



THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.

This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Simulation Methodologies for Mobile GPUs

Kuba Kaszyk

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2021

Abstract

Graphics Processing Units (GPUs) critically rely on a complex system software stack comprising kernel- and user-space drivers and Just-in-time (JIT) compilers. Yet, existing GPU simulators typically abstract away details of the software stack and GPU instruction set. Partly, this is because GPU vendors rarely release sufficient information about their latest GPU products. However, this is also due to the lack of an integrated CPU/GPU simulation framework, which is complete and powerful enough to drive the complex GPU software environment. This has led to a situation where research on GPU architectures and compilers is largely based on outdated or greatly simplified architectures and software stacks, undermining the validity of the generated results. Making the situation even more dire, existing GPU simulation efforts are concentrated around desktop GPUs, making infrastructure for modelling mobile GPUs virtually non-existent, despite their surging importance in the GPU market. Still, mobile GPU designers are faced with the challenge of evaluating design alternatives involving hundreds of architectural configuration options and micro-architectural improvements under tight time-to-market constraints, to which currently employed design flows involving detailed, but slow simulations are not well suited. In this thesis we develop a full-system simulation environment for a mobile platform, which enables users to run a complete and unmodified software stack for a state-of-the-art mobile Arm CPU and Mali Bifrost GPU powered device, achieving 100% architectural accuracy across all available toolchains. We demonstrate the capability of our GPU simulation framework through a number of case studies exploring modern, mobile GPU applications, and optimize them using functional simulation statistics, unavailable with other approaches or hardware. Furthermore, we develop a trace-based performance model, allowing architects to rapidly model GPU configurations in early design space exploration.

Lay Summary

Computers are found all around us, from large data centers, to mobile phones, TVs, cars, refrigerators, and many other places. These computers, are made up of multiple components, one of which, is a Graphics Processing Unit (GPU). GPUs are the component responsible for drawing images to your screen, however, in recent years, they have also been used to do similar tasks to the CPU. This thesis focuses primarily on mobile systems, which typically include any system that has a battery and can operate away from its main power source.

Designing new computer systems, including GPUs, is expensive and time consuming, especially if things go wrong. It's therefore critically important to model as many components as possible, before actually fabricating the chips. One of the most important modelling tools is simulation, which means that you write a program which behaves just like the hardware component that you want to model. This program can then be used to identify any problems with the design, and to predict its performance.

Just like the entire design process, simulation has its challenges. For example, GPUs operate in a complex environment, and it can be very difficult to re-create a realistic environment to simulate a GPU. The first goal of this thesis is to examine and develop techniques which would allow us to model GPUs in the exact same environment as they ultimately execute in. We achieve this by simulating not only the GPU, but the entire surrounding system.

Another challenge is simulation speed. Programs written to simulate hardware are often excruciatingly slow, as they have to model a lot of internal components, especially if you have to model the entire system, as we do when simulating GPUs. One option to speed up the simulation, is to limit the amount of detail modelled. The second goal of this thesis is to ensure that modelling the full environment that a GPU executes in does not slow the simulation down.

Reducing the level of detail is often discounted as a technique, because it limits the amount of useful information that can be obtained from the simulation. However, the final goal of this thesis is to ensure that even with the additional cost of simulating the GPU's environment, and with the loss of detail when accelerating the simulation, there is still useful performance information that can be extracted from the simulation framework. We achieve this goal by splitting the simulation into two phases - the first one collecting information, and the second analyzing it.

Acknowledgements

There are a number of people I would like to thank for their support during my time as a PhD student, starting with my supervisor, Björn Franke, for getting me involved in the topic years before I started my PhD, and his irreplaceable guidance over the years. I would also like to thank my second supervisor, Mike O'Boyle, for the feedback, and many valuable discussions.

Deserving a special mention are Harry Wagstaff and Tom Spink, who laid the groundwork for my own research; Bruno Bodin, for the excitement and mentorship he brought to the project; and Chris Vasiladiotis and Calum Imrie, for their excellent company and collaboration.

Finally, I would like to thank my family, in particular my wife, Roksana, and my daughters, Julia and Sofia, for the patience and understanding they have shown me.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Kuba Kaszyk)

Publications

The following publications have been made during the course of this PhD, some of which are used as the basis for chapters:

- **Kuba Kaszyk**, Harry Wagstaff, Tom Spink, Björn Franke, Michael O’Boyle, Bruno Bodin, Henrik Uhrenholt

“Full-System Simulation of Mobile CPU-GPU Platforms”

In Proceedings of the 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’19), Madison, Wisconsin, February 2019

This publication forms the basis for Chapter 4.

- Valentin Radu, **Kuba Kaszyk**, Yuan Wen, Jack Turner, Josè Cano, Elliot J Crowley, Björn Franke, Amos Storkey, Michael O’Boyle

“Performance aware convolutional neural network channel pruning for embedded GPUs”

In Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC’19), Orlando, Florida, November 2019

This publication forms the basis for case studies presented in Chapter 4.

- **Kuba Kaszyk**, Björn Franke

“Full-System GPU Design Space Exploration”

Workshop on Modeling & Simulation of Systems and Applications (ModSim 2020)

This publication forms the basis of the machine-learning based approach presented in Chapter 5.

- Subhankar Pal, **Kuba Kaszyk**, Siying Feng, Björn Franke, Murray Cole, Michael O’Boyle, Trevor Mudge, Ronald G. Dreslinski

“HETSIM: Simulating Large-Scale Heterogeneous Systems using a Trace-driven, Synchronization and Dependency-Aware Framework”

In Proceedings of the 2020 IEEE International Symposium on Workload Characterization (IISWC’20)

Table of Contents

1	Introduction	1
1.1	The Past, Present, and Future of GPUs	1
1.2	The Complexities of GPU Systems	4
1.3	Simulation as a Key Design and Development Tool	5
1.4	Motivation	7
1.5	Goals	8
1.6	Thesis Overview	8
2	Background	11
2.1	Introduction	11
2.2	How do we program a GPU?	11
2.2.1	The Software Stack	12
2.2.2	Low-Level Programming Frameworks	15
2.2.3	High-Level Libraries	21
2.3	The Arm Mali Bifrost Architecture	23
2.3.1	Overview	24
2.3.2	Starting a GPU Job	25
2.3.3	Interaction with the CPU	25
2.3.4	Details of the Job Manager	25
2.3.5	The Arm Mali Memory System	26
2.3.6	The Bifrost Shader Core	28
2.3.7	The Clause Based Execution Model	29
2.3.8	Graphics Hardware	32
2.4	GenSim - A Head Start on Fast Simulation	32
2.4.1	ArcSim	37

2.5	Captive - Making the most of host hardware	38
2.6	Validating Simulation Through Benchmarking	38
2.6.1	Parboil	39
2.6.2	Polybench	39
2.6.3	AMD APP SDK 2.5	40
2.6.4	Rodinia	40
2.6.5	SLAMBench	41
2.6.6	DeepSmith	41
2.6.7	SGEMM	41
2.6.8	Convolutional Neural Network Channel Pruning	44
2.7	Generating New Benchmarks	46
2.7.1	Measurement	47
2.8	Conclusion	48
3	Simulation Background & Related Work	55
3.1	Speed vs. Detail	55
3.1.1	Cycle-Accurate Simulation	56
3.1.2	Functional Instruction Set Simulation	58
3.1.3	Emulation	59
3.2	Simulation Environments	60
3.2.1	User Mode Simulation	60
3.2.2	Full-System Simulation	60
3.3	Modelling the Software Stack	61
3.3.1	Accuracy of the Simulated Software Stack	62
3.3.2	Speed of Simulation	71
3.3.3	Ease of maintenance as software is updated	72
3.3.4	Usability	73
3.3.5	Comparison Against Existing Hardware and Software	74
3.4	Performance Modelling Techniques	76
3.4.1	Cycle-Accurate Simulation	76
3.4.2	Trace Based Simulation	76
3.4.3	Analytical Modelling	78
3.4.4	Machine Learning and Statistical Modelling	82
3.5	Conclusion	85

4	Full-System Simulation of Mobile CPU/GPU Platforms	87
4.0.1	State-of-the-Art	88
4.0.2	Contributions	90
4.1	Our Simulation Approach	91
4.1.1	CPU Simulation	92
4.1.2	GPU Simulation	92
4.2	Instrumentation	94
4.2.1	Program Execution	95
4.2.2	System	95
4.2.3	Control Flow	95
4.3	Validation and Quantitative Evaluation	95
4.3.1	Validation and Accuracy	96
4.3.2	Simulation Performance	98
4.4	Qualitative Evaluation	100
4.4.1	Accuracy of the Software Stack	101
4.4.2	Speed of Simulation	103
4.4.3	Ease of maintenance	103
4.4.4	Usability & Flexibility	103
4.5	Application Results	104
4.5.1	Identifying Empty Instruction Slots on the GPU	105
4.5.2	Moving Data Closer to the Core	105
4.5.3	Evaluating the Bifrost Clause Model	106
4.5.4	System Level Results	107
4.6	Optimizing OpenCL Applications	107
4.6.1	SLAMBench	107
4.6.2	SGEMM	108
4.6.3	Performance Aware Convolutional Neural Network Channel Pruning	109
4.7	Implementation Details	122
4.7.1	User-Mode Simulation (CPU)	122
4.7.2	GenC Thumb2 Model	122
4.7.3	GenC GPU Model	122
4.7.4	Standalone GPU Hexdump Simulator	124
4.7.5	Fuzz Testing	124
4.7.6	GenSim & Captive Integration	127

4.8	Summary & Conclusion	128
5	Fast Performance Modelling	131
5.1	Motivation	131
5.1.1	The Arm Mali GPU	133
5.2	Prediction Using Machine Learning	134
5.2.1	A Näive Machine Learning Approach	134
5.2.2	Feature Selection	135
5.2.3	Principal Component Analysis	136
5.2.4	Manual Feature Modification	136
5.2.5	An Artificial Neural Network Model	137
5.2.6	Conclusions	137
5.3	REASSEMBLE: A Trace Based Approach to Fast Performance Modelling	140
5.3.1	Design and Methodology	141
5.3.2	Validation	144
5.3.3	Comparison Against State-of-the-Art	147
5.3.4	Design Space Exploration	151
5.3.5	Critical Evaluation	158
5.3.6	Recent Developments	165
5.3.7	Conclusion	167
5.4	Summary	168
6	Conclusions	169
6.1	Contributions	170
6.1.1	Full-System GPU Simulation	170
6.1.2	Fast Performance Modelling	170
6.2	Current Limitations	172
6.2.1	Significant changes to GPU architectures	172
6.2.2	GenSim and Captive Limitations	172
6.2.3	Limitations of the GPU Model	173
6.2.4	Software Availability Assumptions	173
6.3	Future Work	173
6.3.1	Functional Simulation	174
6.3.2	Performance Modelling	174
6.3.3	Power and Area Modelling	175

6.3.4	Applications of Functional Simulation and Fast Performance Modelling	176
6.4	The Future of Simulation	177
6.5	Summary and Final Remarks	177
A	Appendix	179
	Bibliography	183

Introduction

1.1 The Past, Present, and Future of GPUs

GPUs have become ubiquitous in today's computing world. They have grown from being fixed function blocks specific to computing graphics workloads, to general purpose parallel compute accelerators, found in devices ranging from TVs, smartphones and drones to supercomputers, and are being further specialized for accelerating workloads in specific domains. Nvidia, often considered the inventor of the GPU, currently lists 52 different categories whose applications target GPUs, including but not limited to Animation, Data Mining, Bioinformatics, Climate Modeling, Computational Fluid Dynamics, Databases, Computer Vision, and Machine Learning [1], showing just how pervasive GPUs are in modern day computer science.

The first graphics accelerators emerged in the 1970s, with the introduction of video shifters in various arcade systems. The 1980s saw the introduction of fully-integrated graphics display processors for PCs, and in 1986, Texas Instruments released the first fully programmable graphics processor - the TMS34010 [2]. The TMS34010 was essentially a 32-bit CPU with some graphics-oriented instructions, and brought with it the first generalized architecture for graphics processing - *Texas Instruments Graphics Architecture (TIGA)* [3], which guaranteed that a program written for TIGA, would work on any TIGA-compatible machine. The late 1980s and 90s saw the emergence of the first dedicated video cards for PCs used to accelerate fixed-function 2D primitives, with increasing numbers of colors and pixels supported. By the mid-1990s, 3D graphics had found their way into gaming consoles. At the same time, the OpenGL [4] and Glide [5] APIs appeared, and slowly hardware support for OpenGL was rolled out. Towards the late 1990s, DirectX became another popular choice for programming

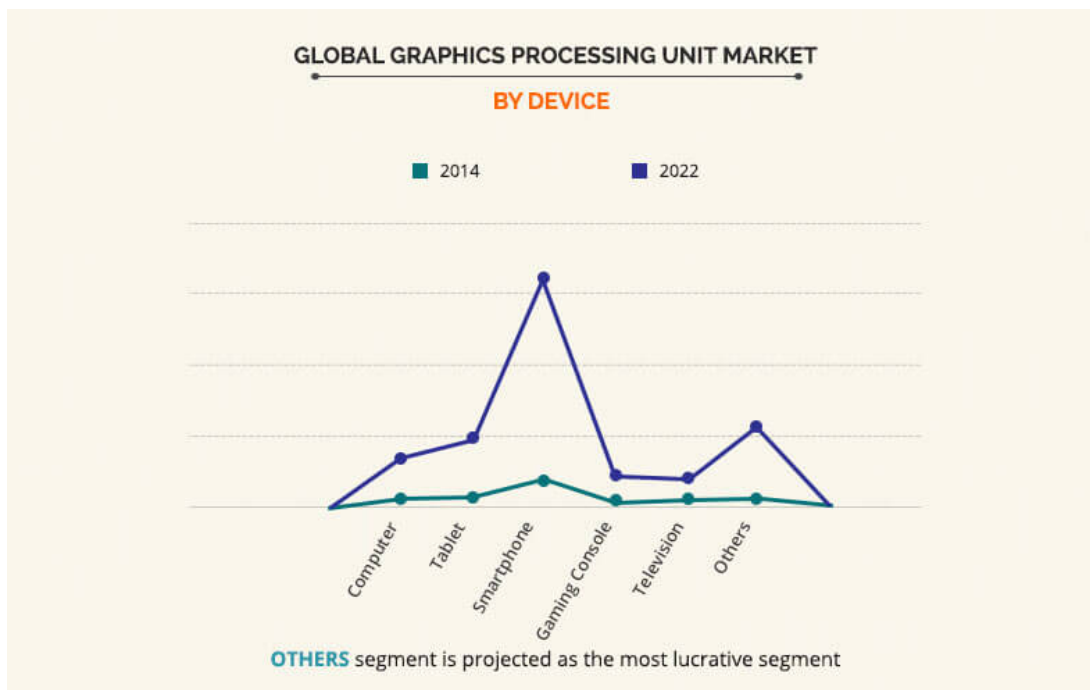


Fig. 1.1: GPU market breakdown by device [10].

graphics. In the early 2000s, Nvidia produced the first chip supporting programmable shading, the GeForce 3, which was used in the original Xbox gaming console [6]. The shading however, was still focused largely around dispatching fixed-function units of work. The Nvidia GeForce 8 series GPUs became the first generalized computing devices and represent Nvidia's first unified shader architecture [7]. This meant that all shaders could handle any type of shading task. In 2007, Nvidia introduced the CUDA platform [8], the earliest widely adopted programming model for GPU computing. This was followed by OpenCL [9]- an open standard developed and supported by many hardware vendors, with Apple breaking away and releasing Metal, its own compute and graphics API. In the last decade, GPUs have become increasingly powerful, with graphics driven by gaming and virtual reality, and compute driven by big data and the emergence of machine learning and computer vision. According to Allied Market Research, the GPU market size is expected to reach \$157.1 billion by 2022, growing by 35.6% during the 2016-2022 period [10]. Figure 1.1, sourced from the same report, breaks down the GPU market by device. It shows, that, as a percentage, the *others* category will have the largest growth as a percentage of its current market share, while *smartphones* are and will remain the largest category by volume. Jointly, smartphones and other, newly-developed emerging devices, will far outpace the remaining categories by 2022.

The mobile GPU space was until recently dominated by Arm, whose GPUs powered 52.6% of smartphones sold in 2018, thanks to strong partnerships with Samsung and Huawei. The mobile GPU space however, is evidently of interest to many leading companies. Samsung, a leader in the field of mobile devices, has recently partnered with AMD to create their own in-house GPU, using AMD intellectual property (IP) for a scalable GPU design [11]. For many years, Apple licensed its GPU IP from Imagination Technologies, and after a brief two-year period of attempting to design GPUs in house, Apple has once again signed an agreement with Imagination to source future graphics IP [12]. This announcement comes shortly after Imagination announced its new, scalable, GPU-of-everything family [13], showing that Imagination GPUs can target a wide variety of devices and domains. Furthermore, Apple is shifting its technology away from separate chips, having recently announced its first System-on-Chip (SoC)-based computers with the M1 chip. The M1 chip, similarly to smartphones, integrates the CPU, GPU, and Neural Processor into a single SoC, sharing memory between all processing units, and delivering a record performance per watt, and outperforming high-end Intel processors in raw performance measurements.

The market for AI augmentation, which requires compute and graphics processing locally on device, is estimated to be worth 3.3 trillion dollars [14]. Coupled with concerns about security and privacy, much of the processing is moving from the cloud to edge devices. Driven by this unique set of circumstances, vendors are specializing their embedded GPUs, optimizing not only for graphics, but highly-parallel computer vision and machine learning applications as well. Computer vision accelerators have been developed by companies such as Movidius (now part of Intel), and many companies, including Arm and Huawei, are increasingly introducing GPUs optimized for machine learning, and even adding additional, dedicated chips with the sole purpose of accelerating machine learning algorithms [15].

GPUs are found all around us - from servers, desktops, and game consoles, to small mobile GPUs in mobile phones and VR headsets - and they're continuously being re-designed and altered to fit emerging applications that require vast parallel compute capabilities. Figure 1.4 shows the number of vendors designing new machine learning accelerators alone. Technology is quickly migrating from being server/desktop dominated into the IoT (Internet-of-Things) space, meaning that energy efficient processors are in high demand. In the future, we'll see more and more demand for these types of systems, and it's critical that we have tools to support their design and development.

Very little of the original graphics-specific GPU design remains visible in modern

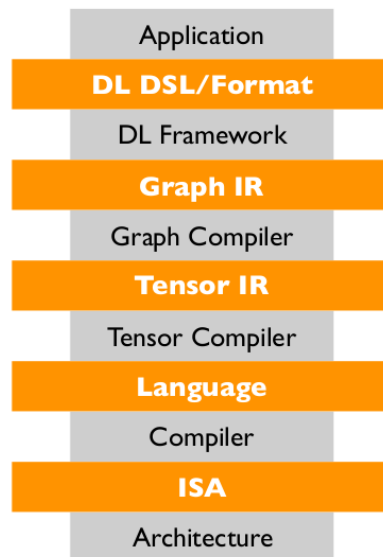


Fig. 1.2: Modern deep learning compute stacks include multiple libraries, with different purposes, and using different intermediate representations. (Fig. Source: [16])

multi-purpose parallel GPGPU architectures. GPUs however, have not evolved on their own. The applications accelerated by GPUs are supported by exceptionally complex heterogeneous systems, with multiple processors, and convoluted software stacks driving the applications, which a standalone GPU would not be capable of executing. The intricacies of systems supporting GPU execution are explored in the following section.

1.2 The Complexities of GPU Systems

Modern GPUs form just one component of complex, heterogeneous systems, with a CPU, and potentially other accelerators executing in parallel, sharing resources, and completing mutual tasks, and consideration for this environment must be at the heart of the design. The systems themselves are growing increasingly complex, with additional levels of the software stack both co-existing and being layered one on top of the next.

The original GPU compute stack has grown to include multiple abstractions and libraries in the software layer, for example, Figure 1.2 shows an abstraction of the deep learning compute stack. A neural network is specified in a high-level programming language within a deep learning framework, which is then compiled down into an intermediate graph representation, which forms the network. The operations performed on this network are further compiled into Tensor IR, which is a matrix representation of the operations to be executed. Only then, these intermediate representations are trans-

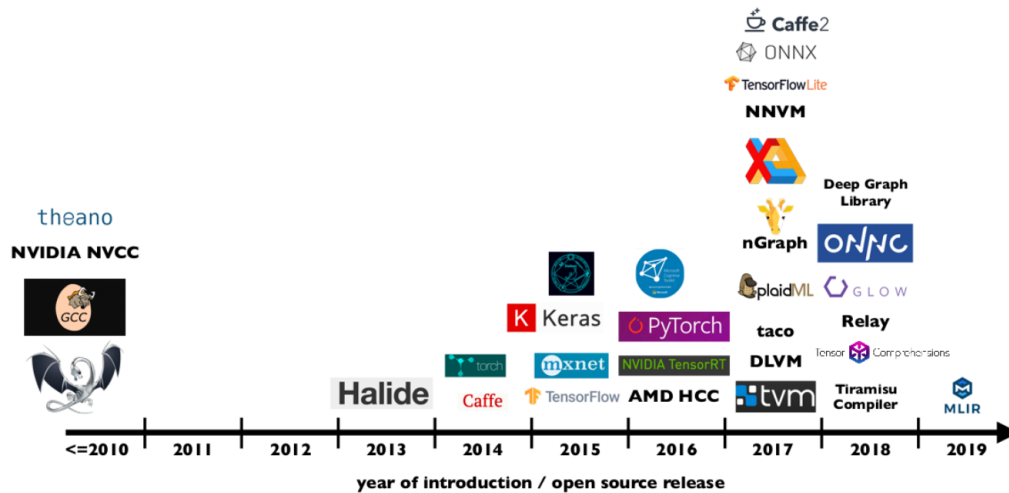


Fig. 1.3: The number of Machine Learning frameworks has grown dramatically in recent years. (Fig. Source: [16])

formed into a more traditional programming language, for example C++, from which point the traditional compute stack, forming the language, compiler, ISA, and architecture, is once again recognizable. But this is just one example - Figure 1.3 shows how the number of programming frameworks for machine learning has exploded in recent years.

The software stack doesn't execute just on the GPU. In fact, only the final, optimized, binary GPU kernel executes on the GPU, while the remainder of the complex software stack executes entirely on the CPU, demonstrating just how critical the CPU is to correct and efficient GPU execution - without CPU execution, there is no GPU execution. A CPU further relies on other system components for correct execution - timers, interrupt controllers, memory. The only way to completely and correctly execute the entire software stack, is to have a complete view of the hardware that the software executes on. This is true not just for native execution, but also for all tools used to design and develop GPUs and their software stacks. The following section introduces simulation - the backbone behind the development of modern GPU devices.

1.3 Simulation as a Key Design and Development Tool

At the heart of GPU design tools, lies simulation. Simulation is a critical component of any computer architecture design phase. It is increasingly used for designing, prototyping, and testing new hardware. Simulation is also used by systems, compiler, and

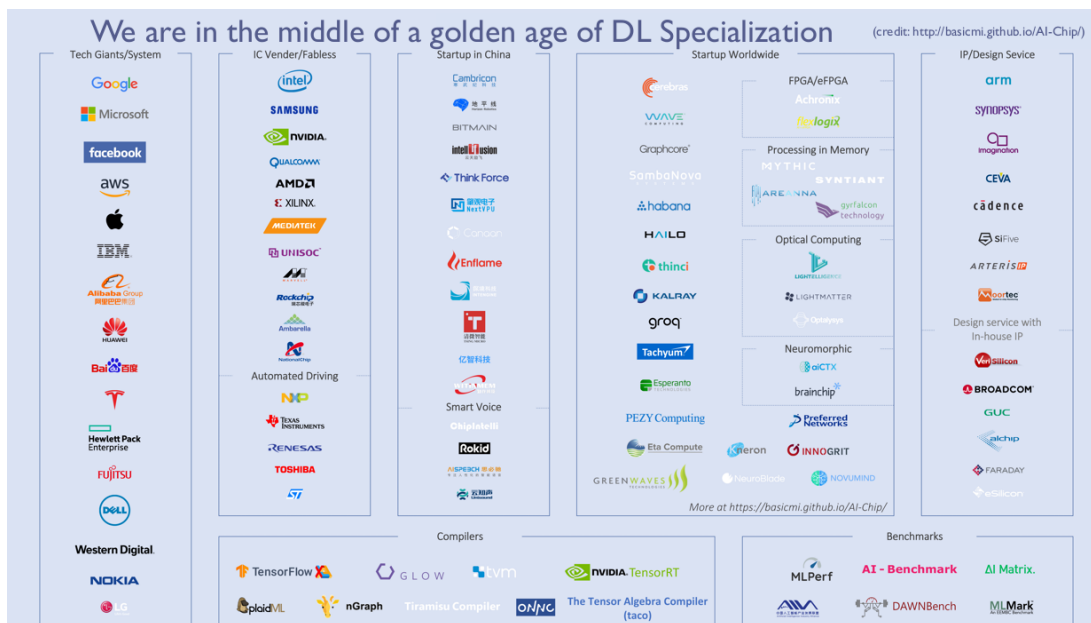


Fig. 1.4: Both established vendors and startups are designing machine learning accelerators. (Source: [16])

software developers to design software before the silicon is available, and can further be exploited to explain causes and effects which can't be observed using real hardware. This thesis primarily concerns itself with two types of simulators - functional, instruction set simulators, and cycle-accurate simulators. Functional, instruction set simulators model the architecture at the ISA-level, executing instruction-by-instruction, without any overview of the micro-architecture. Cycle-accurate simulators on the other hand, follow each instruction through all of the pipeline stages, and at the cost of simulation speed, have greater observability.

The primary use of simulators is two-fold. Firstly, by computer architects, they are used to model hardware prior to developing physical chips. In this case, strong focus is placed on modeling hardware components, and their interactions, in order to accurately predict hardware power consumption, performance, and area. Most commonly for this purpose, cycle-accurate simulators are used by computer architects.

Secondly, simulators are used by software developers, who use them to develop application and system software prior to hardware becoming available, or using them in environments where a high number of potential configurations is possible, and where it would be infeasible to obtain and test all available hardware platforms (e.g. Android app and system development). In this case, a stronger focus is placed on functional correctness, and performance prediction, while still important, becomes secondary.

Commonly, in software development, functional emulators, or instruction set simulators are used, which trade-off accuracy in the performance model for execution speed.

The following section introduces some of the trade-offs that need to be considered when simulating GPUs, as well as limitations of existing GPU simulators.

1.4 Motivation

In both of the scenarios described in the previous section, speed is of utmost importance. When designing hardware, there are infinitely many possible configurations, and with the high cost of detailed simulation, only a limited number of configurations can be explored within the timeframe of the development cycle. Software developers face the problem of long latencies when running simulations and testing different software implementations. For example, a software developer may want to test a new piece of software, and executes it in the simulator. In real hardware, this execution might take a few hundred milliseconds, however in functional simulation, it could take hours, and in detailed simulation, days, weeks, or months. Now let's say this software contains a number of bugs that need to be discovered, and addressed. This is an iterative process, and each bug requires another execution of the software, significantly extending development time.

GPUs are particularly vulnerable to the issues mentioned, as they are highly-parallel SIMD machines designed to accelerate parallel workloads. In short, in the same amount of time as a CPU, they are capable of executing thousands of times more code, provided that the application is parallelizable. As the host hardware (CPU) that the GPU simulator is executing on does not have the same parallel capabilities as the guest (GPU), this has a significant impact on simulation time. Detailed architecture simulation clearly does not scale to increasingly parallel modern workloads.

Furthermore, in order to execute modern applications with a complex software stack, and to guarantee accurate binary execution, the supporting software stack must also be faithfully simulated in full.

Existing state-of-the-art GPU simulators however, primarily focus on detailed modelling of the hardware, while cutting many corners when simulating the software stack and surrounding system by treating the GPU as a standalone device, simulating the GPU at levels other than the binary instruction set level, or replacing the software stack. In addition, simulation speeds associated with detailed simulators make these simulators unsuitable for modelling modern workloads. Finally, existing state-of-the-

art GPU simulators in the open domain are developed around large desktop or server class GPUs, while leaving a void in the mobile GPU space. These problems are explored in detail in Section 3.3, while the following section introduces the goals behind the work presented in this thesis.

1.5 Goals

The work presented in this thesis aims to provide better simulation technology for GPU-based systems that cut fewer corners, but are faster than existing cycle-accurate simulators, and which fully supports mobile GPU simulation. In particular, we aim to develop a simulation system that:

1. Accurately simulates a state-of-the-art mobile GPU in a full-system context, enabling the use of *unmodified* vendor-supplied drivers and JIT compilers, operating in an *unmodified* target operating system, and executing *unmodified* applications.
2. Supports simulation speeds, which enable the user to execute complete and complex applications typical of modern GPU workloads.
3. Provides useful performance statistics, without the overhead of cycle-accurate simulation.

The following section outlines how the goals of the thesis are achieved.

1.6 Thesis Overview

With increasingly complex architectures, software stacks, and systems, it is in the interest of all to reduce simulation times, while maintaining an accurate representation of the system and software stack. This thesis investigates the above issues and solutions through compute applications executing on graphics processing units (GPUs) as a platform and application of interest in the following manner.

First, we provide an overview of background information in Chapter 2, and an in-depth exploration of existing problems and solutions in the GPU simulation space in Chapter 3.

After motivating the problems faced in GPU simulation, we present our holistic solution to modelling GPUs as components of a complete system, allowing us to accurately model the execution of the entire software stack executing both on the CPU and GPU, which is presented in Chapter 4. Additionally, Section 4.6 demonstrates applications of the simulator in OpenCL program optimization, by using detail from the functional simulation to optimize OpenCL applications and explain unexpected performance behaviour in hardware.

As cycle-accurate simulation of large GPU applications is infeasible using current techniques, in Chapter 5, we explore different approaches to fast performance modelling, including machine-learning based approaches and trace-based simulation. Building on our holistic approach, we present a novel tuning and trace-based performance modelling framework, which delivers accuracy on-par with existing cycle-accurate simulators, without compromising on simulation speed. Once again, in Section 5.3.4, we present use cases of our simulation framework, through an extensive design space exploration made possible by our work. We conclude, and suggest future work in Chapter 6.

Background

2.1 Introduction

This thesis builds on a significant amount of prior work in the fields of architecture, programmability, and simulation. This chapter introduces the background concepts necessary for understanding the content of Chapters 3, 4, and 5. First, Section 2.2 introduces the GPU programming model, presenting the concepts behind OpenCL and other popular programming frameworks. Next, Section 2.3 describes the Arm Mali Bifrost architecture, which is the architecture modelled in the work leading to this thesis. The GPU simulation framework we develop originates from, and uses components of existing simulation frameworks, which are described in Section 2.4 and Section 2.5. Section 2.6 describes the benchmarks used for extensive validation of and experimentation using the developed GPU simulation framework, and finally, Section 2.7 presents strategies for collecting data from hardware.

2.2 How do we program a GPU?

GPUs are not standalone devices, and therefore programming a GPU relies on a specific protocol established between the programmer, CPU, and GPU, often defined in the form of a programming framework. The programmer first uses CPU code to prepare the data to be consumed, and code to be executed on the GPU - however, even this isn't always transparent to the modern day GPU programmer. This section presents the protocol established by programming frameworks such as OpenCL, and introduces higher-level programming frameworks where the programmer may not even be aware that they are executing GPU code. Just as in physical hardware, this software stack

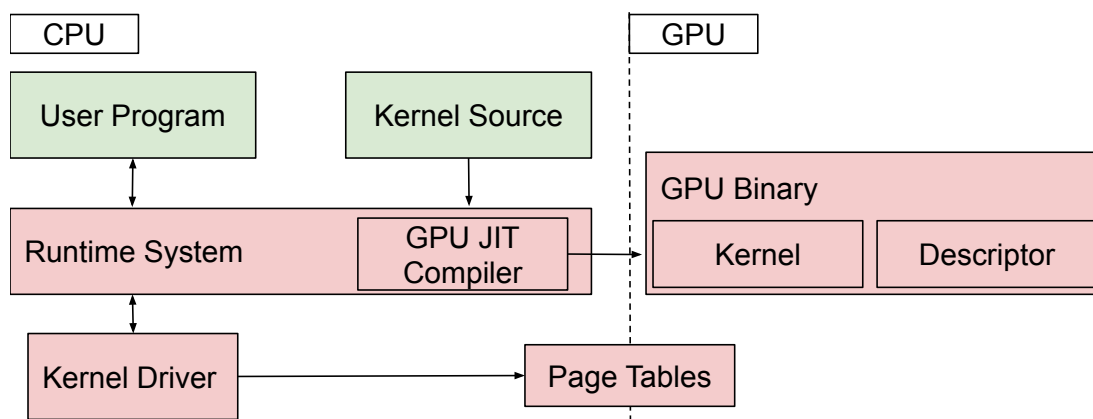


Fig. 2.1: A high level overview of a interactions between different components of a typical GPU software stack. User-provided programs are in green, and the supporting software stack and resulting binaries are coloured red.

must be faithfully executed in simulation in order to achieve completeness and correctness.

2.2.1 The Software Stack

Figure 2.1 presents the typical components of a software stack required to execute programs on the GPU. These comprise the user program, a runtime system, the kernel device driver, and finally, the native GPU binary.

2.2.1.1 User Program

Traditionally, the user program was a C or C++ program with API calls to the relevant programming framework, such as OpenCL, or CUDA. There are a number of frameworks with similar features, so while we provide examples for a number of frameworks, we only present OpenCL in detail. A sample host-side OpenCL program is shown in Listing 2.1 and Table 2.1 lists the function calls exposed by the OpenCL API.

OpenCL host programs typically use a specific pipeline of operations that are similar across all OpenCL programs. Firstly, the programmer must identify the available platforms using the `clGetPlatformIDs` call. The platform specifies a vendor, name, version, and available extensions. Based on the platform, the API also allows the programmer to identify the devices in that platform, using the `clGetDeviceIDs` call, which is the next step. Next, an OpenCL context is created, which keeps track of references and manages all resources used during execution. This is followed by the creation of

clBuildProgram	clEnqueueMigrateMemObjects	clGetProgramInfo
clCompileProgram	clEnqueueNativeKernel	clGetSamplerInfo
clCreateBuffer	clEnqueueNDRangeKernel	clGetSupportedImageFormats
clCreateCommandQueue	clEnqueueReadBuffer	clLinkProgram
clCreateContext	clEnqueueReadBufferRect	clReleaseCommandQueue
clCreateContextFromType	clEnqueueReadImage	clReleaseContext
clCreateImage	clEnqueueTask	clReleaseDevice
clCreateKernel	clEnqueueUnmapMemObject	clReleaseEvent
clCreateKernelsInProgram	clEnqueueWriteBuffer	clReleaseKernel
clCreateProgramWithBinary	clEnqueueWriteBufferRect	clReleaseMemObject
clCreateProgramWithBuiltInKernels	clEnqueueWriteImage	clReleaseProgram
clCreateProgramWithSource	clFinish	clReleaseSampler
clCreateSampler	clFlush	clRetainCommandQueue
clCreateSubBuffer	clGetCommandQueueInfo	clRetainContext
clCreateSubDevices	clGetContextInfo	clRetainDevice
clCreateUserEvent	clGetDeviceIDs	clRetainEvent
clEnqueueBarrierWithWaitList	clGetDeviceInfo	clRetainKernel
clEnqueueCopyBuffer	clGetEventInfo	clRetainMemObject
clEnqueueCopyBufferRect	clGetEventProfilingInfo	clRetainProgram
clEnqueueCopyBufferToImage	clGetImageInfo	clRetainSampler
clEnqueueCopyImage	clGetKernelArgInfo	clSetEventCallback
clEnqueueCopyImageToBuffer	clGetKernelInfo	clSetKernelArg
clEnqueueFillBuffer	clGetKernelWorkGroupInfo	clSetMemObjectDestructorCallback
clEnqueueFillImage	clGetMemObjectInfo	clSetUserEventStatus
clEnqueueMapBuffer	clGetPlatformIDs	clUnloadPlatformCompiler
clEnqueueMapImage	clGetPlatformInfo	clWaitForEvents
clEnqueueMarkerWithWaitList	clGetProgramBuildInfo	

Table 2.1: A list of functions provided by the OpenCL API.

data structures using the `clCreateCommandQueue` and `clCreateBuffer` calls. Transfer of data to the new buffers is performed using the `clEnqueueWriteBuffer` call. Following this, the program binary is compiled using the `clCreateProgramWithSource`, `clBuildProgram`, and `clCreateKernel` API calls. These calls invoke the OpenCL compiler. Arguments are then communicated using the `clSetKernelArg` function. Up until this point, all of the code has been executed on the CPU, and only now, kernels are dispatched to the GPU using the `clEnqueueNDRangeKernel` call. The result of the kernel is read back from memory by the CPU using the `clEnqueueReadBuffer` call, after which all memory is freed.

In many modern programming frameworks there can be multiple layers of what we consider the user program. For example, the PyTorch framework integrates multiple libraries and abstracts away the low-level details of dispatching jobs to the GPU. In such frameworks the user simply specifies the operation, and optionally the target hardware, while the underlying libraries perform all necessary API calls. In the case

of a matrix multiplication, the underlying OpenCL implementation can come from the Arm Compute Library, or it can be generated by TVM, and the implementations can be vastly different from each other. Even more so than when using plain OpenCL, the vast majority of the code is executed on the CPU, with the GPU providing powerful acceleration for specific operations, showcasing how tightly coupled the CPU and GPU execution are. [17] demonstrates that a lot of the execution time is spent on CPU-side code, and aims to fuse GPU operations together in order to reduce CPU-GPU communication overhead.

2.2.1.2 Runtime Library

The runtime library implements the API called by the user program. The runtime library is linked against the host program binary and at execution time, calls are offloaded onto the runtime library. The runtime library performs all data management, kernel compilation, and communication with the GPU via lower-level kernel drivers. The runtime library itself is in most cases provided as a binary by the vendor, and executes on the CPU. This flow is depicted in Figure 2.1.

2.2.1.3 Kernel Device Driver

The device driver is a software component that is integrated into the operating system. In Linux, this is in the form of a kernel module. The device driver is responsible for memory allocation, reading from and writing to the GPU's memory-mapped registers, raising interrupts, and handling faults. The kernel driver also sets up the page tables with mappings from virtual to physical addresses. As the CPU and GPU operate using a shared memory model, the page tables can also be shared between the CPU and GPU. The kernel driver executes on the CPU and is usually provided by the vendor. In the case of the Mali Bifrost GPUs, Arm provides the source for the kernel driver so that it can be integrated into any Linux kernel.

2.2.1.4 GPU Binary

The GPU binary is the only part of the software stack that executes on the GPU, and is compiled by the OpenCL compiler, which executes on the CPU. The program can be compiled ahead of time and loaded by the OpenCL runtime, however in the majority of cases, the kernel source is JIT-compiled. The final binary contains both the kernel code, which is executed by the GPU cores, as well as a series of descriptors, which

specify meta- information necessary for executing the kernels. The descriptors include information such as the number of threads, the workgroup size, and pointers to where the compute kernel resides in the GPUs virtual memory.

2.2.2 Low-Level Programming Frameworks

Here we explore programming frameworks which directly expose the hardware communication between the CPU and the GPU to the end user. We consider these to be low-level programming frameworks in contrast with high-level programming frameworks, which hide away the details of GPU communication and execution.

2.2.2.1 OpenCL [18]

OpenCL is a parallel compute framework targeting heterogeneous platforms. Its most common targets are GPUs, but OpenCL can also be used to program CPUs, FPGAs, and any other hardware that complies with the OpenCL standard. OpenCL is developed by the Khronos Group, a non-profit consortium with dozens of members, including but not limited to Arm, AMD, Nvidia, Huawei, and Intel.

OpenCL is considered to be a low-level language, meaning that the programmer has direct interaction with a lot of hardware components. This is evident during setup and initialization, where the user has to declare and initialize memory buffers. A sample OpenCL program is presented in Listing 2.1.

2.2.2.2 OpenGL [19]

OpenGL is a graphics rendering framework, also developed by the Khronos Group. A sample OpenGL program is presented in Listing 2.2.

2.2.2.3 SYCL [20]

SYCL is a higher-level abstraction of OpenCL, aiming to increase programming productivity. In SYCL, not only is the kernel code embedded directly into the host code, but the compute kernel is also valid C++ code. This means that the code can still be executed on a CPU in the absence of OpenCL-compliant hardware, which is useful for portability and debugging. Similarly to OpenCL and OpenGL, it is developed by the Khronos Group. An example of SYCL code is presented in Listing 2.3.

Listing 2.1: OpenCL vector add example code.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <CL/opencl.h>
5
6  const char *kernelSource =
7      "#pragma OPENCL EXTENSION cl_khr_fp64 : enable\n" \
8      "__kernel void vecAdd( __global double *a, __global double *b, \n" \
9      "                        __global double *c, const unsigned int n) \n" \
10     "{\n" \
11     "    int id = get_global_id(0);\n" \
12     "    if (id < n)\n" \
13     "        c[id] = a[id] + b[id];\n" \
14     "}";
15
16 int main( int argc, char* argv[] ) {
17     unsigned int n = 100000;
18     cl_platform_id cpPlatform; // OpenCL platform
19     cl_device_id device_id; // device ID
20     cl_context context; // context
21     cl_command_queue queue; // command queue
22     cl_program program; // program
23     cl_kernel kernel; // kernel
24
25     size_t bytes = n*sizeof(double);
26     double * h_a = (double*)malloc(bytes);
27     double * h_b = (double*)malloc(bytes);
28     double * h_c = (double*)malloc(bytes);
29     [...] // init data
30
31     size_t localSize = 64;
32     size_t globalSize = ceil(n/(float)localSize)*localSize;
33
34     cl_int err = clGetPlatformIDs(1, &cpPlatform, NULL);
35     err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
36     context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
37     queue = clCreateCommandQueue(context, device_id, 0, &err);
38     program = clCreateProgramWithSource(context, 1,
39                                         (const char **) & kernelSource, NULL, &err);
40     clBuildProgram(program, 0, NULL, NULL, NULL, &err);
41     kernel = clCreateKernel(program, "vecAdd", &err);
42
43     cl_mem d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
44     cl_mem d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
45     cl_mem d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);
46     err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0, bytes, h_a, 0, NULL, NULL);
47     err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0, bytes, h_b, 0, NULL, NULL);
48     err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
49     err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
50     err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
51     err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
52
53     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, &localSize, 0, NULL, NULL);
54
55     clFinish(queue);
56     clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0, bytes, h_c, 0, NULL, NULL);
57
58     clReleaseMemObject(d_a);
59     clReleaseMemObject(d_b);
60     clReleaseMemObject(d_c);
61     clReleaseProgram(program);
62     clReleaseKernel(kernel);
63     clReleaseCommandQueue(queue);
64     clReleaseContext(context);
65 }

```

This code executes on the GPU.

This code executes on the CPU.

Interaction with OpenCL runtime begins here.

This command compiles the kernel.

A GPU view of the host memory buffer is created.

This command dispatches the kernel to the GPU.

This command waits for the GPU to finish.

Listing 2.2: OpenGL example code.

```
1  #include <stdio.h>
2  #include <GL/glut.h>
3
4  void display(void) ← This code executes on the GPU.
5  {
6      glClear( GL_COLOR_BUFFER_BIT);
7      glColor3f(0.0, 1.0, 0.0);
8      glBegin(GL_POLYGON);
9          glVertex3f(2.0, 4.0, 0.0);
10         glVertex3f(8.0, 4.0, 0.0);
11         glVertex3f(8.0, 6.0, 0.0);
12         glVertex3f(2.0, 6.0, 0.0);
13     glEnd();
14     glFlush();
15 }
16
17 This code executes on the CPU.
18 int main(int argc, char **argv)
19 {
20     printf("hello world\n");
21     glutInit(&argc, argv);
22     glutInitDisplayMode ( GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
23
24     glutInitWindowPosition(100,100);
25     glutInitWindowSize(300,300);
26     glutCreateWindow ("square");
27
28     glClearColor(0.0, 0.0, 0.0, 0.0);
29     glMatrixMode(GL_PROJECTION);
30     glLoadIdentity();
31     glOrtho(0.0, 10.0, 0.0, 10.0, -1.0, 1.0);
32
33     glutDisplayFunc(display);
34     glutMainLoop();
35
36     return 0;
37 }
```



Listing 2.3: SYCL example code.

```

1  #include <sycl.hpp>
2
3  using namespace cl::sycl
4
5  #define TOL (0.001)  // tolerance used in floating point comparisons
6  #define LENGTH (1024) // Length of vectors a, b and c
7
8  int main() {
9      std::vector h_a(LENGTH);          // a vector
10     std::vector h_b(LENGTH);          // b vector
11     std::vector h_c(LENGTH);          // c vector
12     std::vector h_r(LENGTH, 0xdeadbeef); // d vector (result)
13     // Fill vectors a and b with random float values
14     int count = LENGTH;
15     for (int i = 0; i < count; i++) {
16         h_a[i] = rand() / (float)RAND_MAX;
17         h_b[i] = rand() / (float)RAND_MAX;
18         h_c[i] = rand() / (float)RAND_MAX;
19     }
20
21     // Device buffers
22     buffer d_a(h_a);
23     buffer d_b(h_b);
24     buffer d_c(h_c);
25     buffer d_r(h_r);
26     queue myQueue;
27     command_group(myQueue, [&]() {
28         {
29             // Data accessors
30             auto a = d_a.get_access<access::read>();
31             auto b = d_b.get_access<access::read>();
32             auto c = d_c.get_access<access::read>();
33             auto r = d_r.get_access<access::write>();
34             // Kernel
35             parallel_for(count, kernel_func([ = ](id<> item) {
36                 int i = item.get_global(0);
37                 r[i] = a[i] + b[i] + c[i];
38             }));
39         }
40     });
41     printf("R = A+B+C: %d out of %d results were correct.\n", correct, count);
42     return (correct == count);
43 }

```

This is the GPU code!



Listing 2.4: CUDA implementation of SAXPY (Single Precision $A \cdot X + Y$).

```

1  #include <stdio.h>
2
3  __global__
4  void saxpy(int n, float a, float *x, float *y) {
5      int i = blockIdx.x*blockDim.x + threadIdx.x;
6      if (i < n) y[i] = a*x[i] + y[i];
7  }
8
9  This code executes on the CPU.
10
11 int main(void) {
12     [...]
13     cudaMalloc(&d_x, N*sizeof(float));
14     cudaMalloc(&d_y, N*sizeof(float));
15     [...]
16     cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
17     cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
18     saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
19     cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
20     [...]
21     cudaFree(d_x);
22     cudaFree(d_y);
23     free(x);
24     free(y);
25 }

```

This code executes on the GPU.

The CUDA API is slightly higher level than OpenCL, and less verbose.

2.2.2.4 Vulkan [21]

Vulkan, similarly to the above frameworks, is developed by the Khronos Group. It provides a lower-level alternative to OpenGL as a 3D graphics API for heterogeneous systems. It also implements a compute API, which reduces overheads especially when the results of these compute applications feed into the graphics pipeline. Vulkan is extremely verbose, with a draw triangle example taking over 1000 lines of code. As there is no easy way of fitting this into the thesis, we omit the Vulkan example. Examples can be found in [22].

2.2.2.5 CUDA [23]

CUDA is a proprietary heterogeneous compute programming framework developed by Nvidia. Its functionality is similar to OpenCL, however, while OpenCL is an open standard available on many platforms, CUDA is a proprietary framework only available for Nvidia GPUs. Traditionally, CUDA is preferred over OpenCL on Nvidia GPUs, as it is better optimized for Nvidia architectures, however its uses are limited in the mobile space, as the vast majority of Nvidia GPUs target desktop and server machines. A CUDA example is presented in Listing 2.4.

Listing 2.5: Example of adding two arrays using Apple's Metal.

```

1  #include <metal_stdlib>
2  using namespace metal;
3
4  kernel void add_arrays(device const float* inA,
5                        device const float* inB,
6                        device float* result,
7                        uint index [[thread_position_in_grid]])
8  {
9      result[index] = inA[index] + inB[index];
10 }
11
12
13
14 int main(int argc, const char * argv[]) {
15     @autoreleasepool {
16
17         id<MTLDevice> device = MTLCreateSystemDefaultDevice();
18
19         // Create the custom object used to encapsulate the Metal code.
20         // Initializes objects to communicate with the GPU.
21         MetalAdder* adder = [[MetalAdder alloc] initWithDevice:device];
22
23         // Create buffers to hold data
24         [adder prepareData];
25
26         // Send a command to the GPU to perform the calculation.
27         [adder sendComputeCommand];
28
29         NSLog(@"Execution finished");
30     }
31     return 0;
32 }
33
34
35 @interface MetalAdder : NSObject
36 - (instancetype) initWithDevice: (id<MTLDevice>) device;
37 - (void) prepareData;
38 - (void) sendComputeCommand;
39 @end

```

This code executes on the GPU.

This code executes on the CPU.

Metal uses similar concepts to OpenCL, while being less verbose.

2.2.2.6 Metal [24]

Metal is a proprietary heterogeneous compute and 3D graphics programming framework developed by Apple. It is designed to be low overhead and combines compute and graphics similar to Vulkan.

Metal takes an object-oriented approach, where the user can create custom objects with specific functions implemented for initializing and loading the data. The main function of the program is then vastly simplified, taking on a more declarative-style look. An example of Metal is presented in Listing 2.5.

2.2.2.7 Direct X [25]

Direct X is a collection of APIs for multimedia-related tasks, including 3D graphics acceleration, developed by Microsoft and available for Microsoft platforms including

Listing 2.6: Example of drawing a triangle in Direct X

```

1
2 class DirectXGame : core::DirectXApp
3 {
4 private:
5     Microsoft::WRL::ComPtr<ID3D11Buffer> vertexBuffer;
6 public:
7     DirectXGame(HINSTANCE hInstance);
8     ~DirectXGame();
9
10    util::Expected<void> init() override;           // game initialization
11    void shutdown(util::Expected<void>* expected = NULL) override; // cleans up and shuts the game down (handles
12                                                                    errors)
13    util::Expected<int> update(double dt);           // update the game world
14    util::Expected<int> render(double farSeer);       // render the scene
15
16    util::Expected<void> initGraphics();             // initializes graphics
17
18    util::Expected<int> run() override;               // run the game
19 };
20
21 util::Expected<void> DirectXGame::initGraphics()
22 {
23     // create the triangle
24     graphics::VERTEX triangleVertices[] = { { 0.0f, 0.5f, 0.0f }, { 0.45f, -0.5f, 0.0f }, { -0.45f, -0.5f, 0.0f } };
25
26     D3D11_BUFFER_DESC bd; // set up buffer description
27     bd.ByteWidth = sizeof(graphics::VERTEX) * ARRAYSIZE(triangleVertices);
28     bd.Usage = D3D11_USAGE_DEFAULT;
29     bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;
30     bd.CPUAccessFlags = 0;
31     bd.MiscFlags = 0;
32     bd.StructureByteStride = 0;
33
34     D3D11_SUBRESOURCE_DATA srd = { triangleVertices, 0, 0 };
35
36     This command draws the triangle using the GPU.
37     if (FAILED(d3d->dev->CreateBuffer(&bd, &srd, &vertexBuffer)))
38         return "Critical Error: Unable to create vertex buffer!";
39
40     return {};
41 }

```

PCs and gaming platforms. DirectCompute is a compute API integrated into Direct X. A Direct X example is presented in Listing 2.6.

2.2.3 High-Level Libraries

High-level libraries that abstract the complexities of low-level APIs have been developed to increase programmer productivity. Some examples of libraries that support GPU computation are the Arm Compute Library, TVM, and PyTorch.

2.2.3.1 Arm Compute Library

The Arm Compute Library is an open-source library developed by Arm, which exposes commonly used complex functions, through a high-level C++ API. The library

includes support for both Arm A-class CPUs, where the functions are implemented using C++, and Arm Mali GPUs, where the functions are implemented using OpenCL. The provided functions include:

- Basic arithmetic, mathematical, and binary operator functions
- Color manipulation (conversion, channel extraction, and more)
- Convolution filters (Sobel, Gaussian, and more)
- Canny Edge, Harris corners, optical flow, and more
- Pyramids (such as Laplacians)
- HOG (Histogram of Oriented Gradients)
- SVM (Support Vector Machines)
- H/SGEMM (Half and Single precision General Matrix Multiply).

2.2.3.2 TVM

The TVM compiler [26] is an open source deep learning compiler stack for CPUs, GPUs, and accelerators. It aims to close the gap between the productivity-focused deep learning frameworks, and performance- or energy-efficiency oriented hardware backends, by using machine learning to translate high-level operators to optimized, low-level, kernel code. TVM provides the following features:

- Compilation of deep learning models in Keras, MXNet, PyTorch, Tensorflow, CoreML, DarkNet into minimum deployable modules on diverse hardware backends.
- Infrastructure to automatically generate and optimize tensor operators on more backends with better performance.

Similarly to the Arm Compute Library, we use TVM in our characterization of convolutional neural networks on embedded devices, presented in Section 4.6.2.

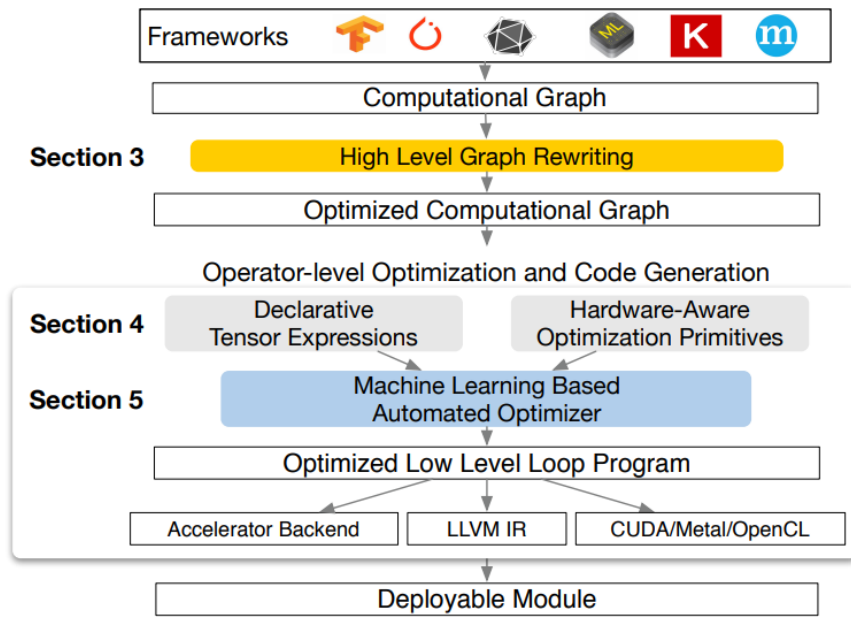


Fig. 2.2: An overview of the TVM deep learning compiler. (Fig. Source: [26])

2.2.3.3 PyTorch

PyTorch is a prime example of a complex CPU/GPU software stack present in modern day applications. It is a deep learning framework exposed through the Python programming language. Deep learning models are typically invoked by selecting a model from a library of available neural networks, and providing inputs. Under the hood, PyTorch implements a number of backends, for the components of the neural networks, typically using existing, optimized compute libraries, for example MKL on CPUs, or the Arm Compute Library and TVM on GPUs.

This section has presented a number of GPU programming frameworks, all of which are automatically supported in our full-system simulation framework, which we present in Chapter 4. Now that we have introduced common GPU programming frameworks, we move on to describing the GPU architecture that the final kernel binaries execute on.

2.3 The Arm Mali Bifrost Architecture

The Mali Bifrost architecture was developed by Arm and has been licensed since 2016, becoming the primary Arm GPU architecture between 2016 and 2019. At the time of release, it was a complete re-design of Arm's line of GPUs, which was preceded by

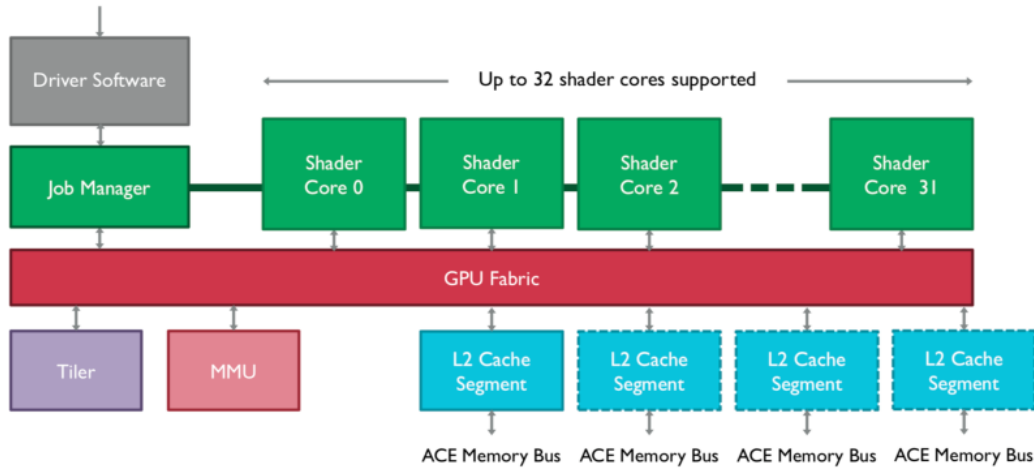


Fig. 2.3: The Bifrost Architecture. Image reproduced with kind permission from Arm Ltd. [27]

the Midgard architecture.

The Bifrost architecture introduced a number of new features, such as clause execution, a scalar ISA, a new core fabric, and quad-based arithmetic units, which overall resulted in a 20% increase in energy efficiency, 40% better performance density, and 20% bandwidth improvement for the Mali-G71 (Bifrost) over the Mali-T880 (Midgard).

In this section, we provide details of the Arm Mali Bifrost architecture, which is the GPU architecture modelled in the work presented in this thesis. The figures in this section are reproduced with the kind permission of Arm Ltd.

2.3.1 Overview

The Mali Bifrost GPU is composed of a scalable number of Shader Cores, a Job Manager, MMU, Tiler Unit, and L2 Cache. In the following sections, we provide details of GPU components used in the compute pipeline. Graphics-specific hardware is mentioned, but not examined in greater detail, as it is not required for understanding this thesis. An overview of the hardware is provided in Figure 2.3. We start with a general overview of software execution in GPU hardware, before moving onto the intricacies of the Bifrost architecture.

2.3.2 Starting a GPU Job

A new GPU Job is communicated to the GPU using the CPU-side kernel driver. The Job Manager receives this request, creates threads, and packs them into threadgroups, which are dispatched to individual Shader Cores. In the Shader Core, threads are packed into Warps, which then take turns executing on the Execution Engines. Once all of the work has been completed, the Job Manager signals back to the CPU.

2.3.3 Interaction with the CPU

The CPU and GPU communicate via three mechanisms - interrupts, memory-mapped registers, and shared memory. The main memory, which is shared between the CPU and the GPU, and is hence shared memory, holds all of the information passed between the CPU and GPU. This is where memory buffers, the GPU kernel binaries, and all of the Job Descriptors are stored. The memory-mapped registers are used for communicating state and pointers to the relevant sections of memory from the CPU to the GPU. Interrupts are used to signal ready states between CPU and GPU. A trace of the communication between CPU and GPU using memory-mapped registers and interrupts during the Linux boot sequence is presented in Listing 2.7. The Job Manager is the main hardware component responsible for the GPU-side interaction with the CPU, and is described next.

2.3.4 Details of the Job Manager

The Job Manager's primary role is to communicate with the device driver, which is executing on the CPU. The Job Manager receives a signal from the CPU-side device driver that a Job is ready to begin executing. This signal is communicated via the GPU's control registers. At this point, the Job Manager can start reading values from memory mapped registers, which were previously written to by the device driver. These memory mapped registers contain information about the Job configuration, for example the number of workgroups (the software term for a hardware threadgroup), their sizes, and their dimensions. They also contain the virtual addresses of memory buffers which were initialized by the CPU-side program, and virtual addresses pointing to the GPU kernel binaries, which were compiled by the OpenCL (or other framework) JIT-compiler. The Job Manager decodes all of this information, creates tasks, and dispatches them to the waiting Shader Cores. Upon Job completion, the Job Manager is

responsible for communicating this, as well as any associated state back to the device driver. Next, we look at the MMU, which is also tightly coupled with the remainder of the system, as the Bifrost GPU shares physical memory with the CPU.

2.3.5 The Arm Mali Memory System

Memory systems in GPUs fall into two broad categories - discrete desktop GPUs traditionally have separate, local, and private memories. Smaller, mobile systems on the other hand, where the GPU is embedded into the same SoC as the CPU, tend to implement unified memory systems, backed by caches.

Having a separate memory improves performance on discrete GPUs, however it comes at the cost of power, area, and heat, all of which are tightly constrained on mobile GPUs.

The Mali series of GPUs, being mobile GPUs tightly coupled with a CPU, embedded into a single SoC, implement a unified memory model. This means that the CPU, GPU, and possibly other interconnected accelerators, all share the main memory. Furthermore, the Mali GPU is capable of sharing physical, or even virtual address spaces with the CPU, if the page tables are configured to be shareable.

2.3.5.1 The Bifrost MMU

The Bifrost Architecture has its own MMU used for translating virtual addresses to physical, and for keeping track of the GPU's multiple address spaces. The dedicated MMU implements the same translation policies as the Arm-v8 MMU, and can share page tables with the CPU-side MMU. The MMU also has its own interrupt line, which it can use to raise faults detected during translation.

2.3.5.2 Synchronizing Memory

Bifrost GPUs have two ways of synchronizing memory accesses. The first way is using the OpenCL `CLK_GLOBAL_MEM_FENCE` or `CLK_LOCAL_MEM_FENCE` directives, which ensure that all global and local memory accesses become visible (respectively). The other mechanism is via atomic operations, which again are exposed via OpenCL. Atomic operations ensure that memory can be read, modified, and written to in a single operation, without interference from other threads. Both of these mechanisms are communicated to the hardware via dedicated machine instructions.

Listing 2.7: Trace of reads from and writes to GPU memory-mapped registers performed by the Device Driver when booting Linux.

```

1
2
3 GPU READ: offset: 0, base: 8080000 # READ GPU ID
4 GPU READ: offset: 4, base: 8080000 # READ L2 FEATURES
5 GPU READ: offset: 8, base: 8080000 # SUSPEND_SIZE Buffer
6 GPU READ: offset: c, base: 8080000 # TILER FEATURES
7 GPU READ: offset: 10, base: 8080000 # MEM_FEATURES
8 GPU READ: offset: 14, base: 8080000 # MMU_FEATURES
9 GPU READ: offset: 18, base: 8080000 # AS_PRESENT
10 GPU READ: offset: 1c, base: 8080000 # JS_PRESENT
11 GPU READ: offset: c0, base: 8080000 # JS0_FEATURES
12 GPU READ: offset: c4, base: 8080000 # JS1_FEATURES
13 GPU READ: offset: c8, base: 8080000 # JS2_FEATURES
14 GPU READ: offset: cc, base: 8080000 # JS3_FEATURES
15 GPU READ: offset: b0, base: 8080000 # TEXTURE_FEATURES_0
16 GPU READ: offset: b4, base: 8080000 # 1
17 GPU READ: offset: b8, base: 8080000 # 2
18 GPU READ: offset: bc, base: 8080000 # 3
19 GPU READ: offset: a0, base: 8080000 # THREAD_MAX_THREADS
20 GPU READ: offset: a4, base: 8080000 # THREAD_MAX_WORKGROUP_SIZE
21 GPU READ: offset: a8, base: 8080000 # THREAD_MAX_BARRIER_SIZE
22 GPU READ: offset: ac, base: 8080000 # THREAD_FEATURES
23 GPU READ: offset: 100, base: 8080000 # SHADER_PRESENT_LO
24 GPU READ: offset: 104, base: 8080000 # SHADER_PRESENT_HI
25 GPU READ: offset: 110, base: 8080000 # TILER_PRESENT_LO
26 GPU READ: offset: 114, base: 8080000 # TILER_PRESENT_HI
27 GPU READ: offset: 120, base: 8080000 # L2_PRESENT_LO
28 GPU READ: offset: 124, base: 8080000 # L2_PRESENT_HI
29 GPU READ: offset: e00, base: 8080000 # STACK_PRESENT_LO
30 GPU READ: offset: e04, base: 8080000 # STACK_PRESENT_HI
31 [2.480000] mali 8080000.gpu: GPU identified as 0x0 arch 6.0.0 r0p0 status 1
32 GPU READ: offset: 300, base: 8080000 # COHERENCY_FEATURES
33 GPU WRITE: offset: 28, base: 8080000 # GPU_IRQ_MASK
34 GPU WRITE: offset: 24, base: 8080000 # GPU_IRQ_CLEAR
35 GPU WRITE: offset: 1008, base: 8080000 # JOB_IRQ_MASK
36 GPU WRITE: offset: 1004, base: 8080000 # JOB_IRQ_CLEAR
37 GPU WRITE: offset: 2008, base: 8080000 # MMU_IRQ_MASK
38 GPU WRITE: offset: 2004, base: 8080000 # MMU_IRQ_CLEAR
39 GPU WRITE: offset: 30, base: 8080000 # GPU_COMMAND
40 GPU WRITE: offset: 28, base: 8080000 # GPU_IRQ_MASK GPU: RAISE

```

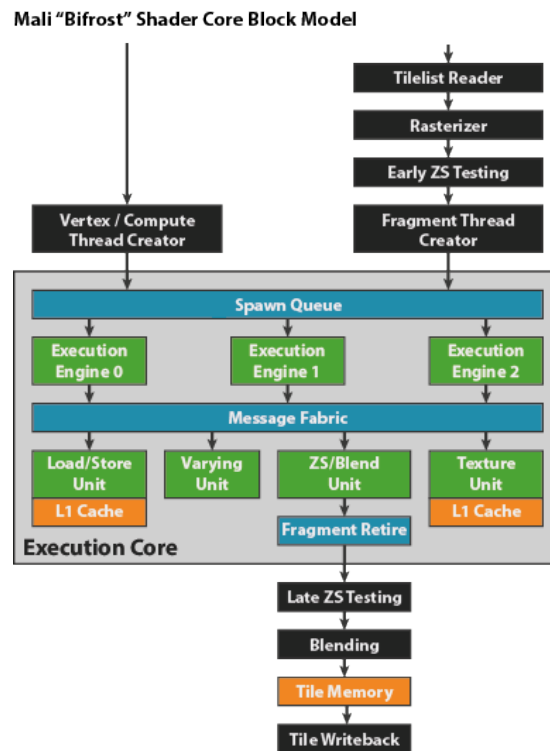


Fig. 2.4: The programmable Bifrost Shader Core, with surrounding blocks demonstrating the fixed function rendering pipeline. Image reproduced with kind permission from Arm Ltd. [28]

2.3.6 The Bifrost Shader Core

The Bifrost Shader Core receives threads from the Compute Thread Creator for compute workloads, or from the Fragment Thread Creator for graphics workloads. These threads are dispatched to the Execution Engines via the spawn queue. The Execution Engines execute the binary Shader - i.e., Bifrost machine code, and perform the majority of computations. Specialized operations are offloaded via the Message Fabric onto one of the additional units within the Shader Core - the Load/Store Unit, the Varying Unit, the ZS/Blend Unit, and the Texture Unit.

2.3.6.1 Bifrost Execution Engines

In order to improve utilization of the GPU's functional units, the Bifrost architecture implements a quad-vectorization scheme. A typical SIMD architecture is often under-utilized when operating on types that don't fit the SIMD unit's width exactly. For example, Figure 2.5 shows the utilization of a SIMD architecture executing on a three-component vector type. Only three out of the four parallel units are utilized in this



Fig. 2.5: A generic SIMD architecture operating on a vec3 type. Image reproduced with kind permission from Arm Ltd. [28]



Fig. 2.6: The Bifrost architecture quad state. Image reproduced with kind permission from Arm Ltd. [28]

case, leaving the fourth idle. The Bifrost architecture on the other hand, breaks all operations down into single threads, and then dynamically composes Warps of four threads.

Bifrost implements a substantial general-purpose register file, comprising 64 32-bit registers, while allowing maximum thread occupancy. The register file can be accessed using various data widths from 8-bits to 128.

In addition to general purpose registers, Bifrost also has 2KB of dedicated read-only memory, called the Fast Access Uniform RAM (FAU RAM), and can inject constants into the instruction cache.

The Bifrost pipeline is presented in Figure 2.7. It comprises two compute stages, the FMA, and the ADD stages, which can each execute a different set of instructions.

2.3.7 The Clause Based Execution Model

The Bifrost architecture provides a hybrid approach between a VLIW and traditional architecture through its Clause execution model. A Shader Program is composed of a number of Clauses. Each Clause is composed of anywhere between one and eight Instruction Words, and each Instruction Word contains two instructions - one for the ADD pipeline and the other for the FMA pipeline. Control flow can only take place

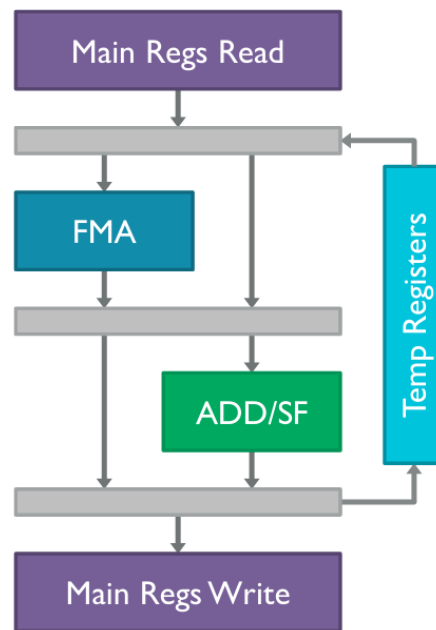


Fig. 2.7: Bifrost has two compute pipeline stages, which can pass values directly from the FMA to the ADD stage, without writing back to the global register file. Image reproduced with kind permission from Arm Ltd. [27]

between Clauses, i.e., a control flow instruction, for example a `BRANCH` or a `JUMP`, has to be the last instruction in a Clause. The FMA and ADD pipelines execute in sequence in that order, and results can be passed directly from the FMA pipeline to the ADD pipeline, without writing back to the Global Register File. At the instruction set level, these reads and writes are visible as accesses to temporary registers.

Figure 2.9 depicts a classic instruction execution model. Instructions are executed in sequence, and before and after each instruction, there is overhead related to decoding the instruction, as well as reading from and writing to registers. Figure 2.10 depicts the Clause execution model found in Bifrost. Instructions are bundled into groups, called Clauses, and any overhead is limited to the Clause boundaries, meaning that the cost is amortized across a number of instructions. Instructions within a Clause share resources, for example, Embedded Constants which are injected into the Clause encoding. Clauses contain Instruction Words, and each Instruction Word contains two instructions. Instructions within an Instruction Word also share resources, for example the Register Block, which is part of a two-step encoding for register accesses, and the base address of the FAU RAM.

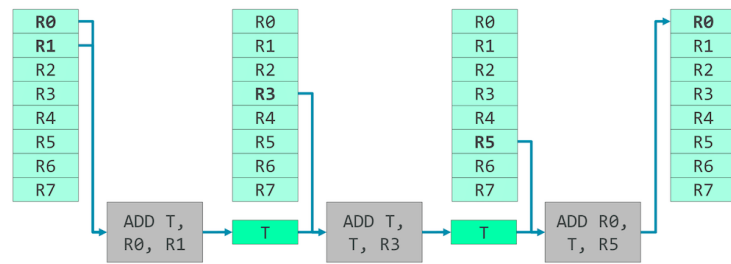


Fig. 2.8: Temporary Register Access. Image reproduced with kind permission from Arm Ltd. [29]

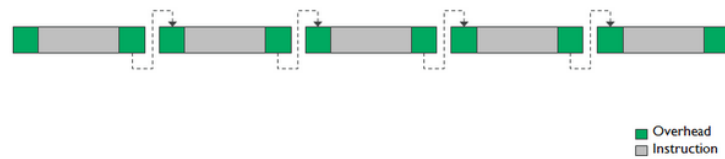


Fig. 2.9: Classic Instruction Execution Model. Image reproduced with kind permission from Arm Ltd. [30]

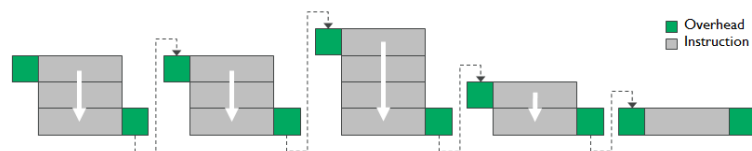


Fig. 2.10: Clause Execution. Image reproduced with kind permission from Arm Ltd. [30]

2.3.7.1 Message Fabric

The Shader Cores are connected to additional supporting units via the Message Fabric. These units are the Load/Store Unit, Varying Unit, ZS/Blend Unit, and Texture Unit. The instruction set exposes the supporting units through specific instructions. This is depicted in the Shader Core diagram in Figure 2.4.

2.3.7.2 Load/Store Unit

The Load/Store Unit is responsible for performing memory operations. As memory operations are more expensive than arithmetic operations, memory operations are off-loaded onto the Load/Store unit, so that the Execution Engine can continue performing other operations. The Execution Engine will be able to progress the current warp until the result of the load is needed.

2.3.8 Graphics Hardware

Other units included in the GPU specific to graphics include the Varying Unit, ZS/Blend Unit, and the Texture Unit. We do not model these in our simulator, and as such, we do not provide details of these units.

Now that we have discussed the GPU architecture, we move onto describing simulation infrastructure that our work builds on. In the following section, we describe GenSim, a simulator generation framework used for both CPU and GPU components of our simulator.

2.4 GenSim - A Head Start on Fast Simulation

Simulators are expensive to develop, and difficult to optimize. However, once developed, the core components of a simulator can be re-used for similar architectures, and only the instruction set needs to be re-implemented. This principle of re-use has been exploited by GenSim [31, 32], a fast, retargetable, full-system simulation framework. The GenSim framework is provided with a high-level architecture description by the user, from which it generates a fast instruction set simulator. Not only does it provide the user with a simulator at negligible engineering cost compared to designing the simulator from scratch, but it is also capable of simulation at faster than native speeds.

GenSim is used to generate both the CPU and GPU modules presented in this thesis. Its internal JIT engine is used as the main engine for the Arm-v7 model, which was

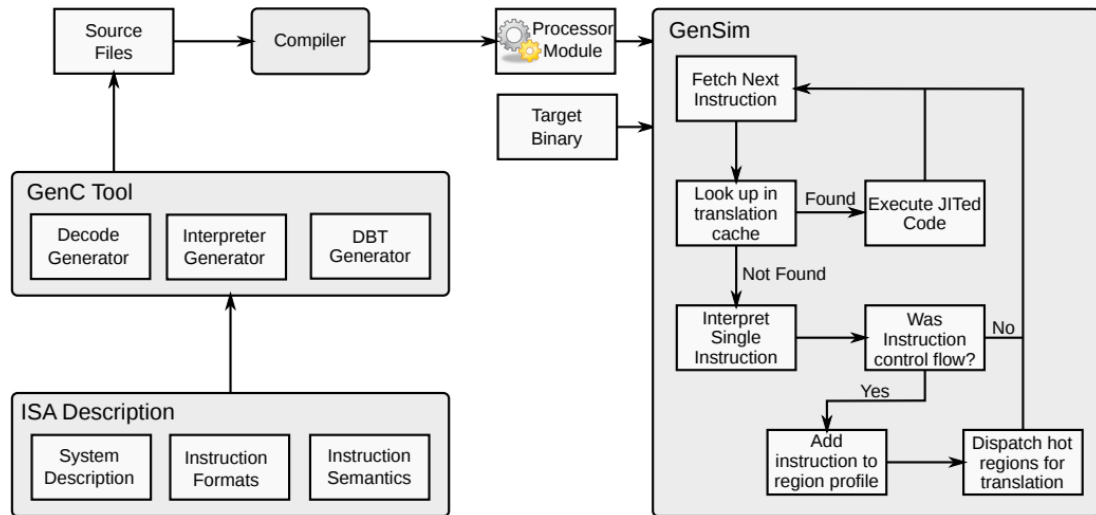


Fig. 2.11: Overview of Gensim framework (Source: [31]).

used as the CPU architecture in the first version of the full-system simulator. Additions made to the GenSim models and infrastructure as a part of this thesis are described in Chapters 4.7.2 and 4.7.3.1.

An overview diagram of GenSim is provided in Figure 2.11. Models used in GenSim are described in the GenC [31] description language, which is originally based off of the ArchC [33] architecture description language. The model comprises three components - the system description, instruction syntax, and instruction semantics, which are described next.

2.4.0.1 System Description

The system specification defines characteristics of the architecture such as the endianness, word size, register files, and status flags. An example is shown in Listing 2.8. Annotation ① defines the register spaces for the Armv7-A model. The **BANK** keyword specifies a register bank, with the type, offset, count, stride, number of elements, element size, and element stride as parameters. Additionally, specific registers within the register bank can be specified using the **SLOT** keyword, for which dedicated access functions are later generated. Annotation ② specifies the word size used in the architecture. The **ARCH_CTOR** denoted by Annotation ③ references the remaining files, and sets the endianness for the architecture.

Listing 2.8: GenC Armv7a System Description

```

1  AC_ARCH(armv7a)
2  {
3      // General Purpose Registers
4      ac_regspace(64) { ①
5          // bank NAME (TYPE, OFFSET, COUNT, REG-STRIDE, # ELEMS, ELEM-SIZE, ELEM-STRIDE)
6          bank RB (uint32, 0, 16, 4, 1, 4, 4);
7          slot PC (uint32, 4, 60) PC;
8          slot SP (uint32, 4, 52) SP;
9      }
10
11      // Floating point registers
12      // Type, offset, count, register stride, element count, element size, element stride
13      ac_regspace(256)
14      {
15          bank FPSP (float, 0, 32, 4, 1, 4, 4);
16          bank FPD (double, 0, 32, 8, 1, 8, 8);
17          bank VD (float, 0, 32, 8, 2, 4, 4);
18          [...]
19      }
20
21      // General Flags
22      ac_regspace(14) {
23          slot C (uint8, 1, 0) C;
24          slot Z (uint8, 1, 1) Z;
25          [...]
26      }
27
28      [...]
29
30      // FSS Regs
31      ac_regspace(16) {
32          slot M (uint8, 1, 0);
33          slot F (uint8, 1, 1);
34          [...]
35      }
36
37      [...]
38
39      ac_wordsize 32; ②
40
41      ARCH_CTOR(armv7a) ③
42      {
43          ac_isa("armv7a_isa.ac");
44          ac_isa("armv7a_thumb_isa.ac");
45          set_endian("little");
46          set_feature(ARM_SDIV_UDIV, 1);
47      };
48
49 };
50

```

The register space defines the register file, as described by the ISA.

Slots are a named view to a specific register in the register bank.

Multiple register files with different data types can be defined.

Standalone registers are defined here.

The word size is specified as 32 bits.

Here we reference the ISA files, which contain the ISA syntax description, set the endianness, and enable additional features.

Listing 2.9: GenC Armv7a Syntax Description

```

1  AC_ISA(arm)
2  {
3
4      ac_fetchsize 32; ④ ← The fetch size is defined as 32 bits.
5
6      include("vfpv4.ac"); ⑤ ← Additional syntax descriptions can be included,
7      include("neon.ac");    for example instruction set extensions.
8
9      ac_format Type_MULT = "%cond:4 %op!:3 %func1!:4 %s:1 %rn:4 %rd:4 %rs:4 %subop2!:1 %func2!:2 %subop1!:1 %rm:4";
10     ac_format Type_SMUL = "%cond:4 0x16:8 %rd:4 0x0:4 %rs:4 0x1:1 %y:1 %x:1 0x0:1 %rm:4"; ⑥
11
12     The format specifies an encoding for an instruction type.
13
14     ac_instr<Type_MULT> swp, swpb, mla, mul; ⑦ ← Here we define which instructions
15     ac_instr<Type_SMUL> smulxy;              belong to an instruction type.
16
17     ac_asm_map cond ⑧ ← Values are mapped to fields in the disassembly.
18     {
19         "eq" = 0;
20         "ne" = 1;
21         "cs" = 2;
22         "cc" = 3;
23         [...]
24     }
25
26     ac_behaviour mul; ⑨ ← Instruction behaviour declaration.
27     [...]
28     ac_behaviour smulxy;
29
30     Values are provided for some fields
31     to disambiguate instructions belonging to
32     an instruction type during decoding.
33     The disassembly format and a reference to
34     the instruction semantics is also provided.
35
36     ISA_CTOR(armv7a)
37     {
38         mul.set_decoder(op=0x00, subop1=0x01, subop2=0x01, func1=0x00, func2=0x00, rn != 15); ⑩
39         mul.set_asm("mul%[cond]%sf %reg, %reg, %reg", cond, s, rn, rm, rs, rd=0x00);
40         mul.set_behaviour(mul);
41
42         [...]
43
44         smulxy.set_decoder(); ⑪
45         smulxy.set_asm("smulxy");
46         smulxy.set_behaviour(smulxy);
47     };
48 };

```

Listing 2.10: GenC Armv7a Semantics Description

```

1  execute(mul)
2  {
3
4      uint32 t;
5      t = (read_gpr(inst.rm)) * (read_gpr(inst.rs)); ④ ← Values are read from registers.
6
7      if (inst.s) update_ZN_flags(t); ⑤ ← Operation is performed.
8
9      write_register_bank(RB, inst.rn, t); ⑥ ← Result is written back to registers.
10 }
11
12 execute(smulxy)
13 {
14     uint32 rm = read_register_bank(RB, inst.rm);
15     uint32 rs = read_register_bank(RB, inst.rs);
16
17     uint32 op1 = inst.x == 0 ? ((uint32)(sint32)(sint16)(rm)) : ((uint32)(sint32)(sint16)(rm >> 16));
18     uint32 op2 = inst.y == 0 ? ((uint32)(sint32)(sint16)(rs)) : ((uint32)(sint32)(sint16)(rs >> 16));
19
20     write_register_bank(RB, inst.rd, op1 * op2);
21 }

```

2.4.0.2 Instruction Syntax

The syntax file (Listing 2.9) first specifies the word fetchsize (annotation ④) and optionally, additional files that can contain instruction set extensions (annotation ⑤). Following these, instruction formats within an instruction set are defined. Many instructions share a common format, and can therefore be grouped together for a more compact representation of the instruction set in GenC, as well as for easier and more efficient decoder generation by GenSim. Annotation ⑥ implements the TYPE_MULT and TYPE_SMUL instruction formats in our Armv7-A model. These instruction types are used predominantly for swap and multiplication instructions. Each instruction in this model is 32 bits wide, and the instruction format assigns each of these bits to a specific field in the instruction format.

Next, all instructions are declared and assigned to an instruction format, as shown by annotation ⑦.

An assembly map can be specified to help with disassembly, as shown by annotation ⑧. This map translates numerical values for a specific field to their text equivalents, as specified by the Armv7-A instruction set manual [34].

Following this, instruction behaviours are declared, as shown by annotation ⑨. The behaviours are used for connecting instructions syntax in this file, to their semantics in the semantics description file.

The final component of the syntax file includes instruction syntax definitions, which assign the encoding, assembly format, behaviour, and any special attributes to each in-

struction. Annotation ⑩ defines the MUL instruction, with multiple fields relating to the instruction format defined in order to differentiate the MUL instruction from the remaining instructions sharing the TYPE_MULT format during decoding. The SMULXY instruction (annotation ⑪) on the other hand is the only one using the TYPE_SMUL format, and therefore needs no additional decode information. In addition to the decode information, the assembly format, and behaviour are specified. The behaviour links the instruction syntax to the semantics.

2.4.0.3 Instruction Semantics

The final component, called the *execute* file, contains the instruction semantics. Instruction semantics are described in a C-like language, with embedded intrinsics for accessing registers, memory, and instruction fields. Helper functions can also be specified in order to share code among instructions. An example is shown in Listing 2.10. Annotation ⑫ demonstrates how values are read using intrinsic functions out of the register state. Annotation ⑬ shows how the instruction fields are accessed using the INST struct. Annotation ⑭ shows how values are written back to the register file following the instruction execution.

2.4.1 ArcSim

GenSim is a heavily modified version of ArcSim [35,36], a simulator originally developed to simulate the EnCore microprocessor. ArcSim contains multiple components critical to fast, full-system simulation. First and foremost, it encompasses the fast JIT-compilation framework that allows Dynamic Binary Translation (DBT) from guest to host instruction set. Secondly, ArcSim implements the memory model of the guest CPU, including a functional MMU implementation. Thirdly, ArcSim supports peripheral devices, that while may not be necessary in a user-mode simulation, are strict requirements for booting and using Linux. ArcSim serves as the back-end JIT compiler, core, and system simulator, while the new GenSim additions generate architectural models specific to the simulated CPU.

In the following section, we describe Captive, a simulator that builds and improves on ArcSim.

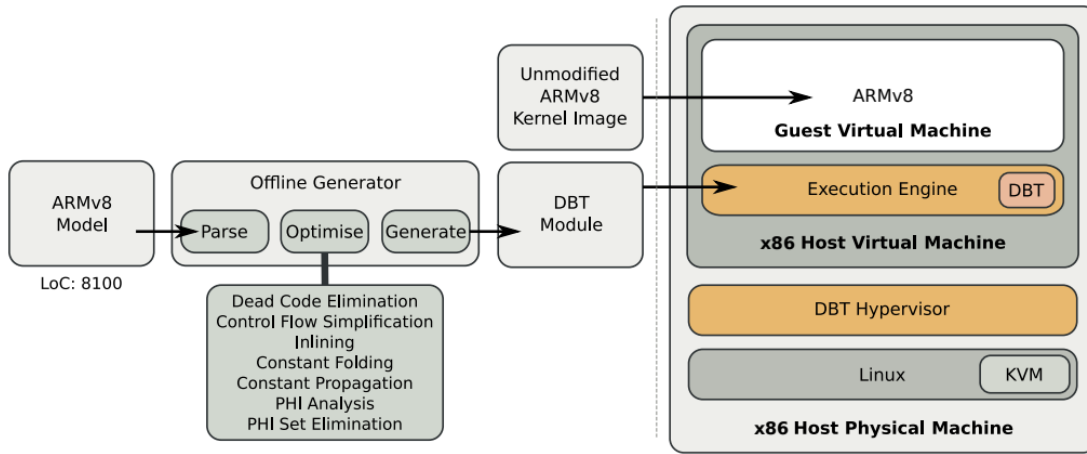


Fig. 2.12: Captive simulation diagram (Fig. Source: [37]).

2.5 Captive - Making the most of host hardware

Captive [37] is an implementation of a cross-architecture virtualization hypervisor. It takes advantage of host virtualization support for accelerating simulation. Furthermore, Captive leverages the GenSim framework to generate guest specific simulation modules from high-level GenC descriptions. We use Captive to simulate the CPU in our full-system GPU simulation system, when using the AARCH64 CPU model. Figure 2.12 provides an overview of Captive.

Following our discussion of simulation frameworks, the next section will discuss the benchmarks used in the development and evaluation of our simulation framework, as well as in use cases showing the potential of the GPU simulator.

2.6 Validating Simulation Through Benchmarking

Benchmarks are key to understanding performance of a program on a specific architecture, optimizing code, and correctness validation. A number of different benchmark suites were used in the work leading to this thesis - first for validating the simulator, then for understanding the performance of the architecture by executing benchmarks in the simulator and in hardware, and finally, for building a performance model. This section presents the benchmarks used in the development of the GPU simulator.

A number of considerations guided the selection of benchmarks. First and foremost, we looked for existing benchmarks capable of executing in our simulator, real hardware, and Multi2Sim, which we used for comparison in Chapter 4. While our simulation approach enables execution of any benchmark, the compiler toolchain required

for Multi2Sim is no longer available, and therefore we relied on pre-compiled binaries from the AMD APP SDK. Second, we looked for a variety of benchmarks from different sources and with different coding styles. This allowed us to exercise different functionalities of the simulator. A final consideration was to select benchmarks commonly executed on mobile GPUs. Today, compute capacity of mobile GPUs is most commonly used for accelerating machine learning and computer vision applications, therefore a number of selected benchmarks are standard computational and filtering kernels, which are commonly found in such applications. Convolutional Neural Networks and SLAMBench were selected as examples of modern compute applications, executing on mobile GPUs, and requiring significant CPU-GPU interaction, which would not be possible to simulate using existing GPU simulators.

While our simulator is capable of executing benchmarks using local memory, mobile GPUs don't have a dedicated GPU memory, and the Arm Mali GPU Programming Guide explicitly states that programmers should not use local memory. This is because local memory in Mali GPUs is implemented by simply copying the data into a different part of the main memory. Benchmarks making use of local memory were used for functional validation of the simulator presented in Chapter 4, however were excluded from the performance modelling experiments presented in Chapter 5.

2.6.1 Parboil

Parboil [38] is a benchmark suite for scientific and commercial throughput computing developed at the University of Illinois at Urbana-Champaign. The benchmarks cover a number of applications, such as image processing, biomolecular simulation, fluid dynamics, and astronomy. While these workloads are not necessarily representative of mobile workloads, they were crucial in validating correctness of our simulation framework, as well as demonstrating the scale and speed of our simulation framework. The benchmark suite contains C++, CUDA, and OpenCL implementations. As the Arm Mali GPU supports only OpenCL, we use the OpenCL implementations. Descriptions of the Parboil benchmarks can be found in Table 2.2.

2.6.2 Polybench

Polybench [39] is a benchmark suite containing static control parts, designed with a goal of uniformizing the execution and monitoring of kernels typically used in publications. Key Polybench features include:

- Single file implementation,
- Non-null data initialization,
- Live-out data dump,
- Syntactic constructs to prevent dead code elimination on the kernel,
- Parametric loop bounds in the kernels
- Clear kernel marking using `#PRAGMA SCOP` and `#PRAGMA ENDSCOP` delimiters.

A summary of the benchmarks can be found in Table 2.3. In this thesis, we use the OpenCL version of the benchmark suite taken from Polybench-ACC [40]. Polybench contains a number of linear algebra and datamining kernels. While these are not specifically optimized for mobile GPUs, they are representative of kernels commonly executed on mobile GPUs - i.e. they use matrix inputs and outputs, and parallelize matrix operations across GPU hardware.

2.6.3 AMD APP SDK 2.5

The AMD APP SDK provides a number of benchmarks optimized for AMD GPUs, and was released as part of the software development kit for AMD customers. However, since they are implemented in OpenCL, they are portable to other GPUs, such as Arm Mali GPUs, which we model. We use the AMD APP SDK for direct comparison against Multi2Sim [41], as binary benchmarks are provided for Multi2Sim, however the toolchain compatible with Multi2Sim is no longer available for compiling new benchmarks. We use the version provided by the authors of Multi2Sim [42]. Benchmark descriptions can be found in Table 2.4.

2.6.4 Rodinia

Rodinia [43] aims to be a benchmark suite for general purpose accelerators, implementing multiple heterogeneous computing infrastructures including OpenCL, and CUDA. The benchmarks cover a wide range of parallel communication patterns, synchronization techniques, and power consumption. The benchmarks are listed in Table 2.5. From Rodinia, we select a number of benchmarks that are commonly executed on mobile GPUs, including NN and BACKPROP.

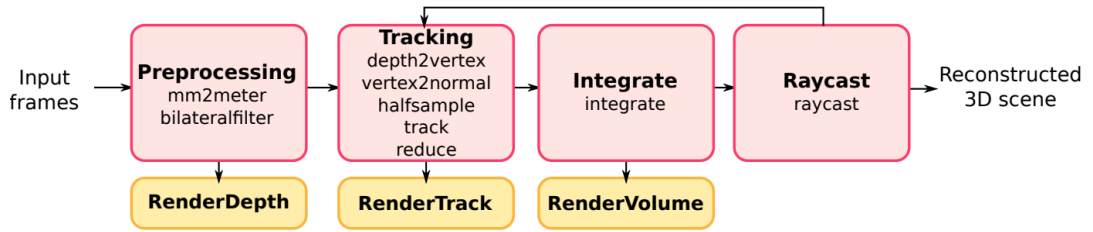


Fig. 2.13: Key computational steps of the KFusion algorithm represented as a task graph. Each task comprises one or more OpenCL kernels as depicted (Source: [46]).

2.6.5 SLAMBench

SLAMBench [44–46] is a benchmarking application for SLAM (Simultaneous Localization and Mapping) algorithms, a critical application in the field of robotics, and an example of a relevant, real-world, multi-kernel application that can be run using our framework. The benchmark was developed with mobile systems in mind, and is available as an application in the Google Play store. The benchmark comprises twelve different kernels, depicted in Figure 2.13. In Chapter 4, we present results from executing the entire SLAMBench benchmarking application in our full-system simulator.

2.6.6 DeepSmith

DeepSmith [47] is a compiler fuzzing framework applied to the OpenCL programming language, designed with the target of discovering bugs in compilers. DeepSmith automatically scrapes GitHub for OpenCL code, and trains a model by learning the structure of real world code. From this learned model, DeepSmith can generate previously unseen, random, OpenCL kernels. We leverage this framework to generate OpenCL kernels for our performance model, however we encounter limitations to this approach when used for performance modelling. We describe our efforts in Chapter 5. An overview diagram of DeepSmith is presented in Figure 2.14.

2.6.7 SGEMM

The SGEMM compute kernel is pervasive in modern day computing, in particular in the fields of machine learning, image processing, and graph analytics. As it takes up a majority of the computation time, it is an obvious target for acceleration, and has been implemented in countless different ways in OpenCL and CUDA for GPU acceleration. Cedric Nugteren presents a tutorial on SGEMM optimization for Nvidia

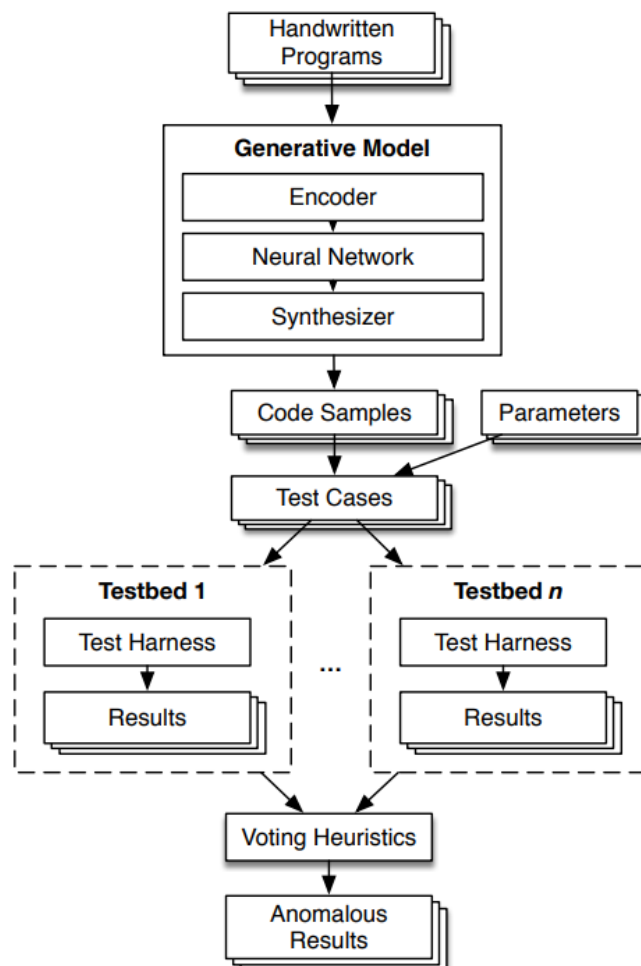


Fig. 2.14: Overview of the DeepSmith random kernel generation framework (Source: [47]).

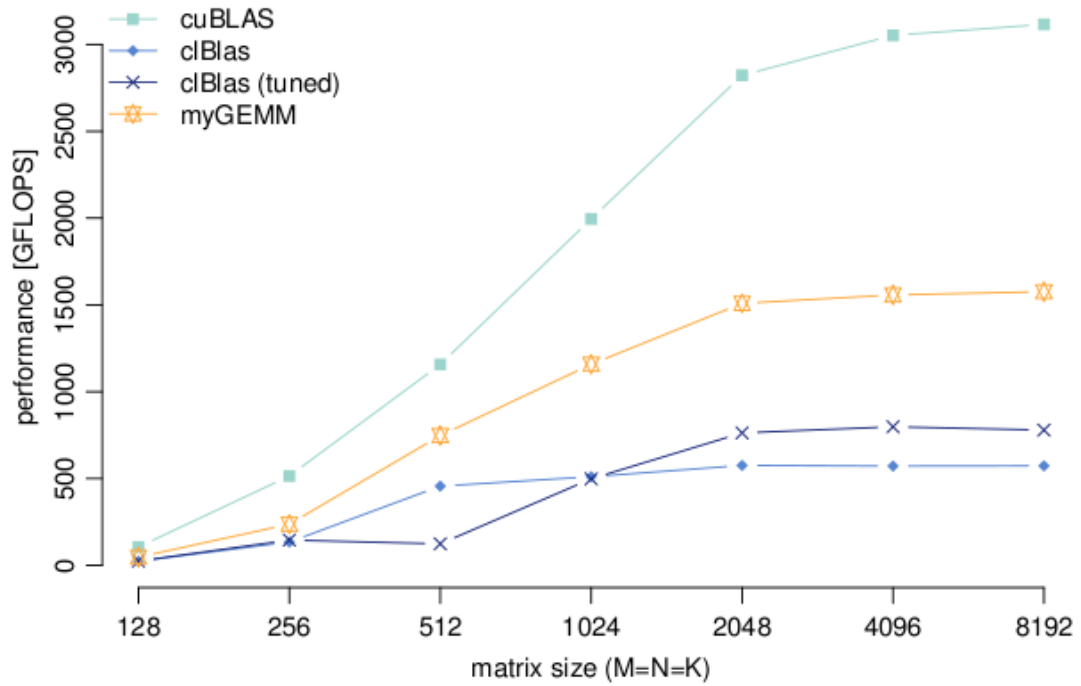


Fig. 2.15: Performance achieved by successive optimizations relative to clBlas and cuBLAS (Source: [48])

Kepler GPUs [48]. We execute the same kernels in our Mali GPU simulator as well as on real hardware, and we investigate if the kernels follow the same performance trends on Mali. We present our results in Chapter 4.6.2.

The 10 different versions presented in [48] implement the following (successive) optimizations:

1. Naive implementation.
2. Tiling in the local memory.
3. Increased work per thread.
4. Wider data-types (vectors).
5. Transposed input matrix and rectangular tiles.
6. 2D register blocking.
7. Wider loads with register blocking.
8. CUDA and Kepler-specific optimisations.

9. Software pre-fetching.

10. Incomplete tiles and support for arbitrary matrix-sizes.

A performance comparison of the final, optimized, version of SGEMM against clBlas and cuBLAS is shown in Figure 2.15.

2.6.8 Convolutional Neural Network Channel Pruning

Due to their superior recognition accuracy, Convolutional Neural Networks (CNN) are dominant in several disciplines: computer vision (for image classification [49–51], image segmentation [52, 53], objects in image detection [54, 55], image style transfer [56], etc.), speech recognition [57] and natural language processing [58, 59]. In this section, we present the background information necessary for understanding a case study, presented in Section 4.6.3, and describe the executed benchmarks.

CNNs are making their way into smaller devices, on mobile phones and home personal assistant devices. However, current CNN models are still too large for immediate deployment on resource-constrained devices. Pruning is a widely accepted practice to make these large models suitable to run on such small devices. It is well understood in the machine learning community that neural networks can produce good inferences even after pruning a substantial amount of their internal parameters (weights) [60–62]. In *Channel Pruning*, entire channels (or filters) are assessed for their importance to determine if these may be removed [63] to produce a slimmer network from the original one, with minimal drop in inference accuracy. Unlike other pruning methods, this produces a compact dense network suitable for the already optimized dense convolutional routines [64].

Several routines exist to perform the convolution operation, although two are dominant across the majority of libraries:

- Direct convolution – this method shifts each filter (channel) one position at a time over an input image with a deep nested loop. This requires the least amount of extra memory, which makes it ideal for devices with limited physical memory, although it is also very slow in terms of computation time.
- General Matrix Multiplication (GEMM) – this method performs the convolution by unrolling each image patch to convolve over into a column of a larger matrix of unrolled patches, while filters (channels) are unrolled into rows to form a

second large matrix, in a process known as *image2col* [65]. The entire convolutional operation over the input image is performed by a single operation of matrix to matrix multiplication on the two large matrices resulting from the unrolling process mentioned earlier. This is a very popular approach due to the readily available, highly optimised matrix multiplication libraries (Blas, CUDA), which make it fast in practice.

2.6.8.1 Channel Pruning

Current large CNNs require some alteration to make them suitable for deployment on smaller devices, which often comes in the form of pruning. Weight pruning, through which some weights based on a signal are reduced to zero [61], is one approach that works well with accelerators of sparse algebraic operations, although the speedup these can offer on general purpose devices has been questioned [64]. Another approach for network size reduction is *channel pruning*, in which entire channels are eliminated if their impact is minimal [63], resulting in better performance than other compression techniques [64], and can be modeled with both accuracy and inference time constraints [66].

As a machine learning technique, CNN pruning is generally performed away from the runtime environment, with the primary metric for the task being inference accuracy. Retraining the model during the pruning process requires substantially more computing resources so this is generally performed on other machines than the final inference device.

Channel Pruning is performed as follows. Assuming the c -th convolutional layer of a neural network has n filters (channels) $k_i, i \in [1, n]$ (before pruning). To prune channel p , with $1 \leq p \leq n$, the new convolutional layer will have a number of $n - 1$ channels and each channel $k_i, i \in [p + 1, n]$ will be re-indexed to $i = i - 1$. For example, in a convolutional layer with 128 channels, pruning the 25-th channel will produce a compact layer with channel 26 becoming channel 25, and so on for the following channels re-indexing to $i - 1$, thus producing a new convolutional layer with channels indexed continuously from 1 to 127. This process is repeated for each pruned channel. As can be observed, by this process the same computation time will be produced no matter which channel is picked for pruning, so we eliminate channels sequentially for our inference time analysis.

2.6.8.2 Models

To generalize the observation of pruning patterns we select three popular deep neural networks prevalent in computer vision for image classification:

- ResNet-50 [67] has 50 layers and consists of residual blocks. There are 23 convolutional layers with filters of size 3×3 and 1×1 (referred to as ResNet.L i , where i is the layer index), and interleaved with other layers, such as batch normalization. Although they are indexed, we do not profile their performance here due to their cost being insignificant. Convolutional layers have a number of filters between 64 and 2048 [67].
- VGG-16 [68], is a feed-forward network with 13 convolutional layers and 3 fully connected layers. Each convolution uses 3×3 size filters. The convolutional layers are indexed similarly to ResNet, with 0, 2, 5, 7, 10, 12, 17, 19, 24 unique shapes (where the convolutional layer shape is repeated in the network, it is considered only once). These convolutional layer have the following number of filters: 64, 64, 128, 128, 256, 256, 512, 512, and 512 respectively.
- AlexNet [49] is the earliest CNN to win the ImageNet competition by a huge margin over the previous top machine learning solution. Compared to more recent CNNs this has only 5 convolutional layers, indexed 0, 3, 6, 8, 10 interleaved by Pooling and Dropout layers. The unpruned convolutional layers have the following number of filters: 64, 192, 384, 256, and 256 respectively.

2.6.8.3 Arm Compute Library

We use CNN implementations provided by the Arm Compute Library, which was described in section 2.2.3.1.

2.6.8.4 TVM Compiler

We use CNN implementations provided by TVM, which was described in section 2.2.3.2.

2.7 Generating New Benchmarks

We collect vast amounts of data to validate our prediction efforts, the majority using compute kernels sourced from benchmark suites presented in Section 2.6. From each

benchmark suite, we source a variety of kernels, and execute them with a number of different inputs. In addition to standard benchmarks suites, we use DeepSmith, a framework designed to test OpenCL compilers, which is able to generate random, but real OpenCL kernels. DeepSmith is described in detail in Section 2.6.6.

2.7.1 Measurement

Kernel runtimes on the HIKEY-960 development board are noisy. While the kernel drivers on the HIKEY-960 do not expose information about the dynamic clock frequency, we see multiple available frequencies in the Linux device tree. Furthermore, we are able to access temperature monitoring data through the kernel, and observe variations in temperature on the board which exceed safe limits, prompting the conclusion that the frequency is being scaled down due to the development board overheating. Figure 2.17 shows the ranges of results collected from running a GEMM kernel with various input sizes on the HIKEY-960. The results show that an order of magnitude difference can be observed in the runtimes using the same kernel and inputs.

We take the following steps to stabilize and filter the runtimes in order to obtain reproducible results. While the kernel does not expose the option to manually set the clock frequency, we can force the changes by allowing only specific frequencies in the device tree. The device tree allows for six different frequencies for the G71 GPU, and as such, we create six separate device trees with which to boot Linux - each with a single possible frequency. To keep the board from overheating, we place the board directly in front of a large office fan, which blows cool air across the surface of the board (see Figure 2.16). Using this setup however, we still see some variation in execution times. Figure 2.20 shows the average observed runtimes broken down by GPU frequency, Figure 2.19 shows the maximum, and Figure 2.18 shows the minimum.

While we can now control the frequency and the temperature of the board, there are still other sources of variation which we can't control. The MALI-G71 GPU shares system resources with the CPU, and as such, GPU performance can be indirectly affected by programs executing on the CPU through resource contention. For example, as the CPU and GPU share memory, the GPU memory could be paged out due to memory-intensive CPU programs executing in parallel. Alternatively, since the GPU communicates with the CPU via memory-mapped registers and interrupts, the GPU has to wait for the CPU to handle the interrupt, which may not be instant. Future work on performance modelling could take these factors into account, however for the time



Fig. 2.16: Cooling a Hikey-960 Development Board to reduce effects of throttling due to temperature fluctuation.

being, we start with modelling just the GPU performance. In our data collection, we use the highest available frequency (1.037 GHz). We execute each kernel 100 times, and we take the minimum value as the ground truth, as we believe this to be the run with the least interference from the remainder of the system.

2.8 Conclusion

This chapter presented information necessary for the understanding of this thesis. Section 2.2 introduced the GPU programming model, Section 2.3 introduced the Arm Mali Bifrost architecture, Section 2.4 and Section 2.5 introduced the supporting simulation frameworks used in the work leading to this thesis, Section 2.6 presented benchmarks used to develop and evaluate our GPU simulator, and demonstrate its uses, and finally, Section 2.7 presented strategies for collecting data from hardware. Chapter 3 continues with a more detailed analysis of simulation techniques and existing GPU simulation frameworks, while simultaneously developing the motivation for the work behind this thesis.

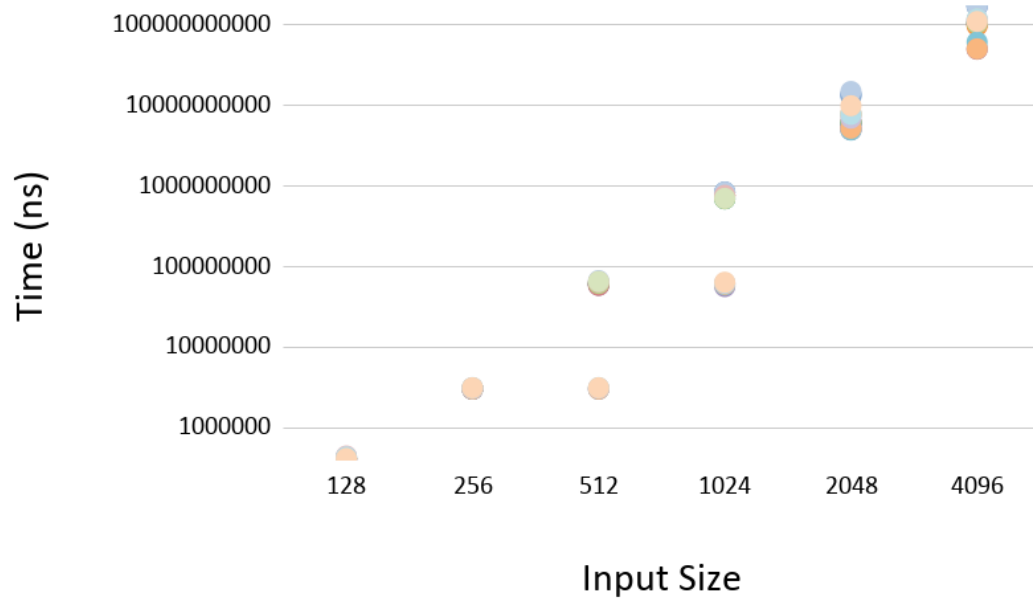


Fig. 2.17: GEMM runtimes vary significantly on each input size.

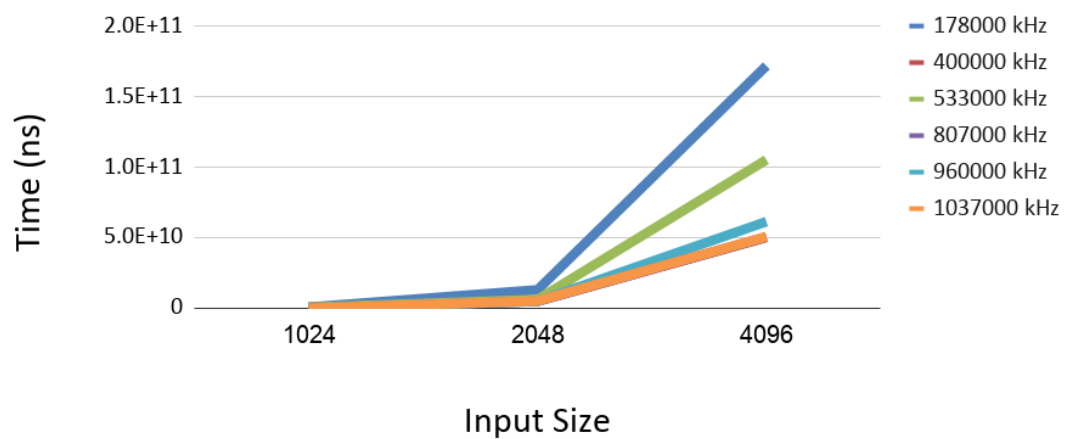


Fig. 2.18: Minimum GEMM runtimes at fixed GPU frequencies.

Application		Description
BFS	Bread-First Search	Computes the shortest-path cost from a single source to every other reachable node in a graph of uniform edge weights by means of a breadth-first search.
CUTCP	Distance-Cutoff Coulombic Potential	Computes the short-range component of Coulombic potential at each grid point over a 3D grid containing point charges representing an explicit-water biomolecular model.
HISTO	Saturating Histogram	Computes a moderately large, 2-D saturating histogram with a maximum bin count of 255. Input datasets represent a silicon wafer validation application in which the input points are distributed in a roughly 2-D Gaussian pattern.
LBM	Lattice-Boltzmann Method Fluid Dy- namics	A fluid dynamics simulation of an enclosed, lid-driven cavity, using the Lattice-Boltzmann Method.
MM	Dense Matrix-Matrix Multiply	One of the most widely and intensely studied benchmarks, this application performs a dense matrix multiplication using the standard BLAS format.
MRI-GRIDDING	Magnetic Resonance Imaging - Gridding	Computes a regular grid of data representing an MR scan by weighted interpolation of actual acquired data points. The regular grid can then be converted into an image by an FFT.
MRI-Q	Magnetic Resonance Imaging - Q	Computes a matrix Q, representing the scanner configuration for calibration, used in a 3D magnetic resonance image reconstruction algorithms in non-Cartesian space.
SAD	Sum of Absolute Differences	Sum of absolute differences kernel, used in MPEG video encoders. Based on the full-pixel motion estimation algorithm found in the JM reference H.264 video encoder.
SPMV	Sparse-Matrix Dense-Vector Multi- plication	Computes the product of a sparse matrix with a dense vector. The sparse matrix is read from file in coordinate format, converted to JDS format with configurable padding and alignment for different devices.
STENCIL	3-D Stencil Opera- tion	An iterative Jacobi stencil operation on a regular 3-D grid.
TPACF	Two Point Angular Correlation Function	TPACF is used to statistically analyze the spatial distribution of observed astronomical bodies. The algorithm computes a distance between all pairs of input, and generates a histogram summary of the observed distances.

Table 2.2: Benchmark descriptions for the Parboil benchmark suite. Table replicated from [38].

Application	Description
2mm	2 Matrix Multiplications ($D=A.B$; $E=C.D$)
3mm	3 Matrix Multiplications ($E=A.B$; $F=C.D$; $G=E.F$)
adi	Alternating Direction Implicit solver
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
correlation	Correlation Computation
covariance	Covariance Computation
doitgen	Multiresolution analysis kernel (MADNESS)
durbin	Toeplitz system solver
dynprog	Dynamic programming (2D)
fdtd-2d	2-D Finite Different Time Domain Kernel
fdtd-apml	FDTD using Anisotropic Perfectly Matched Layer
gauss-filter	Gaussian Filter
gemm	Matrix-multiply $C=\alpha.A.B+\beta.C$
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
gramschmidt	Gram-Schmidt decomposition
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
ludcmp	LU decomposition
mvt	Matrix Vector Product and Transpose
reg-detect	2-D Image processing
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k operations
syrk	Symmetric rank-k operations
trisolv	Triangular solver
tmm	Triangular matrix-multiply

Table 2.3: Benchmark descriptions for the Polybench benchmark suite. Table replicated from [39].

Application	Description
AESDecryptDecrypt	Advanced Encryption Standard encryption and decryption.
BinarySearch	Binary search through an array of elements.
BinomialOption	European option pricing (financial engineering).
BitonicSort	Sorts a sequence of numbers using bitonic sorting algorithm.
BlackScholes	Black Scholes model for European option pricing.
BoxFilter	Box filter, also known as average or mean filtering for noise reduction in an image.
BoxFilterGL	Box filter, also known as average or mean filtering for noise reduction in an image, using OpenGL.
DCT	Discrete Cosine Transform used to transform compressions of 1D and 2D signals such as audio, images, video.
DwtHaar1D	Basic one-dimensional Haar Wavelet transform.
EigenValue	Eigenvalue decomposition on a symmetric tridiagonal matrix.
FFT	Signal conversion traditionally used in engineering, music, science, mathematics.
FastWalshTransform	Efficient implementation of Walsh-Hadamard Transform, a generalized class of Fourier transforms.
FloydWarshall	Dynamic programming approach to compute the shortest path between each pair of nodes in a graph.
FluidSimulation2D	Lattice-Boltzmann method for simulating a fluid on a 2D grid.
Histogram	Assign all values to bins to create a histogram.
HistogramAtomics	Assign all values to bins to create a histogram, usign atomic operations on the GPU.
Mandelbrot	Fractal curve generated from calculating the mandelbrot set.
MatrixMulImage	Matrix multiplication using an image as input.
MatrixMultiplication	2D Matrix Multiplication.
MatrixTranspose	Flips matrix over its diagonal.
MemoryOptimizations	Kernels copying memory using different primitive data sizes and shapes.
MersenneTwister	A widely used pseudorandom number generator.
MonteCarloAsian	Option pricing using Monte Carlo analysis.
MonteCarloAsianDP	Option pricing using Monte Carlo analysis (double precision).
NBody	Simulation of a large number of particles under the influence of physical forces..
PrefixSum	Cumulative sum of a sequence of numbers.
QuasiRandomSequence	Generates points in the sobol sequence.
RadixSort	Radix based sorting algorithm.
RecursiveGaussian	Recursive gaussian filtering for digital signal processing.
Reduction	Divides array into blocks, sums blocks, then sums the block sums.
ScanLargeArrays	Scans arrays of size $> 2 * \text{MAX_BLOCK_SIZE}$, based on prefix sum.
SimpleConvolution	Convolution filter used in image processing - blur, smooth effects, or edge detection.
SimpleImage	Converts 2D to 3D images.
SobelFilter	Sobel edge detection filter.
URNG	Generates noise in an image.

Table 2.4: Benchmark descriptions for the AMD APP SDK benchmark suite [42].

Application	Description
Leukocyte	Detects and tracks rolling white blood cells in video microscopy of blood cells.
Heart Wall	Tracks movement of a mouse heart over ultrasound images.
MUMmerGPU	High-throughput parallel pairwise local sequence alignment program.
CFD Solver	Unstructured grid finite volume solver for three-dimensional Euler equations for compressible flow.
LU Decomposition	Algorithm to calculate the solutions of a set of linear equations.
HotSpot	Tool used to estimate processor temperature based on an architecture floorplan and simulated power movements.
Back Propagation	A machine-learning algorithm that trains the weights of connecting nodes on a layered neural network.
Needleman-Wunsch	Nonlinear global optimization method for DNA sequence alignments.
Kmeans	A clustering algorithm used extensively in data-mining.
Breadth-First Search	Search algorithm that traverses all connected components in a graph.
SRAD	Speckle reducing anisotropic diffusion - a diffusion method for ultrasonic and radar imaging applications based on partial differential equations.
Streamcluster	Clustering based on distance from median value.
Particle Filter	Statistical estimator of the location of a target object.
PathFinder	Dynamic programming method to find a path on 2-D grid.
Gaussian Elimination	Solves for all of the variables in a linear system.
k-Nearest Neighbors	Finds nearest neighbors from unstructured data set based on euclidean distance.
LavaMD2	Calculates particle potential and relocation due to mutual forces within a large 3D space.
Myocyte	Models heart muscle cell and simulates its behavior.
B+ Tree	Search algorithm on an m-ary tree with a variable, but often large number of children per node.
GPUDWT	Discrete wavelet transform - digital signal processing technique.
Hybrid Sort	Sorting using two methods - bucketsort and merge-sort.
Hotspot3D	Tool used to estimate processor temperature based on an architecture floorplan and simulated power movements.
Huffman	Lossless data compression.

Table 2.5: Benchmark descriptions for the Rodinia benchmark suite [43]

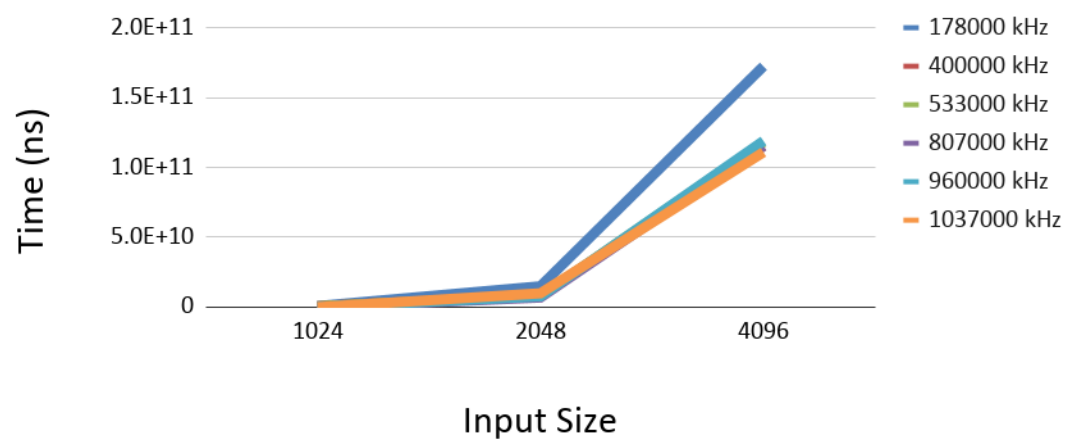


Fig. 2.19: Maximum GEMM runtimes at fixed GPU frequencies.

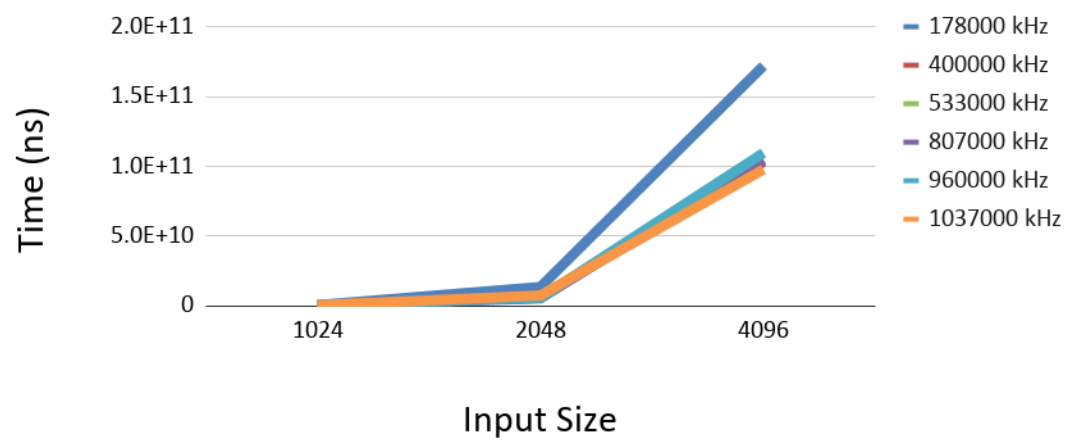


Fig. 2.20: Average GEMM runtimes at fixed GPU frequencies.

Simulation Background & Related Work

Simulation has an extensive history, and even in the relatively new GPU space, there are a number of simulators, which have paved the way for this thesis. This chapter introduces concepts which characterize simulation technology, and on which simulators are built, while simultaneously providing a qualitative evaluation of existing GPU and full-system simulators, and performance modelling techniques. The analysis directly motivates the need for a holistic, fast, full-system approach to GPU simulation, which is presented in Chapter 4. We begin with considering three main design choices that developers are faced with when designing a new simulator:

- the level of detail at which the hardware is modelled (Section 3.1),
- the simulation environment (Section 3.2),
- the level of accuracy at which the software stack is modelled (Section 3.3).

In the final section (3.4), we present performance modelling techniques beyond cycle-accurate simulation, focusing primarily on trace-based, analytical and machine learning based models. In each of these sections, we discuss existing efforts.

3.1 Speed vs. Detail

Hardware can be modelled at various levels of detail. Here, we present cycle-accurate simulation, functional instruction set simulation, and emulation, as the evaluated simulators fall into these categories. Each of these simulation techniques represents a different point on the simulation trade-off graph between speed and level of detail, presented in Figure 3.1. To ease comparison of existing GPU simulation approaches

Simulator	Full System	Guest CPU	Guest GPU	GPU ISA	GPU Toolchain	Prog. Model	Perf. Model	Simulation Model	Max. Rel. Error ¹
Barra [69]	GPU only	N/A	NVIDIA Tesla	Approx. Tesla ISA	Emulated	CUDA	Instruction-Accurate	Execution-Driven	$\leq 81.6\%$
GPGPU-Sim [70]	GPU Only	N/A	NVIDIA-like	PTX GT200 SASS	Custom	CUDA	Cycle-Accurate	Execution-Driven	$\leq 50.0\%$
gem5-GPU [71]	Yes	x86	NVIDIA GTX 580	PTX GT200 SASS	Custom	CUDA	Cycle-Accurate	Execution-Driven	$\leq 22.0\%$
Multi2Sim [41]	Yes	x86/Arm/MIPS	AMD Everg./S.Isrl. NVIDIA Fermi	AMD GCN1 SASS	Custom	OpenCL CUDA	Cycle-Accurate	Execution-Driven	$\leq 30.0\%$
Multi2Sim Kepler [72]	Yes	x86/Arm/MIPS	NVIDIA Kepler	SASS	Custom	CUDA	Cycle-Accurate	Execution-Driven	$\leq 200\%$
ATTILA [73]	GPU Only	N/A	ATTILA	ARB	Custom	OpenGL	Cycle-Accurate	Execution-Driven	N/A ²
GPUOcelot [74]	GPU Only	N/A	NVIDIA AMD Radeon	PTX	Custom	CUDA	Instruction-Accurate	Trace-Based	Not Evaluated ³
HSAemu [75]	Yes	Retargetable/Arm-v7A	Generic	HSAIL	Custom	OpenCL	Cycle-Accurate	Execution-Driven	N/A ²
GPUTejas [76]	GPU Only	N/A	NVIDIA Tesla	PTX GPUOcelot μ -ops	Custom	CUDA	Cycle-Accurate	Trace-Driven	$\leq 29.7\%$
MacSim [77]	Yes	x86	NVIDIA GeForce G80/GT200/Fermi	PTX GPUOcelot μ -ops	Custom	CUDA	Cycle-Accurate	Trace-Driven	Not Evaluated ³
TEAPOT [78]	Yes	Generic	Generic Mobile GPU	Emulated	Custom	OpenGL	Cycle-Accurate	Trace-Driven	N/A ²
QEMU/MARSSx86/PTLSim [79]	Yes	x86	NVIDIA Tesla-like	Generic	Custom	OpenGL	Cycle-Accurate	Execution-Driven	Not Evaluated ³
GemDroid [80]	Yes	x86/Arm-v7A	ATTILA [73]	ARB	Custom	OpenGL	Cycle-Accurate	Execution-Driven	N/A ²
GCN3 Simulator [81]	Yes	x86	AMD Pro A12-8800B APU	GCN3	Vendor	ROCM	Cycle-Accurate	Execution-Driven	$\sim 42\%$
MGPU-Sim [82]	No	N/A	AMD Radeon R9 NANO	GCN3	Custom	ROCM	Cycle-Accurate	Execution-Driven	$\sim 20\%$
Accel-Sim [83]	No	N/A	NVIDIA Volta, Kepler, Pascal, Turing	SASS PTX	Vendor	CUDA	Cycle-Accurate	Execution/Trace-Driven	$\sim 30\%$
Our Simulator	Yes	Retargetable/Arm-v7A/v8A	Retargetable/Arm Mali-G71	Retargetable/Native Binary	Vendor	Any/OpenCL	Instruction-Accurate	Execution-Driven	0.0%

¹ Maximum error of a performance metric reported in the original publication.

³ Original publication does not provide an accuracy evaluation against a hardware implementation of the simulated GPU.

Table 3.1: Feature comparison of existing GPU simulators. Our simulator (presented in chapter 4) is the only full-system CPU/GPU mobile platform simulator capable of hosting an unmodified GPU software stack and supporting true GPU native code execution.

we provide an overview of features in Table 3.1, including each simulator’s maximum relative error as reported in their original publications.

3.1.1 Cycle-Accurate Simulation

Each hardware component of a GPU is composed of wires, registers, and gates, forming a physical block. A cycle is an electrical pulse that propagates through this block. During each cycle, a fixed amount of work can be completed by the hardware. Some operations require multiple cycles to complete. Cycle-accurate simulators model the target architecture on a cycle-by-cycle basis. By modeling not only the functional behaviour, but also the micro-architecture, they are often used to estimate the performance of the modeled system. There are numerous examples in both the CPU and GPU space, the most common of which are gem5 [71], GPGPUSim [84], and Multi2Sim [85], and Accel-Sim [83]. Typically, these simulators are implemented in

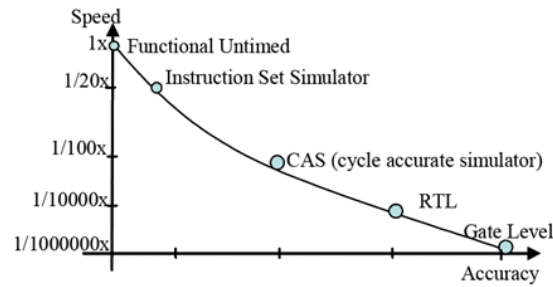


Fig. 3.1: Comparison of simulation speed at different levels of abstraction for modelling SoCs, Source: <https://www.design-reuse.com/articles/18418/modelling-embedded-systems.html>

C++, and are most commonly used in later design phases, for example when exploring various micro-architecture implementations, memory subsystems, power, and energy consumption. [70, 71, 73, 75–83, 85] are all GPU simulators, which are considered cycle-accurate. Despite this, the error figures in Table 3.1 show that performance predictions from these simulators are still very far from the actual hardware performance. From the hardware perspective, some of this error can be explained (but not necessarily justified) by the fact that many existing GPU simulators do not model existing commercial GPUs, but only *simplified GPU architectures* [73]. Additional reasons for the error exhibited by GPU simulators are explored later in this chapter.

Of the existing cycle-accurate simulators, MGPUSim [82] takes the most progressive approach to modelling GPUs, by developing explicit guidelines for developing a good simulator. By simulating parallel components of the GPU in parallel, the authors are able to increase the simulation rate by 16.5x and 33.8x relative to Multi2Sim and GPGPUSim, respectively, while keeping the average error to 5.5%. However, we note that the accuracy is evaluated on a set of only 7 application benchmarks and four microbenchmarks. Nevertheless, parallel simulation is a hard problem, and MGPUSim makes significant progress in solving it.

Recently released Accel-Sim [83] also provides significant improvement over previous GPU simulators, by being the first Nvidia GPU simulator which can simulate unmodified binaries at the Shader Assembly (SASS) level. SASS is the native instruction set of the GPU, whereas many previous simulators only modelled PTX, an intermediate representation. This approach, and the simulation infrastructure however, only apply to Nvidia GPUs. Accel-Sim is further discussed in the trace-based simulation section.

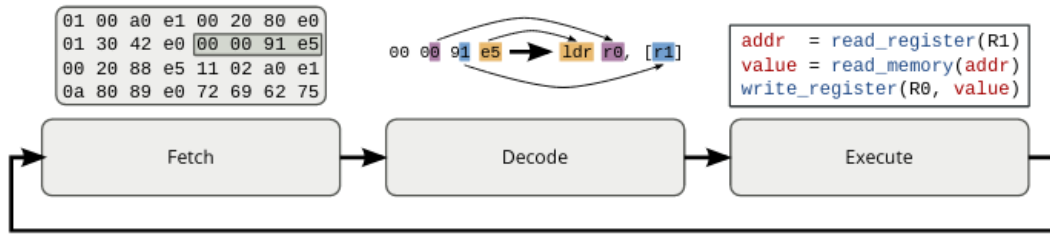


Fig. 3.2: A typical fetch, decode, execute loop implemented in interpreted simulators (Image source: [86])

3.1.2 Functional Instruction Set Simulation

The Instruction Set Architecture (ISA) is an abstract model, which sets out the processor's instruction set, supported data types, registers, and hardware support for accessing main memory (e.g., page table format and the MMU), without specifying implementation details of the processor. This means that the Instruction Set Architecture could have different implementations, optimized for different use cases, while still providing binary compatibility. For example, the Arm-compiled programs are typically binary compatible across all Arm CPUs, however there are many different implementations of Arm CPUs, such as Cortex-M processors, which are optimized for energy, and Cortex-A, which are optimized for performance.

We define a functional instruction set simulator as a simulator designed to mimic the behaviour of hardware, while modelling architecture-specific details of the hardware - the ISA. For example, an instruction set simulator that decodes instructions and executes them, without a micro-architectural model is considered to be an instruction set simulator. By trading off observability, they are able to benefit from faster simulation speeds than cycle-accurate simulators, and due to this, they are often used in early design-space exploration, and early-stage software development, before the hardware is available. Some widely used examples include the Arm Fast Models [87], and QEMU [88]. Many cycle-accurate simulators also have a functional mode, allowing for faster, less detailed simulations. For the purposes of this thesis, we interchangeably use the terms instruction set simulator and functional simulator.

Functional, instruction set simulation, can be defined by the technology that it implements - interpretation or dynamic binary translation (DBT). Functional simulators generally execute in a fetch, decode, then execute fashion, where a guest binary instruction is read from the program binary, decoded, following which its semantics are executed using a pre-defined function. This type of execution is called *interpretation*,

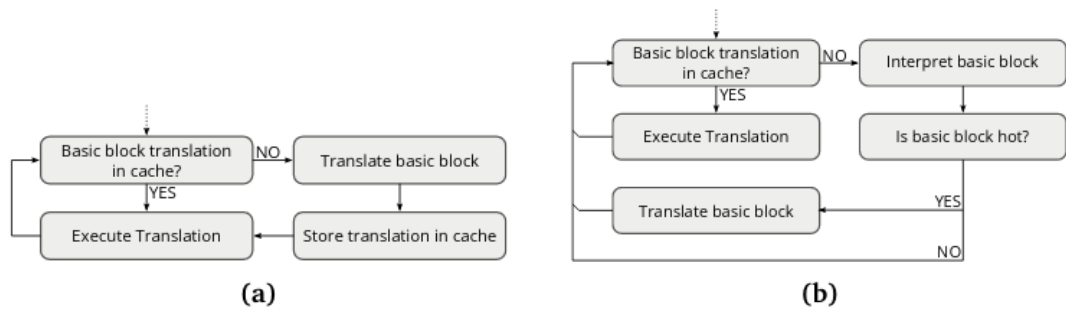


Fig. 3.3: Dynamic binary translation is usually implemented in one of these two fashions. (a) Guest basic blocks are translated on demand. (b) Basic blocks are first executed in an interpreter, until they become hot, at which point they are translated (Fig. Source: [86]).

and this fetch, decode, and execute behaviour is a large bottleneck for anything but the shortest of programs. The loop is depicted in Figure 3.2. GPUOcelot [74] and Barra [69], as well as the GPU component of our proposed full-system simulation approach are all GPU simulators implemented using interpretation.

Dynamic binary translation (DBT) is a technique used to overcome this bottleneck and accelerate simulations. Dynamic binary translators look at short sequences of code, typically basic blocks, and translate them on the fly from the guest instruction set into the host instruction set. These translated instructions are then JIT-compiled into a native binary, often with a one-to-one mapping from guest to host instruction, and executed. Furthermore, these translations are cached, so that decoding and compilation only have to occur once, regardless of how many times the basic block executes. Two different approaches to DBT are presented in Figure 3.3. Many CPU simulators are implemented using dynamic binary translation, for example [37, 88]. No current GPU simulator uses dynamic binary translation, however our simulator is implemented in a framework that would allow for it to be easily extended into a DBT. Captive [37], which does use DBT and is described in section 2.12, is used for the CPU component of our full-system simulator.

3.1.3 Emulation

We define an emulator, or an untimed functional simulator, as a type of simulator designed to mimic the behaviour of a certain piece of hardware, without implementing any architecture or micro-architecture specific features. For example, a software implementation of OpenCL, which intercepts OpenCL calls to a GPU and executes them

on the CPU, would be considered an emulator. This is the technology used in the Android Emulator [89], as well as TEAPOT [78].

3.2 Simulation Environments

The execution environment of a simulator is critical to its usefulness, faithful representation of a real hardware environment, and the accuracy of the end result. In this section, we discuss two different simulation modes - user-mode, which simulates standalone applications, and full-system simulation, which executes a full operating system and supporting software stack within the simulation environment.

3.2.1 User Mode Simulation

User-mode simulation mimics the concept of executing a standalone binary guest program within the execution environment of the host system. Most often, the binaries executing are statically linked, meaning they have no external dependencies. GPU simulation however, is driven by a software stack, which executes on the CPU. In this case, the host side CPU programs must be dynamically linked, as they depend on at least one external library - the runtime (e.g., OpenCL runtime), as well as device drivers which are part of the kernel. Most existing GPU simulators provide execution environments in the form of modified runtime libraries, which provide an interface between the user program and the simulator. This means that often, *GPUs are treated as standalone devices*, not modeling any CPU-GPU transactions [90], impacting the correctness and accuracy of the execution. Furthermore, the practice of maintaining a software stack specific to the simulator is unsustainable, as presented later in this chapter.

3.2.2 Full-System Simulation

Full-system simulation, by contrast, simulates an entire guest system within the execution environment of the host system. Functionally, a full-system simulation should be indistinguishable from execution in real hardware. The guest system comprises an operating system and user-space, enabling the user to execute guest programs and interact with them exactly as if it were a real system. A full-system simulator implements not only the instruction set of the guest CPU, but also a number of peripheral devices such as timers, IRQ controllers, UART, USB, and others. Many of these devices are

required to boot an operating system, while others are implemented for the user's convenience. Another feature of a full-system simulator includes the implementation of a realistic memory model, including an MMU. Critically, the implementation of a full-system simulator allows us to couple CPU simulation with accelerators, for example a GPU. The GPU uses control registers and shared memory to communicate with the CPU, and requires the host program, runtime library, and device drivers (all executing on the CPU) to orchestrate and receive jobs.

gem5 [91] provides one of the most popular CPU and memory design platforms. It is a configurable, cycle-accurate simulator, which provides support for booting a full, unmodified operating system for Arm, x86, RISC-V, SPARC, and Alpha architectures. gem5 has been integrated with multiple GPU simulators, as shown in [71, 81, 92]. QEMU [88] is a fast DBT-JIT based instruction set simulator which also supports full-system simulation. Captive [37], which we use as the CPU simulator in our framework is similar to QEMU, however it provides additional acceleration by taking advantage of host resources. Captive is described in Section 2.5. Full-system simulators are also available from industry, with Synopsys providing the DesignWare Arc Simulation Tools [93], and Arm providing Fast Models [87]. One of the most widely used full-system simulators is the Android Emulator [89], based on the previously discussed QEMU, which allows Android developers to test apps on a variety of simulated hardware platforms before deployment.

3.3 Modelling the Software Stack

GPU execution relies on a complex software stack, with numerous components executing on the CPU including the application code (which can constitute multiple layers), the runtime (e.g., OpenCL or Vulkan), the JIT-compiler (invoked by the runtime), the kernel driver (used to communicate between CPU and GPU), and the final binary executing on the GPU. Existing simulation frameworks implement only certain parts of this software stack.

There are a number of aspects one has to consider when designing a simulator, as described so far in this section. These design choices have significant consequences, in terms of speed and accuracy from a hardware perspective, while different simulation modes allow for different simulation environments. However, just as significant as the previous two concerns is fast, faithful, and reliable execution of the entire software stack, which existing GPU simulation approaches fail to deliver. In response, we pro-

pose a holistic approach to tackle many of the existing problems, and prevent new ones from arising, as described in later chapters. Now, we motivate the need for a *fast, functional, full-system* GPU simulation framework through an evaluation of related work. There are numerous motivations, including but not limited to:

1. Accuracy of the simulated software stack.
2. Performance of all simulated components.
3. Ease of maintenance as software is updated.
4. Ease of use and usefulness for end users.

3.3.1 Accuracy of the Simulated Software Stack

From a software perspective, GPU simulation often suffers from the following problems: (a) instruction sets are not accurately modeled, but approximated by an *artificial, low-level intermediate representation* [74,94], and (b) instead of using vendor provided driver stacks and compilers, GPU simulators often rely on *simplified system software*, which may behave entirely differently to original tools, straying from what is executed in real hardware. [41, 72].

Accurately executing the exact same software stack as is executed in real hardware in its entirety is critical to achieving an accurate overall simulation result. The GPU software stack can be broken down into a minimum of five components: the GPU instruction set, the device driver, the runtime, the JIT-compiler, and the CPU program. In modern applications, this can easily increase, with further higher level libraries being added on top of the CPU program.

Significant error is introduced by the use of outdated or non-standard GPU tool chains required by several simulators, as is demonstrated in the following examples. It is more than likely that simplified or non-vendor supplied tool chains used by other GPU simulators introduce even greater simulation error, as also highlighted in [81].

In this thesis we claim that without a truly accurate GPU simulation model and a full-system environment, capable of running an unmodified GPU software stack and applications, it is not possible to gather reliable performance metrics to underpin mobile GPU architecture research.

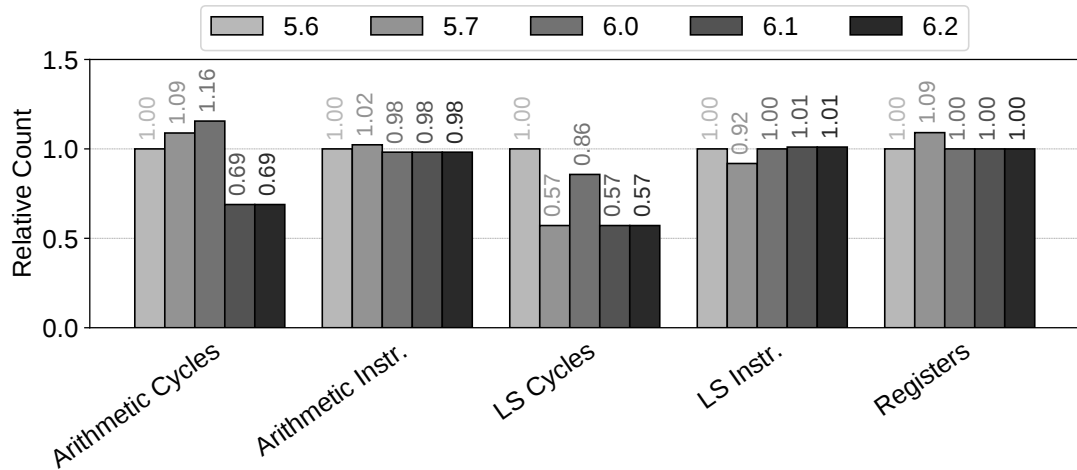


Fig. 3.4: MatrixMul: Different versions of Arm's OpenCL compiler result in substantially different code for the G-71.

3.3.1.1 GPU Instruction Set

The first software component we discuss is the binary program that executes on the GPU. When code executes on a GPU, it executes in that specific GPU's native instruction set. Despite this, a number of existing GPU simulators replace this component with an intermediate representation. In the case of GPGPUSim, instead of simulating the native, binary instruction set, PTX is used as the lowest level software component. The AMD GPU simulators originally packaged with gem5, use HSAIL. PTX and HSAIL, however, have many differences to the native GPU instruction sets. They are used as an intermediate representation common to all GPUs of the same vendor, which can then be specialized at JIT-compile time to the host instruction set. As a result, simulators modelling at this level, are modelling a common format, shared across GPUs with varying architecture and micro-architecture. Gutierrez et al. [81] show that modelling AMD GPUs at the HSAIL level, results in an average error of 75%, while changing from IR to the native instruction set lowers this error to 42%, as shown in Figure 3.6. As we can see from Figure 3.8, there is a large difference in the number of instructions executed between the two versions of simulator. Obtaining the work item of the current thread - a unique, global, identifier for the thread is written as just one instruction in HSAIL. However, in the GCN3 instruction set, the same operation takes five instructions, including a memory access, which is generally far more expensive than arithmetic operations. Similarly, as seen in Figure 3.7, loading compute kernel arguments in HSAIL is abstracted as just one instruction. However, in real execution

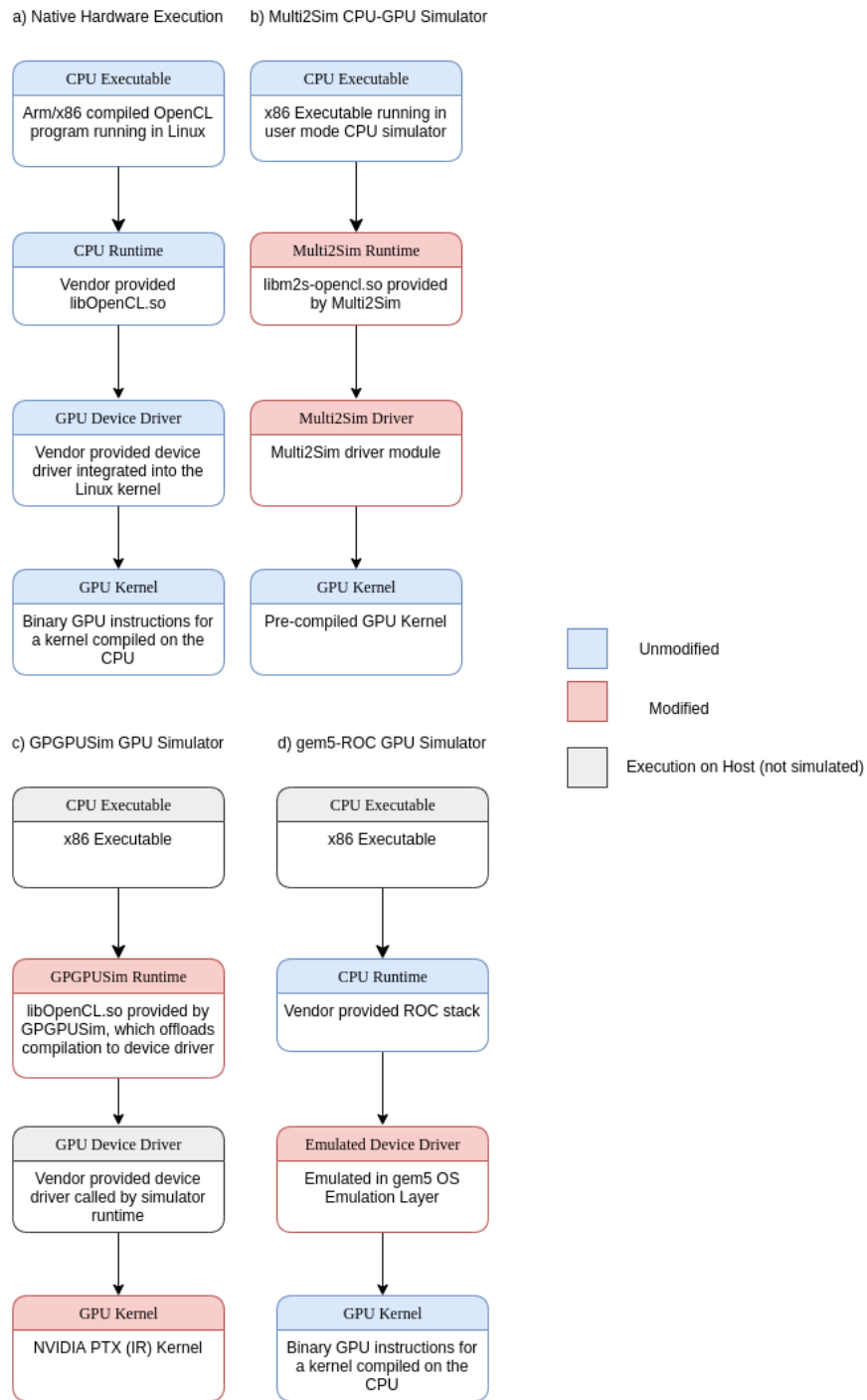


Fig. 3.5: Comparison of the GPU kernel execution model and software stack for (a) a native execution environment, (b) Multi2Sim, (c) GPGPU-Sim, and (d) gem5 with the Radeon Open Compute software stack. For MultiSim, GPGPU-Sim, and gem5, we have highlighted non-standard software components thus represent a source of inaccuracy.

Correlation		Avg. Absolute Error	
HSAIL	GCN3	HSAIL	GCN3
0.972	0.973	75%	42%

Fig. 3.6: Simulation Error Comparing HSAIL IR and GCN3, the native AMD Instruction Set (figure from Gutierrez et al. [81])

HSAIL	GCN3
#load kernarg at addr %arg1 ld_kernarg \$v[0:1], [%arg1]	# mv kernarg base to v[1:2] v_mov v1, s6 v_mov v2, s7 #load kernarg into v3 flat_load_dword v3, v[1:2]

Fig. 3.7: HSAIL vs GCN3 kernel address calculation (figure from Gutierrez et al. [81])

terms this would take three instructions, including a memory access. These examples show, that by replacing the native, binary, GPU instruction set, we already lose observability into the details of the GPU's architecture and execution, and more critically, it's performance.

3.3.1.2 Device Driver (Linux Kernel Driver)

Now that we have understood the significance of executing an accurate binary GPU program, we move onto the GPU device driver. The GPU device driver acts as the interface between the CPU and the GPU. It is responsible for all interaction between the CPU and the GPU, which includes creating and configuring jobs to be dispatched to the GPU, and handling the transfer of results back to the CPU as well as handling any faults. At a lower level, this includes mapping and managing memory available to

HSAIL	GCN3
# Return Absolute WI ID workitemabsid \$v0, 0;	# Read AQL Pkt s_load_dword s10, s[4:5], 0x04 # wait for number of s_loads = 0 s_waitcnt lgkmcnt(0) # Extract WG.x Size from Pkt s_bfe s4, s10, 0x100000 # Calculate WG.x ID s_mul s4, s4, s8 # Calculate Global TID v_add v117, vcc, s4, v0

Fig. 3.8: HSAIL vs GCN3 work item id calculation (figure from Gutierrez et al. [81])

the GPU, handling interrupts between the CPU and GPU, and writing to and reading from the GPU's memory mapped registers. The GPU we are modelling uses the Arm Bifrost architecture.

Memory model - In this architecture, the CPU and GPU share memory, and the GPU has no dedicated memory. Instead, to reduce memory latency, the architecture implements a complex, multi-level cache hierarchy. The GPU also has its own Memory Management Unit (MMU), which supports multiple address spaces, meaning that different jobs executing on the GPU can have different virtual-to-physical address mappings. In a situation like this, multiple jobs might all be accessing different memory buffers, or they can be using different virtual address spaces, but access the same physical address space. Without the device driver mapping this memory and preparing the page tables, the GPU would have to execute in a flat address space, forcing the simulator developer to make a decision to either use the same address space for all jobs, or to separate all jobs into different virtual address spaces that don't map to the same physical address space - neither of which are realistic. This would have further implications for TLB modelling, cache modelling, and in the end, would result in an inaccurate overall cycle count.

Another feature of this shared memory model is that the CPU and GPU can share page tables, which allows the CPU and GPU to use the same virtual-to-physical address mappings. Again, without the support of the device driver, modelling this would not be possible, leading to inaccurate modelling of the cache behaviour.

Multi-kernel workloads - The device driver receives information for job creation from the OpenCL runtime. This information includes a pointer to the program binary, pointers to memory buffers, and additional metadata describing the execution modes. The device driver consolidates all of this information, constructs the necessary data structures in memory, and once they are all ready, it signals the information to the GPU by writing values into the GPU's memory mapped registers. An embedded system with a CPU and GPU is a highly interactive one. Often, there are multiple kernels executing on the GPU, and continuously communicating their results to the CPU. For example, the SLAMBench [44] benchmark suite executes 12 different kernels, but the data and interaction between them is managed by the CPU. Faithful execution of the device driver is fundamental to correct modelling of multi-kernel workloads.

Fault Handling - The Bifrost kernel driver is responsible for handling any faults raised by the GPU. This includes not only any faults resulting from errors, but also situations that require further dynamic interaction between the CPU and GPU. For

example, in cases where a large amount of memory is required by the GPU, a lazy memory allocation scheme is used, and not all of it is allocated immediately. In these cases, the driver will allocate a portion of memory, which the GPU will operate on, and at some point during the execution, the GPU will encounter a translation fault, *i.e.* the address it is trying to access isn't mapped. At this point the GPU will write status information about the fault to specific memory mapped registers, and then raise an interrupt, signalling that a fault has occurred. The device driver executing on the CPU will read the information from the registers, handle the fault, by allocating more memory, updating the page tables, and signalling to the GPU that it can continue. Without a faithful, full-system simulation, this type of interaction would not be possible.

3.3.1.3 OpenCL Runtime

We have seen how critical it is to accurately and completely execute the GPU binary and device driver. The next component of the stack is the OpenCL runtime, which implements the interface visible to the programmer, controlling all aspects of the GPU execution. The programmer uses specific functions from the OpenCL API in order to select the OpenCL device, initialize memory buffers, create the OpenCL program, and dispatch jobs to the GPU. The OpenCL runtime is also used by a number of libraries, as the lowest level exposed interface to the GPU. The Arm Compute Library, for example, implements a number of neural networks, which are implemented as a series of OpenCL kernels. TVM [26] builds on top of the Arm Compute Library, while the Lift optimizing compiler [95] compiles higher-level kernels written in a functional programming language down to OpenCL.

OpenCL isn't completely transparent however, as it comes with no performance guarantees, meaning that different configurations of the same program may have different relative performance across different architectures. Furthermore, the vendor-supplied implementation can include heuristics for optimization, which do not always perform as expected. One example is automatic workgroup size and shape selection, which impacts how threads are grouped and scheduled, and which in turn can shift the performance bottleneck to different GPU components. In this example, the user can easily take control by specifying the workgroup size, however, we observed further unexpected performance variation when optimizing Convolutional Neural Networks (CNNs) on a Mali-G71 GPU. Channel pruning is a technique expected to improve accuracy of CNNs, however, in Figure 3.10, we demonstrate that channel pruning can result in significantly longer execution times on a Mali-G71 GPU. In Section 4.6.3, we

```

// Create graph
graph << common_params.target
    << common_params.fast_math_hint
#if defined (M_DIRECT)
    << arm_compute::graph::ConvolutionMethod::Direct
#elif defined (M_GEMM)
    << arm_compute::graph::ConvolutionMethod::GEMM
#elif defined (M_WINOGRAD)
    << arm_compute::graph::ConvolutionMethod::Winograd
#endif

```

Fig. 3.9: This is how the programmer specifies to use the GEMM kernel from the Arm Compute Library.

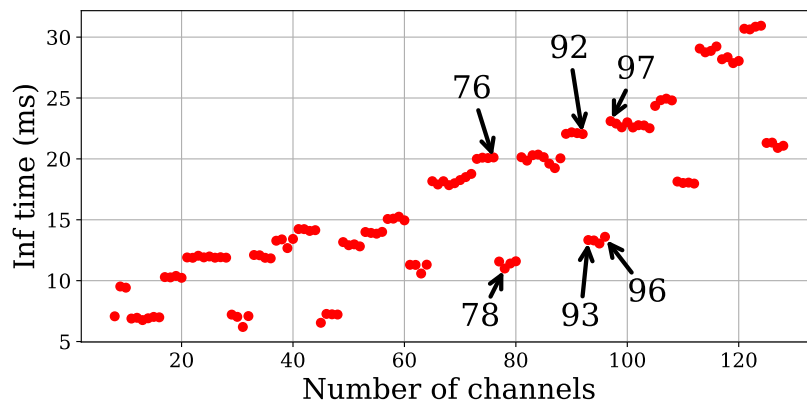


Fig. 3.10: Execution pattern observed for channel pruning of ResNet-50 layer 16 implemented with Arm Compute Library GEMM on HiKey 970 Mali GPU.

show that this is due to choices made by the OpenCL runtime, which are invisible to the user.

Some GPU simulators, for example Multi2Sim, replace the native runtime with a custom, simulator-specific version. However, the implementation of the runtime library has a significant impact on GPU performance, which cannot accurately be recreated with custom runtimes. Therefore, simulations executing with a custom runtime would manifest vastly different behaviour compared to the real hardware platform, with the fault lying not with the GPU simulator itself, but at a higher level of the software stack, which drives the GPU simulation.

With modified or completely replaced versions of the OpenCL runtime, we lose observability over these nuances, and our simulations stray further away from reality.

3.3.1.4 OpenCL Compiler

Just as it is critical to execute the real, vendor-provided OpenCL runtime, it is also critical to execute the real compiler, which is often packaged with the runtime. Compiler toolchains develop at a rapid pace. In example, since the release of the Hikey-960 development board, the first development board with a Bifrost GPU, 28 versions of the OpenCL runtime, compiler, and kernel driver have been released. Successive versions of the toolchain provide bug fixes and optimizations that could radically change the generated code, and the execution of it in hardware. As such, it is critical to keep up to date with the most recent software stack, both for software development and hardware exploration. Our full-system simulator provides full flexibility, as any toolchain that can be executed on the real hardware can also be executed in simulation. This is counter to toolchains used by existing state-of-the-art GPU simulators. Multi2Sim replaces the vendor provided OpenCL runtime and compiler with its own version, which helps with code generation for the simulator. As such, it is developed with specific OpenCL features in mind, which inadvertently results in Multi2Sim being forcibly tied to a specific toolchain. For example, Multi2Sim was developed when AMD toolchain 2.5 was available. Since then, that toolchain was replaced by newer versions, and is no longer available publicly. However, when using the oldest currently available versions (2.7, 2.8), different code is generated for the same kernels. The generated binaries contain different sections. They also contain different instructions, and the runtime takes advantage of different OpenCL features, all of which prevent the simulation from completing due to missing features in Multi2Sim. As a result of all of these changes, Multi2Sim is tied to a specific version of the toolchain, which is no longer available. This problem is further examined in section 3.3.3.

Being tied to a specific toolchain limits the lifespan and usefulness of a simulation framework. Toolchains develop quickly, and problems that architects are working on may already be solved in software, unbeknownst to the architect, or, the toolchain may simply be no longer available, preventing users from using the simulation framework. The binary code generated from two different versions of a compiler, from the same source code, can be vastly different. We compiled a number of different benchmarks with different versions (5.6, 5.7, 6.0, 6.1, 6.2) of the Arm Mali Bifrost Offline Compiler, which presents static characteristics of the benchmark. Figure 3.11 shows the result for the matrix multiplication kernel - a workload commonly accelerated on GPUs. Each measured category shows differences in the compiler code between ver-

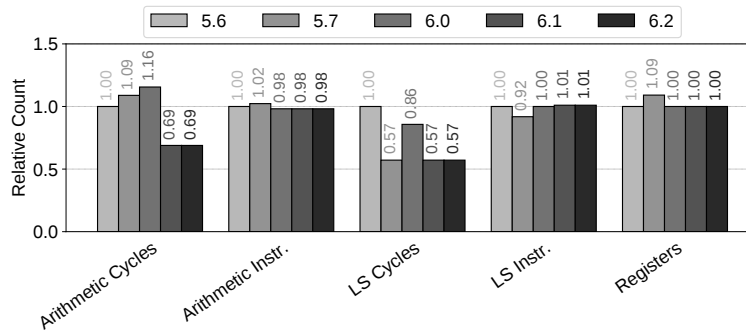


Fig. 3.11: The same code compiled by different versions of the compiler can have vastly different characteristics.

sions of the compiler. We can see from this figure, that the number of arithmetic cycles can differ by as much as 47%, and the number of load/store cycles by as much as 43%. As toolchains develop, it is necessary to be able to use them in the simulation. Having full flexibility and control over the version of the toolchain is a critical component of any accurate simulation.

3.3.1.5 OpenCL Host Program

OpenCL kernels execute on the GPU, but the setup and dispatch are controlled from code that is executing on the CPU, and it is important to faithfully execute this component as well during a full-system simulation. The user can either explicitly control the GPU through the OpenCL API, or, as often is the case, a higher-level library can make the calls to OpenCL.

The Arm Compute Library provides multiple implementations of the same kernel, optimized for different settings, or inputs. The choice of which one to use will often depend on dynamic runtime information, held by the code executing on the CPU. Selecting and simulating a specific kernel without this runtime interaction wouldn't be representative of a real workload, as the incorrect parameters, and in result, the incorrect kernel or kernel configuration could be chosen. For example, there are 22 different implementations of GEMM (Generalized Matrix Multiplication) in the Arm Compute Library, and the choice of which one to execute depends on the inputs to the program.

The driver for Direct Convolution, also implemented in the Arm Compute Library, contains runtime heuristics for optimal workgroup size selection, using the library's predictions based on runtime information about the incoming data. Section 3.3.1.3 shows how different versions of convolution can be dispatch from the Arm Compute

Library.

OpenCL programs are often interactive with the system, and can delay decisions until the point of dispatch. For example, the CPU-side program can query the system to identify how much local memory a system has, or how many different OpenCL devices are available, before deciding on a specific kernel or configuration to dispatch.

This section has shown that supporting the full software stack through a full-system simulator is critical to providing a faithful execution environment. However, full-system simulation implies that additional, costly simulation is needed. In the next section, we investigate the performance implications of executing CPU code alongside GPU code.

3.3.2 Speed of Simulation

Consideration should be given to all components of the simulation, in order to avoid bottlenecks in the simulator. In the context of GPU simulation, this includes both GPU and CPU simulation. The size of the software executing on the GPU is relatively small compared to the CPU. The CPU side software includes the entire operating system with multiple processes running, the device driver, runtime system, and user code, while the GPU executes just the kernel code, albeit with thousands of threads executing in a highly parallel fashion. Using existing CPU-GPU simulators, we have observed that the CPU simulation quickly becomes a bottleneck for the simulation.

Multi2Sim, for example, is capable of executing some components of a CPU simulation - it executes the CPU-side user program, and a modified OpenCL driver - but no Operating System. However the interpreted simulation behind Multi2Sim limits Multi2Sim's usefulness on larger workloads. Figure 3.12 shows how the CPU runtime increases as the size of the input into the GPU kernel increases. At a relatively modest by today's standards image size - 1536x1536 - the CPU side of the simulation is 32 times slower than for the smallest tested image size - 256x256. The Multi2Sim CPU-GPU framework, while it is a step in the right direction, is limited by its CPU simulation performance.

A fast, full-system GPU simulator not only enables realistic workloads through executing real software, but also provides simulations that are orders of magnitude faster than competing frameworks, and enables high volumes of data to be collected in small amounts of time.

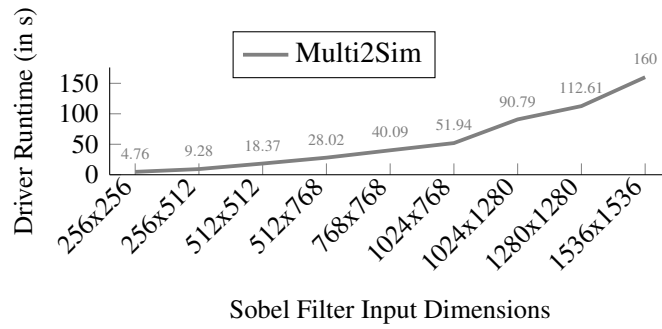


Fig. 3.12: The software stack executing on Multi2Sim.

3.3.3 Ease of maintenance as software is updated

Further complications related to using modified software stacks arise from the inevitable need for maintenance of the software stack. Issues arising from not using the correct version of the compiler were demonstrated in Section 3.3.1.4. We described how Multi2Sim replaces the vendor-provided OpenCL runtime and compiler with its own version, which helps with code generation for the simulator. As such, it is developed with specific OpenCL features in mind, which inadvertently results in Multi2Sim being forcibly tied to a specific toolchain. For example, Multi2Sim was developed when AMD toolchain 2.5 was available. Since then, that toolchain was replaced by newer versions, and is no longer available publicly. However, when using the currently available versions (2.7, 2.8), different code is generated for the same kernels. The generated binaries contain different sections, different instructions, and the runtime also takes advantage of different OpenCL features. Many of these features weren't considered during the development of Multi2Sim, and as such, Multi2Sim often fails to decode, and then execute, binaries that are generated by newer versions of the toolchain. These missing features aren't integral components of the GPU architecture itself - rather, they are inherent to the supporting toolchains.

The gem5-APU model [81], also referred to in later chapters as gem5-ROCm, exhibits similar problems, despite being actively maintained. The infrastructure is closely tied to ROCm 1.6, which is significantly outdated. Additionally, we have found that much of the infrastructure is tied to deprecated versions of Python and cmake, further impacting usability and ease of maintenance. The authors acknowledge this is a problem, and work is currently underway to implement a solution similar to our own (presented in 4) in gem5 [96].

By using a full-system simulation framework, we automatically support any and all

features provided by any toolchain, including future releases of toolchains, for a given hardware platform. Using our approach, the software isn't aware that it is executing in a simulation, and therefore no simulator-specific code needs to be implemented in the software stack. The simulator simply sees the executing code as a sequence of machine instructions, which it decodes and executes.

Critically, a full-system simulator allows the user to update the drivers as and when they please, just like in a real system. For example, this can be done through the native package manager, or by downloading the relevant binaries from the vendor, and placing them in the correct location in the system.

3.3.4 Usability

Usability is an important component of developing a simulator. In order to be able to efficiently use a simulator, it needs to be as close to the real system as possible. In the case of our Mali full-system simulator, it is completely indistinguishable from a real system. The system boots a Linux operating system, uses vendor-provided device and user space drivers, and inherently supports any workload that can be executed on a physical device implementing the same architecture. Furthermore, the system provides fast enough simulation to be interactive.

Existing GPU simulators on the other hand, require far more steps in the setup, which are not representative of the real system. By default, Multi2Sim requires the GPU binaries to be pre-compiled, while in the majority of cases GPU kernels are JIT-compiled at dispatch time to the GPU. This setup adds additional overhead for the user, and may discourage the user from further exploration. In many cases, kernels are specialized to the platform that they are executing on during JIT-compilation. The scope of optimizations when pre-compiling GPU kernels is greatly reduced.

The Arm proprietary simulator operates on hexdumps, which are portions of memory specifically extracted from GPU execution, which can then be fed through the simulator. The hexdump format also includes information about interrupts and memory-mapped registers, which the simulator can then use to dispatch the job. This type of hexdump however, is not trivial to generate without a full-system. Additional proprietary software is required to mimic the interaction of the CPU and GPU via interrupts, to perform arbitrary memory mappings, to set up page tables, and prepare the memory. In a full-system simulation, these steps happen automatically, and using only the tools provided by the vendor as standard. Furthermore, any updates to the software stack are

completely compatible with the simulated system.

GPGPUSim on the other hand requires that Nvidia drivers are installed natively on the host machine, and the simulator then interacts with these. This however, requires the user to sacrifice their own host setup in order to use specific versions of the driver compatible with GPGPUSim. Furthermore, this approach isn't feasible when modeling an embedded system, such as an Arm CPU and and Arm GPU. The system would require Arm-compatible binaries, which couldn't execute on a typical, x86 host system.

Since our simulator doesn't model specific components of the software stack, and instead inherently supports all compatible software through an accurate model of the underlying hardware, any software stack is inherently supported in our simulator. This includes support not only for newer versions of OpenCL, but also OpenGL, Vulkan, and any other new APIs that are being developed. The only requirement for this is that the user installs the new software, just like in a real system.

The speed of simulation, as described in Section 3.3.2 is also a critical component of usability. The simulator that we present in this thesis is fast enough to be interactive, and provides users with feedback at a quick turnaround time.

3.3.5 Comparison Against Existing Hardware and Software

Simulators are key tools used in both pre- and post-silicon environments. Fast simulators provide information in early-design space exploration (for example ISA design), while slower, more detailed simulators can help guide detailed micro-architectural design. Simulators are also used extensively by software developers, who are tasked with developing toolchains and applications for future hardware platforms. Even after fabrication, simulators can be used to gain insight and control over the simulated platform that isn't available with hardware. Here we briefly outline some of the uses cases for our simulator that aren't possible with hardware or existing simulators.

3.3.5.1 Hardware Platforms

Many platforms allow the user to access hardware counters in order to identify hardware utilization, and effects of the executing code on the hardware. These counters typically show numbers of cycles for various components and cache utilization. These counters however, are often at too small of a granularity to completely understand the performance of the CPU or GPU. For example, the counters might report the number of instructions executed as well as the number of threads, but will provide no informa-

tion about how the work is distributed across the threads. Furthermore, in the case of some systems, like the Arm Mali GPU, software support to read hardware counters is not included with the publicly available drivers.

Simulation allows the user to inspect the workload at any desired granularity - for a GPU, this can be at workload, job, threadgroup, warp, or thread level, or even inspecting individual components of the pipeline. These statistics can be accessed at any point in time, i.e. the simulation can be stopped at any point of execution, and the performance counters can be inspected there and then - something that is not possible with hardware counters.

3.3.5.2 Existing GPU Simulators - an Architectural Perspective

Existing GPU simulators model large, Nvidia and AMD GPUs, but provide no support for realistic modelling of embedded GPUs such as the Arm Mali. A dedicated simulation framework is required for modelling such GPUs, as there are key differences in their architectures, supporting systems, and workloads executed on them. Machine learning workloads are more than just growing in popularity - they have taken over as one of the core applications to be accelerated using parallel hardware, such as GPUs. However, very different parts of these machine learning applications are accelerated on desktop GPUs compared to embedded GPUs. Let's consider a personal assistant, such as Amazon's Alexa, or Apple's Siri, which use speech recognition to interact with humans. Speech recognition is a complex task, which is often implemented using Convolutional Neural Networks - specialized versions of Deep Neural Networks. Using neural networks requires two phases - a training phase, where weights are learned by the model by showing it examples, and an inference phase, where the learned model is applied to new, incoming data. Training is expensive, often takes months, and requires vast parallelism to even reach these time frames, and hence is performed on large desktop GPUs, or even clusters of GPUs. Inference on the other hand, is performed close to the user, on an "edge" device, where it benefits from low latency, and increased security during its interaction with the human. Therefore, once training of the model is completed, the model is then transferred to an edge device, which performs inference. However, even in this case only simply neural networks are run on device, and others are transferred to the cloud to yet again run on different devices. As is evident, even though embedded and desktop GPUs share some characteristics, they are employed in very different environments, and need to be suited to very different tasks, and to optimize hardware for neural networks, both desktop and embedded

GPU simulators are needed. As such, we provide the first ever accurate embedded GPU simulator available to the public for design and optimization of machine learning workloads and specialized hardware.

3.4 Performance Modelling Techniques

We have so far only discussed the practicalities and performance of GPU simulators, but accuracy and speed of the performance model is equally important. In this section, we discuss different approaches to GPU performance modelling, while searching for an approach that will not impede high simulation speeds required for executing full software stacks in a full-system simulation environment.

3.4.1 Cycle-Accurate Simulation

There is a wide variety of performance modelling techniques used in existing simulators. Cycle-accurate simulation has already been introduced, as a detailed model advancing one cycle at a time. In this type of simulation, the functional execution and performance prediction are completely coupled, with the performance prediction arising from detailed modelling of every step of execution. Cycle-accurate simulation provides the most observability out of the examined approaches, and has the potential to be completely accurate. However, it suffers from poor performance, is difficult and time consuming to implement correctly, and difficult to maintain, as each iteration of the hardware will require vast updates to the model, not only of architectural components, but also of the micro-architectural model. Poor performance often results in only small, unrealistic workloads and micro-benchmarks being executed. Large scale cycle-accurate modelling is also expensive. The common approach in industry to increase throughput is to run thousands of simulations in parallel, amounting to millions of hours of compute time on servers.

3.4.2 Trace Based Simulation

Trace based simulators operate in in two phases. In the first phase, a trace is collected. The trace can contain anything that reflects the execution over time, but generally contains details for each memory access performed, or for instructions executed. Certain GPU emulators collect information about API calls to the OpenGL runtime. This trace

can then be fed through a timing model in a second phase, which will predict the performance. While the traces can be large and time consuming to generate, they only need to be collected once. They can also be accelerated by emitting binary traces, using faster, functional simulators, or by instrumenting code executing on real hardware. This style of simulation also allows for flexibility in the second phase of the simulation. For example, when exploring the memory system of a new architecture, a functional model can be used to collect the trace of a program, and then multiple memory models can be applied to it in the second phase, to identify the best one.

Trace-based simulation provides a promising alternative to pure execution-driven cycle-accurate simulation, as the trace collection does not necessarily need to hinder fast full-system simulation, but still allows for flexibility in the second phase of the simulation. Furthermore, when combined with other techniques, presented in Chapter 5, we continue to achieve both good performance, and good accuracy relative to existing solutions.

Accel-Sim is a recently released GPU simulator that is built on the long-standing GPGPUSim [84]. Accel-sim splits functional and timing simulation of GPGPUSim into two phases, and implements trace-based simulation. It is most similar to our approach, presented in Chapter 5, however there are notable differences. Accel-Sim supports Nvidia GPUs, while we model embedded, mobile GPUs and provide a tracing plugin for an Arm MALI GPU simulator. Traces for Accel-Sim are generated using a binary instrumentation tool called NVBit [97]. This is a significant improvement over previous GPU simulation approaches, as it means that the full vendor-provided software stack can be used without modification. However, NVBit only supports Nvidia GPUs, and requires a physical Nvidia GPU to work. Our simulator on the other hand uses an existing functional GPU simulator with an interchangeable software stack and functional model, enabling us to explore not only new hardware configurations, but also couple them with new software, and use the trace based simulator for optimizing code using new software stacks. The authors of Accel-sim use microbenchmarks to identify specific architectural and microarchitectural parameters of the GPU they are modelling, however this is not possible in our case, as we later demonstrate. Instead, we use a regression to tune unknown parameters in our simulator. Furthermore, we target a different use case than Accel-Sim - while the detail of Accel-Sim allows for detailed micro-architectural exploration, we trade detail off in exchange for faster simulation turnaround time.

GPUTEjas [76] models a GPU using traces generated with GPUOcelot [74]. GPUO-

celot is a functional simulator that replaces the CUDA runtime library, and functionally executes PTX, however it models GPUs at the PTX level, which has been shown to introduce significant error [98]. Our framework supports a full, native, vendor-provided software stack. MacSim [77] supports Nvidia GPUs via PTX, and Intel GPUs [99]. Traces for Intel GPUs are collected using an instrumentation tool built on GT-Pin [100]. TEAPOT [78] is a trace-based GPU simulator, designed for the evaluation of mobile GPUs and has a cycle accurate GPU model for evaluating performance. TEAPOT supports OPENGLES 1.1/2.0 and runs unmodified Android applications, but relies on the open-source GALLIUM3D drivers for a generic softpipe GPU.

3.4.3 Analytical Modelling

Analytical models take certain features of execution in hardware and define a cost model based on those features. The features can be taken from various sources, for example from hardware counters in order to explain the performance of software on a given hardware platform, or from high level statistics gathered during functional simulation to try to predict performance in real hardware.

Early performance analysis for GPUs focused on specific applications, and relied on constrained programming models and hardware availability. [101] analyzed the performance of dense matrix multiplication on GPUs. [102] presented a novel memory model for analyzing and improving the performance of GPU-based scientific algorithms. [103] breaks down the cost of using the graphics pipeline for general purpose computation.

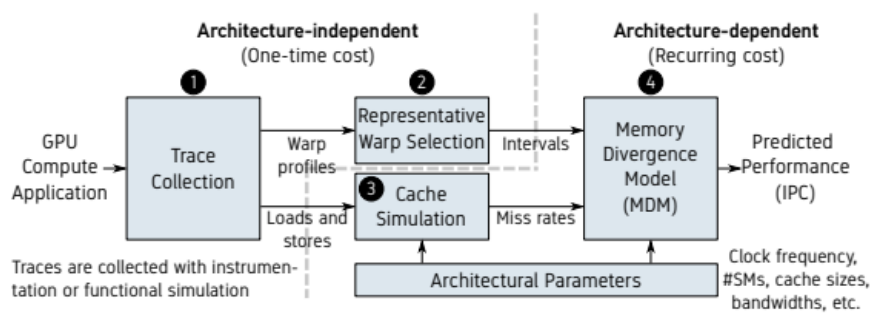


Fig. 3.13: MDM-based performance model. (Fig. Source: [104])

Current state-of-the-art models include MDM [105] (building on GPUMech [106]), which predict performance using a hybrid trace-based and analytical approach through interval modelling. The MDM based performance model is depicted in Figure 3.13.

Traces are collected using either hardware or functional simulation. Next, a representative warp is selected to be used as the basis for data collection for the analytical model. Cache statistics are provided using a multi-core cache simulator which processes the traces. The level of detail of the cache simulator is not provided in the MDM publication. MDM then provides a performance prediction by calculating the steady state of the kernel execution, minus the side effects of network and DRAM contention.

MDM provides a significant 65x speedup over detailed simulation, when simulating just one configuration. However, even more importantly, MDM's approach scales to large design space explorations, with its speedup over detailed simulation rising to 6371x over 1000 configurations. The timings of MDM compared to detailed simulation are presented in Figure 3.14.

Architectural Configurations	Detailed Simulation	MDM	
		One-Time Cost	Recurrent Cost
1	9.8 days	3.6 hours	2 minutes
10	3.3 months	3.6 hours	20 minutes
100	2.7 years	3.6 hours	3.3 hours
1000	27 years	3.6 hours	1.4 days

Fig. 3.14: MDM provides a speedup of 65x over a single iteration of detailed simulation, and 6371x when simulating 1000 configurations, due to its low recurrent cost. (Fig. Source: [104]).

While this analytical approach provides significant speedup over cycle-accurate simulation, there is still a significant one-time cost associated with generating the traces and selecting the representative warp. Crucially, this is only a one-time cost when exploring certain architectural configurations. However, other experiments, for example changing the warp size of the architecture, would result in having to once again select the representative warp.

Another problem encountered by MDM is that selecting a representative warp is an approximation that will work well for regular applications, however may fall short when modelling irregular workloads. GPU kernels can read the thread ID, and often use this mechanism for selecting which threads perform which part of the kernel. If half of the threads executed one branch of the code, and the other half another branch of the code, there would be no single representative warp, which would result in a biased performance prediction.

Taking the above into consideration, we find that a trace-based approach is more

suitable to our requirements, however we leave analytical modelling open for consideration in combination with our approach in future work.

[107] provided the first analytical GPU model available to academia. It is a static analytical model that can be used without executing the program. The authors propose that estimating the cost of memory operations is the key component of understanding GPU performance, as application execution time is dominated by memory access latency. However, as GPUs are massively parallel, this latency can be hidden with parallel memory requests. They define two metrics, MWP (Memory Warp Parallelism) and CWP (Compute Warp Parallelism) as metrics for their analytical model. Memory warp parallelism is a metric for memory bandwidth consumption, while CWP is a metric defining how much computation can be done by other warps while one warp is waiting for values from memory. The analytical model using these metrics provides insights into performance bottlenecks of parallel applications on GPU architectures. This analytical model achieves an average error of 13.3% on GPU computing applications. However, the proposed model only estimates the number of dynamic instructions and memory requests, as the analysis is performed statically. The authors do this by scaling the number of static instructions in a kernel by the number of data elements. While this approach can approximate the instruction count, it will vary widely on workloads that include control flow operations, and especially ones that demonstrate warp divergence. Furthermore, the authors use PTX as an instruction set, and claim that there is a one-to-one mapping between PTX and the native instruction set, but [81] shows that this is not the case.

The same authors present an integrated power and performance model using analytical modelling techniques [108]. They use the results of their previous analytical model [107] as inputs into their analytical model and predict the optimal number of cores to maximize performance per watt. As this model relies on their previous work, it has the same limitations, i.e. it does not model a real instruction set, and does not consider the effects of control flow operations. [109] also builds on the analytical model presented in [107], by using it within a code skeleton framework for accelerating CPU code on GPUs.

Similarly, [110] presents an integrated model to predict performance and energy consumption in order to aid GPU programmers. However, instead of a static approach, the authors use Ocelot [74] to analyze PTX codes to obtain several input parameters, such as the number of memory transactions and data size. The model considers instruction-level and thread-level parallelism, and achieves an accuracy of almost 90%

when compared to real hardware. However, as discussed before, Ocelot provides only an abstraction of GPU model and executes PTX - meaning analytical models such as this one rely on inaccurate execution details.

3.4.3.1 A micro-benchmark-first approach

[111] presents a micro-benchmark-first approach to designing an analytical model. Rather than build a model first, like in previous approaches, they design micro-benchmarks, and design a throughput model for the instruction pipeline, shared memory, and global memory using the results. This approach allows them to limit the number of factors that influence the analytical model, by selecting characteristics that most closely correlate with performance. This work also focuses on identifying program bottlenecks, rather than an overall execution prediction. Furthermore, this work uses dynamic analysis using statistics collected from the Barra simulator, which in turn executes native assembly code, rather than PTX, albeit using a modified CUDA runtime.

[112] presents TEG (Timing Estimation Tool for GPU), a tool used to analyze GPU scaling performance behaviour. It's primary input source is a CUDA binary, disassembled using CUBOJDUMP. For more accurate results, in particular when the kernel contains more complicated control flow, instruction traces generated using the Barra [69] simulator can be used. The analytical model uses execution latency, multi-warp issue latency, same-warp issue latency, and memory latency as parameters to the analytical model. The values for these are obtained using the `CLOCK()` command in CUDA kernels. The work claims an error of approximately 10%, however experimental results are limited to micro-benchmarks. Furthermore, the framework does not consider dynamic scheduling, or a cache model, both of which are found in modern GPUs. This approach provides two improvements over prior work. Firstly, Barra executes native assembly code, rather than PTX, and secondly, the analytical model uses full traces, rather than just end statistics from simulations. However, as discussed previously, the Barra simulator relies on a software stack vastly different to a vendor-provided software stack.

[113] used Pareto-optimal curves to prune the search space for GPU optimizations. However, they did not model memory latency, and assumed that none of the kernels are memory bound, which in the general case is not a convincing assumption for GPUs, as many studies have shown that memory is a bottleneck for GPUs.

3.4.3.2 A compiler-based approach

[114] uses compiler based approach, and builds a program dependence graph PDG, originally presented in [115] as a basis for analytical modelling. Based on the PDG, they can identify control and data dependencies within a single framework. They also provide a framework for symbolic execution to help identify data access patterns and control flow patterns, meaning they can easily estimate control flow divergence, memory bank conflicts, and memory coalescing, which weren't considered by previous approaches. The performance factors are measured in isolation from each other, and later combined in the model. While this approach proposes major improvements over previous work, it still relies on an intermediate program representation, and not an actual program binary.

[116] presents a framework for estimating performance of CUDA kernels in an automated manner. The authors propose the quadrant-split model, which provides insight on the performance limiting factors of multiple devices with respect to a particular kernel, with the key feature being different compute-memory bandwidth ratios. The authors first extract a set of kernel features through automated profiling executions. Secondly, they extract devices features for the target GPU using micro-benchmarks and architecture specifications. They use this information to determine the performance limiting factor and to estimate the kernel execution time.

[117] is a recent work that presents an analytical modelling framework for multi-threaded code on multi-core platforms. In this work, a profiler collects micro-architecture-independent characteristics of a workload's behaviour. The profile contains per-thread characteristics, as well as inter-thread interactions, for example synchronization and shared memory access behaviour. The profile is then used to predict performance on new multi-core architectures.

3.4.4 Machine Learning and Statistical Modelling

The final fast performance modelling technique considered in this thesis is a machine learning based approach. Machine learning and statistical approaches identify specific higher level features and use them to predict performance in hardware. They combine the performance benefits of faster simulators, with the performance prediction capabilities of more detailed simulators. In addition to the performance benefits, machine learning models are easy to use, as they don't require the implementer or user to have any knowledge of the architecture. However, in order to train a good model and make

an accurate prediction, vast amounts of data are required, which are often not available, or require a cycle-accurate simulator or hardware to collect. Furthermore, once a model has been trained, it is difficult to change it, meaning that while it can be of immense benefit for systems development and software engineering, it can be a difficult tool to use in architectural and micro-architectural exploration.

To alleviate some of these concerns, machine learning models often combine expert knowledge in the form of feature selection or a combined machine-learning + analytical model to reduce training speeds and improve performance predictions. Analytical models can also make architectural exploration easier, for example by including architecture-specific variables that can be directly manipulated by the user. We consider machine learning based approaches when developing our performance model, and we present our findings in Chapter 5.

[118] presents a machine learning model using two simulators at different points on the speed vs. detail curve. At the more detailed, slower end, is a cycle-accurate simulator, and at the less detailed, faster end is a functional instruction set simulator. A number of benchmarks are executed using both simulators. A machine learning model is then built to predict the cycle counts of the cycle accurate simulator using features extracted from the functional simulations.

Nagasaka et al. [119] present a machine learning model using performance counters as independent variables in a linear regression. The linear regression is used to model power in the GPU. The approach achieves average accuracy within 4.7%, however it requires performance counters to be present. This has two major drawbacks. Firstly, as the authors themselves point out, performance counters aren't always available for all components of the GPU. For example, this work vastly underestimates power for kernels with texture reads, because of lack of performance counters monitoring texture accesses. Furthermore, this approach cannot predict power offline - i.e. it requires access to hardware for each kernel execution. A further improvement could be to use a simulator to predict the values of the performance counters for each kernel, and then use this model to predict power based on the simulation results.

Chen et al. [120] also use a statistical approach to predict power. Instead of a linear regression, they use tree-based methods, however the largest contribution is using a simulator instead of hardware. GPGPUSim [121] is used to collect statistics, which are unavailable with traditional performance counters, overcoming the obstacles encountered in [119].

Zhang et al. [122] use a random forest based approach, where they build a model

based on an earlier collected performance and power profile. From this model, they are able to extract instructive principles, useful both to the GPU programmer as well as the hardware architect. This is also the first model for an ATI GPU - the previous models were designed for Nvidia GPUs.

Song et al. [123, 124] present a system-level power-performance efficiency model for emergent GPU architectures. Their hardware counter based approach uses a combined analytical approach with an artificial neural network, and unlike previous work, are able to capture non-linear relations between power and performance. Furthermore, Song et al. are the first to consider the entire system that a GPU operates in, and the power and performance model also consider the CPU and Operating System.

Boye et al. [125] present GROPHECY++, an extension to GROPHECY [109], by including a data transfer model. They use a linear model with tunable parameters. The kernels used are the best kernels identified using GROPHECY.

While previous work has focused around CUDA, Karami et al. [126] present a statistical performance model for OpenCL workloads, with the aim of identifying performance bottlenecks and reporting them to the programmer. Results are gathered using CUPTI, a CUDA profiling framework. The analysis uses principal component analysis, from which the authors extract particular principal components, and relate them to architectural bottlenecks exacerbated by the code.

Baldini et al. [127] developed a model for estimating GPU performance before porting a CPU program to GPU. They apply a supervised machine learning algorithm to dynamic data from instrumented program execution on a CPU. Using this data, they predict whether or not it is worthwhile porting code from CPU to GPU. They use very simple, high-level features, similar to our own approach, including the total number of instructions, the ratio of computation over memory, conditional and unconditional branches, and a few others. A key difference, is that our statistics are gathered via a simulator, without the need for changing the program binary and altering the performance behaviour of the program. Furthermore, their model requires an initial OpenMP implementation on the CPU.

Ardalani et al. [128] took a similar approach to Baldini et al. and developed XAPP, a tool for cross-architecture performance prediction. Furthermore, their tool aims to predict the actual performance on the GPU, and not just improvement in terms of order of magnitude. They use more detailed statistics including re-use distance and stride, a more fine-grained breakdown of instruction types, and information about memory throughput.

3.5 Conclusion

This chapter introduced concepts which characterize simulation technology, and on which simulators are built, while simultaneously providing a qualitative evaluation of existing GPU and full-system simulators and performance modelling techniques. The analysis shows that advances in GPU simulation have been focused on detailed modelling of the hardware, while largely ignoring the impact of inaccurate modelling of the software stack. Some steps have been taken in this direction, for example in accurate modelling of the GPU instruction set, however corners are still being cut in modelling the remaining components of the software stack, which is just as critical when developing a faithful software representation of the execution.

State-of-the-art performance modelling techniques are also focused around cycle-accurate simulation, with some examples of analytical and machine learning based modelling also present. However, similarly, none of the existing performance models support modern, mobile applications, and offer poor performance and accuracy trade-offs.

These observations inform our decision to develop a full-system simulation framework used to drive GPU simulation, backed with an offline trace-based performance model. This novel, flexible, holistic approach, solves all of the problems presented in this chapter. Details of the developed framework can be found in Chapter 4, while Chapter 5 presents the performance model.

Full-System Simulation of Mobile CPU/GPU Platforms

Until this point, this thesis has presented background information, related work, and motivated the need for a new approach to GPU simulation. This chapter is the first technical chapter. We start by restating the goals of this thesis, which are to develop a simulation framework, which:

- Accurately simulates a state-of-the-art mobile GPU in a full-system context, enabling the use of *unmodified* vendor-supplied drivers and JIT compilers, operating in an *unmodified* target operating system, and executing *unmodified* applications.
- Supports simulation speeds, which enable the user to execute complete and complex applications typical of modern GPU workloads.
- Provides useful performance statistics, without the overhead of cycle-accurate simulation.

We propose a fundamentally different approach to GPU simulation, avoiding the motivating issues presented in Chapter 3.

We re-iterate the main limitations of existing GPU simulators: (a) instruction sets are not accurately modeled, but approximated by an *artificial, low-level intermediate representation* [74, 94], (b) GPU simulators do not model existing commercial GPUs, but only *simplified GPU architectures* [73], (c) instead of using vendor provided driver stacks and compilers, GPU simulators often rely on *simplified system software*, which

may behave entirely differently to original tools [41, 72], and (d) *GPUs are treated as standalone devices*, not modeling any CPU-GPU transactions [90].

We focus on functional instruction set CPU/GPU simulation, i.e. without detailed timing information. While this method sacrifices cycle-accuracy, it enables us to improve simulation performance to a level where it is feasible to run complex CPU/GPU workloads. Such a functional simulator is also a prerequisite to detailed timing simulation and can still provide useful execution statistics, such as instruction counts, execution and memory traces, and CPU-GPU transaction details. Simultaneously our system guarantees optimal GPU feature support, and ensures that our virtual platform executes identical code to that on physical hardware. Our fast simulation approach also supports interactive workloads, and new Application Programming Interfaces (APIs) (e.g. Vulkan) without additional engineering.

Notable use cases for our full-system CPU/GPU simulation technology are (1) early GPU design space exploration, where a GPU currently under design can be evaluated and (2) virtual platforms for both system and user level software development, both without producing a physical version. These use cases benefit particularly from the accuracy and performance that our integrated Central Processing Unit (CPU)/GPU simulation approach offers.

4.0.1 State-of-the-Art

In order to further motivate our full-system approach to CPU/GPU simulation, we initially review three popular GPU simulators: GPGPU-Sim [70], Multi2Sim [41, 72], and gem5 with the GCN3 model and the Radeon Open Compute Software stack [81]. In Figure 4.1 we compare the GPU kernel execution and software stacks for a native execution environment, our full-system simulation, Multi2Sim, GPGPU-Sim, and gem5.

In native hardware (Figure 4.1(a)), an OpenCL CPU executable is run in Linux on an Arm CPU. This executable includes an embedded OpenCL kernel, and loads a vendor provided runtime library, e.g. `libOpenCL.so`, to JIT compile the OpenCL kernel to GPU instructions. This runtime interacts with a GPU device driver—a vendor-specific kernel module for low-level CPU-GPU interaction—which manages the setup of GPU jobs. Finally, the GPU executes binary instructions from memory.

Our full-system simulation model (Figure 4.1(b)) implements both the CPU and GPU completely and accurately. We run the original unmodified executable and the

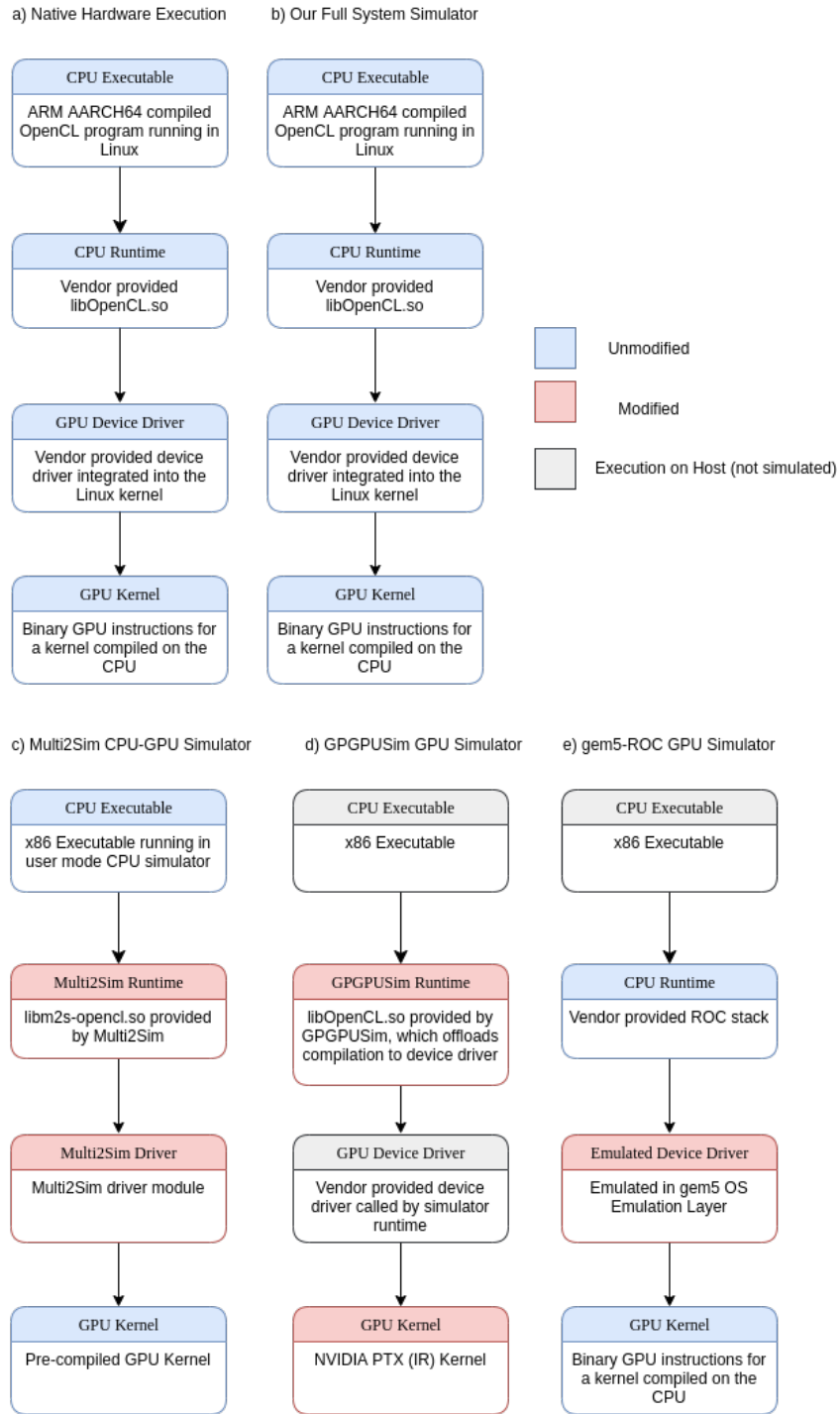


Fig. 4.1: Comparison of the GPU kernel execution model and software stack for (a) a native execution environment, (b) our full-system simulator, (c) Multi2Sim, (d) GPGPU-Sim, and (e) gem5 with the Radeon Open Compute software stack. For MultiSim, GPGPU-Sim, and gem5 we have highlighted non-standard software components that are different from the vendor-supplied driver stack and thus represent a source of inaccuracy.

original CPU runtime environment for the GPU and GPU driver. Our GPU simulation component completely simulates Arm’s Bifrost architecture, executing the same binary as the physical GPU implementation in Figure 4.1(a). Our GPU simulator interacts with the simulated CPU and driver executing on the CPU in the same way as its physical counterpart, making the simulation identical to a physical GPU for the entire software stack.

Compare this to Multi2Sim in Figure 4.1(c). Multi2Sim’s OpenCL stack differs substantially from the native stack. OpenCL function invocations are handled by a non-standard runtime, which is intercepted by the CPU simulator, and redirected to the GPU simulator to launch the kernel execution.

Tools like Multi2Sim require heavy maintenance as toolchains advance. We have seen that code compiled by newer versions of AMD’s OpenCL compiler, which the Multi2Sim toolchain Multi2C relies on, often contains features unsupported in Multi2Sim. The user then must rely on an outdated (and now unavailable) version of the OpenCL compiler.

Even with compatible OpenCL tools, it would still be impossible to execute kernels which rely on host runtime information for compilation. For example, a program might query how much memory is available on the platform, before deciding the mapping of data for the executing application.

GPGPU-Sim (Figure 4.1(d)) provides a model for Parallel Thread Execution (PTX) or SASS execution, where PTX is a scalar low-level, data-parallel virtual Instruction Set Architecture (ISA) defined by Nvidia, and SASS is the native shader assembly for Nvidia GPUs. While PTX is an intermediate representation, SASS is closer to the actual GPU instruction set. However, GPGPU-Sim requires its own runtime libraries and device drivers, which (a) differ substantially from the vendor supplied libraries, (b) are not feature complete, and (c) introduce significant accuracy problems.

4.0.2 Contributions

In this chapter we present a full-system simulation environment for a mobile platform, enabling users to run a complete and unmodified software stack for a state-of-the-art mobile Arm CPU and Mali-G71 GPU powered device (Section 4.1). In Section 4.2 we describe our instrumentation efforts, which allow us to collect runtime statistics. We validate our simulator against a hardware implementation as well as Arm’s stand-alone GPU simulator, achieving 100% architectural accuracy across all available toolchains

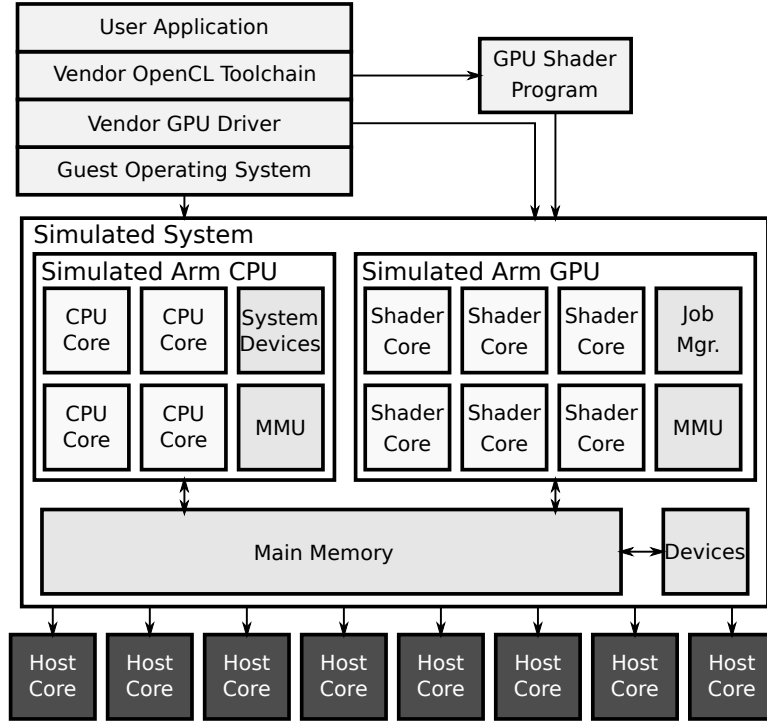


Fig. 4.2: Guest applications execute on simulated CPU-GPU platform, running native Arm Linux with unmodified GPU device drivers. Guest cores map onto host threads.

(Section 4.3). We present a qualitative evaluation in Section 4.4, and demonstrate the flexibility of our instrumentation in Section 4.5. In Section 4.6, we demonstrate the capability of our GPU simulation framework in running full-scale modern GPU applications by optimizing an advanced Computer Vision application using simulated statistics unavailable with other simulation approaches or physical GPU implementations. We then make a direct comparison against desktop GPUs, and show that memory usage is hugely significant to mobile GPU performance. Finally, we use the simulator to examine and explain unexpected performance degradation of a Convolutional Neural Network workload in hardware. In Section 4.7 we describe the implementation details of the simulator, and conclude in Section 4.8.

4.1 Our Simulation Approach

Our simulation environment, as shown in Figure 4.2, provides a full-system view of a CPU/GPU platform. Such an approach also requires additional components to be emulated including a Memory Management Unit (MMU), interrupt controller, timer devices, storage and network devices. In order to benefit from existing device drivers,

we model the Arm VERSATILE EXPRESS and JUNO platforms, each augmented with an Arm Mali-G71 GPU.

Both the simulated CPU and GPU are modeled using high-level architecture descriptions [129], and generated using a retargetable simulation framework [130], which also supports other architectures. They each run in separate threads on the host CPU, providing concurrent and asynchronous operation. Synchronization between the CPU and the GPU is provided through implementation of GPU and CPU communication constructs, for example interrupts and atomic operations.

4.1.1 CPU Simulation

We simulate the CPU through full-system Dynamic Binary Translation (DBT), using Captive (described in Section 2.5), which boots a Linux kernel and user space compiled for ARMV8, from a file system mounted by the simulated storage device. For complete and accurate modeling, we simulate essential platform devices, ensuring that our simulator can support a full software stack without simulation-specific adaptation of any software component.

4.1.2 GPU Simulation

We generate an interpretive GPU simulation module for the programmable GPU Shader Core (SC)s from the Mali architecture description using GenSim (background in Section 2.4 and implementation details in Section 4.7). This generated module implements the core of the execution engine in the GPU. All other components are implemented in C++.

4.1.2.1 CPU-GPU Interface

The GPU interfaces with the CPU via memory mapped registers, hardware interrupts, and memory, through which the simulated GPU exposes its Job Manager (JM) to the CPU. For GPU compute jobs, the OPENCL driver sets up shader programs in the shared CPU-GPU memory space, and then signals the GPU by writing to a control register, indicating that a job is ready for execution. These control registers are read by the JM, which begins execution.

4.1.2.2 Shader Core Simulation

The generated simulator code comprises the instruction decoder and main Execution Engine (EE)s of the GPU. The interpretive execution model is split into two phases: (1) decode, and (2) execution. During phase one, the shader program and its associated metadata are decoded for later use. In phase two, a dispatcher iterates over the job dimensions and creates simulated *GPU* threads. These threads are grouped into “warps”, where all threads execute in lockstep. Warps are in turn grouped into threadgroups, i.e. OPENCL workgroups.

4.1.2.3 Performance Optimizations

The simulation is broken up into two stages - decode, and execution. During the decode stage, the GPU extensively caches guest code, which is then accessed during the execution phase. This model ensures that the entire shader program is decoded exactly once.

In hardware, each SC executes one threadgroup at a time. In our simulator, however, the number of SCs and host threads is individually configurable. For example, instead of mapping 8 SCs onto 8 host threads, we can map the executing threadgroups onto 32 host threads, creating *virtual cores*.

This necessitates additional measures for managing local storage. The GPU driver allocates local storage for 8 threadgroups corresponding to the 8 detected SCs. To support more threadgroups executing in parallel, the simulator allocates additional local memory for each host thread, outwith the guest system. The original program isn’t aware of this memory, since it is only simulation memory, and it lives outside of the GPU’s address space, however for the purposes of functional simulation, this is irrelevant, as local memory can only be accessed by the currently executing workgroup. Local guest memory accesses are intercepted and mapped to host memory, guaranteeing functional correctness.

4.1.2.4 Job Manager Simulation

In our GPU simulator the JM operates in its own host simulation thread. It fully implements the functionality of its hardware counterpart such as parsing job descriptors and orchestrating the operation of the SCs.

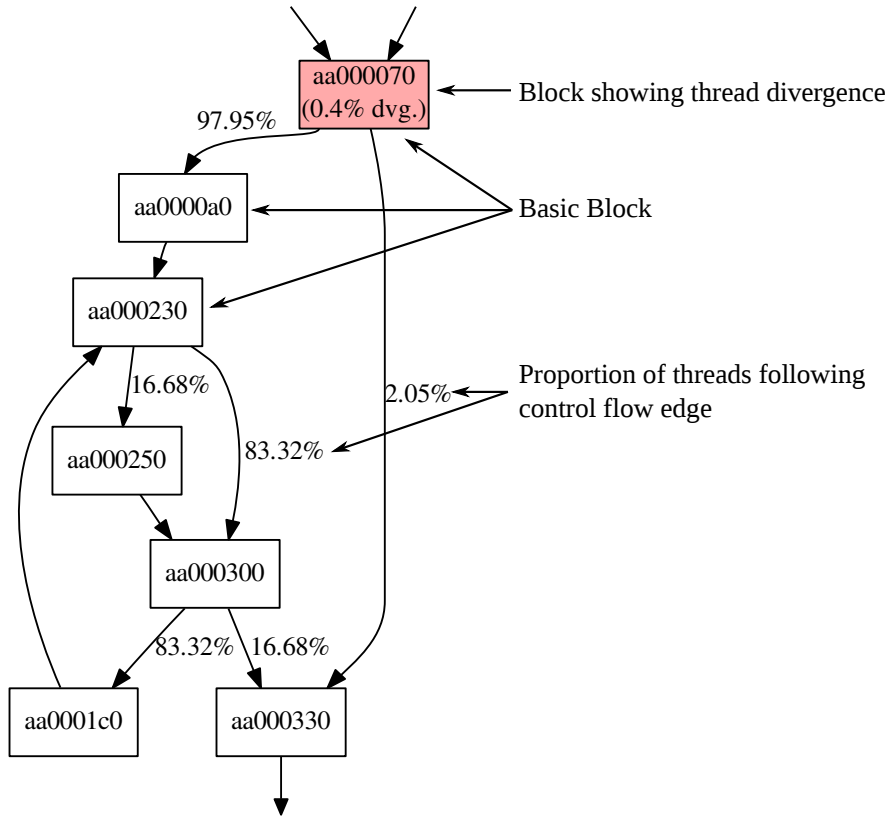


Fig. 4.3: *BFS*: Our simulator generates a control flow graph pinpointing the divergence on actual GPU instructions.

4.1.2.5 Memory Management Unit Simulation

Our simulator incorporates a complete software implementation of the GPU's MMU. The driver provides the MMU with page table pointers, and the MMU reports errors (permissions violations, faults) to the driver through memory mapped registers and interrupts.

4.2 Instrumentation

We previously discussed the high performance overheads of cycle-accurate simulation. Instead of following this method, our simulator is implemented as a functional instruction set simulator. However, this approach still allows us to gather useful statistics, which are described in this section, while maintaining a good simulation rate, enabling us to execute full, modern, mobile applications.

4.2.1 Program Execution

We gather *instruction counts and breakdowns, data accesses, and clause information* - statistics directly relating to the executing instructions. From these we see instruction type ratios, such as ratios of memory instructions to arithmetic, and types of memory accesses - both vital to understanding performance implications of the executed code. Each clause is instrumented with detailed metrics at decode time, and during execution, we record clause frequency. If executing with multiple host threads, this is gathered by each parallel unit. Metrics can then be viewed either per-core, or can be totalled at job completion, requiring no further synchronization.

4.2.2 System

The GPU operates as an accelerator, therefore it is vital to understand its interaction with the rest of the system. The number of pages accessed by the GPU shows the interaction with the memory system and MMU, which are expensive in terms of performance. Interrupts and system register accesses describe the communication with the CPU –also a bottleneck.

4.2.3 Control Flow

Control flow execution in the GPU monitors thread divergence, which occurs when threads within a warp take different paths after a conditional branch. This is a serious performance problem, as if a thread diverges, other threads in the warp must stall until the diverging thread reconverges. We monitor this by tracking the PC on clause boundaries, and building a Control Flow Graph (CFG). This CFG shows which thread executes which path, and identifies diverging threads at their divergence point, as shown in Figure 4.3.

4.3 Validation and Quantitative Evaluation

First, we present the validation strategy for our simulator against Arm hardware and a proprietary simulator, achieving 100% architectural accuracy across all available toolchains. We define architectural accuracy as the accuracy when comparing our simulation statistics to architecturally exposed state (e.g. instruction counts, instruction breakdowns, divergence, register accesses, memory accesses) of GPU hardware and

Simulated Platform	Arm-v7A/v8A CPU
	Arm Mali Bifrost GPU - G71, 8 Cores
	Arch Linux (Kernel 4.8.8)
	Arm Mali Bifrost DDK r3p0/r9p0
Multi2Sim Eval. Platform	x86 CPU, Southern Islands GPU
Evaluation Platform	HiKEY960 - Arm-v8A CPU
	Arm Mali Bifrost GPU - G71, 8 Cores
	Android-O/Debian Linux
	Arm Mali Bifrost DDK r3p0/r9p0
Host Platform 1 (main experiments)	Intel(R) Core(TM) CPU i7-4710MQ 4 cores with HT, 2.50GHz
Host Platform 2 (Parallel scaling, Figure 4.7)	Intel(R) Xeon(R) CPU L7555 32 cores with HT, 1.87GHz

Table 4.1: System configurations for performance evaluation.

proprietary simulator. We then compare our simulator’s performance and effectiveness against Multi2Sim 5.0, whose approach is most similar to our own. Unless explicitly stated, all comparisons against Multi2Sim use Multi2Sim’s functional simulation mode. Finally, we demonstrate the versatility of our simulator through a series of use cases. Our evaluation focuses on the widely accepted OPENCL compute API, which allows for direct comparison with other GPU simulators.

Details of our host and guest platforms are provided in Table 4.1. As different benchmarks scale in different ways, the default host configuration uses 8 host threads for GPU simulation. We show additional results for selected benchmarks.

We chose kernels from a variety of benchmark suites. First, we include AMD APP SDK 2.5 as pre-compiled GPU binaries packaged with Multi2Sim enable direct comparison. AMD driver 2.5, which Multi2Sim, and its compiler Multi2C, rely on, is no longer available, and code compiled using newer versions often contains features unsupported by Multi2Sim. We also report results for Parboil [38] and Rodinia [43] benchmark suites, which provide larger, more complex workloads. The benchmarks and inputs are presented in Table 4.2.

4.3.1 Validation and Accuracy

Functional and architectural correctness of our full-system simulation approach has been established by comparison against the commercially available HiKEY 960 with

Suite	Benchmark	Input Type & Size
Rodinia 3.1	Back Propagation	65536 nodes
Parboil	Breadth First Search	1257001 nodes
AMD APP 2.5	Binary Search	16777216 elements
AMD APP 2.5	Binomial Option	512 samples
AMD APP 2.5	Bitonic Sort	2048 elements
Parboil	Cutoff-limited Coulombic Potential (cutcp)	67 atoms
AMD APP 2.5	DCT	10000x1000 matrix
AMD APP 2.5	DwtHaar1D	8388608 signal
AMD APP 2.5	Floyd Warshall	256 nodes
AMD APP 2.5	Matrix Transpose	3008x3008 matrix
Rodinia 3.1	Nearest Neighbor	5 records 30 latitude 90 longitude
AMD APP 2.5	Recursive Gaussian	1536x1536 image
AMD APP 2.5	Reduction	9999360 elements
AMD APP 2.5	Scan Large Arrays	1048576 elements
Parboil	SGEMM	128x96, 96x160 matrices
AMD APP 2.5	SobelFilter	1536x1536 image
Parboil	Sparse Matrix Vector Mult.	1138x1138x2596 matrix 2596 elements
Parboil	Stencil	128x128x32 matrix 100 iterations
AMD APP 2.5	URNG	1536x1536 image
clBLAS	SGEMM	1024x1024 matrix

Table 4.2: Benchmarks and data set sizes.

a Mali-G71 MP8 GPU. We also validated the GPU part of our simulator against a detailed proprietary simulator for the target GPU architecture. Our comparisons have shown complete accuracy for the evaluated benchmarks, for all evaluated metrics. This is possible only because our simulation is driven by the exact binary that is executed in hardware, thanks to the full support of a native software stack.

4.3.1.1 Comparison to Hardware

Validation has focused on: (a) Correctness of OPENCL kernel execution on the GPU, evaluated through extensive testing, (b) correctness of performance metrics, including instruction counts, instruction breakdowns, clause sizes, data access breakdowns, and divergence, for which we compare results from our instrumented simulator to hardware performance counters on the HiKEY 960.

4.3.1.2 Comparison to Reference Simulator

We have also validated our simulator against a proprietary, detailed standalone GPU simulator. We executed selected kernels on both simulators using an instruction tracing mode, where individual instructions and their effects are observable. Additionally, we employed fuzzing techniques for rigorous instruction testing, covering an extensive range of inputs.

4.3.2 Simulation Performance

Simulation performance is critical when executing interactive applications and running large scale design space explorations. In this section, we evaluate three key full-system simulation performance characteristics. First, we measure the GPU simulation speed, followed by CPU simulation speed, and finally we look at optimizations to simulation. The experimental platform is described in Table 4.1.

4.3.2.1 GPU OpenCL Simulation Speed

Figure 4.5 presents execution performance of our GPU simulator relative to Multi2Sim, where most benchmarks exhibit similar performance levels. Exceptions are *Binary-Search* and *SobelFilter*, where our simulator is up to 10x slower than Multi2Sim, and *sgemm*, where our simulator is 8.8x faster. While this disparity is due to implementation differences between the simulators and simulated architectures, the results demonstrate that accurate full-system simulation of a GPU platform is feasible and yields

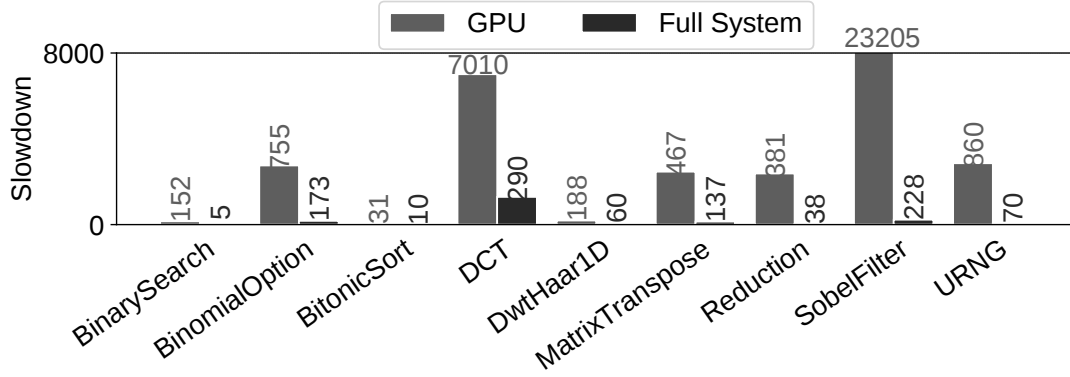


Fig. 4.4: Simulation slowdown relative to the HiKEY960 for GPU only, and for the entire benchmark (CPU+GPU).

competitive performance. Figure 4.4 shows simulation slowdown over native execution. The average slowdown is 4561x. This is a much smaller slowdown than 8700x functional slowdown and 44000x architectural slowdown exhibited by Multi2Sim, however we should also consider that the hardware modelled by Multi2Sim is a desktop GPU, far more powerful than the mobile GPU we model.

Full instrumentation of the GPU simulation generally adds $<5\%$ overhead, due to the approach described in Section 4.2. This means that we provide useful statistics, with performance similar to Multi2Sim’s, which by default only reports instruction breakdown and job dimensions. In cycle-accurate mode, Multi2Sim reports additional statistics, including active execution units, compute unit occupancy, and stream core utilization, however, in our tests it failed to complete the majority of workloads, due to large inputs. On smaller workloads, we observe slowdowns of up to 10x over functional simulation.

4.3.2.2 CPU OpenCL Driver Simulation Speed

Full-system GPU simulation, executing the full software stack on the CPU, adds substantial stress to CPU simulation. Figure 4.6 shows software stack runtimes for *SobelFilter* with different input sizes. While Multi2Sim spends $>150s$ on CPU-side execution for the largest tested input, our JIT-based CPU simulator executes the entire stack in $<10s$, resulting in better performance, while maintaining faithful execution. Overall, Figure 4.4 shows that slowdown for the entire system over native hardware is low, averaging only 223x slowdown.

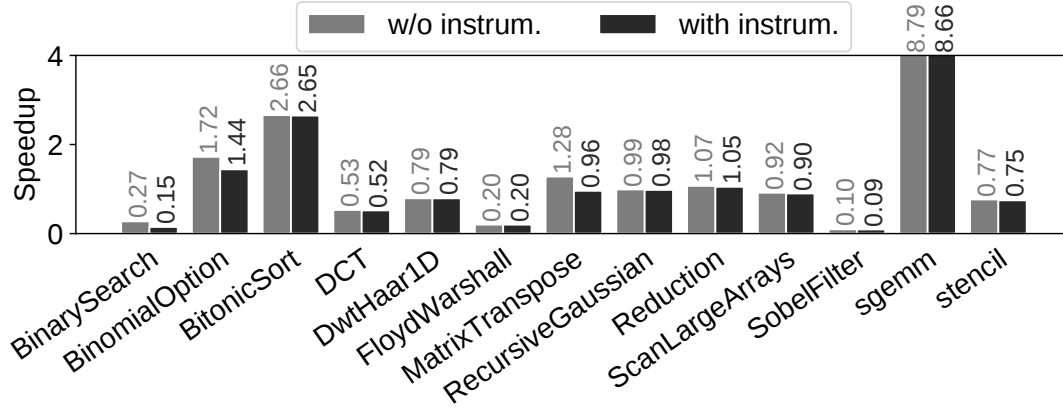


Fig. 4.5: Our simulator's speed with and without instrumentation, relative to Multi2Sim functional simulation (=1.0). This graph shows only the GPU component of the simulation.

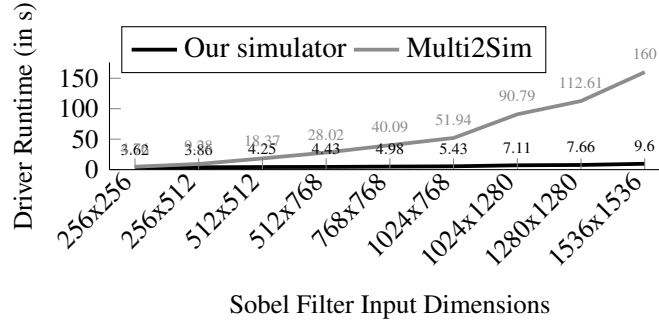


Fig. 4.6: The software stack executing on our DBT CPU simulator scales exceptionally well relative to Multi2Sim.

4.3.2.3 Simulation Performance Optimizations

In Figure 4.7 we evaluate the performance optimization introduced in Section 4.1.2.3, mapping GPU SCs onto multiple host threads. In the worst case, *BinarySearch* is iterative, with short kernels executing with heavy CPU interaction, limiting improvement. For *SobelFilter*, the best case, large thread-group sizes executed for a single kernel enable efficient parallel execution, resulting in steady speedup as host threads are added.

4.4 Qualitative Evaluation

This section examines our full-system GPU simulator through the lens of the motivation presented in Section 3.3, by looking at accuracy of the software stack, re-iterating key results on the speed of the simulation, discussing the ease of maintenance using

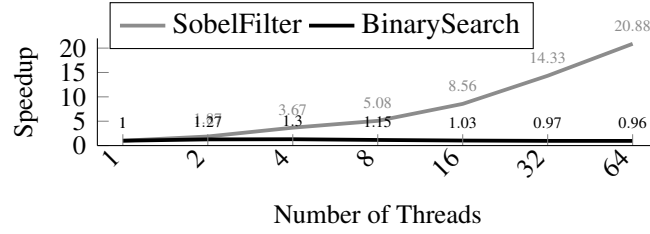


Fig. 4.7: Increasing the number of host simulation threads yields vast performance improvements for certain benchmarks.

our technique, and discussing usability and flexibility aspects of full-system simulation compared to existing approaches.

4.4.1 Accuracy of the Software Stack

Using our full-system approach, the executed software stack is completely *unmodified*, meaning the exact same code is executed in simulation and in hardware on both the CPU and GPU.

4.4.1.1 Device Driver

As the executed software stack is completely unmodified, the device driver executes in the same way as it would in real hardware, meaning the memory model, multi-kernel workloads, and fault handling are available out of the box.

Memory Model - The physical memory between the CPU and GPU is shared. In our implementation, all we need to do to model this is point the base pointer for the simulated CPU memory and simulated GPU memory to the same physical address in our model. All address resolution from virtual to physical addresses is handled automatically by the modelled MMU, using the page tables provided by the device driver. The only synchronization required is that which is defined by the software model - i.e. barrier operations for the GPU only, and atomic operations, which must lock the memory for both the CPU and the GPU. Barriers are handled implicitly by the BARRIER instruction of the Mali instruction set. Barriers are implemented at the workgroup level, and are implemented by pausing each GPU thread on on the instruction until all threads in the workgroup reach the same point. Atomic operations are also part of the native instruction set, and are inserted by the user at the OpenCL level. In our model, they are handled using `STD::LOCK_GUARD`. In the general case, the structure of the host program should prevent the CPU and GPU from writing to the same memory si-

multaneously, and as such, we do not support locking the same memory address from both the CPU and the GPU. We however acknowledge, that this may occur in some applications, in particular if an OpenCL program is executing kernels on both the CPU and the GPU. If this were the case, in order to capture correct behaviour, the code executing on the CPU would also require an explicit atomic operation. A fast and correct approach to handling such operations in fast simulation is described in [131].

Multi-kernel workloads - Modern day workloads, for example SLAM algorithms or Deep Neural Networks execute multiple kernels in sequence on the GPU, all dispatched via the OpenCL driver on the CPU, with buffers being shared and with intermediate computation on the CPU. Simulation of these types of workloads is only possible with a full, unmodified software stack, and a fast CPU simulator, which is exactly what is provided by our framework. The scalability of our CPU simulation compared against Multi2Sim is presented in section 4.3.2.2.

At the time of development, we did not observe multi-kernel workloads, which execute multiple kernels simultaneously. This feature however is expected to appear in future updates to the runtime libraries.

Fault Handling - Occasionally, faults can occur during GPU execution. For example, an attempted access to an invalid address would result in a translation fault from the GPU's MMU. This is communicated via an interrupt raised from the GPU back to the GPU driver executing on the CPU, where the fault is handled. In our simulation framework, this executes exactly as it would in real hardware using an unmodified driver stack.

4.4.1.2 OpenCL Runtime

Our simulation framework supports the unmodified OpenCL runtime library, which can be updated at any time, and behaves identically to the runtime library executing in real hardware. This means that we can observe subtleties that are not visible when the driver is emulated or replaced, as first presented in Section 3.3.1.3, and as will be presented in more detail in Section 4.6.3.5.

4.4.1.3 OPENCL Compiler

Cutting edge research requires the latest tools and software stacks, and the OpenCL compiler is being continuously updated with the runtime. Our simulation framework supports any available version of the OpenCL compiler, allowing compiler versions

between different platforms to be easily matched. Furthermore, our full-system framework can support the compiler and runtime for any other framework, without the need to maintain a specific simulator-friendly version separately from the main compiler.

4.4.1.4 Host Program

The host program driving the GPU execution can be executed in our framework without any modification - from a simple OPENCL program, to complex PyTorch applications with multiple layers of the software stack. We support any binary that is compiled for the guest CPU architecture.

4.4.2 Speed of Simulation

Our CPU simulation speed can outperform native execution [37], which makes executing the full software stack feasible. Our GPU simulator is on-par with existing functional GPU simulators, but is capable of providing performance information directly from the functional simulation, and accurate performance predictions using bolt-on performance models, which are described in Chapter 5. Details on simulation performance are presented in Section 4.3.2.

4.4.3 Ease of maintenance

Core instruction sets rarely change, and when they do, it is generally in the form of instruction set extensions. Even so, the simulators for both the CPU and GPU instruction sets used in our framework are generated from high-level, easy to implement, architecture descriptions implemented using GenSim, whose details are presented in Section 2.4. There is no additional maintenance required for any of the software stack. Any software updates that can be performed in hardware can also be performed in our full-system simulator.

4.4.4 Usability & Flexibility

When using the simulator in a terminal environment, the user should notice no difference between our simulated platform and a real hardware platform - aside from the speed of the GPU. Programs are invoked exactly as they would be using a physical platform.

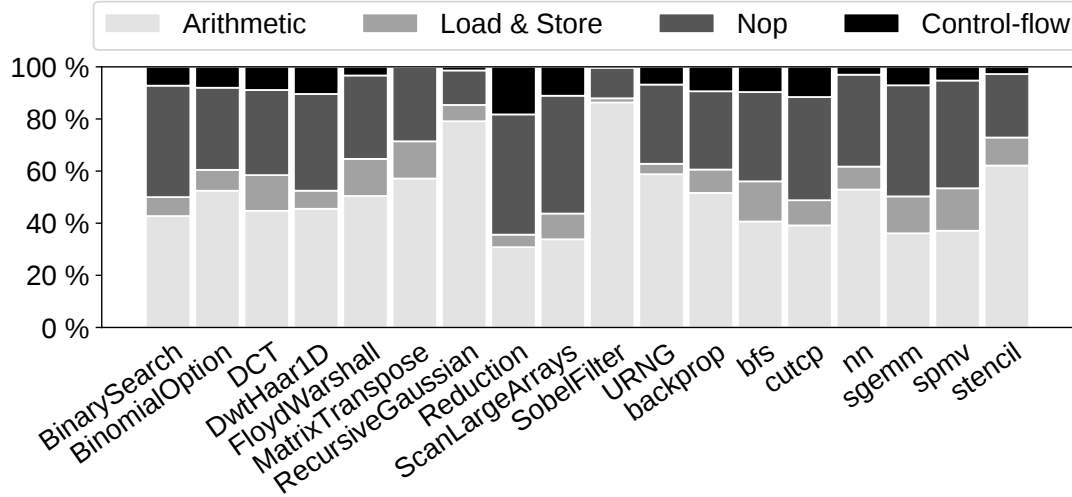


Fig. 4.8: A breakdown of instruction mixes and empty slots can help us identify bottlenecks in GPU code.

We implement an ARMV8 CPU and MALI BIFROST GPU, however the simulation methodology extends to any architecture. Furthermore, the use of Captive as the CPU simulator (described in Section 2.5) enables us to execute any guest architecture on any host architecture, while GPGPUSim is limited to executing native x86 code. Such support is particularly critical in modern simulation environments, as the lines between architectures that were historically developed for mobile devices, and others for desktops and servers, are becoming blurred.

4.5 Application Results

We demonstrate the versatility of our simulator through a series of use cases. We first focus on architectural features of Bifrost which would be useful in early design space exploration. Next, we show the capability of our simulation framework by optimizing an advanced Computer Vision application using simulation statistics. We then demonstrate how performance optimizations for desktop GPUs inadvertently trigger bottlenecks on embedded GPUs, and show the significance of efficient memory usage in mobile GPUs. Finally, we demonstrate the value of the simulator when exploring channel pruning for convolutional neural networks on embedded GPUs.

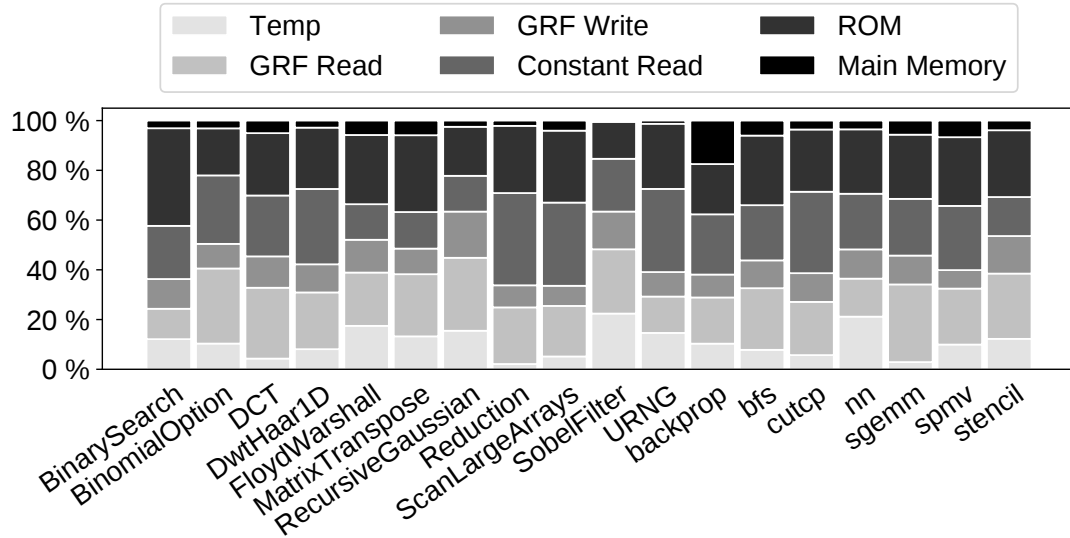


Fig. 4.9: Data access breakdowns show a complete view of each architecturally visible level of memory hierarchy.

4.5.1 Identifying Empty Instruction Slots on the GPU

Figure 4.8 shows instruction mixes for OpenCL benchmarks. For example, *SobelFilter* is a compute-intensive filter with very few empty slots and memory accesses and almost no control flow. In contrast, the number of empty slots in *Reduction* and *ScanLargeArrays* indicates low GPU utilization. On average, 50% of instructions are arithmetic operations, while local memory and control flow each contribute around 10%. Performance can be substantially improved by reducing the number of empty instruction slots introduced by the OpenCL toolchain.

4.5.2 Moving Data Closer to the Core

Different types of data storage have various access latencies, which when poorly utilized can lead to colossal drops in performance. Ideally, data should be kept as close to the GPU’s execution cores as possible. Our simulator shows exact data placement throughout the hierarchy, and can be used to guide optimization.

Data breakdowns are shown in Figure 4.9. *SobelFilter* exhibits few main memory accesses, while the figures for *backprop* suggest that it could benefit from enhancements to the OPENCL compiler, more registers, or a better algorithm. Fast accesses to temporary values, constants and ROM dominate. More reads from than writes to global registers suggest effective reuse of register data. Global memory accesses account for

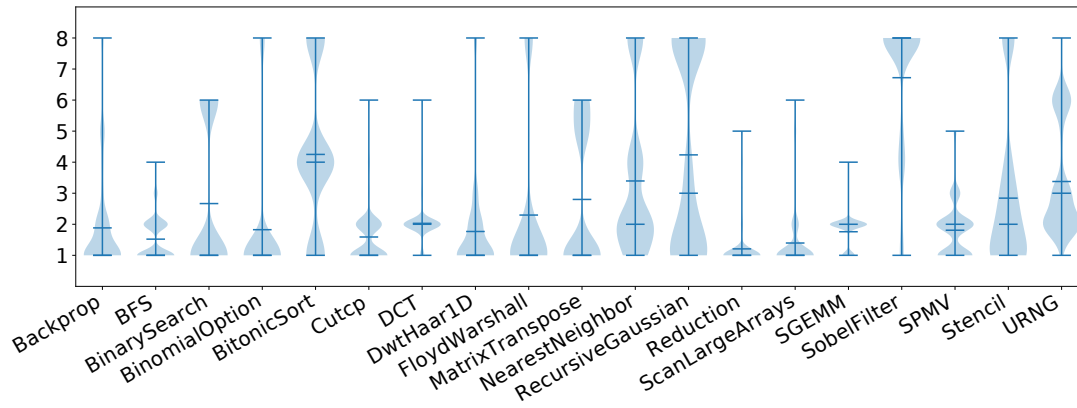


Fig. 4.10: A clause size distribution presents optimization targets and bottlenecks for the Bifrost clause model.

<10% of accesses, except for a single case, *backprop*.

4.5.3 Evaluating the Bifrost Clause Model

Clauses contain up to 8 instruction words (16 instructions), which execute unconditionally. Longer clauses are preferable - they reduce global register file accesses through temporary register use and limit the scope for control flow and thread divergence.

Figure 4.10 shows the distribution of clause sizes for all benchmarks. Several, including *BinomialOption* and *FloydWarshall* exhibit a majority of clauses of size 1 or 2, and occasionally size 8. Others peak at mid-size clauses, e.g. *BitonicSort*, or are bimodally distributed, e.g. *RecursiveGaussian*. Compare this to the instruction mix in Figure 4.8, where e.g. *RecursiveGaussian* features a larger fraction of arithmetic instructions and few empty slots, whereas *Reduction* is reversed. Overall, kernels with larger clauses feature fewer empty slots, while short clauses and empty slots show some correlation.

Potentially, some kernels perform little work between control flow operations, or the compiler is unable make use of available slots. Benchmarks with shorter clauses also display a large proportion of memory accesses, suggesting that memory bottlenecks limit the potential of the clause model. The model might suit graphics workloads, as they benefit from additional data processing units and exhibit regular behaviour, however re-visiting the model for compute might be worthwhile.

Benchmark	Page	Ctrl. Reg	Ctrl. Reg	Interr.	Comp.
	Acc.	Reads	Writes	Asserted	Jobs
BFS	51723	308098	66209	8022	1003
Binomial Option	31	136	70	4	1
SobelFilter	4609	136	70	4	1
Stencil	99603	14795	1982	105	100

Table 4.3: System statistics detail the CPU-GPU interaction.

4.5.4 System Level Results

CPU-GPU communication can account for as much as 76% of execution time [43]. In our full-system environment, we are able to gather system-level statistics unavailable to other GPU simulators or hardware. Our approach provides the capability to observe CPU-GPU interactions, allowing us to monitor memory usage, interrupts, and control register accesses, presented in Table 4.3 for selected benchmarks. While CPU-GPU communication is greatly reduced in a shared memory system, our full-system approach can also be applied to systems where the CPU and GPU aren’t so tightly coupled.

While *SobelFilter* exhibits little CPU-GPU interaction, *BFS* touches more pages, and involves a higher number of transactions.

Page use differs by up to three orders of magnitude across benchmarks, with *stencil* and *BFS* dominating this metric. *BFS* is particularly heavy on control interactions showing an unusually high number of control register accesses and interrupts resulting from over 1000 individual compute jobs.

This information provides useful system-level profiles of applications. Real-world examples of how this data is used is presented in Figure 4.6.2.

4.6 Optimizing OpenCL Applications

4.6.1 SLAMBench

We demonstrate the capabilities of our full-system simulator by evaluating the OpenCL SLAMBENCH [44] computer vision application, which comprises several compute kernels and dataflow orchestrated by the CPU. In its full configuration, SLAMBENCH executes 40000 kernels, impossible to simulate with existing GPU simulators out-of-the-box, due to their limitation to single kernels, tool chain incompatibilities or lack of

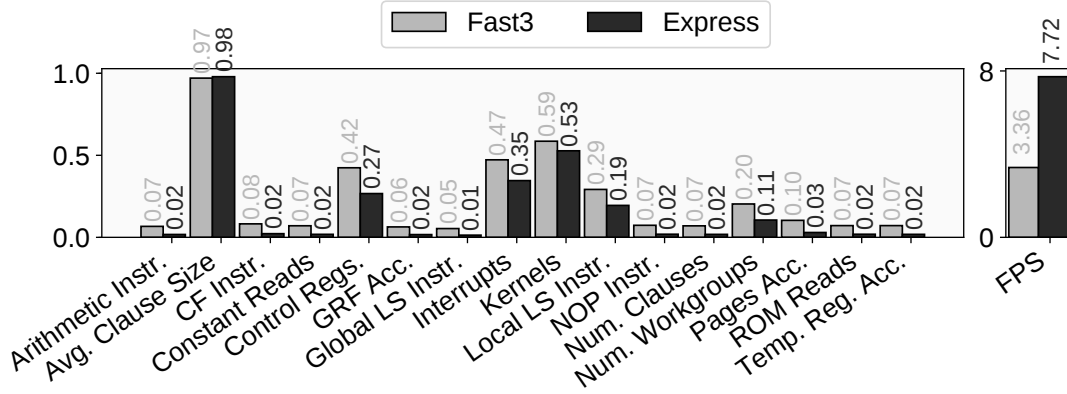


Fig. 4.11: Simulated SLAMBENCH statistics directly relate to HW performance, aiding the search for optimal configurations.

support for CPU-GPU interactions.

Countless configuration options are available in the benchmark, each with varying performance. We execute the KFusion benchmark, with *standard*, *fast3*, and *express* configurations. Figure 4.11 shows metrics for *fast3* and *express* relative to *standard*. Both show major improvement. The relative instruction count for each category is at most 8% for *Fast3* and just 2% for *Express*, while the ratio for local memory instructions is much higher - 29% for *Fast3* and 19% for *Express*, meaning increased local memory use relative to total instruction count. In the case of Mali GPUs, using local memory is discouraged, as local memory is simply mapped to global memory. Again, our full-system modelling technique can apply to any type of system, and the results may have different implications for GPUs with discrete memory.

Our metrics can easily guide us to a good solution, without requiring hardware. While we cannot predict the exact frame rate, the simulated metrics suggest successive improvement between *standard*, *Fast3*, and *Express*. This is truly the case - *Fast3* is 3.35 times faster than *standard* and *Express* is 7.72 times faster than *standard*.

4.6.2 SGEMM

[132] shows that optimizations applied to the same code targeting different architectures result in greatly different performance relative to hand-tuned code. This is exacerbated in mobile GPUs, whose architectures are completely different to desktop GPUs [95]. We evaluate this claim through six SGEMM kernels ([48], [133]), a core component of linear algebra and machine learning applications, which are increasingly moving to mobile devices. Starting with (1), the kernels in Figure 4.12 are iteratively

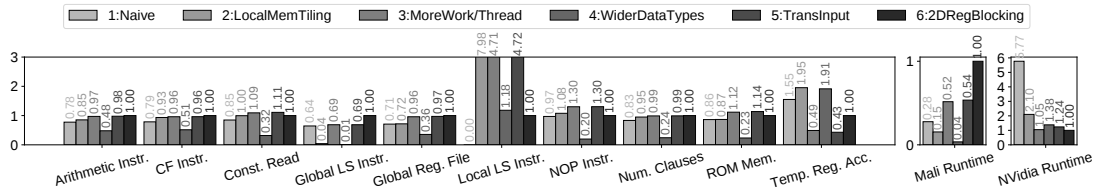


Fig. 4.12: Mali simulation statistics for different versions of SGEMM compared against native Mali, and Nvidia K20m runtimes. All statistics are normalized to SGEMM6, the slowest kernel on Mali.

optimized for NVIDIA GPUs. There is no correlation between speedups on Mali and Nvidia, indicating vastly different architectures.

Again, simulation statistics directly relate to native runtimes, and we see that the optimal solution on Mali (4) executes far fewer instructions than the slowest version (6). Between (5) and (6), arithmetic and control flow instruction counts are similar, however (5) is almost twice as fast as (6). Interestingly, (6) meant to increase register usage, but the increase is just 3% on Mali. Instead, (6) greatly reduces local, and increases global memory accesses relative to (5). (4) almost completely avoids global memory, shifting instead to local memory. This supports the claim in [134], that data movement in mobile platforms is a major contributor to execution time and cost.

4.6.3 Performance Aware Convolutional Neural Network Channel Pruning

While exploring Convolutional Neural Networks (CNNs) in parallel to our simulation work, we encountered unexplained hardware performance variations. In this section, we provide a brief overview of CNNs, followed by execution results from hardware. We then use our functional simulator to explain some unexpected results seen in hardware.

Due to their superior recognition accuracy, CNNs are dominant in several disciplines: computer vision (for image classification [49–51], image segmentation [52,53], objects in image detection [54,55], image style transfer [56], etc.), speech recognition [57] and natural language processing [58,59].

These solutions are making their way into smaller devices, on mobile phones and home personal assistant devices. However, current CNN models are still too large for immediate deployment on resource-constrained devices. Pruning is a widely accepted practice to make these large models suitable to run on such small devices. It is

well understood in the machine learning community that neural networks can produce good inferences even after pruning a substantial amount of their internal parameters (weights) [60–62]. In *Channel Pruning*, entire channels (or filters) are assessed for their importance to determine if these may be removed [63] to produce a slimmer network from the original one, with minimal drop in inference accuracy. Unlike other pruning methods, this produces a compact dense network suitable for the already optimized dense convolutional routines [64]. The details of Convolutional Neural Networks and Channel Pruning are provided in Section 2.6.8.

The parallel nature of computations required by neural networks exposes GPUs as the compute unit of choice, including on mobile and embedded systems for superior FLOPS per watt performance. Dominant in this space are Arm Mali GPUs and Nvidia embedded Jetson GPUs, each programmed via different computing libraries (OpenCL and CUDA). These are called by higher level libraries, such as the Arm Compute Library (ACL) and cuDNN. However, little is known about the performance of these libraries on custom deep learning workloads.

We ran a different neural networks with varying levels of pruning on a number of Mali and Nvidia mobile platforms. When executed on a mobile GPU, we find that uninstructed channel pruning can hurt performance dramatically, up to $2\times$ slowdown in some cases when pruning just 12% of layer channels. We develop the case that inference time on the target device should also be considered when producing smaller networks through channel pruning.

In this study we expose the characteristics of higher level libraries used for deep neural network computations on embedded GPUs, showing their unintuitive behavior in response to changes to convolutional layer size. We experiment with two deep learning libraries - the Arm Compute Library and TVM, observing unintuitive performance patterns caused by their internal heuristics. Intrigued by these observations, we take an in-depth perspective by highlighting these patterns using our Mali GPU simulator where we find that bad splits of convolutional workload into multiple kernels adds substantial overhead, hurting performance. Additional results from Nvidia GPUs can be found in [135].

Our findings are relevant in both the systems and machine learning communities. First, it is important to understand the impact of pruning on inference time, not just classification accuracy, and to identify how the number of channels can be calibrated to improve on both metrics simultaneously. Second, designing new neural network architectures for specific devices should consider the best sizes of convolutional layers

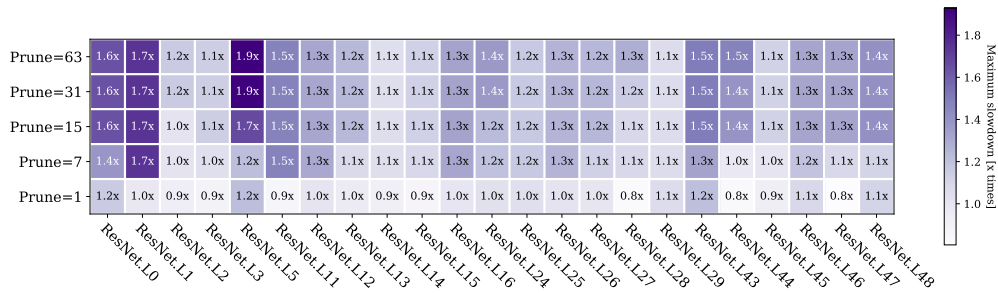


Fig. 4.13: Potential slowdown in execution time of pruned network layers compared to original large model when pruning a number of channels (*Prune*) from the initial number of channels for each convolutional layer of ResNet-50. Performance observed when running on a mobile GPU (Mali G72).

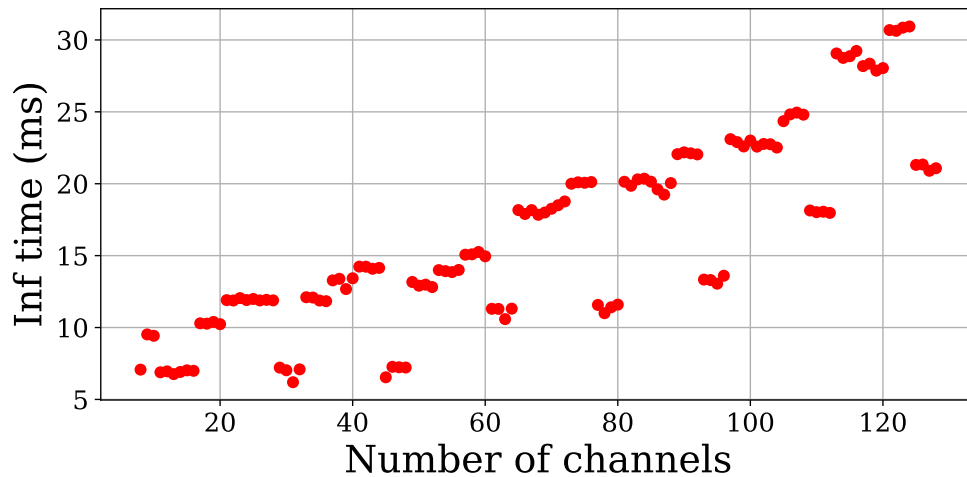


Fig. 4.14: Inference time of a convolutional layer of ResNet-50 run with the Arm Compute Library with varying amount of channel pruning.

for each library and hardware, thus building specialized networks for each runtime environment. And third, library heuristics for workload optimization should be revisited to capture the increasing variation of neural networks and computing devices.

In this analysis we make the following contributions:

- We expose the behavior of two popular deep learning libraries on varying convolutional layer sizes across four different devices.
- These run-time performances are analyzed in-depth through a GPU simulator to understand the built-in heuristics for optimizations and how this performs unjustified splits of workload hurting performance.

4.6.3.1 Channel Pruning

In this work we perform channel pruning without considering the accuracy impact, but our channel pruning approach has the same effect on inference time as when done with accuracy conditions.

Observing the execution time of different pruning levels of a ResNet-50 convolutional layer on a Mali G72 GPU implemented with the Arm Compute Library (Figure 4.14) shows a pattern with two parallel staircases. This can have severe consequences depending on which performance step the pruned layer falls on. In fact, pruning risks introducing slowdown in execution time, with pruned networks potentially running slower than the original unpruned larger network, if libraries and hardware performance are not considered in the pruning process. This situation is presented in Figure 4.13 for running an implementation of pruning with the Arm Compute Library using the GEMM method on the HiKey 970. Pruning at a distance of only 64 channels can match a performance step that introduces up to $2\times$ slowdown in execution time compared to the initial layer (unpruned). Intuitively, some performance steps will offer speedups, but having some levels of pruning that can lead to slowdowns is hazardous and contrary to our expectation that using pruning (fewer network parameters and operations) will produce an universally faster network for any device and with any deep learning libraries.

This unintuitive behavior of deep learning computing libraries, each driven by their own internal optimisations is what motivates this exploration. In the following sections we expose the optimal number of channels for a few deep neural networks, with a range of deep learning libraries and on various devices, expressing the speed-ups achievable by performance aware pruning.

4.6.3.2 Arm Compute Library using the Direct Convolution

In many cases where memory is tightly limited, Direct Convolution is the only option to implement a convolutional layer, due to GEMM expanding the matrix of input patches, which requires almost one order of magnitude more memory for a 3×3 filter, as in the ResNet-50 and in other networks. Here we empirically explore the heuristics adopted in the ACL for these optimizations.

Figure 4.17 shows that these heuristics lead to three execution levels alternating for different channel sizes of ResNet-50 layer 15. Having a linear pattern was expected, since each channel incrementally adds extra work in the deep nested loop of Direct

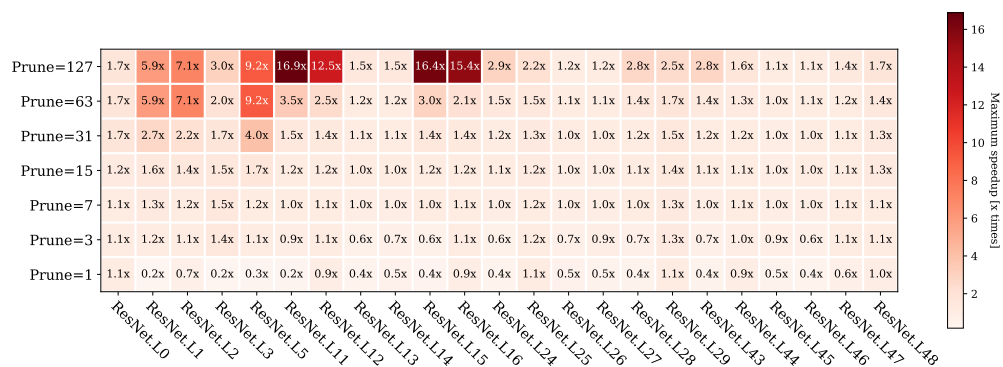


Fig. 4.15: Speedups observed when pruning at different distances within each layer of ResNet-50 using the Arm Compute Library Direct convolution implementation running on the HiKey 970.

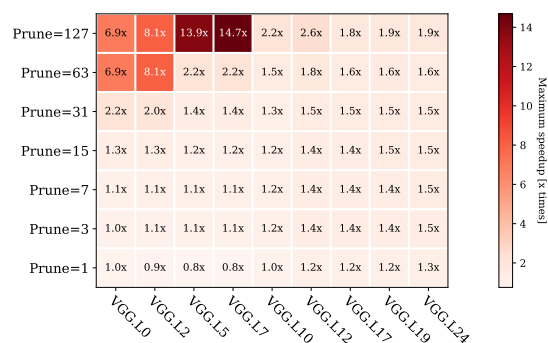


Fig. 4.16: Speedups observed when pruning at different distances within each layer of VGG-16 using the Arm Compute Library Direct convolution implementation.

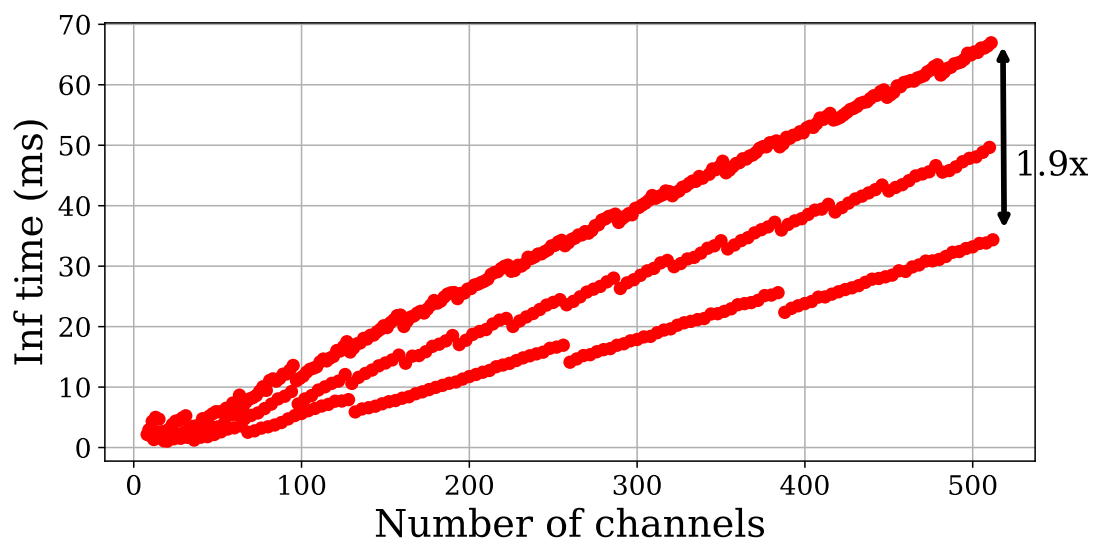


Fig. 4.17: Execution pattern observed for channel pruning of ResNet-50 layer 14 implemented with Arm Compute Library Direct Convolution on HiKey 970 Mali GPU.

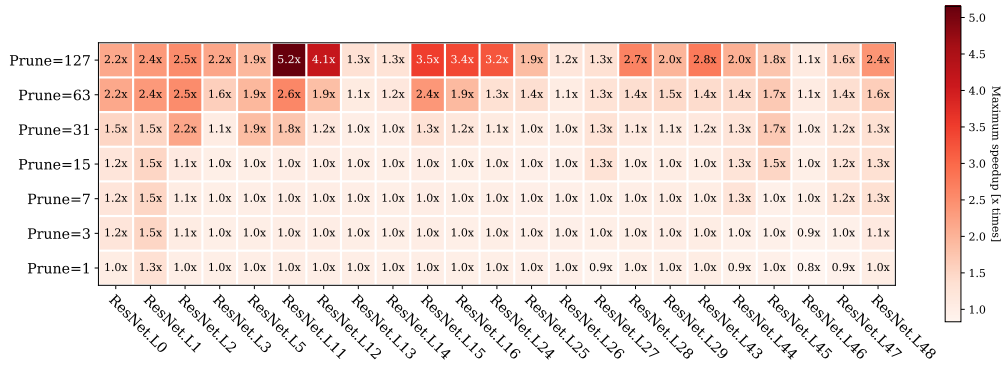


Fig. 4.18: Speedups observed when pruning at different distances within each layer of ResNet-50 using the Arm Compute Library GEMM implementation running on HiKey 970.

convolution, however the three execution levels with up to $1.9\times$ performance difference is unintuitive, and we explore this further in this section with the GPU simulator.

Pruning by just one channel for most of ResNet-50 layers shows a sub-unit speedup (or in actual terms a slowdown) as presented in Figure 4.15, going as low as $0.2\times$ speedup or 80% drop in performance, which is substantial. This indicates to us that optimization heuristics in the ACL are tuned for the standard shape of most popular neural networks, with even a small drop in the number of channels per layer leading to bad decisions from the built-in optimizer. A similar situation is observed for VGG-16 evaluated under the same conditions with the Direct Convolution of ACL (Figure 4.16). Similar patterns were observed when running both on the HiKey 970 and on the Odroid XU4. Considering that Direct Convolution is generally slower than all the other methods, it is understandable that not much development effort has been invested in optimizing this, although for many small devices with limited memory space this may be the only method that can actually execute at all.

4.6.3.3 Arm Compute Library using the GEMM method

A more popular and faster approach for performing the convolutional workload is through GEMM which is also available in ACL. We run the pruned layers with a GEMM implementation, observing some unintuitive patterns.

Figure 4.19 presents the execution time pattern for layer 16 of ResNet-50. Although we see similar steps to those in the cuDNN implementation (which uses an optimised GEMM variant), this implementation of ACL presents two parallel staircases. Also observed from this is that each level is in groups of 4 which matches the

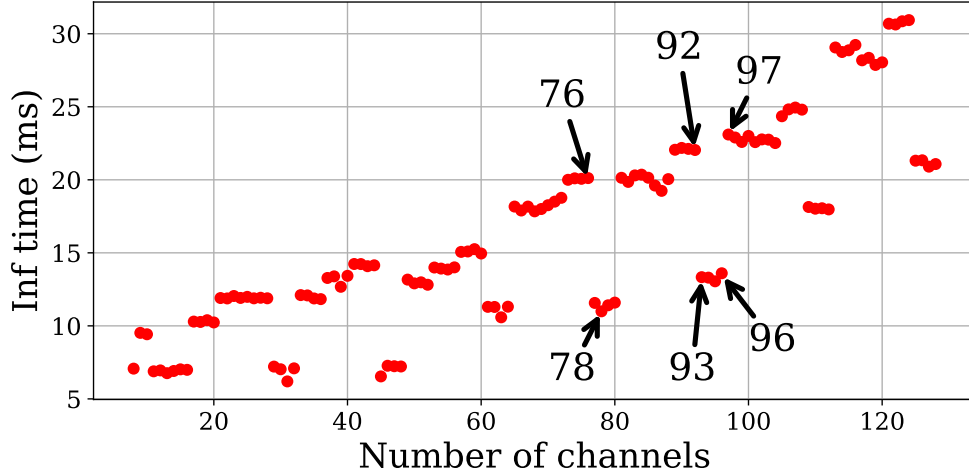


Fig. 4.19: Execution pattern observed for channel pruning of ResNet-50 layer 16 implemented with Arm Compute Library GEMM on HiKey 970 Mali GPU.

size of vectorization, with channels 93 to 96 executing in 14 ms, while near channel sizes 92 and 97 jumping to 23 ms. Another observation is that between 76 and 78 channels (with only just 2 channels difference) inference time is improved from 20.12 ms to 10.996 ms, a $1.83\times$ speedup between the two sizes.

An even wider gap in inference time between close number of channels is observed for layer 45, with 2036 channels inference is performed in 19.69 ms, while for 2024 channels this is performed in 7.67 ms, with a speedup of $2.57\times$, as presented in Figure 4.20.

Similarly to previous implementations, GEMM achieves a speedup of $5\times$ for some layers of ResNet-50 for different levels of pruning (Figure 4.18). Relevant to observe here is that there is no slowdown in the vicinity of the initial number of channels as observed for the Direct convolution, showing that heuristics for this optimization are uniformly modeled for different sizes. This is also observed for the other two networks VGG-16 (Figure 4.21) and AlexNet (Figure 4.22).

4.6.3.4 TVM OpenCL Code Generator

An atypical behavior pattern is observed with code generated by the TVM library. This shows a hybrid behavior between the Direct Convolution implementation of ACL and the GEMM implementation of ACL. Figure 4.25 presents the execution time of pruned layer 14 from ResNet-50. While most channel counts are optimised with the GEMM implementation, there is a significant number of optimization calls instructed to use

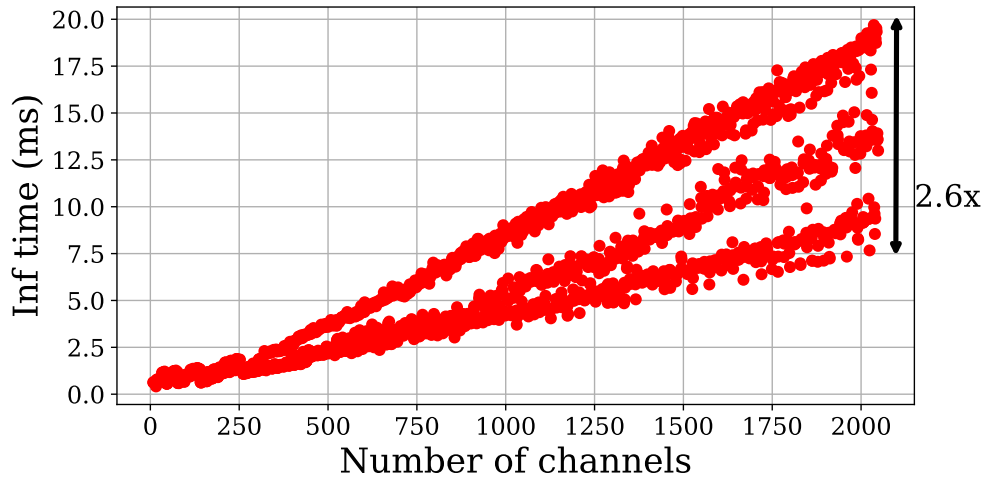


Fig. 4.20: Large gap in inference time between small variations in the number of channels using the GEMM implementation with Arm Compute Library on layer 45 of ResNet-50.

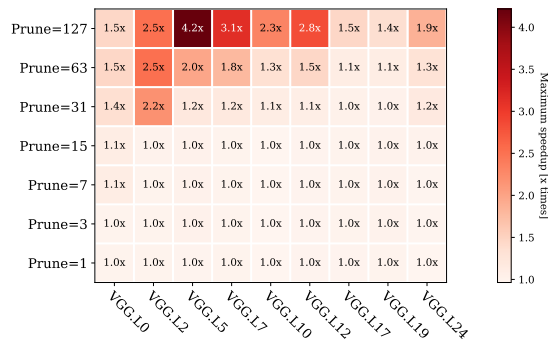


Fig. 4.21: Speedups observed when pruning at different distances within each layer of VGG-16 using the Arm Compute Library GEMM implementation.

direct convolution which we know is generally slower, independent of the underlying hardware specifications. These occasional bad decisions are also observed on the other Mali platforms (Odroid XU4), leading to dramatic drops in performance, up to $13\times$ as observed from Figure 4.24 for some layers. This may also be due to the version of the library, with dynamic developments happening in this space.

4.6.3.5 Channel Pruning Observed Through GPU Simulation

Through the use of higher level libraries, like the ACL, we lose observability that we would normally have when working directly with OpenCL. To understand all the calls and kernel management performed by the Arm Compute Library for different sizes of a convolutional layer, as well as lower-level details about the execution in hardware,

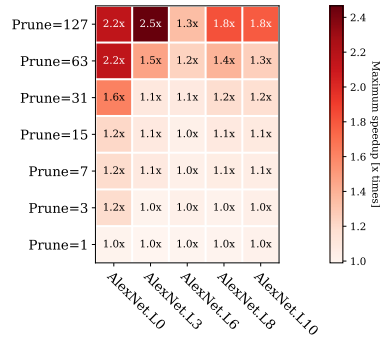


Fig. 4.22: Speedups observed when pruning at different distances within each layer of AlexNet using the Arm Compute Library GEMM implementation.

Kernel Name	No Arithm. Instr.	No Mem. Instr.
im2col3x3_nhwcc	1,365,198	212,152
reshape_to_columns	44,183,104	3,615,808
gemm_mm	706,713,280	36,267,840
gemm_mm	106,006,992	5,440,176

Table 4.4: Arm Compute Library execution for layer 16 of ResNet-50 with 92 output channels.

we executed the workloads in our full-system Mali GPU simulator [136].

As observed in previous experiments, there are unexplained performance differences when we vary the number of channels. In this section, we present our analysis of simulation results for GEMM and Direct Convolution implementations using the Mali GPU Simulator, and relate these directly to runtimes on the Hikey-970.

4.6.3.6 Simulating the GEMM Method

The GEMM method is performed with the 32-bit Arm Compute Library Bifrost implementation. In hardware (Figure 4.20), we observe that inference time dramatically drops when using 93 channels vs. 92 channels, and increases again between layer configurations with 96 and with 97 channels. We developed an OpenCL profiling tool, which instruments OpenCL calls. Using our profiling tool, we can see that all dispatched kernels are the same between the two versions. Upon further inspection with our GPU simulator we can see that when using 93 channels, the number of jobs dispatched to the GPU is the same as the number of OpenCL calls made (OpenCL calls were observed with a profiling tool). However, when using 92 channels, additional

Kernel Name	No Arithm. Instr.	No Mem. Instr.
im2col3x3_nhwc	1,379,034	214,458
reshape_to_columns	44,183,104	3,615,808
gemm_mm	848,055,936	43,521,408

Table 4.5: Arm Compute Library execution for layer 16 of ResNet-50 with 93 output channels.

Kernel Name	No Arithm. Instr.	No Mem. Instr.
im2col3x3_nhwc	1,420,542	221,376
reshape_to_columns	44,183,104	3,615,808
gemm_mm	848,055,936	43,521,408

Table 4.6: Arm Compute Library execution for layer 16 of ResNet-50 with 96 output channels.

jobs are dispatched to the GPU, meaning that the OpenCL runtime makes the decision to split the work. In Figure 4.23 we show the differences in number of jobs executed, as well as additional system-level results. Additional job creation and dispatch requires further communication between the CPU and GPU, and adds to the initialization cost on the GPU. This overhead often outweighs the benefits of dispatching workloads to accelerators. The difference in executed instructions is shown in Tables 4.4 and 4.5 for 92 and 93 channels and similarly for configurations with 96 and 97 channels in Tables 4.6 and 4.7. While the `im2col` and `reshape_to_columns` kernels remain relatively steady while we vary the number of channels, the number of instructions in the `gemm_mm` kernel increases by 4.35%. The bulk of the computation for the `gemm_mm` kernel however, is done in the first kernel, while the second kernel is responsible for only 13% of the computation, showing the scope for improvement.

4.6.3.7 Simulating the Direct Convolution Method

In the direct convolution implementation, we no longer see differences in the number of jobs dispatched, however we still see differences in performance. OpenCL work-group size selection is critical to performance, as it heavily impacts scheduling and caching on the GPU. [137] shows that auto-tuning the OpenCL work group size provides mean speedup of 3.79x over the baseline configuration. In our experiments, the

Kernel Name	No Arithm. Instr.	No Mem. Instr.
im2col3x3_nhwc	1,434,378	223,682
reshape_to_columns	44,183,104	3,615,808
gemm_mm	848,055,936	43,521,408
gemm_mm	35,335,664	1,813,392

Table 4.7: Arm Compute Library execution for layer 16 of ResNet-50 with 97 output channels.

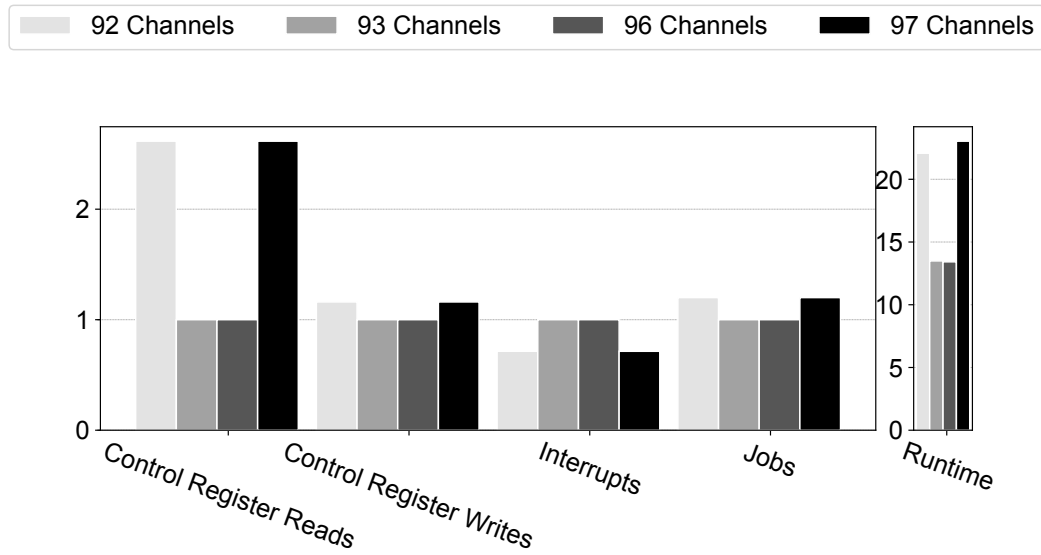


Fig. 4.23: Relative System-Level Results for the GEMM implementation using 96 and 97 channels compared to runtimes on Hikey-970 board.

selection of the work group size for the dispatched OpenCL programs is left to the Arm Compute Library, and is completely invisible to the user. Examining channels 90-93, we see a wide range of reported runtimes, despite the fact that the number of executed instructions only increases by approximately one percent with each added channel. However, we observe different work-splitting paradigms between successive layer sizes. As shown in Table 4.8, the slower instances (91,93), use work group dimensions 1x1x8, while 90 and 92 channels use 2x1x8 and 4x1x1 respectively. Auto-tuning of the workloads and examining the effects of scheduling and caching have been left for future work.

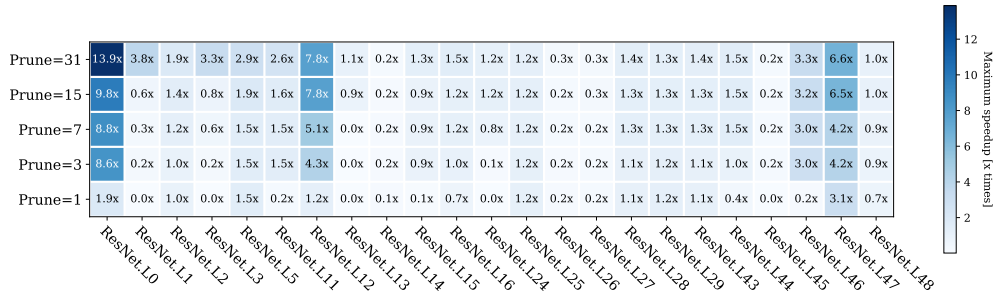


Fig. 4.24: Speedups observed when pruning at different distances within each layer of ResNet-50 using a TVM library implementation on HiKey 970.

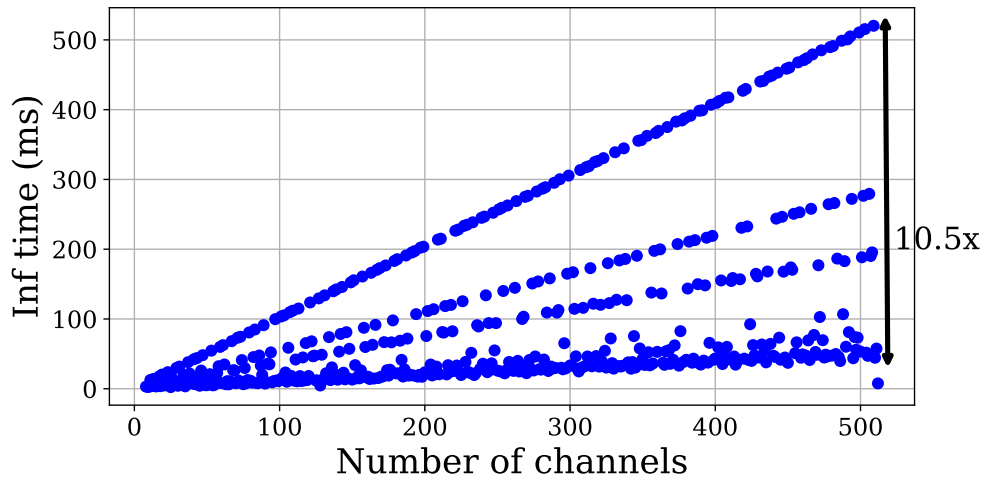


Fig. 4.25: Layer 14 of ResNet-50 implemented with TVM OpenCL. Many sizes are untuned out of the box, showing a large variation due to uninstructed heuristics on HiKey 970.

Number of Channels	X	Y	Z	Relative Executed GPU Instructions	Measured HW Time
90	2	1	8	1.0	167.8716
91	1	1	8	1.011	198.0468
92	4	1	1	1.023	168.8311
93	1	1	8	1.034	202.7299

Table 4.8: Arm Compute Library Direct Convolution work group sizes identified using GPU Simulator vs. Runtime measured on Hikey-970.

4.6.3.8 Discussion

Our exploration highlights some important limitations in deep learning libraries, showing that pre-designed heuristics fail for some arbitrary sizes of neural networks. As seen, pruning a number of channels can introduce slowdown rather than speedup, thus hurting performance, so these levels of pruning should be avoided. However, as expected, other pruning levels will run faster than the initial network configuration, where the library produces efficient GPU kernels. These optimal configurations can be found by profiling the kernel execution. These observations are relevant for pruning to the right number of channels and avoiding those levels that instruct optimizations which hurt performance. Instead by profiling, we can reduce the search space to the ones with superior speedup to test for accuracy in the network size-inference accuracy trade-off. Runtime optimal neural networks can be generated by coupling profiled performance on device with convolutional inference accuracy of pruned layers to instruct the best pruning level. We have initiated work in this direction showing that both execution time and inference accuracy can be considered simultaneously for efficient network compression to a target device [66], although other research directions in library optimization and hardware design can also be considered.

From this exploration we find that no optimal library exists to outperform across all neural network layers. Neither Arm Compute Library, nor TVM dominates even with their auto-tuning enabled. Future solutions integrating optimizations from across different deep learning libraries could adapt their computation based on network and layer configuration to improve execution with hardware aware performance.

4.7 Implementation Details

This section dives deeper into the implementation details of the simulation framework, explaining further what we did, how we did it, and identifying supporting work.

4.7.1 User-Mode Simulation (CPU)

The original development efforts around this work were focused on the Mali Midgard GPUs, which preceded Bifrost. At the time, the GPU OpenCL driver (and all dependent libraries) was taken from an Odroid-XU3 development board, and the first simulation infrastructure was developed as a user-mode simulation with dynamically loaded libraries. For simple programs this can work, however we found this approach to be unsustainable in the long-term, as it required manually copying libraries from a real platform into our simulation space every time a library was missing. User-mode simulation also doesn't allow us to execute the Linux kernel, into which the GPU drivers are integrated. As such, we moved to full-system simulation.

4.7.2 GenC Thumb2 Model

The libraries taken from the Odroid-XU3 were compiled for Arm's Thumb-2 instruction set, for which there was no GenC model at the time. Naturally, executing the code from the Odroid-XU3 board in the simulator, required us to implement the Thumb-2 GenC model, and as such, it is a contribution to this thesis. The Thumb-2 model added additional instructions to the Thumb instruction set, and also added 32-bit instructions, which are interleaved with the 16-bit Thumb instruction set. This required the addition of additional decode infrastructure able to identify the difference between 16 and 32 bit instructions, which we implemented. Furthermore, we implemented the Thumb-2 model, which included 30 new instruction formats, and 206 Thumb-2 instructions. Encoding such a model, with the necessary improvements in infrastructure was a multi-month project, the fruits of which can be found at gensim.org/home.

4.7.3 GenC GPU Model

GenC is a flexible, but easy language, and can support the majority of GPU features in its description. We described the GPU instruction set in GenC, allowing us to automatically generate the execution core. Other components of the GPU, for example job dispatch, thread grouping, and scheduling, are implemented in C++, similarly to how

GenSim uses auto-generated instruction set modules, but a hand-coded engine. We add an additional layer to the auto-generation, as the entire syntax description is auto-generated from an HTML version of the Bifrost Instruction Set Architecture Reference Manual. The majority of instruction semantics are also described in GenC, with some exceptions for features which are not supported by GenC, such as reading program descriptors. The program descriptors are nested structures of arbitrary length, stored in memory, which cannot currently be described in GenC. Instruction semantics which are not described in GenC, are implemented in C++. Future work could include generating the semantics from a description in the reference manual as well. For this, the reference manual would need to contain parse-able descriptions of instruction semantics, for example in GenC, or using formal models, as proposed in [138]. GenC support for additional metadata, such as program descriptors could be a further addition.

4.7.3.1 Indirect Register Access

Bifrost uses indirect register addressing in its instruction set. Instructions are packed into instruction words, which contain:

- An FMA pipeline instruction.
- An ADD pipeline instruction.
- A register block description,
- An FAU RAM index.

Rather than addressing register directly, each instruction can address one of seven fields with the register block. These fields in turn specify whether the instruction will read values from:

- The register file,
- clause constants,
- directly from a previous instruction,
- a NULL value,
- or the FAU RAM.

Describing the infrastructure for decoding this flow possible in GenC. The general purpose register file, FAU RAM, and pipeline registers are all described as part of the system specification in GenC. Listing 4.1 shows the Bifrost system description. Annotation ① shows the general purpose register file, ② shows the FAU RAM definition, and ③ shows the pipeline registers. The semantics file then defines a helper function for decoding the register block, which is shown in Listing 4.2. Embedded constants and the FAU RAM are read-only state that reside in dedicated caches, which are automatically filled in from the program descriptors. The program descriptors are nested structures of arbitrary length, which cannot currently be described in GenC. As such, the decoder for the program descriptors is implemented in C++, and additional intrinsics are defined to access the FAU RAM and embedded constants from the GenC model.

4.7.4 Standalone GPU Hexdump Simulator

For functional verification against the reference simulator (kindly provided by Arm), we developed a standalone mode for our GPU simulator, where execution is driven from a hexdump of program memory. The hexdump includes the program shader, descriptors, and a trace of system register accesses and interrupts. The hexdump format is used by our reference simulator, making the two simulators binary compatible.

4.7.5 Fuzz Testing

Debugging a simulator is a time consuming and not always rewarding process. Instruction implementations can sometimes work correctly in 90% of cases, but sporadically, on certain inputs, may operate incorrectly. In order to support debugging efforts and improve resilience of the simulator, we developed an instruction fuzzer, which allows us to use the GPU assembly format to automatically test specific instructions with either randomized, or specifically selected inputs.

To use the fuzz testing infrastructure, the user provides the assembly format for the instruction which is to be fuzz tested. This includes the instruction name, the registers accessed, and any additional modifiers, which can for example specify the data type. Next, the driver for the fuzz testing framework matches the provided assembly format to an assembly template. The assembly template is the simplest possible Bifrost assembly program, which can capture the instruction execution. The template also ensures that the program is sensible, i.e. it provides a harness for providing inputs to a

Listing 4.1: GenC Bifrost System Description

```

2 AC_ARCH(bifrost)
3 {
4     // General Purpose Registers
5     ac_regspace(256) ① → The register space defines the register
6     {                                                         file, as described by the ISA.
7         bank RB16 (uint16, 128, 2, 2, 0);
8         bank RB32 (uint32, 64, 4, 4, 0);
9         bank RB64 (uint64, 32, 8, 8, 0);
10
11         bank RBF (float, 64, 4, 4, 0);
12         bank RBD (double, 32, 8, 8, 0);
13
14         slot PC (uint32, 4, 63) PC;
15         slot SP (uint32, 4, 52) SP;
16     }
17
18     ac_regspace(16384) ② → We define the Bifrost FAU RAM
19     {                                                         as a register space.
20         bank FAU32 (uint32, 512, 4, 4, 0);
21         bank FAU64 (uint64, 256, 8, 8, 0);
22         bank FAUF (float, 512, 4, 4, 0);
23         bank FAUD (double, 256, 8, 8, 0);
24     }
25
26     // Pipeline Registers
27     ac_regspace(28) { ③ → Here we define pipeline registers, along
28         slot PFMA32 (uint32, 4, 0) PFMA32;                   with their shadow copies, so that current
29         slot TFMA32 (uint32, 4, 8) TFMA32;                   and previous values can be preserved.
30         slot PADD32 (uint32, 4, 16) PADD32;
31
32         slot PFMA64 (uint64, 8, 0) PFMA64;
33         slot TFMA64 (uint64, 8, 8) TFMA64;
34         slot PADD64 (uint64, 8, 16) PADD64;
35
36         slot PFMAF (float, 4, 0) PFMAF;
37         slot TFMAF (float, 4, 8) TFMAF;
38         slot PADDF (float, 4, 16) PADDF;
39
40         slot PFMAD (double, 8, 0) PFMAD;
41         slot TFMAD (double, 8, 8) TFMAD;
42         slot PADDD (double, 8, 16) PADDD;
43
44         slot SIDEBAND32 (uint32, 4, 24);
45     }
46
47     ac_regspace(1){ → Here we define a runtime flag that
48         slot WRITES_PC(uint8, 1, 0) WRITES_PC;               specifies if the PC was written to.
49     }
50
51     ac_wordsize 32;
52
53     ARCH_CTOR(bifrost) → Here we refer to the ISA files,
54     {                                                         which describe instruction syntax.
55         ac_isa("bifrost_fma32_isa.ac");
56         ac_isa("bifrost_add32_isa.ac");
57         ac_isa("bifrost_fma64_isa.ac");
58         ac_isa("bifrost_add64_isa.ac");
59         set_endian("little");
60     };
61 };

```

Listing 4.2: GenC Bifrost Instruction Semantics

```

1
2 execute(fma_64_IMAD_i64)
3 {
4     Three values are read from registers, FAU RAM, or embedded
5     constants, as specified by the instruction encoding.
6
7     uint64 val = read_regblock_fma_01_64_t_u64(inst.access_type_0, inst.src0_global);
8     uint64 val1 = read_regblock_fma_01_64_t_u64(inst.access_type_1, inst.src1_global);
9     uint64 val2 = read_regblock_fma_23_64_t_u64(inst.access_type_2, inst.src2_global);
10
11     uint64 result = (val * val1) + val2; ← The IMAD operation is performed.
12
13     write_register(TFMA64, result); ← The result is written to
14     a temporary register.
15 }
16
17 helper uint64 read_regblock_fma_01_64_t_u64(uint8 access_type, uint32 src)
18 {
19     switch(access_type)
20     {
21         case 0:
22             return read_register_bank(RB64, src); ← Read from the register file.
23         case 1:
24             return read_register_bank(RB64, src);
25         case 3:
26             return (uint64)0; ← Read zero constant.
27         case 4:
28             {
29                 if (src < 7)
30                     return fau_local_read_64(src); ← Read from local memory.
31                 else if (src < 128)
32                     return ec_read_64(src); ← Read embedded constant.
33                 else
34                     return fau_read_64(src); ← Read from FAU RAM.
35             }
36         case 5:
37             return (uint64)0;
38         case 6:
39             return read_register(PFMA64); ← Read from temp FMA register.
40         case 7:
41             return read_register(PADD64); ← Read from temp ADD register.
42         default:
43             {
44                 trap();
45                 return (uint64)0;
46             }
47     }
48     return (uint64)-1;
49 }

```

single instruction, and reading the output value. The driver then inserts the provided assembly instruction into the template, and generates random values to populate the registers used as arguments to the instruction. The hexdump is then executed using both our simulator (using the standalone hexdump mode), and the reference simulator, and results are compared. If the results are identical, the test passes, otherwise, a bug is recorded. We typically execute the fuzz tester with at least 10 random values, for each possible configuration of the instruction (which can number in the hundreds for complex instructions).

4.7.6 GenSim & Captive Integration

The GPU simulator is designed to work coupled with a CPU simulator inside a full system simulation framework, which enables correct execution of the complete software stack, and in turn, a faithful simulation of the GPU. We integrate the GPU both with the GenSim simulation framework, which uses the ARMV7-A CPU model, and with Captive, which takes further advantage of host virtualization infrastructure, and uses the AARCH64 CPU model. The integration is similar across both frameworks, and the GPU implementation remains the same.

4.7.6.1 GPU Device

The Mali GPU is defined as a device within the simulation framework, similarly to other, already existing devices, such as timers, memory controllers, interrupt controllers, network and storage devices, and many more. This can be seen in Figure 4.3 shows the definition of the Mali GPU within the Captive VIRT platform specification. ④ defines the base address of the GPU's system registers. ⑤ defines the GPU ID, which is stored in a hardware register, and queried by the Bifrost kernel driver. ⑥ provides a map of available job slots. Mali uses job slot 2 for OpenCL kernel execution, which we enable by passing in the value 8, which in binary is an on bit in the third position, with other bits off (0b100). ⑦ defines the interrupts. The Mali GPU uses three interrupts in the generic interrupt controller (GIC) - one for the GPU, one for the GPU's Job Controller, and one for the GPU's MMU. ⑧ shows how the GPU is registered in the system. This can easily be related to the device tree used when booting Linux within our simulation framework. Listing 4.4 shows the entry for the Mali GPU in the device tree. ⑨ and ⑩ show that the GPU's system registers start at address 0x8080000 and take up 0x4000 bytes. ⑪ shows the interrupt assignment for

Listing 4.3: Mali Device Definition

```

1 mali = new Mali(0x08080000, ④
2   0x60000001, ⑤
3   8, ⑥
4   hcfg.gpu_num_host_threads,
5   3,
6   &gic0->get_irq_line(46), // GPU ⑦
7   &gic0->get_irq_line(44), // JOB
8   &gic0->get_irq_line(45), // MMU
9   false, // Dump Shader
10  hcfg.gpu_metrics, // Metrics
11  hcfg.gpu_warp_metrics,
12  hcfg.gpu_append_metrics,
13  hcfg.gpu_mem_tracing,
14  hcfg.gpu_icache_tracing,
15  hcfg.gpu_tracing, // Tracing
16  hcfg.gpu_fast_tracing, // Fast Tracing
17  hcfg.gpu_disasm, // Disasm
18  false, // Logging
19  false, // Special Log
20  false, // Timing
21  false, // Replay
22  stringer); // Stringer device for communicating to host from guest
23  cfg.devices.push_back(GuestDeviceConfiguration(0x08080000,*mali)); ⑧
24

```

The GPU is registered at the address specified in the device tree.

This is the GPU ID for the Mali-G71.

Number of shader cores.

The interrupts registered as specified in the device tree.

Various metric collection options are exposed to the user.

Listing 4.4: Mali Linux Device Tree Entry

```

1 gpu@8080000 { ⑨
2   compatible = "arm,mali-t602", "arm,mali-t60x", "arm,mali-t6xx", "arm,mali-midgard";
3   reg = <0x0 0x08080000 0x0 0x4000>; ⑩
4   interrupts = <0 44 4 0 45 4 0 46 4>; ⑪
5   interrupt-names = "JOB", "MMU", "GPU";
6   };
7
8

```

The GPU address.

The GPU physical address space and size.

GPU->CPU interrupt sources registered in Linux Kernel.

the GPU.

4.8 Summary & Conclusion

In this chapter we have presented the first ever fully retargetable full-system simulator supporting an unmodified software stack for a commercially available, state-of-the-art mobile GPU, fulfilling the first goal of this thesis:

- ✓ To develop a simulation framework, which accurately simulates a state-of-the-art mobile GPU in a full-system context, enabling the use of *unmodified* vendor-supplied drivers and JIT compilers, operating in an *unmodified* target operating system, and executing *unmodified* applications.

Its validated instruction-accurate performance model enables more accurate insights into the GPU's operation than with simulators claiming cycle-accuracy for crudely

approximated architectures and non-standard runtime environments. Our full-system approach will ensure a long-lasting simulator, requiring little maintenance as new toolchains are released. While we draw on several known simulation techniques, we have demonstrated the feasibility of accurate full-system CPU/GPU simulation at performance levels better than those of existing simulators with inaccurate supporting software stacks, fulfilling the second goal of this thesis:

- ✓ To develop a simulation framework, which supports simulation speeds that enable the user to execute complete and complex applications typical of modern GPU workloads.

Our simulation approach enables us to gain insights into mobile GPU workloads including system-level transactions between the CPU and GPU - inaccessible using other GPU simulation approaches. Our simulator can characterize mobile GPU applications with accuracy unavailable using existing GPU simulators and provides a most useful tool to researchers and developers alike, fulfilling the third goal for our simulation framework:

- ✓ To develop a simulation framework, which provides useful performance statistics, without the overhead of cycle-accurate simulation.

However, further improvements can still be made. While the functional simulator can provide some insights regarding application performance, it does not provide the flexibility desired by architects. Chapter 5 builds and improves on the framework presented in this chapter, by exploring numerous techniques for fast performance modelling, and developing a configurable, novel trace-based approach, where the model is tuned against existing hardware. The configurable parameters in the trace-based simulator allow for both architectural and micro-architectural flexibility in developing GPU designs.

Fast Performance Modelling

One of the key uses for simulators is performance modelling, however, as we have already discussed, (1) simulating large design spaces using detailed simulators is infeasible, and (2) simulating modern applications driven by real software stacks is not possible using existing cycle-accurate simulators. Chapter 4 demonstrated that our full-system simulation framework already provides useful insights into Arm Mali Bifrost GPU performance, achieving the third goal of this thesis:

- ✓ To develop a simulation framework, which provides useful performance statistics, without the overhead of cycle-accurate simulation.

The simulator provides insights into application performance, but it does not provide the flexibility desired by architects. For example, architects may want to explore how performance changes if we change the number of cores, or the warp size, but the functional simulator will tell us very little if we change these configurations. How can we improve on this goal?

After motivating the requirement for fast modelling in Section 5.1, we examine two approaches to fast modelling - first machine-learning based modelling in Section 5.2, followed by trace-driven simulation in Section 5.3, both of which are used in combination with our full-system simulator presented in Chapter 4.

5.1 Motivation

Fuelled by fierce competition in the mobile phone market, mobile GPUs evolve at a rapid pace. For example, between 2012 and 2016, Arm developed four different generations of the Mali Midgard GPU architecture with a total of twelve different

	Midgard						Bifrost						Valhall			
	T720	T760	T820	T830	T860	T880	G71	G72	G31	G51	G52	G76	G57	G77	G68	G78
Warp Size (#)	N/A	N/A	N/A	N/A	N/A	N/A	4	4	4	4	8	8	16	16	16	16
EEs (#)	N/A	N/A	N/A	N/A	N/A	N/A	3	3	2	1 or 3	3	3	1	1	1	1
Cores (#)	1-8	1-16	1-4	1-4	1-16	1-16	1-32	1-32	1-6	1-3	1-6	4-20	1-6	7-16	6	7-24
L1C (KB)	16	16	16	16	16	16	16	16	4	16	16	16	16	16	16	16
L2C (KB)	64-256	256-2048	32-256	32-256	256-2048	256-2048	128-2048	128-2048	32-512	32-512	32-512	512-4096	64-512	512-2048	512-2048	512-2048
Max Workgroup (#)	256	256	256	256	256	256	384	384	512	768	768	768	1024	1024	1024	1024
Year	2014	2014	2015	2015	2015	2016	2016	2017	2018	2016	2018	2018	2019	2019	2020	2020
Variants (#)	3	4	1	3	1	3	3	3	1	2	3	3	1	2	None Yet	1

Table 5.1: Evolution of Arm GPU architectures from Midgard through Bifrost to Valhall. While new architectures are introduced every 4-5 years, GPU variants for different market segments or with feature improvements appear on a bi-annual release cycle. Common changes include scaling the number of cores, execution engines, warp size, and cache size.

GPU variants. Between 2016 and 2018, Arm released three generations of the Bifrost GPU architecture, with six different variants of the GPU, and have since moved onto the Valhall architecture, of which there are already three variants available. Further revisions of these GPU architectures have been released by customers of Arm’s GPU IP, which feature different numbers of cores, core frequencies, and L2 cache sizes. The evolution of Arm GPUs and all recent variants is described in Table 5.1.

The fast succession of mobile GPU designs dramatically increases the pressure on the early GPU design and development phase, where key architecture parameters, configuration options and micro-architectural decisions are evaluated before commencing implementation. GPU design teams are presented with a large number of possible design options – even before considering the effects of the surrounding components of the System-on-Chip (SoC) – such as memory and CPU – both of which impact on GPU performance, and which can also be customized. Traditional detailed simulation approaches, which require substantial development effort and are slow in their use, are not sustainable. Instead, what is needed are flexible and scalable tools supporting chip designers in early GPU Design Space Exploration (DSE), providing approximate yet reliable estimates of a GPU design under consideration.

Short design cycles require architects to rapidly explore the design space, but by definition, cycle-accurate simulators, which are the primary tool, contradict this requirement. Furthermore, GPUs operate as accelerators linked to a CPU and a tightly coupled software stack, however existing detailed GPU simulators are not fast or flexible enough to execute complete multi-kernel applications with heavy CPU-GPU inter-

action. At the same time, programmers must already start developing software drivers, compilers, and applications, before the designs are finalized, and long before any chip is eventually fabricated.

Cycle-accurate simulation however, is only a strong requirement when evaluating micro-architectural improvements. The majority of these designs on the other hand, are iterative, meaning that the overall design of the GPU remains the same, with new GPU families only introduced every couple of years. Instructions may be added, the warp size or core count may be changed, but all members of the same architecture family will share the majority of the instruction set. We argue that for early, iterative DSE, the requirement for cycle accuracy can be relaxed, and cycle-accurate models can be replaced with faster models exhibiting good performance prediction.

5.1.1 The Arm Mali GPU

Between 2014 and 2020, Arm released 16 different GPUs, all with varying architectural and micro-architectural configurations. The variants are presented in Table 5.1. Each of these have been further configured by Arm’s customers, with at least 34 variants having been implemented at the time of writing. However, these GPUs all belong to just three GPU families - Midgard, Bifrost, and Valhall. Within a GPU family, the architecture remains largely the same, with minor variations to the instruction set, and some micro-architectural improvements. However, as presented in Section 5.3.4, optimizations to the pipeline or memory have far less impact on performance than the high-level architectural decisions which are at the forefront of designing different variants of GPUs. These parameters include both configurations that are exposed to the customer, for example the cache size and number of cores, as well as configurations decided by Arm, for example the number of execution engines per core and the number of threads per warp. Simulating these different variants is necessary, however detailed simulation is not required to predict the performance of different GPU variants within a single family.

In order to demonstrate potential variations, we turn back to the Arm Bifrost GPU architecture, and the MALI-G71 GPU, which we use as the baseline for our model. The architecture features up to 32 unified SCs, and a single logical L2 GPU cache that is split into several fully coherent physical cache segments. Full system coherency support and shared main memory tightly couples the GPU and CPU memory systems. For this, Bifrost features a built-in MMU supporting AArch64 and LPAE address modes.

A central Job Manager interacts with the driver stack and orchestrates GPU jobs.

Shader Core (SC)s are blocks consisting of Execution Engine (EE)s—three in the Mali-G71—and a number of data processing units, linked by a messaging fabric.

The EEs are responsible for executing the programmable shader instructions, each including an arithmetic processing pipeline as well as all of the required thread state.

The arithmetic units implement a vectorization scheme to improve functional unit utilization. Threads are grouped into bundles of four (a “quad”), which fill the width of a 128-bit data processing unit. Further details can be found in Section 2.3.

5.2 Prediction Using Machine Learning

Modern advances in machine learning have clearly shown its value, however there are significant trade-offs to consider when developing machine learning based models. On one hand, many machine learning models are available out of the box, and can easily be used to train a performance model, which then provides rapid performance predictions. However, even with the availability of models, there is still significant work required to build a good predictor. Features to be used in the model must be carefully selected, and vast amounts of data must be collected to achieve accurate predictions without over-fitting.

Once this effort has been made, and an accurate predictor has been developed, the information available from the model is still limited - while providing a performance prediction, the model offers no explanation for the prediction. Furthermore, the parameters used as weights in the machine learning model are not necessarily easy to comprehend, and therefore cannot be modified by humans to predict changes in performance when changing the characteristics of the architecture or executing program.

In our efforts to build a machine learning based model, we rely on metrics already available to us from our functional, full-system simulator as described in Chapter 4.

5.2.1 A Naïve Machine Learning Approach

Our initial approach attempts to simply use the outputs of the functional simulation directly in a linear regression model. We use attributes listed in Table 5.2 without any pre-processing. We use the pre-packaged linear regression model from WEKA [139], with 10-fold cross-validation. It is immediately clear however, that this approach does not give good results, with a Mean Absolute Percent Error (MAPE) of 703%. The

equation for predicting runtime is presented in Equation 5.1.

$$\begin{aligned}
 \text{Runtime} = & 1.3\text{global_ls_i} + 2 \times 10^{-1}\text{nop_i} - 4 \times 10^{-1}\text{cf_i} \\
 & - 3 \times 10^2 \text{barriers} + 1.4 \times 10^3 \text{div_warps} + 2 \times 10^{-1}\text{temp_reg_r} \\
 & - 1 \times 10^{-1}\text{grf_r} - 2 \times 10^{-1}\text{grf_w} + 1 \times 10^{-1}\text{const_reads} \\
 & + 5.6 \times 10^4 \text{ctrl_reg_w} - 1.1 \times 10^6 \text{interrupts} - 6.0 \times 10^5
 \end{aligned}
 \tag{5.1}$$

Attribute
Global Load/Store Instructions
NOP Instructions
Number of Clauses Executed
FAU RAM Reads
Executed Instruction Count
Constant Reads
GRF Reads
Control Flow Instructions
GRF Writes
Arithmetic Instructions
Temporary Register Reads
Local Load/Store Instructions
Barriers Hit
Divergent Warps
Pages Touched
Number of Workgroups
Divergence Percentage
Number of Control Registers Read
Number of Interrupts
Number of Control Registers Written

Table 5.2: Attributes sourced from functional simulation.

5.2.2 Feature Selection

The first step in building a machine learning predictor is to examine the available data. The functional simulator makes available 20 different parameters characterizing GPU execution. The graphs in Figure 5.1 show clear correlation between the runtime and some, but not all of the modelled attributes. The visual perception is reinforced by numerical values for correlation listed in Table 5.4. The most strongly correlated attributes are focused around instruction counts and breakdown of instruction types.

These correlations reinforce the idea in Chapter 4 that statistics gathered during functional simulation can be used to make valid performance comparisons.

We use the correlation as an attribute selector for our machine learning model, choosing the five attributes which correlate most strongly with measured runtimes. The new feature set comprises: Global Load/Store Instructions, NOP Instructions, Number of Clauses Executed, FAU RAM Reads, and Executed Instruction Count. However, this approach results in significantly worse performance than the naïve model, with a MAPE of 1398%.

5.2.3 Principal Component Analysis

Another popular technique used for feature selection and dimensionality reduction is Principal Component Analysis (PCA). A reduction in the dimensions of the dataset results in the features presented in Table 5.3. Once again however, the resulting error using PCA is much higher than the baseline model, at 2296% error.

$3 \times 10^{-1} \text{exec.i} + 3 \times 10^{-1} \text{fau.r} + 3 \times 10^{-1} \text{grf.r} + 3 \times 10^{-1} \text{const.r} + 3 \times 10^{-1} \text{grf.w}$
$- 5 \times 10^{-1} \text{ctrl.reg.w} - 5 \times 10^{-1} \text{interrupts} - 5 \times 10^{-1} \text{ctrl.reg.r} - 3 \times 10^{-1} \text{avg.clause.size} - 3 \times 10^{-1} \text{div.warps}$
$4 \times 10^{-1} \text{div.warps} + 4 \times 10^{-1} \text{barriers} - 3 \times 10^{-1} \text{global.ls.i} + 3 \times 10^{-1} \text{div.pct} - 3 \times 10^{-1} \text{ctrl.reg.r}$
$- 7 \times 10^{-1} \text{wkgrps} - 5 \times 10^{-1} \text{avg.clause.size} + 4 \times 10^{-1} \text{div.pct} - 3 \times 10^{-1} \text{pages} - 2 \times 10^{-1} \text{temp.regs}$
$6 \times 10^{-1} \text{pages} + 5 \times 10^{-1} \text{local.ls.i} - 2 \times 10^{-1} \text{avg.clause.size} - 2 \times 10^{-1} \text{wkgrps} + 2 \times 10^{-1} \text{cf.i}$
$- 6 \times 10^{-1} \text{div.pct} - 5 \times 10^{-1} \text{wkgrps} + 3 \times 10^{-1} \text{temp.regs} - 3 \times 10^{-1} \text{local.ls.i} + 3 \times 10^{-1} \text{pages}$
$- 5 \times 10^{-1} \text{div.pct} - 5 \times 10^{-1} \text{pages} + 4 \times 10^{-1} \text{barriers} + 3 \times 10^{-1} \text{div.warps} - 3 \times 10^{-1} \text{temp.regs}$

Table 5.3: Principal components created through Principal Component Analysis on the full dataset.

5.2.4 Manual Feature Modification

The default outputs from the functional simulator make no reference to any of the inherent parallelism found in GPUs, exposing only aggregated counts of instruction types. Instead, we rely on the machine learning model to identify the hidden relationships between the functional simulation output and GPU performance. We can however, approximate the effect of using multiple parallel units using the information at hand. To do this, we modify the instruction counts for all different types of instructions to account for the dynamic workgroup size, and number of concurrent workgroups executing on the GPU. We extract the workgroup size using the functional simulator - the GPU driver compiles the workgroup dimensions into the job descriptor. Each GPU core can execute one workgroup at a time, so the number of concurrently executing

workgroups is 8. The workgroup size has a significant impact on the performance of a kernel. A small workgroup size means that the GPU's parallel capabilities are being under-utilized. Larger workgroup sizes on the other hand enable the GPU to hide the latency of long memory operations by scheduling instructions from different warps during memory stalls. Furthermore, all threads in a workgroup will share the L1 cache, which is beneficial to applications with strong locality. When modifying the features, we first calculate the average number of instructions per workgroup, by dividing the total number of instructions by the workgroup size. We use these manually processed features in the following experiments.

$$\boxed{Total\ Instruction\ Latency = \#Instructions / \#Workgroups * \lceil \#Workgroups / 8 \rceil} \quad (5.2)$$

5.2.5 An Artificial Neural Network Model

Artificial Neural Networks (ANNs) are inspired by the structure of the human brain, with multiple input neurons connected to an output via a network of multiple hidden layers. Each connection applies a weight calculation, finally resulting in the predicted output from the final layer. Neural networks can handle more complex relationships between input and prediction than linear regression, as they can handle arbitrarily deep feature sets and non-linear relationships. For example, when identifying people in an image, each layer of the neural network can apply to a different feature of the human face - shape of the head, placement of eyes and nose, or length of hair. Predicting GPU performance can be similarly complex. The performance of a GPU is a complex function of a number of hardware and software features, which are not necessarily linear.

We use the multi-layer perceptron model in WEKA to train and test a neural network for predicting GPU performance. We once again train and test the model with six different inputs - first with no modification, feature selection using correlation, and dimensionality reduction using PCA. These experiments are then repeated with manually modified features to account for the parallelism in the GPU. Error results from the predictors are presented in Table 5.5.

5.2.6 Conclusions

The best average error we were able to achieve across our benchmarks is 222% error, training the ANN model using the manually modified feature set. This however, is



Fig. 5.1: Correlations between parameters collected using functional simulation and runtimes measured using hardware.

Correlation	Attribute
9.5×10^{-1}	Global Load/Store Instructions
9.3×10^{-1}	NOP Instructions
9.0×10^{-1}	Number of Clauses Executed
8.3×10^{-1}	FAU RAM Reads
8.1×10^{-1}	Executed Instruction Count
8.1×10^{-1}	Constant Reads
8.0×10^{-1}	GRF Reads
7.1×10^{-1}	Control Flow Instructions
6.8×10^{-1}	GRF Writes
5.9×10^{-1}	Arithmetic Instructions
3.4×10^{-1}	Temporary Register Reads
1.7×10^{-1}	Local Load/Store Instructions
1.5×10^{-1}	Barriers Hit
1.2×10^{-1}	Divergent Warps
0.9×10^{-1}	Pages Touched
0.5×10^{-1}	Number of Workgroups
0.3×10^{-1}	Divergence Percentage
0.2×10^{-1}	Number of Control Registers Read
0.2×10^{-1}	Number of Interrupts
-0.01×10^{-1}	Number of Control Registers Written

Table 5.4: Correlation of each attribute sourced from functional simulation with measured runtime.

still too high to make accurate performance predictions for GPUs. Furthermore, this model provides a performance prediction for a single GPU, without the possibility of exploring the design space.

There is no single method for identifying how much data is required for a machine learning algorithm to be effective, without first having the data. We can however, look at existing data sets which have been used in machine learning. One notable example is the 2006 Netflix competition [140], where researchers were competing to design the best collaborative filtering algorithm that would predict user ratings for films. Netflix released a dataset with over 100 million data points. A prize wasn't awarded until 2009 [141]. While datasets used in machine learning can be smaller, predicting GPU performance is a non-linear task with multiple unknown variables, likely requiring a far larger dataset than we have collected. We identify two main reasons for which our machine learning model does not perform as well as we would like - the quantity and quality of the data, and the level of detail modelled by the functional simulator.

In order to overcome the difficulties of collecting a large number of OPENCL kernels, we opted to use DeepSmith, which can automatically generate random collections

Inputs	LR Prediction Error	ANN Prediction Error
Naïve Inputs	706	567
Feature Selection - Corr.	1398	333
Feature Selection - PCA	2296	584
Manually adjust data	452	229
Manually adjust data + Feature Selection - Corr.	2969	335
Manually adjust data + Feature Selection - PCA	3355	634

Table 5.5: Error of Linear Regression and ANN models with different approaches to feature selection. Error measured using MAPE.

of kernels. At first, this appeared to be a promising approach to quickly generate large amounts of data, however on closer examination, we realized that the kernels generated by DeepSmith are not necessarily representative of real OPENCL kernels. For example, the kernels could often perform unnecessary computation, which wouldn't be used in the output, and would be optimized away by the compiler. In addition, many of the generated OPENCL kernels are similar to each other, which creates a strong bias for a specific type of kernel in a machine learning model. Finally, many of the generated kernels are short, executing just a few instructions. This not only creates a bias for the machine learning model, but also makes it difficult to discern between kernels, as there is a lot of noise created by startup and dispatch of the kernel.

We observe that manually modifying the feature set to consider the GPU's parallel hardware capacity significantly reduces the prediction error. However, this is the limit to what we can do with functional features. In reality, the effects of warp scheduling and cache effects, which are not captured by our model, will have a dramatic impact on the end-to-end latency of the execution. In order to capture this, we develop a trace-based approach to GPU performance modelling, described in the following section.

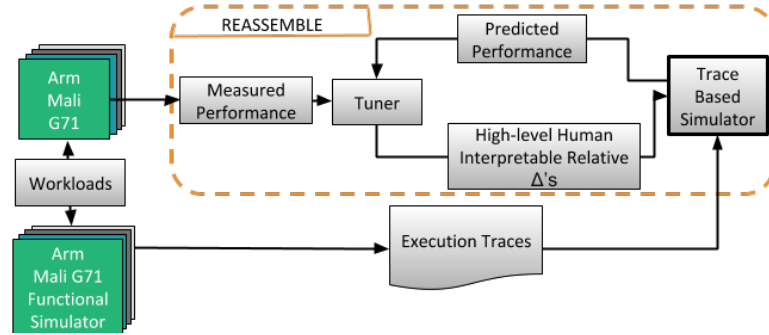
5.3 REASSEMBLE: A Trace Based Approach to Fast Performance Modelling

In this section, we propose a multi-phased approach to simulation, with a fast, full-system, functional simulation backed by an offline trace-based approach. By consciously limiting the detail of both the functional and trace based models, and tuning unknown parameters against a ground truth, we are able to make performance predictions that correlate strongly with real results, and are characterized by near-perfect rank correlation - both valuable metrics for early DSE. We present REASSEMBLE, a trace-

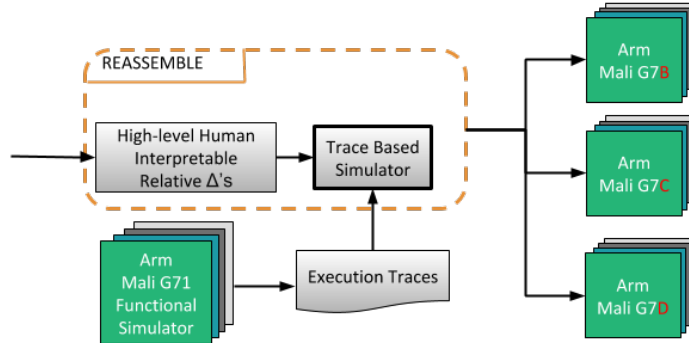
based approach with automatic parameter tuning achieving an average performance two orders of magnitude faster than existing cycle-accurate simulators. Furthermore, we demonstrate its usefulness in program optimization and iterative hardware design.

The remainder of this section is structured as follows: Section 5.3.1 describes the design of our system, followed by an extensive evaluation in Section 5.3.2, comparison against state-of-the-art in Section 5.3.3, a comprehensive design space exploration in Section 5.3.4, a critical evaluation of the presented work in Section 5.3.5. In Section 5.3.6, we compare our work against recent developments in the field, and conclude in Section 5.3.7.

5.3.1 Design and Methodology



(a) A performance model is automatically tuned for the root member of a GPU architecture family, here the Mali Bifrost G71 GPU.



(b) The tuned model from the first phase can be used to estimate the performance of GPU variants, e.g. Mali Bifrost G72 by changing the human-interpretable high-level parameters(Δ s).

Fig. 5.2: REASSEMBLE's tuning and deployment phases.

Our simulation framework takes a functional first approach, where the user executes the target program in a full-system, functional GPU simulator [136]. During ex-

ecution, the simulator generates traces, which are then fed into an offline trace-based simulator. The trace-based simulator parses the traces, simulates the execution, and provides a performance prediction. Unknown micro-architectural parameters are tuned using a set of 60 different kernel and input combinations listed in Table 5.8, similarly to tuning approaches used in [142, 143, 143, 144].

5.3.1.1 Functional Simulation

In order to explore the performance of a real device, with a complete software stack executing on it, we extend our existing full-system, functional GPU simulator [136], capable of executing unmodified binaries, with trace generation capabilities. Functional simulation only needs to be executed once for each program - it validates the functional correctness of the program, and generates the execution trace. This trace can later be re-used across multiple architectural configurations in the performance model. We use a similar trace format to [145]. This trace differentiates between memory, barrier, and arithmetic instructions, captures the PC, memory addresses of reads and writes, and the associated thread id. It does not however differentiate between different arithmetic instructions, as most of these will take the same number of cycles to complete. The trace also does not capture which core the thread is executing on, as this is a parameter that will be captured and applied when processing the trace.

5.3.1.2 Trace Based Simulator

The trace based simulator processes the trace in accordance with an architectural view of the GPU, as shown in Figure 5.2b. Critical components of the architecturally visible micro-architecture are modelled, for example branch divergence and memory coalescing - however, in order to maintain a fast simulation rate, only the micro-architectural details that have the most impact, and are common among the majority of GPUs are selected. Caches are approximated using a re-use distance model [146], and warp scheduling is implemented using the first-available method. A list of the most significant features impacting performance and details of their implementations in the simulator can be found in Table 5.6. For the baseline model, the remaining micro-architectural details are tuned to a hardware platform, depicted in Figure 5.2a. While these details are not individually modelled, their effects are compounded into metrics relating to arithmetic and memory latencies. The tuner and trace-driven simulator are completely decoupled from each other, meaning that the baseline micro-architectural parameters can also be provided to the trace-driven simulator directly by the user. The

traces can be piped directly from the functional simulator to the trace-driven simulator in order to provide a seamless simulation environment.

5.3.1.3 Tuning Unknown Parameters

Details of the majority of micro-architectural parameters and features in mobile GPUs are never released by the vendor. Furthermore, debug tools containing access to hardware counters are not readily available to the public. Specifically designed microbenchmarks also do not provide the answers that we need, as is shown in subsection 5.3.5. These problems pose significant challenges when designing accurate simulators. However, we find that these details can be successfully aggregated into latency ratios between the arithmetic pipeline, components of the memory hierarchy, and the remaining known parameters. We approach this problem by tuning unknown parameters against existing hardware as the ground truth, where known parameters are taken directly from vendor specifications.

Our design flow, depicted in Figure 5.2a is as follows. First we obtain results from execution in hardware, which we use as our ground truth. We then configure our trace based simulator with publicly available parameters, i.e. the cache organization, number of cores, number of execution engines. The remaining parameters - arithmetic latency, L1 cache hit latency, L1 cache miss latency, and L2 cache miss latency, are unknown for two reasons. First, as already mentioned, micro-architectural details are not publicly available, and second, because rather than representing an absolute cycle count, these numbers should be thought of as weights, which impact the model through their relative ratios. These unknown parameters are tuned by iteratively executing the traces through the trace based simulator, updating the parameters on each iteration. Where the user has expert knowledge, initial values or ranges can be provided as a starting point to reduce the tuning space. The parameters are updated using binary search, using results from the trace-based simulator as inputs. We tune one parameter at a time in order to reduce the search space, starting with the parameters with the largest range. Once all parameters have converged, the entire flow is executed again, in order to tune all parameters relative to each other.

Initial likely parameter ranges are provided by the user. We use benchmark subsetting [147] to both accelerate the training phase, and to avoid over-fitting the model to the benchmarks.

Once all parameters have converged, the tuned model can be used for software optimization on the tuned target, or new variants of the GPU design can be explored,

Feature	Implementation	Flexibility
Warp Size	Publicly Available	Variable
Core Count		
Execution Engine Count		
Cache Organization		
Arithmetic Throughput (Ratio)	Learned	
Memory Latency (Ratio)		
Cache Latencies (Ratios)		
Branch Divergence	Implemented in Trace Based Simulator	Fixed
Memory Coalescing		
Cache Replacement Policy	Approximated Using Re-use Distance	
Scheduling	Approximated with First Available	

Table 5.6: Main GPU features implemented, the sources for their configurations and their flexibility within REASSEMBLE.

Detail of Simulator	MAPE
Arithmetic, Memory, Barrier Operations, Warp Divergence	74
+ L1 Re-use Distance Based Cache Model	53
+ L2 Re-use Distance Based Cache Model	44
+ Memory Coalescing	39

Table 5.7: Incrementally adding new features to REASSEMBLE’s trace-driven simulator helps us understand the relative benefits of modelling each additional implemented detail.

by changing any of the variable parameters, as depicted in 5.2b.

5.3.2 Validation

In this section, we demonstrate our validation efforts. First, we evaluate our approach against existing hardware platforms by tuning our model to the HIKEY-960 development board with a MALI-G71 GPU. Then, we predict the performance of a HIKEY-970 development board with a MALI-G72 GPU and an ODROID-C4 development board with a MALI-G31 GPU using the tuned model. We continue with a software optimization case study and present performance results.

5.3.2.1 Validating against hardware

We first validate our approach against the HIKEY-960 development board. To develop the model, we follow the steps outlined in Figure 5.2a.

We start with the tuning phase by executing a set of benchmarks, listed in Table 5.8, on the HIKEY-960 development board in order to obtain a hardware reference.

Next, we execute the exact same set of benchmarks in the full-system, functional GPU simulator, which generates traces during functional execution. We feed the

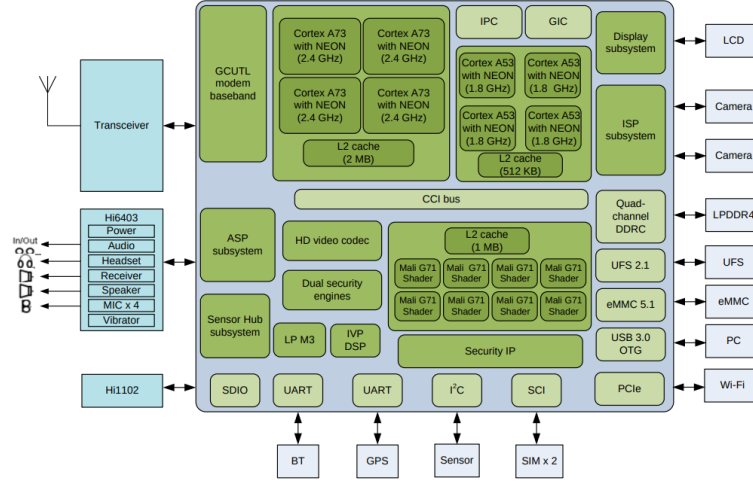


Fig. 5.3: Hikey-960 Block Diagram. Source: [148]

traces through our trace-based simulator with fixed known architectural parameters and ranges of potential values for the unknown parameters, which we call Δs . While we do not explicitly know any of the parameters, we are able to assume the role of the expert through comparisons to CPU technology. In addition to the GPU, the HIKEY-960 contains four A53 CPU cores, and four A73 CPU cores, as shown in Figure 5.3. We assume that CPUs and GPUs packaged together in the same System-on-Chip (SoC) will be implemented using similar technology, and therefore will implement similar micro-architectural parameters. Furthermore, CPUs and the GPUs sharing the same memory will experience the same memory access latency. Micro-architectural benchmarks for the A53 CPU (using 7-zip¹ and sbc-bench²) revealed latencies of 3ns, 15ns, and 143ns for the L1 Cache, L2 Cache, and memory, respectively, which we use as starting parameters for the tuner.

We measure error using MAPE, as shown in Equation 5.3, where A_t represents the actual value, and F_t the predicted value.

$$M = \frac{1}{n} \sum_{t=1}^n \left| \frac{A_t - F_t}{A_t} \right| \quad (5.3)$$

The best tuned configurations for minimizing error, maximizing correlation, and maximizing rank correlation with the HIKEY-960 are listed in Table 5.9. The best configuration exhibits near perfect correlation and rank correlation.

Using the model tuned against the HIKEY-960 with the MALI-G71 GPU, we enter the deployment phase and make predictions for the HIKEY-970 with the MALI-G72

¹7-cpu.com/cpu/Cortex-A53.html

²<https://github.com/ThomasKaiser/sbc-bench>

GPU and the ODROID-C4 with the MALI-G31 GPU. Once again, we don’t explicitly know the implementation details of the GPUs, so we rely on micro-benchmarks to provide parameters into our model.

sbc-bench reveals that the ODROID-C4 with the G31 has a significantly improved memory bandwidth (2.23x) and memory latency (0.6x) over the HIKEY-960. We use these parameters as scaling factors for memory-related latencies in our model. We further make the assumption that an iteration of G71 would have an improved arithmetic latency, and so we select 0.8 as a scaling factor. We use the same methodology to derive the parameters for the HIKEY-970 with the G72. The configuration parameters are presented in Table 5.9, on which the model achieves 45.27% error for the G31 and 43.01% for the G72.

Suite	Benchmark	Kernels
Polybench-ACC	2mm	mm2_kernel_1, mm2_kernel_1
	3mm	mm3_kernel1
	atax	atax_kernel1, atax_kernel2
	correlation	mean_kernel, std_kernel, reduce_kernel
	covariance	reduce_kernel
	gemm	gemm
	gemver	gemver_kernel_1, gemver_kernel_2, gemver_kernel_3
	gesummv	gesummv
Rodinia	backprop	bpnn_adjust_weights_ocl
AMD APP SDK 2.5	BlackScholes	–
	MemoryOptimizations	copy1Dfloat4, copy2Dfloat, copy2Dfloat4, NoCoal, Split
	MersenneTwister	MersenneTwister
	SimpleConvolution	SimpleConvolution
clBLAS (Tutorial [48])	GEMM1	GEMM1

Table 5.8: Kernels used in testing and tuning the performance model. We use multiple inputs and shapes where the kernels allow for it.

Origin	Arith. Latency	Store Latency	L1H	L1M	L2M	MAPE	Correlation	Rank Corr.
Micro-benchmark (G71)	2	4	3	15	143	66.48	0.96	0.88
Tuned (G71)	1	1	1	61	123	39.19	0.73	0.85
	1	1	1	10	325	204.31	0.92	0.83
	1	1	1	122	123	50.81	0.61	0.86
Predicted (G72)	1	1	1	42.0	86.0	43.01	0.82	0.89
Predicted (G31)	0.8	0.44	0.44	26.84	54.12	45.27	0.87	0.93

Table 5.9: Parameters for the MALI-G71 are tuned using micro-benchmark results as a starting point. The tuned model is then used to predict the performance of the MALI-G31 and MALI-G72 with good accuracy.

5.3.2.2 Program Modification

REASSEMBLE can effectively be used for program optimization for new targets before silicon is available, offering not only integration with a full-system simulation framework indistinguishable from hardware, but also a faster turnaround time than cycle-accurate simulation. Figure 5.4 demonstrates the effectiveness of using rank correlation to evaluate program behaviour. Ten different source code changes are applied to a baseline GEMM program, originating from the Polybench benchmark suite. When ranking these programs by estimated cycle count, nine cases are within one of the measured rank, and one is within two. The only significant outlier is modified version H, which both changes the default data-type from float to float4, and increases the workgroup size. On its own, changing data from float to float4 (version F) significantly increases the runtime, due to the increased amount of memory read and written. Increasing the workgroup size provides the scheduler for each core a larger number of warps to schedule. This is helpful in hiding the latency of memory operations, by scheduling ready warps during memory stalls. Conversely, a larger number of warps simultaneously reading wide data structures from memory increases memory contention, possibly up to the point of the Shader Cores stalling. This means that version H exhibits both positive (increasing workgroup size) and negative (float4 datatype) interference. The predicted time for H is 71112ns, while the actual measured time is 93322ns, meaning the error is just 23% - less than the average error of our model. The main reason H is a significant outlier in the rank estimation is due to a tightly packed space, where the measured runtimes of A,B,D,H,I,J are all within 15% of each other. However, we still see that the general trend is correct. Further improvements could be made to the model in order to reduce this error, for example, training the model with additional data, adding new micro-architectural features, or explicitly modelling memory bandwidth.

5.3.3 Comparison Against State-of-the-Art

Now, we compare our simulator against existing state-of-the-art simulators, first cycle-accurate simulators, and then against fast performance modelling techniques.

Despite the fact that our simulator is only prototyped in pure Python, with no additional acceleration, there is a clear performance benefit over existing detailed GPU simulators. As an interpreted language, Python is slower than compiled languages such as C++ or Go, and we are confident that an optimized compiled implementation

Version	Description	Measured Cycles	Measured Rank	Predicted Rank	Diff.
A	Baseline GEMM	86 515	2	2	0
B	Add arithmetic instructions	92 618	4	5	1
C	Barriers in inner loop	138 535	8	8	0
D	Barriers in outer loop	90 663	3	4	1
E	Poor workgroup shape $65 \times 65 \times 5 \times 5$	190 587	9	9	0
F	Change data to float4	263 513	10	10	0
G	Add large # of arithmetic instructions	1 374 746	11	11	0
H	float4 data, larger workgroup size	98 891	6	1	-5
I	Change data to int	85 984	1	2	1
J	Barrier in outer loop, small workgroups	93 332	5	7	2
K	Introduce warp divergence	124 274	7	6	-1

Table 5.10: Performance, rank, and predicted rank for 11 variations of a GEMM kernel. Predicted rank using REASSEMBLE closely matches actual. Visualization provided in Figure 5.4.

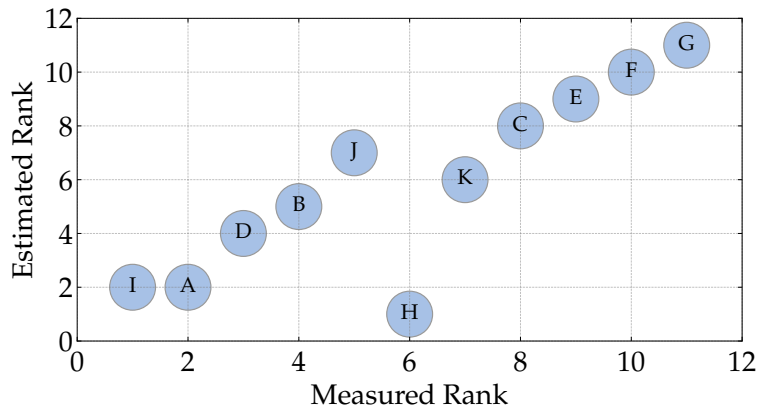


Fig. 5.4: Measured vs. predicted rank on variations of GEMM. Details in 5.10

will provide significant further benefits. A performance comparison against existing simulators is presented in Figure 5.5.

Similarly to a cycle-accurate simulator, our trace-based model can be modified in order to add additional features, while still relying on the tuned performance model for the original hardware. In the absence of hardware, tuning can be performed using a detailed, existing simulator. While this is an added development cost, the performance benefits of REASSEMBLE are clear, and the cost would be quickly recouped.

Accel-Sim is most similar to our approach, presented in Chapter 5, however there are notable differences. Accel-Sim supports Nvidia GPUs, while we model embedded, mobile GPUs and provide a tracing plugin for an Arm MALI GPU simulator. Traces for Accel-Sim are generated using a binary instrumentation tool called NVBit [97], which only supports Nvidia GPUs, and requires a physical Nvidia GPU to work. Our simulator on the other hand uses an existing functional GPU simulator with an interchangeable software stack and functional model, enabling us to explore not only

new hardware configurations, but also couple them with new software, and use the trace based simulator for optimizing code using new software stacks. Furthermore, we target a different use case than Accel-Sim - while the detail of Accel-Sim allows for detailed micro-architectural exploration, we trade detail off in exchange for faster simulation turnaround time. For example, using our model we can quickly explore the effects of changing the warp size or number of cores, however we have no insight into performance of registers or interface bottlenecks.

Analytical models provide another performance modelling alternative to cycle-accurate simulation. Existing solutions were discussed in Section 3.4.3. Amongst these, MDM [104], which takes an interval modelling approach is currently considered state-of-the-art.

MDM shares many similarities with our own approach, as it also relies on traces collected using either functional simulation or hardware in the first phase. In the second phase however, MDM uses a nearest neighbour algorithm to select a representative warp. While we acknowledge that this approach can significantly speed up the performance prediction, we are also concerned that selecting a representative warp is only valid for regular applications. Irregular applications, which can execute many different branches of code, will not have a representative warp, and as such, we choose to faithfully model each warp.

In its third phase, MDM runs the traces through a cache simulator, and in its fourth and final phase, MDM predicts the performance using an analytical model. In our case, we replace the cache simulator with a re-use distance model, which provides significant performance benefits over detailed cache modelling. The re-use distance model is a component of our trace-based simulation, meaning each trace is only processed once per simulation.

The configurable parameters of our simulator also differ from MDM. Our focus on high-level parameters such as core counts, number of execution engines per core, and warp size, as well as micro-architectural Δ parameters is driven by real-world observation of the GPU design space, as presented in Table 5.1. As MDM provides a more detailed view of the memory, we envision using MDM to explicitly represent some of our Δ parameters in future work.

The MDM model has not been publicly released, and it models a different architecture to our own, therefore we were not able to directly compare it against REASSEMBLE. However, from published literature we know that MDM provides a 65x speedup over detailed simulation on a single execution, and a 6371x speedup when simulat-

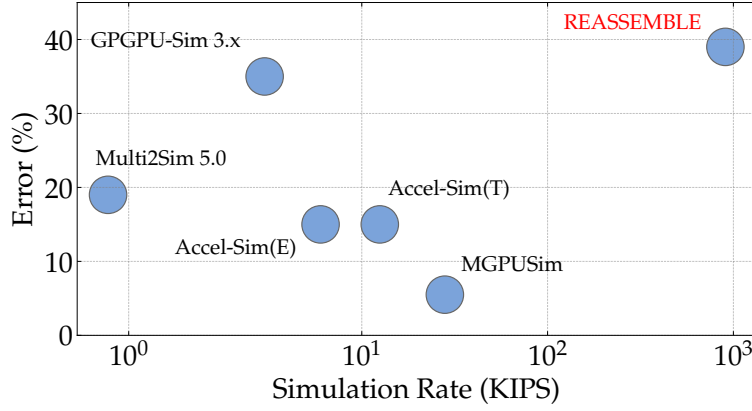


Fig. 5.5: REASSEMBLE provides two orders of magnitude improvement on simulation rate over detailed simulators with an accuracy comparable to GPGPUSim-3.x. Average error for REASSEMBLE was calculated using MAPE. MAPE for competing simulators were extracted from [83]. Additionally, gem5-APU with HSAIL and GCN3 exhibits 75% and 42% error [81], respectively, however the publications do not provide a simulation rate.

ing 1000 configurations, due to its low recurrent cost. REASSEMBLE falls within this range, with 2-3 orders of magnitude improvement over existing cycle-accurate simulators. MDM reports a 40% prediction error, which again, is on-par with our approach.

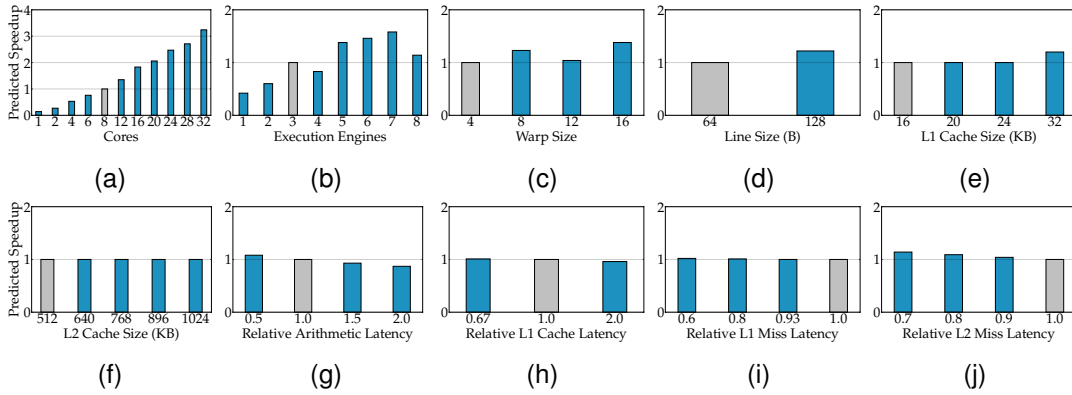


Fig. 5.6: MALI architecture characterization using REASSEMBLE. Gray shows measured baseline, blue shows predicted variants. In isolation, increasing core counts is most beneficial, while increasing execution engines and warp size are only beneficial in certain cases. Reducing memory latency is more beneficial than optimizing the arithmetic pipeline.

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G72	<1	<1	<1	<1	1.51*
1	1	1	1	1	1.35
2	1	0.67	1	0.9	1.41
3	1	0.67	0.8	1	1.37
4	1	0.67	0.8	0.8	1.48
5	1	0.5	1	0.8	1.5
6	1	0.5	0.87	0.8	1.51
7	1	0.5	0.8	0.8	1.53
8	0.5	1	1	1	1.45
9	0.5	1	0.87	1	1.46
10	0.5	0.5	0.8	1	1.5
11	0.5	0.5	0.8	0.9	1.55
12	0.4	0.67	0.87	0.8	1.61
13	0.4	0.67	0.8	0.72	1.69
14	0.3	0.67	0.67	0.72	1.74
15	0.3	0.4	0.8	0.8	1.65

Table 5.11: 12 SCs, 3 EEs, Warp Size 4, Cache Line Size 64

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G72	<1	<1	<1	<1	1.51*
16	1	1	1	1	1.61
17	1	0.67	1	0.9	1.68
18	1	0.67	0.8	1	1.64
19	1	0.67	0.8	0.8	1.75
20	1	0.5	1	0.8	1.74
21	1	0.5	0.87	0.8	1.76
22	1	0.5	0.8	0.8	1.77
23	0.5	1	1	1	1.73
24	0.5	1	0.87	1	1.76
25	0.5	0.5	0.8	1	1.81
26	0.5	0.5	0.8	0.9	1.87
27	0.4	0.67	0.87	0.8	1.95
28	0.4	0.67	0.8	0.72	2.04
29	0.3	0.67	0.67	0.72	2.09
30	0.3	0.4	0.8	0.8	2.02

Table 5.12: 12 SCs, 3 EEs, Warp Size 4, Cache Line Size 128

5.3.4 Design Space Exploration

In this section, we demonstrate the flexibility of our simulator in exploring different potential configurations for next generation GPUs, anchored against our trained MALI-G71 model, and identify configurations that offer better predicted performance to area trade-offs. Unless otherwise defined, Δ parameters in this section are presented as fractions relative to the baseline MALI-G71, whose Δ s are all 1.0.

5.3.4.1 Mali Architecture Characterization

We first consider each exposed component in isolation from any other changes. All explored configurations are presented in Figure 5.6 as variants of the MALI-G71 GPU, which we extensively refer to in this section. The data presented will be benchmark

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G72	<1	<1	<1	<1	1.51*
31	1	1	1	1	1.38
32	1	0.67	1	0.9	1.48
33	1	0.67	0.8	1	1.43
34	1	0.67	0.8	0.8	1.57
35	1	0.5	1	0.8	1.56
36	1	0.5	0.87	0.8	1.58
37	1	0.5	0.8	0.9	1.51
38	1	0.5	0.8	0.8	1.59
39	0.5	1	1	1	1.46
40	0.5	1	0.87	1	1.47
41	0.5	0.5	0.8	1	1.51
42	0.5	0.5	0.8	0.9	1.58
43	0.4	0.67	0.87	0.8	1.62
44	0.4	0.67	0.8	0.72	1.69
45	0.3	0.67	0.67	0.72	1.78

Table 5.13: 8 SCs, 5 EEs, Warp Size 4, Cache Line Size 64

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G72	<1	<1	<1	<1	1.51*
46	1	1	1	1	1.23
47	1	0.67	1	0.9	1.29
48	1	0.67	0.8	1	1.25
49	1	0.67	0.8	0.8	1.37
50	1	0.5	1	0.8	1.38
51	1	0.5	0.87	0.8	1.39
52	1	0.5	0.8	0.9	1.34
53	1	0.5	0.8	0.8	1.4
54	0.5	1	1	1	1.27
55	0.5	1	0.87	1	1.3
56	0.5	0.5	0.8	1	1.34
57	0.5	0.5	0.8	0.9	1.42
58	0.4	0.67	0.87	0.8	1.41
59	0.4	0.67	0.8	0.72	1.47
60	0.3	0.67	0.67	0.72	1.53

* value measured, not predicted

Table 5.14: 8 SCs, 3 EEs, Warp Size 8, Cache Line Size 64

dependent, however we have chosen a representative sample of workloads, and we are interested in the average performance improvement. The data presented in the tables is the average of all examined benchmarks.

Shader Cores – Selecting the number of cores is a customization exposed to Arm’s customers. Currently, variants of the G71 containing 8 and 20 cores are in existence. In our simulator, we see a clear performance benefit while increasing the number of cores (Figure 5.6a), and a drop in performance when decreasing the number of cores. However, the returns are diminishing. Adding 8 cores to the baseline system offers only $1.83\times$ speedup while decreasing the number of cores down to one drops the performance to 0.15 of the baseline ($7\times$ slowdown).

Execution Engines – Scaling the number of execution engines (Figure 5.6b) shows

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G31	<1	<1	<1	<1	0.25*
61	1.0	1.0	1.0	0.8	0.18
62	1.0	1.0	1.0	0.6	0.2
63	1.0	1.0	1.0	0.4	0.23
64	1.0	1.0	0.8	0.6	0.2
65	1.0	1.0	0.8	0.4	0.23
66	1.0	0.67	1.0	0.6	0.2
67	1.0	0.67	1.0	0.4	0.23
68	1.0	0.67	0.8	0.8	0.18
69	1.0	0.67	0.8	0.4	0.23
70	0.5	1.0	1.0	0.8	0.19
71	0.5	1.0	1.0	0.6	0.22
72	0.5	1.0	0.8	0.8	0.19
73	0.5	1.0	0.8	0.4	0.25
74	0.5	0.67	1.0	0.6	0.22
75	0.5	0.67	1.0	0.4	0.25

Table 5.15: 2 SCs, 2 EEs, Warp Size 4, Cache Line Size 64

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G31	<1	<1	<1	<1	0.25*
76	1.0	1.0	1.0	0.6	0.24
77	1.0	1.0	1.0	0.4	0.27
78	1.0	1.0	0.8	0.6	0.24
79	1.0	0.67	1.0	0.8	0.22
80	1.0	0.67	1.0	0.6	0.24
81	1.0	0.67	0.8	0.8	0.22
82	1.0	0.67	0.8	0.6	0.24
83	0.5	1.0	1.0	0.8	0.24
84	0.5	1.0	1.0	0.6	0.26
85	0.5	1.0	0.8	0.8	0.24
86	0.5	1.0	0.8	0.4	0.3
87	0.5	0.67	1.0	0.8	0.24
88	0.5	0.67	1.0	0.6	0.27
89	0.5	0.67	0.8	0.8	0.24
90	0.5	0.67	0.8	0.6	0.27

Table 5.16: 2 SCs, 2 EEs, Warp Size 4, Cache Line Size 128

a mixed result on the other hand. There is in fact a decrease in performance when selecting four execution engines, despite this providing additional compute capacity over the default three. Scaling the number of execution engines is a valid concern for Arm, and scaling down can be observed in the efficiency-oriented MALI-G31, with two execution engines, and MALI-G51, with two different shader cores - one containing a single execution engine and the other containing three.

Warp Size – Scaling the warp size (Figure 5.6c) is similarly affected to scaling execution engines. While it adds further compute capacity, there is varied benefit. For example, changing the warp size to 8 provides $1.23\times$ speedup, but increasing further to a warp size of 12 results in only $1.04\times$ speedup relative to the baseline. This is due to the fact that in these variants we do not increase the L1 cache size, while increasing compute significantly, increasing pressure on the L1 cache, and reducing the impact of

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G31	<1	<1	<1	<1	0.25*
91	1.0	1.0	1.0	0.8	0.23
92	1.0	1.0	1.0	0.4	0.29
93	1.0	1.0	0.8	0.6	0.26
94	1.0	0.67	1.0	0.8	0.23
95	1.0	0.67	1.0	0.6	0.26
96	1.0	0.67	1.0	0.4	0.29
97	1.0	0.67	0.8	0.6	0.26
98	0.5	1.0	1.0	0.8	0.25
99	0.5	1.0	1.0	0.4	0.32
100	0.5	1.0	0.8	0.6	0.28
101	0.5	1.0	0.8	0.4	0.32
102	0.5	0.67	1.0	0.8	0.25
103	0.5	0.67	1.0	0.4	0.33
104	0.5	0.67	0.8	0.6	0.29
105	0.5	0.67	0.8	0.4	0.33

Table 5.17: 4 SCs, 1 EE, Warp Size 4, Cache Line Size 64

GPU Variant	Relative Arithmetic Δ	Relative L1C Hit Δ	Relative L1C Miss Δ	Relative L2C Miss Δ	Predicted Speedup
G71	1	1	1	1	–
G31	<1	<1	<1	<1	0.25*
106	1.0	1.0	1.0	0.8	0.14
107	1.0	1.0	1.0	0.6	0.16
108	1.0	1.0	0.8	0.8	0.15
109	1.0	1.0	0.8	0.6	0.17
110	1.0	1.0	0.8	0.4	0.19
111	1.0	0.67	1.0	0.6	0.17
112	1.0	0.67	1.0	0.4	0.2
113	1.0	0.67	0.8	0.6	0.17
114	1.0	0.67	0.8	0.4	0.2
115	0.5	1.0	1.0	0.6	0.18
116	0.5	1.0	1.0	0.4	0.21
117	0.5	1.0	0.8	0.6	0.18
118	0.5	0.67	1.0	0.6	0.18
119	0.5	0.67	1.0	0.4	0.21
120	0.5	0.67	0.8	0.6	0.18

* value measured, not predicted

Table 5.18: 2 SCs, 1 EE, Warp Size 8, Cache Line Size 64

increasing the warp size. Scaling the warp size has been observed in the MALI-G76 and G52 to eight threads, and again in the new VALHALL architecture, with 16 threads executing in a warp. The configurations are presented in Table 5.1.

Caches – Scaling both the L1 (Figure 5.6e) and L2 (Figure 5.6f) cache sizes in isolation, on average, provides no benefit over the baseline cache size. However, we note with reference to Table 5.1 that L1 cache sizes have remained consistent across all instances of the MALI architecture, with the exception of the energy-optimized MALI-G31, and the range for available cache sizes only tends to change every couple of iterations, as seen in Table 5.1. Furthermore, inspecting the cache sizes on the G71-MP8 and G71-MP12 using CLINFO, a standard OpenCL-based tool, we can see that L2 cache sizes are fixed to 512K in both instances. Increasing the cache line

size (Figure 5.6d) however, from 64 bytes to 128 bytes in both the L1 and L2 caches provides a speedup of $1.22\times$, particularly benefiting applications which operate on wide data structures.

Δs – The MALI GPU is far less sensitive to changes in the relative Δs , which relate to improvements in the micro-architecture, or changes in the clock frequency. Arithmetic throughput would have to be improved by 50% in exchange for just $1.08\times$ speedup (Figure 5.6g). Similarly, memory latency would need to be reduced by 30% for a similar $1.14\times$ speedup (Figure 5.6j).

5.3.4.2 Designing the MALI G72-MP12

This section presents GPU configurations discovered using REASSEMBLE, during the design of a G72-like GPU. In addition to micro-architectural optimizations to the G72, we also discover configurations that achieve similar or better performance implementing different architectural characteristics, and as such providing different performance to area trade-offs.

We consider likely combinations of architectural and micro-architectural parameters. For example, we know from the previous section that increasing the number of cores is effective, while increasing the number of execution engines or warp size is only beneficial in certain cases. Furthermore, we know that by modifying the Δs , we can model further micro-architectural improvements in addition to architectural changes.

For the purpose of this case study, we place ourselves back in time, in the early design stages of the MALI-G72. Our only design requirement is a next generation GPU with $1.5\times$ speedup, and we have the capacity for both architectural and micro-architectural improvements to the GPU. We carefully design a search space, and using our tuned model, we make small modifications to the variable parameters.

Due to limited information, we do not explicitly model area in our model, however of the three architectural parameters that we model - shader cores, execution engines, and warp, reducing the number of shader cores will have the largest area impact, as each core includes execution engines, which in turn contain the execution logic replicated for each thread in a warp. Additionally, shader cores contain L1 cache slices. An overview of the GPU can be found in Section 2.3.

We draw starting points for the design from Figure 5.6. A GPU with no modifications other than increasing the number of cores from 8 to 12, provides $1.35\times$ speedup over the G71-MP8. Scaling the number of execution engines per core from three to five provides similar speedup of $1.38\times$. Increasing the warp size from four to eight

provides $1.23\times$ speedup. Increasing the cache line size from 64 to 128 bytes provides $1.22\times$ speedup. Changes to the micro-architecture, described in our model as changes in relative Δ s can bridge the gap between the architectural changes to the baseline, and the target of $1.5\times$ speedup.

Tables 5.11, 5.12, 5.13, 5.14 show the explored configurations in the second phase of the DSE. For each selected configuration from Figure 5.6, we explore additional micro-architectural parameters. We scale the relative Δ s, by a fraction of the original configuration, and also explore various combinations of different relative Δ s.

12 Shader Cores – Table 5.11 shows a configuration with 12 shader cores. A number of variants fall close to the target of $1.5\times$. Variant 5 does so, by keeping the arithmetic and L1C Miss Δ s the same, but reducing the L1C Hit and L2C Miss Δ s to 0.5 and 0.8 respectively. Variant 10 arrives at a similar result, using a different configuration - by improving arithmetic, L1C Hit, and L1C Miss (0.5,0.5,0.8), but keeping the L2C Miss Δ constant at 1.0.

12 Shader Cores + 128 byte cache line – Table 5.12, which additionally modifies the cache line size, predicts far better performance than the target $1.5\times$.

Five Execution Engines Per Core– Table 5.13 shows a different approach, and instead of scaling the number of shader cores, the number of execution engines is increased. A number of variants fall close to the target of $1.5\times$, for example variant 37, which reduces the Δ s for L1C Hit, L1C Miss, and L2C Miss to 0.5, 0.8, and 0.9 respectively, in addition to using five execution engines per core. However, variant 36 also stands out here, as the micro-architectural configuration is identical to variant 6 in Table 5.13 - but provides a predicted $1.58x$ speedup instead of $1.51\times$. Rather than increasing the core count by 4 like variant 6, variant 36 instead increases the number of execution engines, in total adding 16. Even when discounting graphics specific logic, the shader core contains an L1 cache and significant additional logic - and therefore area. Accounting for this in addition to performance gains, configurations with five execution engines provide a more area-efficient design point.

Warp Size 8 – Table 5.14 increases the warp size in combination with multiple variations of the micro-architecture. The data here shows that making additional changes to the micro-architecture has a smaller impact when combined with increasing the warp size, than when changing other architectural parameters. Variant 60 falls close to the target improvement of $1.5\times$, however this requires significant improvements to the micro-architecture, with Δ s of 0.3, 0.67, 0.67, and 0.72 for Arithmetic, L1C Hit, L1C Miss, and L2C Miss respectively. However, this does not necessarily mean that in-

creasing the warp size is a bad idea - on the contrary, it may suggest that other changes to the architecture may be needed to fully benefit from a larger warp size.

5.3.4.3 Designing the MALI G31-MP2

This section presents GPU configurations discovered using REASSEMBLE, during the design of a G31-like GPU - focusing on the lower end of the market. We discover GPUs similar to the G31, however there are also many alternatives which closely match the performance of the G31, with potentially much lower area.

To lower the cost, such a GPU would require a reduced area at the price of a performance reduction. Reduced area can be achieved through reducing the number of cores or execution engines, as well as by reducing the cache sizes. As this is still an iteration of the G71, we also include micro-architectural improvements in our DSE, modelled through our Δ parameters. Our goal in this study is to explore options for reducing the area, while maximizing the performance. We draw starting points from Figure 5.6. We use a measured performance of 0.25 for the MALI-G31, as our performance target. Results are presented in Tables 5.15, 5.16, 5.17, 5.18.

Two Shader Cores – Reducing the number of cores and execution engines is an obvious starting point. Table 5.15 presents a configuration of two cores and two execution engines per core, with different variations of micro-architectural improvement. Of these, the best performance predicted as 0.25 of the MALI-G71, is achieved by variant 75, whose arithmetic, L1C Hit, and L2C Miss Δ s are 0.5, 0.67, and 0.4 respectively. The L1C Miss Δ remains constant at 1.

Two Shader Cores + 128 byte cache line – Increasing the line size from 64 to 128 provides consistent improvement of around 0.05 (Table 5.16), which has a significantly smaller impact than making the same change with 12 cores and 3 execution engines (Table 5.12). We expect that at the L1 level, large cache lines are more beneficial with a larger number of execution engines, and similarly, at the L2 level are more beneficial with a larger number of cores, as both of these imply more warps executing in parallel, and a higher chance of two warps accessing the same cache line.

Four Shader Cores + One Execution Engine Per Core – A GPU with four shader cores with one execution engine each will have a larger footprint than a GPU with two shader cores and two execution engines per core, due to each shader core containing and L1 cache. However, rather than being a poor configuration, it provides a different point in the design space, as the additional area is compensated by better performance. Variant 103 in Table 5.17 provides a performance of 0.33 and variant 75 in Table 5.15

provides a performance of 0.25, relative to the baseline. 103 is a four core, one execution engine configuration, and 75 contains two cores, two execution engines. The micro-architectural Δ s are the same across both.

Warp Size 8 – Table 5.18 presents results of our DSE over configurations containing two shader cores, one execution engine per core, and 8 threads per warp. In terms of area, this configuration will be the cheapest of all explored configurations. However, this results in performance loss, as the relative performance is significantly lower than for more expensive configurations.

Overall, variants 73 and 75 came closest to the G31 in terms of performance and area cost. However, configurations in Table 5.16 and (d) provide low area solutions which match, or nearly match the target performance, demonstrating scope for lower cost GPUs.

5.3.5 Critical Evaluation

In this section, we critically review our work, explaining some of the unexpected results, and contrasting against existing approaches.

5.3.5.1 Why aren't targeted microbenchmarks used to tune specific parameters of the model?

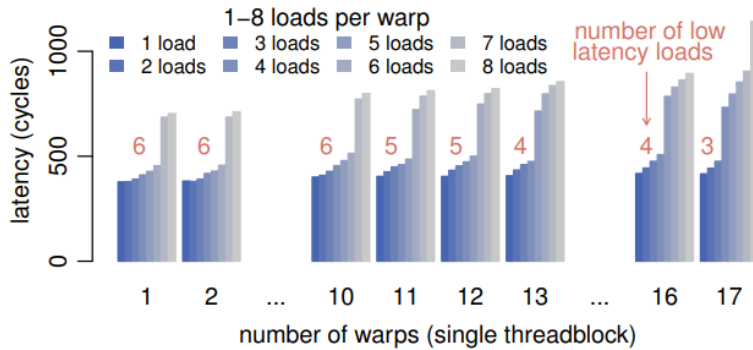


Fig. 5.7: Results of microbenchmark to identify number of MSHRs on Nvidia GTX470 GPU. Source: [146]

Targeted microbenchmarks are commonly used to identify architectural parameters in GPUs, as can be seen in [146] and [149], where the authors identify the number of miss status holding registers (MSHRs) on an Nvidia GTX470 GPU and an Nvidia Titan X GPU, respectively. MSHRs are registers that hold information about outstanding memory accesses, setting a constraint on how many accesses can be in-flight at

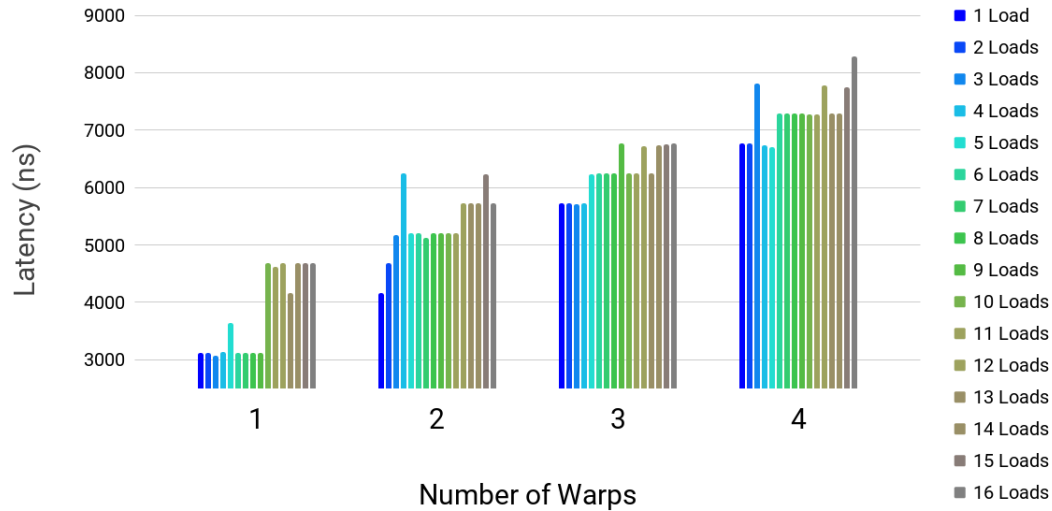


Fig. 5.8: Results of microbenchmark to identify number of MSHRs on Mali-G71 GPU.

Listing 5.1: OpenCL microbenchmark to identify number of MSHRs per Shader Core.

```

1  __kernel void mshr_mb(__global float *input, __global float *output)
2  {
3      int gid = get_global_id(0);
4      if (gid % 4 == 0) { // One thread in a warp requests memory
5          for (int i = 0; i < NUM_LOADS; i++) {
6              // Calculate load address to new cache line
7              output[gid] = input[16 * (gid + i * NUM_WARPS*4)];
8          }
9      }
10     return;
11 }
12

```

any particular moment. Each time memory is requested, an MSHR is filled in with details of the requested cache line. Once all of the MSHRs are full, no further memory requests can be made, and the GPU can only process non-memory instructions. If a memory instruction is encountered, the warp stalls, until an MSHR is freed.

The results of the study in [146] are presented in Figure 5.7. Each warp in the first 10 warps can make memory requests to 6 separate cache lines before stalling. Following this, warps 11 and 12 can request 5 cache lines in parallel, warps 13 through 16 can request 4, and so on. The authors interpret this as 64 MSHRs being available per core, with each warp being able to use 6, however as this information isn't public, they acknowledge that this behaviour can be the result of undocumented components of the GPU.

Similarly to Nvidia GPUs, there is no available information regarding the micro-architectural details of Mali GPUs. We attempt to identify the number of MSHRs in a Mali-G71 core in a similar way, by performing a configurable number of non-overlapping loads without dependences. The kernel we invoke is shown in Listing 5.1,

which we have translated directly from the CUDA kernel in [146]. In our experiments, we measure the latency for up to 16 loads per warp, allowing for potential increases in the number of MSHRs since the Nvidia GTX470 was released in 2010. Our results are presented in Figure 5.8, however they do not provide the same clarity that the microbenchmark executed on an Nvidia GPU does.

Firstly, we see that there is a lot of noise in the data, despite our best efforts to reduce it (details on reducing noise are in Section 2.7). As the number of warps increases, this error becomes more pronounced, partly obscuring the obvious steps that we see with the Nvidia data. Secondly, there is no specific trend to follow. With a single warp, there is a clear increase in latency after 9 loads. With two warps on the other hand, there are immediate increases in latency after each of the first four loads, before the latency drops again. Finally, after 11 loads, we see a permanent increase in latency, however this is still obscured by variation in the data. Increasing the number of warps to 3 and 4 shows further variation. From this data, it is not possible to identify the number of MSHRs.

We believe the significant noise in our measurements to be a product of the full-system environment that the Mali GPU operates in, as well as the lack of established performance monitoring and debug tools for embedded GPUs. First of all, our execution environment is significantly different to Nvidia's. While the Nvidia kernel presented in [146] accesses dedicated GPU memory, our kernel accesses global memory, which is shared with CPU, and is subject to system noise. We believe one component of the system noise may be cache coherency, however we leave this exploration as future work. Secondly, there is a significant difference in the availability and maturity of performance measuring and debugging tools between the two GPUs. The Nvidia benchmark was executed with CUDA, which exposes performance counters to the user, and allows reading hardware performance counters within the compute kernel source. Reads from the performance counter were inserted around the body of the kernel, discounting any overhead from kernel dispatch. We, on the other hand, use OpenCL, which only provides a global timer that measures end to end execution of the entire kernel. Even if OpenCL did provide additional measurement tools, hardware counters for the Mali GPU are not publicly available. This study demonstrates that using microbenchmarks is not a feasible method of determining micro-architectural parameters of Mali GPUs.

5.3.5.2 What causes the predicted performance drop when scaling Execution Engines and Warp Size in Figure 5.6?

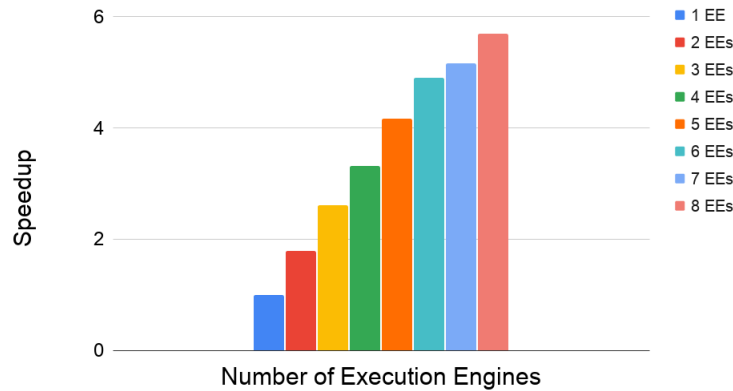


Fig. 5.9: Speedup as we increase the number of Execution Engines per core, and proportionally increase the size of the L1 and L2 caches.

Execution Engines - Increasing the number of EEs increases the compute capacity of the GPU. Each EE can process one warp at a time, so increasing the number of EEs from three to four allows four warps to execute simultaneously instead of three. However, this also means that additional memory requests can be issued by these warps, increasing the pressure on caches and memory, and the experiment presented in Figure 5.6 does not increase the cache sizes. We suspect that cache oversubscription causes the dips in performance at 4 and at 8 EEs per core. To confirm this, we repeat the experiment, while simultaneously increasing the size of the L1 and L2 cache, proportionally to the number of execution engines per core. The results of this experiment, presented in Figure 5.9, show that increasing the cache sizes along with the number of EEs no longer causes unexpected performance loss at 4 and 8 EEs.

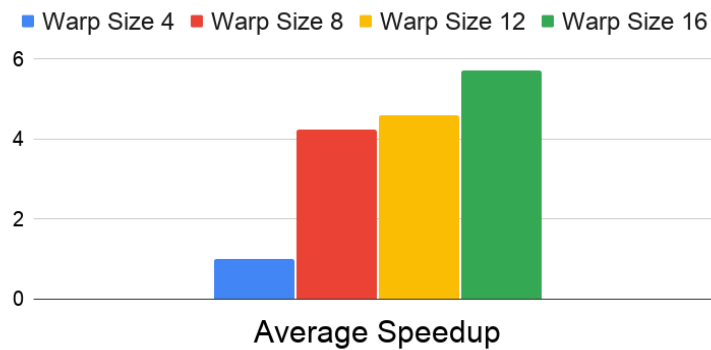


Fig. 5.10: Speedup following increasing the warp size, proportionally increasing the cache size, and increasing the cache line size to 96 bytes.

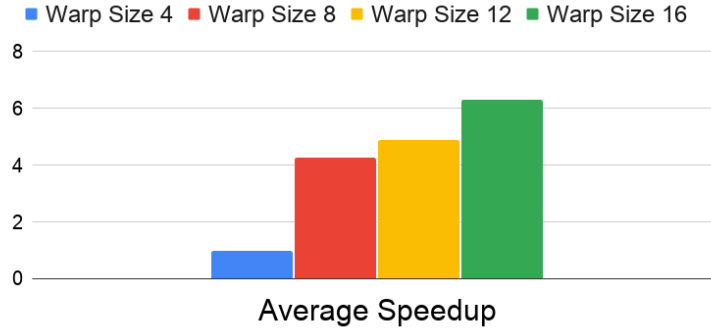


Fig. 5.11: Speedup following increasing the warp size, proportionally increasing the cache size, and increasing the cache line size to 128 bytes.

Warp Size - Increasing the warp size also increases the compute capacity of the GPU, however with different tradeoffs to increasing the number of EEs. Most importantly, all threads in a warp operate in lockstep, meaning that well balanced applications, in particular compute-bound applications, benefit from larger warp sizes, while divergent workloads can see severe performance reductions. Larger warp sizes can also benefit memory-bound applications, as memory requests originating from the same warp can be merged into a single memory request, provided that all threads are requesting the same cache line. This means that a warp with a warp with 4 threads will need to issue four times as many memory requests as a warp with 16 threads - provided that the cache line size is large enough. This in turn reduces bandwidth pressure on the lower level caches and main memory. Figure 5.10 and Figure 5.11 show average speedups across our simulated benchmarks, when increasing the warp size, proportionally increasing the L1 and L2 cache sizes, and increasing the cache line size to 96 bytes and 128 bytes, respectively.

5.3.5.3 How do we ensure that a trace-based GPU simulation can be accurate when hardware changes can impact the critical path and the trace itself, especially on irregular workloads?

Our simulation infrastructure is separated into two distinct phases. First, our fast, functional simulator collects execution traces for each thread, which store architecturally visible events. Next, these traces are fed through a trace-based performance model. As the traces only record architecturally visible information, they are completely unaware of any hardware changes, and our modelled hardware cannot modify their state. Organization of threads into warps and workgroups, as well as any scheduling are all performed in the second, trace-based phase of the simulation.

5.3.5.4 Would it be possible for the simulation to lead to a suboptimal design due to all the approximations used? If so, how can users detect/avoid this?

We do not claim that the simulator will always provide an optimal design, and in fact, we promote it as an early design space exploration tool, which can help identify the most promising and filter out the least promising architectural configurations. We presented an anticipated use case in Section 5.3.4, where we explored the design space for various iterations of the Mali-G71 GPU, and rather than selecting a single, optimal point, we selected the most promising candidates. In the next phase of the design exploration, we would anticipate using a more detailed simulator to investigate these points further, saving a significant amount of time by not needing to simulate all of the discarded points in the design space exploration in detail.

5.3.5.5 Where does the remaining error come from, and what can be done to improve it in future versions of REASSEMBLE?

We are selective in the hardware components that we model, focusing on the components that are most likely to impact performance. The remaining error likely comes from a variety of sources. As we do not have access to performance counters, we plot the absolute error of each benchmark against metrics obtained from the functional phase of our simulation in Figure 5.12.

We first look at the size and shape of the dispatched job, by inspecting the number of workgroups (Figure 5.12a) and the workgroup sizes (Figure 5.12b). We see little correlation however, between these features and the absolute error, as benchmarks with both small and large workgroup sizes, and small and large overall numbers of workgroups can exhibit error across the spectrum.

We turn our attention instead, to the types of instructions executed (Figure 5.12c). Once again, there is a lot of variation in the benchmarks that exhibit between 0 and 100% error. We note however, that there is very little variation in the benchmarks that exhibit more than 100% error. All of these benchmarks execute a very small number of instructions, compared to all other benchmarks. We do note however, that these benchmarks contain control flow instructions amongst the executed instructions. We first hypothesize, that given the presence of control flow instructions, undocumented effects of warp divergence may be responsible for the poor accuracy. However, when plotting the percentage of divergent warps against the absolute error (Figure 5.12d), we see that benchmarks with similar amounts of divergence can exhibit both low and

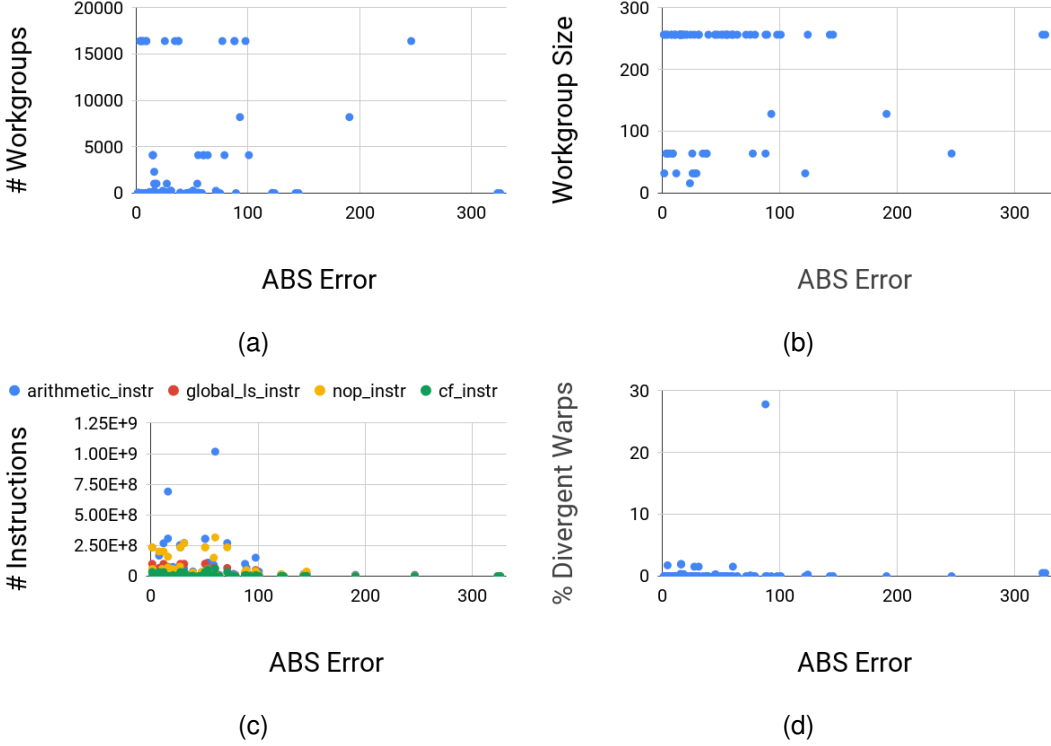


Fig. 5.12: We plot the absolute error of REASSEMBLE's performance prediction across our benchmarks, relative to features collected during the functional simulation phase.

high error. The presence of control flow instructions however, can also point us in a different direction. One component, which we do not model, but can have an impact on kernels with control flow, is the instruction cache.

To demonstrate the impacts of the instruction cache, we develop two microbenchmarks executing the same task of adding two vectors, but with different implementations. Both use a single thread, the first contains a number of addition operations sequentially in the source code, while the second adds the same number of elements in a loop. The results are presented in Figure 5.14, and show that the sequential version consistently takes 65% longer to execute than the loop version, in the examined range of 1000 to 1950 elements added.

Other components which we don't model include memory bandwidth (we assume no constraints) and dependency tracking. These, along with other features will be carefully considered along their performance/accuracy tradeoffs in future versions of REASSEMBLE.

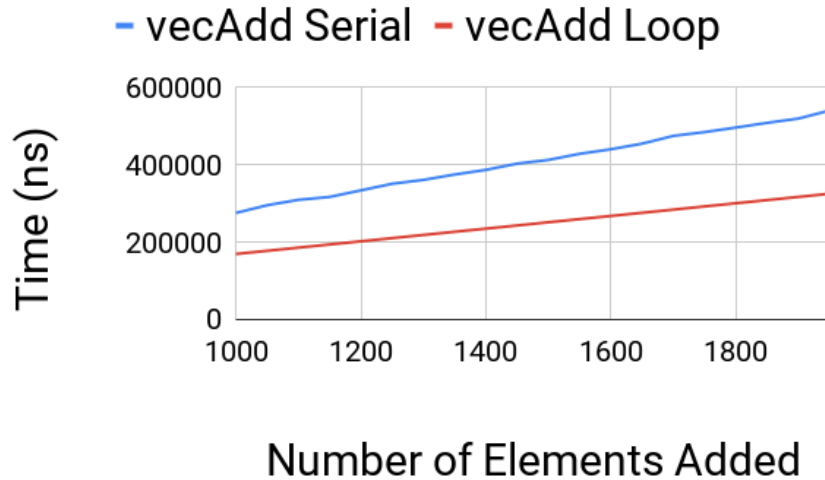


Fig. 5.14: We compare two different implementations of vector addition. Both use a single thread, the first contains a number of addition operations sequentially in the source code, while the second adds the same number of elements in a loop.

5.3.6 Recent Developments

This section discusses a recent industry-track publication from Nvidia, describing their efforts in designing NVArchSim, a fast, yet accurate GPU simulator - *Need for Speed: Experiences Building a Trustworthy System-Level GPU Simulator* [150].

NVArchSim was developed at Nvidia over the past 7 years, concurrently to the work described in this thesis, with the support of a large modelling team, and with access to proprietary information unavailable to others. Coupled with their methodology, and perseverance, this has enabled them to develop a simulator that is orders of magnitude faster than existing, publicly available GPU simulators (see Figure 5.15), while only exhibiting an average 17% error.

This number is higher than would be expected from an in-house development team, however the authors provide an interesting insight, that largely matches our own experiences - that complete accuracy is not critical to make good performance predictions. The authors argue “*that overly precise and/or overly slow architectural models hamper an architect’s ability to evaluate new features within a reasonable time frame, hurting productivity.*” They further claim, that accurate, but slow simulators, are unlikely to be adopted, because they are not able to simulate large enough workloads, which are of interest to computer architects. Just as in the case of REASSEMBLE, NVArchSim is not the only tool in Nvidia’s toolbox, and there are cases when other tools may be more appropriate. Like us, they provide correlation metrics, and focus on balancing speed

and accuracy, only including more detail in their model when a) there is a significant improvement in accuracy, and b) the improvement in accuracy related to more detailed modelling does not significantly impact performance.

There are many other similarities in our approaches. First of all, NVArchSim takes a trace based approach to performance modelling. Their traces are captured using NVBit [97], similar to Accel-Sim.

Secondly, NVArchSim also uses a CPU model to drive the execution, however their simulation is not executed in full-system mode - instead, the CPU execution begins on encountering a specific token in the trace, which then executes a (modified) CUDA driver. This approach would not be suitable for a mobile GPU, as it does not provide a method of accurately sharing memory between the CPU and the GPU.

Contrasting to our own approach, the authors of NVArchSim do not tune their model to existing hardware, instead relying on proprietary information for architectural and micro-architectural parameters. This is a luxury that few in the field of GPU simulation have, and one that is inconceivable in the space of *mobile* GPU simulation. While there is some public knowledge of Nvidia and AMD GPUs, public information about Arm, Apple, or Qualcomm GPUs is non-existent.

We also note, that we do not tune every model that we evaluate. We only tune our model against the Mali-G71 baseline. Following the tuning phase, we are able to accurately predict the performance Mali-G72 and Mali -G31 GPUs, by just modifying the exposed parameters.

5.3.6.1 Potential Improvements to REASSEMBLE

Two features in NVArchSim stand out, that could potentially provide significant improvements to REASSEMBLE. The first is dependency tracking. The authors express their surprise at the effectiveness of dependency tracking in both increasing the accuracy, as well as reducing simulation time. While some benefit to accuracy was expected, the scale of it was not. Furthermore, the performance improvement was unexpected, and resulted from the significant reduction in memory stalls, which are costly not only in real hardware, but in simulation as well. While we have included dependency tracking as a feature in our tracing format, we have not yet implemented it in REASSEMBLE, anticipating that it would be expensive in terms of simulation time. It appears however, that this is an interesting case where more accurate modelling also results in performance improvements.

Another performance improvement implemented in NVArchSim, is the complete

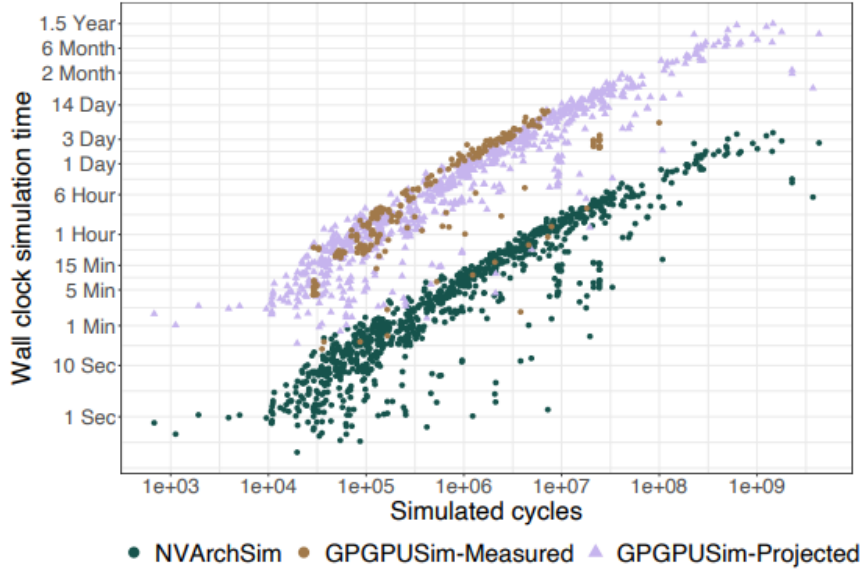


Fig. 5.15: NVArchSim is orders of magnitude faster than GPGPUSim, taking at most a day to execute workloads which can take over a year with GPGPUSim. Figure source: [150].

elimination of busy-ticking, which can result in significant overhead during simulation. While our model avoids some busy-ticking by modelling fewer components, it is still present, and could be further reduced.

5.3.7 Conclusion

While mobile GPUs evolve at a rapid pace with manufacturers releasing between two and three new designs each year, most of these designs are incremental. Nonetheless, GPU architects are faced with evaluating large design spaces to identify the next best configuration, to which current tools, created specifically for validation, are not well-suited.

In this section, we presented REASSEMBLE, a trace-driven simulator aiming at early and incremental design space exploration. We demonstrated the usefulness of the simulator in step by step selection of architectural configurations as well as for pre-silicon program optimization. Our framework achieves performance at least two orders of magnitude faster than cycle-accurate simulation, with a tuned error of 39%.

Future work will focus on further bridging the gap between cycle-accurate and fast simulation, taking advantage of analytical and machine learning models to augment trace-based simulation, providing further flexibility in the designs, as well as specific use cases, such as DSE of machine learning and computer vision enabled mobile GPUs ([135, 151]).

5.4 Summary

At the start of the chapter, we reflected on the goals and targets of the thesis, in particular on the third goal:

- ✓ To develop a simulation framework, which provides useful performance statistics, without the overhead of cycle-accurate simulation.

We recognized that while the simulation framework already provides useful performance insights, further flexibility and detail would significantly improve the framework for computer architects. Furthermore, existing cycle-accurate simulators don't provide the necessary infrastructure to execute full, multi-kernel workloads with complete accuracy. In this chapter, we explored machine learning, and trace-driven approaches to faster simulation with a complete software stack, and presented REASSEMBLE, a trace-driven simulator that can be effectively tuned to existing hardware. The presented framework further improves the already achieved goals of developing a fast framework, which provides useful performance statistics.

Conclusions

This thesis has identified a number of problems with existing GPU simulation frameworks, including but not limited to:

1. Existing GPU simulators often model approximated GPUs.
2. Existing GPU simulators often replace, approximate, or ignore significant components of the software stack.
3. Existing GPU simulators do not support environments required for simulating modern applications executing on mobile GPUs.

These problems lead to GPU architects relying on tools that are difficult to use and maintain, and that lead in inaccurate modelling of GPU execution. We sought to address these concerns by developing a framework with the following characteristics:

- ✓ Accurately simulates a state-of-the-art mobile GPU in a full-system context, enabling the use of *unmodified* vendor-supplied drivers and JIT compilers, operating in an *unmodified* target operating system, and executing *unmodified* applications.
- ✓ Supports simulation speeds, which enable the user to execute complete and complex applications typical of modern GPU workloads.
- ✓ Provides useful performance statistics, without the overhead of cycle-accurate simulation.

Chapter 3 motivates the problems faced in the implementation, development, and use of existing GPU simulators. Chapter 4 provides a holistic approach for accurate modelling, ease of development, and future-proofing against any changes to software stacks. Chapter 4 also presents real life uses cases of the *functional* GPU simulator for performance modelling, hi-lighting the importance of full-system simulation. Chapter 5 then explores different methods of fast performance modelling, and presents a novel, high-level, trace-based simulation approach combined with automatic tuning of

unknown parameters.

6.1 Contributions

The underlying goals of this thesis are to improve simulation environments and fast performance modelling for GPUs. Existing solutions treat GPUs as standalone devices, and lack the capability to execute the software stack in a fast simulation environment. These detailed simulators are also too slow to explore large design spaces. This thesis presents a methodology and tool for fast GPU simulation in a full-system environment, along with a trace-based early-design space exploration focused performance model.

The explored topics result in improvements over state-of-the-art GPU simulators, and open up the field for significant future work, presented in Section 6.3.

6.1.1 Full-System GPU Simulation

Existing GPU simulators do not accurately model the software stack, which introduces inaccuracies in the performance prediction, impacts maintenance and usability, and prevents simulation of modern day applications. Motivated by these issues, we developed a holistic full-system approach to GPU simulation, which faithfully executes the entire software stack using high-speed GPU simulation. Not only does this enable completely accurate simulation of the final GPU binary, but it also allows for easy maintenance of the simulation infrastructure, and enables simulation of complete end-to-end workloads typical of modern day GPU applications. Finally, full-system simulation is critical when simulating devices which operate using a shared memory model, as is the case with Arm GPUs.

In addition to completely and accurately executing the entire software stack, our fast simulation framework is scalable thanks to fast execution of the CPU-side code driving the GPU applications, which isn't the case for existing GPU simulators. Figure 6.1 recaps the performance benefits of our simulation framework relative to Multi2Sim. Figure 6.2 once again demonstrates the ability to execute multi-kernel applications with significant CPU and GPU interaction using our simulation framework.

6.1.2 Fast Performance Modelling

This thesis explores a number of fast performance modelling techniques, and due to the uncovered shortcomings of machine learning based performance modelling techniques, presents a novel fast, trace-driven performance model automatically tuned to

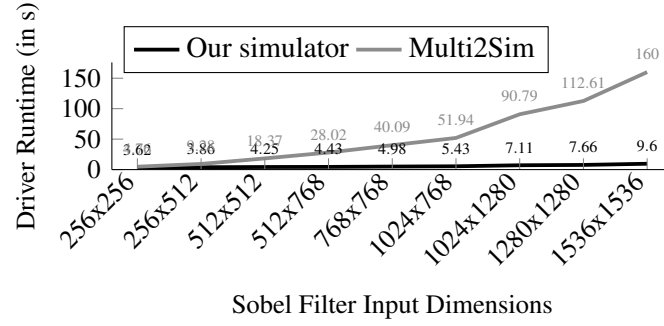


Fig. 6.1: A recap of the software stack executing on our DBT CPU simulator scales exceptionally well relative to Multi2Sim.

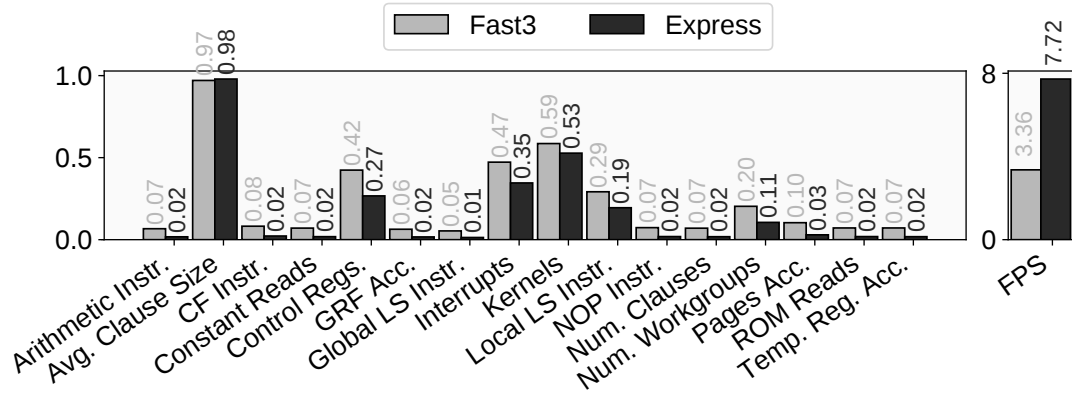


Fig. 6.2: Simulated SLAMBENCH statistics directly relate to HW performance, aiding the search for optimal configurations.

existing hardware, REASSEMBLE. This technique accurately predicts GPU performance, but also provides flexibility to the user to explore different GPU designs, by modifying the tuning parameters. The technique was extensively verified by tuning the performance model against an existing hardware platform, and achieving 39% error, which is on-par with many existing detailed GPU simulators, with two orders of magnitude improvement in simulation speed. Following this, the parameters in the performance model were modified to represent a different, existing GPU within the same architecture family. The predicted performance error was within a similar range to the original trained model - 43% error for the predicted Mali-G72 model relative to real hardware, and 45% for the Mali-G31 relative to real hardware. Figure 6.3 recaps the performance and accuracy benefits of REASSEMBLE.

The next section introduces improvements and future work relating to both the full-system simulation framework and to the GPU performance model.

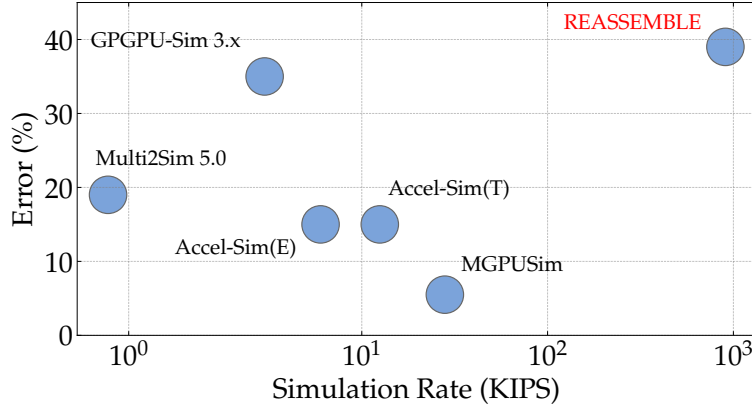


Fig. 6.3: REASSEMBLE provides two orders of magnitude improvement on simulation rate over detailed simulators with an accuracy comparable to GPGPUSim-3.x. Average error was calculated using MAPE. Metrics for remaining simulators were extracted from [83].

6.2 Current Limitations

The work described in this thesis provides clear benefits over existing solutions, however we take the time to identify potential limitations of this work.

6.2.1 Significant changes to GPU architectures

The performance model is tuned against an existing hardware implementation, and theoretically could also be trained against an existing detailed model. However, if the GPU architectures were to change significantly, both the traced based model and the tuning infrastructure around it would have to be re-designed.

6.2.2 GenSim and Captive Limitations

Captive is currently optimized to execute Arm code on x86 architectures, with specific guest components, such as the Arm MMU, mapped onto the host. While this could be extended to other architectures, MMUs that are significantly different may not map well. Furthermore, Captive uses GenSim to generate its CPU model, which has certain limitations. For example, there is limited built-in provision for modelling branch delay slots, context-sensitive instruction decoding, and out-of-order execution.

Similarly for automatic generation of the GPU simulator from GenSim, we have already found certain GPU features that cannot easily be described, such as accessing specific internal descriptors that are not part of the instruction set, and are not accessible without a known reference to main memory. Future work could look at extending

this, but with each iteration of the architecture, new features should be anticipated.

6.2.3 Limitations of the GPU Model

We currently do not model any graphics components of the GPU. Furthermore, as graphics rendering relies on different hardware, the presented trace-based performance model is not immediately applicable to graphics workloads. Additionally, sometimes graphics-specific instructions are used in compute applications, for example for loading images into GPU memory. These compute applications would not execute in our simulation environment.

The performance model presented in this thesis provides a good accuracy to performance tradeoff, however the model is not capable of detailed micro-architectural investigation. Instead, our model should be used to identify interesting points in the design space exploration to explore further using a detailed model. An interesting next step could be unifying the trace format of our model and an existing detailed GPU simulator, for example Accel-Sim, which is already capable of trace-based simulation. This would enable a seamless transition between the first stage preliminary design space exploration, and the more detailed second stage.

6.2.4 Software Availability Assumptions

While our technique is evaluated using an Arm Mali Bifrost GPU, and an Arm-v8 based CPU, we believe it to be extensible to other architectures. We assume that a vendor-provided binary runtime is available, in our case the Arm Mali OpenCL driver, which is available from the Arm website. We also rely on publicly available kernel driver source code, which we compile into the Linux kernel.

6.3 Future Work

This thesis only scratches the surface of what is possible with full-system simulation and fast performance modelling, and our hope is that it will provide a stepping stone for future work. This section outlines the potential next steps to be taken to advance the state of full-system and GPU simulation, as well as exploration that could be done using the current state of the simulator.

6.3.1 Functional Simulation

6.3.1.1 Graphics Simulation

This thesis has focused on simulating compute functionality. However, graphics simulators are few and far between, and both the academic community and industry would benefit from the availability of a hardware simulator capable of faithfully modelling graphics. This full-system approach presented in this thesis paves the way for graphics simulation, by inherently supporting any graphics software stack executing on the modelled hardware. **This work could lead to a research paper on the topic, and is an enabling technology for further graphics architecture research.**

6.3.1.2 Accelerating GPU Simulation

The simulator developed during this thesis is an interpreted, functional simulator. However, it has largely been developed using the GenSim simulator generation framework, which generates simulators supporting JIT-compilation into host code. Extending GenSim to fully support GPU execution would open up the possibility to JIT-compile GPU code to CPU code. Further extensions to GenSim could open up the possibility to JIT-compile guest GPU code onto a host GPU, while maintaining instruction accuracy of the guest GPU. **This work could lead to a research paper on the topic.**

6.3.2 Performance Modelling

6.3.2.1 Microbenchmarks and Performance Measurement Techniques for Mobile GPU

Collecting performance data on mobile GPUs is challenging, due to numerous sources of noise, and lack of mature debugging and performance measuring tools. A study identifying all of the sources of noise, and learning how to fully mitigate them would be of great benefit to the mobile systems community. **This work could lead to a research paper on the topic.**

6.3.2.2 Full-System Performance Modelling

This thesis lays the groundwork for full-system simulation of CPU/GPU platforms, however the performance modelling focuses only on GPU performance modelling. A natural extension to this would be modelling the performance of the entire system, including the CPU and the shared memory, as the end-to-end application always contains both the CPU and the GPU. This area is of growing importance as chip designers

move towards single-board computers such as the Apple M1. **This work could lead to a research paper on the topic, and is an enabling technology for further systems research.**

6.3.2.3 Flexible Tracing and Trace Simulation

Trace based simulation aims to alleviate the cost of detailed simulation, but trace collection and trace processing can in itself be quite expensive. However, different features collected in the trace can have different costs associated with them, and provide varying improvement to the accuracy. Flexible tracing and trace processing software could provide the infrastructure to generate and simulate the same trace at multiple granularities - for example by including or omitting specific features based on the cost and accuracy they provide. **This work could lead to a research paper on the topic, and is an enabling technology for further architecture research.**

6.3.2.4 Analytical Modelling

While the fast modelling techniques in this thesis are focused around trace-driven simulation, analytical modelling provides an alternate method of performance modelling. Current analytical models appear to work well on regular applications and architectures. Further work could explore the accuracy of existing analytical models on irregular workloads, as well as provide extensions and generalizations for tiled architectures, with parallelism present at numerous granularities. **This work could lead to a research paper on the topic, and is an enabling technology for further architecture research.**

6.3.2.5 Automating Performance Modelling

The performance models used in this thesis are manually included. Enabling the user to describe the performance impacts of different instructions or different GPU features using the GenSim model would allow for the automation of performance modelling. **This work could lead to a research paper on the topic, and is an enabling technology for further architecture research.**

6.3.3 Power and Area Modelling

GPU performance must be carefully balanced against power consumption and area, especially in mobile GPUs, which will have strict thermal limits. The performance

model could be extended to include power modelling using existing information already collected by the functional GPU simulator, such as memory accesses, cache hit rate, and register usage. Area models can be incorporated at multiple granularities, as demonstrated in 5.3. **This work could lead to a research paper on the topic.**

6.3.4 Applications of Functional Simulation and Fast Performance Modelling

6.3.4.1 Redundancy Analysis

GPUs execute thousands of threads in parallel, which largely execute the same code across all threads. Often, the only difference results from a single load from the input buffers, meaning that the vast majority of the executed code is redundant. Redundancies like these can be identified using functional instruction set simulation. The performance benefits can be modelled using fast performance modelling techniques. **This work could lead to a research paper on the topic.**

6.3.4.2 Application-Specific Instruction Set Extensions

A growing number of applications are using GPUs, with Nvidia listing 52 different categories including, but not limited to, Animation, Data Mining, Bioinformatics, and Computer Vision. Many of these applications will execute specific patterns of instructions, or would benefit from specific instructions that do not yet exist. Instruction set extensions can easily be modelled using our GenSim-driven Full-System Simulation Framework. The performance benefits of these extensions can be modelled using fast simulation techniques presented in this thesis. **This work could lead to a research paper on the topic.**

6.3.4.3 Optimal Hardware Mapping

Chapter 5.3 presents different hardware configurations achieving the same predicted performance. A complete design space exploration over the space could reveal multiple candidates for optimal hardware implementations. The performance model could also be used to explore different mappings of software on the same hardware, including re-compilation, making use of idle units, and re-purposing of existing units for new applications. **This work could lead to a PhD thesis.**

6.4 The Future of Simulation

The future of computer architecture relies heavily on simulation. Simulators are critical to the initial steps of designing a new architecture, optimizing the micro-architecture, and developing software before the hardware is available. This thesis introduced an approach to fast and accurate modelling of systems with multiple components, however, it only scratches the surface of simulating larger, emerging systems. A standard car already has dozens of chips inside it. Autonomous cars, which are still in their infancy have hundreds more - in the future, one could imagine vast networks of autonomous cars communicating with each other in order to share information in real time and avoid accidents, rather than relying only on computer vision techniques. Such vast systems, with potentially millions or billions of nodes will require simulation infrastructure that has not yet been invented, and will require new and innovative simulation techniques. Other questions that should be asked is how we'll simulate completely new and unseen architectures? The rise of quantum computing has the potential to completely re-invent certain sub-fields of computer science, where fast simulation will be just as critical as it is for existing architectures.

6.5 Summary and Final Remarks

This thesis has identified problems with existing GPU simulators and provides a holistic approach for overcoming these problems using a fast, full-system simulation environment. Alongside the full-system simulation, this thesis presents fast performance modelling techniques necessary for advancing modern day architectures and applications.

Appendix **A**

Appendix

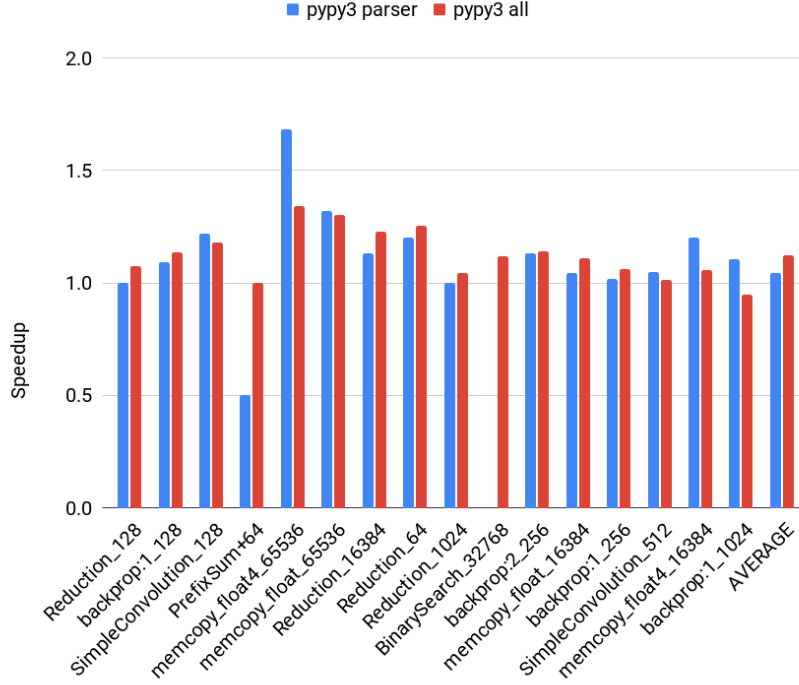


Fig. A.1: PyPy [152] is a fast, alternative implementation of Python. Instead of re-implementing our simulator in C++ to achieve performance improvements, we measured our simulator's performance using PyPy instead of the standard Python interpreter.

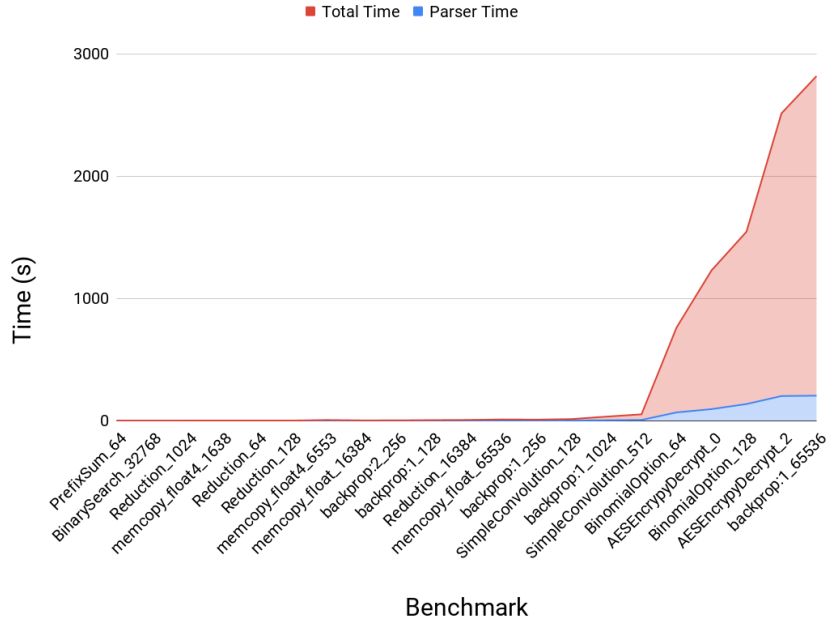


Fig. A.2: The parser is a potential bottleneck in our simulator. The parser takes around 10% of the total REASSEMBLE execution time. This could be alleviated by executing the parser in parallel.

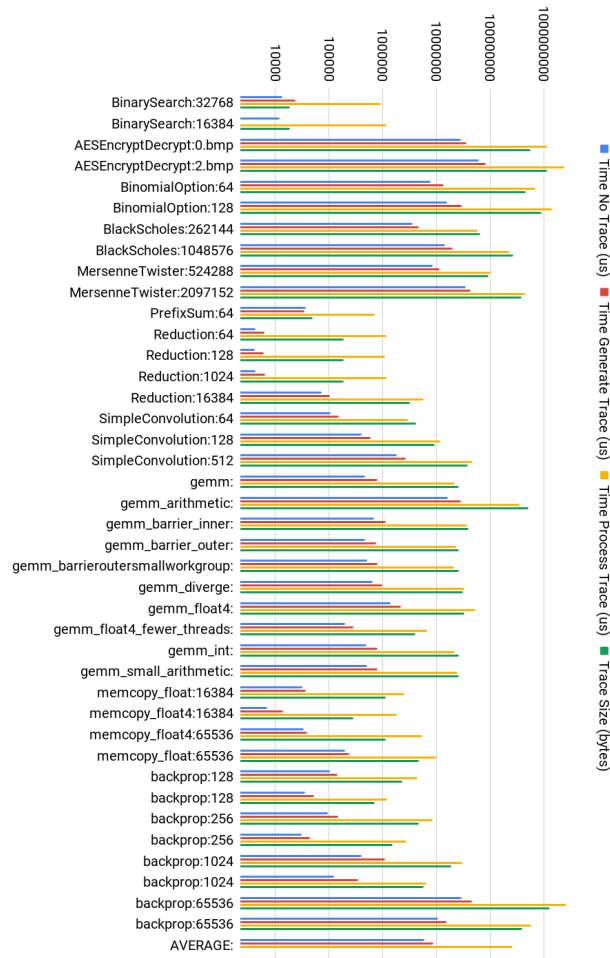


Fig. A.3: This figure shows the relative costs of our performance model. The cost of functional execution without tracing, with tracing, the execution of REASSEMBLE, and the size of trace files in the current version of REASSEMBLE.

JX1 JY1 JZ1 WX20 WY1 WZ1 ← Job Dimensions
T0 ← Thread ID
P ffff9b000000 5 0 0 ← New Clause: PC Length Dependency Tracking Information
L 3 fffa0c00000 ← Load: Bytes Address
T1
P ffff9b000000 5 0 0
L 3 fffa0c000008

Fig. A.4: REASSEMBLE uses a tracing format similar to [145]. A sample trace is presented in this figure.

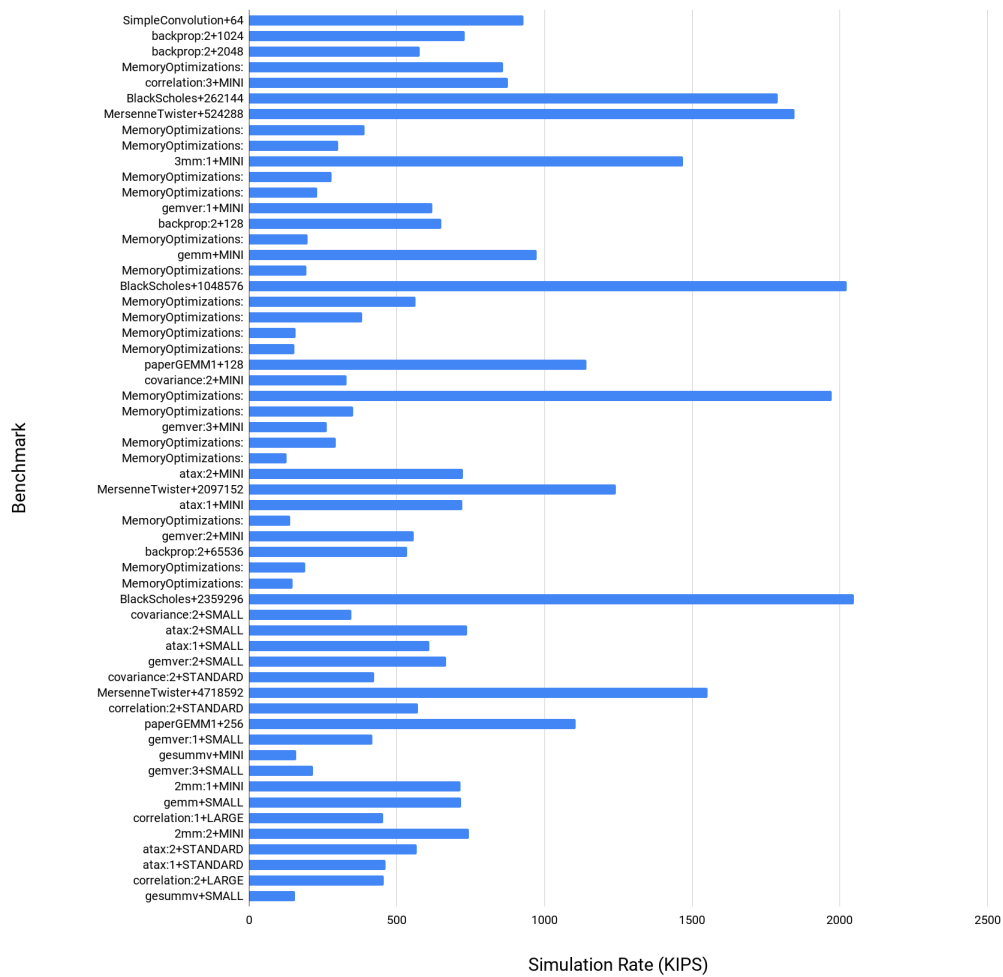


Fig. A.5: This figure shows a detailed breakdown of the simulation rate for all of the simulated benchmarks.

Bibliography

- [1] Nvidia GPU Applications. <https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/catalog/>. Accessed: 2020-01-16.
- [2] K.M. Gutttag, T.M. Albers, M.D. Asal, and K.G. Rose. The tms34010: an embedded microprocessor. *IEEE Micro*, 8(3):39–52, 1988.
- [3] Mike Asal, Graham Short, Tom Preston, Richard Simpson, Derek Roskell, and Karl Gutttag. The texas instruments 34010 graphics system processor. *IEEE Computer Graphics and applications*, 6(10):24–39, 1986.
- [4] George S Carson. Standards pipeline: The OpenGL Specification. *ACM SIGGRAPH Computer Graphics*, 31(2):17–18, 1997.
- [5] Glide API. <http://glide.sourceforge.net/>. Accessed : 2020-01-20.
- [6] Michael Macedonia. The gpu enters computing’s mainstream. *Computer*, 36(10):106–108, 2003.
- [7] Erik Lindholm and Stuart Oberman. The nvidia geforce 8800 gpu. In *2007 IEEE Hot Chips 19 Symposium (HCS)*, pages 1–17. IEEE, 2007.
- [8] Ian Buck. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*, pages 6–es. 2007.
- [9] Aaftab Munshi. The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.
- [10] Allied Research GPU Markets. <https://www.alliedmarketresearch.com/graphic-processing-unit-market>. Accessed: 2020-01-16.

- [11] AMD Samsung GPU. <https://www.counterpointresearch.com/amd-ready-shake-mobile-gpu-market/>. Accessed: 2020-01-16.
- [12] Imagination Apple IP Agreement. <https://www.eetimes.com/imagination-gets-back-into-apple/>. Accessed: 2020-01-16.
- [13] Imagination GPU of Everything. <https://www.eetimes.com/imagination-unveils-new-128-wide-alu-gpu-of-everything-family/>. Accessed: 2020-01-16.
- [14] Qualcomm AI. <https://www.qualcomm.com/invention/artificial-intelligence>. Accessed: 2020-01-16.
- [15] Arm GPU AI. <https://www.eetimes.com/arm-gpu-gets-more-ai-muscle/>. Accessed: 2020-01-16.
- [16] Thierry Moreau. The Past, Present, and Future of Deep Learning Acceleration Stacks. Arm Research Summit, 2019.
- [17] John Magnus Morton, Kuba Kaszyk, Lu Li, Jiawen Sun, Christophe Dubach, Michel Steuwer, Murray Cole, and Michael F. P. O’Boyle. Delayreplay: Delayed execution for kernel fusion in python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, page 43–56, New York, NY, USA, 2020. Association for Computing Machinery.
- [18] Khronos group : Opencl. <https://www.khronos.org/opencl/>. Accessed: 2021-10-30.
- [19] Khronos group : Opencl. <https://www.khronos.org/opengl/>. Accessed: 2021-10-30.
- [20] Khronos group : Sycl. <https://www.khronos.org/sycl/>. Accessed: 2021-10-30.
- [21] vulkan.org. vulkan.org. Accessed: 2021-10-30.
- [22] Github: Khronos group : Vulkan examples. <https://github.com/KhronosGroup/Vulkan-Samples>. Accessed: 2021-10-30.
- [23] Cuda zone. <https://developer.nvidia.com/cuda-zone>. Accessed: 2021-10-30.

- [24] Apple developer metal. <https://developer.apple.com/metal/>. Accessed: 2021-10-30.
- [25] Direct x. <https://developer.nvidia.com/directx>. Accessed: 2021-10-30.
- [26] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA, 2018. USENIX Association.
- [27] Jem Davies. The Bifrost GPU architecture and the ARM Mali-G71 GPU. In *HotChips*, August 2016.
- [28] The mali gpu: An abstract machine, part 4 - the bifrost shader core. <https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/the-mali-gpu-an-abstract-machine-part-4---the-bifrost-shader-core>. Accessed: 2021-10-30.
- [29] The bifrost quad: Replacing ilp with tlp. <https://www.anandtech.com/show/10375/arm-unveils-bifrost-and-mali-g71/2>. Accessed: 2021-10-30.
- [30] Bitesize bifrost 1: The benefits of clause shaders. <https://community.arm.com/arm-community-blogs/b/graphics-gaming-and-vr-blog/posts/bitesize-bifrost-1-the-benefits-of-clause-shaders>. Accessed: 2021-10-30.
- [31] Harry Wagstaff. *From High Level Architecture Descriptions to Fast Instruction Set Simulators*. PhD thesis, The University of Edinburgh, School of Informatics, 2015.
- [32] Harry Wagstaff, Tom Spink, and Björn Franke. Automated isa branch coverage analysis and test case generation for retargetable instruction set simulators. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, New York, NY, USA, 2014. Association for Computing Machinery.

- [33] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The archc architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, 2005.
- [34] ARM Holdings. Arm architecture reference manual, armv7-a and armv7-r edition. *Arm Holdings*, 2014.
- [35] Tom Spink, Harry Wagstaff, Björn Franke, and Nigel Topham. Efficient code generation in a region-based dynamic binary translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, page 3–12, New York, NY, USA, 2014. Association for Computing Machinery.
- [36] I. Böhm, B. Franke, and N. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 1–10, 2010.
- [37] Tom Spink, Harry Wagstaff, and Björn Franke. A retargetable system-level {DBT} hypervisor. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 505–520, 2019.
- [38] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D. Liu, and Wen-mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, March 2012.
- [39] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. URL: <http://www.cs.ucla.edu/pouchet/software/polybench>, 2012.
- [40] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10. Ieee, 2012.
- [41] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

- [42] AMD Staff. OpenCL and the AMD APP SDK v2.5 (Multi2Sim Bench), 2014.
- [43] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.
- [44] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O’Boyle, Graham Riley, Nigel Topham, and Steve Furber. Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, May 2015. arXiv:1410.2167.
- [45] Bruno Bodin, Harry Wagstaff, Sajad Saecdi, Luigi Nardi, Emanuele Vespa, John Mawer, Andy Nisbet, Mikel Luján, Steve Furber, Andrew J Davison, et al. Slambench2: Multi-objective head-to-head benchmarking for visual slam. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.
- [46] Bruno Bodin, Luigi Nardi, M. Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Krishna Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Luján, Björn Franke, Paul H. J. Kelly, and Michael F. P. O’Boyle. Integrating algorithmic parameters into benchmarking and design space exploration in 3d scene understanding. In Ayal Zaks, Bilha Mendelson, Lawrence Rauchwerger, and Wen-mei W. Hwu, editors, *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, pages 57–69. ACM, 2016.
- [47] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. Deep-smith: Compiler fuzzing through deep learning, 2018.
- [48] Cedric Nugteren. myGEMM. <https://github.com/cnugteren/myGEMM>. GitHub Repository (accessed 2018-07-30).
- [49] Sergey Ioffe and Christian Szegedy. Imagenet classification with deep convolutional neural networks. In *International Conference on Machine Learning*, pages 448–456, 2015.

- [50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.
- [51] G. Huang, Z. Liu, L. v. d. Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269, July 2017.
- [52] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3431–3440, June 2015.
- [53] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.
- [54] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, June 2017.
- [55] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [56] L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2414–2423, June 2016.
- [57] Ying Zhang, Mohammad Pezeshki, Philemon Brakel, Saizheng Zhang, César Laurent, Yoshua Bengio, and Aaron C. Courville. Towards end-to-end speech recognition with deep convolutional neural networks. In *INTERSPEECH*, 2016.
- [58] Yoon Kim. Convolutional neural networks for sentence classification. In *The Conference on Empirical Methods in Natural Language Processing*, 2014.
- [59] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014.

- [60] Stephen Jose Hanson and Lorien Y. Pratt. Comparing biases for minimal network construction with back-propagation. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 1*, pages 177–185. Morgan-Kaufmann, 1989.
- [61] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [62] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.
- [63] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, volume 2, page 6, 2017.
- [64] Jack Turner, José Cano, Valentin Radu, Elliot J Crowley, Michael O’Boyle, and Amos Storkey. Characterising across-stack optimisations for deep convolutional neural networks. In *Proc IISWC*. IEEE, 2018.
- [65] Yangqing Jia. *Learning semantic image representations at a large scale*. PhD thesis, UC Berkeley, 2014.
- [66] Jack Turner, Elliot J Crowley, Valentin Radu, José Cano, Amos Storkey, and Michael O’Boyle. Distilling with performance enhanced students. *CoRR*, 2018.
- [67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [68] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.
- [69] Sylvain Collange, David Defour, and David Parello. Barra, a parallel functional GPGPU simulator. In *18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS, 2010.

- [70] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2009, April 26-28, 2009, Boston, Massachusetts, USA, Proceedings*, pages 163–174. IEEE Computer Society, 2009.
- [71] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A heterogeneous CPU-GPU simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, Jan.-June 2015.
- [72] Xun Gong, Rafael Ubal, and David R. Kaeli. Multi2Sim Kepler: a detailed architectural GPU simulator. In *Proceedings of International Symposium on Performance Analysis of Systems and Software, ISPASS*. IEEE, April 2017.
- [73] V. M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and Espasa E. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS-2006*, pages 231–241. IEEE, March 2006.
- [74] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of PTX kernels. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 3–12, Washington, DC, USA, 2009. IEEE Computer Society.
- [75] Jiun-Hung Ding, Wei-Chung Hsu, BaiCheng Jeng, Shih-Hao Hung, and Yeh-Ching Chung. HSAemu - a full system emulator for HSA platforms. *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2014.
- [76] Geetika Malhotra, Seep Goel, and Smruti R. Sarangi. GpuTejas: A parallel simulator for GPU architectures. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*, pages 1–10. IEEE Computer Society, 2014.
- [77] H.Kim, J.Lee, N.B.Lakshminarayana, J.Sim, J.Lim, and T.Pho. MacSim: A CPU-GPU heterogeneous simulation framework. Technical report, Georgia Institute of Technology, 2012.

- [78] 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*, pages 37–46, New York, NY, USA, 2013. ACM.
- [79] Po-Han Wang, Gen-Hong Liu, Jen-Chieh Yeh, Tse-Min Chen, Hsu-Yao Huang, Chia-Lin Yang, Shih-Lien Liu, and James Greensky. Full system simulation framework for integrated CPU/GPU architecture. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test, VLSI-DAT 2014, Hsinchu, Taiwan, April 28-30, 2014*, pages 1–4. IEEE, 2014.
- [80] Nachiappan Chidambaram Nachiappan, Praveen Yedlapalli, Niranjan Soundararajan, Mahmut Taylan Kandemir, Anand Sivasubramaniam, and Chita R. Das. GemDroid: A framework to evaluate mobile platforms. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14*, pages 355–366, New York, NY, USA, 2014. ACM.
- [81] Anthony Gutierrez, Bradford Beckmann, Alexandru Dutu, Joseph Gross, John Kalamatianos, Onur Kayiran, Michael LeBeane, Matthew Poremba, Brandon Potter, Sooraj Puthoor, Mark Wyse, Jieming Yin, Akshay Jain, Tim Rogers, Xianwei Zhang, and Matt Sinclair. Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level. In *Proceedings of The 24th IEEE International Symposium on High-Performance Computer Architecture, HPCA*, February 2018.
- [82] Yifan Sun, Trinayan Baruah, Saiful A Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, et al. Mgpusim: Enabling multi-gpu performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 197–209, 2019.
- [83] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

- [84] Tor M Aamodt, Wilson WL Fung, Inderpreet Singh, Ahmed El-Shafiey, Jimmy Kwa, Tayler Hetherington, Ayub Gubran, Andrew Boktor, Tim Rogers, Ali Bakhoda, et al. Gpgpu-sim 3. x manual, 2012.
- [85] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 335–344. IEEE, 2012.
- [86] Thomas Spink. *Efficient cross-architecture hardware virtualisation*. PhD thesis, The University of Edinburgh, School of Informatics, 2017.
- [87] Arm. Fast Models. <https://developer.arm.com/tools-and-software/simulation-models/fast-models>. Accessed : 2020-01-20.
- [88] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [89] Android. Android emulator. <https://developer.android.com/studio/run/emulator>. Accessed : 2020-01-20.
- [90] Sangpil Lee and Won Woo Ro. Parallel GPU architecture simulation framework exploiting work allocation unit parallelism. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21-23 April, 2013, ISPASS*, pages 107–117. IEEE, 2013.
- [91] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.
- [92] Rene De Jong and Andreas Sandberg. Nomali: Simulating a realistic graphics driver stack using a stub gpu. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 255–262. IEEE, 2016.
- [93] Synopsys. Designware. <https://www.synopsys.com/designware-ip/processor-solutions/arc-development-tools/simulation-tools.html>. Accessed : 2020-01-20.

- [94] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.
- [95] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 100–112. ACM, 2018.
- [96] Matthew Poremba, Alexandru Dutu, Gaurav Jain, Pouya Fotouhi, Michael Boyer, and Bradford M. Beckmann. Towards full-system discrete gpu simulation.
- [97] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 372–383, 2019.
- [98] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. Rogers. Lost in abstraction: Pitfalls of analyzing gpus at the intermediate language level. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 608–619, 2018.
- [99] Prasun Gera, Hyojong Kim, Hyesoon Kim, Sunpyo Hong, Vinod George, and Chi-Keung CK Luk. Performance characterisation and simulation of intel’s integrated gpu architecture. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 139–148. IEEE, 2018.
- [100] Melanie Kambadur, Sunpyo Hong, Juan Cabral, Harish Patil, Chi-Keung Luk, Sohaib Sajid, and Martha A Kim. Fast computational gpu design with gt-pin. In *2015 IEEE International Symposium on Workload Characterization*, pages 76–86. IEEE, 2015.
- [101] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, 2004.

- [102] Naga K Govindaraju, Scott Larsen, Jim Gray, and Dinesh Manocha. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 89–es, 2006.
- [103] Weiguo Liu, Wolfgang Muller-Wittig, and Bertil Schmidt. Performance predictions for general-purpose computation on gpus. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 50–50. IEEE, 2007.
- [104] L. Wang, M. Jahre, A. Adileho, and L. Eeckhout. Mdm: The gpu memory divergence model. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1009–1021, 2020.
- [105] Lu Wang, Magnus Jahre, Almutaz Adileh, and Lieven Eeckhout. Mdm: The gpu memory divergence model. *Mars*, 13(6):0, 2013.
- [106] Jen-Cheng Huang, Joo Hwan Lee, Hyesoon Kim, and Hsien-Hsin S Lee. Gpumech: Gpu performance modeling technique based on interval analysis. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 268–279. IEEE, 2014.
- [107] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 152–163, 2009.
- [108] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 280–289, 2010.
- [109] Jiayuan Meng, Vitali A Morozov, Kalyan Kumaran, Venkatram Vishwanath, and Thomas D Uram. Grophecy: Gpu performance projection from cpu code skeletons. In *SC’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2011.
- [110] Cheng Luo and Reiji Suda. A performance and energy consumption analytical model for gpu. In *2011 IEEE ninth international conference on dependable, autonomic and secure computing*, pages 658–665. IEEE, 2011.

- [111] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *2011 IEEE 17th international symposium on high performance computer architecture*, pages 382–393. IEEE, 2011.
- [112] Junjie Lai and André Seznec. Break down gpu execution time with an analytical method. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 33–39, 2012.
- [113] Shane Ryoo, Christopher I Rodrigues, Sam S Stone, Sara S Baghsorkhi, Sain-Zee Ueng, John A Stratton, and Wen-mei W Hwu. Program optimization space pruning for a multithreaded gpu. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.
- [114] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP ’10*, page 105–114, New York, NY, USA, 2010. Association for Computing Machinery.
- [115] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.
- [116] Elias Konstantinidis and Yiannis Cotronis. A practical performance model for compute and memory bound gpu kernels. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 651–658. IEEE, 2015.
- [117] Sander De Pestel, Sam Van den Steen, Shoaib Akram, and Lieven Eeckhout. Rppm: Rapid performance prediction of multithreaded workloads on multicore processors. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 257–267. IEEE, 2019.
- [118] Björn Franke. Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 69–78, 2008.

- [119] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *International conference on green computing*, pages 115–122. IEEE, 2010.
- [120] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. Statistical gpu power analysis using tree-based methods. In *2011 International Green Computing Conference and Workshops*, pages 1–6. IEEE, 2011.
- [121] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420. IEEE Computer Society, 2007.
- [122] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. Performance and power analysis of ati gpu: A statistical approach. In *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*, pages 149–158. IEEE, 2011.
- [123] Shuaiwen Song and Kirk Cameron. System-level power-performance efficiency modeling for emergent gpu architectures. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 473–474, 2012.
- [124] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. A simplified and accurate model of power-performance efficiency on emergent gpu architectures. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 673–686. IEEE, 2013.
- [125] Michael Boyer, Jiayuan Meng, and Kalyan Kumaran. Improving gpu performance prediction with data transfer modeling. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1097–1106. IEEE, 2013.
- [126] Ali Karami, Sayyed Ali Mirsoleimani, and Farshad Khunjush. A statistical performance prediction model for opengl kernels on nvidia gpus. In *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADSD 2013)*, pages 15–22. IEEE, 2013.

- [127] Ioana Baldini, Stephen J Fink, and Erik Altman. Predicting gpu performance from cpu runs using machine learning. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 254–261. IEEE, 2014.
- [128] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737, 2015.
- [129] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The ArchC architecture description language and tools. *International Journal of Parallel Programming*, 33(5):453–484, Oct 2005.
- [130] Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *Proceedings of the 50th Annual Design Automation Conference, DAC '13*, pages 21:1–21:6, New York, NY, USA, 2013. ACM.
- [131] Martin Kristien, Tom Spink, Brian Campbell, Susmit Sarkar, Ian Stark, Björn Franke, Igor Böhm, and Nigel Topham. Fast and correct load-link/store-conditional instruction handling in dbt systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3544–3554, 2020.
- [132] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. Performance portable gpu code generation for matrix multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 22–31. ACM, 2016.
- [133] Cedric Nugteren. Clblast: A tuned opencl blas library. *arXiv preprint arXiv:1705.05249*, 2017.
- [134] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google workloads for consumer

- devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '18*, pages 316–331, New York, NY, USA, 2018. ACM.
- [135] V. Radu, K. Kaszyk, Y. Wen, J. Turner, J. Cano, E. J. Crowley, B. Franke, A. Storkey, and M. O’Boyle. Performance aware convolutional neural network channel pruning for embedded gpus. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 24–34, 2019.
- [136] Kuba Kaszyk, Harry Wagstaff, Tom Spink, Björn Franke, Mike O’Boyle, Bruno Bodin, and Henrik Uhrenholt. Full-system simulation of mobile cpu/gpu platforms. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 68–78. IEEE, 2019.
- [137] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. Autotuning opencl workgroup size for stencil patterns. *CoRR*, abs/1511.02490, 2015.
- [138] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for armv8-a, risc-v, and CHERI-MIPS. *Proc. ACM Program. Lang.*, 3(POPL):71:1–71:31, 2019.
- [139] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.
- [140] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. Citeseer, 2007.
- [141] Andreas Töschel, Michael Jahrer, and Robert M Bell. The bigchaos solution to the netflix grand prize. *Netflix prize documentation*, pages 1–52, 2009.
- [142] A. Adileh, C. González-Álvarez, J. Miguel De Haro Ruiz, and L. Eeckhout. Racing to hardware-validated simulation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 58–67, 2019.

- [143] Nicolas Derumigny, Fabian Gruber, Théophile Bastian, Christophe Guillon, Louis-Noel Pouchet, and Fabrice Rastello. From micro-ops to abstract resources: constructing a simpler cpu performance model through microbenchmarking, 2020.
- [144] Anthony Gutierrez, Joseph Pusdesris, Ronald G Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher D Emmons, Mitchell Hayenga, and Nigel Paver. Sources of error in full-system simulation. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–22. IEEE, 2014.
- [145] S. Pal, K. Kaszyk, S. Feng, B. Franke, M. Cole, M. O’Boyle, T. Mudge, and R. G. Dreslinski. Hetsim: Simulating large-scale heterogeneous systems using a trace-driven, synchronization and dependency-aware framework. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–24, 2020.
- [146] Cedric Nugteren, Gert-Jan Van den Braak, Henk Corporaal, and Henri Bal. A detailed gpu cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 37–48. IEEE, 2014.
- [147] Vignesh Adhinarayanan and Wu-chun Feng. An automated framework for characterizing and subsetting gpgpu workloads. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 307–317. IEEE, 2016.
- [148] Hikey 960 soc reference manual. https://github.com/96boards/documentation/blob/master/consumer/hikey/hikey960/hardware-docs/HiKey960_SoC_Reference_Manual.pdf. Accessed: 2021-11-02.
- [149] Mahmoud Khairy, Akshay Jain, Tor M. Aamodt, and Timothy G. Rogers. Exploring modern GPU memory system design challenges through accurate modeling. *CoRR*, abs/1810.07269, 2018.
- [150] Oreste Villa, Daniel Lustig, Zi Yan, Evgeny Bolotin, Yaosheng Fu, Niladri Chatterjee, Nan Jiang, and David Nellans. Need for speed: Experiences building a trustworthy system-level gpu simulator. In *2021 IEEE 27th International*

Symposium on High Performance Computer Architecture (HPCA), pages 868–880. IEEE, 2021.

- [151] S. Saeedi, B. Bodin, H. Wagstaff, A. Nisbet, L. Nardi, J. Mawer, N. Melot, O. Palomar, E. Vespa, T. Spink, C. Gorgovan, A. Webb, J. Clarkson, E. Tomusk, T. Debrunner, K. Kaszyk, P. Gonzalez-De-Aledo, A. Rodchenko, G. Riley, C. Kotselidis, B. Franke, M. F. P. O’Boyle, A. J. Davison, P. H. J. Kelly, M. Luján, and S. Furber. Navigating the landscape for real-time localization and mapping for robotics and virtual and augmented reality. *Proceedings of the IEEE*, 106(11):2020–2039, 2018.
- [152] Eli Biham and Jennifer Seberry. Pypy: another version of py. *eSTREAM, ECRYPT Stream Cipher Project, Report*, 38:2006, 2006.