first inducing delay to the next cache events. If the OEQ is not full, the cache event is inserted into the OEQ. Then the TRS of the thread serving this cache event is updated according to the formula 3.4. The clock cycles are given the same value to represent the time needed by the thread to serve the events of this instruction.

Currently, these two queues are utilized to handle the cache events, but they can be extended in the future to accommodate any other event that exhibits blocking behavior. Moreover, their size is adjustable, allowing for future reconfiguration to accommodate more or less events if needed.

3.5. Thread scheduler

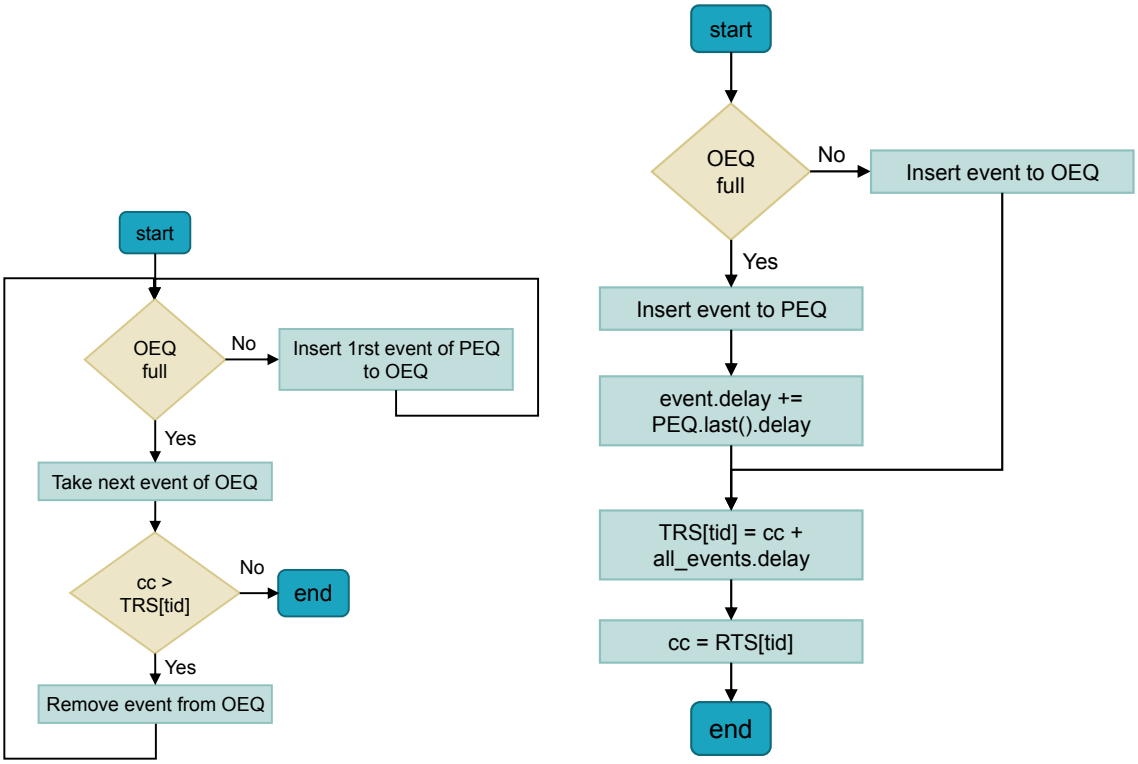
The thread scheduler manages the execution of multiple threads and is responsible for determining the order and duration of thread execution, effectively allocating the available time to different threads. It ensures fairness by providing each thread with a fair share of time, preventing any single thread from monopolizing system resources.

As depicted in Figure 3.2 the thread scheduler is part of the functional simulator. Despite that, adding a timing simulator to the system necessitates changing the thread scheduler to account for the timing information when managing the threads. For this reason, the thread scheduler is extended with four scheduling policies enabling the concurrent execution of multiple threads and facilitating efficient resource management. These are Round Robin, Grouped Round Robin with 8 threads, Grouped Round Robin with 32 threads and Minimum. The scheduling policy Round Robin is the only one that does not account for the timing information. As a result, it can be used when the timing simulator needs to be disabled allowing for meeting the user activation requirement.

3.5.1. Round Robin

The Round Robin scheduling policy ensures that threads are executed in a specific predetermined order. The order is defined based on the indices of the threads, where the first thread to be scheduled is thread 0, followed by thread 1, then thread 2, and so on. It is worth mentioning that no preemption is used, since the architecture is multithreaded and can run multiple threads in parallel. Hence, once a thread enters the pipeline it cannot be removed unless it properly goes through all the stages of the pipeline.

This scheduling policy is simple enough and is used as a first implementation which does not take



(a) Update queues

(b) Management of a new cache event

Figure 3.3: Flow charts showing how the event manager handles events and updates the queues.
tid: Thread ID, representing the unique identifier of the thread used to serve this event.
event.delay: The delay corresponding to the event being managed at each iteration.
PEQ.last().delay: The delay corresponding to the last event of PEQ.
all_events.delay: The sum of all the delays of the events. It is used to implement the formula 3.4

into account the timing of the model. That means that it does not check the TRS and schedules them in their predefined order. This behavior may lead to multiple pipeline stalls resulting in a performance degradation.

Figure 3.4 shows the different scheduling policies for the threads depending on their TRS. The array $TRS[thread\,id]$ has the time in cc in which the thread $thread\,id$ will be ready; for example, $TRS[0] = 15$ indicates that thread 0 will be ready in cycle 15 while $TRS[7] = 22$ implies that thread 7 will be ready in cycle 22 etc. The Round Robin (RR) scheduling policy will schedule the threads in the predefined order disregarding their ready field. This means that if the current cycle is 10 and because $TRS[0] = 15$, the pipeline will stall for 5 cycles, while waiting for thread 0 to become ready.

3.5.2. Grouped Round Robin (8 threads)

This scheduling policy works like Round Robin but checks the threads in groups of 8 while also considering the TRS. This means that in one cycle the scheduler is capable of checking only 8 threads to find which one to schedule. If none of the threads are ready, then in the next cycle it will check the next group of 8 threads etc.

Since NEOX-V has 32 hardware threads, this scheduling policy will check in the first cycle threads 0-7, and will try to find an available thread to execute. If none of them is ready, in the second cycle the scheduler will check threads 8-15, in the third cycle threads 16-23 and then in the fourth cycle threads 24-31. So, if in the first 3 groups of 8 threads (threads 0-23) none of the threads are ready and only in group 24-31 there are ready threads, the pipeline will have to stall for 3 cycles until the scheduler finds the ready thread.

While checking the groups of 8 threads, once the scheduler finds a ready thread it executes it without checking the rest threads of the group. This means that it is possible that other threads are ready before the chosen thread but are not executed because the scheduler never checked their TRS.

This scheduling policy can hinder the efficiency and performance of the processor but was introduced as a temporary solution for the RTL design.

As seen in Figure 3.4, the RR policy with 8 threads chooses thread 9 after stalling for one cycle. This happens because the current cycle is 10 and at this moment none of the threads 0-7 is ready. In the next cycle (cycle 11), the scheduler checks threads 8-15 and schedules the first thread that checks and is already ready (this is thread 9). It is worth mentioning that thread 11 is also ready at cycle 10, but the scheduler never checks this thread since it sent thread 9 for execution.

3.5.3. Grouped Round Robin (32 threads)

This scheduling policy works in a similar way as the previous policy but checks all the threads which are 32 in the NEOX-V design. This is an improvement of the previous scheduling policy which aims to eliminate the idle cycles introduced by the scheduler. The scheduler scans the TRS of all the threads and schedules the first that is ready.

Similar to the previous scheduling policy, once the scheduler finds a ready thread it executes it without checking the rest threads of the group. Idle cycles are not introduced in this case, but the waiting time of the threads is not optimized.

As seen in Figure 3.4, the RR policy with 32 threads chooses thread 9 since current cycle is 10 and the previous threads are not ready yet. This eliminated the stall of 1 cycle that the scheduling policy RR with 8 threads caused.

3.5.4. Minimum

The minimum scheduling policy operates based on the principle of minimizing the time between a thread becoming ready to execute and actually starting its execution. The scheduler continuously scans the TRS of each thread to identify the first thread that becomes eligible for execution.

By prioritizing the first ready thread, the minimum policy aims to minimize the time spent in the ready queue, thereby reducing overall waiting times and maximizing CPU utilization. It ensures that threads are promptly granted execution opportunities as soon as they become ready, without any delay caused by further comparisons or evaluations. It operates based on the principle of minimizing the time between a thread becoming ready to execute and actually starting its execution.

As seen in Figure 3.4, the minimum scheduling policy is not introducing any stall in the pipeline

since it checks the TRS of all threads and schedules the one that will be ready first. In that case, the scheduler chooses thread 11 whose ready field is the minimum of the TRS array. In the next cycle, thread 9 is scheduled, then thread 10 etc. This scheduling policy minimizes the time each thread waits to be executed.

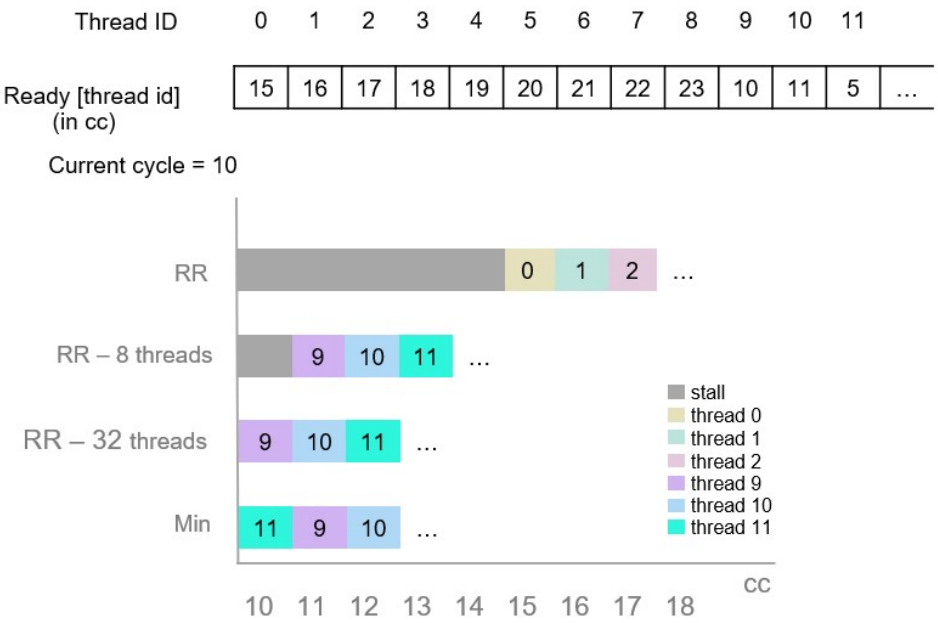


Figure 3.4: Comparison between the different scheduling policies. The RR scheduling policy is possible to introduce a lot of idle cycles. The grouped RR with 8 threads reduces the idle cycles but the one with 32 threads eliminates the stall completely. The Min policy does not introduce stalls but also minimizes the waiting time of the threads.

3.6. Using simulator as feedback

While GPU simulators usually are used to model the hardware and explore the design space and performance optimizations, they can also give valuable feedback for the current hardware implementation. This thesis uses the RTL as the golden reference to evaluate the model, however the model development was proved fruitful for the RTL development as well.

The timing model provided insights into the following aspects:

- Scheduling policy: Initially, RTL's scheduling policy utilized a grouped Round Robin approach with 8 threads. However, as the implementation of various scheduling policies for the timing model progressed, the limitations of this initial policy became apparent. This realization prompted the enhancement of the scheduling policy, increasing the number of threads from 8 to 32.
- Thread scheduler fairness: While exploring the effect of different scheduling policies on the performance, we discovered that the RTL scheduler exhibits unfairness towards one of the threads, resulting in inconsistent scheduling.
- Busy threads: While analysing the RTL and model traces, we noticed that, in some rare cases, certain threads would remain in a busy state without transitioning to an idle state. This caused the thread scheduler to overlook them and not select them for execution negatively impacting the performance.
- Cache hits: Modeling the timing of the cache network proves valuable in assessing the cache's performance. Our findings reveal that multiple store operations targeting the same cache line may experience unnecessary delays while waiting for the tag update. However, updating the tag is not required when the event is a cache hit.
- Cache misses: Another discovery revealed the presence of unnecessary stall of execution in certain cache events. Some requests in the data cache were experiencing longer delays than expected in cases of multiple data cache misses and more than 32 threads.

- New instruction: The introduction of a new instruction in the model created a different thread scheduling, resulting in longer stalls than expected when multiple instruction cache misses occurred.

The identified findings were promptly communicated to the engineers responsible for addressing such issues, and appropriate actions were taken in response.

3.7. Conclusion

This chapter presented the implementation details of the NEOX-V timing simulator using event-driven modeling. The proposed solution estimates the simulation time of a program execution in clock cycles. The timing simulator is added as an extension to the functional simulator, which models the execution of a program consisting of multiple instructions. For each iteration of the simulation, the functional simulator simulates the execution of an instruction by a thread. At the same time, it gives as input to the timing simulator the instruction status, which includes some information about the instruction that was just simulated. Using this information, the timing simulator predicts the thread ready status, which is the clock cycle in which the thread executing this instruction will be ready. With the term ”ready”, we refer to the fact that the thread has completed executing an instruction and is ”ready” to execute a new one. At the end of the simulation, the timing simulator provides as output the total clock cycles needed to execute the whole program.

This estimation of clock cycles is achieved using event-based modeling. To model each specific NEOX-V component, we utilize distinct types of events. Consequently, there is a one-to-one correspondence between the main NEOX-V components and the events within the timing simulator. In each iteration of the simulation, the execution of an instruction is split into different events. The timing simulator consists of an event generator, which creates these events using the input from the functional simulator. These events can be core, data cache, instruction cache, scratchpad, or DMA events, representing the different components the instruction had to utilize. To handle these events, there is an event manager. This event manager uses the information about the events to predict the clock cycles needed for their completion. Cache events are a special case since NEOX-V supports only blocking caches. To control these events, the event manager creates and manages two queues, one for the pending events and one for the ongoing events, called pending event queue and ongoing event queue respectively.

Moreover, thread scheduling is of great importance to developing a timing simulator with high accuracy compared to the RTL equivalent. Therefore, the thread scheduler, which is part of the functional simulator, has to account for the timing information of the threads. It is enriched with multiple scheduling policies that consider the thread ready status. These scheduling policies include a typical round robin, a grouped round robin with 8 threads, a grouped round robin with 32 threads, and a minimum scheduling policy.

Finally, it is worth mentioning that the simulator serves as valuable feedback for the RTL and functional model of NEOX-V, aiding in the identification and resolution of scheduler and cache limitations. All these components and insights collectively contribute to the successful implementation and refinement of the timing model of NEOX-V.

4

Validation Framework

In the previous chapter, we extensively discussed the implementation details of the timing simulator employing an event-driven technique. Specifically, the modeling of essential components such as the core, the scheduler, the multilevel cache network, the scratchpad memory, and the DMA was presented. We highlighted the simulator's capability not only to model the NEOX-V accelerator but also to provide valuable feedback on the existing implementation of RTL and the functional simulator. With the implementation phase now complete, this chapter's focus shifts towards validating the developed system. Chapter 5 will showcase the results obtained from this validation process.

The validation process plays a critical role in ensuring the reliability of the research findings and evaluating the extent to which the goals of the thesis were accomplished. To effectively evaluate the timing model under normal and peak loads, an extensive test suite was developed and ported to NEOX-V using the NEOX API. To test the timing of different event types, the test suite includes a wide range of micro kernel tests (Section 4.1) as well as an automatic test generator tool (Section 4.2). This tool targets the events or combinations that cannot be extensively tested by the micro kernel tests. Essentially by testing the different events of the timing simulator, we test the corresponding NEOX-V component that each event targets. Section 4.3 presents the tests developed to validate the performance of the interaction of the system's different events, while Section 4.4 presents the AI tests used as a real-life application. This test suite serves a twofold purpose: to measure a) the simulated number of cycles accuracy and b) the actual simulation time interval in milliseconds.

4.1. Micro kernel tests

The purpose of these tests is to examine the timing correctness of the different components of NEOX-V. These tests are simple and small, but they cover various cases to identify any deficiencies in the implementation of the timing model. Their name reflects their critical nature, focusing on the essential aspects of the system and providing clarity in communicating their purpose within the testing process. They include testing the following components:

- Core: These tests target the core of the GPU, by mainly utilizing core events. This means that they examine the performance of the timing model by using instructions that avoid components other than the core. Of course the instruction cache events cannot be avoided, but the focus remains to the core. These instructions can be, for example, add, multiply, jump, branch, or others [21].
- Instruction cache: Testing the instruction cache events involves evaluating the timing performance of the instruction cache in storing and retrieving instructions. These test cases deliberately access instructions that are already present in the cache and instructions that are not present in the cache. The timing model can then be evaluated to predict the execution cycles of cache hits and misses.
- Data cache: Testing of the data cache events is used to assess the timing model's ability to simulate accesses to the data cache. This includes load and store instructions that are hits and misses. It is important to take into account cases that are in the same or different cache lines, as the latencies introduced vary based on the access patterns.

- Scratchpad: To effectively test the scratchpad, these tests exclude any accesses to the instruction or data caches and focus only on accessing the data cache scratchpad by including mainly scratchpad events.
- DMA: These tests target the transfer of data between the main memory and the scratchpad by bypassing any processing elements. They include extensive use of DMA events.
- Stack accesses: Given the multithreaded architecture of NEOX-V, it becomes necessary to test the process of passing multiple arguments through the stack of each thread. This testing aims to ensure the seamless and accurate transmission of multiple arguments within the stack for concurrent execution in a multithreaded environment. These tests include extensive data cache misses, because of the stack accesses, by utilizing the data cache events.

Testing a specific component involves generating a significant number of corresponding events. For instance, when testing the data cache, there will be extensive use of data cache events. This does not imply the absence of other events, but rather that data cache events dominate these tests.

Table 4.1 includes a list of the micro kernel tests, providing details about the targeted component under testing and the corresponding size of each test. This table serves as a comprehensive reference for all the specific micro kernel tests developed during the testing process.

4.2. Test generator

To demonstrate the robustness of the results obtained, a flexible automatic test generator is introduced. This tool generates test cases that can be easily reproduced, modified, or extended, enabling rigorous and quantitative validation. Apart from the test suite, it provides the environment setup needed to execute and evaluate the tests.

4.2.1. Purpose

The purpose of the test generator is to validate the timing of the data cache. Two main reasons resulted in the development of the test generator. Firstly, due to the blocking nature of the data cache, the handling of data cache events required the implementation of two queues: the pending event queue and the ongoing event queue. Extensive testing of these queues and their control by the event manager became evident as a crucial step in ensuring their effectiveness. Secondly, throughout the implementation of the thesis, it was discovered that the execution time of the various test cases was mainly affected by the different types of accesses to memory and the number of threads employed during their execution. This trend is depicted in Figure 4.1, which shows the execution of a data cache test for different numbers of threads and hit/miss accesses. It is evident that the simulation cycles increase when the number of misses or threads increases. This is reasonable since each instruction of a test is executed as many times as the number of threads, resulting in the test size growing as the number of threads grows. Moreover, miss cache accesses require fetching data from the main memory, which is more time-consuming than reading the data immediately from the cache. Therefore, encompassing tests with varying memory access patterns and numbers of threads can be a primary source for identifying any potential bugs, errors, or deficiencies. Developing a tool that automatically generates a variety of such tests is the most reliable means of effectively stress-testing the provided solution.

4.2.2. Implementation

Algorithm 1 shows the pseudocode of the test generator. The test generator takes as input a list of integers representing the threads and a list of powers of two. Its purpose is to generate a set of tests and the necessary files to execute them. These tests target the data cache, and as a result, they should include different access patterns (different combinations of hits and misses). To achieve that, for every combination of the elements of the two lists given as input, the test generator calculates the number of hits and misses. Once we know these numbers, we can finally generate the assembly code for the test.

The assembly code includes all the instructions needed to generate a test with a predefined number of misses and hits based on the number of threads. These involve saving necessary values to registers, memory load or store operations, and basic arithmetic operations (such as mod, div, mul, and add). The basic formulas that the assembly code implements are:

Table 4.1: The list of micro kernel tests and the component targeted

Micro kernel test	***Testing component***	***Explanation***
$icache\_hits$	Core & Instruction cache	Use of the same instructions to ensure multiple instruction cache hits. No data cache is used.
$icache\_misses$	Core & Instruction cache	Branch to a different instruction based on the thread ID to achieve multiple instruction cache misses. No data cache is used.
$icache\_hits\_misses$	Core & Instruction cache	A combination of the two aforementioned test cases.
$dcache\_load\_hits$	Data cache	Instructions that involve load operations on the same data to achieve multiple hit cache accesses.
$dcache\_load\_misses$	Data cache	Instructions that involve load operations on different data to achieve multiple miss cache accesses.
$dcache\_load\_misses\_hits$	Data cache	A combination of the two aforementioned test cases.
$dcache\_store\_hits\_same\_cache\_line$	Data cache	Instructions that involve store operations on data of the same cache line to achieve hit cache accesses.
$dcache\_store\_hits\_diff\_cache\_line$	Data cache	Instructions that involve store operations on data of different cache lines to achieve hit cache accesses on different cache lines.
$dcache\_store\_misses$	Data cache	Instructions that involve store operations on different data to achieve multiple cache misses.
$scratchpad\_small\_scale$	Core & scratchpad	Simple test using the scratchpad instead of the data cache.
$scratchpad\_large\_scale$	Core & scratchpad	Extensive test using the scratchpad instead of the data cache.
dma_load	DMA & Scratchpad	DMA is used to load data from the scratchpad memory. No data cache is used.
dma_write_back	DMA & Scratchpad	DMA is used to write back data to the scratchpad memory. No data cache is used.
$stack$	Stack	Pass multiple arguments to the test's kernel function to ensure extensive stack use.

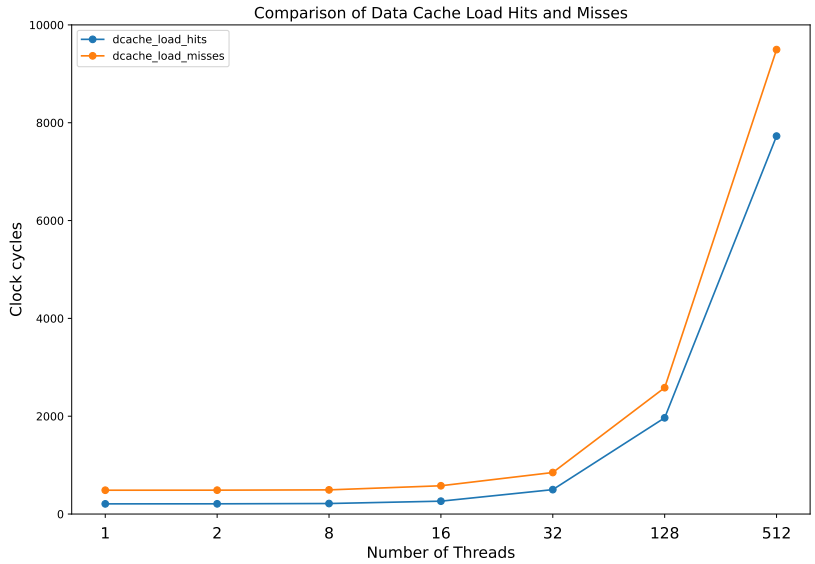


Figure 4.1: Simulation time in clock cycles of a test with multiple hit (dcache_load_hits) and miss (dcache_load_misses) data cache accesses.

$$mod\_val = thread\_id\%power \tag{4.1}$$

$$address = base\_address + mod\_val * cache\_line\_size \tag{4.2}$$

where $thread\_id$ is the thread ID currently running the test, $power$ is the power of 2 of the current iteration in Algorithm 1, mod_val is the remainder of their division, $base\_address$ is the initial value of the address, $cache\_line\_size$ is the size of the cache line, and $address$ is the final address we use to load or store a value. It becomes evident that depending on the number of threads and the power of 2, a different final address is generated, which will be used for load or store accesses to the data cache. In other words, based on the remainder of the number of threads divided by a power of 2, the test accesses a different or the same cache line, resulting in different hits and misses. For example, if we run a test with four threads, each thread will run the assembly code once. Depending on the value of $power$, the four threads will read or write to the same or different cache lines, resulting in various combinations of misses and hits. For example, if $power = 2$, we will have:

- For $thread\_id = 0$: $mod\_val = 0 => address = base\_address$
- For $thread\_id = 1$: $mod\_val = 1 => address = base\_address + 1 * cache\_line\_size$
- For $thread\_id = 2$: $mod\_val = 0 => address = base\_address$
- For $thread\_id = 3$: $mod\_val = 1 => address = base\_address + 1 * cache\_line\_size$

In this case, we have 2 cache hits and 2 cache misses since threads 0 and 2 read from the same cache line and threads 1 and 3 read from another cache line. Of course, the hit/miss rate would be different for a different value of $power$. That is the reason we include multiple values for the $power$ variable.

To further automate this process, after having created all the test cases, the test generator produces a test suite and a Makefile to execute these tests. The Makefile builds the tests, while the test suite is essentially a single file that automatically runs the tests with different configurations (read memory latencies, base addresses, or cache line sizes).

Algorithm 1: Test generator pseudocode

Input : A list of integers representing the threads: $threads\_list$ i.e. [1, 4, 32]
A list of powers of 2: $power\_2\_list$ i.e. [1, 2, 16]
Output : Generation of test code and necessary files

1: **for** each $thread$ in $thread\_list$ **do**
2: **for** each $power$ in $power\_2\_list$ **do**
3: **if** $threads >= power$ **then**
4: $misses = power$
5: **else**
6: $misses = threads\%power$
7: **end if**
8: $hits = threads - misses$
9: Generate assembly code for test $test\_t\{thread\}\_m\{misses\}\_h\{hits\}$
10: **end for**
11: **end for**
12: Generate test suite and Makefile to run the test suite

To further clarify how the test generator works, the following example is considered. We assume $thread = 4$ and $power\_2\_list = [1, 2, 16]$. Using the Algorithm 1, we calculate:

- For $power = 1$: $misses = 1, hits = 3$
- For $power = 2$: $misses = 2, hits = 2$
- For $power = 16$: $misses = 4, hits = 0$

As a result, the code generates 3 test cases:

- $test\_t4\_m1\_h3$
- $test\_t4\_m2\_h2$
- $test\_t4\_m4\_h0$

For the validation of the timing model, we use the values $thread_list = [1, 2, 4, 8, 16, 32]$ and $power\_2\_list = [1, 2, 4, 8, 16, 32]$ as input for the test generator. Table 4.2 displays the range of values for the number of threads, cache misses, and cache hits that result from this input. Since NEOX-V has 32 hardware threads, it is possible to run tests with more than 32 threads, referred to as software threads. This allows for the utilization of the 32 hardware threads multiple times, effectively creating a larger thread count. The test generator takes advantage of this capability by generating tests with up to 32 threads, specifically to explore cache performance. In each test case, it is inevitable to encounter the first cache miss. Consequently, the minimum number of cache misses observed is always 1. Considering the presence of 32 threads, the range of cache misses spans from 1 to 32, while the cache hits vary from 0 to 31. For instance, when there are 32 threads with only 1 cache miss, the remaining memory accesses result in 31 cache hits. These value ranges offer numerous combinations, all derived from the powers of 2. This enables a wide array of scenarios to be explored and analyzed in relation to cache behavior and performance.

Table 4.2: The values range of the number of threads, cache misses and hits provided by the test generator.

Test parameter	Value range
Number of threads	1-32
Number of misses	1-32
Number of hits	0-31

Summarizing, the test generator has the following characteristics:

- Parameterization: The input parameters are the list of number of threads i.e. [1, 16, 32] and the list of powers of 2, i.e. [1, 2, 4, 8, 16, 32]. Depending on these values, different cache misses and hits are generated.
- Dynamic test size: The size of the test is directly influenced by the number of threads employed during execution. In other words, increasing the number of threads results in more iterations of the test, thereby expanding its size.

- Intelligent cache simulation: The assembly code takes into account the specific characteristics of the cache simulation, such as the cache line size, to generate the required patterns.
- Integration with testing frameworks: The generated tests are integrated with the testing framework of NEOX-V to enable seamless execution of the generated tests.

4.3. System level tests

The system level tests are intended to validate the interaction of the various components and reassure their seamless cooperation. They assess the ability of the timing model to predict the execution time of tests that employ the system as a cohesive unit. They compose the basic interface that we have with the system for new additions on the timing simulator, as they encompass the fundamental memory and arithmetic operations that will be repeatedly performed.

The testing process commences with a simple memory copy test and gradually progresses in complexity, encompassing vector addition and ultimately concluding with a matrix multiplication test. This incremental approach facilitates the systematic escalation of test complexity, with matrix multiplication being recognized as the fundamental operation in AI tests. These initial tests establish the cornerstone for subsequent AI tests, giving insights into the timing characteristics of NEOX-V and thus achieving a more accurate simulation of the underlying system.

4.3.1. Copy memory region

Copying and memory management are key factors that define the performance characteristics of an underlying architecture model. DRAM, being the primary memory component of NEOX-V, requires fast access for both I/O purposes and internal organization handling. Although copying a chunk of memory from one address to another may sound trivial, it is the fundamental block for multiple memory operations. It is important to note that evaluating the performance of a simple internal copy operation provides valuable insights not only into functional correctness but also into the timing characteristics of modeling. This test involves exactly that: copying a memory region from one location of NEOX-V to another.

4.3.2. Vector addition

Shifting from purely memory operations to arithmetic operations that require fetching from memory and execution on the ALU, we develop a test that implements the fundamental operation of addition. Instead of simply adding two numbers, we opt to create two vectors and store the result of their addition in a new vector. We consider this another crucial operation that characterizes the system's behavior, and therefore, we choose to include it as a system test in our test suite. The purpose of this test is to evaluate the timing performance of a simple O(1) operation using 1-D matrices. This test serves as an intermediate step in the process of transitioning from testing memory to testing the matrix multiplication operation.

4.3.3. Matrix multiplication

Matrix multiplication (MM) is the backbone of rendering graphics [19] and machine learning [22], making this the most critical among the system level tests. As a typical application with computational and memory-intensive features, its performance is of great importance for any GPU.

MM is fundamental to the creation of visually compelling and realistic graphics. It is used to perform various operations such as transformations, projections, lighting, shading, texture mapping, and others [19].

MM is also widely used in a mathematical linear operation between matrices called convolution [23]. Figure 4.2 shows the convolution operation wherein a kernel, a small matrix of weights, traverses across the input data (shown in blue). During this traversal, the kernel conducts element-wise multiplication with the corresponding portion of the input, subsequently summing the outcomes to produce an output (shown in green). Particularly in ML applications, convolutions are used in Convolutional Neural Networks (CNNs) to perform image classification, computer vision, or natural language processing [22].

The factors mentioned above enable MM to be utilized in extracting profound insights and drawing

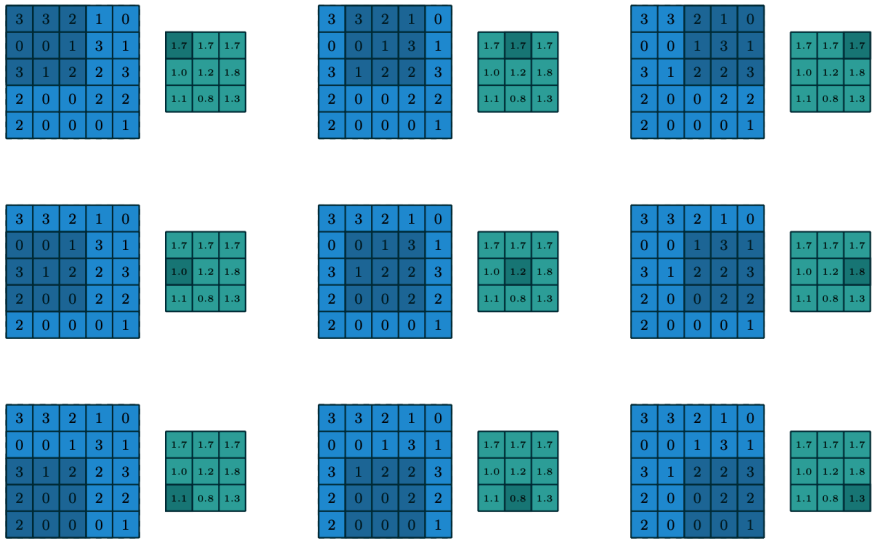


Figure 4.2: The convolution operation is essentially matrix multiplication. The input data is the matrix in blue, while the output data is the matrix in green. At each iteration, the shaded area of the output is calculated. This calculation happens as follows: a kernel traverses over the input data and performs element-wise multiplication with the corresponding portion of the input. The values of the input are in the center of the input matrix, while the values of the kernel are shown at the lower-right corner of the input matrix.[24]

significant conclusions for accelerators [25]. This test is performed with and without tiling. Tiled matrix multiplication leverages data locality and cache reuse, improving efficiency. This is done by dividing the matrices into smaller submatrices, called tiles, which fit into the cache more effectively. The data loaded into the cache for one tile can be reused in subsequent computations, improving data locality. This minimizes the need to fetch data repeatedly from main memory, reducing the frequency of accessing data from main memory.

Figure 4.3 shows the tiled matrix multiplication. M and N matrices are partitioned into 2×2 tiles, as demarcated by the bold lines. Since the output matrix P comprises 16 elements, a non-tiled matrix multiplication employs 16 threads, with each thread responsible for calculating a single output element. To perform the dot product, each thread requires complete access to one row of input matrix M and one column of input matrix N. In this example, each thread performs a total of 8 memory accesses. In the tiled version, the dot product calculations are divided into phases. Instead of accessing a full row or a full column, only parts of the rows and columns are multiplied to calculate the intermediate result of a specific element. For instance, to calculate $P_{0,0}$ the corresponding thread would have to access $M_{0,0}$ and $N_{0,0}$ and then $M_{0,2}$ and $N_{2,0}$ because at the same time another thread accesses $M_{0,1}$, $N_{1,0}$ and then $M_{0,3}$ and $N_{3,0}$. So, each thread ends up loading two elements from input M and two elements from input N, which results in a total of four accesses per thread.

It becomes evident that conducting a MM test with varying tile sizes can yield valuable insights into cache usage. Thus, to evaluate the performance of the timing model on a larger scale, this test is performed using 1×1 (not-tiled equivalent), 2×2 and 4×4 tiling.

4.4. AI tests

Considering that NEOX-V is targeted towards end-user AI applications, we deploy a suite of AI tests to ensure its robust performance in addressing challenges commonly encountered in machine learning. Anomaly detection, image classification, keyword spotting, and visual wake words set the basis for numerous AI applications and are considered the foundational building blocks for more complex algorithms in this field.

We have specifically chosen these test cases to provide extensive coverage and ensure the high-

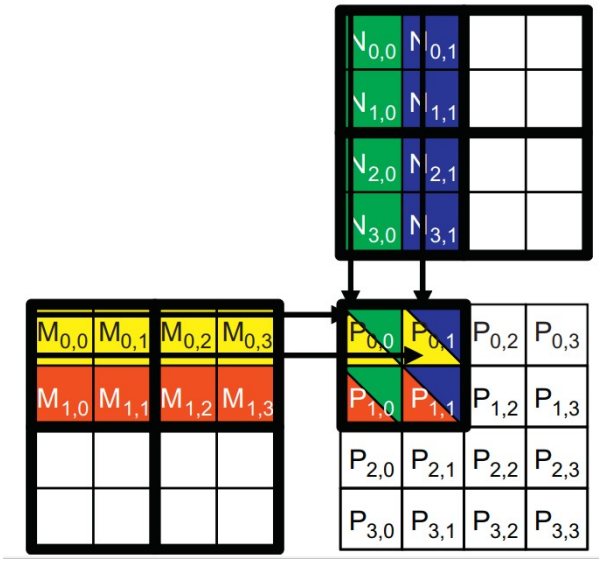


Figure 4.3: Tiled matrix multiplication [26].

quality performance of NEOX-V against these fundamental yet challenging problems. It is self evident that not only is it expected for the real system to perform successfully against these tests, but the development of the timing simulator should precede and ensure equivalent behavior. It is worth mentioning that these tests differ significantly from the system level tests, as they are attached to the performance of the end application rather than the intrinsic aspects of the hardware system.

4.4.1. Anomaly Detection

In applications that deploy pattern recognition, anomaly detection refers to the problem of systematically detecting outliers within a standard expected behavior [27], [28]. Figure 4.4 shows outliers in a 2-dimensional data set. The data set has two normal regions, N_1 and N_2, where the majority of observations are concentrated. Any points that exist far outside these regions, such as O_1 and O_2, as well as points within region O_3, are considered anomalies.

Given that AI systems make statistical observations, mispredictions are quite common. In order for the end product to sustain a high level of accuracy, anomaly detection tests are being integrated into the production cycle to isolate global, contextual, or collective outliers [29]. In the process of developing a realistic and representative simulation of NEOX-V, these tests establish a benchmark for the timing model, ensuring its quality and approximation to the behavior of the real system.

4.4.2. Image Classification

Image classification serves as a building block for various real-world applications such as object recognition, scene understanding, and visual search. As one of the extensively researched areas in computer vision [30], the techniques employed for image classification have been transitioning from classical computer vision to deep learning. In line with this shift, it is of utmost importance for an embedded device to incorporate image classification capabilities and for a timing model to precisely predict its execution time. This ability provides valuable information about the performance of the final design, enabling insights into its efficiency and effectiveness.

Specifically, image classification refers to the task of automatically labeling images into different predefined classes or categories. It involves training a machine learning model, typically a convolutional neural network (CNN), to analyze the visual features of an image and assign it to the appropriate class. Figure 4.5 shows how deep CNN assigns labels to a set of images. Each image is accompanied by the correct label displayed underneath it. The neural network evaluates five possible labels for each image and assigns a probability to each one. Notably, in the majority of cases, the correct label is among these top five predicted labels.

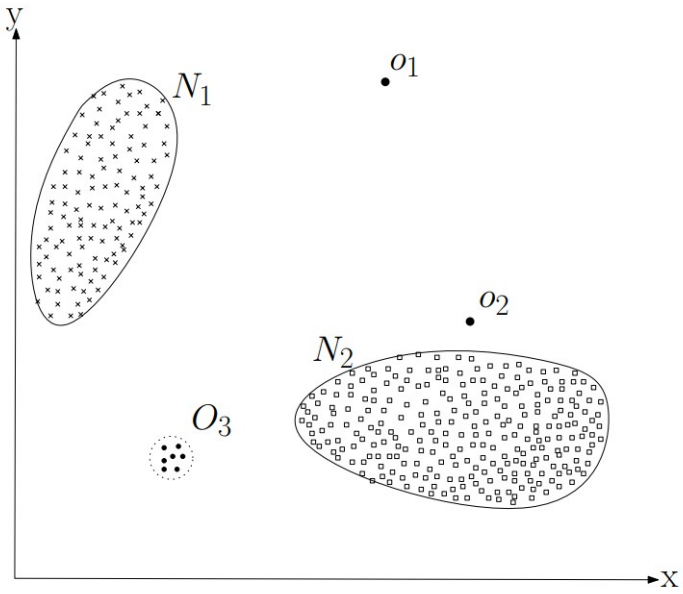


Figure 4.4: A simple example of anomalies in a 2-dimensional data set. N_1 and N_2 are the normal regions where most observations lie while O_1, O_2, and O_3, are the outliers [27].

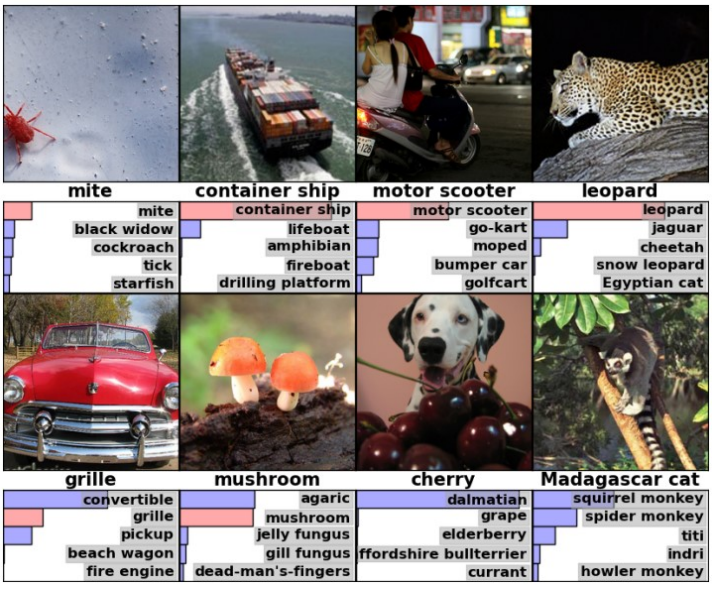


Figure 4.5: Image classification: A deep convolutional neural network considers the five most probable labels for each shown image along with their probabilities. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5 labels) [31].

4.4.3. Keyword spotting

Transitioning from visual to acoustic elements, we recognize the importance of a predictive model that can extract key words from continuous speech utterances, typically derived from voice recordings. By calculating posterior probabilities for keyword occurrence, keyword spotting plays a significant role in translating audio patterns into their corresponding textual context [32].

This capability finds application in virtual assistants, text generation, and data scraping, among others [33]. For example, today's virtual assistants are triggered by a wake-up word or phrase, such as ”Hey Siri” or ”Hey Google”, when said by the user. This includes detection of the wake-up word in accordance with the owner's voice. To ensure NEOX-V's timing simulator is on par with state-of-the-art hardware products that encompass a broad range of ”smart tools” in everyday tasks, we have incorporated tests to evaluate its performance in keyword spotting tasks.

4.4.4. Visual Wake Words

Particularly popular in the realm of IoT and significantly demanding in terms of power consumption, visual wake words are useful in ”waking up” the device on a specific visual detection signal [34]. For instance, a smartphone's screen turns on when the user stares directly at it. This eliminates the need for activation words like ”Hey Siri,” as the device can be activated simply by bringing it in front of one's face.

Accurate and swift device activation is crucial, necessitating the inclusion of test cases that ensure our system's responsiveness. Factors such as false positives (device activation with someone else's face) and time constraints (detecting and opening the home screen rapidly) directly impact the reviews and performance evaluation of embedded or wearable devices such as NEOX-V. The timing model can be extremely useful in addressing these considerations to meet user expectations and deliver a satisfactory user experience.

4.5. Conclusion

In conclusion, the validation process is of great importance as it ensures that the proposed solution effectively addresses the research problem. For this purpose, an extensive test suite is developed, including micro-kernel tests, system level tests, AI tests, as well as an automatic test generation tool. The purpose of the micro-kernel tests is to evaluate the performance of each individual NEOX-V component. This is why we developed tests that target the core and the memory system, including the instruction and data caches, the scratchpad memory, and the DMA, by extensively utilizing the corresponding types of events.

During the development of the timing simulator, it became evident that the results of the test cases are primarily influenced by the different memory access patterns and the number of threads exploited. This is why the validation process introduces a test generator that produces a wide range of test cases with different combinations of memory access patterns and the number of threads.

Transitioning from individual components to the system as a whole, system-level tests serve the purpose of validating the interaction and seamless cooperation among various components within the system. This is done by utilizing different combinations of the various event types. The tests begin with a straightforward memory copy test and progressively escalate in complexity, incorporating vector addition and culminating in a matrix multiplication test. This incremental approach enables a gradual increase in test complexity, with matrix multiplication being the fundamental operation in AI tests. These initial tests serve as the foundation for subsequent AI tests, building upon the basic operations to evaluate the system's performance in more advanced and intricate tasks.

The culmination of these tests lies in the evaluation of NEOX-V's performance through the AI tests, which target the specific applications it is designed to handle. As an embedded AI accelerator, NEOX-V is expected to excel in tasks like anomaly detection, image classification, keyword spotting, and visual wake words. Assessing its performance using the developed timing simulator provides valuable information regarding the design's efficiency, its functional correctness, as well as timing characteristics. This extensive evaluation enhances our understanding of NEOX-V's capabilities and ensures its suitability for AI applications.

5 Results

In the previous chapter, we introduced the validation framework utilized to verify the developed timing simulator. We described how a comprehensive set of tests is employed to validate the timing performance of the different components and events within NEOX-V. To model the timing behavior of the individual hardware components, we employ a diverse set of micro kernels alongside an automatic test generator tool. These tests enable us to assess the functionality and timing behavior of each component in isolation. Moving on to the interaction of these components, we utilize system-level tests to validate the timing accuracy of the model at the system level. These tests encompass various tasks such as memory copying between different regions, vector addition, and matrix multiplication. By evaluating the system-level behavior, we ensure that the components work harmoniously and deliver the expected timing behavior collectively. Considering the end-user AI applications that NEOX-V targets, we deployed a suite of AI tests to ensure its robust performance in real-world scenarios. These tests include anomaly detection, image classification, keyword spotting, and visual wake words.

This chapter aims to provide a comprehensive overview of the results as well as insightful comments and analysis regarding their implications. It is important to note that the goal of this thesis is to develop a timing simulator to achieve a relative error of 20% compared to the RTL version. The validation framework presented in the previous chapter is now used to examine whether this initial objective has been accomplished. Section 5.1 provides details about the calculation of timing model error, the variables considered in the tests, and an overview of the performance impact of the developed timing model. Section 5.2 focuses on the micro kernel tests results, Section 5.3 presents the outcomes from the test generator tool, Section 5.4 presents the results of the system-level tests, and Section 5.5 provides an overview of the AI tests.

5.1. Experimental setup

This section provides an overview of the experimental setup used to assess the accuracy and robustness of the NEOX-V timing simulator. It outlines the calculation of timing model error, the usage of each input parameters such as thread counts and read memory latencies, and the impact of the new timing model on the simulation execution time.

The results of the tests that follow show the percentage error of the timing model's simulation cycles compared to the cycles of the RTL simulator. This error is calculated using the formula:

$$e = \frac{rtl\_cycles - sim\_cycles}{rtl\_cycles} * 100 \tag{5.1}$$

where e represents the error, rtl_cycles denotes the test's execution time in number of cycles run by the RTL, while sim_cycles is the execution time in number of cycles simulated by the timing model. This formula applies to all the subsequent figures.

To evaluate the accuracy and robustness of NEOX-V timing simulator, the tests are conducted with different numbers of threads, ranging from 1 to 512. Additionally, different read memory latencies are taken into consideration, specifically 0 cycles and 40 cycles. The read memory latency refers to the

number of cycles that NEOX-V accelerator requires to read data from the main memory. By conducting tests with varying thread counts and read memory latencies, we can comprehensively assess the accuracy and robustness of NEOX-V timing model across a wide range of scenarios.

Finally, it is worth mentioning that the new timing model has been carefully designed to minimize the extra overhead (in terms of simulation time) on the overall execution time of the simulation. After extensive testing and optimization, we are pleased to report that the new timing model introduces a mere 0.7% increase in the simulation's execution time. This marginal overhead demonstrates the efficiency and effectiveness of the new model in accurately capturing timing behavior while maintaining the simulation's overall performance.

5.2. Micro kernel tests

As explained before, the micro kernel tests are small tests (also called unit tests) that target to evaluate the timing behavior of individual components of NEOX-V. Figure 5.1 shows the timing model's error for the instruction cache micro kernel tests. As observed, the error consistently remains below 2% for all the test cases, with an average of 0.97%, encompassing the following scenarios: only cache hits, only cache misses, and combinations of both. We observe that, in some cases, the error tends to decrease, particularly in situations involving instruction cache misses.

Upon a closer examination of the RTL traces compared to the model traces, we notice that certain instructions in the RTL require additional time to complete their execution, typically 2 or 3 cycles, which is not accounted for in the timing model. The exact reason remains unclear (it is related to the propagation delays of specific signals in the design), but it appears that certain threads remain busy for a longer duration than anticipated. However, it is crucial to note that such cases are infrequent and primarily affect smaller tests, creating an impression of a larger error compared to the larger tests. Despite this disparity, it is important to note that the impact on the error is minimal. As a result, there is no reason to incorporate this additional timing behavior into the model or investigate this further, as it does not significantly affect the overall accuracy of the simulation.

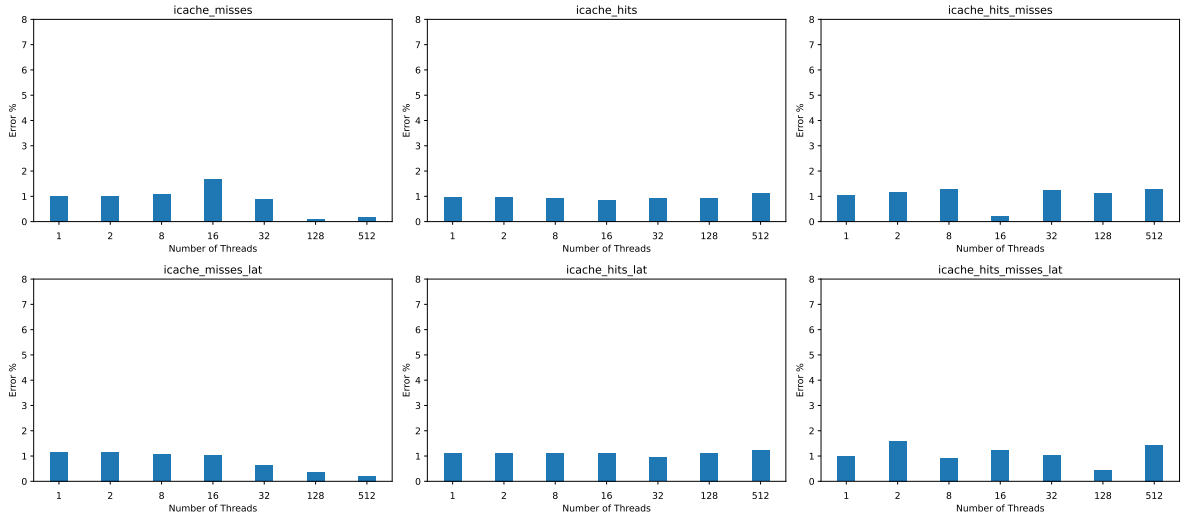


Figure 5.1: The error % of the timing model compared to the RTL simulation when testing the instruction cache (no data cache is used). The tests include cache accesses that are hits (icache_load_hits), misses (icache_load_misses) or a combination (icache_load_hits_misses). Each test is run with a memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the name of each test).

Figure 5.2 depicts the timing model's error for the data cache micro kernel tests involving load instructions. Notably, the average error across all tests is 1.12%, while every individual test exhibits an error below 2%. Tests that exhibit a higher number of data cache misses tend to display a slightly elevated error rate. Moreover, it is important to consider that the current low error, although minimal, could potentially arise due to additional cycles required by the RTL simulation. More specifically, the current timing simulator does not incorporate modeling for bank conflicts and line buffers, which could contribute to the observed error. Bank conflicts occur when multiple memory accesses target the same

memory bank simultaneously, potentially leading to delays and affecting timing behavior. Additionally, line buffers are utilized between the core and the data cache to manage and buffer memory accesses, further impacting overall timing.

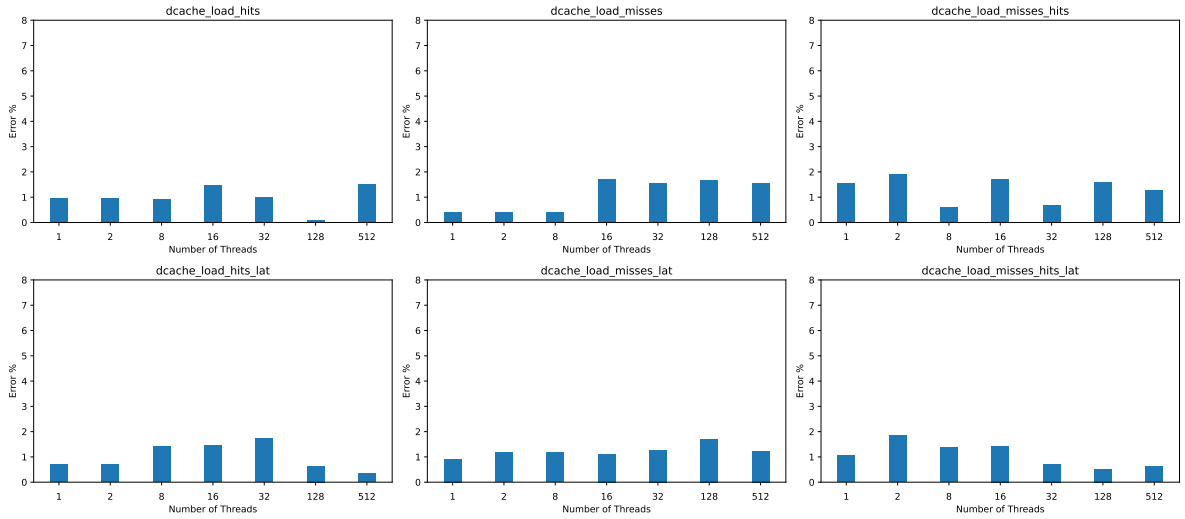


Figure 5.2: The error (%) of the timing model compared to the RTL simulation when testing load operations from the data cache. The tests include cache accesses that are hits (dcache_load_hits), misses (dcache_load_misses) or a combination (dcache_load_hits_misses). Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

Figure 5.3 depicts the timing model's error for the data cache micro kernel tests consisting of store instructions. The error consistently remains below 2% for all test cases, with an average error rate of 1.21%. It is worth noting that this error is primarily attributed to the factors previously explained, such as the absence of modeling for bank conflicts and line buffers. It is important to highlight that there is a slight tendency for tests with a higher number of data cache misses to exhibit a slightly increased error. Furthermore, upon analyzing the graphs, it becomes apparent that the error is exceptionally low for cases with 512 threads. This observation can be attributed to the following reason: when executing with a smaller number of threads, there are instances where execution stalls due to data cache misses. However, in scenarios with a larger number of threads, this stall is alleviated due to the presence of multiple active threads, allowing for improved concurrency and reduced impact from individual cache misses.

Figure 5.4 illustrates the timing model's error for the scratchpad micro kernel tests. As we can see, the error consistently remains below 2% for both larger and smaller tests, with an average error rate of 1.12%. These findings underscore the accuracy and robustness of the developed timing model in simulating the timing behavior of a software-controlled memories like the scratchpad memory.

Figure 5.5 displays the timing model's error for the DMA micro kernel tests, specifically evaluating load and write back operations. Notably, the results demonstrate that the error consistently remains below 2%, with an average error rate of 1.24%. This signifies the high accuracy of the timing model in capturing the timing behavior of both load and write-back DMA operations.

Figure 5.6 illustrates the timing model's error for the micro kernel tests that extensively utilize stack memory. Notably, we observe that the error consistently remains below 3% with an average of 1.24%, indicating excellent performance across all test cases, even when encountering multiple cache misses and heavy stack memory usage.

5.3. Test generator

In the previous section, it was shown that tests with a higher number of data cache misses exhibit a slightly elevated error rate. The test generator creates a list of tests with different thread counts and memory accesses to examine this behavior. Figure 5.7 displays the percentage error of the timing model when compared to the RTL equivalent. The test name follows the format:

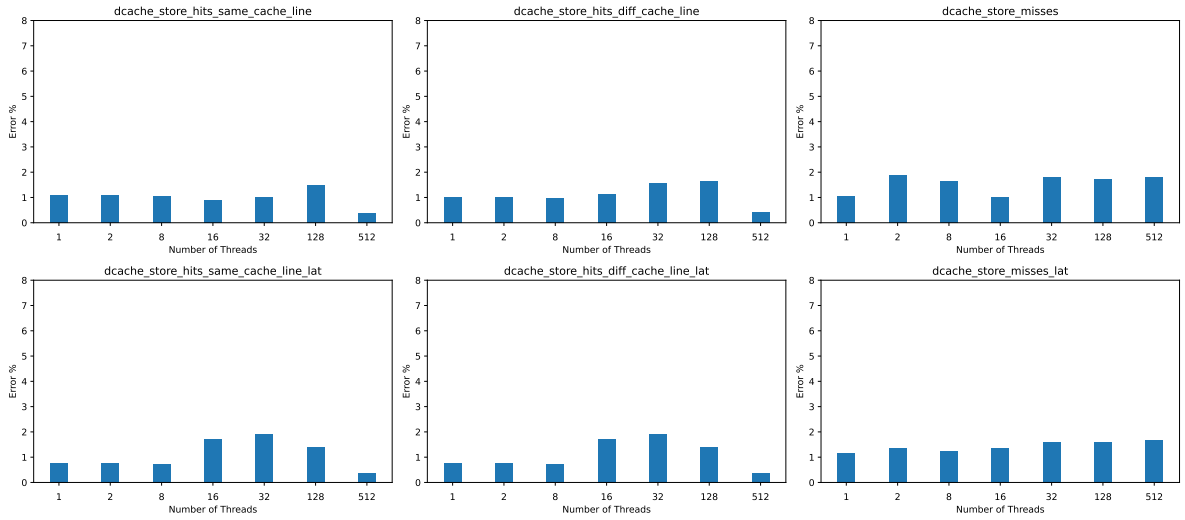


Figure 5.3: The error (%) of the timing model compared to the RTL simulation when testing store operations from the data cache. The tests include cache accesses that are hits on the same (dcache_store_hits_same_cache_line) or different (dcache_store_hits_diff_cache_line) cache line or misses (dcache_store_misses). Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

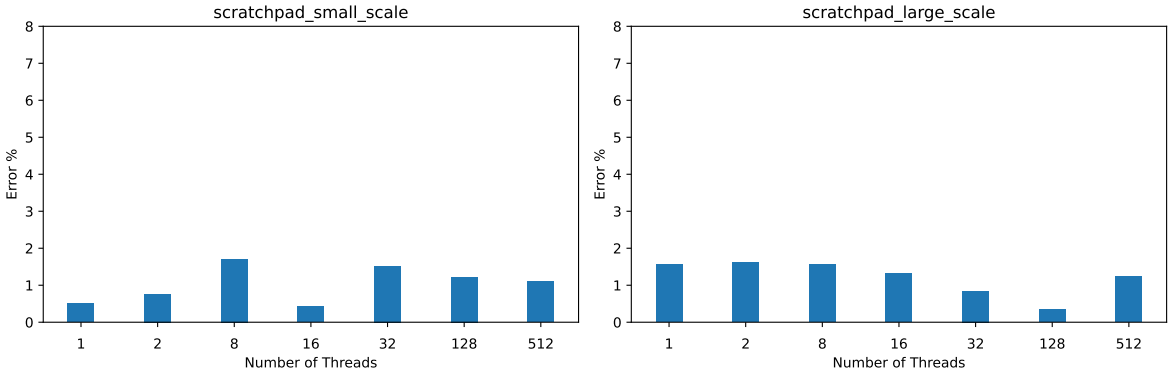


Figure 5.4: The error (%) of the timing model compared to the RTL simulation when testing the scratchpad in small (scratchpad_small_scale) and large scale(scratchpad_large_scale).

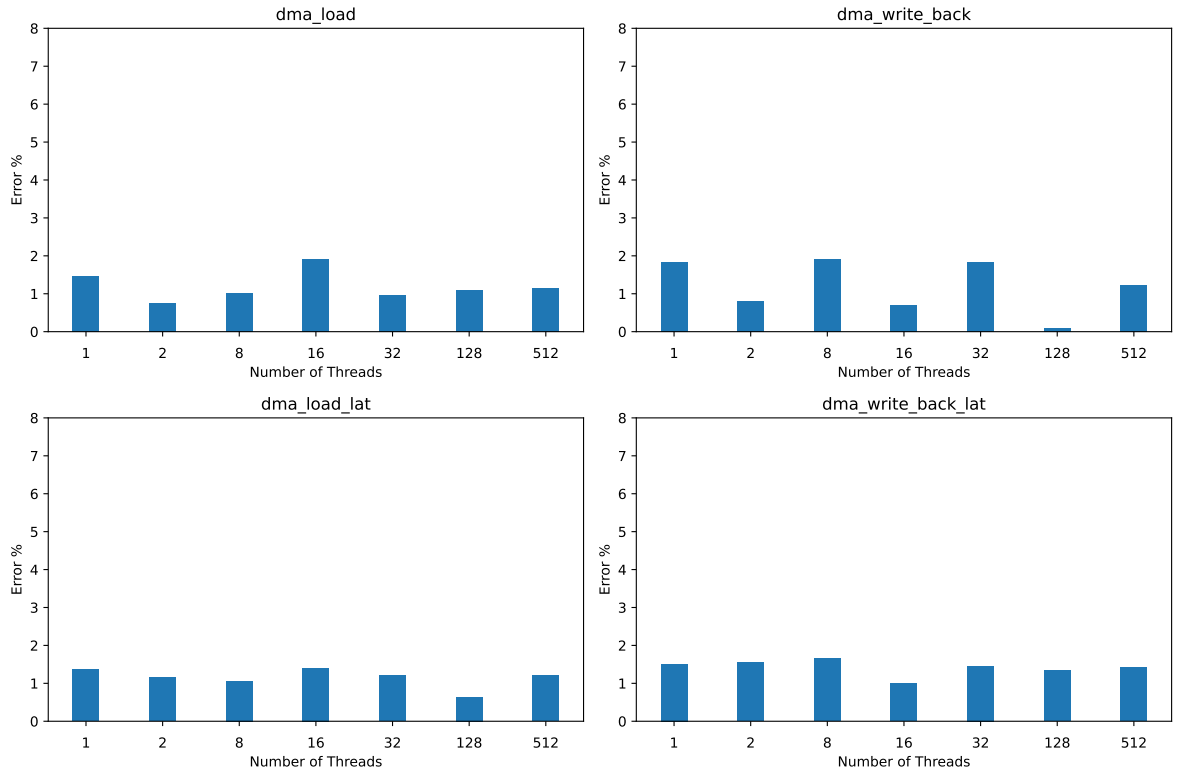


Figure 5.5: The error (%) of the timing model compared to the RTL simulation when testing DMA's load and write back operations. Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

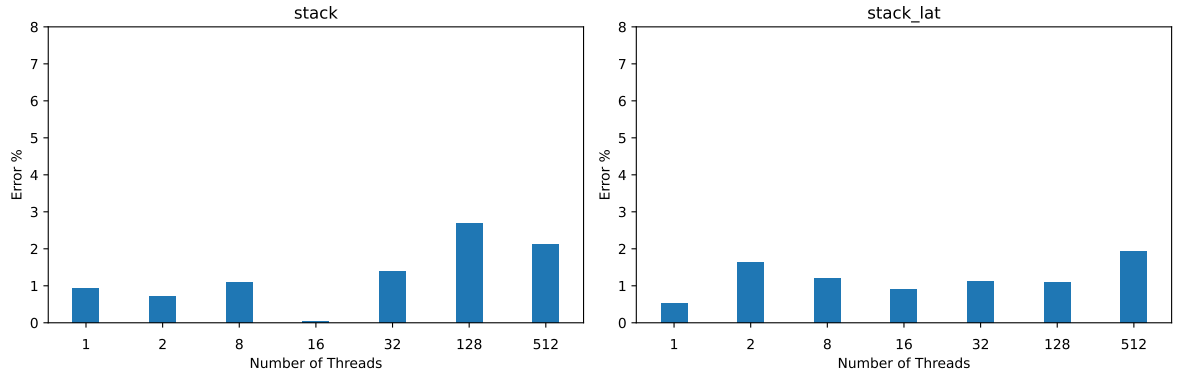


Figure 5.6: The error (%) of the timing model compared to the RTL simulation when testing the stack memory. Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

$$test\_t\{threads\}\_m\{misses\}\_h\{hits\},$$

where $threads$ represents the number of threads, $misses$ indicates the number of cache misses, and $hits$ denotes the number of cache hits. The average error is 1.91%, while all tests have an error lower than 7%. It can be observed that there is a trend indicating an increase in error with a higher number of cache misses. It is important to note that this behavior is not attributed to the modeling of the cache itself but rather to the scheduling of threads. The discrepancy arises from the fact that the scheduler may not have instantiated all the threads yet. Consequently, thread 0 may run again before thread 1, while the RTL execution takes into account the intended order of execution. This scheduling difference can contribute to the observed variations in errors between scenarios with different cache miss counts. Despite this difference in scheduling, it is important to note that the error remains very low, indicating that it does not pose a significant problem. The observed variations in errors between scenarios with different cache miss counts, attributed to the scheduling differences, do not significantly impact the overall accuracy of the timing model. The low error values suggest that the timing model performs well despite the scheduling discrepancies.

5.4. System level tests

Now that the timing performance of the individual components has been evaluated, the focus shifts to their coordination. The system level tests are used to evaluate the performance of the system as a whole.

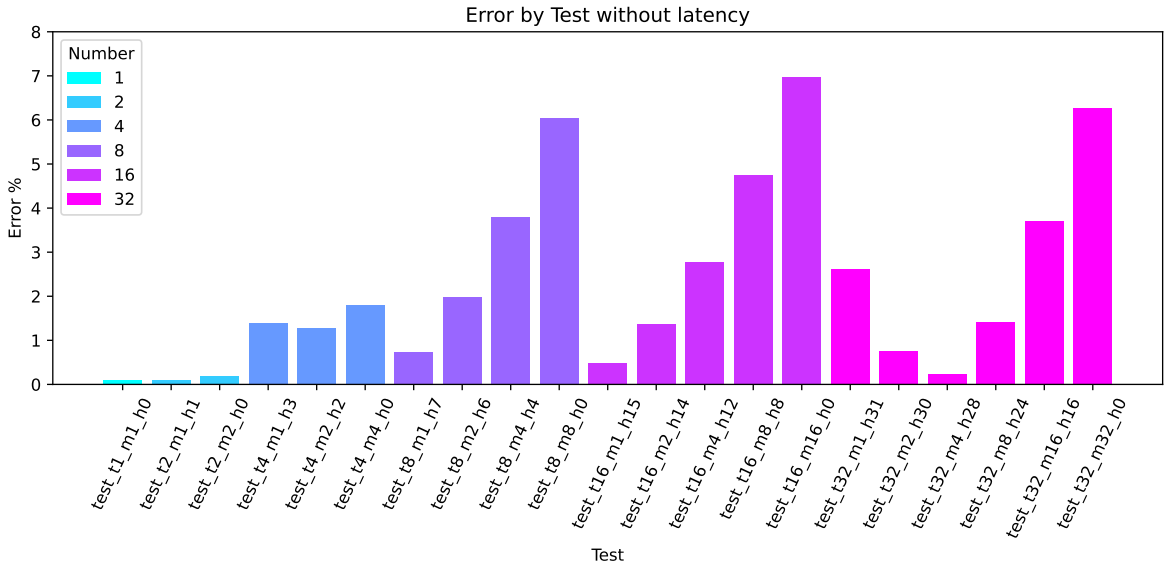
Figure 5.8 illustrates the timing model's error for the system-level test involving the copying of a memory region to another. It is noteworthy that the error consistently remains below 8% for all test cases, with an average error rate of 2.18%. This error is influenced by several factors previously discussed, such as the absence of modeling for bank conflicts or line buffers, different scheduling behavior, and potential discrepancies in simulation cycles compared to the RTL simulation.

These factors have distinct characteristics and can result in varying error rates depending on factors like the number of threads, test sizes, and instructions used. To ensure a comprehensive coverage, this section incorporates different tests that generate diverse assembly code, thereby encompassing a broader range of scenarios. By including various test cases, the aim is to account for the variability in error rates arising from these different factors, providing a more comprehensive evaluation of the timing model's performance.

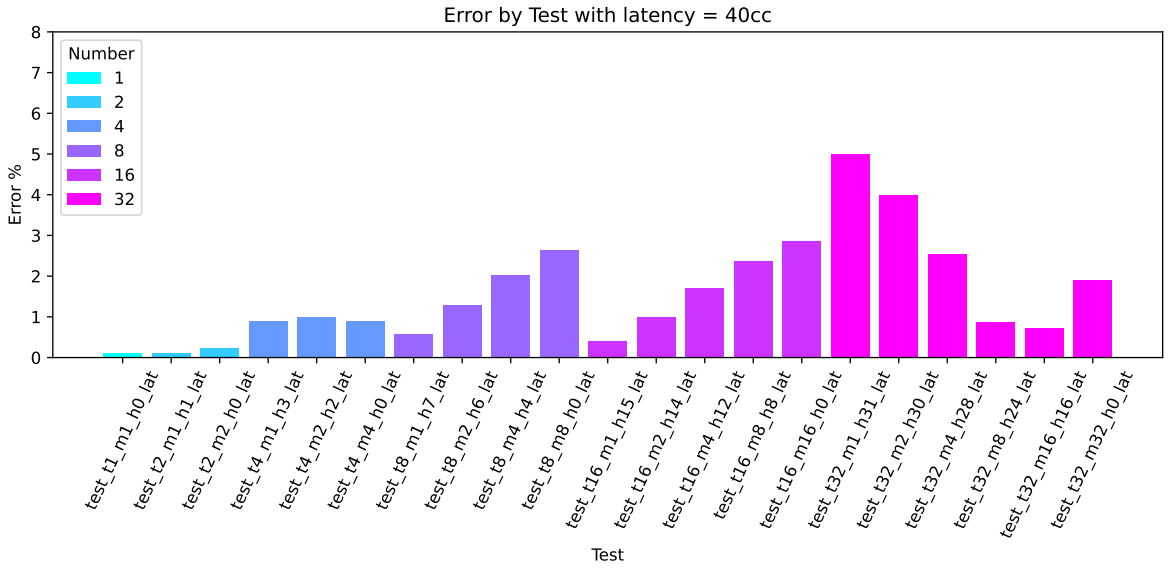
Figure 5.9 presents the timing model's error for the system-level test involving a vector addition operation. It is observed that the error consistently remains below 7% for all test cases, with an average error rate of 2.24%. This error is primarily influenced by a combination of factors discussed earlier. Furthermore, it is noteworthy that the test with 512 threads, when executed with a 40cc latency, exhibits a decreased error. This behavior can be attributed to the presence of multiple threads, which enable concurrent execution without the need for frequent execution stalls. Consequently, the impact of cache misses and latency is mitigated, resulting in a lower error rate.

Figure 5.10 presents the timing model's error for the system-level test involving a matrix multiplication operation. The average error is calculated to be 3.84%, and it is worth noting that all errors remain below the 8% threshold. An interesting observation is that the error tends to increase as the number of threads increases. This behavior indicates that the timing model's accuracy may be influenced by the concurrency and parallelism introduced by multiple threads. As more threads are involved in the computation, the potential for contention and synchronization increases, leading to a higher likelihood of timing discrepancies. Consequently, in this case, the error rate tends to be higher in scenarios with a larger number of threads.

Figure 5.11 shows the timing model's error for the system level test performing a tiled matrix multiplication, with an average error of 5.19%. An interesting observation is that the error decreases as the size of the tile increases. This can be attributed to the exploitation of more parallelism when using larger tile sizes, resulting in shorter execution times. As the execution time decreases, the impact of any timing discrepancies or modeling limitations on the overall error diminishes. Consequently, tests with shorter execution times tend to exhibit smaller errors, which is a reasonable outcome. This analysis highlights the relationship between tile size, execution time, and error, emphasizing the significance of parallelism in minimizing timing discrepancies and improving the overall precision of the timing model.



(a) Latency = 0cc



(b) Latency = 40cc

Figure 5.7: Error (%) for tests created by the test generator tool with and without memory access latency.

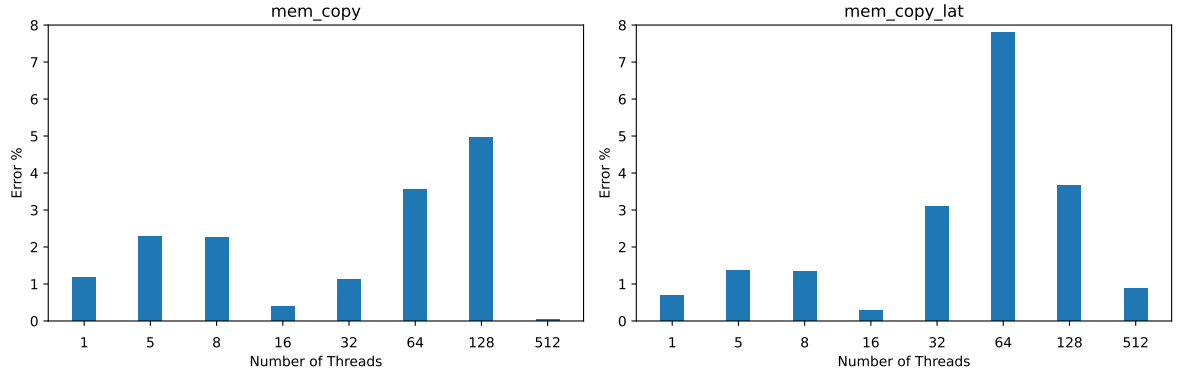


Figure 5.8: The error (%) of the timing model compared to the RTL simulation when testing a memory copy operation. Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

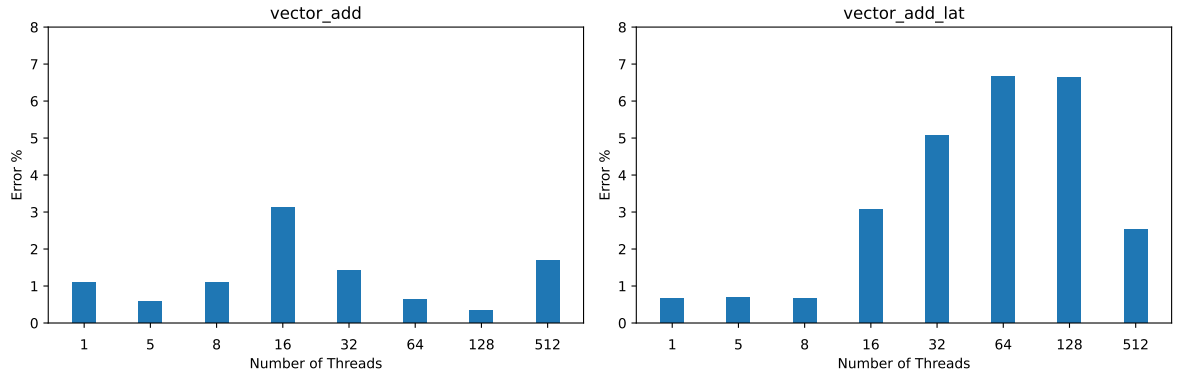


Figure 5.9: The error (%) of the timing model compared to the RTL simulation when testing the addition of two vectors. Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

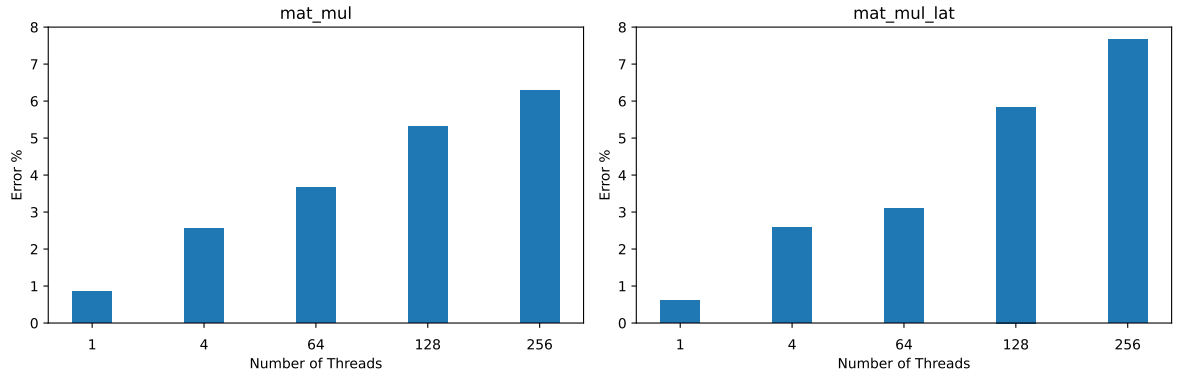


Figure 5.10: The error (%) of the timing model compared to the RTL simulation when testing the multiplication of two matrices. Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

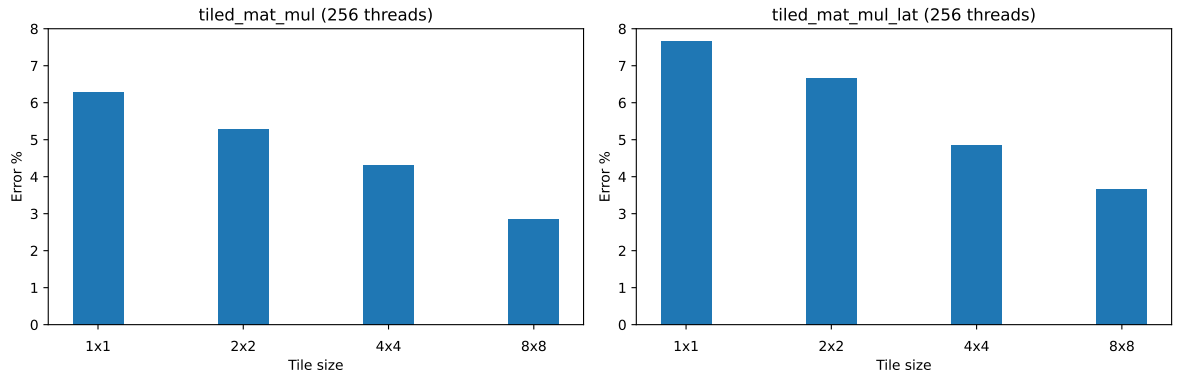


Figure 5.11: The error (%) of the timing model compared to the RTL simulation when testing the multiplication of two matrices with tiling. These tests use 256 threads and varying tile sizes. Each test is run with memory read latency equal to 0 and 40 (indicated with "_lat" at the end of the test name).

5.5. AI tests

The AI tests serve as a real life application for the NEOX-V accelerator. Recognizing the need for accelerated execution compared to simulation, an FPGA was employed to achieve significantly faster AI processing. To ensure accurate simulation results, the FPGA latencies were also incorporated into the timing simulator. This integration enabled the simulator to faithfully replicate the timing behavior observed on the FPGA, allowing for comprehensive evaluation and performance analysis of the NEOX-V accelerator in AI workloads, as seen in the figures that follow.

Figure 5.12 illustrates the error of the AI tests in comparison to the RTL. Remarkably, the average error is calculated to be a mere 0.69%, with all AI tests exhibiting errors below 3%. This exceptional performance of the timing model can be attributed to several factors. Firstly, the AI tests are optimized to ensure a well-balanced workload among different threads. This balanced distribution of tasks minimizes the potential for timing discrepancies and contributes to the overall accuracy of the timing model. Secondly, the AI tests specifically utilize the scratchpad instead of the data cache. By leveraging the scratchpad, the tests mitigate the potential effects of cache-related timing variations, leading to more reliable and consistent timing behavior. These factors collectively result in significantly lower errors when compared to the matrix multiplication tests. The optimized workload distribution and utilization of the scratchpad play vital roles in reducing timing discrepancies and enhancing the precision of the timing model for the AI tests. This emphasizes the importance of tailored optimizations and architectural considerations in achieving high accuracy in timing simulations.

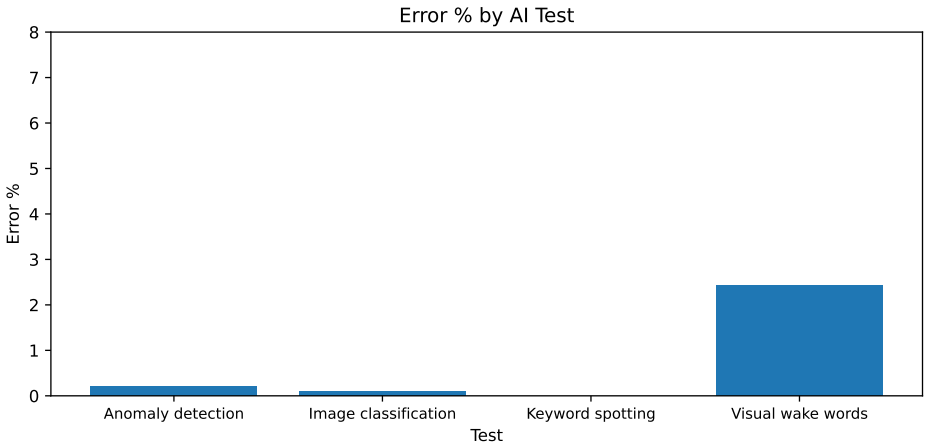


Figure 5.12: The error (%) of the timing model compared to the RTL simulation for the AI tests.

Figure 5.13 showcases the error per layer for the image classification test, representing the perfor-

mance of the timing model at a more fine-grain manner. The purpose of presenting this error breakdown is twofold. Firstly, it demonstrates that the timing model can provide insights into the performance of individual layers within a test, allowing for a detailed analysis of the timing behavior. Secondly, it verifies that the low error observed in the overall test is not a result of canceling out larger errors when averaging them across layers.

Upon examination, it becomes evident that all the errors for the individual layers of the image classification test remain below 1%, resulting in an impressively low average error of 0.08% for the entire test. This finding underscores the high accuracy and reliability of the timing model, as the individual layer errors contribute to the overall error with minimal impact. Once again, the timing simulator's granular assessment and predictive capabilities for real-life applications are reaffirmed.

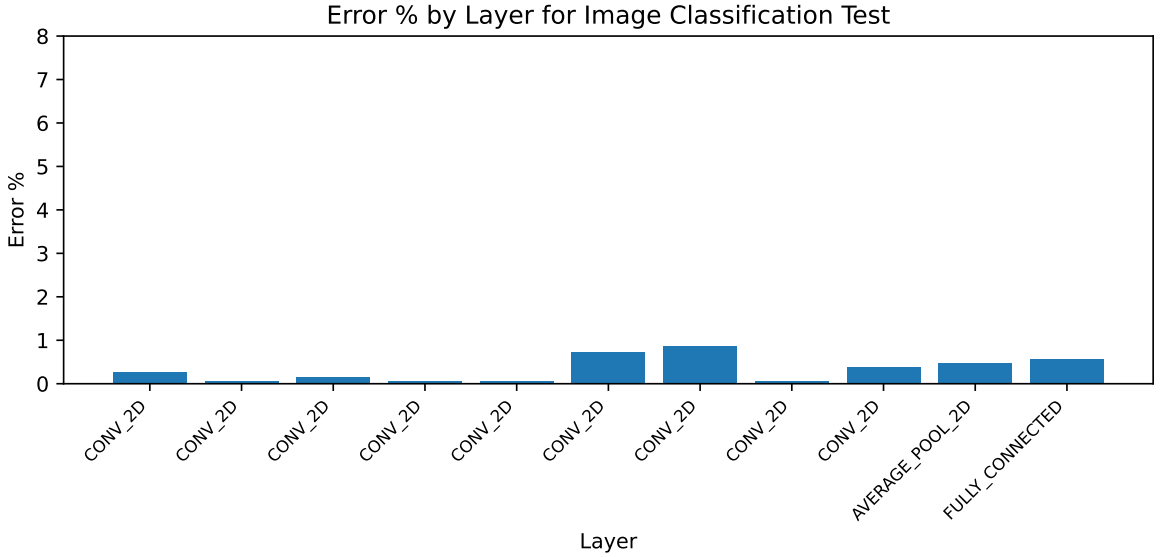


Figure 5.13: The error (%) of the timing model compared to the RTL simulation for each layer of the Image Classification AI test.

5.6. Conclusion

This chapter presented the results of the thesis, providing valuable insights into the performance and accuracy of the proposed timing model in various test scenarios. For this purpose, significant factors such as thread count, memory access patterns, and workload distribution were considered. Through a comprehensive analysis of micro kernel tests, automatically generated tests, system-level tests, and AI tests, the capabilities and limitations of the timing model were evaluated.

The results of the evaluation showcase the exceptional performance of the timing model, consistently maintaining errors below predefined thresholds. The timing model not only meets the initial requirement of maintaining a relative error below 20% compared to the RTL version but also demonstrates errors consistently below 8% across all test cases.

In more detail, micro-kernel tests showcase errors below 2% and average at 1.15%. A comprehensive test set generated by the test generator further examined the behavior of the data cache, encompassing various combinations of thread counts and access patterns. This evaluation resulted in an average error of 1.91% for these tests, all of which remained below 7%. These findings highlight the non-trivial nature of modeling the data cache, given its complexity as a system. Furthermore, the system-level tests performed exceptionally well, with errors consistently below 8% and an average error rate of 3.36%. This signifies the timing model's excellent predictive capabilities in simulating the execution of instruction combinations while considering various components and the system as a whole. Lastly, the AI tests demonstrated outstanding performance, with an average error of 0.69% and all tests falling below the 3% threshold. These impressive results can be attributed to the optimized and well-balanced nature of the AI tests, as well as the utilization of the simpler scratchpad memory

instead of the data cache. This further confirms the timing model's excellent performance in accurately simulating the target application of NEOX-V, which focuses on AI workloads.

Despite certain factors, such as the absence of modeling for bank conflicts or line buffers and variations in scheduling, the impact on the overall error was found to be minimal. Furthermore, it is worth noting that the timing model introduces a negligible increase of only 0.7% in the simulation's execution time. This marginal overhead showcases the efficiency and effectiveness of the timing model in accurately capturing timing behavior while minimizing any negative impact on the overall performance of the simulation.

Through this comprehensive validation process, we strive to guarantee the accuracy, reliability, and robustness of the NEOX-V new timing simulator, aligning it with the requirements of end-user AI applications. Not only are the robustness of the timing model and the integrity of the presented results verified, but researchers can also get a deeper insight into the behavior and performance of the system.

6 Conclusion

This chapter serves as the culmination of the thesis, presenting an extensive summary of the research findings and their implications. Section 6.1 presents an overview of each step of the research process including the related work and background, the implementation details, the validation framework, and its results. By revisiting the problem statement, goals and research question in Section 6.2, we can ascertain the extent to which the research objectives were achieved and evaluate the significance of the findings. Furthermore, Section 6.3 provides an opportunity to discuss the limitations of this study, propose future research directions, and highlight the practical implications and potential applications of the research findings. Overall, this chapter consolidates the thesis by synthesizing the key contributions and offering a conclusive perspective on the research outcomes.

6.1. Summary

Over time, engineers and researchers have been continually developing various state-of-the-art techniques to effectively model a GPU's functionality and predict its execution time. Chapter 2 presents these techniques, which range from simple and fast simulators to heavyweight cycle-accurate ones or even machine learning methods. SimpleScalar [7] is a simple but powerful architecture simulator that models the microarchitecture of a processor while also providing timing information. It exhibits certain limitations, though, such as limited ISA support, flexibility, or parallelism due to its simplistic nature. On the other hand, Gem5 [8, 9] is a complex cycle-accurate simulator that offers detailed modeling and exploration of various configurations. Despite its powerful nature, its complexity makes it a heavyweight simulator with a long execution time. When execution time is prioritized, its accuracy decreases. Often, analytical models are preferred over architectural simulators to perform GPU modeling. A mechanistic performance model [14] proposes that effective modeling can be achieved if the execution time is split into intervals disrupted by miss events. It focuses on modeling the events that degrade a GPU's performance. On the contrary, a simple BSP-based model [15] splits the execution time into two phases: communication and computation. It then uses mathematical models to predict the execution time based on events that happen during these two phases.

While these techniques are commonly employed, it is important to acknowledge their limitations. Some of them have a theoretical and generic nature, disregarding the complexities of complex memory systems, multi-threaded or multi-core architectures, and ISA-specific configurations. On the other hand, the more detailed ones can be overwhelming, requiring significant effort and expertise to effectively utilize the tool. Additionally, their complexity introduces significant computational overhead and simulation runtime, impacting time-sensitive projects or situations that demand faster iterations.

Considering these limitations of the existing simulators, NEOX-V stands out as a promising option for simulation due to its unique features and capabilities. NEOX-V is an ultra-low-power RISC-V based GPU processor optimized for both GPGPU and AI workloads, which serves as a case study for the current thesis. It is multi-core and multi-threaded, with a configurable number of cores serving a wide range between power, performance, and functional integration with different levels of the SoC platform. Multithreading hides long latency delays from the external memory controller, maintaining high computation throughput for the entire system. NEOX-V caters to a range of domains, including AI, IoT/Edge,

and performance media processing, targeting both consumer and industrial markets.

Its architecture incorporates AI-specific ISA extensions and SIMD vectors of varying lengths, including 8-bit data types. Additionally, it optionally includes Graphics ISA Extensions for Unified Shader Architecture, Tile Based Rendering, Color/Vertex processing, and Vector Support. Moreover, the system comprises dedicated hardware modules such as a rasterizer, texture unit, tile management unit, and texture caches. The current implementation of NEOX-V includes the RTL code, the SDK for graphics and AI applications, and a high-level functional simulator in C/C++. At its current state, this simulation is constrained to the functional level, implying that it does not provide timing information pertaining to the program execution. As a result, NEOX-V offers an ideal opportunity for exploring how we can incorporate timing characteristics into a functional simulator to achieve a balanced trade-off between accuracy and execution time.

However, building a timing simulator for NEOX-V is a challenging task. Some of these challenges arise from its highly multi-threaded architecture and the need to synchronize multiple computational and memory components. Moreover, the requirement to strike a balanced trade-off between accuracy and execution time adds another layer of complexity. Achieving a small error percentage between the timing model and RTL implementation is difficult, given the importance of maintaining low execution times. Additionally, the dynamic nature of the accelerator's development requires adaptability and flexibility in the simulation process to incorporate ongoing iterations and bug fixes.

After delving into the relevant literature and foundational knowledge required to understand the research question, the focus now shifts to the implementation details. The subsequent chapter will demonstrate how these theoretical concepts are translated into a tangible implementation, overcoming the challenges presented. This will allow us to further advance towards building an event-driven timing simulator with high accuracy and low execution time.

Chapter 3 presents the implementation details of the NEOX-V timing simulator using event-driven modeling. The proposed solution estimates the simulation time of a program execution in clock cycles. The timing simulator is added as an extension to the functional simulator, which models the execution of a program consisting of multiple instructions. For each iteration of the simulation, the functional simulator simulates the execution of an instruction by a thread. At the same time, it gives as input to the timing simulator the instruction status, which includes some information about the instruction that was just simulated. Using this information, the timing simulator predicts the thread ready status, which is the clock cycle in which the thread executing this instruction will be ready. With the term ”ready”, we refer to the fact that the thread has completed executing an instruction and is ”ready” to execute a new one. At the end of the simulation, the timing simulator provides as output the total clock cycles needed to execute the whole program.

This estimation of clock cycles is achieved using event-based modeling. To model each specific NEOX-V component, we utilize distinct types of events. Consequently, there is a one-to-one correspondence between the main NEOX-V components and the events within the timing simulator. In each iteration of the simulation, the execution of an instruction is split into different events. The timing simulator consists of an event generator, which creates these events using the input from the functional simulator. These events can be core, data cache, instruction cache, scratchpad, or DMA events, representing the different components the instruction had to utilize. To handle these events, there is an event manager. This event manager uses the information about the events to predict the clock cycles needed for their completion. Cache events are a special case since NEOX-V supports only blocking caches. To control these events, the event manager creates and manages two queues, one for the pending events and one for the ongoing events, called pending event queue and ongoing event queue respectively.

Moreover, thread scheduling is of great importance to developing a timing simulator with high accuracy compared to the RTL equivalent. Therefore, the thread scheduler, which is part of the functional simulator, has to account for the timing information of the threads. It is enriched with multiple scheduling policies that consider the thread ready status. These scheduling policies include a typical round robin, a grouped round robin with 8 threads, a grouped round robin with 32 threads, and a minimum scheduling policy.

It is worth mentioning that the simulator serves as valuable feedback for the RTL and functional model of NEOX-V, aiding in the identification and resolution of scheduler and cache limitations. All these components and insights collectively contribute to the successful implementation and refinement of the timing model of NEOX-V.

Chapter 4 presents the validation process used to ensures that the proposed solution effectively addresses the research problem. For this purpose, an extensive test suite is developed, including micro-kernel tests, system level tests, AI tests, as well as an automatic test generation tool.

The purpose of the micro-kernel tests is to evaluate the performance of each individual NEOX-V component. This is why we developed tests that target the core and the memory system, including the instruction and data caches, the scratchpad memory, and the DMA, by extensively utilizing the corresponding types of events.

During the development of the timing simulator, it became evident that the results of the test cases are primarily influenced by the different memory access patterns and the number of threads exploited. This is why the validation process introduces a test generator that produces a wide range of test cases with different combinations of memory access patterns and the number of threads.

Transitioning from individual components to the system as a whole, system-level tests serve the purpose of validating the interaction and seamless cooperation among various components within the system. This is done by utilizing different combinations of the various event types. The tests begin with a straightforward memory copy test and progressively escalate in complexity, incorporating vector addition and culminating in a matrix multiplication test. This incremental approach enables a gradual increase in test complexity, with matrix multiplication being the fundamental operation in AI tests. These initial tests serve as the foundation for subsequent AI tests, building upon the basic operations to evaluate the system's performance in more advanced and intricate tasks.

The culmination of these tests lies in the evaluation of NEOX-V's performance through the AI tests, which target the specific applications it is designed to handle. As an embedded AI accelerator, NEOX-V is expected to excel in tasks like anomaly detection, image classification, keyword spotting, and visual wake words. Assessing its performance using the developed timing simulator provides valuable information regarding the design's efficiency, its functional correctness, as well as timing characteristics. This detailed evaluation enhances our understanding of NEOX-V's capabilities and ensures its suitability for AI applications.

Chapter 5 presents the results of the thesis, providing valuable insights into the performance and accuracy of the proposed timing model in various test scenarios. For this purpose, significant factors such as thread count, memory access patterns, and workload distribution are considered. Through a thorough analysis of micro kernel tests, automatically generated tests, system-level tests, and AI tests, the capabilities and limitations of the timing model are evaluated.

The results of the evaluation showcase the exceptional performance of the timing model, consistently maintaining errors below predefined thresholds. The timing model not only meets the initial requirement of maintaining a relative error below 20% compared to the RTL version but also demonstrates errors consistently below 8% across all test cases.

In more detail, micro-kernel tests showcase errors below 2% and average at 1.15%. An extensive test set generated by the test generator further examined the behavior of the data cache, encompassing various combinations of thread counts and access patterns. This evaluation resulted in an average error of 1.91% for these tests, all of which remained below 7%. These findings highlight the non-trivial nature of modeling the data cache, given its complexity as a system. Furthermore, the system-level tests performed exceptionally well, with errors consistently below 8% and an average error rate of 3.36%. This signifies the timing model's excellent predictive capabilities in simulating the execution of instruction combinations while considering various components and the system as a whole. Lastly, the AI tests demonstrated outstanding performance, with an average error of 0.69% and all tests falling below the 3% threshold. These impressive results can be attributed to the optimized and well-balanced nature of the AI tests, as well as the utilization of the simpler scratchpad memory instead of the data cache. This further confirms the timing model's excellent performance in accurately simulating the target application of NEOX-V, which focuses on AI workloads.

Despite certain factors, such as the absence of modeling for bank conflicts or line buffers and variations in scheduling, the impact on the overall error was found to be minimal. Furthermore, it is worth noting that the timing model introduces a negligible increase of only 0.7% in the simulation's execution time. This marginal overhead showcases the efficiency and effectiveness of the timing model in accurately capturing timing behavior while minimizing any negative impact on the overall performance of the simulation.

Through this extensive validation process, we strive to guarantee the accuracy, reliability, and robustness of the NEOX-V new timing simulator, aligning it with the requirements of end-user AI applica-

tions. Not only are the robustness of the timing model and the integrity of the presented results verified, but researchers can also get a deeper insight into the behavior and performance of the system.

6.2. Main contributions

In this section, we will revisit the problem statement and goals of the thesis in order to evaluate the extent to which the research question was addressed and the goals were achieved. The research question aimed to explore the incorporation of timing characteristics into a functional simulator to achieve a balanced trade-off between accuracy and execution time.

To assess the proposed timing model, we verify if the following requirements are met:

- Accuracy: The requirement set for the timing simulator was to achieve a relative error of 20% compared to the RTL version. Notably, the thesis not only met but exceeded this initial requirement, demonstrating errors consistently below 8% across all test cases. In particular, for the AI tests that belong to the target application of NEOX-V, an average error of 0.69% and a maximum error of only 2.42% are achieved. The reported outcome significantly surpasses the original target, highlighting the substantial improvement attained through the research and validation process.
- Overhead: The requirement for the execution time of the timing simulator was to not increase more than 10%. The timing model introduces a negligible increase of only 0.7% in the simulation's execution time. This is very important, as it indicates that the model can seamlessly integrate into the simulation workflow without significantly extending the simulation duration.
- Flexibility: The timing simulator offers easy configuration with various parameters that are specified at the beginning of each simulation and can differ for each run. These parameters include the number of cores, the number of threads, the memory configuration, and the instruction latencies. These latencies encompass the time in clock cycles required for different pipeline stages, cache accesses (miss or hit), and store or load operations. Furthermore, the test generator, which is utilized to validate the proposed solution, can be adjusted to generate a wider or narrower range of test cases based on specific requirements.
- Extensibility: The timing simulator is implemented in a modular way, enabling the seamless addition of new features. Various elements of the timing simulator can be extended to support additional functionalities or accommodate new NEOX-V components. For instance, the event generator and event manager can be easily expanded to generate and handle new types of events. Additionally, the queues used to manage cache events can be utilized to serve other blocking NEOX-V components with minimal modifications. Furthermore, the test generator employed to validate the proposed solution can be slightly modified to generate new test cases.
- User activation: The timing simulator can be seamlessly enabled or disabled without disrupting the overall functionality of the system. Integrating timing information involved modifying the thread scheduler to accommodate the timing of each thread. However, one of the implemented scheduling policies is designed not to take this timing information into account, allowing the timing simulator to be disabled when needed.

It is evident that the proposed timing model strikes a favorable balance between cycle-accurate simulators, which offer high accuracy but can be time-consuming, and event-driven simulators, which prioritize speed but may sacrifice accuracy to some extent. More specifically, the goals established to successfully meet the above requirements were:

- Deep understanding of the functional simulator: A test suite was developed to thoroughly study the simulation traces from the model, contributing to a deep understanding of the underlying principles, architecture, and internal operations of the simulator.
- Identification and modeling of critical components: By examining the simulation traces from the RTL simulations, the major latency-inducing computational and memory components were identified. These are the NEOX-V core, the cache network, thread scheduler, scratchpad memory, and DMA operations.
- Enhancement of the existing simulator: The timing of the identified components of the system was simulated using event-driven modeling. Each NEOX-V component is modeled using specific types of events, resulting in a one-to-one correspondence between the main NEOX-V components and the events within the timing simulator.

- Prediction of execution time: The simulator was extended to incorporate timing characteristics for the aforementioned components, enabling the prediction of the required clock cycles for program execution. This extension utilized an event-driven methodology to model the associated latencies in an accurate manner.

Based on the above, we can conclude that the main contributions of the thesis are as follows:

- Extension of the functional simulator: The functional simulator of NEOX-V was expanded to incorporate timing information, enabling the clock cycles to be extracted for a given input program.
- Implementation of event-driven simulation: The utilization of an event-driven simulation method ensures a lightweight approach, avoiding complex solutions that may introduce significant overhead to the system.
- Accuracy and efficiency improvements: The accuracy of the new simulator remains below 8% for all applications, while the actual simulation time only increased by 0.7%.
- Development of a flexible validation framework: An extensive validation framework was established, facilitating the evaluation of not only the current version of the simulator but also future iterations and enhancements.
- Feedback mechanism: The new simulator serves as a valuable feedback mechanism for system designers, aiding in the evaluation and optimization of the NEOX-V architecture and associated components.

In summary, this thesis project has made significant contributions to the field by building an event-driven timing simulator for NEOX-V. By accomplishing the established goals and contributing these key advancements, it successfully addressed the research question and achieved a significant improvement in incorporating timing characteristics into the functional simulator. These contributions advance the understanding and analysis of timing behavior for an embedded hybrid GPU-AI accelerator and pave the way for further research and development in this area. Finally, it is important to mention that the developed timing model has already been included in the product offerings of Think Silicon S.A.

6.3. Future work

The future work for this thesis involves several areas of enhancement and expansion for the simulator. These include incorporating additional hardware components, improving modeling accuracy, supporting a multicore implementation, and developing a power estimator.

One important aspect is to extend the simulator to include support for additional hardware components, particularly those relevant to graphics applications. This includes modeling components such as the rasterizer, texture unit, and constant caches. By incorporating timing information for these graphics-related components, the simulator can accurately simulate graphics tasks and broaden its applicability to a wider range of applications.

Furthermore, while the current version of the simulator demonstrates high accuracy with an error lower than 8% for all test cases, there are still certain functions that have not been fully modeled. These include main memory bank conflicts, register bank conflicts, and line buffers. Integrating these functions into the timing model would introduce a fine-grained level of modeling to the simulator, potentially enhancing its accuracy and expanding its capabilities. Future work could involve incorporating these additional functions to further improve modeling accuracy and provide a more comprehensive simulation experience.

Additionally, there is potential for expanding the simulator to support a multicore implementation. By incorporating the capability to simulate multiple cores and their interactions, the simulator would provide useful predictions for NEOX-V which is a multicore accelerator.

Another area of future work involves developing a power estimator for the simulator. By integrating power modeling, the simulator could provide insights into the power consumption of the modeled system, enabling power-aware design and optimization.

By addressing these areas for future improvement, the thesis project can continue to advance the functional simulator, broaden its scope, and increase its accuracy in modeling various components and scenarios.

References

[1] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf. *Readings in Hardware/Software Co-Design*. Morgan Kaufmann Publishers Inc., 2002. ISBN: 9781558607026.

[2] Robert S. French. “A General Method for Compiling Event-Driven Simulations”. In: *32nd Design Automation Conference*. 1995, pp. 151–156. DOI: 10.1109/DAC.1995.250080.

[3] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. “Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation”. In: *SC ’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–12. DOI: 10.1145/2063384.2063454.

[4] Ali Bakhoda et al. “Analyzing CUDA workloads using a detailed GPU simulator”. In: *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 2009, pp. 163–174. DOI: 10.1109/ISPASS.2009.4919648.

[5] V.M. del Barrio et al. “ATTILA: a cycle-level execution-driven simulator for modern GPU architectures”. In: *2006 IEEE International Symposium on Performance Analysis of Systems and Software*. 2006, pp. 231–241. DOI: 10.1109/ISPASS.2006.1620807.

[6] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. “TEAPOT: A Toolset for Evaluating Performance, Power and Image Quality on Mobile Graphics Systems”. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS ’13. Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 37–46. ISBN: 9781450321303. DOI: 10.1145/2464996.2464999. URL: https://doi.org/10.1145/2464996.2464999.

[7] Doug Burger and Todd M. Austin. “The SimpleScalar Tool Set, Version 2.0”. In: *SIGARCH Comput. Archit. News* 25.3 (June 1997), pp. 13–25. ISSN: 0163-5964. DOI: 10.1145/268806.268810. URL: https://doi.org/10.1145/268806.268810.

[8] Nathan Binkert et al. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: https://doi.org/10.1145/2024716.2024718.

[9] Jason Lowe-Power et al. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: 2007.03152 [cs.AR].

[10] AMD. *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*. https://web.archive.org/web/20170619232736/http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf. 2013.

[11] AMD. *AMD Graphics Core Next (GCN) Architecture*. https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf. 2012.

[12] AMD. *Graphics Core Next Architecture, Generation 3*. https://www.amd.com/system/files/TechDocs/gcn3-instruction-set-architecture.pdf. 2016.

[13] Christian Menard et al. “System simulation with gem5 and SystemC: The keystone for full interoperability”. In: *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2017, pp. 62–69. DOI: 10.1109/SAMOS.2017.8344612.

[14] Maximilien Breughe, Stijn Eyerman, and Lieven Eeckhout. “A mechanistic performance model for superscalar in-order processors”. In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. 2012, pp. 14–24. DOI: 10.1109/ISPASS.2012.6189202.

[15] Marcos Amaris et al. “A Simple BSP-based Model to Predict Execution Time in GPU Applications”. In: *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. 2015, pp. 285–294. DOI: 10.1109/HiPC.2015.34.

[16] Marcos Amarís et al. “A comparison of GPU execution time prediction using machine learning and analytical modeling”. In: *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*. 2016, pp. 326–333. DOI: 10.1109/NCA.2016.7778637.

[17] Think Silicon. *Think Silicon*. 2023. URL: https://www.think-silicon.com/ (visited on 07/15/2023).

[18] R. Banakar et al. “Scratchpad memory: a design alternative for cache on-chip memory in embedded systems”. In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*. 2002, pp. 73–78. DOI: 10.1145/774789.774805.

[19] John F. Hughes et al. *Computer graphics: principles and practice (3rd ed.)* Boston, MA, USA: Addison-Wesley Professional, July 2013, p. 1264. ISBN: 0321399528.

[20] Kevin Galligan. *Finding Mona Lisa in the Game of Life*. 2023. URL: https://kevingal.com/blog/mona-lisa-gol.html (visited on 09/05/2023).

[21] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. UCB/EECS-2014-54. EECS Department, University of California, Berkeley, May 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html.

[22] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. “Understanding of a convolutional neural network”. In: *2017 International Conference on Engineering and Technology (ICET)*. 2017, pp. 1–6. DOI: 10.1109/ICEngTechnol.2017.8308186.

[23] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid Nawab. *Signals & Systems (2nd Ed.)* USA: Prentice-Hall, Inc., 1996. ISBN: 0138147574.

[24] Vincent Dumoulin and Francesco Visin. “A guide to convolution arithmetic for deep learning”. In: (Mar. 2016).

[25] Gordon E. Moon et al. *Evaluating Spatial Accelerator Architectures with Tiled Matrix-Matrix Multiplication*. 2021. arXiv: 2106.10499 [cs.DC].

[26] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors, Third Edition: A Hands-on Approach*. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128119861.

[27] Varun Chandola, Arindam Banerjee, and Vipin Kumar. “Anomaly Detection: A Survey”. In: *ACM Comput. Surv.* 41.3 (July 2009). ISSN: 0360-0300. DOI: 10.1145/1541880.1541882. URL: https://doi.org/10.1145/1541880.1541882.

[28] Guansong Pang et al. “Deep Learning for Anomaly Detection”. In: *ACM Computing Surveys* 54.2 (Mar. 2021), pp. 1–38. DOI: 10.1145/3439950. URL: https://doi.org/10.1145%2F3439950.

[29] Divya D. and Suvanam Sasidhar Babu. “Methods to detect different types of outliers”. In: *2016 International Conference on Data Mining and Advanced Computing (SAPIENCE)*. 2016, pp. 23–28. DOI: 10.1109/SAPIENCE.2016.7684114.

[30] Òscar Lorente, Ian Riera, and Aditya Rana. *Image Classification with Classic and Deep Learning Techniques*. 2021. arXiv: 2105.04895 [cs.CV].

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[32] Iván López-Espejo et al. *Deep Spoken Keyword Spotting: An Overview*. 2021. arXiv: 2111.10592 [cs.SD].

[33] Assaf Hurwitz Michaely et al. “Keyword spotting for Google assistant using contextual speech recognition”. In: *2017 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. 2017, pp. 272–278. DOI: 10.1109/ASRU.2017.8268946.

[34] Aakanksha Chowdhery et al. “Visual Wake Words Dataset”. In: *CoRR* abs/1906.05721 (2019). arXiv: 1906.05721. URL: http://arxiv.org/abs/1906.05721.