GPU Power Modeling and Architectural Enhancements for GPU Energy Efficiency

vorgelegt von
Dipl.-Ing.
Jan Lucas
geb. in Berlin

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 7. Dezember 2018

Berlin 2019

CONTENTS

ABSTRACT

Graphics Processing Units (GPUs) can now be found in nearly every PC and smartphone. Initially designed for 3D graphics, they evolved into general purpose accelerators, able to outperform CPUs on many tasks. The architecture of GPUs is optimized for massively parallel applications. This reduces the required control logic but also results in lower performance in applications with irregular control flow. The energy per instruction is often lower in GPUs than in CPUs, but due to their high throughput, discrete GPUs can still use 200 W and more. GPU performance is limited by power consumption, as the power dissipation at higher speeds would exceed the cooling abilities. Better energy efficiency does not only extend battery life and reduce power bills but also enables higher performance.

To increase the energy efficiency, we measure and model the energy consumption of existing GPUs. A custom GPU power measurement infrastructure and an architectural power simulator called GPUSimPow are described and evaluated. Due to the lower control overhead in GPUs, accurately modeling the power consumption of the memory interface and execution units is important. Regular architectural simulators do not model the data-dependent energy consumption but assume that energy consumption per operation does not depend on the data. We show that this assumption is not true, but that GPU power consumption can vary by more than 60% with different data and present two data-dependent power models.

Afterwards, we focus on architectural enhancements to improve the energy efficiency. Two techniques focus on the memory interface: A novel approximation technique reduces the DRAM refresh energy and an optimized encoding scheme reduces the power consumption of the external interface between GPU and DRAM by up to 6%. We continue with enhancements to improve the energy efficiency of the GPU cores. We evaluate an alternative to the conventional SIMT GPU architecture called temporal SIMT (TSIMT) and extend it to spatiotemporal SIMT. Temporal SIMT makes the execution of code with irregular control flow more efficient but can reduce the performance of applications by decreasing the ability of the GPU to tolerate memory latency. Spatiotemporal SIMT provides a good combination of conventional SIMT and TSIMT. In both architectural variants so-called Scalarization can be used to remove redundant operation. We show that spatiotemporal SIMT with Scalarization improves the energy-delay product by 26.2% compared to conventional GPUs.

ZUSAMMENFASSUNG

Graphics Processing Units (GPUs) sind heute Teil nahezu jedes PCs oder Smartphones. Ursprünglich für 3D Grafik entwickelt, wurden sie zu allgemein nutzbaren Beschleunigern weiterentwickelt, die viele Aufgaben schneller als CPUs erfüllen. Ihre Architektur ist optimiert für massiv parallele Anwendungen. Dies reduziert die nötige Kontrollogik, aber senkt auch die Rechenleistung bei uneinheitlichem Kontrollfluss. Pro Instruktion verbrauchen GPUs oft weniger Energie als CPUs, aber durch ihren hohen Durchsatz, können sie trotzdem 200 Watt und mehr verbrauchen. Die Rechenleistung wird dabei von der elektrischen Leistungsaufnahme beschränkt, weil bei höhren Geschwindigkeiten die Kühlung überfordert würde. Eine höhere Energieeffizienz führt daher nicht nur zu einer verlängerten Batterielaufzeit und geringeren Energiekosten, sondern ermöglicht auch höhere Rechenleistung.

Um die Energieeffizienz zu erhöhen, messen und modelieren wir zunächst den Energieverbrauch existierender GPUs. Eine speziell angepasste Meßinfrastruktur und ein architekturelles Powermodell namens GPUSimPow werden vorstellt und getestet. Der geringe Kontrolloverhead in GPUs macht die genaue Modellierung der Energie von Speicherinterface und Ausführungseinheiten besonders wichtig. Gewöhnliche Architektursimulatoren modellieren keinen datenabhängigen Energieverbrauch, sondern nehmen einen konstanten Energieverbauch pro Operation an. Wir zeigen, das diese Annahme nicht zutrifft und der Energieverbrauch der GPU sich durch andere Daten um mehr als 60% erhöhen kann und präsentieren zwei datenabhängige Powermodelle.

Anschließend zeigen wir Verbesserungen der GPU Architektur zur Erhöhung der Energieeffizienz. Zwei Techniken setzen am Speicherinterface an: Eine neuartige Näherungstechnik reduziert den Energieverbrauch des DRAM Refresh und ein optimiertes Kodierungsverfahren reduziert die Energie der Schnittstelle zwischen GPU und DRAM um bis zu 6%. Danach verbessern wir die Energieeffizient der GPU Kerne. Wir untersuchen "temporal SIMT" (TSIMT), eine Alternative zu konventionellen SIMT GPU Kernen und erweitern es sie zu "spatiotemporal SIMT" (STSIMT). TSIMT ermöglicht eine effektivere Ausführung von Programmcode mit uneinheitlichem Kontrollfluss, reduziert aber auch die Möglichkeiten der GPU Speicherlatenzen zu tolerieren. STSIMT ist eine gute Kombination von konventionellen SIMT mit TSIMT. In beiden Architekturvarianten kann Skalarisierung verwenden werden, um redundante Operationen zu vermeiden. Die Kombination von STSIMT mit Skalarisierung kann das Energie-Verzögerungs-Produkt um 26.2% gegenüber einer konventionellen GPU verbessern.

PRE-PUBLISHED PAPERS

Parts of this thesis are based on the following pre-published papers.

JOURNAL

J. Lucas, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "Spatiotemporal SIMT and Scalarization for improving GPU efficiency," *ACM Transactions on Architecture and Code Optimization*, vol. 12, no. 3, 32:1–32:26, Sep. 2015, ISSN: 1544-3566. DOI: 10.1145/2811402

CONFERENCE

J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2013. DOI: 10.1109/ISPASS.2013.6557150 © 2013 IEEE

J. Lucas and B. Juurlink, "ALUPower: Data dependent power consumption in GPUs," in *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, 2016. DOI: 10.1109/MASCOTS.2016.21 © 2016 IEEE

J. Lucas, S. Lal, and B. Juurlink, "Optimal DC/AC data bus inversion coding," in *Design, Automation and Test in Europe, DATE*, EDAA, 2018. DOI: 10.23919/DATE.2018.8342169

WORKSHOP

J. Lucas, M. Alvarez-Mesa, M. Andersch, and B. Juurlink, "Sparkk: Quality-scalable approximate storage in DRAM," in *The Memory Forum*, 2014

1

INTRODUCTION

Today GPUs can be found in nearly every personal computer as well as in devices such as smartphones, gaming consoles or tablets. GPU stands for Graphics Processing Unit, but their usage is not limited to graphics anymore, instead, GPU serve as general purpose Turing-complete computing devices. They are able to outperform CPUs on many tasks. CPUs and GPUs complement each other and are now often integrated into a system on a chip (SoC).

This chapter provides an introduction for the thesis. We start with a high-level overview of the architecture of GPUs and their programming model. We then provide a short history of GPUs and their usage for non 3D graphics applications, the so-called general purposed GPU computing or short GPGPU. The thesis continues with an overview of current usages of GPGPU computing. We then explain the issues caused by GPU power consumption and why measuring and modeling the GPU power consumption is important and continue with a discussion of architectural enhancements for improved energy efficiency. The main research questions of this thesis are formulated in Section 1.7. We conclude this introduction with an overview of the structure of this thesis.

1.1 GPU ARCHITECTURE AND PROGRAMMING MODEL

GPUs complement CPUs because their architecture focuses on the effective execution of algorithms that are embarrassingly parallel and offer high levels of data parallelism. High performance CPUs use large parts of their area and power budget for complex control logic that enables OutOfOrder execution and for large caches. These design choices enable high performance execution of single-threaded code, but due to the complex logic and high power consumption, only a few of these cores fit on the silicon die. Embarrassingly parallel algorithms execute well on GPUs, as they do not require high single threaded performance. GPUs remove large caches and simplify the control logic and add a higher number of cores and very wide SIMD (Single Instruction Multiple Data) execution units per core. This results in a slow execution speed of each thread but if a high number of threads runs

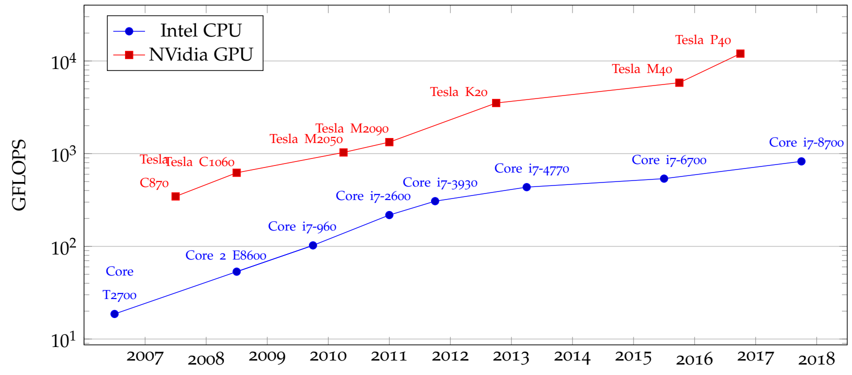


Figure 1.1: CPU vs. GPU Peak Performance

in parallel, GPUs can reach a very high throughput, one or even two orders of magnitude above CPUs. This difference in peak performance can also be seen in Figure 1.1. It compares the peak performance of high performance NVIDIA GPUs with high performance Intel CPUs over time. The difference in peak performance is not due to better process technology nor clock speed. NVIDIA GPUs typically are one or two process nodes behind Intel CPUs and clock speeds are significantly lower. Despite executing their instruction using SIMD execution units, GPUs are typically programmed using SPMD (Single Program Multiple Data) programming models. This programming model is often easier to use for programmers than directly using very large masked SIMD instruction but allows GPUs to amortize the cost of instruction fetch, decoding and scheduling over groups of threads called warps (NVIDIA) or wavefronts (AMD). If all threads in a warp follow the same control flow, the full throughput of the SIMD execution units can be utilized. If the control flow of the threads within a warp differs, the control flow is serialized and the execution units are only partially active. This is also called control flow divergence and results in a reduced throughput. NVIDIA coined the term SIMT (single instruction multiple threads) for this programming model that executes using SIMD execution units but mostly looks like programming independent scalar threads to the programmer. The execution model is efficient as long as control flow divergence is rare, however, with more complex and irregular applications running on GPUs the control flow often becomes divergent and performance, as well as power efficiency, is reduced. As kernels executed on the GPU get increasingly more complex, this problem becomes more important.

The high performance of GPUs in many applications would not be possible without the development of special high bandwidth memories. GPUs use special memory technologies such as GDDR5/5X and HBM/HBM2 that focus

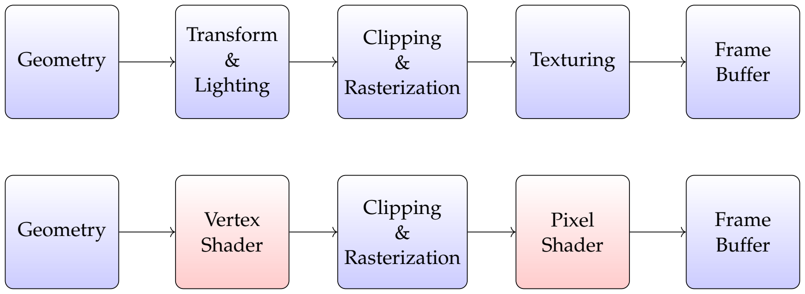


Figure 1.2: 3D Graphics Pipelines

on very high bandwidth while accepting their higher cost and smaller density compared to mainstream CPU memory technologies such as DDR3 and DDR4.

1.2 (GP)GPU HISTORY

GPUs evolved out of fixed function 3D graphics accelerators. The top of Figure 1.2 shows a simplified, typical 3D pipeline. 3D geometry as triangles is ingested into the pipeline at the start, the geometry is then rotated and translated to a specific viewpoint. Lighting information is calculated and a perspective transform is applied. Clipping removes triangles and parts of triangles that are outside of the viewing area. Rasterization breaks the triangles down into individual pixels. Finally, texturing applies a wrapped image ("texture") to these pixels. In the first 3D accelerators only parts of this pipeline were present. Evans and Sutherland's Picture System [1] from 1974 employed a vector display that used the electron beam to directly draw lines to the screen and was used to draw wire-frame graphics only. Lighting, Rasterization, and Texturing were not required in such a system. SGI's IRIS systems were early systems that implemented nearly the whole pipeline, albeit without texturing. These early systems employed complex designs using many chips. The geometry engine was built from one the first VLSI chips. Twelve of these chips were combined to implement 4×4 matrix multiplication, clipping and scaling to the display coordinates [2]. SGI's IRIS systems were also very influential in regards to the programming interface to the 3D hardware. They introduced the IRIS GL API which would later evolve into the OpenGL standard. These early systems were very expensive high-end systems or workstations. In 1989 Namco released their "System 21" arcade system board [3] that employed hardware accelerated polygonal 3D graphics for entertainment in video game arcades. In the 1990s accelerated 3D graphics arrived in mainstream PCs and video game consoles. One of the first successful 3D accelerator for PCs was the 3Dfx "Voodoo" accelerator. It focused on texture mapping and used the PC CPU for calculating the perspective transformation and set up the triangle rasterization engine on each triangle. Similar to the old high-end 3D accelerators a multi-chip design was employed. One chip was used for texturing while the other chip handled the framebuffer and the PCI interface. NVIDIA's Geforce 256 included hardware support for transform and lighting [4]. NVIDIA coined the term "Graphics Processing Unit" (GPU) for this design launched in late 1999 [5]. It combined transform and lighting, triangle setup and texturing into a single chip. However, these first GPUs had little in common with today's GPGPU as they were restricted fixed-function units, designed for a single task. Demand for more realistic graphics resulted in more flexibility being added in the next few generations of GPUs. NVIDIA's Geforce 3 GPUs supported vertex- and pixel shaders: Small programs running on the GPU to control the transformation of the scene and the shading of each pixel. In the beginning, these shaders were severely limited in their capabilities. Only short shaders with a few instructions were possible and the control flow was restricted. Later generations lifted these restrictions. These early GPUs also employed different hardware units (shaders) for vertex and pixel calculations. Vertex transformation and lighting calculation were performed using floating point arithmetic while pixel shaders often used lower accuracy fixed-point arithmetic. The Xbox 360 was the first gaming console utilizing unified shaders that are used to execute both pixel and vertex shaders [6]. Research showed that this leads to higher performance and better area efficiency and adds the flexibility to efficiently render scenes with varying pixel and vertex shader workload balances [7].

As early as 2004 people started applying GPUs to regular computing tasks such as sorting numbers or linear algebra tasks using APIs intended for 3D graphics [8]. The use of GPUs for general-purpose computing purposes is often called general-purpose computing on graphics processing units or short GPGPU. APIs tailored towards GPGPU such as Brook for GPUs, NVIDIA's CUDA and OpenCL from Khronos Group helped to kick-start a wide adoption of GPUs for computing [9], [10].

GPUs were especially successful as accelerators for high-performance computing. At the end of 2010 three out of first seven supercomputers with more than 1 petaflops utilized GPUs [11]. GPUs performed well especially in terms of energy efficiency and at the end of 2011, almost all the Top 30 slots of the Green 500 list of the most power efficient supercomputers used GPUs [12].

GPUs are now ubiquitous, not just in HPC, desktop PCs and laptops but also in smartphones, tablets and gaming consoles. Even on these low power, mobile platforms GPUs are not only employed for rendering 3D graphics but also to perform GPGPU workloads such as neural network interference [13], [14], face recognition [15] or image processing [16].

1.3 GPGPU APPLICATIONS

GPU have also been popular for the so-called "mining" of cryptocurrencies, as they greatly increased the hashrate compared to CPUs. Before the availability of FPGA and ASIC based mining devices, GPU quickly replaced CPUs for bitcoin mining [17]. Newer cryptocurrencies such as Ethereum employ proof-of-work function specifically engineered to be GPU friendly [18]. The popularity of GPU mining resulted in strong demand for GPUs and pushed up AMD's share price [19].

An even more important market for GPUs is artificial intelligence (AI) and in particular, "deep learning". In 2009 Raina et al. demonstrated a speedup of $70\times$ by using GPU for training a large scale deep neuronal network [20]. Various artificial intelligence frameworks such as Theano [21], Tensorflow [22], Torch [23] and Caffe [24] support GPU acceleration. In 2013 Coates et al. showed how to train very large network using 12 GPUs while previously training a similar network at the same speed required 16000 CPU cores [25]. In parts of the training, each GPU exceeded 1 TFlops in application performance. Mayor cloud computing platforms such as Amazon Webservices, Google Cloud Platform, and Microsoft Azure now offer virtual machines with GPUs and often advertise GPUs for artificial intelligence applications [26]–[28]. AMD offers GPUs specifically aimed at the AI market [29] and NVIDIA added special execution units called tensor cores to some of their GPUs that provide even higher throughput for some linear algebra operations common in AI [30].

1.4 GPU POWER CONSUMPTION

Especially in these battery-powered devices, power consumption is an important topic for the design of GPUs. High GPU energy consumption would cause a short battery life. At the same time, energy conservation applies and the energy consumed by the GPU is turned into heat. This heating is often an issue, even when battery life is not a concern. Many mobile GPU containing devices have relatively direct contact to the skin of its user and must limit their surface temperatures to $45°C$ or less [31]. This limits the possible power consumption of the GPU even further. GPU performance is limited by the maximum possible power dissipation is not limited to mobile devices. Even the performance of discrete desktop GPUs with an elaborate active cooling system is often limited by the maximum possible power dissipation. The used semiconductor technology would allow the use of higher frequencies but running at these frequencies would cause the GPU to quickly exceed the maximum allowed temperature. GPUs now often employ "boost clocks" several $100\,Mhz$ above their base frequency. These boost clocks allow the

GPU to run at a higher frequency if the workload has a low average power consumption. This happens, e.g.: if the workload is concentrated into short peaks with gaps in between where the GPU is able to cool down again or if the workload does not fully utilize the GPU. Another issue directly linked to power consumption is power delivery. Current desktop GPUs can easily use more than $250\,W$ and at the same time they utilize core voltages only slightly above $1\,V$. This can result in currents of $200\,A$ and more. At these extremely high currents even tiny resistances can easily cause significant voltage drops [32].

1.5 GPU POWER MEASUREMENT AND MODELING

In order to understand the power consumption of GPUs, we need to measure and model the power consumption. Discrete GPUs are plugged into a computer system and they receive power via the PCIe slot and also via additional power cables. In this thesis, we developed power measurement testbeds that allow us to measure the power consumption of GPUs.

We can, however, only measure existing GPUs and can only measure the power consumption of the whole GPU, but cannot measure the power consumption of individual GPU components within an application. However, to improve the power efficiency we need to be able to estimate the power efficiency of modified (and potentially improved) GPU designs and gain insight into the power consumption of the individual components. This thesis presents an architectural power simulator along with two improvements. The power simulator allows us to estimate the power consumption of a GPU workload. The configuration of the GPU can be determined using a configuration file. Performance and energy benefits of architectural enhancements can estimated by implementing the proposed changes in the simulator.

1.6 GPU ARCHITECTURAL ENHANCEMENTS FOR ENERGY EFFICIENCY

Measuring and modeling the energy consumption of GPUs is the first step to increase the energy efficiency of GPUs. But what concrete architectural enhancements are possible for improving the energy efficiency? Many proposals can improve the performance of GPUs, but cause an even stronger increase of the power consumption and reduce the energy efficiency of the GPU.

1.7 RESEARCH QUESTIONS

In this thesis, we aim to answer to following research questions:

(A) How can we measure the power consumption of GPUs and kernels running on GPUs?
(B) How can we estimate the power consumption using an architectural simulator?
(C) Can architectural enhancements improve the energy efficiency of GPUs?

Each of these three broad main research questions, generates multiple more detailed questions. For power measurements, we need to answer to following questions:

(A_1) How can we acquire high quality GPU power measurements?
(A_2) How can power measurements be combined with application level event information?

In terms of power estimation and modeling, our main questions are:

(B_1) Can an architectural simulator predict the power consumption of a GPU from its architectural level configuration?
(B_2) How to design microbenchmarks to measure the power consumption of individual GPU components?
(B_3) How can microbenchmarks and power measurements be used to discover unpublished architectural details?

Regarding architectural enhancements, we ask our self the following questions:

(C_1) Which GPU components can we change to improve the power consumption?
(C_2) Can enhancements that improve performance, but also increase power consumption still result in gains in energy efficiency?
(C_3) What kind of architectural enhancements will increase the applicability of GPUs for new applications and still improve energy efficiency?

We provide detailed answers to these questions in the individual chapters of this thesis and a summary in Chapter 12

1.8 THESIS STRUCTURE

After this introduction, the thesis starts in Chapter 2 with a detailed look at GPU architecture with a focus on NVIDIA GPUs. Chapter 3 continues with an overview of the related work.

We aim to improve the power efficiency of GPUs, but before we can start to improve the power efficiency, we first need to understand the power consumption of existing GPUs. We need to be able to measure it and have models that allow us to break down the power consumption into individual parts. In Chapter 4, we describe the requirements for our measurement infrastructure, as well as the development of custom hardware for fast and accurate measurements of the power consumption of both GPUs and System on Chips (SoCs) containing (mobile) GPUs. The chapter also describes, how the raw measurement data can be processed, in order to measure the energy consumption of events such as GPU kernel executions.

Chapter 5 describes the development and validation of the initial GPUSimPow power model. It is based on a combination of architectural modeling as well as measurement-based models. The measurement infrastructure from the previous chapter is used to validate the model as well as for the development of some models.

In Chapter 6, we refine the model by taking data values into account. In this chapter, we show that processed data values significantly influence the power consumption of the GPU and describe a model to accurately estimate the power consumption of the ALUs based on the processed data.

Another refinement is presented in Chapter 7. Here we present an extension to the GPUSimPow model that also considers the effect of data values during memory access.

Armed with the knowledge gained by investigating and modeling the power consumption of GPUs, we developed various optimizations to the GPU architecture that aim to improve the energy efficiency. We start with two chapters describing how the memory and memory interface could be made efficient. Chapter 8 describes an improved encoding scheme for the data transfer between GPU and DRAM, that is able to reduce the power consumption of the interface by up to 6%. The next Chapter describes a technique that reduces the refresh power of DRAM using approximation. This is especially important for mobile applications as refresh power is even consumed while CPU and GPU are in a sleep state.

After we have proposed multiple optimizations of the memory interface we look at the main GPU architecture and try to improve its power efficiency. We start by making the execution of divergent workloads more efficient. In Chapter 10 of this thesis, we look at an alternative GPU architecture called spatiotemporal SIMT, that is able to execute code with branch divergence both faster and more efficiently.

Chapter 11 explains an additional technique called Scalarization. The use of SIMD units and an SPMD programming model often leads to redundant calculations as the threads of a warp often perform calculations with identical inputs and outputs. This leads to a higher power consumption as a calculation is performed 32 times instead of once when each warp consists of 32 threads and it also leads to an inefficient use of the register files, as the results need to be stored for each thread instead of storing the results just once per warp. Scalarization aims to reduce these redundant calculations and stored values. The chapter presents a novel algorithm to identify scalar instructions as well as scalar values in regular GPU kernels automatically and details how Scalarization can easily be integrated into spatiotemporal SIMT and evaluates the performance as well as energy efficiency benefits of Scalarization on this GPU architecture.

Finally, the thesis concludes with Chapter 12. An overview of the results of this thesis is provided, conclusions are drawn and an outlook into research directions for the future is provided.

After this overview of the thesis structure, this chapter ends and we continue the thesis with an overview of GPU architecture in the next chapter.

2

GPU ARCHITECTURE

This chapter provides an overview of current GPU architecture. It focuses on the GPU architectures used by NVIDIA and AMD, as those are the two biggest vendors of discrete desktop GPUs. Less information is available regarding the architecture of embedded mobile GPUs. However, small versions of NVIDIA's Kepler and Maxwell GPU cores were employed in the Tegra SoCs. Different vendors and programming standards employ different terms for the same parts of GPU architecture. This thesis mostly follows the terminology used by NVIDIA's CUDA programming model. This chapter starts in Section 2.1 with a discussion of the programming model employed for (GP)GPU programming. Section 2.2 continues with a top level overview of NVIDIA and AMD GPUs. The core data path and the GPU register file is discussed in Section 2.3. In the following Section 2.4, the GPU memory interface within each core is examined. Section 2.5 delves into warp scheduling and the warp control unit. The DRAM interface is explored in Section 2.6. Finally, a short summary is provided in Section 2.7.

2.1 PROGRAMMING MODEL

GPUs are programmed either with 3D graphics APIs or GPGPU computing APIs. Common 3D graphics APIs are OpenGL (ES), DirectX or Vulkan. Recent version of these API also contain interfaces for GPGPU computing. GPGPU computing uses APIs such as OpenCL or CUDA. In this thesis, we focus on GPGPU computing.

OpenCL and CUDA offer a single program multiple data (SPMD) programming model. In this programming model, the same program is executed on multiple data items, if possible in parallel. As the programming model allows for parallel execution of the different data items but does not require parallel execution, the same program can be executed on different GPUs with different numbers of parallel execution units and the parallelism can be adjusted to match the capabilities of the hardware. CUDA is limited to NVIDIA GPUs, however, OpenCL is an open standard supported on a wide range of GPUs, GPUs, FPGAs and other accelerators. The support of multiple architectures in OpenCL comes at a price: OpenCL is often more limited in capabilities

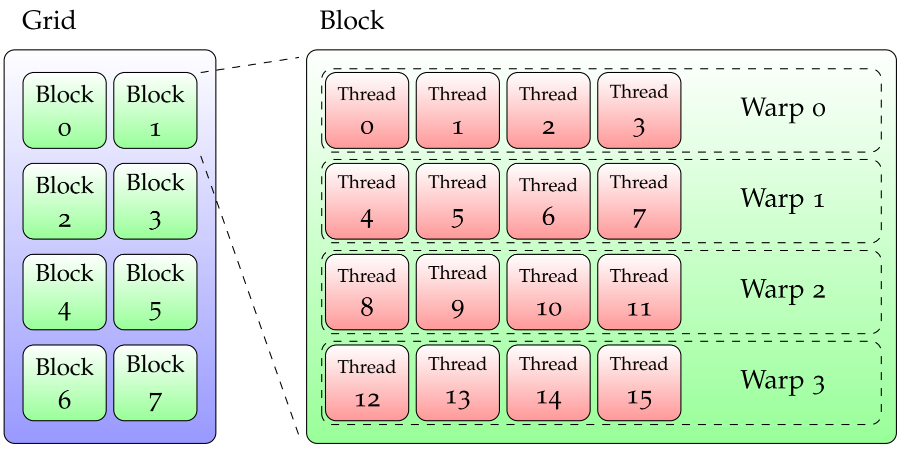


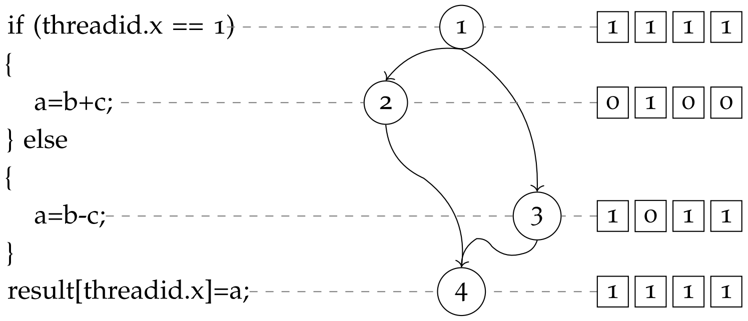
Figure 2.1: Thread Hierarchy in CUDA and OpenCL

than CUDA, because it needs to be efficiently executable on a wider range of architectures and thus can only implement features that can be executed efficiently on all supported platforms.

Both OpenCL and CUDA avoid explicitly launching single threads. Instead, a grid[1] of threads is launched. This concept is further illustrated in Figure 2.1. The grid is composed out of smaller blocks. The total number of threads launched per grid is *blocks* $\times$ *threads per block*. The threads within one block are running in parallel on the same GPU core. The threads from different blocks, however, can be executed in parallel or sequentially depending on the available GPU resources. To execute the threads within each block GPUs use a combination of data level parallelism (DLP) and thread level parallelism (TLP). Each block is broken down into groups of threads called warps. NVIDIA uses 32 threads per warp and AMD uses 64 threads per warp in their GCN architecture and calls them wavefronts.

These warps are executed using SIMD execution units. Despite execution on SIMD units, each thread in the warp is (almost) able to follow its own control flow as if they would be completely independent threads. NVIDIA and AMD GPUs, however, maintain only one program counter (PC) per warp and use predicated execution, an active mask and a special stack to maintain the illusion of independent thread within each warp. NVIDIA calls this execution model Single Instruction, Multiple Threads (SIMT). Figure 2.2 provides a short example of divergent branch execution with SIMT: On the left, the figure shows a small CUDA example, the middle shows the corresponding control flow graph and on the right, the active mask is displayed. The first line is executed on all threads, the active mask is thus one for all threads.

[1]Grid is CUDA terminology, NDRange is the equivalent term in OpenCL



Step	PC / Mask	Stack	Comment
1	1 / 1111	empty	Divergent Branch
2	2 / 0100	[3 / 1011]	
3	4 / 0100	[3 / 1011]	Reconvergence Point
4	3 / 1011	empty	
5	4 / 1011	empty	Reconvergence Point
6	4 / 1111	empty	

Figure 2.2: SIMT execution with Reconvergence Stack

When the branch is executed, first the taken path is executed [33] and the not taken path is pushed to the reconvergence stack [34]. When node 2 is executed, only thread 1 uses the taken path, and only thread 1 is enabled in the active mask. Node 4 is called the reconvergence point. It is the immediate post-dominator of our branch node 1. A post-dominator of a specific node is a node where every path that passed through the node is also guaranteed to pass by [35]. Node 4 is also the immediate post-dominator of node 1, the first node that is a dominator of node 1. When the control flow from node 2 reaches the reconvergence point at node 4, it switches the current PC and active mask to one stored on the top of the reconvergence stack. Node 3 is then executed on all threads but thread 1 and when the reconvergence point is reached again, execution continues for all threads. While the SIMT execution relies on hardware support, it relies on compiler support as well. The reconvergence points are identified at compile time and some branches are replaced by predicated instructions.

AMD uses a slightly different approach: Instead of maintaining a dedicated hardware stack, the stack is stored in regular registers. Always executing taken branch first can require up to one entry per thread in the stack for nested branches. AMD reduces the number of stack entries needed by always executing the path with fewer (or equal) active threads first [36]. With this optimization, each time a new stack entry is pushed to the stack, the number of active threads is at least halved. When only a single thread is active,

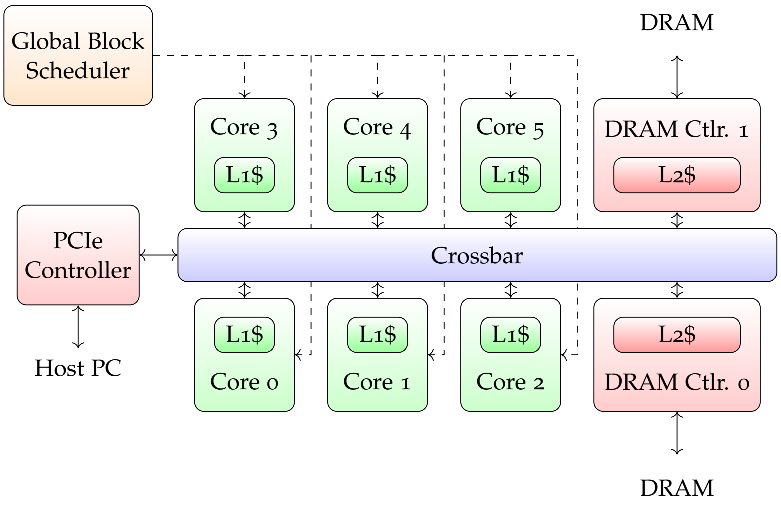


Figure 2.3: GPU Architecture Overview

divergent branches are not longer possible and no deeper reconvergence stack entries can be created. After the GPU has executed the path with fewer active first and has reached the reconvergence point, the stack entry is removed. Remembering the already executed path is not required and thus does not require a reconvergence stack entry. As AMD uses a warp size of 64 threads, this limits the maximum number of stack entries to $log_2 64 = 6$ which is significantly lower than the 64 entries that would be required without this optimization.

2.2 TOP-LEVEL ARCHITECTURE

A high-level overview of a typical GPU architecture is shown in Figure 2.3. Several cores, in this example six, are connected to a crossbar that links the cores to the memory controller and PCIe controller. Each core contains a small L1-cache and each memory controller contains a part of the L2-Cache. Both NVIDIA and AMD report thousands of cores in their documentation, e.g.: Geforce GTX1080 GPU is reported to have 2560 "CUDA Cores" and the Radeon RX Vega 64 spots 4096 "stream processors". However, these cores do not feature the elements normally expected in a core, e.g.: they do not have their own control logic or caches. Each of these "cores" is a single precision floating point ALU. Together with the clock frequency can be used to calculate peak single precision flops. In this thesis core refers to what NVIDIA calls "streaming multiprocessor" or AMD calls "compute unit" (CU): Mostly self-contained cores with multiple floating ALUs configured as one or multiple

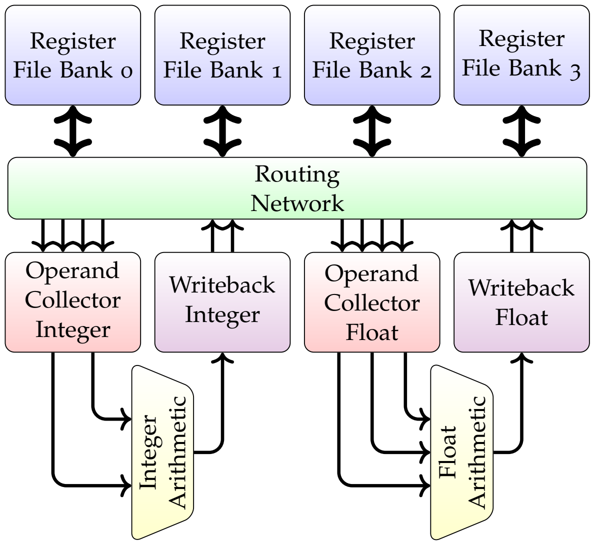


Figure 2.4: GPU Datapath [39] © 2016 IEEE

SIMD execution units, the required interface to the memory interconnect and various caches. AMD's RX Vega 64 contains 64 of these compute units and NVIDIA's GTX1080 contains 20 "streaming multiprocessors" (SM). Each SM contains 128 single precision floating point ALUs and each of the CUs contains 64 floating point ALUs in their SIMD units.

Outside the of the cores, a global block scheduler distributes new work to the cores if both additional work and enough resources on the core are available. GPGPU-sim models this scheduler as a simple round-robin scheduler, however, the actual behaviour is slightly more complex and has advantages in terms of locality [37], [38]. The PCIe controller allows DMA transfers and direct access from the host PC memory to the GPU DRAM. It also enables the host PC to submit new work to the GPU and query the execution status of already submitted work.

2.3 GPU CORE DATAPATH & REGISTER FILE

A simplified GPU data path is shown in Figure 2.4. While CPUs often used multi-ported memory for register file to allow fetching several operands in the same cycle, GPUs often used several register banks with only one or two ports each. Typical GPUs are required to store the register content of a high number of threads. Often hundreds or thousands of threads per core. As each thread typically uses 16 to 128 32-bit registers, this results in a requirement of several MB or at least hundreds kB storage in the register files. The total amount of storage in the register files of a GPU can reach more than 10 MB [40]. Typically, GPUs do not use a fixed number of architectural registers but can configure the numbers of registers per thread within a certain range such as 32 to 128 registers. While the total amount of registers per core is fixed, this flexibility allows developers to trade a higher number of registers per threads for a lower number of active threads or vice versa. It is often assumed that code with a higher number of active threads (also called occupancy) enables higher performance. However, in some applications using more registers per thread can enable code that exhibits more ILP. This can lead to a higher performance, even with a smaller number of active threads [41]. Autotunning can be used to find the optimal trade-off between the number of registers per thread and the number of concurrent threads [42].

As multi-ported memory requires large amounts of area per storage bit, GPUs simulate multi-ported memory by using several banks of memory with only one or two ports each and sequentially fetch the required operants over multiple cycles. Several possible variants of this technique are explained an NVIDIA patent [43]. An operand collector collects the operands required for each instruction over several cycles and stores them until all operands for one instruction are available. When all operands are collected, the instruction will be issued to the execution unit. In the example shown in Figure 2.4 four register file banks are used. A routing network, e.g. a crossbar, connects the register file banks to the operands collectors. The number of register file banks and the ports per register file determine, how many operands can be read and written from the register file in each cycle. This requires that the operands are evenly distributed to all register file banks. Different schemes with various advantages and disadvantages exist to map registers from the individual threads to different register file banks. As the GPU register file only simulates multi-ported memory but is not a truly multi-ported register file, all mapping schemes can yield an effective register file bandwidth significantly below the peak register file bandwidth, if the requested operands are not evenly distributed to the banks.

As the register file uses a significant part of the area and power budget of a GPU, many authors have described improvements to reduce the area and/or power consumption of the register file. Wing-Kei et al. proposed the use of a hybrid SRAM-DRAM register file [44], other authors proposed the use of STT-RAM [45] or racetrack memory [46]. Gebhart et al. proposed the use of multi-levels of compiler managed register file caches [47] and unifying the register file with the first level cache and shared memory [48]. NVIDIA's Volta architecture uses small compiler managed register caches [49]. Lee et al. proposed the use of compression to increase the effective size of the register file [50].

In the figure, two execution units are shown, an integer arithmetic unit and a floating arithmetic unit. Several common executions units are not

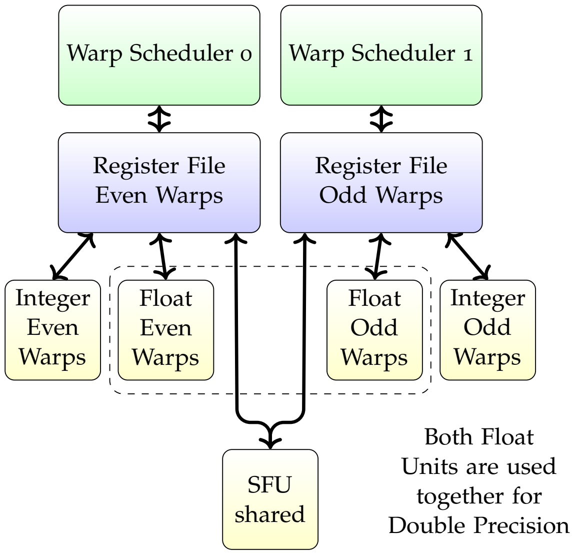


Figure 2.5: Fermi Datapath [39] © 2016 IEEE

shown in the figure. A load/store unit provides the interface to the memory interface, it will be discussed in Section 2.4. GPUs commonly also provide execution units, often called special function units (SFU), for more complex functions such as trigonometric functions, reciprocals or square root. These SFUs were designed for use in graphical applications and are fast but often do not provide full single precision accuracy. They are used via special intrinsics or via a `-use_fast_math` compiler flag that trades accuracy for speed. As these instructions are not used as often as basic arithmetic instructions, the throughput of these units is often significantly lower than the throughput of the regular floating point units. As GPU workloads are typically heavy in floating point operations, GPUs often offer a higher single precision floating point throughput than integer throughput. The ratio of double precision throughput to single precision varies strongly between different GPUs. Some models aimed at high performance computing offer a 1:2 ratio [2] while other models aimed at the graphics market offer a meager 1:32 ratio [3].

Figure 2.5 shows a simplified version of the datapath used in NVIDIA's Fermi architecture. In Fermi every core contains two warp schedulers each responsible for one-half of all warps assigned to the core. Each of the warp schedulers has its own register file and some execution units that are only usable for the warps assigned to that warp scheduler, but the SFU unit is shared between the two warp schedulers and both schedulers can submit

[2]e.g.: Tesla P100, V100
[3]e.g.: GTX1070, GTX1080

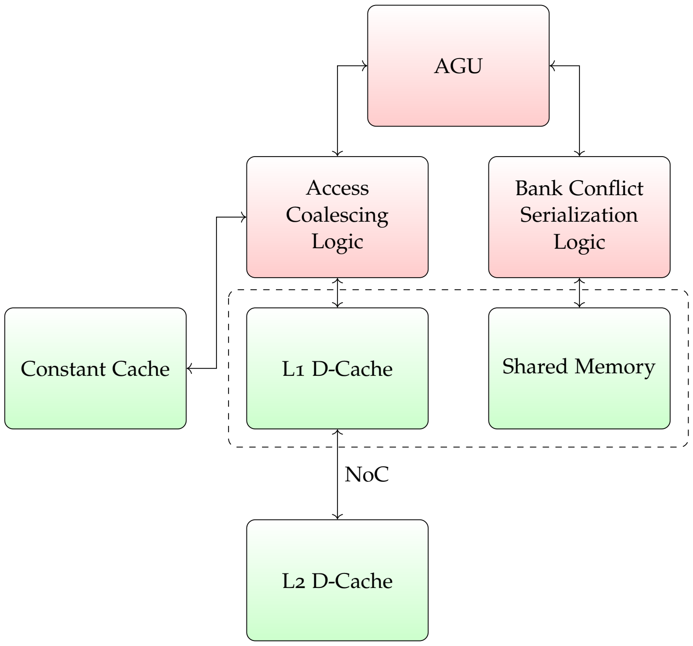


Figure 2.6: High-level Overview of Load/Store Unit

instructions to this shared SFU. In NVIDIA's Kepler architecture we also notice a similar kind of mix of execution units that partially shared and partially exclusive to a part of the warp. In NVIDIA's Maxwell architecture the amount of shared execution units is reduced but each core contains four warp schedulers. Shared units in Maxwell are limited to memory and texture access, which makes it easier for the compiler to schedule the instructions as it removes the non-determinism caused by the interaction of several warps schedulers sharing the same execution units. Some parallels can also be drawn to AMD's Bulldozer CPU architecture that is composed of multiple modules and each module implements two cores and shares the FPUs between two cores.

2.4 MEMORY INTERFACE

A high level overview of a GPU Load/Store (LDST) unit is shown in Figure 2.6. An address generation unit (AGU) generates one address per active thread in the warp. Normal DRAM requests are sent to the address coalescing logic which tries to bundle these requests into DRAM transactions. If the threads try to access nearby accesses the number of DRAM transactions is much smaller than the number of active threads, as each DRAM transaction typically loads or stores at least 32 consecutive bytes at a time. This depends on the width of each DRAM channel (usually 32 or 64-bit per channel) and the burst size of the employed DRAM technology. The coalesced requests are then submitted to the L1 data cache within the core or a special constant cache that caches memory values that are constant during the kernel execution and can only be changed before the kernel is launched. If values are not contained in the respective local cache they are forwarded through the interconnection network to the memory controller that is responsible for the addressed part of the memory and are fetched from DRAM or L2 cache. Shared memory accesses are sent to a bank conflict serialization logic. Shared memory is a small internal memory used for exchanging data that is shared between the different threads. As the shared memory is a local part of each GPU core, it offers energy efficient, high bandwidth and low latency storage. The shared memory is local to each GPU core, so long wires are not required to connect the memory and because it can only be accessed by locally running threads and is not mapped into the address space of threads running on different cores, no cache coherency or MMU is required. Because it can only be accessed by threads running within the same core, it is only useful for facilitating intra-block cooperation of the threads. In some NVIDIA GPUs, the shared memory reuses parts of the L1 cache and allows programmers to choose different configurations such as 16 KB shared memory plus 48 KB of L1, 32 KB+32 KB or 48 KB shared and 16 KB L1.

Shared memory instruction allows each thread in a warp to access a different location in the shared memory. Depending on the SIMD width of each GPU core a naive implementation would require a multi-ported memory with 8 to 32 ports. As this is not feasible within a reasonable area, a similar architecture to the register file is used: Multiple single ported banks are used and requests are serialized over several cycles if required. This is shown in Figure 2.7. A conflict checker compares the bank addresses of the incoming requests and selects a bank conflict-free subset to send to the address crossbar. If not all requests could be handled in the same cycle, this is repeated in the next cycle until the requests from all threads were executed. An address crossbar is used to direct the address from each thread to the SRAM banks. A data crossbar is used to direct the data from or to the SRAM banks to the right thread. Performance counters exposed by the NVIDIA profiler show that the generation of the additional request is implemented using a replay mechanism. Instructions are resent to the load-store unit until the requests of all active threads could be handled.

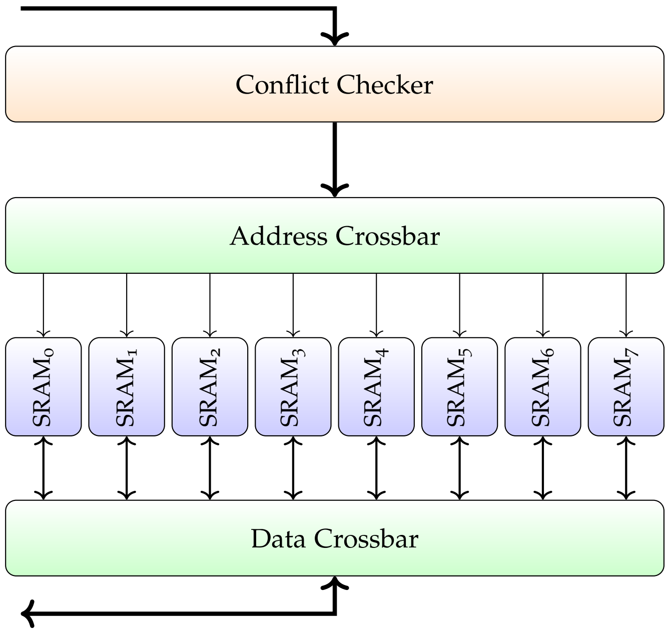


Figure 2.7: GPU Shared Memory

2.5 WARP CONTROL UNIT

The Warp Control Unit (WCU) shown in Figure 2.8 is responsible for scheduling and managing the warps. It is a key part of the GPU architecture. Within one warp, GPUs issue the instructions strictly in program order. GPUs can, however, dynamically switch between different warps. This scheme often provides a high tolerance against long memory latencies. The scheme also partly explains the large register files. Little's law, shown below, links throughput, latency and concurrency [51], [52].

$$\textit{mean concurrency} = \textit{mean latency} \times \textit{throughput}$$

A high memory throughput, together with a high latency means that many concurrent memory accesses are required. With the current GPU architecture, many concurrent memory accesses requires many concurrent threads, as every thread can only trigger a few concurrent transactions. If many concurrent threads are required, then thread context storage is also required for all of them.

The warp fetch schedule logic selects a warp with space in the instruction buffer and fetches one or multiple instructions from the instruction cache into the instruction buffer. As it only fetches the instructions but does not schedule

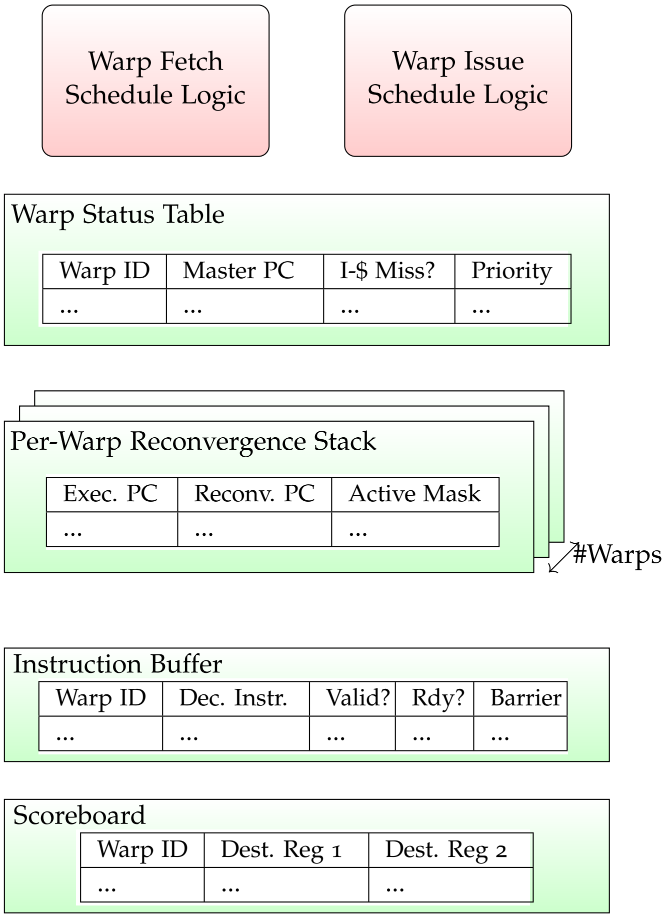

Warp Fetch Schedule Logic
Warp Issue Schedule Logic
Warp Status Table
Warp ID | Master PC | I-$ Miss? | Priority
... | ... | ... | ...
Per-Warp Reconvergence Stack
Exec. PC | Reconv. PC | Active Mask
... | ... | ...
#Warps
Instruction Buffer
Warp ID | Dec. Instr. | Valid? | Rdy? | Barrier
... | ... | ... | ... | ...
Scoreboard
Warp ID | Dest. Reg 1 | Dest. Reg 2
... | ... | ...

Figure 2.8: High-level Overview of Warp Control Unit

the execution the logic does not need to evaluate whether data dependencies of the instruction are met or not. It simply looks at the instruction buffer and if a slot is empty and the warp is not yet finished, it fetches the next instruction for the warp that owns the empty slot of the instruction buffer.

The warp issue scheduler is responsible for issuing warps from the instruction buffer to the execution unit. Before the instruction is issued, the scheduler checks that the instruction is ready to be issued or if conflicts with in-flight instructions prevent the instruction from being issued at the moment. If multiple instructions are ready for issue, the warp issue scheduler can use a simple round-robin scheme or more sophisticated techniques, e.g.: some proposed schedulers optimize for better memory locality [53], [54]. Lee and Wu propose a scheduler that reduces tail effects [55]. Xu and Annavaram propose a scheduler to optimize power gating [56].

Many NVIDIA GPUs use scoreboards for checking which instructions are ready for scheduling, newer NVIDIA GPUs use a combination of scheduling hints generated by the compiler and scoreboarding for long latency operations [57]. AMD GCN GPUs also do not fully check all instruction dependency and force the compiler to insert NOPs or reorder instructions in some cases. In other cases, a special wait instruction needs to be used. It checks the number of outstanding long latency memory requests and waits if above a compiler-determined threshold [36]. These compiler-aided schemes reduce the hardware required for dependency checking and improve the energy-efficiency.

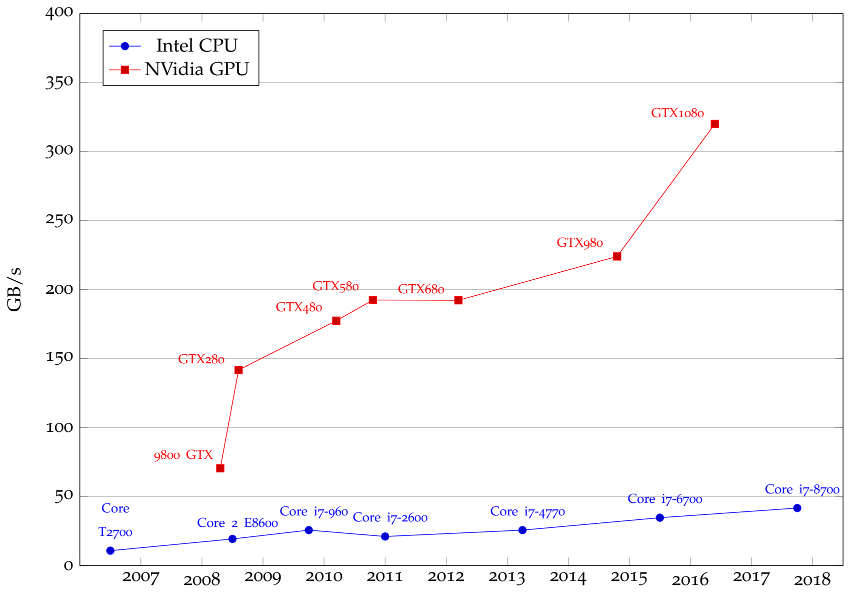


Figure 2.9: CPU vs. GPU Peak Bandwidth

2.6 DRAM INTERFACE

With thousands of threads active at the same time and often very large working sets, large caches are ineffective for GPUs. GPUs also feature very high arithmetic throughput. Due to these factors GPUs need significantly higher bandwidths than CPUs. At the same time GPUs usually do not need as much memory capacity as CPUs: Rarely used data can quickly be uploaded via PCI express when required. GPU memory is typically not used to cache disk content or large databases.

Due to these reasons GPU memory interface typically have different characteristics than CPU memory interfaces. Figure 2.9 shows the peak bandwidth of various CPU and GPU memory interfaces. The bandwidth provided by GPU is significantly higher. Even the slowest and oldest GPU provides a significantly higher memory bandwidth than the newest and fastest CPU listed in the chart. The fastest GPU in chart features almost 8 times the memory bandwidth of the fastest CPU. Figure 2.10 lists the bandwidth per data pin, it can be seen that the higher bandwidth is partly provided through the use of faster signaling standards. NVIDIA's GTX1080 uses a GDDR5X interface at 10 Gbps, while the Core i7-8600 uses DDR4 at 2.66 Gbps. The signaling employed by the GPU is almost 4 times faster.

A wider interface explains the remaining gap in the peak bandwidth: The interface used by the CPU is 128-bit wide, while the GPU uses a 256-bit

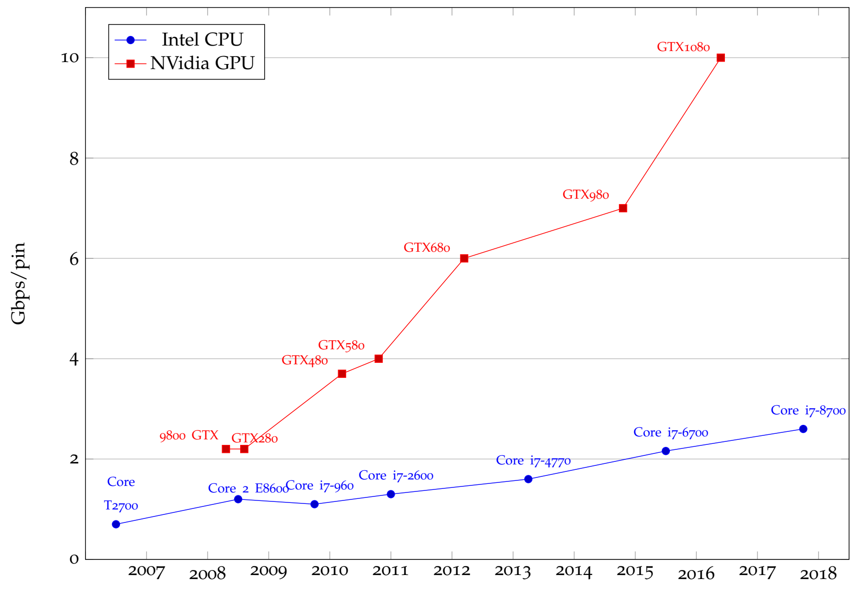


Figure 2.10: CPU vs. GPU Bandwidth per Pin

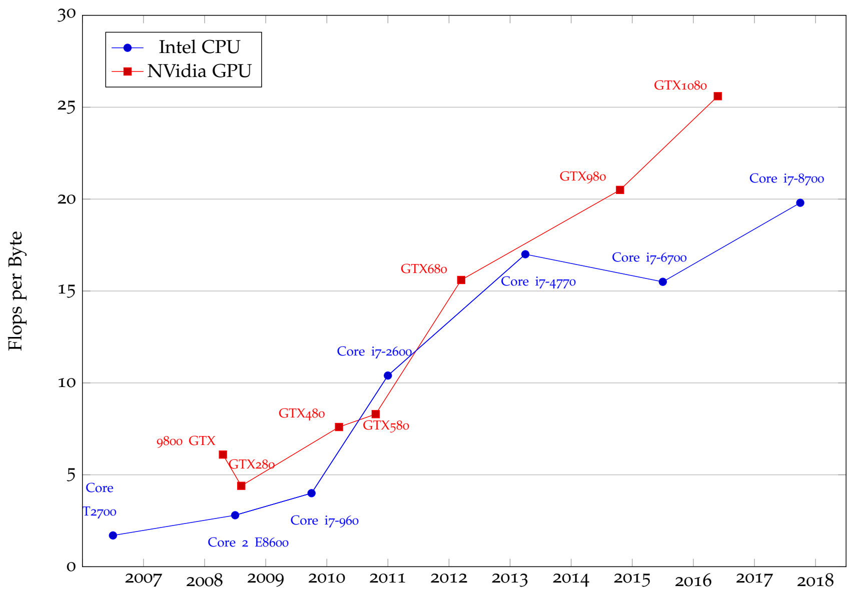


Figure 2.11: CPU vs. GPU Flops per Byte

wide interface. But the CPU memory interface also has many advantages: The CPU can use up to 64 GB of memory and memory can be extended or replaced using memory modules, while the GPU has 8 GB of GDDR5X directly soldered to the PCB and memory cannot be extended or replaced. GDDR5 [58] and GDDR5X [59] are also significantly more expensive per storage byte than DDR4 [60].

Connectors such as the DIMM sockets used by CPUs create crosstalk and EMI [61]. CPUs commonly allow the connection of multiple memory chips to each data pin, however, the resulting multi-drop bus the parasitic capacitances of connectors can distort the signal [61]. Soldering memory to the PCB instead of using sockets and using point-to-point connections instead of multi-drop bus provides a better signal quality and thus allows for higher data rates. However, it also prevents memory upgrades and high memory capacity.

However, despite the high bandwidth in absolute terms, GPU memory bandwidth is often a significant bottleneck, as we also need to consider the very high computational throughput of GPUs. Figure 2.11 shows the flops per byte for various GPUs and CPUs. For both CPUs and GPUs, peak performance is growing faster than memory bandwidth. Applications on both CPUs and GPUs must compute many floating point operations per byte to avoid being limited by memory bandwidth. NVIDIA's GTX1080 needs to perform 25.6 or more floating point operations per byte in order to avoid being limited by memory bandwidth. Despite having only 1/8 of the memory bandwidth, the Core i7-8600 only needs 19.8 flops per byte.

To solve this bandwidth shortage some GPUs employ high bandwidth memory (HBM) [62]. HBM and its successor HBM2 use a silicon interposer and through silicon vias (TSV) to include the DRAM in the same package as the GPU and offer a very wide interface [63]. The very wide interface possible due to the interposer and TSVs allows reducing the per pin bandwidth to reduce the power consumption and simplify the required drivers, receivers and clocking circuits while still providing a bandwidth increase. AMD's Radeon R9 Fury X GPU uses a 4096-bit wide interface to provide 512 GB/s bandwidth while running at just 1 Gbps per pin. GDDR5 interface width is typically limited to a maximum of 512-bit, with 256-bit wide interfaces being more common. NVIDIA's (very expensive) Tesla V100 reaches 900 GB/s using HBM2 and a 4096-bit wide interface [64].

Beside increasing the actual memory bandwidth, compression and tile-based rendering are often used techniques for increasing the effective bandwidth, mainly for 3D rendering. Tile-based rendering was originally proposed for parallel rendering but is also highly useful for reducing the required memory bandwidth [65]. Both lossless and lossy compression techniques have been used to reduce the required bandwidth. Textures are often stored using lossy compression algorithms [66], [67]. Z-Buffers often use a hierarchical compressed storage to reduce the required bandwidth and allow the early z-rejection of blocks of pixels [68], [69]. However, while the use of memory compression is common for 3D rendering, compression is usually not used in GPGPU applications. In a paper, co-authored by the author of this thesis, Lal et al. proposed E^2MC as a memory compression technique for GPGPUs [70]. As already mentioned above, Lee proposed a compressed register file [50], Vijaykuma et al. propose a clever hardware/software solution, where unused compute resources are used to provide memory compression [71]. Pekhimenko et al. describe a compression technique that aims to reduce the interface energy by reducing the number of bit toggles [72]. Rhu et al. propose a specialized DMA based compression for deep learning, one of the most common GPGPU applications [73].

2.7 SUMMARY

This chapter provided an overview about current GPU architecture. We described the programming models employed by GPGPU APIs such as CUDA and OpenCL. We explained how threads are bundled together into warps and how branching is enabled while threads are executed on SIMD execution units. We illustrated how the GPU architecture focus differs from CPU architecture by focusing on throughput instead of latency. The use of multiple warps and warp scheduling to compensate for memory and arithmetic latency was described as well as the large GPU register files, shared memory and the high bandwidth external memory interfaces used by GPUs. The next chapter provides a survey of related work for the entire thesis.

3

RELATED WORK

This chapter reviews work related to this thesis. We start with additional introductions into GPU architecture in Section 3.1. Section 3.2 lists related work for GPU power measurements and Section 3.3 continues with architectural GPU power modeling. Section 3.4 takes a look at work linked to data-dependent power modeling. The second half of the chapter proceeds with literature related to the architectural enhancements proposed in Chapters 8 to 11. Section 3.5 reviews papers related to the proposed memory interface power reduction encoding. We then cover work related to our refresh power reduction technique in Section 3.6. After these two memory-related sections, the next Section continues with (Spatio-)Temporal SIMT related research. The related work chapter finishes with Section 3.8 related to Scalarization.

3.1 GPU ARCHITECTURE

In addition to the introduction found in Chapter 2, many books and papers provide further insight into GPU architecture. While we cannot list them all here, we selected a list of works we found especially valuable for the readers of this thesis. Kirk and Hwu's book provides a good history of GPUs as well as a great introduction into CUDA programming [74]. The 5th edition of Hennesey and Patterson's classic book was updated with an excellent chapter on "Data-Level Parallelism in Vector, SIMD, and GPU Architectures" that helps to put GPUs into perspective [75].

The GPGPU-Sim manual describes the GPU architecture modeled by the popular simulator and can be understood as a best guess description of NVIDIA's Tesla and Fermi architecture GPUs [37].

Both NVIDIA and AMD provide many white papers on the architecture of their GPUs (e.g.: [57], [76]–[78]), however, these descriptions often lack detail and target programmers instead of architects.

AMD provides many details for their GCN architecture GPUs as part of their "GPU Open" initiative and their support for open source GPU driver development. Information released by AMD includes a complete documentation of the instruction set of their latest GPUs as well as an overview of the architecture of the whole GPU and the individual GPU cores in particular [36].

3.2 POWER MEASUREMENT

Some related work [79], [80] employed commercial wall-plug power meters. These meters are inserted between the PC power supply and the power outlet. They measure the power consumption of the whole PC, instead of measuring only the power consumption of the GPU. Hong and Kim [79] assume the GPU power can be calculated by measuring the power of the entire PC under load and subtracting the power of the PC in idle state. This assumption yields high inaccurate results, because the power used by the remaining PC components is usually not constant and the measurement results will include power supply losses. Large bypass capacitors inside the power supply prevent the accurate measurement of power for kernels which run fewer than 50 ms.

Ma et al. [80] tried to solve the first issue by using a second ATX power supply powering only the GPU but this approach ultimately suffers partially from these issues as well because the GPU also receives parts of their power supply from the PCIe slot and these supplies are provided by the same ATX power supply that also powers the mainboard, CPUs, memory and other non-GPU components.

Other papers [81]–[83] use improved measurement methodologies but still exhibit multiple limitations. These published methodologies either fail to measure all power sources, e.g. do not measure the power provided via the graphics card slot [81], measure only current and assume constant voltages [82], or use low sampling frequencies that prevent them from measuring short-term power variations [81], [83].

Burtscher et al. explain how power sensors included in some GPUs can be used to measure the power consumption, however, these sensors lack the resolution and sampling rate of the measurement testbed presented in this thesis[84].

3.3 ARCHITECTURAL GPU POWER MODELING

For general GPU power modeling, the available body of previous work was rather small, when the work in Chapter 5 was initially published. On the one hand, there have been approaches such as the ones from Hong and Kim [79] or Ma et al. [83] which are based entirely on measured data. While this type of power model is able to deliver superior accuracy for the architecture it was built from, it lacks the capability to make accurate predictions about GPUs with other architectural parameters and designs. On the other hand, several researchers have built purely analytic power models, such as Ramani et al. [85]

and Wang [86]. While such approaches generally show a strong correlation between different simulated and hardware GPU architecture configurations, they typically cannot provide reasonable absolute accuracy due to the lack of either industrial or measured anchor data. Our power simulator improves upon all these prior approaches by *combining* both empirical and analytical component models to create a system that is both architecturally flexible and shows reasonable absolute accuracy. A similar approach to ours has previously been used to estimate CPU power consumption by the well-known McPAT tool [87].

As already explained in the previous section, many GPU power modeling papers made strong assumptions about the hardware they measure, leading to inaccurate measurement methodologies. After this work was initially published, several other papers presented similar studies. Leng et al. presented their power simulator GPUWattch [88]. Just like GPUSimPow, it combines gpgpu-sim with a power model based on an extended version of McPAT. It also uses a good GPU power measurement, similar to the measurement testbed used for GPUSimPow and a clear improvement compared to the older papers listed above. GPUWattch also relies on analytical models for modeling the energy consumption of many GPU components. The two simulators differ in how they employ microbenchmarks. GPUWattch provides power models for all main GPU components based on CACTI, McPAT or Synopsys Power Compiler. It then uses microbenchmarks in a refinement stage to rescale the outputs of the power models. For increased accuracy the outputs of the analytical models are multiplied with the refinement factors estimated in the refinement stage. The use of these factors has been criticized to render results of the analytical model to be "mathematically irrelevant" [89]. GPUSimPow does not use a refinement stage but instead models some components using models calibrated with microbenchmark based measurements. Lim et al. also describe a power simulator for GPUs based on an extended McPAT and gpgpu-sim [90]. The authors list a different focus of the paper as the main difference to GPUWattch. Diop et al. describe a power model for heterogeneous processors containing GPUs [91].

Since its release, several papers have used GPUSimPow for estimating area and/or power consumption of GPUs. Libuschewski et al. used GPUSimPow inside a multi-objective energy optimization framework to estimate the energy used by a mobile GPU in a mobile biosensor application [92] as well as for design space exploration for mobile GPGPUs [93]. Sankaranarayanan et al. used GPUSimPow to evaluate the energy benefits of a modification to the GPU memory hierarchy they propose [94]. Nath et al. employed GPUSimPow to estimate the static power consumption of various GPU configurations to build a model of DVFS in GPUs [95]. Dhar and Chen used GPUSimPow for estimating the area of a GPU core and its static power consumption [96].

3.4 DATA-DEPENDENT POWER MODELING

Many popular architectural power simulators for CPUs and GPUs ignore the data values processed by the datapath. McPAT's CPU core power model [87], for example, counts various register reads, uses of integer and floating point ALUs, integer multiplies, register renaming, but neither the exact instruction type nor any statistics about the processed data values are used to predict the power consumption.

GPUWattch [88] and GPUSimPow [97] are power simulators for GPUs. Both simulators are based on gpgpu-sim and extend it with a McPAT-based GPU power model. These architectural simulators count activity factors for various GPU units and use them to estimate the power consumption of the GPU. Counted activities are integer or floating point instructions, register reads and writes, memory accesses, etc. None of the used activity factors measures how often datapath lines switch between 0 and 1.

One exception to ignoring data values is the original Wattch power simulator [98]. Wattch contains a DYNAMIC_AF option in its source code that collects activity factors based on average population count of the processed values, but this is only used for some internal buses and memories but not for the ALU. In fact, the Wattch source code contains the comment: "FIXME: ALU power is a simple constant, it would be better to include bit AFs and have different numbers for different types of operations". The Wattch authors apparently recognized that this was a weak point in their simulator. For a CPU power simulator where control logic dominates the datapath, using such a simple model might still be acceptable, but accelerators such as GPUs try to keep the control logic small and simple and use large parts of their power and area budget for execution units and register files. In Chapter 6, we will show that for these accelerators more accurate power models are required.

Kim, Austin, Mudge and Grunwald [99] also recognize that architectural power simulators ignore values and memory addresses in their power estimation. They describe how an architectural simulator for CPUs that considers values could be built and developed a prototype based on SimpleScalar but did not validate their model.

Adhinarayanan measures and models the data depend interconnection power on an AMD GCN GPU with OpenCL [100]. Our work uses a NVIDIA Fermi GPU with CUDA and also measures the external interface.

Some related work exists for microprocessors. Sarta, Trifone and Ascia propose a data dependent power model for a simple DSP with a 2-stage pipeline [101]. They find that operands strongly influence the energy consumption and also employ linear least square fitting. Kerrison and Eder [102] model the energy consumption of a hardware multi-threaded microprocessor. They consider the overhead of switching from one instruction to another and the influence of data values on energy consumption.

3.5 OPTIMAL DBI ENCODING

Hollis [103] described the DBI DC and DBI AC schemes and recognized that both the number of transmitted zeros and the number of signal transitions are important for the power consumption of the memory interface. The slight increase of the signal transitions in DBI DC and the slight increase of transmitted zeros in DBI AC was also described in the same paper. Hollis proposes to combine DBI AC and DC by switching between DBI DC and DBI AC encoding modes. The proposed DBI ACDC scheme encodes the first byte of a group of bytes using DBI DC and then encodes the remaining bytes using DBI AC. We found that this scheme indeed provides a slight improvement compared to pure DBI AC. However, the encoding proposed in this thesis outperforms the DBI ACDC scheme. In this thesis, we assume that all lines transmitted ones prior to transmitting the evaluated burst. Due to this boundary condition, DBI AC performs identically to DBI ACDC in our evaluation.

Chang et al. [104] propose schemes that aim to reduce both zeros and transitions per burst. However, instead of finding the minimal energy encoding for each burst, they propose heuristic schemes that find good but not necessary optimal encodings.

In a patent, Hollis [105] proposes a technique to target both signal transition and zeros. This technique uses additional signal lines and requires a different and more complex decoding process than regular DBI schemes.

Ihm et al. propose an analog circuit for DBI DC encoding [106]. Analog implementation could also reduce the overhead of the technique proposed in Chapter 8 and DBI encoding seems to be well suited for analog implementation as rare inaccurate encoding decision are unlikely to causes application errors.

Stan and Burleson [107] provide theoretical background on DBI encoding, however, they only consider the reduction of signal transition and do not consider the reduction of zeros.

Narayanan et al. [108] describe additional coding schemes that can reduce the number of signal transitions beyond DBI, but require an even higher number of lines and more complex encoding and decoding.

Kim et al. describe DBI DC in GDDR4 and show how it reduces simultaneous switching output noise [109].

Pekhimenko et al. describe a toggle-aware compression scheme for GPU [72]. This scheme reduces signal transition by choosing data representations with reduced toggling.

3.6 SPARKK

Among others, Jamie Liu et al. [110] recognized that most DRAM rows do not contain high leakage cells and thus can tolerate lower refresh rates. Most cells retain their data for a much longer time than the short regular refresh period, that is required for error-free storage [111]. They proposed a mechanism named RAIDR to refresh these rows at a lower rate. Because different refresh periods cannot be achieved with the conventional DRAM internal refresh counters, they add a refresh counter to the memory controller. This refresh counter runs at the rate necessary to meet bit-error free operation, including rows that contain cells with high leakage. The memory controller then generates activate and precharge commands to manually refresh the rows. Manual refresh cycles are skipped if the memory controller determines that they are not necessary. The manual refresh by the memory controller needs slightly more power per refresh operation than the standard auto refresh as the row addresses need to be transmitted to memory. But the authors show that the power saved by the reduced refresh frequency outweighs the power consumed by the more complex refresh signaling. The idea of a memory controller managed refresh and memory controller internal row counter is also used in Chapter 9. In RAIDR, the memory controller uses bloom filters to classify row addresses into different refresh bins. Depending on a row's refresh bin, the memory controller issues an actual refresh command only every fourth, second or every time the refresh counter reaches it. This scheme results in rows getting refreshed at 256, 128 or 64 ms periods depending on their refresh binning. Our proposal extends RAIDR by introducing additional refresh bins for approximate data.

Song Liu et al. propose Flikker [112], a memory area with reduced refresh periods for non-critical data. They propose a modified DRAM to enable longer refresh periods on a part of the DRAM. The authors of RAIDR recognized that their work can be combined with the Flikker proposal of Liu et al. Our approach can be seen as an extension of Flikker. It provides an additional reduction of refresh activity for non-critical data. Different from the storage area proposed by Liu et al. this storage area uses varying refresh periods for different bits based on their importance.

Ware and Hampel proposed threaded memory modules [113]. In a threaded memory module, a DRAM rank is split into multiple subranks. The subranks have separated chip select (CS) lines but otherwise share the address and control signals. The CS line controls whether the DRAM chip reacts to transmitted commands or ignores them. By providing multiple CS lines instead of a single CS signal per rank, commands can be issued to a subset of the DRAM rank. Ware and Hampel list various advantages, such as finer granularity transactions, higher performance and lower power consumption. Our proposal also relies on subranks, but uses them to provide bits with different energy/error rate trade-offs simultaneously.

Sampson et al. worked on approximate storage in multi-level cells in Flash or PCM [114]. They recognized that storage bits from one cell have different levels of reliability and errors can be minimized with their striping code. This striping code is very similar to the permutation proposed in Chapter 9.

After this work was initially published several additional related articles were published. Raha et al. performed an experimental study and found that the quality of the different DRAM pages differs significantly, they propose to sort the pages into different quality levels and allocate data according to their error tolerance [115]. Ranjan et al. suggested an approximate STT-MRAM with different configurable quality levels for different bit groups [116]. Jung et al. showed that it is often possible to omit the refresh completely [117]. Chen et al. suggested to use different supply voltages for the various bits with an approximate SRAM [118].

3.7 temporal simt and spatio-temporal simt

Work related to temporal SIMT can be grouped into two categories: First, work describing temporal SIMT, second, studies focusing on efficient execution of branch divergent codes.

temporal simt The general idea of temporal SIMT execution is described in an NVIDIA patent by Krashinsky [119], but the details provided are insufficient to derive an implementation and furthermore no performance evaluation is included. In the academic world, TSIMT has also been mentioned by Keckler et al. in a paper that describes that moving data across the chip is more energy-consuming than actual computation [120]. This paper introduces the *Echelon* GPU architecture that offers TSIMT execution as well as many other features. The authors mention the potential benefits of TSIMT for branch divergent applications, but does not present an implementation or an evaluation of Echelon.

branch divergence A large body of work has been performed on how to improve GPU performance when there is control divergence. Many techniques such as *Dynamic Warp Formation* [121], *Thread Block Compaction* [122] and *Large Warp Microarchitecture* [123] reorder threads from multiple warps into fewer warps with more active threads per warp. All these techniques keep the spatial SIMD property: All lanes execute the same instruction at the same time, but differ in when and how threads are reordered. Furthermore, because these techniques reorder threads between warps, memory divergence can increase and correctness problems for applications that rely on warp-level synchronization can arise.

Another, related technique called *Simultaneous Branch and Warp Interweaving* (SBWI) was introduced by Brunie et al. [124]. They proposed enhancements to the GPU's microarchitecture to *a)* co-issue instructions from two different branch paths and *b)* co-issue instructions from different warps to the same SIMD unit. Interestingly, this architecture shares a property with TSIMT, namely that the different lanes do not need to share the same instruction. In SBWI, however, only two different instructions can be executed at the same time, while in TSIMT each lane can execute a different instruction.

A common disadvantage of the techniques described above is that they can only improve performance when the active mask meets certain conditions. One of the reasons for these conditions is that the individual contexts of the threads that run on the GPU are stored in a specific part of the GPU's register file [125]. As a result, it is generally not possible to freely swizzle and re-group threads for execution. TSIMT takes a different approach that avoids this problem almost entirely at the expense of higher issue throughput requirements.

Vaidya et al. proposed an architecture in which 16-wide SIMD instructions are executed over multiple cycles on 4-wide SIMD units [126]. Two techniques are proposed to accelerate execution when only a subset of threads is active: Basic Cycle Compression (BCC), where SIMD subwords are skipped if no thread is active, and a more costly but also more powerful technique called Swizzled Cycle Compression (SCC) that employs crossbars to permute the operands prior to compaction to enable a more efficient compaction.

Lee et al. group different possibilities for data parallel accelerators into five different groups: MIMD, Vector-SIMD, Subword-SIMD, SIMT and Vector-Thread (VT) [127]. TSIMT can be seen as another variant. The programming model is MIMD, but the execution units are similar to density-time vector lanes [128]. The lanes share the same instruction fetch and decode frontend but are not bundled in groups that execute the same instructions at the same time as in the architectures classified as Vector-SIMD and SIMT. On the other hand, the TSIMT-lanes are also not as independent as the lanes in the VT architecture. The control logic in each lane is limited to a register storing a single instruction, control logic for the sequential register fetch and the density-time execution of the stored instruction, while in VT each lane is able to fetch its own instructions and can use a shared control processor.

3.8 SCALARIZATION

Lee et al. discussed Scalarization as well mentioned TSIMT [129]. They developed a Scalarizing compiler for SIMT architectures and evaluated architecture independent metrics such as the percentage of scalar instructions but did not evaluate performance. They also described potential GPU architectures exploiting scalarized code, such as SIMT datapaths with an additional scalar unit, SIMT datapaths with scalars in a single SIMD lane, and TSIMT datapaths. The authors recognized that Scalarization and TSIMT match each other well, but, as mentioned before, no actual implementation or evaluation is provided.

A similar analysis has been performed by Collange et al. [130]. Like the previous study, Collange implemented compiler support for Scalarization, but in a just-in-time form using GPUOcelot [131]. The author presented the Scalarization metrics of the resulting code, i.e. dynamic instruction counts of scalar and vector instructions, but no performance analysis is presented.

Coutinho et al. transform GPU kernels to an SSA-based intermediate representation called μ-SIMD, afterward they analyze the divergence of the program in its μ-SIMD representation [132]. They recognized that instructions could sometimes be scalarized, even during divergent control flow, if their output registers are not alive at the immediate post-dominator of the potentially divergent branch. However, they do not perform register allocation and thus only report how many of the registers in SSA form could be scalarized, but do not report how many registers of which type are actually needed after register allocation. The Scalarization algorithm presented in Chapter 11 directly works on the representation of GPU kernels used by NVIDIA and does not require a transformation to an SSA-based representation and back. In Chapter 11 register allocation is performed on the scalarized code and we discuss the modifications to register allocation that are required for scalarized code.

Xiang et al. studied the problem of Scalarization as well and also consider uniform values across warps [133]. They introduce a hardware Scalarization mechanism for intra-warp uniform instructions. This mechanism, however, does not allow a higher occupancy. A scalar register file based architecture is also presented, but does not enable higher occupancy but only reduces energy consumption. Scalars are processed using the same 8 element wide SIMD execution units as vector instructions, but scalar operations are finished in 1 cycle instead of 4. While this improves performance, it leaves 7 out of 8 ALUs unused during the execution of scalar instructions.

Finally, Kim et al. studied the relationship of the different values processed by a warp [134]. They named this value structure and identified several important classes such as uniform vector where all elements contain the same value and affine vectors where all elements share a simple affine relationship to *block-* and *thread-ids*. The authors focus on an architecture named fine-grained SIMT (FG-SIMT) that is closer to purely compute-focused SIMT accelerators than to GPUs. They propose microarchitectural mechanisms to exploit uniform and affine values, including an affine register file as well as dedicated affine execution units. The proposed mechanisms were evaluated on a conventional NVIDIA-like GPU architecture, but the authors state explicitly that the lack of public knowledge about GPGPU architectures prevents them from performing more than a preliminary design space exploration. Instead, they focused on an evaluation using a VLSI implementation of an FG-SIMT design.

After this review of related work, the next chapter presents the power measurement infrastructure used within this thesis.

4

GPU POWER MEASUREMENT

To improve the energy-efficiency of GPUs, we first need to comprehend the energy consumption of current GPUs. One of the first steps for gaining insight into the energy consumption of GPUs is measuring it. Together with microbenchmarks, measurements can be used to narrow down the energy consumption of specific parts of the GPUs or specific activities that the GPU performs. These results can then be a part of a power model that predicts the power consumption of the GPU based on activity factors estimated using an architectural simulator. The accuracy of the predictions of this simulator can be compared against energy measurements of the same benchmarks running on an actual GPU. In this chapter, the design of two different measurement testbeds is described. The first testbed was designed to enable energy measurements of discrete GPUs and the second testbed was designed to alleviate some limitations of the old testbed and also allow energy measurements of embedded SoC platforms. As these testbeds were designed for the LPGPU and follow up LPGPU2 projects, we refer to them as the LPGPU1 and LPGPU2 testbeds, respectively.

In the following sections, we first describe in detail how the analog power measurements are performed and then describe the software for the LPGPU1 testbed. Finally we describe the LPGPU2 testbed.

4.1 POWER MEASUREMENT CONCEPT

We measure power by measuring the voltage, as well as the current provided to the GPU or an SoC with integrated GPU. The measured voltages and currents are multiplied by the software to calculate the power supplied to the device.

The testbeds measure current indirectly by measuring the voltage drop caused by the current over a shunt small resistor. This is shown in Figure 4.1. Measuring the current using a shunt provides a wide analog measurement bandwidth. The wide bandwidth allows measuring direct current as well as quickly changing currents. Both slow and long events can be measured, such as the static power consumption as well as short kernels. The shunt is inserted

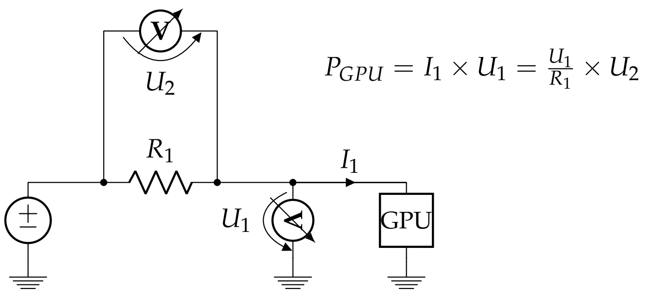


Figure 4.1: High Side Shunt GPU Power Measurement Scheme, Two Voltages U_1 and U_2 are measured to calculate the GPU power consumption P_{GPU}

into the positive voltage rail ("high-side measurement") to prevent distortions of the ground-level [135], [136]. In this measurement topology, a small voltage drop (e.g.: $100\,mV$) across the shunt needs to be measured while the much larger common mode voltage (e.g.: $12\,V$) needs to be rejected. Without excellent common mode rejection, small fluctuations in the supply voltage could cause significant errors in the current measurements. The required common mode rejection can be achieved with either matched high precision resistors or special instrumentation amplifiers or current sense amplifiers that contain internal highly matched resistors. For the LPGPU1 testbed, we designed a signal conditioning circuit based on the Analog Devices AD8210 current sense amplifier [137]. For the LPGPU2 testbed, a similar AD8218 amplifier was used [138].

4.2 LPGPU1 TESTBED

The LPGPU1 measurement testbed is used to measure the power consumption of GPUs and other PCIe cards. An overview of the LPGPU1 testbed is shown in Figure 4.2. The testbeds consist of four main hardware parts: An USB analog to digital converter (ADC), a riser card that inserts current measurement resistors between card and mainboard, PCIe power extension cables with added measurement resistors and signal conditioning boards. Power is measured by measuring voltages and currents of all power rails connected to card. Using $P = UI$ the consumed power can be calculated. An important part of the testbed not shown in the block diagram, is the software that supports the measurements. The software collects the raw data from the ADC and as well as from the GPU profiler and calculates how much energy is consumed in each kernel executed on the GPU.

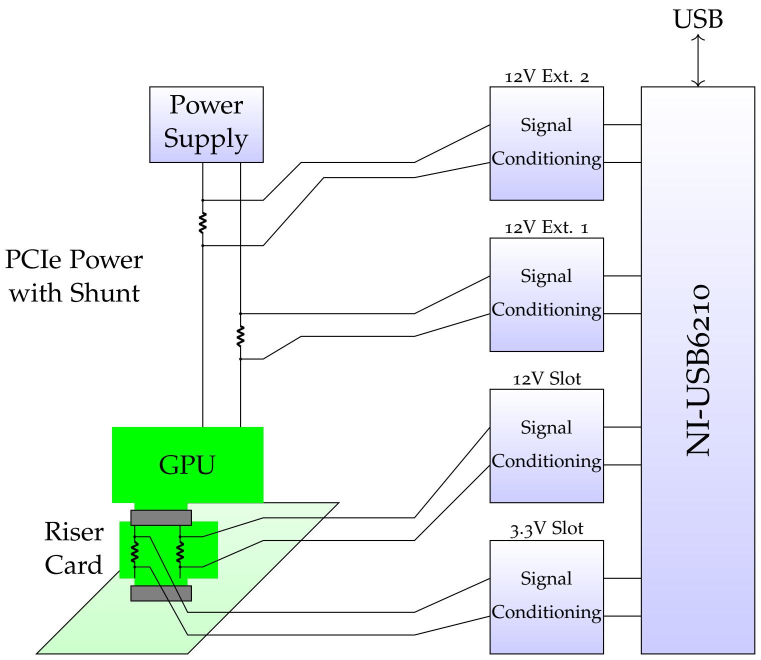


Figure 4.2: LPGPU1 GPU Power Measurement Testbed Block Diagram with 4 voltage rails being measured

4.2.1 *Analog to Digital Conversation*

To perform analog to digital conversion, we selected a "commercial off-the-shelf" (COTS) NI USB-6210 analog to digital converter [139]. This converter offers 8 analog differential inputs, 16-bits of resolution and a maximum sample rate of $250\,kHz$. One internal analog digital converter is shared between all input channels. Large discrete GPUs receive power from four different inputs: a $3.3\,V$ rail as well as a $12\,V$ from the PCIe slot, as well as two PCIe power connectors with $12\,V$. For all four voltage rails we need to measure the current and because the voltages are not exactly $3.3\,V$ and $12\,V$ but fluctuate slightly under load, also the voltages. This reduces our maximum sample rate to $250\,kHz/8 = 31.25\,kHz$.

National Instruments provides Linux drivers as well as an SDK for this ADC. However, these closed source drivers are only supporting older kernel versions [140]. Packages are only supplied for a few Linux distributions. The drivers are also not fully stable and fail to measure occasionally. A complete restart of the system was sometimes required to make new measurements possible. These issues were resolved in the LPGPU2 testbed by replacing the NI USB-6210 with our own complete measurement circuit, firmware and drivers.

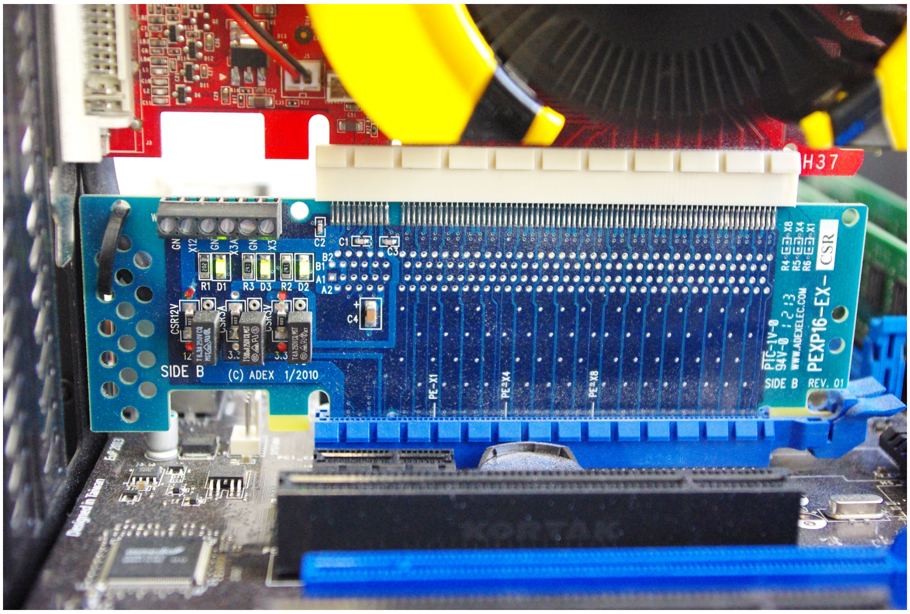


Figure 4.3: Adex PEXP16-EX-CSR riser card with GPU and current shunts

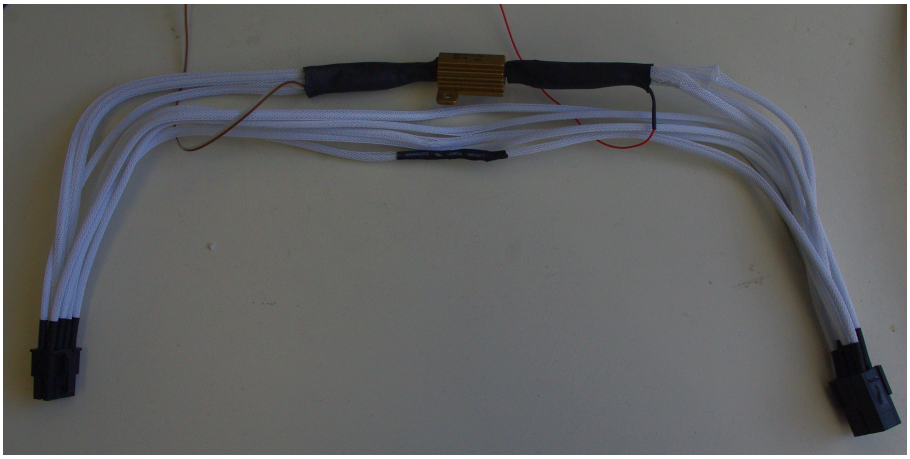


Figure 4.4: PCIe Power Cable with inserted shunt resistor and sensing wires

4.2.2 *Insertion of Current Shunts*

Discrete PCIe GPUs receive power from different power supplies. Up to 75 *W* can be supplied directly via the PCIe slot. This slot power is enough for small desktop GPUs. Larger GPUs require more power and have one to three power connectors to receive extra power via PCIe power cables. 6-pin power connectors can supply up to 75 *W* per cable, 8-pin connectors supply up to 150 *W* per cable. GPUs can use different power supplies simultaneously, e.g: A GPU that uses 200 Watt might draw 60 *W* from the PCIe slot and 140 *W* from an 8-pin PCIe power cable. As GPU vendors can implement different schemes to divide their power requirements between the different supplies, all supplies need to be measured in order to measure the total power consumption of the GPU. Therefore, a shunt resistor needs to be inserted into every voltage rail. As we did not want to modify the GPU or the motherboard we used a special riser card. This card is inserted between motherboard and GPU. We use an Adex PEXP16-EX-CSR riser card [141], as shown in Figure 4.3. It adds current sensing resistors into the power supply rails. We need to connect to both sides of the resistors to measure both the voltage drop over the resistor and the voltage to ground. The regular 3.3 *V* and the 12 *V* power rails need to be measured. The board provides big vias to the on-board shunt resistors (marked red in the picture). A connection to a cable can easily be soldered to these vias.

Modified PCIe power extension cables are used to measure the power going into the GPU via the external power connectors. The modified cables are shown in Figure 4.4. For measuring the current, a shunt resistor is inserted into a PCIe extension power cable. We used a 20 $m\Omega$ shunt resistor with

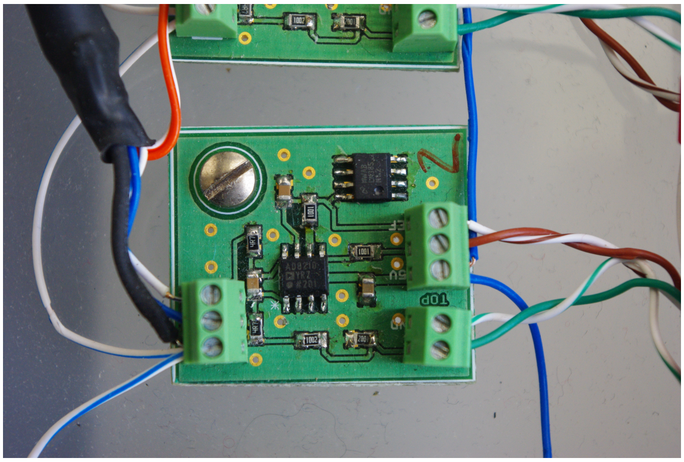


Figure 4.5: Analog signal condition board

$10\,W$ rating from Dale. The shunt resistors are inserted into the positive voltage lines, and on each side of the resistor a sensing line is added to the cable. These two sensing lines are then connected to the signal processing board.

4.2.3 *Signal conditioning*

Without additional circuits, the NI-USB6210 ADC is able to measure analog voltages between $-10\,V$ to $10\,V$ [139] . Our measurement setup needs to measure signals that can be outside of that range (e.g.: $12\,V$ DC). In addition and as already explained above, the voltage drop across the current measurement resistor is very small, e.g.: a current of $5\,A$ causes a voltage drop of just $50\,mV$ across a $10\,m\Omega$ shunt. The voltage drop needs to be shifted into the operating range of the ADC and requires analog amplification for accurate measurements. An AD8210 amplifier is used to perform both steps. It removes the common mode voltage and amplifies the voltage difference between its two input ports by $20\times$. It also enables the addition of small common voltage. This option is used to add a small voltage offset. This avoids issues with the output stage of the amplifier, as output voltages very close to $0\,V$ would be distorted due to the internal circuit design of the amplifier. The voltage offset is then later subtracted. A simple resistive voltage divider is used to scale the voltage into a range usable by the ADC. High precision resistors are used to avoid accuracy issues. The cables between the shunt resistors and the signal processing board were twisted. This twisted-pair cabling provided a significant reduction of electromagnetic noise. Differential

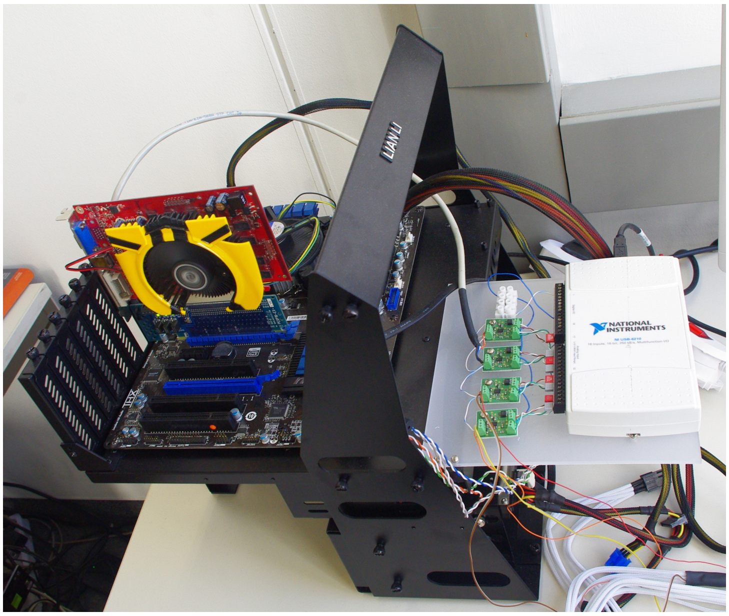


Figure 4.6: Completed Power Measurement System

connections between signal processing board and the ADC are employed to reduce the influence of noise. Our voltage divider was built from 1% resistors and has a gain accuracy of ±1.7% and no offset error. The AD8210 has a gain accuracy of ±0.5% and an offset error of ±1 mV at its output. At 12 V, the offset error translates to an error of up to 60mW in power measurements. The error range of signal conditioning and measurement is thus ±1.5% for currents and ±1.7% for voltages. In the relevant −5 to 5 V range, the DAQ has a specified gain accuracy of 0.0085% and an offset error of 0.1 mV. Not taking the small offset errors into account, overall, our system thus measures power within ±3.2%.

Each board contains the signal conditioning circuit for one voltage rail. GPUs using the only power from the PCIe slot have two voltage rails and thus require the use of two of these boards, GPUs using power from the slot and one external PCIe power connector need three of these boards, and GPUs using two external power connectors need four of these boards.

Figure 4.5 shows one of the analog signal condition boards. The terminals on the left of the board are connected to one of the shunt resistors. The terminals on the right provide the connection to the ADC and also 5 V power. Finally, Figure 4.6 shows the complete system including signal condition, riser card, PCIe power cables and ADC.

4.2.4 *Software*

An essential part of the measurement testbed is the software that drives the measurements. The ADC delivers raw samples of our voltages and currents. As its first step, the software converts these raw samples into proper units such as Volt and Ampere. It then calculates power and energy from these currents and voltages. For analyzing the power consumption of the GPU, while running a specific software, these almost raw measurements are not very useful. For such an analysis, we want to know how much energy was consumed by the execution of specific commands such as the execution of GPU kernel or memory transfers between GPU and host. To calculate how much energy each command used, we employ profiling functions offered by CUDA and OpenCL to receive a list of relevant commands and their timings. CUDA provides a command line profiler that can be activated by setting specific environmental variables before executing an application. After the application was executed, the profiler creates a small text file that contains a list of executed kernels and data transfers as well as the start and end time of these actions. To gain this type of information from unmodified OpenCL applications more work was required. While applications can use OpenCL APIs to record the start and end times of these actions, this typically requires modifying applications to request the information from the API and record the returned information into a file. Modifying applications requires access to the source code of the applications and would have required a significant amount of work for every single application to be measured. We, therefore, developed a different solution: An interposer library is inserted between the real OpenCL API and the application and automatically adds requests for profiling information to the intercepted API calls and writes the received data to a file similar to the CUDA command line profiler. For OpenCL profiling the testbed currently uses an improved interceptor library developed by Guilherme Calandrini based on a prototype interceptor by the author of this thesis.

The CUDA command line profiler or the OpenCL interposer provide us with start and end times of the relevant commands on the GPU and the ADC provides power samples, but to complete a meaningful power profile from the information we need to go one step further. The profiling information uses one clock source while the ADC uses a different clock source. We need to perform a very exact synchronization of these two clock sources. To allow synchronization, we first launch a special application before our main application under measurement. This special application performs multiple kernel launches with high GPU activity and pauses with low GPU activity. This causes a similar pattern in the GPU power consumption as shown in the first graph in Figure 4.7. The profiler provides us with the exact start and end times of our kernels. We then use this information to build a "power

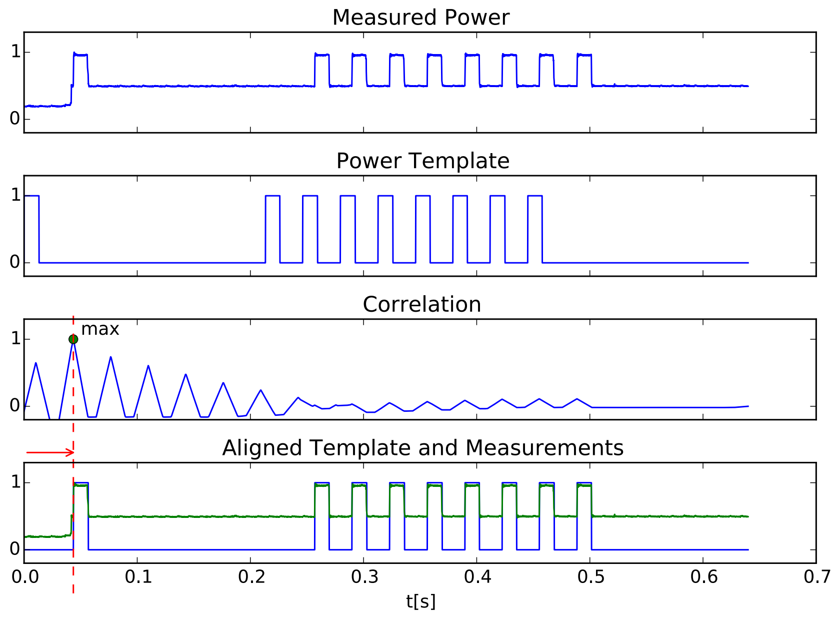
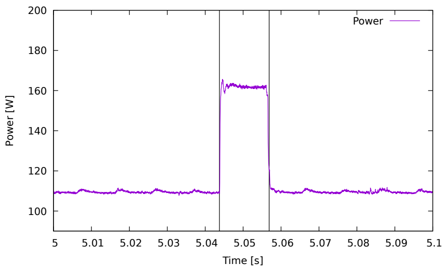


Figure 4.7: Alignment of Profiler and Sample Clock using Correlation

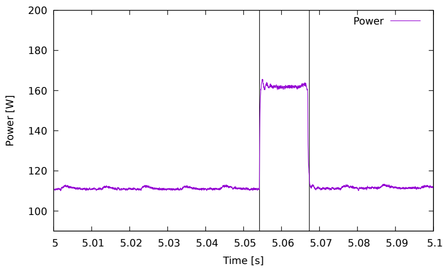
template" that shows the expected power consumption pattern as shown in the second graph of the Figure. We then calculate the correlation between our measurements and the power template with various different potential alignments. The alignment with the highest correlation between measured power and power template provides us with the most likely offset between the profiler clock and the sample clock. This offset can then be used to convert profiling information to the sampling clock and vice versa.

We can now calculate the energy and power consumption of specific kernel launches or data transfers: We convert the start and end time of the action to sample clocks and then sum up the energy consumption during this time interval. To calculate the average power consumption, we divide the energy consumption by the duration.

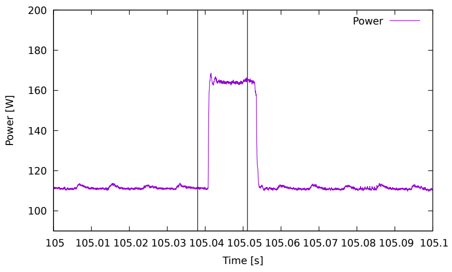
This basic measurement setup worked well for actions longer than a few ms together with total measurement duration of up to two minutes. Several improvements were required for longer measurement durations and shorter kernels. When measuring for longer durations, it becomes noticeable that the profiler clock and our sample clock are running at slightly different rates, due to oscillator tolerances. We found that there is a difference in the rates of a few parts per million between sample clock and profiler clock. This difference in the clock rates matches well with the typical accuracy of regular crystal



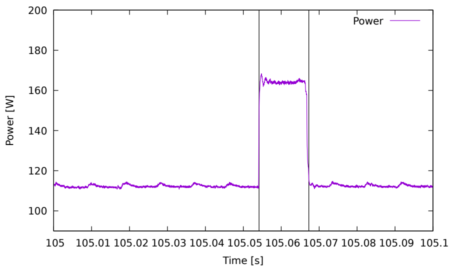
(a) Start without clock drift compensation



(b) Start with clock drift compensation



(c) End without clock drift compensation



(d) End with clock drift compensation

Figure 4.8: Measurements with and without clock drift compensation

oscillator as they are used on the GPU and in the ADC. Because the initial offset between sample and profiler clock drifts away with a speed of around 1 sample per second of measurement. When performing longer measurements our conversion between sample and profiler clock is no longer accurate, and we would measure the power consumption slightly too early or too late.

For this reason, we implemented a correction algorithm in our tool. When this algorithm is activated, the software tries to estimate rate difference between the two clocks and corrects it. At the start of each kernel execution, there is usually a strong increase in power consumption, this feature is used to estimate the rate difference. The algorithm scans a window around the expected start of each kernel for an increase in power consumption. If the strongest increase is found earlier or later than expected the estimated rate is adjusted accordingly. A low pass filter on the estimated rate is used to prevent rate adjustments purely based on measurement noise. We found that correction algorithm works well: With the correction algorithm, the power data from 15 minutes and longer measurements agree with reference data from short measurement durations and the estimated start and end times of the kernel align well with power consumption patterns, while the drifting is easily noticeable without this algorithm. Figure 4.8 shows the improvement provided by our clock drift compensation algorithm. Figures 4.8a and 4.8b

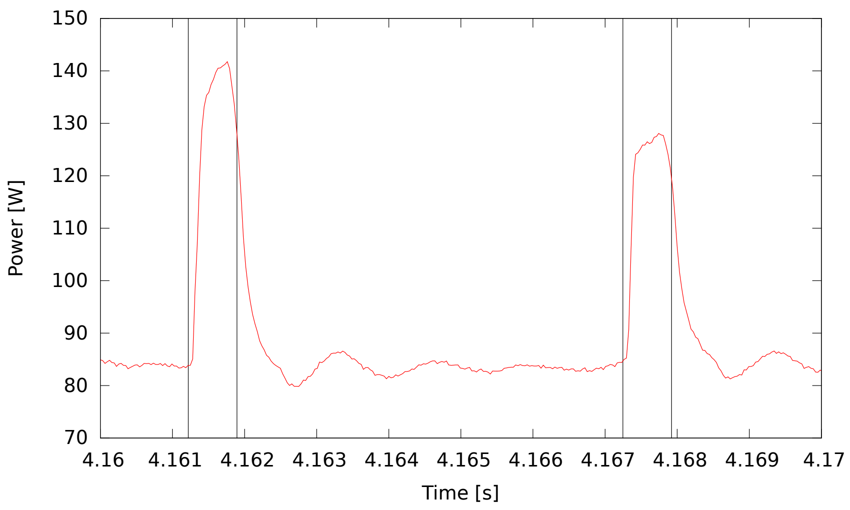


Figure 4.9: Power measurement of very short kernel

show a kernel launch at the beginning of our measurement. The black bars mark the kernel end and start times according to the profiler. In both cases, the conversion from profiler clock to sample clock was successful and the start and end marks are well aligned with the higher power consumption of the GPU caused by the kernel. This picture changes in Figure 4.8c, here we show a launch of the same kernel, but almost 2 minutes later and without drift compensation. The small differences in the clock rate of profiler and sample clock have caused our profiler marks to be placed too soon. Our testbed would then underestimate the energy consumed by this kernel as part of the energy of kernel is consumed past the end mark and the start mark is placed too early, causing the tool to record idle power instead of kernel power. Activating the clock drift compensation solves this issues. Figure 4.8d shows a measurement similar to the measurement in Figure 4.8c but with drift compensation enabled. Now start and end marks are perfectly aligned with the power consumption.

Measurements of very short kernels remains challenging. Figure 4.9 shows a measurement of a kernel with less than 1 ms runtime. Instead of the nearly rectangular shape of the power measures as previously seen in Figure 4.8, on this scale the power supply circuits smooth the power consumption over a longer period of time and we can see a "ringing" effect in response to the load change after the kernel ends. A significant part of the power is consumed prior to our measurement window and even tiny offsets in the kernel start and end times can change the measurement results significantly.

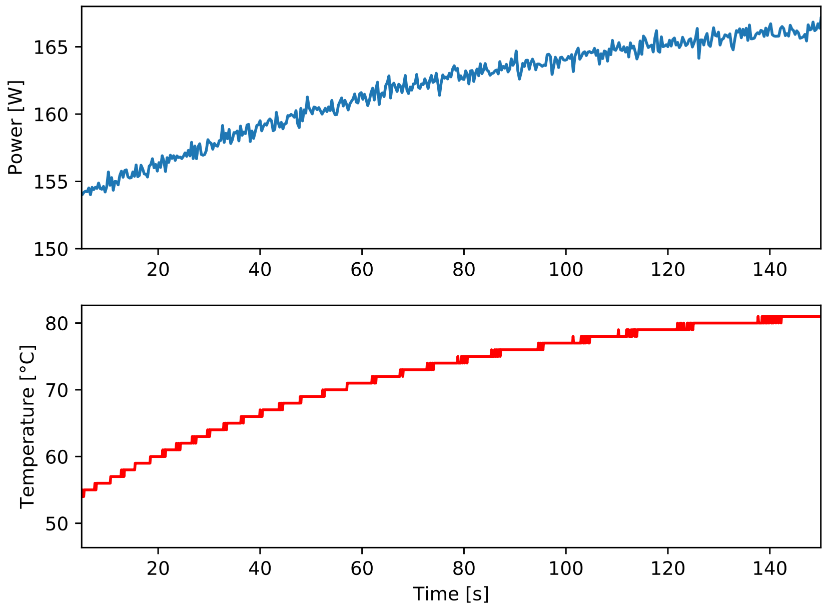


Figure 4.10: Microbenchmark shows increasing leakage with increasing temperature

Another encountered measurement issue is the leakage power consumption. CMOS leakage power depends on temperature [142]. Our measurement results in Figure 4.10 show this effect. For this figure we executed the same kernel with same parameters and data many times. The dynamic power consumption of this kernel should thus be constant, however, the total power consumption of the GPU changes from less than 155 W to more than 165 W, as the temperature of GPU increases from 55 °C to more than 80 °C. This is not a measurement error per se, however, it can lead to measurement errors when we want to compare the power consumption of two kernels, but the kernels are executed at different GPU temperatures. Fortunately, the GPU temperature changes slowly over time. We can compare the power consumption of two different kernel by executing them at nearly the same time, as this also ensures that the temperature (and thus also the leakage power) of the GPU will be very similar. When we want to compare the dynamic power consumption of a large set of different kernels, we can interleave our measurements with measurements of a common "baseline" kernel and subtracting the power of the previous baseline kernel run from each of the real measurements. This way we can compare measurement results generated at different GPU temperatures. An alternative would be to measure the temperature and subtract the estimated leakage.

4.3 LPGPU2 TESTBED

In this section, the LPGPU2 testbed and its improvements and other changes are described. We first describe the new hardware of the LPGPU2 testbed and then describe the design of the employed firmware and the new software that allows measurements on embedded Android platforms.

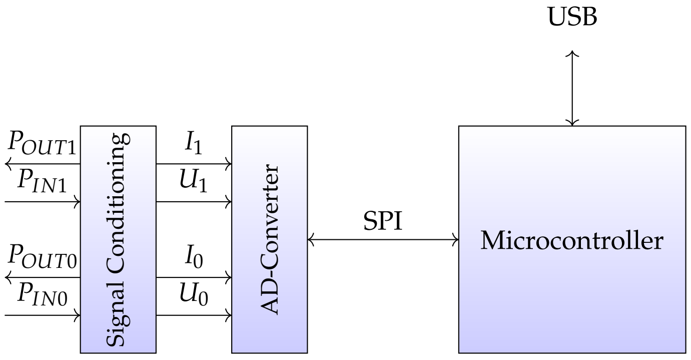


Figure 4.11: LPGPU2 Testbed Block Diagram

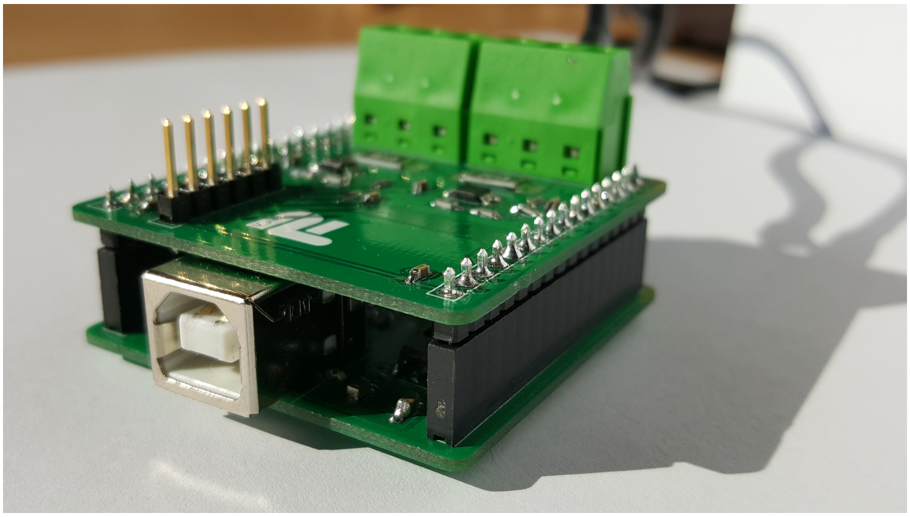


Figure 4.12: LPGPU2 Measurement Testbed Prototype, Analog board at the top, Digital board below

4.3.1 *LPGPU2 Testbed Hardware*

This section describes the hardware of the testbed as well as the microcontroller firmware running on the hardware. The testbed hardware is shown in Figure 4.12. Figure 4.11 shows a block diagram of the main hardware of the LPGPU2 testbed. First, the analog signal is prepared for analog to digital conversion with a signal conditioning stage, then converted to digital signals by an analog-digital converter. A microcontroller receives the signals using an SPI bus and transmits them to the host PC via USB. The testbed consists of two custom designed PCBs, the top PCB contains the analog signal processing circuits as well as the shunt resistors while the lower PCB contains the digital part of the measurement hardware. This modular concept allows a replacement or redesign of either part of the system without requiring changes to the other part. An annotated photograph of the analog measurement board is shown in Figure 4.13. The analog PCB contains screw mounts for the power supply cables, two shunt resistors, two sense amplifiers for power measurement with the required large common mode range and passive components for filtering and scaling the signal. After this signal conditioning, the analog signals are transmitted to the lower PCB using the connector in the left. The digital measurement board, shown in Figure 4.14, converts the analog signals to digital signals and continuously transmits them to a microcontroller using an SPI bus. The microcontroller buffers the data and transmits the data to the host PC via a USB port. As a microcontroller, we selected an STM32F103CB

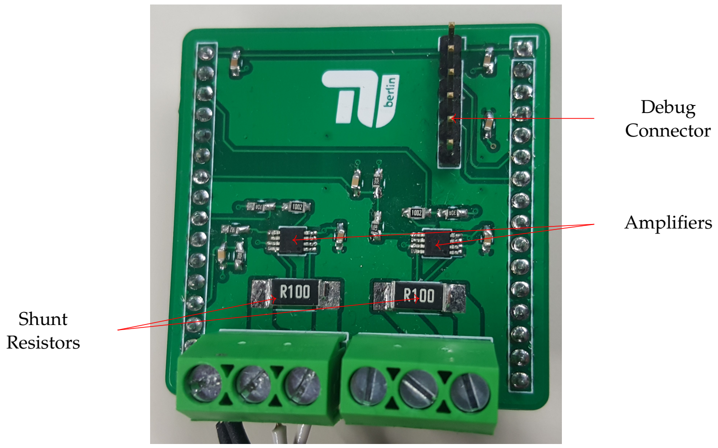


Figure 4.13: LPGPU2 Analog Measurement Board

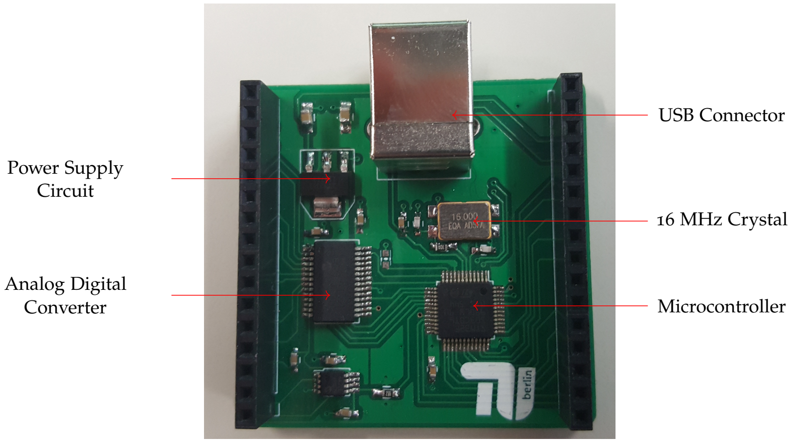


Figure 4.14: LPGPU2 Digital Measurement Board

microcontroller [143]. It uses a 32-bit ARM Cortex-M3 architecture, can run at up to 72 *Mhz* and contains the required SPI and USB interfaces as well as a powerful DMA controller that allows us to offload large parts of the data transfer to the DMA controller and ensure that all samples are recorded. Microchip's MCP3912 was selected as an analog-digital converter (ADC) [144]. It offers high resolution measurements on 4 input channels. This allows us to measure two voltages and two currents at exactly the same time and also measure power consumption on platforms with multiple power sources such as discrete PC GPUs. If more than two channels are required, multiple measurement adapters can be used. The MCP3912 also allows high sampling rates at up to 125 *kHz*.

4.3.2 *LPGPU2 Testbed Firmware Programming and Calibration*

When its firmware is initially programmed into the LPGPU2 testbed, a calibration procedure is performed on every LPGPU2 testbed. This reduces the systematic error caused by variations of the analog components. The hardware used for the programming and calibration procedure is shown in Figure 4.15. Spring-loaded contacts provide a connection to the power measurement terminals and the pins required for flash programming. Using a USB relay board a number of known 0.1% resistors is connected to the board to provide known currents. At the same time the voltage is measured using a highly accurate Voltcraft VT650 benchtop multimeter[1]. The ground truth measurements are used to calculate calibration factors for each device. The calibration factors are stored in the firmware of each device and can be retrieved via USB and used by the measurement software to convert the raw measurements of the ADC into currents and voltages while taking the effect of analog component variations into account.

4.3.3 *Microcontroller Firmware*

The microcontroller firmware is responsible for communicating with the host PC via USB and streaming the data from the analog-digital converter to the PC. Before the actual power measurement starts the firmware needs to initialize the internal registers of the ADC. Then, the DMA controller of the microcontroller is programmed to continuously read the data from the ADC at a fixed rate into a circular buffer. The microcontroller provides the clock to the ADC and adjusts the SPI transfer rate according to the sample rate to receive the samples exactly at the sample rate. Because of this exact match between sample rate and data transfer rate, the software does not require checking, if new samples are available, but will automatically receive every

[1] $\pm 12.5\,mV$ in the used measurement range

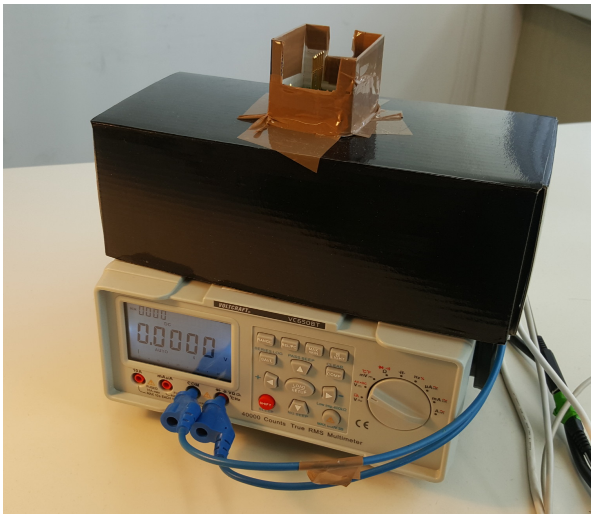


Figure 4.15: LPGPU2 calibration setup

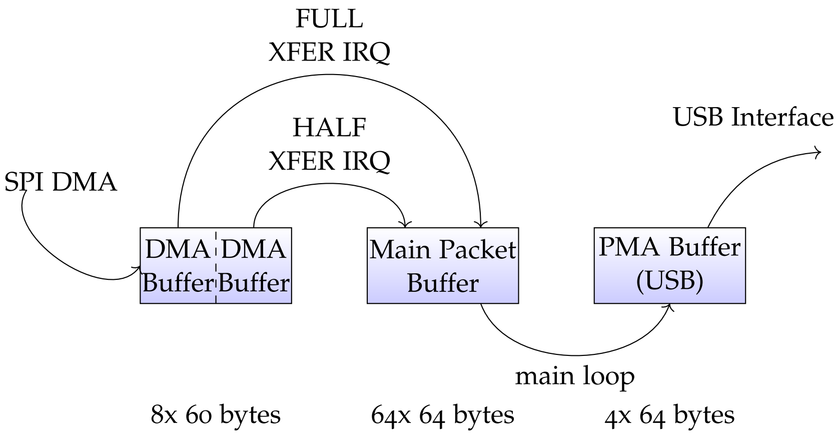


Figure 4.16: USB Buffer Management

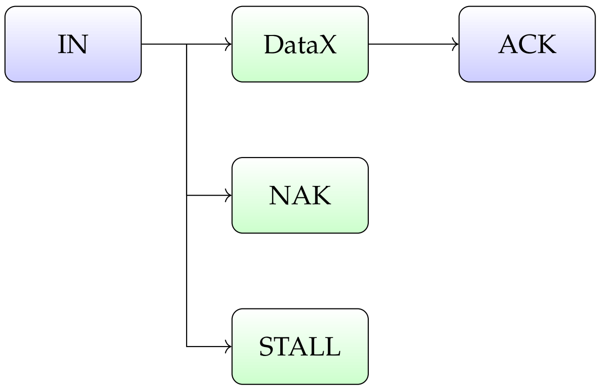


Figure 4.17: USB Data Transfer

new sample exactly once. The DMA controller triggers interrupts, when the current write pointer reaches the midway or end point of the circular buffer. The firmware must then quickly copy the data to a different location before it will be overwritten. At the same time, it formats the data into small 64 byte packets and adds a frame counter to the raw data. This enables the detection of buffer overrun events.

While the data transfer between ADC and microcontroller runs continuously, the data transfer on the USB interface happens in bursts. This requires a complex buffer management, as shown in Figure 4.16 and explained later in this paragraph. The data transmission via USB is managed by the USB controller in the host PC. The USB controller requests new data and the microcontroller needs to react to these requests. When the PC wants a new data packet, the microcontroller must respond either with a new packet or with a special "NAK" message, if no full data packet is available yet, or a special stall packet, that signals an error. This concept is shown in Figure 4.17. This process is performed partly in hardware, as the USB device needs to react immediately to the data request received from the PC. Packets that should be sent to the PC via USB need to be stored within a tiny 512 byte buffer. As this buffer is not large enough to store data packets while the PC is not requesting data, data cannot be copied directly from the circular DMA buffer to the USB buffer. A larger main packet buffer is used inside the microcontroller SRAM to store packets from the circular DMA while they are waiting for USB transmission. Slots for four 64 byte packets are kept within the USB buffer. This way the firmware can transfer data to one packet slot to the USB buffer while the hardware transmits a data packet from a different slot. Once the transfer is finished the firmware only needs to change the address of the next packet to the next slot that contains a valid, not yet transferred packet.

4.3.4 *Host Software*

The measurement hardware is connected to a host computer via USB. A C++ program runs on the host computer which uses libusb to capture the continuous stream of measurements [145]. It then converts the measurements into floating point values and calculates power from voltages and currents. To synchronize the recorded performance counter data and other events with the power measurements, the same technique is used that was already employed in the LPGPU1 testbed: At the start of the measurement, a special program is executed on the device that generates a series of power consumption spikes. The start and end times of these power consumption spikes are recorded and turned into a power template. Then the alignment process uses the correlation between measured power and power template to find the offset between the sampling clock and the device clock.

4.3.5 *Android Software*

The LPGPU1 testbed was measuring the power consumption of discrete GPUs in high performance PCs. In this setting, we were able to use the same computer to record and process our samples that was also running the applications. We were only interested in GPU power consumption so a slightly higher CPU power consumption due to the measurement did not matter. In the LPGPU2 testbed, however, we are measuring the power consumption of embedded SoCs which include both the CPU and the GPU. Extra CPU power consumption would show up in our measurements as well, so we need to minimize it. The amount of memory is very limited, so we cannot reserve large memory buffers to store our samples during the measurement. For this reason, our LPGPU2 testbed works differently: An extra host PC is used to collect and process the measurement data. Only a lightweight process runs on the embedded SoCs and collects performance counter data. The host PC and the embedded system are connected via USB and the "Android Debug Bridge" is used to forward data from the performance counter collector to the host PC and for the host PC to automatically launch applications on the embedded system.

4.3.6 *Summary*

This chapter described our LPGPU1 and LPGPU2 measurement testbeds. It explained how power is measured by measuring currents and voltages and highlighted the analog signal condition challenges solved by the testbed circuits. It also explained how our measurement software is able to combine the analog measurements with the event traces to generate a power profile. This power profile provides the user with precise measurements of the energy consumption of individual kernels. The chapter explained the synchronization and clock drift compensation algorithms developed to enable the power profiling. The chapter also explained the LPGPU2 testbed, the firmware developed for the testbed and its USB connection. We also illustrated how performance counter data from embedded devices is forwarded to the measurement host and how this data is combined with the power measurements. In the next chapter, we use our power measurement testbed to build and validate a GPU power simulator called GPUSimPow.

5 ARCHITECTURAL GPU POWER MODELING

The work presented in this chapter was previously published: J. Lucas, S. Lal, M. Andersch, *et al.*, "How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator," in *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, ©2013 IEEE, 2013.

With processor designs becoming ever more complex and chip manufacturing processes becoming harder to control, the inability of computer architects to produce working prototypes of their designs for testing is a more pressing problem than ever before. With chips rapidly approaching (and, nowadays, touching) the power wall, the conventional design space of processor architecture has been extended by another dimension: Energy consumption.

Over the past years, it has become apparent that the chips consuming the most energy by far are modern Graphics Processing Units. With GPUs turning into major devices for general purpose computations, so-called general-purpose computation on GPUs (GPGPU), this trend has only accelerated as more and more parties are striving to drive GPU performance up. The inability to manufacture chips to evaluate architectural design choices, however, remains, as does the looming power wall.

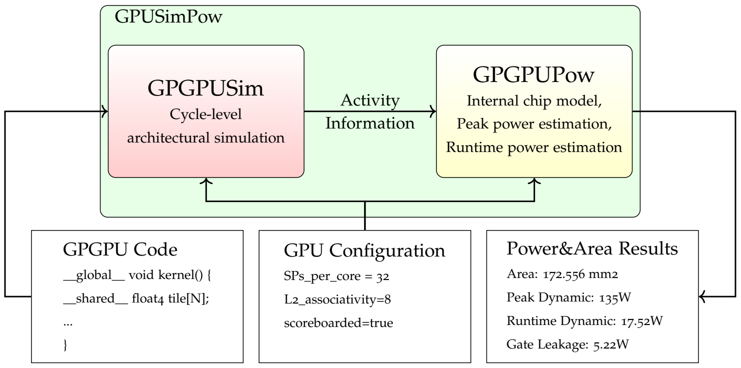


Figure 5.1: GPUSimPow Block Diagram

So how do the design of new GPU architectures, the inability to manufacture chips just for testing, and the requirement to not only estimate a chip's performance, but also its power *during development* come together? On the one hand, if we disregard power and only consider performance, this question has been answered by several researchers over the past years: GPU architects now must rely on building cycle-accurate architectural simulators in high-level languages and evaluate novel designs using these simulators. On the other hand, there are several accepted tools and frameworks to model and estimate power consumption for CPUs such as Wattch [98]. To the best of our knowledge, however, no one ever combined architectural GPU simulators with power models to create *GPU power simulators* before.

We seek to mitigate this issue by introducing *GPUSimPow*, a power simulation framework for contemporary GPGPU architectures. GPUSimPow is able to estimate multiple characteristics of a hypothetical GPU architecture such as chip area, gate leakage, and peak dynamic power, as well as precisely simulate the power consumed during execution of GPGPU workloads. With this power simulator, computer architects can evaluate design choices early from a power perspective, and GPGPU programmers gain an effective way to investigate their GPGPU codes, so-called *kernels*, to optimize power consumption from a software perspective. To make GPUSimPow flexible enough that architectural design choices can easily be carried out while still maintaining reasonably high accuracy of the power predictions compared to real hardware, we model the components of the GPU architecture in two ways: Regular components such as memories are modeled *analytically* using the well-known McPAT [87] tool (which, by itself, integrates CACTI6.5 [146]). Irregular components such as address generation units (AGUs) or special-function units (SFUs), on the other hand, are modeled *empirically* by acquiring measurement data on real hardware. To obtain accurate measurements, we propose a specialized *measurement testbed*.

In summary, we make the following contributions:

- We develop the GPUSimPow power simulator that is able to generate area, power and runtime power estimations for contemporary GPGPU micro-architectures and GPGPU kernels.
- We propose a novel measurement methodology to be able to accurately measure GPU power consumption on real hardware down to the individual kernel.

GPUSimPow was developed jointly by the author of this thesis, Sohan Lal and Michael Andersch. Text and figures by Sohan Lal and Michael Andersch were included in this thesis with their permission to provide a full overview and evaluation of the GPUSimPow simulator. Sohan Lal developed most of the activity factor extraction from the gpgpu-sim performance simulator. In addition, he was responsible for the GDDR5 power model, the shared memory power model as well as the adaptation of various caches models from McPAT to GPUSimPow. Michael Andersch developed the power models for the Load-Store Unit including the address generation unit as well as models for the texture cache and warp control unit including the warp scheduler. The author of this thesis developed the power models for the register files, the empirical performance models for the execution units, core and cluster power, adapted the McPAT NoC power model as well as the measurement infrastructure used in this chapter for the calibration of the empirical models and the accuracy evaluation of the power simulator.

This chapter is organized as follows: In Section 5.1, we introduce our power simulator, describe its structure and the simulated architecture. Then, Section 5.2 presents the measurement setup we used to estimate component power and validate the simulator results, as well as the rest of our experimental setup for simulator runs. Section 5.3 presents the simulator results and compares them with power measurements on real hardware.

Work related to this chapter was already presented in Chapter 3, Section 3.3. A short summary is presented in Section 5.4. Conclusions related to this chapter are drawn, at the end of thesis, in Chapter 12.

5.1 THE GPUSIMPOW TOOL

In this section, we present our power simulation framework called *GPUSimPow*. An overview of GPUSimPow's structure is given in Section 5.1.1. In Section 5.1.2 we are providing information about the power model and its link to the performance simulator. The baseline architecture details are explained in Section 5.1.3. Finally, Section 5.1.4 gives details on how the empirical parts of the model have been derived.

5.1.1 *Overview*

GPUSimPow is a power simulator for GPGPU workloads, i.e. given a configuration of a particular GPU architecture and a GPGPU kernel written in CUDA [147] or OpenCL [148], GPUSimPow is capable of producing both architectural information such as static power, peak dynamic power, and area, as well as runtime dynamic power for the kernel at hand. The simulator is designed to be flexible regarding the architecture that is simulated to allow architects to utilize the simulator as a high-level tool to explore the GPU architecture design space. Therefore, the key parameters of the simulated architecture are supplied using a simple XML-based interface. For example, GPUSimPow is able to coherently simulate an architecture with a varied number of cores.

5.1.2 *Power Model*

In general, power for switching circuits is described by the well-known Eq. 5.1 [87].

$$P_{\text{total}} = \alpha C V_{dd} \Delta V f_{\text{clk}} + V_{dd} I_{\text{Short-circuit}} + V_{dd} I_{\text{leakage}} \tag{5.1}$$

The first term is the dynamic power that is spent charging and discharging capacitive loads when the circuit switches state. An important factor for this chapter is the *activity factor* α that describes the percentage of the circuit's capacitance being charged during switching. The second term of Eq. 5.1 is the short-circuit power being consumed when *both* pull-up and pull-down networks in a CMOS circuit are on for a short amount of time. Thus, the total power consumed by a circuit during switching is the sum of the dynamic and short-circuit power terms. Finally, the third term of Eq. 5.1 is the *static* power consumed due to leakage current through the transistors, where leakage consists of two distinct types: Subthreshold leakage, where a transistor that is switched off leaks current between its source and drain, and gate leakage, where current leaks through the transistor's gate terminal.

Structurally, GPUSimPow consists of two main parts, visualized in Figure 5.1: First, a cycle-accurate GPGPU simulator that simulates the given kernel and thereby generates utilization information and activity factors α for all components of the GPU architecture, and second, a chip representation with a power model for each component that uses the activity information from the simulator to produce power numbers for a particular kernel. From the chip representation, statistics about area and peak, leakage, and short-circuit power are inferred as well.

For the cycle-accurate GPGPU simulator, we employ a modified version of the most recent GPGPU-Sim [149] that has been altered to produce access counts and other activity information for all parts of the simulated architecture. GPGPU-Sim has been developed for an architecture that is not equal but comparable to many current off-the-shelve GPUs such as NVIDIA's Fermi [76] or AMD's GCN [78]. Further details about the architecture are given in the next section.

The chip representation and power model are provided by a heavily modified variant of McPAT [87] we name *GPGPU-Pow*. McPAT integrates three different modeling tiers *hierarchically* to provide a flexible and highly accurate power model for *CPUs*: The architectural tier, where a processor is broken down into major components such as cores, caches, and memory controllers, the circuit tier, where the architectural components are mapped to basic circuit structures such as arrays or clocking networks, and the technology tier, which provides the physical parameters, such as current densities and capacitances, of the circuits. Besides this hierarchy, a unique advantage of McPAT is its *combination* of analytical and empirical models for the individual components. We embrace both the hierarchical as well as the combined nature of McPAT and develop a McPAT-based model for GPUs. On the one hand, this requires many modifications to McPAT, as multiple components that are present in the CPU architecture model such as register alias tables cannot be reused for GPUs, and various core components of GPU architectures, such as stacks to handle thread divergence, are not present in CPUs. On the other hand, using McPAT enables us to utilize all the integrated low-level technological information, e.g. to scale the GPU power model for a specific manufacturing process node, we can use the ITRS roadmap scaling techniques within McPAT.

5.1.3 *Modeled Architecture*

The GPU micro-architecture we designed our power model for is comparable to the one given in GPGPU-Sim to ensure a good "fit" between GPGPU simulator and power model. Generally, it is a single-instruction multiple-thread (SIMT) architecture that uses a stack-based divergence handling mechanism, well representative of many current GPUs.

On the highest level, a GPU chip in our model consists of a memory controller (MC), a Network-on-Chip (NoC), a PCIe controller (PCIeC), and a collection of cores. Besides the actual chip, we model the GDDR5 graphics DRAM as well. For NoC, MC, and PCIeC, we re-used the highly configurable models already present in McPAT and adjusted their parameters to fit the different requirements of a GPU. The internal structure of a core consists of a *Warp Control Unit* (WCU), a highly banked register file, a set of SIMD execution units (INT, FP, SFU), and a load/store unit (LDSTU). In the following, each of these components, as well as our GDDR model, are briefly introduced.

Warp Control Unit

The WCU represents the front-end of a single core. As such, the WCU is responsible for keeping the execution back-end, i.e. the functional units and the load/store hardware, supplied with instructions at all times. Thus, the WCU handles thread management (i.e. formation of *warps* from threads and the relation of per-thread control flow under warp constraints), warp scheduling, warp instruction fetching, decoding, dependency resolution, and renaming. An overview of our WCU model is depicted in Figure 5.2.

The information needed for each warp to fetch instructions and manage the warp threads is contained in a single multi-ported RAM table, the Warp Status Table (WST). The WST contains one entry for each in-flight warp the core can handle. To select a warp to fetch an instruction for, a rotating-priority (round-robin) warp scheduler is modeled. Such schedulers consist of a set of inverters, a wide priority encoder, and a phase counter. These components

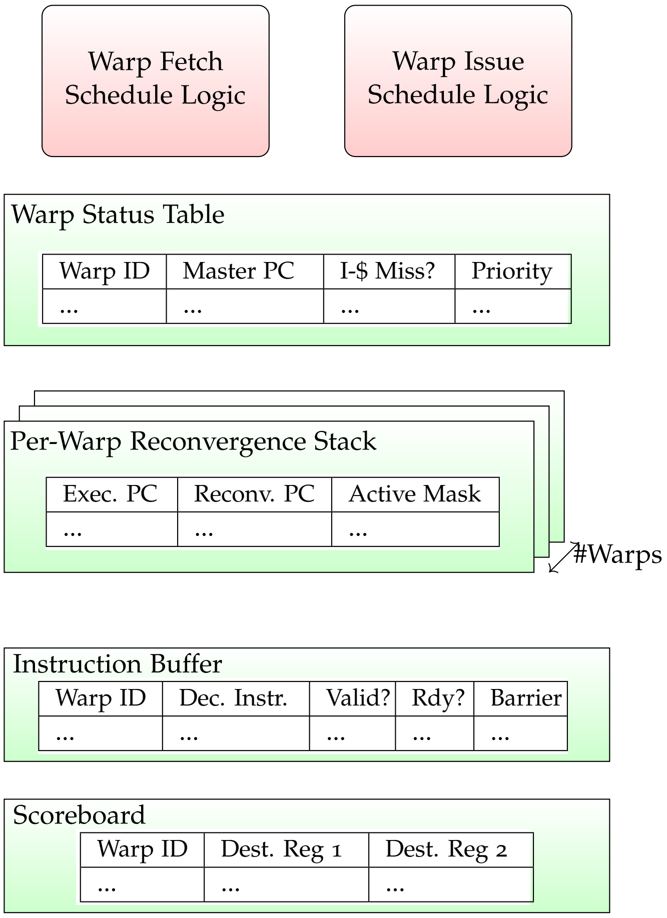


Figure 5.2: High-level Overview of Warp Control Unit. (Figure reproduced from Chapter 2 for the reader's convenience.)

have been modeled from appropriate circuit plans [150] using McPAT's circuit and technology layers. After instructions have been fetched from the I-Cache, they are decoded. For this, we re-use the instruction decoder hardware models already present in McPAT.

As is common for most GPGPU applications on modern GPU architectures, the individual in-flight threads often execute different dynamic instruction paths. The grouping of threads into SIMD bundles (warps) implicitly forces the thread PCs to have the same value at all times, however. If this is not the case due to the threads branching into different dynamic execution paths, the execution of threads in a single warp but with different PCs is serialized. To achieve this serialization and keep track of the thread IDs that have to execute certain branch outcomes, the hardware uses a stack memory called the reconvergence stack [151]. For each individual in-flight warp, the hardware maintains a separate stack. In our model, a stack consists of tokens, each of which contains an execution PC, a reconvergence PC, and an active mask for that warp and code block.

Once an instruction has been decoded, the WCU places the instruction into an instruction buffer (IB) slot. The instruction resides in its buffer slot until it is ready to execute, that is, if its register dependencies have been resolved (in scoreboarded architectures) or the previous instruction from the same warp has been committed (in blocking barrel-processing architectures). The instruction buffer is a cache-like structure that is tagged by the warp ID and has an associativity greater than one, i.e. each instruction can be buffered in one of several slots tagged by its parent warp ID.

For resolving register dependencies, GPUs (e.g. NVIDIA Fermi) use simple approaches based on scoreboarding [152]. In our models, a scoreboard is a cache-like table tagged by the warp ID.

Register File

The GPU register file model is based on an NVIDIA patent [153] and built from multiple single ported RAM banks. Operands are collected over multiple cycles to simulate a multi-ported register file. Different threads will have their registers stored in different banks. This scheme increases the area density of the register file. A crossbar is used to connect the different register banks to a set of operand collector units which are two-ported four-entry register files.

Execution Units

The basic unit of execution flow in the SIMT core is the warp. The GPU has a set of SIMD execution units which execute the warp threads in lock step. For example, the SIMT core in the NVIDIA GT240 has eight fully pipelined floating point units (FPUs), eight pipelined integer units (IUs) and

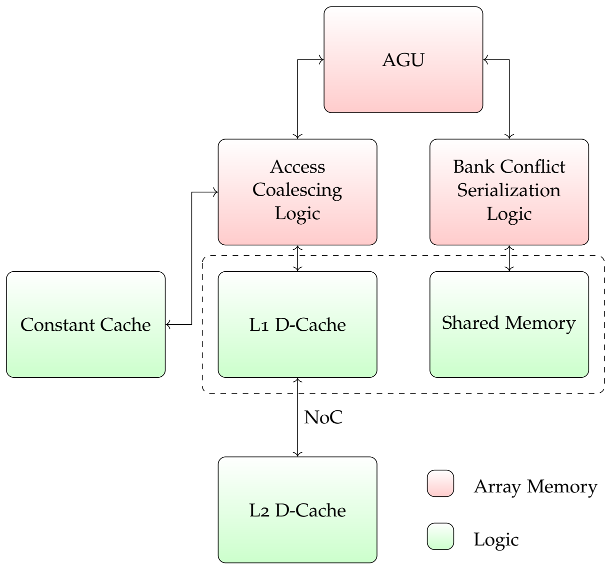


Figure 5.3: High-level Overview of Load/Store Unit (Figure reproduced from Chapter 2 for the reader's convenience.)

two special function units (SFUs) to execute transcendental instructions such as sine, cosine, reciprocal, and square root. In our power model, we used the area numbers published by Sameh et al. [154] for FPUs, the results of Caro et al. [155] for power and area of the SFUs with scaling for the desired process technology, and our own measurements for power of integer units and floating point units (see Section 5.1.4).

Load/Store Unit

The load-store unit (LDSTU) is functionally responsible for handling instructions that read or write any kind of memory. In the power model, the LDSTU encapsulates the top-tier memory structures of the core, i.e. L1/SMEM, the constant caches and the L2 caches. In a future variant of the model, the LDSTU will contain the texture caching subsystem, i.e. texture caches and texture mapping units, as well. An overall overview of the LDSTU is depicted in Figure 5.3.

As the figure shows, a memory access instruction for an entire warp is first passed to the address generators. Given base addresses as well as strides and offsets, the address generation unit (AGU) generates one memory address per thread in the warp. Given reasonable warp sizes of 32/64 threads in modern architectures, this requires very high bandwidth address generation units that supply the later stages of the memory subsystem with 32/64 memory addresses each cycle. We model the complete AGU as an array of parallel high-bandwidth sub-AGUs (SAGU), each of which is able to generate 8 memory addresses per cycle [156]. Given the memory address bundle for all threads in the warp, the address bundle is further analyzed depending on the type of memory the instruction accesses.

If the instruction accesses constant memory, the addresses are checked for equality. The number of generated constant cache / constant memory accesses is equal to the number of different addresses in the address bundle, e.g. if all addresses are equal, the memory access can be serviced with a single constant memory request, allowing for high-bandwidth operation. The constant memory segment is cached in an entire hierarchy of constant caches [157].

If the instruction accesses global memory, it is first coalesced before being passed to the L1/L2 caches and/or DRAM. The coalescing system is modeled after a corresponding NVIDIA patent [158] and consists of an input queue, output queue, pending request table, and a finite state machine. The goal of coalescing is to service the addresses requested by the memory access in as few memory requests as possible. As our research revealed, CACTI cannot be used to model buffers with few but very large entries such as the pending request table and input queue of the coalescer. Instead, we compute the total amount of bits which must be held in the coalescing system at any time and model the required storage using D-FlipFlops.

In several modern GPUs, shared memory (SMEM) and the L1 data cache are portions of the same physical memory structure. The distribution of physical memory to L1 and SMEM is configurable. Therefore, we model L1 and SMEM as an integrated physical memory structure and convert accesses to SMEM and L1 hits to accesses to that memory structure. The physical memory consists of multiple banks to be able to supply multiple accessing threads with data at a high rate. Besides the physical memory banks, the SMEM/L1 consists of interconnects for addresses and data, both modeled as crossbars, and a bank conflict checking unit [159]. The L2 cache is shared over the entire GPU and connected to the cores through the NoC.

Global Memory

The global memory in GPUs has high bandwidth but long latency. The current generation of GPUs such as Fermi use either DDR3 SDRAM or GDDR5 SGRAM chips to implement the global memory. The power consumed by typical DDR or GDDR chips can be divided into background, activate, read/write, termination, and refresh power [160]. We extract numbers for each of these components from industry data sheets [161].

5.1.4 *Deriving Power Empirically*

Using our custom power measurement setup described in Section 5.2.1, we were able to build an empirical, measurement-based model for the INT and FP execution units by running custom microbenchmarks to estimate the energy per INT/FP operation. As described earlier in Section 5.1.3, todays GPUs use a SIMT approach to execute multiple threads at the same time on SIMD hardware.

We can use this SIMT-style of execution to our advantage by enabling different numbers of execution units while keeping the activity of all other units, except for the register files, constant. This way, we can estimate power for the execution units with reasonably high accuracy. For both integer and floating point operations, we launch one thread block for each core and use 512 threads per block to ensure all cores and targeted execution units are busy. We used unrolling to make the loop overhead of our testing loops negligible. In the loop nest of our integer test code, we are simulating Linear Shift Feedback Registers while for the floating point case we are using Mandelbrot set iterations. In both cases, we are alternately configuring the test kernels to use 31 enabled threads per warp and 1 enabled thread per warp. With 31 instead of 32 active threads, both configurations use divergent branches, cause the same activity of the reconvergence stack and have the same execution time. We then calculate the energy difference between these two kernel launches and divide the result by the number of executed instructions, number of cores and difference in execution units enabled to arrive at an estimate for the energy used by a single execution unit executing a single instruction.

Our measurements show that integer instructions are using approximately $40\,pJ$ while floating point instructions are using about $75\,pJ$ per instruction. NVIDIA reports 50 pJ per floating point instruction [120]. The power model of the execution units is based on our measurements.

While we developed all component models to the best of our knowledge, there are areas of GPU architecture where publicly available information is especially scarce, such as the raster operations pipelines (ROPs) or fixed-function video decode hardware, which consume power due to leakage even if they are not used in GPGPU applications. While we cannot model such components accurately because of the lack of information, we know nonetheless that these components are part of the chip. To be able to account for the amount of power they contribute, we used our measurement equipment to build empirical models of "base power" for cores and core clusters (called TPCs[1] or, more recently, GPCs[2] in NVIDIA terminology). These base power numbers are derived by measuring core/cluster power and subtracting the power of all

[1] "Thread Processing Cluster"
[2] "Graphics Processing Clusters"

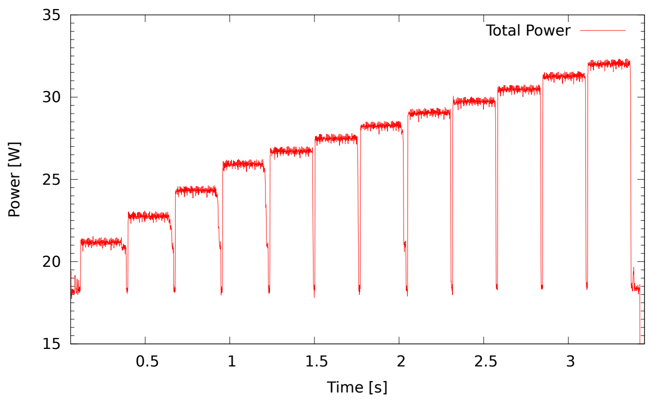


Figure 5.4: Power measurement results of a GT240 card running the same kernel 12 times with increasing number of thread blocks. The GT240 features 12 cores distributed evenly over 4 core clusters. [97] © 2013 IEEE

components we know about. An example of a measurement used to estimate cluster power is shown in Figure 5.4. The figure shows that increasing the number of thread blocks used for computation gradually increases the power, up to a certain limit when the entire chip is occupied (not shown). More interestingly, the figure shows that up to 4 blocks, adding another thread block to the computation increases power by a larger margin than beyond 4 blocks. The reason for this is the way the hardware scheduler distributes thread blocks: Until the entire chip is occupied, blocks are distributed first not only to unoccupied cores but also *to unoccupied clusters*, i.e. when a second thread block is added after the first one, it is placed on a core in another cluster than the first one. As we see in the figure, the activation of such a core cluster consumes 0.692 W additional power. Once all clusters are activated, in this case after the fourth block, adding more thread blocks increases power only by the power of the additional core. On another note, the figure also illustrates how the activation of the very first core/cluster consumes even more power than for the remaining clusters. This extra power (3.34 W) can be attributed to the activation of the global scheduler which distributes thread blocks to cores.

5.2 EXPERIMENTAL METHODOLOGY

To validate our GPUSimPow power models for contemporary GPUs, we must compare the simulator output to the power consumption of real hardware for various GPGPU workloads. Thus, in this section, we present our experimental

Table 5.1: Overview over the various GPGPU benchmarks used for experimental evaluation. [97] © 2013 IEEE

Name	#Kernels	Description	Origin
backprop	2	Multi-layer perceptron training	Rodinia
heartwall	1	Ultrasound image tracking	Rodinia
kmeans	2	k-means clustering	Rodinia
pathfinder	1	Dynamic programming path search	Rodinia
bfs	2	Breadth-first search	Rodinia
hotspot	1	Processor temperature estimation	Rodinia
matmul	1	Matrix-matrix multiplication	CUDA SDK
blackscholes	1	Black-Scholes PDE solver	CUDA SDK
mergesort	4	Parallel merge-sort	CUDA SDK
scalarprod	1	Scalar product of two vectors	CUDA SDK
vectoradd	1	Addition of two vectors	CUDA SDK

methodology, i.e. our test setup to measure power consumption both for validation of the simulator output as well as to infer power models for some of the irregular components. Besides the test setup, we also describe our test system configuration and the benchmark suite we used for evaluation.

5.2.1 *Measurement Equipment*

Our LPGPU1 testbed described in Section 4.2 was used to validate the simulator against real GPUs.

Table 5.2: Summary of the key features of the GPU architectures used in experimental evaluation. [97] © 2013 IEEE

Feature	GT240	GTX580
#Cores	12	16
#Threads per core	768	1536
#FUs per core	8	32
Uncore clock	550 MHz	815 MHz
Shader-to-Uncore	2.47×	2×
#Warps in-flight	24	48
Scoreboard	×	✓
L2-$ size	×	768KByte
Process node	40nm	40nm

5.2.2 *System Configurations*

For evaluating the output of the power simulator, we chose two real GPUs, the NVIDIA GT215 chip on a GeForce GT240 graphics card and the GF110 chip on a GeForce GTX580. Core parameters for both chips are given in Table 5.2.

The GT215 GPU is based on the GT200 Tesla design from 2009 and provides a good insight into many key features of modern GPUs. An initial advantage of using a GT200-based architecture is that the GPGPU-Sim simulator shows the highest correlation to real hardware for such architectures. The GF110 is based on the more recent Fermi architecture from 2010. The GT240 is a low-end card while the GTX580 is high-end enthusiast market card.

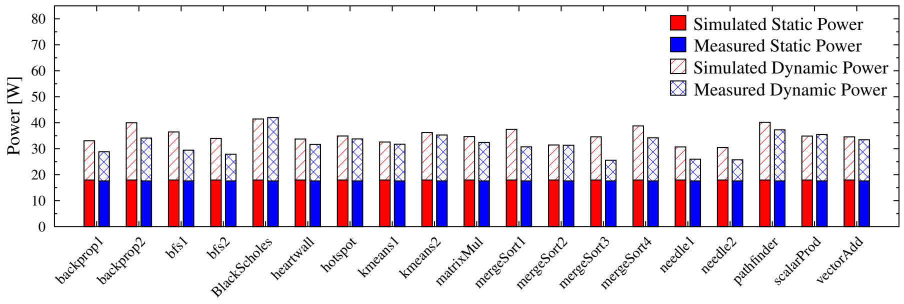
Table 5.3: Summary of our experimental setup. [97] © 2013 IEEE

Feature	Measurement	Simulation
OS	Ubuntu 10.10	Ubuntu 10.10
Kernel	2.6.35-22	2.6.35-22
NVIDIA driver	304.43	-
CUDA version	3.1	3.1
GPGPU-sim base version	-	3.1.1
McPAT base version	-	0.8

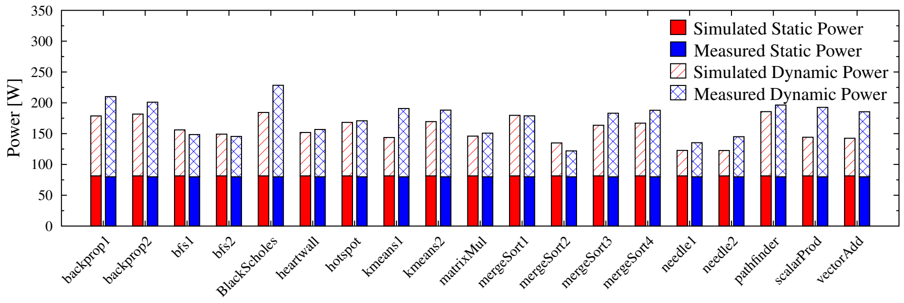
We performed both measurements and simulations for a series of kernels selected from recent GPGPU benchmark suites (see next Section 5.2.3) for each of the two GPUs presented in Table 5.2. For each kernel and GPU, we recorded hardware dynamic and static power, simulated dynamic and static power, and simulated as well as hardware execution time. Table 5.3 details the parameters of our experimental environment used to acquire the results. To estimate hardware static power, we ran the same benchmark at stock frequency and at a 20% lower frequency. Then, we performed linear extrapolation from the two data points to estimate the power the chip would consume at a frequency of 0 Hz. As Eq. 5.1 shows, there is no dynamic power consumption at 0 Hz and therefore, the result of the extrapolation must be equal to the static power of the chip. Unfortunately, using this methodology was only possible on the GT240 card, as the NVIDIA Linux drivers do not yet support changing the clock speed for the GTX580. Therefore, we estimate hardware static power for the GTX580 by measuring the idle power between two kernel executions and multiplying it by the ratio between idle power and static power we found on the GT240.

5.2.3 *Benchmarks*

The benchmarks whose kernels are used in our evaluation are shown in Table 5.1. As the table reveals, all benchmarks originate either from the popular Rodinia benchmark suite [162] or from the CUDA SDK [147]. These 11 benchmarks span not only a variety of application domains but, as Section 5.3.2 will show, an equally wide variety of algorithmic (and thus, dynamic power) characteristics. As our analysis focuses on the power consumed by the graphics card and GPU, we are only interested in the *GPGPU kernels* contained in each



(a) GT240



(b) GTX580

Figure 5.5: Measurement and simulation results for all benchmarks. Bars with the same benchmark name but different number, e.g. bfs1 and bfs2, correspond to different kernels of the same benchmark. Each bar shows the total runtime power, i.e. the sum of static and dynamic power. [97] © 2013 IEEE

benchmark. The second column of Table 5.1 shows the number of different kernels in each benchmark. In some benchmarks, there are kernels with very short execution times (less than 500μs). Because these kernels are too short for reliable measurements, we modified such benchmarks to execute the same kernels 100 times.

5.3 RESULTS

In this section, we present the simulation results for the benchmarks described in the previous section and compare these results to measurements. In Section 5.3.1, we describe the results from a per-benchmark perspective. Then, in Section 5.3.2, we show how modeling the GPU on the architectural level enables code developers and chip architects to generate *power profiles* that break down the power to the individual components on the chip.

5.3.1 *Simulated and Measured Power*

For each individual benchmark, we measured the total power consumed by the cards during the execution of each of the benchmark's kernels. For kernels that are executed multiple times during one benchmark run, we calculated arithmetic averages of all relevant power numbers. In the end, the power numbers we obtain are simulated and measured dynamic power and runtime *for each kernel* as well as simulated and measured static power *for the GPU*. As static power is consumed regardless of the circuit's switching activity, it is the same for each kernel. Table 5.4 shows the results from the static power and area estimation for both GPUs. The estimated chip areas are slightly smaller than the actual chip areas. However, our simulator provides a very good match for the static power consumption of both GPUs.

Table 5.4: Static Power and Area for GT240 and GTX580 [97] © 2013 IEEE

		Static [W]	Area [mm^2]
GT240	Simulated	17.9	105
	Real	17.6	133
GTX580	Simulated	81.5	306
	Real	80	520

The results of our experiments on the GT240 card are shown in Figure 5.5a. The figure shows total measured and simulated power for all benchmark kernels. Each total power bar in the figure is divided into a static power part common to all kernels and a kernel-specific dynamic power amount. Using the static power measurement technique explained in Section 5.2.2, we estimate the static power for GT240 to be 17.6 W. The card seems to do some power gating to reduce static power while no kernels are being executed. If no kernel was executed the card is using around 15 W, while for some milliseconds before and after the execution of a kernel the card consumes 19.5 W. About 90% of the power consumed by the card in this state thus seems to be static power. The GTX580 is using 90 W in the same state, so we estimate its static power to be 80 W.

In general, the figure shows a strong similarity between the measurement and simulation results for most benchmarks. For all benchmarks but `BlackScholes` and `scalarProd` the simulator overestimates the power used by the card. When averaging errors, we always average the absolute value of errors, so that under- and overestimates do not cancel out. On average the simulation is 11.7% off from the real power consumed by the GPU, we call this average relative error. The maximum relative error of 35.4% occurs in the third `mergeSort` kernel. This is likely a measurement artifact. The execution time of the kernel is short (1 ms) and the benchmark could not easily be changed to call it multiple times because the kernel does in-place processing of its data.

In nearly every benchmark kernel, the simulator slightly overestimates the true power consumed by the chip. This trend is mostly caused by the estimation of dynamic power, while the simulation result of static power is just 0.3 *W* (1.7%) larger than the real hardware power. Not considering static power, the average relative error in runtime dynamic power is 28.3%.

Figure 5.5b shows the results of our experiments using the GTX580 GPU. A strong similarity between measurements and simulations be seen again. Our empirically derived models also work well on this card even though they were obtained using the GT240 card. The average relative error for GTX580 is 10.8%. `scalarProd` is the kernel with the largest relative error (25.2%) on GTX580. The average relative error for the runtime dynamic power on GTX580 is 20.9%. GPUSimPow estimates the static power of GTX580 to be 81.5 *W* which almost completely matches our measurement result from the real hardware (80 *W*). As already explained in Section 5.2 we could not use measurements at different clock speeds on GTX580 to estimate the static power. As a result of this limitation, we used a different method to estimate the hardware static power. The better match of hardware static power with simulated static power could be a result of a more accurate static power estimate from GPUSimPow or it could be caused by the different hardware static power estimation methodology we used for GTX580.

5.3.2 *Power Profiling*

While relatively accurate estimations of dynamic and total power for the execution of a particular benchmark are helpful tools, in some cases, the *distribution* of power consumption over the hardware components on the GPU matters. As GPUSimPow contains hardware models for the internals of each core, interface and controller on the GPU, it automatically produces detailed power statistics for these internals. Therefore, it is possible to generate a *power profile* for a particular benchmark kernel that breaks the overall power down to individual components with the desired level of accuracy. Table 5.5 shows such a power profile for the `blackscholes` benchmark. Please note that this table does not include the power consumed by the external DRAM (4.3 *W*).

In the top part of the table, both static and dynamic power for the top-level components on the GPU is shown. It can be seen that by far the largest fraction of total power is, as one would expect, consumed by the GPU cores (82.2%). Previous researchers have reported similarly high percentages, for example in [163], the authors employed a simple, high-level power model to estimate the total core power to be 70% of the entire chip. According to the output of GPUSimPow, the next-most power after the cores themselves is

Table 5.5: Blackscholes benchmark power breakdown on GT240 for the individual components on the entire GPU (top) and a single GPU core (bottom). [97] © 2013 IEEE

		Static [W]	Dynamic [W]	Percent
GPU	Overall	17.934	19.207	100
	Cores	15.393	15.132	82.2
	NoC	1.484	1.229	7.3
	Memory Controller	0.497	1.753	6.1
	PCIe Controller	0.539	0.992	4.1
Core	Overall	1.283	1.031	100
	Base Power	0	0.199	8.6
	WCU	0.042	0.089	5.65
	Register File	0.112	0.173	12.3
	Execution Units	0.0096	0.556	24.43
	LDSTU	0.234	0.014	10.7
	Undiff. Core	0.886	0	38.3

consumed by the network on chip (7.3%), followed by the memory controllers (6.1%) for memory access and PCIe interfacing (4.1%). Note that some other top-level GPU structures such as the global scheduler and video decoder hardware are not modeled in detail and are therefore included in the (per-core) undifferentiated core and base power.

Increasing the level of detail, it is possible to look at the power consumed by the individual parts of a single core (bottom of Table 5.5). Overall, the core consumes 2.31 W. As the table reveals, surprisingly, the largest fraction of the total power is attributed to the core base power and undifferentiated core (8.6% and 38.3%). While the former includes all the *per-core* components we can only model empirically due to lack of information (see Section 5.1.4), the latter includes a per-core fraction of the *global* GPU components that can only be modeled empirically. Naturally, as we have no detailed models for undifferentiated components, we cannot generate any activity factors for them in GPGPU-Sim and thus the entire power consumption for the undifferentiated core is attributed as static power. Taking base power and undifferentiated core aside, the most power is consumed by the execution units (24.43%). After the execution hardware, the next-most power is used in the register file (about 12.3%). This number has been confirmed by previous research [163]. As one might expect from a SIMD architecture, the smallest part of the core power is consumed by the fetch and decode frontend, warp management, and schedulers (5.65%). GPUSimPow enables even more detailed analysis, e.g. investigating the power consumed by the different memories in the warp control unit or investigating the power impact of code sections with branch divergence on each hardware unit in detail. Further details of the per component power consumption can be found in a paper coauthored by the author of this thesis [164].

5.4 SUMMARY

In this chapter we presented the GPUSimPow power simulation framework. We illustrated the design of the various analytical models of the simulator as well as the measurement based models. We validated the simulator by comparing the simulator to two different GPUs and found an average relative error of 10.8% for the GTX580 GPU as well as an average relative error of 11.7% for the GT250 GPU. Our simulator allows us to estimate the energy consumption of short kernels and enables us to estimate the power consumption of individual GPU components. However, the GPUSimPow energy model only considers the number of instructions and their type but not the processed data. In the next chapter, we use a microbenchmark to show that the processed data can change the power consumption by more than 60% and present a power model to consider this data-dependent power consumption.

6 DATA-DEPENDENT ALU POWER MODELING

The work presented in this chapter was previously published: J. Lucas and B. Juurlink, "ALUPower: Data Dependent Power Consumption in GPUs," in *IEEE Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, ©2016 IEEE, 2016.

In the previous section we described our basic architectural power model GPUSimPow. This power model estimates the energy consumption of the GPU by counting various architectural events such as reading a value from the register file, scheduling and decoding an instruction or accessing the DRAM. However, while these architectural events are easy to count, they do not consider the data values used in these operations. In this chapter, we describe an improvement to our model, that allows us to also consider these data values when modeling the energy consumption of the ALU and the register file.

CMOS circuits dissipate dynamic power when they charge or discharge gates and wires. The power consumption of a circuit depends on how often each of the millions of gates and wires in the circuit change state. This activity of the circuit does not only depend on the circuit itself, but also on the data the circuit processes. RTL power simulators such as PowerMill [165] use test vectors to estimate activity factors for all signals.

Often, higher level simulators are required. Architectural power simulators are typically used in early design phases when circuit details are not yet known. Complex circuits consisting of thousands of gates are often abstracted into simple power macro models [166], [167]. Abstracting away individuals signals provides a tremendous reduction of simulation time and allows architectural power simulators to estimate power consumption for workloads that are out of the reach of circuit or RTL power simulators. Many of these high-level power modeling techniques do not model the data dependency of the power consumption [168]. To the best of our knowledge, no currently publicly available architectural simulator for GPUs takes the dependency of execution power consumption on actual data values processed into account. Modeling the energy consumption of the GPU datapath is important as it consumes

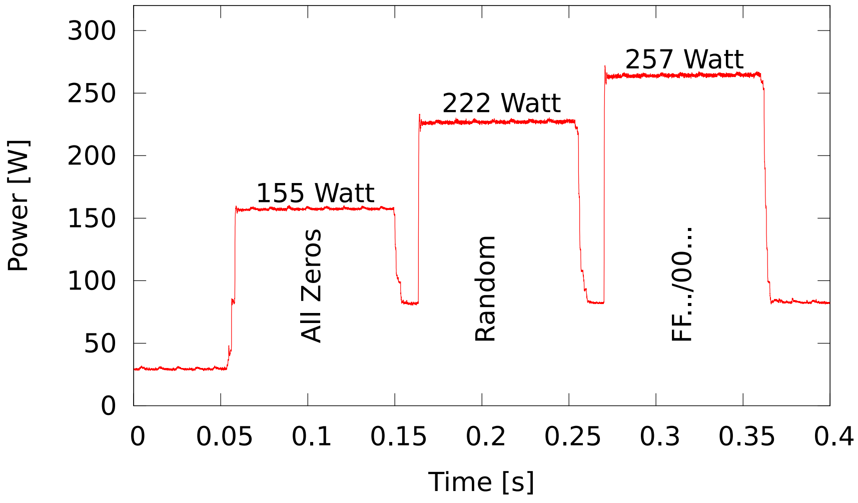


Figure 6.1: Power Consumption of FMUL test kernel with three different input vectors on GTX580 [39] © 2016 IEEE

significant parts of total energy consumption. Leng et al. [88] estimate that on average 44.9% of the GPU power is consumed by execution units, register files and pipelines. Our measurements reveal large changes of more than 100 *W* in total power consumption depending on the data values processed by the datapath. We executed a small test kernel on an NVIDIA GTX580 GPU with three different input vectors. The results of this experiment are shown in Figure 6.1. The test kernel reads input vectors from DRAM and then executes FMUL operations on these inputs. First, the power consumption of an all zeros test vector is measured. Executing the test kernel on this input keeps the floating point ALUs and most parts of the datapath idle. On this input, the GPU consumes 155 *W*. Then we use random input vectors, where each thread works on different values and on average half of the floating point multipliers input bits flip each cycle, this results in an increase of 77 *W*. Finally we execute the test kernel with an input vector of all zeros and all ones, where all input bits switch each cycle. In this case the total power consumption is 102 *W*; 65.8% higher than with all zero values.

We observe that the energy consumption of the three kernel runs differs substantially, even though all three kernels execute exactly the same number of instructions, the execution order has not changed nor the bandwidth consumed. Only the data values processed are different. The activity factors used by simulators such as GPUWattch [88] or our own GPUSimPow [97] for all three kernel executions would be completely identical. As a result, the predicted power consumption would also be identical. Therefore, not taking data values into account results in large prediction errors for the power consumption of this microbenchmark. Microarchitectural modifications such as different schedulers that change the order of execution might change the energy consumption of arithmetic units significantly, but these changes cannot be evaluated using current simulators.

By developing an accurate yet light-weight power model of real GPUs, we enable many new kinds of microarchitectural optimizations. Even techniques that might increase the runtime or the number of operations could be beneficial, if they can provide a significant reduction in the average number of bit flips in the arithmetic unit.

This chapter is structured as follows. An overview of related work has already been provided in Chapter 3, Section 3.4. Section 6.1 explains relevant architectural details of the Fermi GPU datapath as well as their influence on the design of our microbenchmarks. Section 6.2 describes how the actual measurements were performed and how our test vectors were generated. Before looking at the details of ALU Power consumption on the Fermi GPU, in Section 6.3.1 we employ a portable CUDA microbenchmark to show that data values impose a strong influence on GPU power consumption, even on the cards using the latest Maxwell architecture [77] and embedded GPUs. Section 6.3.2 explains how the register file and data values influence the power consumption even if the input values supplied to the functional units are constant. The following Section 6.3.3 provides an overview of the results of our measurements. Section 6.4 builds the ALUPower power model for GPU ALUs using the measured values and provides an initial discussion of its accuracy. In Section 6.5 the accuracy of the power model is evaluated and compared to the constant energy per instruction models employed by current GPU architectural simulators. Conclusions for this chapter can be found in Section 12.1.

6.1 MEASURING GPU ALU ENERGY

Measuring the energy consumed by the GPU datapath requires knowledge about its structure and on how microbenchmarks can be designed that trigger specific test patterns at the functional units and register files without large unwanted and unpredictable side effects. This section focuses on the datapath details of the Fermi architecture. A general introduction to the architecture of NVIDIA GPUs can be found in [76], [149] and in Chapter 2.

Figure 6.2 shows a simple GPU datapath with register files and integer and floating point arithmetic units. Many GPUs hide the latency of memory access and functional units by switching execution between multiple warps [5]. They also use very wide SIMD units. Because of these design choices, GPUs require

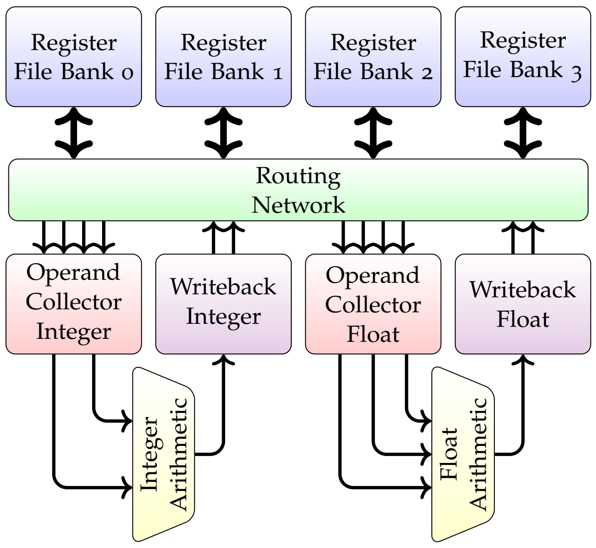


Figure 6.2: GPU register file and datapath (Figure reproduced from Chapter 2 for the reader's convenience.) [39] © 2016 IEEE

huge register files. The GTX580 GPU employed in this chapter contains register files with a total capacity of 2 *MB*. Conventional multi-ported register files enable high-performance CPUs to fetch many operands in a single cycle but consume large amounts of energy and die space even for register files of just a few kBs. In order to enable large register files, NVIDIA GPUs split their register file into several banks of single-ported SRAM [153]. Single-ported SRAM is more area and energy efficient than multi-ported RAM. Operand collectors connected via a routing network to the register bank fetch the operands in several cycles. If operands are distributed evenly over the banks, this organization can approach the performance of a conventional multi-ported register file. In our example, we show four register file banks.

The datapath of the Fermi architecture is even more complex (Figure 6.3). Each core (called streaming multiprocessor by NVIDIA) contains two warp schedulers. Each warp scheduler has its own register file. One of the warp schedulers executes all even warps, while the other warp scheduler executes all odd warps. Some execution units are exclusively connected to one warp scheduler, while others are accessible by both schedulers. Integer performance is half the floating point performance and the integer units are potentially shared as well or are just smaller units with lower throughput.

Unfortunately, we cannot build a power model by directly measuring the energy consumption of a single ALU while it performs a single instruction. Two main reasons prevent such an approach to GPU ALU power modeling.

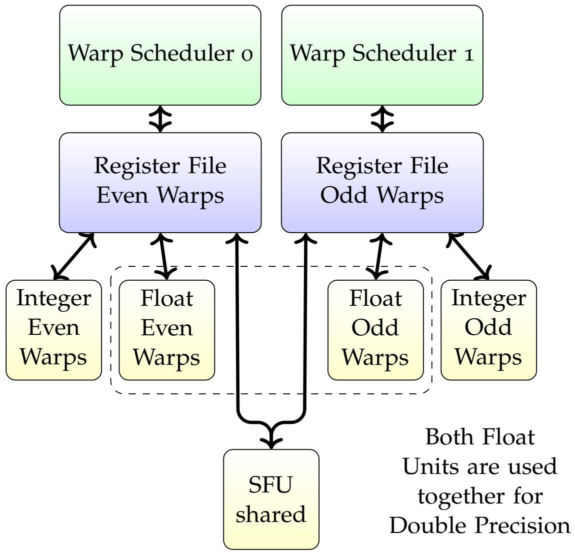


Figure 6.3: Fermi datapath (Figure reproduced from Chapter 2 for the reader's convenience.) [39] © 2016 IEEE

The first one is that the energy consumed in the execution of one instruction does not only depend on the instruction itself and its operands but also depends on the previously executed instruction and previous operands. A meaningful model must, therefore, measure the power consumption of pairs of instructions and pairs of input operands. The second reason is that we cannot isolate the ALU from other components of the GPU and measure only the power consumed by a single ALU. Isolated measurements of the power consumption of a single ALU are only possible using detailed circuit simulations or special test chips manufactured for such purpose. Both would require access to a wealth of proprietary design files and other secret information only GPU designers have access to. Estimating GPU ALU energy consumption using self-developed RTL descriptions of such an ALU would allow using circuit and RTL level power tools and measuring the energy in isolation. The energy estimations produced by such a model could be much higher than commercial GPU ALUs because they lack the many person-years of manual optimization performed on them, or they could also underestimate the actual energy consumption because they would likely lack some functionality available in commercial GPU ALUs.

The energy consumption of the ALUs can be estimated, without being able to measure only the ALU itself, by measuring the GPU power consumption twice. While changing the input operands to the ALU and keeping all other activities of the GPU constant, the energy consumption of the ALU can be

```
(...) // calculate test vector address in R2
// load test vectors into registers
LD.E R4,[R2+0x00]
LD.E R5,[R2+0x04]
(...) // load loop counter in R2
!Label loop
FMUL R4,R5,R6
(...) // 63x more FMUL
IADD R2,R2,-1
ISETP.EQ P0, P7, R2, RZ
@!P0 BRA !loop
EXIT
```

Figure 6.4: Microbenchmark Code in Fermi Assembly [39] © 2016 IEEE

calculated. As each operation uses only a tiny amount of energy, for precise measurements we need to repeat the operation many times and also execute the operation on many identical ALUs in parallel. This way differences in energy consumption that are within the pJ range per operation can be measured using energy measurements in the mJ to J range.

Measuring the GPU power consumption requires kernels that mostly execute a specific pair of instructions with a specific pair of inputs and nothing else. Writing such a code in a high level language such as CUDA or OpenCL would cause the optimizer in the compiler to spot that the code executes many redundant operations and eliminate most of the test code. Even if the test code is written using an intermediate language such as SPIR, HSAIL or PTX, optimizers in the driver can still remove parts of the test code or reorder instructions. Both would prevent valid measurements of the GPU power consumption. For this reason, the test code needs to be written in the actual ISA of the GPU and should run without changes on the GPU. Unfortunately, most GPU manufacturers do not support that developers write code directly in the actual ISA of their GPUs. NVIDIA provides a disassembler for the ISA, but does not provide an assembler nor a description of the instruction set. The disassembler allowed Yunqing to reverse engineer an assembler for Fermi called asfermi [169]. Not all instructions are supported yet, but enough instructions are supported to test the most important functional units: integer arithmetic, logic and basic floating point operations such as FADD and FMUL. Our test code first loads the test vectors from DRAM into registers and then executes instruction pairs in a loop with aggressive unrolling. Unrolling ensures that the GPU mostly executes our test instruction pair and that the energy and time spent on executing loop handling code is negligible.

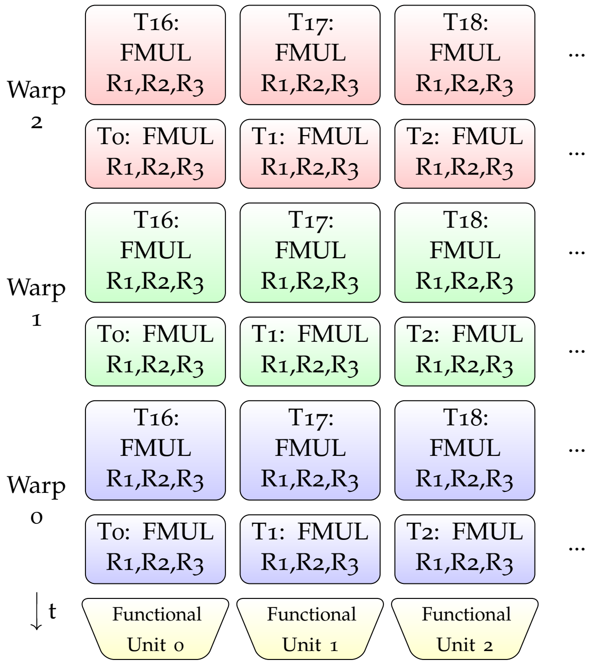


Figure 6.5: Execution of Warps on GPU Functional Units [39] © 2016 IEEE

Figure 6.4 shows a short version of our microbenchmark. First, we load the test vectors into registers and then we execute a long unrolled loop. Only one set of registers is used. Fermi GPUs use 32 threads per warp, but use only 16 wide functional units to execute each warp. First, the first half of the warp is executed, then in a second cycle, the second half of the threads is executed. This is illustrated in Figure 6.5, but for brevity, only 3 of the 16 units are shown. By loading different test vectors into the first and the second half of the warp, we can stimulate the functional unit with different test vector pairs and measure how much energy is consumed due to the transition. As the two halves of the warp are always executed together we could even execute multiple identical warps at the same time and would still get the same transitions, no matter how the warp scheduler schedules the warps. We, however, used only one active warp at a time to avoid additional interference at the register file.

6.2 Experimental methodology

We executed our test code on an NVIDIA Geforce GTX580 card based on NVIDIA's Fermi architecture. A short overview of its parameters is provided

Table 6.1: GPU configuration in experimental evaluation. [39] © 2016 IEEE

Parameter	Value	Parameter	Value
GPU cores (SMs)	16	Integer units / core	16
Warp Scheduler / Core	2	Float units / core	32
Core clock	1.5 Ghz	Memory clock	2 Ghz

in Table 6.1. By using a Fermi architecture card the test code could be written using asfermi. To measure the energy consumption, a power measurement test-bed similar to the ones used in GPUSimPow [97] and GPUWattch [88] was used. The NVIDIA CUDA command line profiler was used to gather kernel start and end times. Before the main test code was executed, a few test kernels were executed to generate a known series of power consumption spikes. These spikes were used to calculate the offset between the profiler clock and the sample clock. The power samples and the start and stop times from the profiler were then used to calculate the energy consumption of each executed kernel.

As we want to quantify the dynamic power of executing an instruction with different inputs, for every input vector we executed our measurement kernel twice: Once with the test vector and once with a baseline vector of all zeros. We assume that this all-zeros test vector triggers the minimum number of signal transitions and allows us to measure static power and the power used for fetching and scheduling the instructions. In both cases, we execute the same number of instructions in the same order. DRAM memory transactions triggered by our kernel are equal in both runs and are insignificant as our code runs from cache and only a few kilobytes of code and test vectors are loaded from DRAM on each kernel execution. Even though our baseline vector is the same for all measurements, we execute it again for every new test vector. This is required because the leakage power consumption depends on the temperature of the GPU. By executing both the test vector and the baseline test vector at nearly the same time and each only for a few milliseconds, the large thermal inertia of the GPU ensures that the temperature of GPU and thus the temperature dependent leakage power is almost constant in both measurements. All differences in energy consumption between the two kernel executions should, therefore, be due to the different input vectors.

We tested our measurement equipment and microbenchmarks by measuring the energy consumption of 10 values 100 times each for all configurations. A high repeatably of the measurements was observed and we found an average standard derivation of only $0.9\,pJ$ for repeated measurements of the same data point. Even this small measurement error will, however, add to our prediction error, as we cannot predict the noise.

A simple metric to characterize the value pairs in our test vector set is the Hamming distance; the number of bits that differ between two words. Initially, we tried using random test vectors but discovered that almost all random test vectors had medium Hamming distances and vectors with low or high Hamming distance were rare. We then improved our test vector generation. The Hamming distance of 32-bit values is between 0 and 32. With two inputs and one output test vectors, $33 \times 33 \times 33$ combinations of Hamming distances could exist. We tried to find 10 samples for each combination. As the output Hamming distance depends on the inputs for some combinations no test vectors or only a smaller number of test vectors could be found. For most instructions, we started with an initial set of around $220,000$ test vectors. This test vector set turned out to be too large, since it resulted in measurement duration of several days without noticeable improvement in accuracy compared to smaller sets. We then randomly selected two disjoint sets of $50,000$ values each from our initial set of test vectors. One set was used for fitting the coefficients, and the other set for initial validation.

6.3 experimental results

In this section we first perform an experiment to validate that data dependent energy consumption is not exclusive to Fermi GPUs. Then we perform more detailed measurements on Fermi GPUs, at first results obtained with static input values and then we present measurement results with changing test vectors.

6.3.1 *Data-Dependent Power Consumption on Fermi and Maxwell*

As explained in the last section, the assembler available for Fermi allows us to perform detailed measurements of Fermi GPUs, but we wanted to verify that our hypothesis, i.e. that data values strongly influence the power consumption of GPUs, is also true on more recent GPUs. For this reason, we designed a special microbenchmark that can be written in a high level language such as CUDA or OpenCL. The microbenchmark does not allow the detailed measurements required to build an ALU power model, but measuring its power consumption substantiates our hypothesis. The microbenchmark evaluates 32 linear feedback shift registers (LFSR) in parallel using a bit-sliced implementation. Figure 6.6 shows a 5-bit LFSR and the bit-sliced implementation. In this implementation, the state of each LFSR is not stored in a single register, but instead the state of LFSR 0 is stored in the bits at position 0 of registers `r0` to `r4`, the state of LFSR 1 is stored in the bits at position 1 of the same registers, and so on. Shifting instructions are not used but instead, the mapping between logical LSFR bits to physical registers rotates each cycle to account for the shifting. After unrolling, this results in a long chain of xor instructions in a loop.

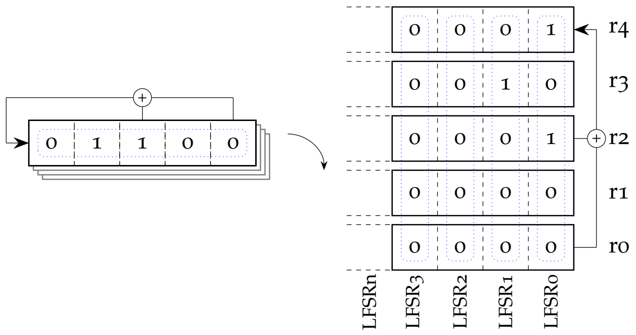


Figure 6.6: Bit-sliced Linear Feedback Shift Register [39] © 2016 IEEE

LFSRs are often used as pseudo-random generators, but have a special property that we exploit for our microbenchmark: An LFSRs with an initial state of all zeros stays in that state constantly, while any other initial state generates a pseudo random sequence. Controlling the initial state of the LFSRs allows us to adjust the number of bits that flip during the execution of this microbenchmark. If all our 32 LFSRs are loaded with all zeros, the power consumption should be low as the xor input will be constantly zero. If all LFSRs are loaded with a non-zero initial state the power consumption should be higher as the inputs will change. In Figure 6.6 LFSRs 2 and 3 are loaded with all zero bits and stay in the locked up state, while the state of LFSR 0 and 1 is changing in every cycle. As this code is written in pure CUDA we can execute it on the GTX580, but can also use a GTX750Ti card that employs NVIDIA's more recent Maxwell architecture.

Figure 6.7 shows the results of this experiment. The energy consumption displays an almost completely linear relationship with the number of active LFSRs on both cards. The total energy consumption of GPU is not shown in the diagram but in both cards the change between 0 LFSRs active and all 32 LFSRs active amounts to around 30% of average total energy consumption (GTX580: 28.8%, GTX750Ti: 31.2%). We scaled the working set size with the number of cores: the smaller GTX750Ti card only executes 5/16 the work of the bigger GTX580 card, but the GTX750Ti card with its Maxwell architecture is still almost 5 times as energy efficient. After the work in this chapter was initially published, we also ported this microbenchmark to OpenCL and executed it on a PowerVR G6430 GPU in an embedded SoC. The results of this experiment are shown in Figure 6.8. In this experiment the static power consumption was not subtracted as in the previous experiment. This experiment confirms that significant data-dependent power consumption is relevant on many platforms and is not limited to discrete NVIDIA GPUs.

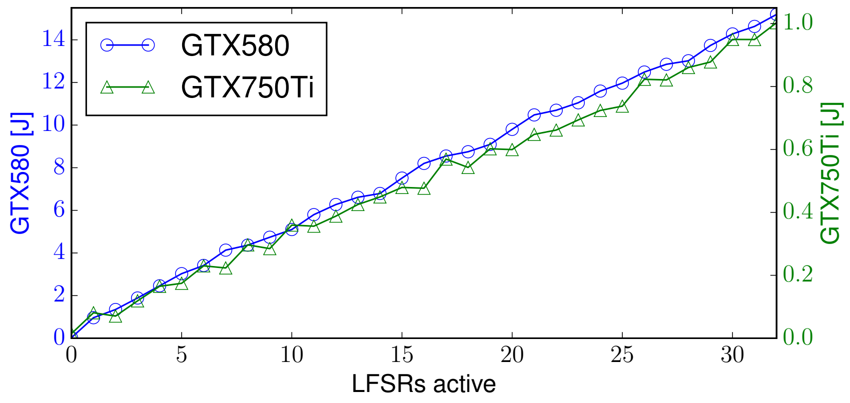


Figure 6.7: ALU energy consumption of GTX580 and GTX750Ti depending on the number of active LFSRs [39] © 2016 IEEE

It can also be shown to exist on mobile embedded GPUs from a different vendor.

6.3.2 *Impact of Register File*

We initially assumed that, if we execute the same instructions over and over again using constant inputs, the signals and gates of the GPU datapath should stay in a constant state and their power consumption should not change depending on which constant inputs are used. The measurements revealed, however, that this assumption is not true and that the energy consumption of the GPU datapath depends on the constants used and also on the warp that is executing the instruction. The results of these measurements, for even and odd warps, are shown in Figure 6.9a and 6.9b. We assume that the differences between even and odd warps are caused by differences in the physical layout of the execution units and register files or by different operand collector schedules. In this experiment, we measure the energy consumption of the GPU with various constant inputs relative to the energy consumption with all zeros as input. Negative values indicate that the operation with these inputs uses less energy than the same operation with all zeros as input. We picked test vectors with different Hamming distances between the operands and with different number of set bits in the operand, also known as population count (POPC). In Figure 6.9, below the energy measurements, we show the properties of the test vectors: on the top the Hamming distance between the two inputs, followed by the population count of the first operand (called a) and the second operand (called b). There is also a large difference depending

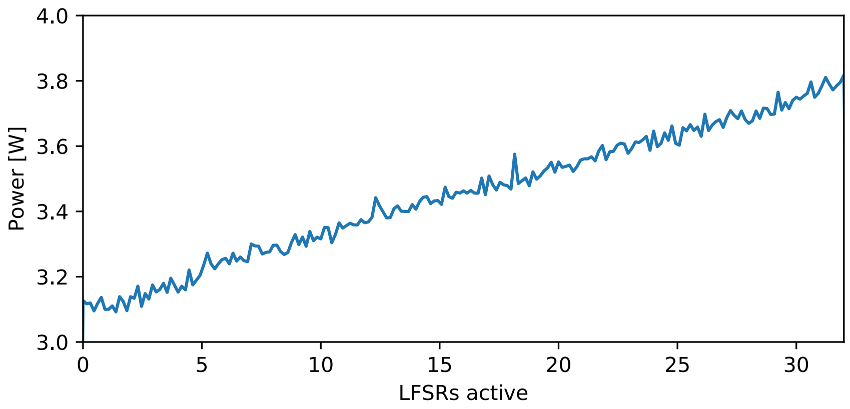
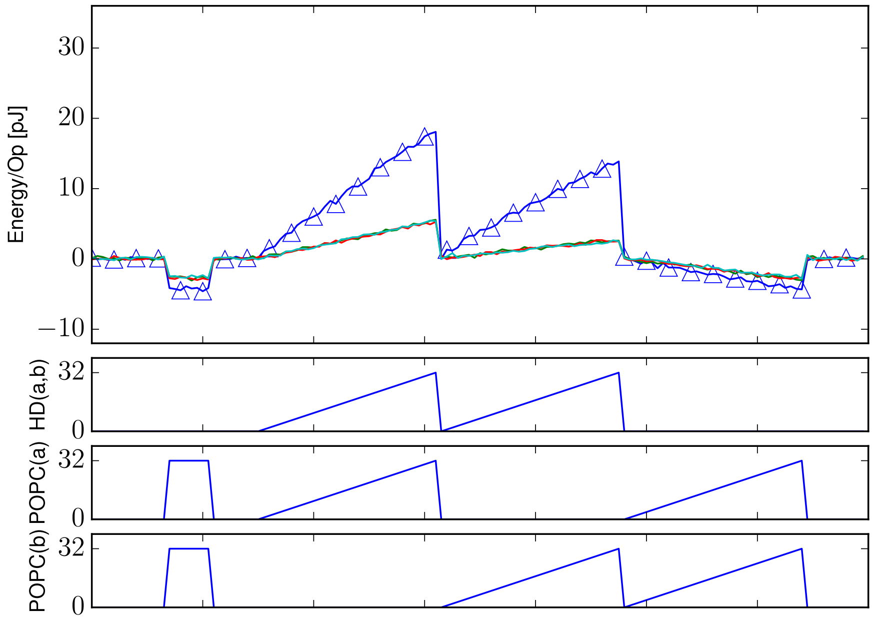


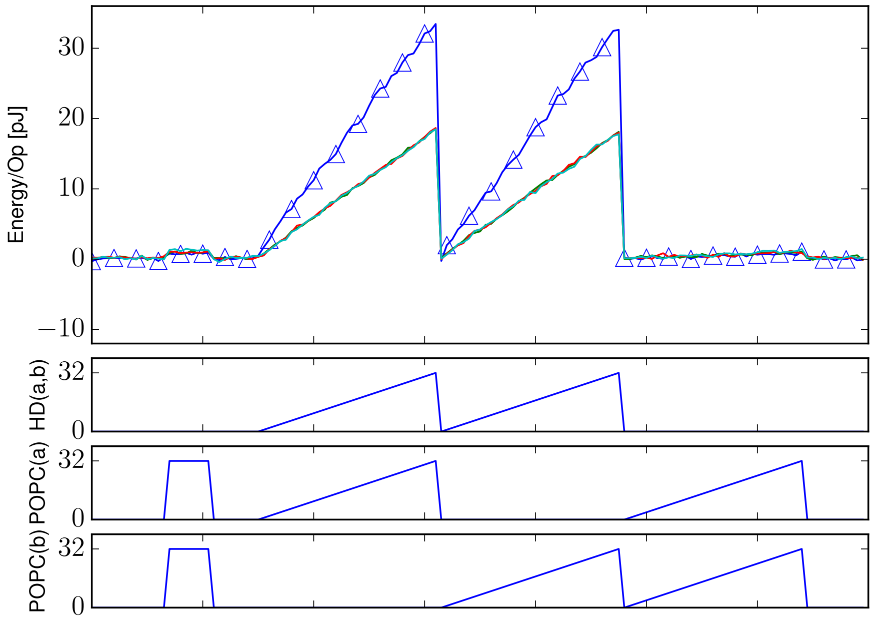
Figure 6.8: ALU energy consumption of PowerVR G6430 in Nexus Player depending on the number of active LFSRs

on the registers used to store the input values. As explained earlier NVIDIA GPUs use banked registers and we found that the power consumption is higher if the register number of registers a and b modulo 4 matches. This is shown in the triangle marked line with higher energy consumption. The three unmarked lines show experiments where the test vectors are fetched using different register banks. This makes it likely that 4 register banks are used and the higher power consumption happens when both operands need to be fetched sequentially from the same bank. On the other hand, if different banks are used each bank can stay in a constant state. This presents a power optimization opportunity for the compiler: During register allocation, the compiler should avoid fetching values with large Hamming distances from the same register bank.

Even if there are no differences between the first and second operand, a difference in power consumption exists depending on how many bits are set. In even warps, some input vectors use less power than our all zero base line. This can happen if some wires or gates are charged to 1 first and need to be recharged only if a 0 is transmitted but can stay constant otherwise. The energy consumed by the four different register banks is almost identical in our measurements. The same effects but smaller and with opposite sign happen in the odd warps. We also notice that in even warps set bits on input port b consumed less energy than those on port a. For many instructions input ports a and b can be swapped. Using this information the compiler could swap the input ports based on input statistics for a small power reduction.



(a) Even Warps



(b) Odd Warps

Figure 6.9: Register file energy consumption for LOP.AND [39] © 2016 IEEE

6.3.3 *Energy Consumption per Instruction*

After these initial experiments, we measured the energy for each instruction with pairs of test vectors. These results are employed in the next section to develop a model and evaluate the power model, but first, we look at the average power consumption of each instruction and how much their energy consumption differs with different input values.

The average energy consumption of AND, OR, XOR and IADD is very similar with 60.9, 60.9, 60.7 and 61.5 pJ, respectively. 95% of the AND instructions use between 28.3 and 91.5 pJ. OR and IADD instructions share a similar upper end with 97.5% of the instruction below 91.4 and 94.4 pJ, respectively. Upper bound for XOR is slightly lower at 81.3 pJ. Lower bounds are similar for OR, XOR and IADD with 28.8 and 30.6, 29.6 pJ, respectively. FADD on average uses 94.0 pJ while the average for FMUL is 95.3 pJ. 95% of the FADD instructions use between 36.8 to 135.0 pJ, while FMUL instructions consume between 49.3 to 131.5 pJ. IMUL is the instruction with the highest average power consumption (165.6 pJ) as well as the largest interval (76.7 to 218.3 pJ).

6.4 ALU ENERGY MODEL

Based on our previous findings we selected 6 parameters, one coefficient for every parameter, and one constant offset. The following equation describes how energy estimates are predicted by our model:

$$\begin{aligned} E(a_0, b_0, a_1, b_1) = c_0 &+ c_1 HD(a_0, a_1) + c_2 HD(b_0, b_1) \\ &+ c_3 HD(o_0, o_1) + c_4 HD(a_0, b_0) + c_5 HD(a_1, b_1) \\ &+ c_6 (POPC(a_0) + POPC(a_1) + POPC(b_0) + POPC(b_1)) \end{aligned} \tag{6.1}$$

In this equation a_0 and b_0 are the inputs of the first executed instruction and a_1 and b_1 are the inputs of the second executed instruction. o_0 is the output of the first instruction and o_1 is the result of the second instruction. Four parameters are Hamming distances (HD) between the input operands. Parameters $HD(a_0, a_1)$ and $HD(b_0, b_1)$ represent changes to the input wires. These parameters reflect the energy consumed by the wiring to the functional unit. The parameters $HD(a_0, b_0)$ and $HD(a_1, b_1)$ are designed to catch interference between two operands from different input ports of the functional unit. This either happens in the internal logic of the functional unit or in the register file. In addition, our model uses the Hamming distance of the two different outputs ($HD(o_0, o_1)$), and based on our findings from the register file also one parameter for the population count of the inputs. For this parameter, we do not differentiate between the inputs and add together the population counts (POPC) of all four input values. The coefficients c_0 to c_6 depend on

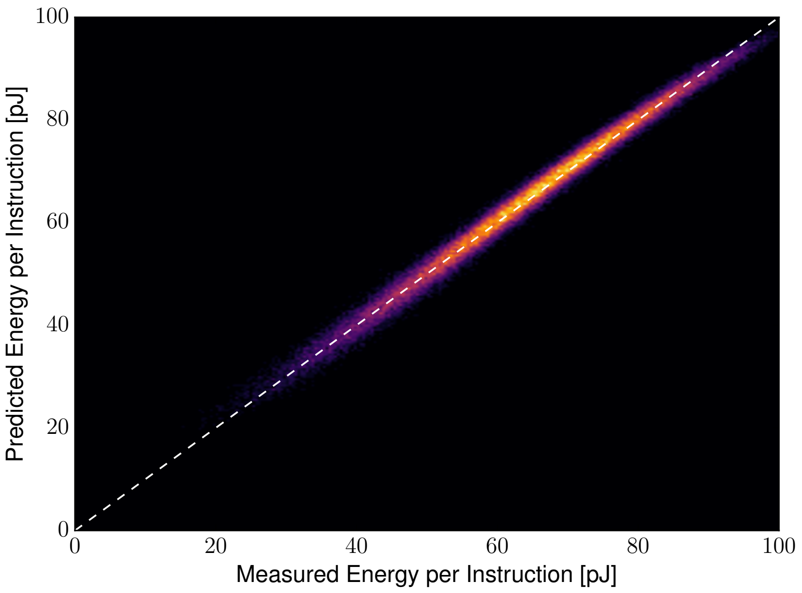


Figure 6.10: Predicted Energy vs. Actual Energy for LOP.AND (Even Warps) [39] © 2016 IEEE

the instruction type, even or odd warp, and the register banks used by input registers.

One limitation of this model is that it does not model state changes within functional units due to different operations. For example, OR and AND will likely be executed by the same functional unit. Even if their inputs and the output do not change, changing the executed operation will likely change parts of the internal state, which will consume additional energy. Modeling the energy consumption of these internal state changes is considered future work. ALUPower estimates energy consumed. To estimate power the architectural simulator must accumulate the energy estimates and divide the accumulated energy by the collection period. Because ALUPower is a linear model, the energy does not have to be evaluated each time an instruction is executed, but the input parameters can be accumulated and the energy can be calculated at the end of each evaluation period. Hamming distances can be calculated very quickly, especially when hardware population count instructions are available such as the x86 popcnt instruction.

We used linear minimal least squares fitting on our measured data points to determine the values of the coefficients c_0 to c_6. One set of coefficients was calculated for each instruction and for each of the four different combinations

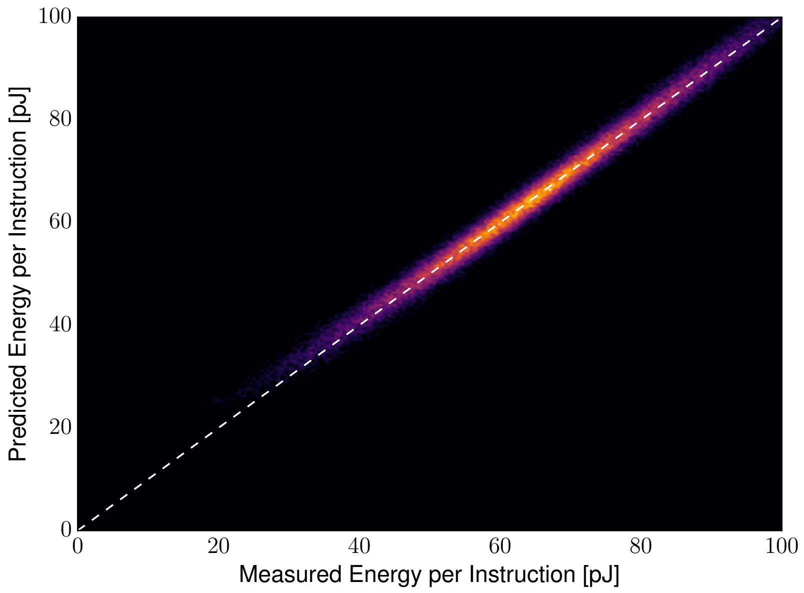


Figure 6.11: Predicted Energy vs. Actual Energy for IADD (Even Warps) [39] © 2016 IEEE

of even or odd warp and input registers from the same bank or from different banks. To evaluate our model we measured the energy consumption of another set of test vectors disjoint from our initial set. Then our coefficients were used to predict the energy consumption of these new test vectors. After these steps, for each test vector the actual, as well as the predicted energy consumption, is available. Heat maps are used to show how many samples we found for each combination of predicted and measured energy. The heat-map for the LOP.AND instruction is shown in Figure 6.10. In a perfect model together with perfect noiseless measurements, all points would fall on the diagonal dashed white line. For samples that are above the diagonal the predicted energy is higher than measured energy, while for samples below the diagonal the predicted energy is lower than the actually measured energy. Samples that are further from the diagonal have a larger prediction error. The prediction error stems from a different source: limitations of the power model, but also measurement noise. Samples in the center of the heat-map are more common because test vectors with few or many bit flips are relatively rare compared to test vectors with average bit flip statistics. If measurement noise and the test vector distribution is taken into account, the model's predictions of the AND instruction are very close to optimal. As discussed in Section 6.2,

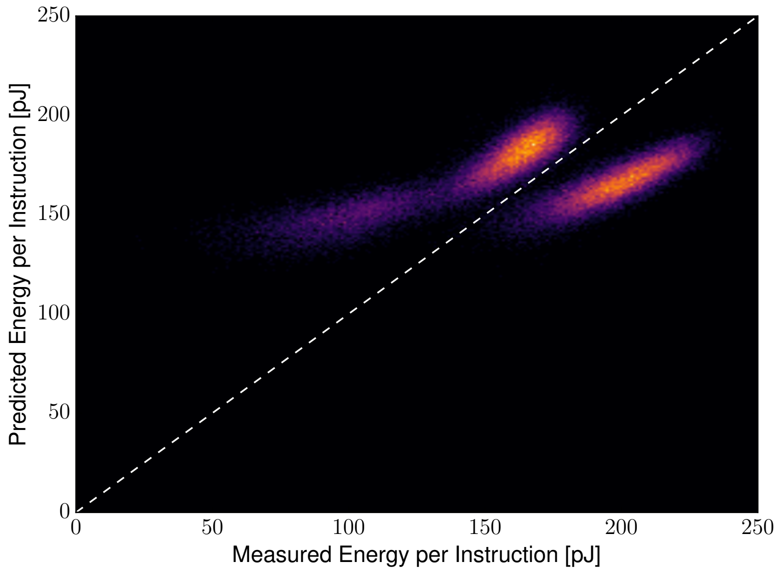


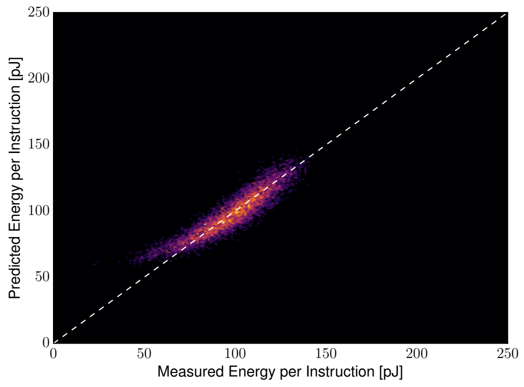
Figure 6.12: Predicted Energy versus Actual Energy for IMUL (Even Warps) [39] © 2016 IEEE

Table 6.2: IMUL test vectors from each category [39] © 2016 IEEE

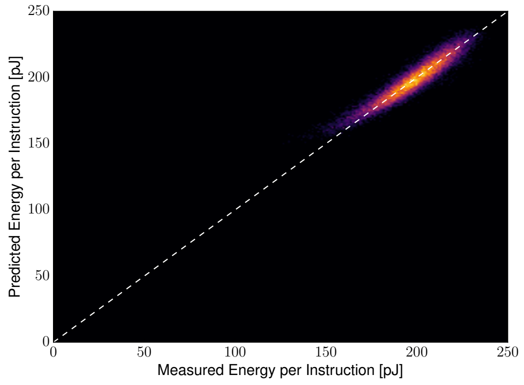
0 sign bit flips		**1 sign bit flips**		**2 sign bit flips**	
Op 1	***Op 2***	***Op 1***	***Op 2***	***Op 1***	***Op 2***
3×5	2×1	3×5	-2×1	3×-5	-2×1
-2×-8	-1×-4	-2×8	-1×-4	-2×-8	-1×-4
1×-2	7×-3	1×-2	7×3	1×-2	-7×3

even with a perfect model, electrical noise would cause an average prediction error of 0.9 pJ and our prediction error for AND is just slightly larger at 1.3 pJ. Heat-maps for the two other logic operations OR and XOR show very similar results and are therefore not shown. Figure 6.11 shows the prediction heat-map for the IADD instruction and again the predictions are very accurate, even though IADD is more complex than simple logic operations. No internal circuit details such as the state of the carry chain are modeled, but the energy model still performs very well.

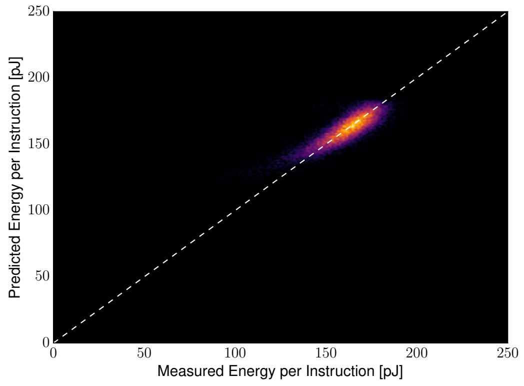
Figure 6.12 shows the results of our initial model for the IMUL instruction. Here we notice a different picture: Instead of a line close to the diagonal three clouds of samples can be observed, two of them above the diagonal, one below. We extracted some test vectors from each cloud and searched for a property



(a) No sign bit flips



(b) One sign bit flips



(c) Both sign bits flips

Figure 6.13: Predicted Energy vs. Actual Energy for IMUL for each category (Even Warps) [39] © 2016 IEEE

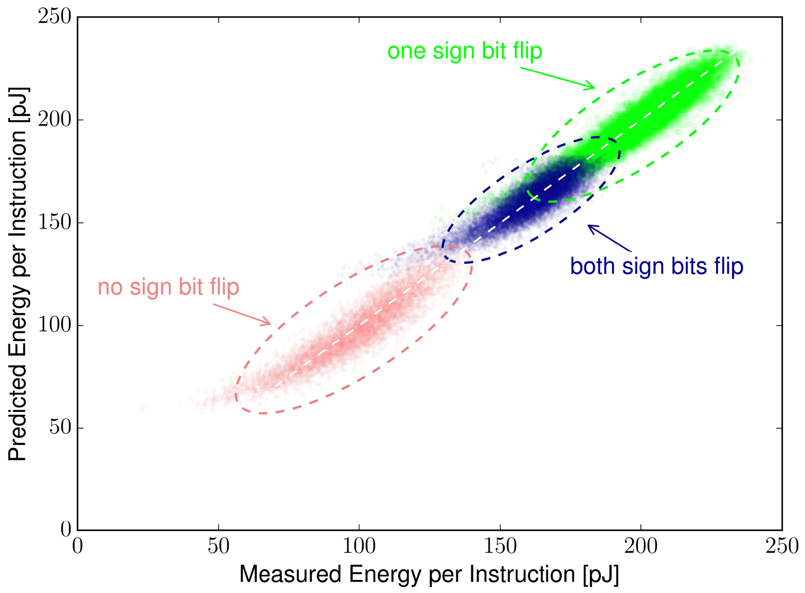


Figure 6.14: Predicted Energy vs. Actual Energy for IMUL using classification (Even Warps) [39] © 2016 IEEE

that can be used to classify the test vectors. The important difference between the samples are the sign bits of their test vectors. We classified each IMUL test vector into three categories and fitted different coefficients for each category. Table 6.2 shows example test vectors to clarify the categories: If all sign bits stay constant this results in the lowest power consumption. The results of our prediction for this category is shown in Figure 6.13a. The highest power consumption appears if only one of the sign bits flips (Figure 6.13b). Very likely the reason for the behaviour is the flipped sign of the output. If both input sign bits flip, then the output sign does not change. In our measurements, this results in a slightly lower power consumption than if only one sign bit flips as shown in Figure 6.13c. This property has been integrated into our model by using three different sets of coefficients for the IMUL instruction. Figure 6.14 shows the results of the integrated model. A scatter plot instead of a heat map is used to also display the classification of each data point. Most points are close to the diagonal now, albeit not as close as the points for the simpler logic and IADD instructions.

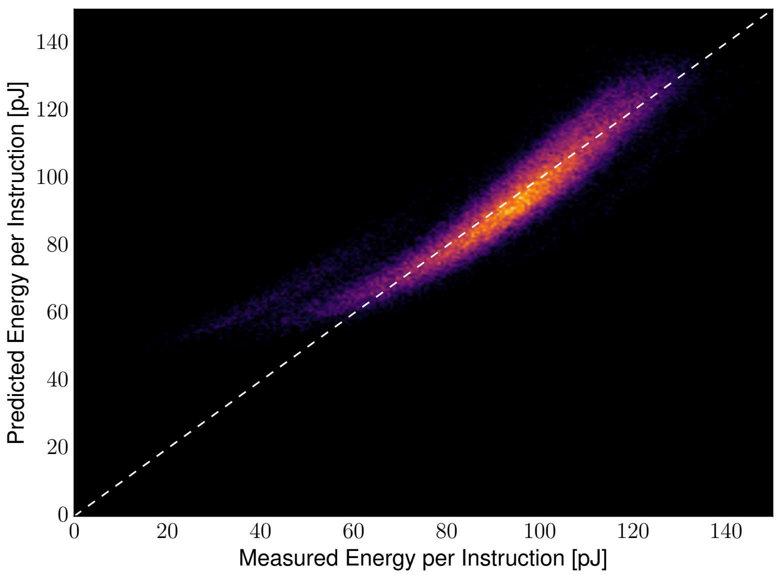


Figure 6.15: Predicted Energy vs. Actual Energy for FMUL (Even Warps) [39] © 2016 IEEE

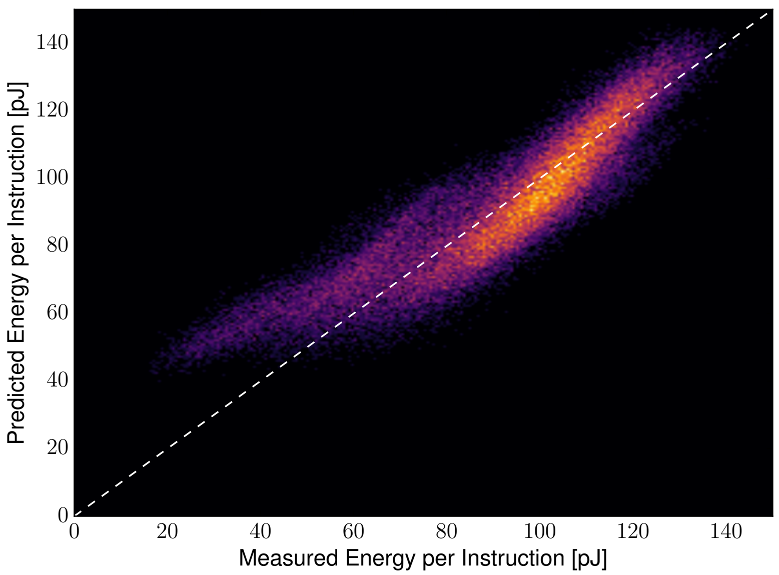


Figure 6.16: Predicted Energy vs. Actual Energy for FADD (Even Warps) [39] © 2016 IEEE

Results for the floating point multiply and add operation are shown in Figure 6.15 and 6.16. For these two operations, our model is not as accurate as for integer and logic instructions but much better than our first version of the IMUL model. The heat maps seem to indicate that there might be a property of the test vectors that influences the energy consumption that is not incorporated in our model. We tried to improve the accuracy of the model for floating point instructions by considering the exponent values (for example, if the exponents of the FADD differ a lot, the output will be close to one of the inputs), but this did not improve the prediction accuracy. In order to provide an even more accurate energy model of the floating point unit, additional insight into the design of this specific execution unit is required.

Table 6.3: Coefficients for Odd Warps, same register file bank, Coefficients in pJ

Coeff.	c_0	c_1	c_2	c_3	c_4	c_5	c_6	Avg. Energy
	1	$HD(a_0, a_1)$	$HD(b_0, b_1)$	$HD(o_0, o_1)$	$HD(a_0, b_0)$	$HD(a_1, b_1)$	$POPC$	-
LOP.AND	17.97	0.82	0.93	1.00	0.06	0.52	-0.09	*64.10*
LOP.OR	5.18	0.82	0.93	1.00	0.07	0.52	0.11	*64.16*
LOP.XOR	12.32	0.87	0.99	0.72	0.10	0.54	0.01	*64.09*
IADD	13.97	0.87	0.98	0.71	0.06	0.51	0.01	*65.02*
IMUL no sign	43.09	1.97	3.32	0.38	0.15	0.57	0.06	*169.16*
IMUL one sign	135.22	1.57	1.97	0.13	0.14	0.59	0.00	*169.16*
IMUL both sign	120.55	1.05	1.00	-0.05	-0.37	0.08	0.05	*169.16*
FMUL	43.42	1.15	1.28	0.18	0.10	0.58	-0.03	*94.07*
FADD	45.94	1.52	1.41	0.31	0.01	0.42	-0.17	*92.73*

Table 6.4: Coefficients for Even Warps, same register file bank, Coefficients in pJ

Coeff.	c_0	c_1	c_2	c_3	c_4	c_5	c_6	Avg. Energy
	1	$HD(a_0,a_1)$	$HD(b_0,b_1)$	$HD(o_0,o_1)$	$HD(a_0,b_0)$	$HD(a_1,b_1)$	$POPC$	-
LOP.AND	14.64	0.63	0.95	0.99	0.06	0.12	-0.01	*56.34*
LOP.OR	4.54	0.63	0.95	0.98	0.07	0.11	0.15	*56.27*
LOP.XOR	10.38	0.68	0.99	0.76	0.09	0.13	0.07	*56.68*
IADD	11.65	0.67	0.99	0.74	0.06	0.11	0.07	*57.53*
IMUL no sign	43.45	1.77	3.33	0.38	0.08	0.11	0.12	*161.61*
IMUL one sign	134.20	1.36	1.98	0.14	0.11	0.16	0.06	*161.61*
IMUL both sign	117.75	0.85	1.01	-0.05	-0.34	-0.29	0.11	*161.61*
FMUL	42.46	1.07	1.35	0.17	0.08	0.17	0.14	*96.49*
FADD	45.80	1.56	1.39	0.31	-0.01	0.04	-0.04	*94.67*

Table 6.5: Coefficients for Odd Warps, different register file bank, Coefficients in pJ

Coeff.	c_0	c_1	c_2	c_3	c_4	c_5	c_6	Avg. Energy
	1	$HD(a_0,a_1)$	$HD(b_0,b_1)$	$HD(o_0,o_1)$	$HD(a_0,b_0)$	$HD(a_1,b_1)$	$POPC$	-
LOP.AND	12.26	1.34	0.76	0.92	-0.01	0.28	-0.06	*60.38*
LOP.OR	3.01	1.34	0.76	0.92	-0.01	0.28	0.08	*60.47*
LOP.XOR	8.36	1.38	0.80	0.72	0.01	0.29	0.01	*60.02*
IADD	9.66	1.39	0.80	0.70	-0.02	0.27	0.01	*60.82*
IMUL no sign	41.97	2.49	3.13	0.37	-0.03	0.24	0.07	*165.00*
IMUL one sign	132.21	2.08	1.78	0.14	0.02	0.30	0.01	*165.00*
IMUL both sign	114.88	1.56	0.81	-0.06	-0.41	-0.12	0.06	*165.00*
FMUL	39.56	1.66	1.09	0.17	0.01	0.32	-0.02	*89.94*
FADD	42.06	1.71	1.53	0.30	-0.08	0.18	-0.16	*88.65*

Table 6.6: Coefficients for Even Warps, different register file bank, Coefficients in pJ

Coeff.	c_0	c_1	c_2	c_3	c_4	c_5	c_6	Avg. Energy
	1	$HD(a_0,a_1)$	$HD(b_0,b_1)$	$HD(o_0,o_1)$	$HD(a_0,b_0)$	$HD(a_1,b_1)$	$POPC$	-
LOP.AND	5.90	1.53	0.98	0.86	0.01	0.06	0.03	*62.65*
LOP.OR	1.58	1.53	0.98	0.85	0.01	0.05	0.10	*62.64*
LOP.XOR	4.29	1.55	1.00	0.76	0.01	0.04	0.07	*62.03*
IADD	5.37	1.56	1.01	0.73	-0.02	0.03	0.07	*62.77*
IMUL no sign	41.51	2.67	3.33	0.37	-0.13	-0.10	0.12	*166.82*
IMUL one sign	130.04	2.25	1.98	0.14	-0.03	0.02	0.06	*166.82*
IMUL both sign	109.04	1.73	1.01	-0.06	-0.33	-0.29	0.11	*166.82*
FMUL	36.68	1.94	1.35	0.17	0.01	0.07	0.12	*100.51*
FADD	39.87	2.03	1.78	0.30	-0.08	-0.04	-0.03	*100.12*

The coefficients used in our model and the average energy per instruction are shown in Table 6.3. Coefficients for odd and even warps are different as odd and even warps are managed by different schedulers and layout differences cause differences in energy consumption. The coefficients for input registers from the different banks are shown in Table 6.5 and 6.6. The coefficients for input registers from different register banks are very similar. The main difference to the coefficients from Tables 6.3 and 6.4 are smaller coefficients for $HD(a_1, b_1)$, which is due to the reduced interference of the operands in the register file, as discussed in Section 6.3.2.

6.5 ACCURACY

After developing the energy model and performing an initial visual evaluation, we calculate the root mean square error (RMS error) for every instruction and compare it to the previously used constant energy models. These models do not consider data values but assume a constant energy consumption for each instruction. For these models, we assume that every instruction uses the average amount of energy for that instruction type, as this minimizes the average error of a constant energy model. Figure 6.17 shows the root mean square error for the constant-energy and ALUPower energy models. Our data-dependent ALUPower model is significantly more accurate for all instructions. The geometric mean of the RMS error is 85.6% smaller than the error of current architectural power models for GPUs. The largest accuracy improvement is for AND instructions, where ALUPower provides 91.9% more accurate predictions. Even for the FADD instruction our energy model is 60.5% more accurate. The constant energy model has different coefficients for different instruction types and odd/even warps, while GPUWattch and GPUSimPow [88], [97] only differentiate between integer and floating point instruction. These even simpler models would perform even worse than the constant energy model employed here.

Test vectors that consume large amounts of energy in the real GPU ALU should also consume large amounts in the model, i.e. measurements and predictions should be strongly correlated. Figure 6.18 shows the Pearson correlation coefficient of measurements and our predictions. A correlation coefficient of 0 implies no correlation, while 1.0 would be perfect correlation. Since the constant energy model does not consider the data values at all, its predictions are not correlated at all to actual power consumption and its correlation coefficient is 0 for all instructions. The ALUPower energy model exhibits a high average correlation of 0.976 between the actual energy consumption and the prediction. The OR instruction shows the highest correlation of 0.996, while the FADD and FMUL instructions show a correlation of

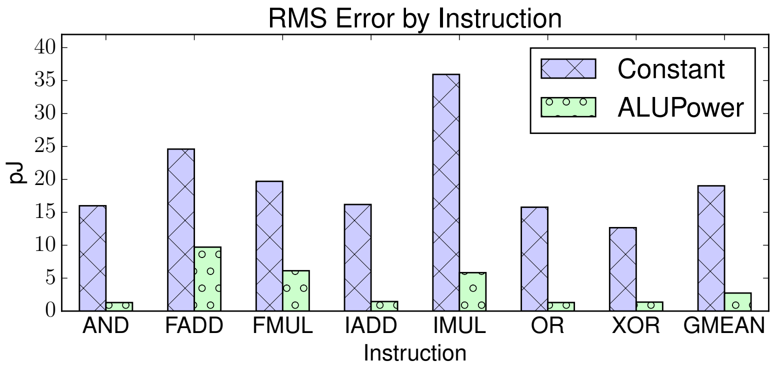


Figure 6.17: RMS Error of Constant-Energy and ALUPower [39] © 2016 IEEE

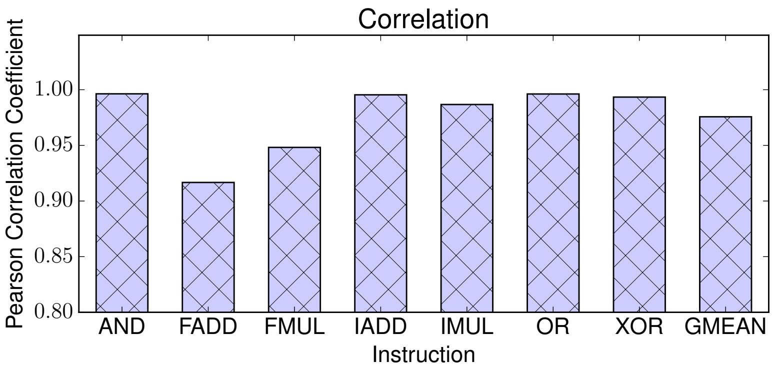


Figure 6.18: Pearson Correlation Coefficient [39] © 2016 IEEE

0.917 and 0.948, respectively. The predictions for IMUL have a correlation of 0.987.

The error results so far were calculated with test vectors with similar statistical properties to the test vectors used to perform the linear least square fitting of the model. The high accuracy on these test vectors could have been the result of overfitting. To validate that our model works well even with different test vector sets, we executed several Rodinia [162] GPU benchmarks and benchmarks included with the simulator on a modified gpgpu-sim[149]. For every tested instruction, gpgpu-sim was modified to extract value pairs used in thread 0 of each block and write them to a file. We randomly picked 1000 disjoint test vectors for each instruction, measured their power consumption on the actual GPU and compared them to the predictions. The

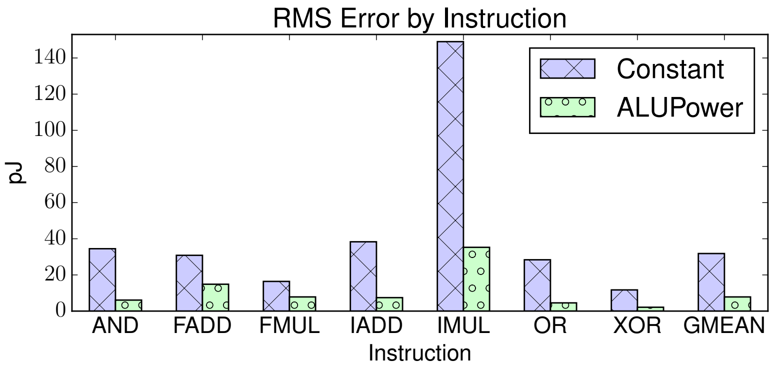


Figure 6.19: RMS Error of Constant-Energy and ALUPower (Validation Dataset) [39] © 2016 IEEE

results of this experiment are shown in Figure 6.19. On this set of test vectors, ALUPower has an average RMS error of 7.8 pJ, instead of 2.7 pJ on the generated test vectors. However, the error of the constant-energy model has increased from 19.0 pJ to 31.8, pJ. So the RMS error of the constant energy model increases even more than that of ALUPower. In this data set small unsigned integer values are much more common than in the generated data sets. For this reason, especially the constant energy model for IMUL predicts significantly worse.

6.6 SUMMARY

In this chapter, we have presented the ALUPower power model. While older models do not take data values into account, when estimating the energy consumption of instructions, our model uses population count and Hamming distances to enable significantly more accurate predictions. We designed custom microbenchmarks to measure the power consumption using an GPU assembler. We show that data-dependent power consumption exists on a wide range of GPUs including embedded platforms. We demonstrated that ALU power consumption depends on factors such as register allocation and warp scheduler. Our model improves the accuracy of the energy estimations by 85.6% over older models and shows a strong correlation of 0.976 to our measured results from a real GPU. After this chapter has presented ALUPower, a data dependent power model for the GPU ALU and register files, the next chapter reveals MEMPower, a data dependent power model for the memory subsystem.

7 DATA-DEPENDENT MEMORY MODELING

GPUs focus on applications with high computational requirements and substantial parallelism that are insensitive to latency [170]. Large caches are ineffective for GPUs due to the execution of thousands of parallel threads [171]. These factors cause GPUs and many GPU applications to require memory interfaces that provide significantly higher DRAM bandwidth than what is required and provided for regular CPUs. GPUs usually achieve the high memory bandwidth by using special graphics DRAM memories with lower capacity but wider and faster interfaces, such as GDDR5. These high throughput memory interfaces consume a significant amount of power. Modeling their power consumption accurately is thus important for architectural GPU power simulators.

In the previous chapter, we have shown that data values influence the energy consumption of GPU ALU operation significantly. While executing the same sequence of instructions the power consumption changed from 155 W to 257 W, when the processed data values were changed. In this chapter, we demonstrate that energy cost of memory transaction also is influenced significantly by the data values written to the DRAM or read from the DRAM. MEMPower provides predictions that consider the data values used in a transaction as well as the location of the transaction.

Most current discrete GPUs employ GDDR5 or GDDR5X memories [58], [59]. Both memory types employ pseudo open drain signaling (POD) [172]. In POD signaling, additional current flows when transmitting a zero, while no current flow happens when transmitting a one. To improve energy consumption as well as to limit the number of simultaneously switching outputs, both types memories use data bus inversion (DBI) [103], [109]. DBI encoding transmits data inverted if that results in a lower energy consumption and uses an extra signal line to allow the receiver to reverse the inversion of the data, if required. The POD signaling, together with DBI encoding, is a source of data dependent energy consumption of the memory interface.

CMOS circuits consume dynamic power when their internal circuit nodes are recharged to a different state. How much energy is consumed, depends on the load capacitance of this node and the voltages. Bus wires providing long

Table 7.1: GPU configuration in experimental evaluation.

Parameter	Value	Parameter	Value
GPU cores (SMs)	16	Integer units / core	16
GPCs	4	Float units / core	32
Core clock	1.5 Ghz	Memory clock	2 Ghz
CUDA	6.5	Driver	343.36

on chip distance routing are usually structures with high load capacitance. External off-chip interfaces, also contain large loads in their drivers, receivers, wires as well as parasitic package capacitances. How often each of the wires is recharged, depends on the data and the encoding of the data transmitted over the wire. The recharging of wires and other circuit nodes partly explains, why the energy cost of memory transaction depends on the transmitted data. The capacitance of a bus wire increases with increased length of the wire. For this reason, we also want to reveal information about the physical layout of the GPU to estimate distances.

Memory transactions are generated within the GPU cores (called streaming multiprocessors (SMs) by NVIDIA). In the GTX580 GPU the SMs are organized into graphics processor clusters (GPCs) [173]. Each GPC contains 4 SMs. The GTX580 uses a full GF100 die with all four 4 SMs activated in each of the 4 GPCs.

This chapter is structured as follows: Section 7.1 describes our experimental setup including our microbenchmarks. The following Section 7.2 shows how latency measurements can be used to discover the mapping between memory addresses and memory channels. It also describes the properties of the mapping and insights gained from latency measurements. Section 7.3 introduces the design of the data dependent energy model and evaluates the accuracy of the model. Section 7.4 provides a short summary and some conclusions for this chapter. Additional conclusions are provided at the end of this thesis in Chapter 12, Section 12.1.

7.1 EXPERIMENTAL SETUP

For our experiments, we used an NVIDIA GTX580 GPU with a full GF100 chip using the Fermi architecture [173]. A short overview of its parameters is provided in Table 7.1. This GPU was selected for two main reasons:

1. GPGPU-Sim currently does not support more recent GPU architectures. Energy was measured using the GPU power measurement testbed that has been described in Chapter 4.

2. The work presented in the last chapter resulted in a data-dependent power model for the ALUs of this GPU. This chapter adds the missing

memory power model to enable the creation of an architectural power model of the GTX580 GPU, that includes both ALU and memory data dependent power.

In order to measure the power consumption of memory transaction, we developed custom microbenchmarks. These microbenchmarks execute the tested memory transaction millions of times. This allows us to measure the small energy used per transaction. In order to measure only the data dependent energy of each transaction we measure every transaction twice: Once with the test vector and once with a baseline vector of all ones. Then the energy consumed by the baseline vector is subtracted to calculate the energy difference caused by the specific test vector. Both measurements are performed at nearly the same time to ensure that the GPU temperature stays approximately constant in both measurements to avoid errors. Without this step, GPU temperature variations could result in different amounts of static (leakage) power.

The microbenchmarks use inline PTX assembler to generate special load and store instructions that mostly bypass the L2 cache (`ld.global.cv.u32` and `st.wt.u32`). Even with these instructions, using the nvprof profiler, we detected that multiple accesses to the same address, issued at nearly the same time, are still combined at the DRAM. Our microbenchmark was then redesigned to avoid this issue by making sure that the different SMs are not generating accesses to the same location at nearly the same time. The profiler was used to verify that our microbenchmark generates the expected number of memory transactions. Each measurement was performed 128 times and averaged. The order of the measurements was randomized.

7.2 MEMORY LAYOUT

According to NVIDIA the GTX580 features 6 different memory channels [173]. CUDA allows us to allocate space in the GDDR5 but does not provide any control over which memory channels are used for the allocation. We suspected that the different memory channels might have different properties in terms of energy consumption due to different PCB layout of the memory channels as well as internal layout GF100 differences. To use all available memory bandwidth allocations are typically spread over all memory channels, so that all the capacity can be used and all memory bandwidth can be utilized. However, when we want to measure a specific memory channel we need to identify where a specific memory location is actually allocated. As no public API is available to query that information, we hypothesized that the differences in physical distance between the GPU cores and the memory channels would also result in slightly different latencies when accessing the memory. CUDA offers a special `%smid` register that can be used to identify the SM executing the code and a `%clock` register that allows very fine-grained time measurements. We used these two features to measure the memory latency of reading from each location from each SM. We measure the latency of each location 32 times and averaged our measurements to reduce measurement noise. For each location, this results in a 16 element latency vector, where each element of the vector shows the average memory read latency from that SM to the memory location. We detected that the latency to the same memory location is indeed different from different SMs and different memory locations show different latency patterns. We noticed that the latency pattern stays constant for 256 consecutive naturally aligned bytes. This means the granularity of the mapping from addresses to memory channels is 256 bytes, and we only need to perform our latency measurements once for each 256 byte block to identify the location of the whole block.

As the memory latency is not completely deterministic but changes slightly, e.g. due to background framebuffer accesses running in parallel to the measurement, all the latency vectors are slightly different. We solved this issue using k-means clustering [174]. We initially tried to map our latency vectors into six clusters corresponding to the six memory controllers listed in NVIDIA's descriptions of the GF100 [173]. This, however, failed to provide a plausible mapping of the memory locations, but mapping the latency vectors into twelve clusters was successful.

When we assume twelve clusters, all latency vectors are located close to one of the twelve centroids and the second closest centroid is much farther away. The number of points that gets assigned to each cluster is also approximately equal. When we access only locations mapped to one centroid, we achieve approximately 1/12 of the bandwidth achieved, when all locations from all channels are used. This pattern also continues if we selected larger subsets of the centroids, e.g. selecting locations from two clusters results in 1/6 of the bandwidth. The nvprof profiler also provides additional hints that the identified mapping is correct: Many DRAM counters are provided twice, one counter for something called subpartition 0 and another counter for subpartition 1. If we access only locations from a single cluster, we notice that only one of these two performance counters is incremented significantly, while the other counter stays very close to zero. This indicates all locations in each of the clusters are part of the same subpartition.

Lopes et al. list six L2 Cache banks with two slices each for GTX580 [175]. The GTX580 has a 384-bit wide memory interface. Six 64-bit wide channels together with the 8n prefetch of GDDR5 would result in a fetch-granularity of 64 bytes per burst. Memory access patterns that only access 32 consecutive bytes and do not touch the next 32 bytes would always overfetch 32 bytes per transaction and would result in an effective bandwidth of less than half the peak bandwidth. However, our experiments showed better than expected performance for 32 byte fetches. An additional hint at 32 byte

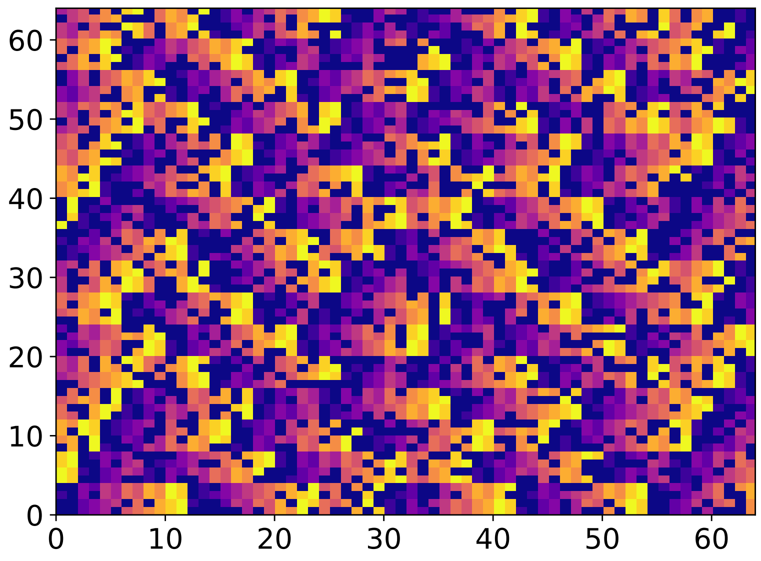


Figure 7.1: 1 MB Memory block with recovered memory channel mapping, each pixel is equivalent to a 256 byte block

transaction is also provided by the NVIDIA profiler, where many DRAM related performance counters are incremented by one per 32 bytes. This indicates that the GTX580 can fetch 32 bytes at a time, which is consistent with twelve 32-bit channels. From these findings, we estimate that the GTX580 uses six memory controllers with two subpartitions in each controller and one 32-bit wide channel per subpartition.

As twelve is not a power of two, the GTX580 cannot simply use a few address bits to select the memory channel. Round-robin mapping of addresses to memory channels is conceptually simple but would require a division of the addresses by twelve.

Figure 7.1 provides a graphical representation of the recovered memory mapping of 1 MB block of memory. Each pixel represents a 256 byte block, each of the 64 lines represents $64 \times 256B = 16kB$. The memory mapping seems to be structured, but does not use any simple round robin scheme. With this mapping twelve consecutive 256B blocks, on average, use 10.6 different memory channels. A simple round robin scheme would likely result in some applications having biased memory transaction patterns that favor some memory channels over others, which would result in a performance reduction. The mapping is likely the output of a simple hash function, that makes it unlikely for applications to use a biased memory access patterns

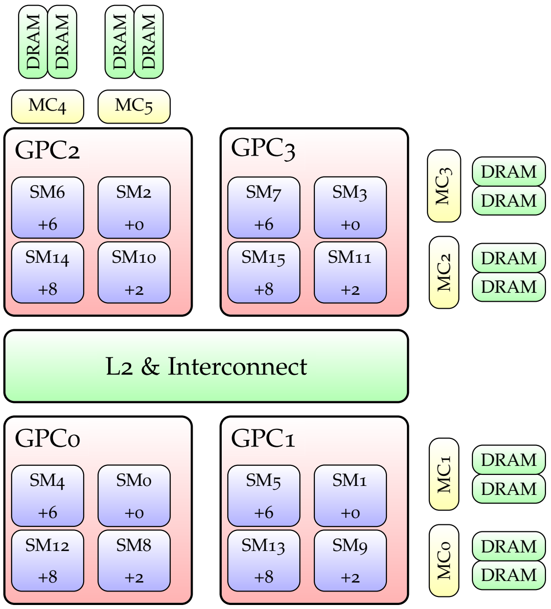


Figure 7.2: GF100 Organization

by chance. Sell describes a similar scheme used by Xbox One X Scorpio Engine [176].

We also analyzed the latency vectors to reveal more information about the internal structure of the GPU. DRAM Latency without load varies between 400 cycles and 430 cycles. We first notice that all SMs in the same GPC have nearly the same latency the memory channels. The first SM in each GPC seems to have the lowest latency. The other SMs are approximately 2,6 and 8 cycles slower. This additional latency within the GPC does not depend on the memory channels addressed. It is also identical for all four GPCs. This indicates an identical layout of all four GPCs and a shared connection of all SMs of a GPC to the main interconnect. The latency of four memory channels is lowest at GPC1. This is also true for GPC2 and GPC3. There are no memory channels where GPC0 provides the lowest latency. We suspect that is the result of a layout such as shown in Figure 7.2. This also matches well with the PCB layout of a GTX580 where DRAM chips are located on 3 of the four sides of the GF100 and the PCIe interface can be found at the bottom.

7.3 DATA-DEPENDENT ENERGY CONSUMPTION

As already described in the introduction, we expect two main reasons for data dependent energy consumption:

Table 7.2: DRAM Latency

GPC0	GPC1	GPC2	GPC3
Added Latency vs. GPC1			
3.6	-	3.6	7.5
3.9	-	3.8	7.5
7.4	-	3.7	11.2
11.3	-	5.7	15.1
Added Latency vs. GPC2			
3.6	3.8	-	0.0
3.8	3.8	-	0.1
9.4	3.8	-	5.8
11.2	2.0	-	7.6
Added Latency vs. GPC3			
4.0	7.6	4.0	-
3.9	7.7	3.9	-
3.9	9.5	5.9	-
3.8	9.6	5.7	-

1. Special signaling lines such as the GDDR5 DQ lines with additional energy consumption at a certain signal level.

2. State changes of wires and other circuit nodes.

Our model should allow a fast and simple evaluation, for this reason, we selected a simple linear model. Every memory transaction is mapped to a small vector that describes the relevant properties of the block. A dot product of this vector with a coefficient vector results in the estimated energy consumption for this transaction. The coefficient vector is calculated in a calibration process.

The following properties of the block are used to estimate the energy consumption. We model signal level related energy consumption by including the population count of the block. The population count is the number of set bits. We also need to estimate the amount of recharging of internal wires and circuitry caused by the transaction. Memory transactions travel through several units and various connection until they finally reach the DRAM. A simplified diagram is shown in Figure 7.3. We know that the transaction travels through a 32-bit wide interface between DRAM and memory controller. Unless a reordering of bits is performed, we know which bits will be transmitted through the same wire and could cause switching activity

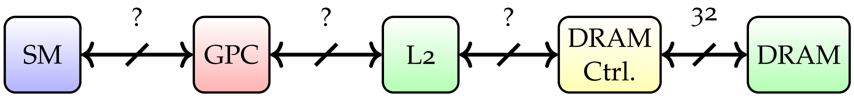


Figure 7.3: Memory Datapath

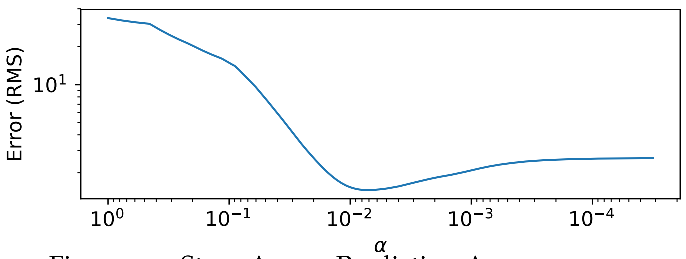


Figure 7.4: Store Access Prediction Accuracy vs. α

on these wires, e.g: bits 0, 32, 64, ... are transmitted on the same DQ line, bits 1,33,65, ... are transmitted on the next DQ line, etc. While we know the width of the DRAM interface itself, the width of the various internal interconnections is unknown. We assume the internal link width are powers of two and are at least byte wide. The coefficients for all potential link sizes are first added to the model. During the calibration of the model, the best subset of coefficients is selected, and we indirectly gain knowledge about the internal interconnections. Because GDDR5 memory can use DBI encoded data, an extra version of each of the previously described coefficients is added to our model. This second version assumes DBI encoded data.

A synthetic set of test vectors was generated to calibrate the model. The calibration test vectors are designed to span a wide range of combinations in terms of toggles at various positions and in terms of population count. We measured the real energy consumption of our test vectors. Initially, the model uses a larger number of coefficients and some of these likely have no corresponding hardware structure in the GPU. This causes a significant risk of overfitting the coefficients to our calibration measurements. We avoid this issue by using LASSO regression as an alternative to regular least square fit [177]. Instead of fitting the calibration data as closely as possible LASSO also tries to reduce the number of used coefficients and reduces their size. The hyperparameter α controls the trade off between number and size of the coefficients and prediction error with the calibration set.

In addition to the set of calibration vectors, we generated another set of test vector to validate our model. The validation vectors are generated to mimic real application data. The vectors use various integer and floating-point data types, a mixture of random distributions with different parameters was used to generate realistic data. Real application data is often also highly correlated, some test vectors used a Gaussian process to provide correlated data.

Figure 7.4 shows the prediction error at various values of α. $\alpha = 0.007$ results in the smallest error in the validation set for store transaction. Smaller values of α overfit the calibration set, while larger values discard important coefficients. Table 7.3 shows the coefficients, it should be noted that the

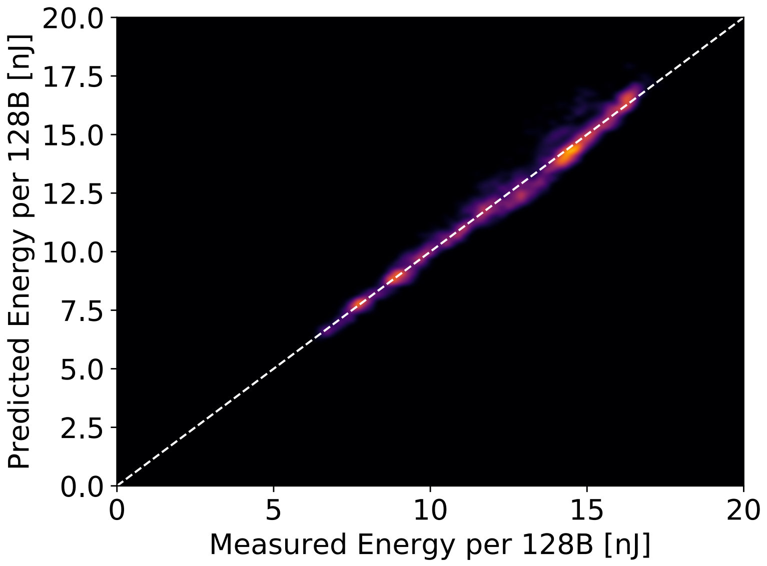


Figure 7.5: MEMPower energy prediction for store access

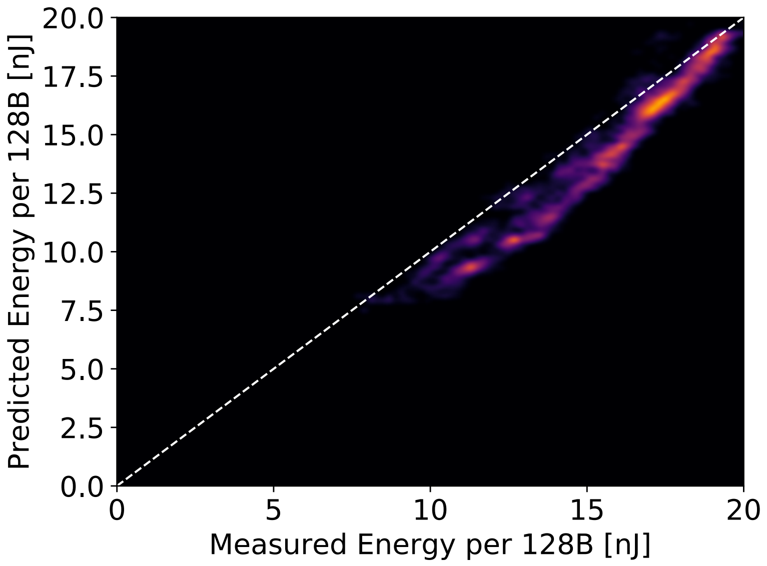


Figure 7.6: MEMPower energy prediction for read access

Table 7.3: 128B Transaction Coefficients

Store

Coefficient	DBI	Value (nJ)
Const	No	7.536
Pop Cnt.	No	-3.046
Pop Cnt.	Yes	-0.487
Toggle 1	No	0.013
Toggle 1	Yes	0.020
Toggle 2	No	0.004
Toggle 2	Yes	0.017
Toggle 4	No	0.930
Toggle 4	Yes	0.059
Toggle 8	No	0.782
Toggle 8	Yes	0.000
Toggle 16	No	2.272
Toggle 16	Yes	0.014
Toggle 32	No	9.341
Toggle 32	Yes	0.140
Toggle 64	No	5.150
Toggle 64	Yes	0.044

Load

Coefficient	DBI	Value (nJ)
Const	No	9.175
Pop Cnt.	No	-3.908
Pop Cnt.	Yes	-0.582
Toggle 1	No	0.020
Toggle 1	Yes	-0.043
Toggle 2	No	0.019
Toggle 2	Yes	-0.043
Toggle 4	No	1.677
Toggle 4	Yes	-0.007
Toggle 8	No	0.445
Toggle 8	Yes	-0.036
Toggle 16	No	1.039
Toggle 16	Yes	-0.049
Toggle 32	No	7.434
Toggle 32	Yes	0.043
Toggle 64	No	9.917
Toggle 64	Yes	1.901

coefficients were calculated per 512 bitflips for numerical reasons. None of the DBI coefficients are used, which indicates that the GPU is not using DBI encoding for stores. The largest coefficient corresponds to a 32 byte wide link. Coefficients for 4 and 8 byte wide links are small. Narrow 1 or 2 byte wide links are not employed. The large coefficient for a 64 byte wide link could be linked to SM internal power consumption, as the SMs use 16 wide SIMD units with 32-bits per unit.

The heatmap in Figure 7.5 shows the prediction accuracy of our model for 128 byte store transactions. If the model would offer perfect prediction all points would be on the dashed white line. However, all our predictions are very close to the line which indicate a great prediction accuracy. Our RMS error is $0.39\,nJ$ and the relative error is just 3.1%. Smaller transactions use different coefficients, results are not shown here because of the limited space. But one interesting result is that register values from disabled threads influence the energy consumption. Likely these register values are still transmitted through parts of the interconnect but marked as inactive. Taking data values into account instead of assuming a constant average energy per transaction improves the prediction error from an average error of $1.7\,nJ$ to a error of just $0.39\,nJ$.

Figure 7.6 shows the prediction accuracy of our load model. In general, the model achieves a good prediction accuracy of 9.1% but tends to underestimate the energy required for cheaper transactions. Our load kernel achieves a significantly lower bandwidth than the store kernel as it will not send the next load transaction before the last transaction returned, while stores will be pipelined. The lower bandwidth results in a reduced signal to noise ratio of the measurements. The load coefficients printed in Table 7.3 indicate that

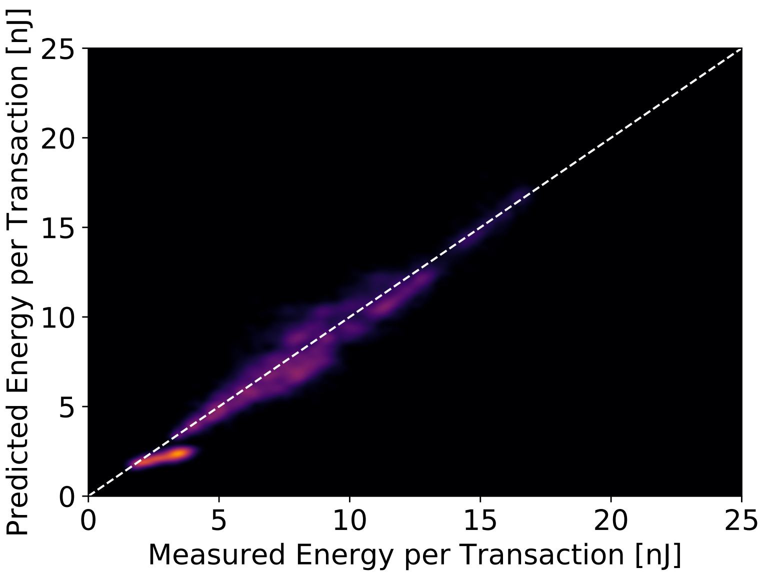


Figure 7.7: MEMPower energy prediction for store access for all transaction sizes

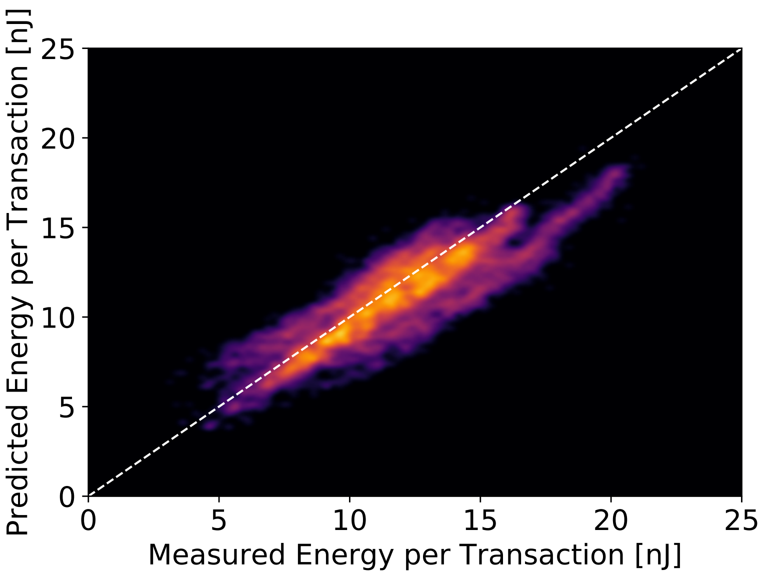


Figure 7.8: MEMPower energy prediction for read access for all transaction sizes

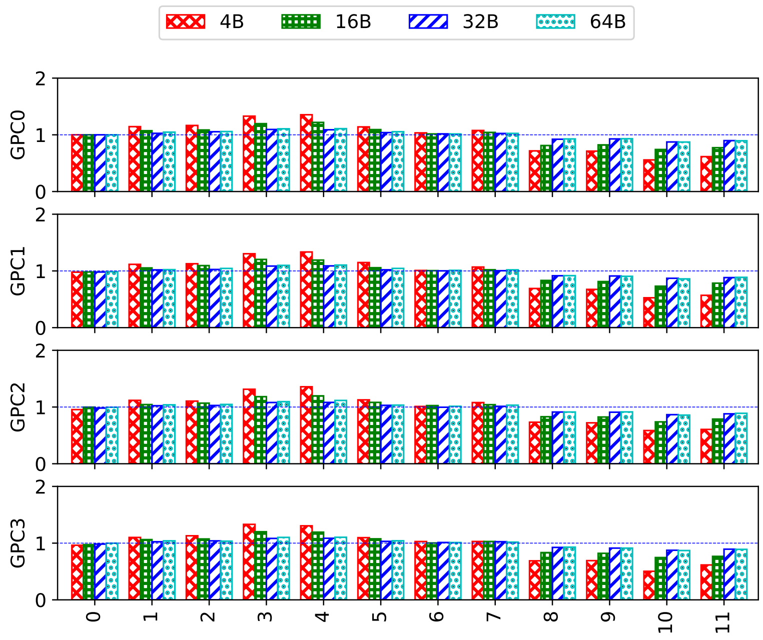


Figure 7.9: Normalized Memory Channel Energy Consumption

load transaction are employing DBI encoding. Error improves from 2.3 nJ to 1.43 nJ.

Our previous model is limited to transactions that use a full 128B transaction, we extended our model to also handle different transaction sizes, by training the model with different coefficients for each possible mask. Figures 7.7 and 7.8 show the prediction accuracy for memory transactions of all sizes and patterns. Energy consumption is more varied when these different access patterns are also considered. Write prediction error improves from 3.3 nJ to 0.8 nJ, prediction error for read transactions improves from 2.9 nJ to 1.6 nJ. Relative average RMS is 12.0% for writes and 13.1% for read transactions.

We combined the microbenchmarks with the memory channel identification technique from Section 7.2 to check for energy differences between different memory channels and SMs. We tested the first SM from each GPC and used simplified test vectors to check for changes of our most important coefficients. The normalized results are shown in Figure 7.9. We detected only small differences between the different SMs, however, the blue coefficient for switching activity on a 4 byte wide bus shows a large variance between different memory channels. Memory transactions to channels 8 to 11 are significantly cheaper than memory transactions on Channels 0 to 3 and 5 to

7. Memory transactions on Channels 3 and 4 are more expensive. As these results are consistent for all four GPCs, these differences are likely the result of slightly different PCB layout of the different memory channels instead of chip internal routing.

7.4 SUMMARY

This chapter continued our work on data dependent power models for GPUs. We have demonstrated how we can improve the accuracy of our power predictions by considering data values. We have shown that latency measurements can be used to identify the memory channel used for a memory transaction and how different memory channels differ slightly in their energy cost. We show how the LASSO algorithm can be used to improve the quality of the coefficients and increase the accuracy. MEMPower improves the average prediction accuracy by 37.8% for 128B loads and by 77.1% for stores compared to a non-data dependent model. With this chapter the power modeling part of this thesis ends and the next chapter starts the architectural enhancements part. In this chapter we have learned about data bus inversion (DBI) and in the next chapter we describe an improved DBI encoding that is able to reduce the interface power by up to 6%.

8 SAVING DRAM INTERFACE POWER

The work presented in this chapter was previously published: J. Lucas, S. Lal, and B. Juurlink, "Optimal DC/AC data bus inversion coding," in *Design, Automation and Test in Europe, DATE*, EDAA, 2018.

GDDR5 and DDR4 memories use data bus inversion (DBI) coding to reduce termination power and decrease the number of output transitions. Two main strategies exist for encoding data using DBI: DBI DC minimizes the number of outputs transmitting a zero, while DBI AC minimizes the number of signal transitions. We show that neither of these strategies is optimal and reduction of interface power of up to 6% can be achieved by taking both the number of zeros and the number of signal transitions into account when encoding the data. We then demonstrate that a hardware implementation of optimal DBI coding is feasible, results in a reduction of system power and requires only an insignificant additional die area.

Up to 50% of the power used by the memory is consumed by the external interconnect [179]. GDDR4/5/5X [58], [59], [180], as well as DDR4 [60] memories, use a pseudo open drain (POD) electrical interface [172]. While the previously used SSTL interfaces terminate to a voltage at $0.5V_{DDQ}$, the POD interface is terminated to V_{DDQ}. In a terminated SSTL interface, DC current is always flowing, transmitting a zero or a one just changes the path of the current flow. In the POD interface, also illustrated in Figure 8.1, DC current through the termination resistors is only flowing when transmitting a zero. Transmitting ones does not cause DC current through the termination. Memory using POD signaling reduces the termination current by employing data bus inversion (DBI) [181]. For every 8 DQ (data) lines, a ninth DBI line is added. Transmitting a zero on this line signals that the 8 DQs lines contain an inverted data byte, while a one on the DBI wire indicates transmission of the non-inverted byte. The simplest DBI scheme is called DBI DC and simply counts the number of zeros in each byte and transmits the byte in its non-inverted form if it contains 4 or fewer zeros. If the byte contains 5 or more zeros, the byte will be inverted. A byte with 5 zeros, will contain 3 zeros after inversion, however, the DBI bit will contain an additional zero indicating

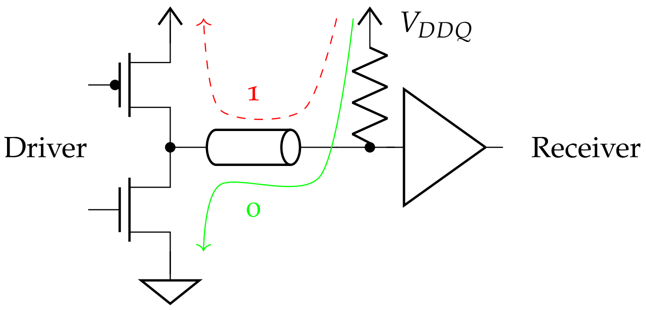


Figure 8.1: Pseudo open drain (POD) interface

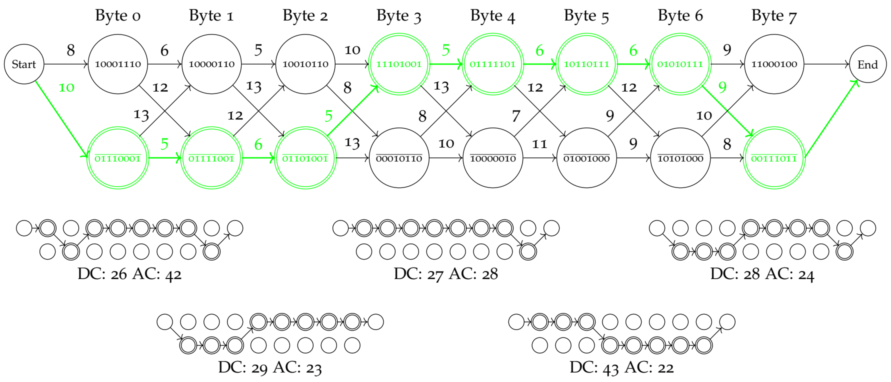


Figure 8.2: Optimal DBI encoding as a shortest path problem

the inversion. This scheme guarantees that never more than 4 zeros per byte are transmitted.

In addition to the interface energy consumed by DC termination current, transitions from zero to one or one to zero consume dynamic power by charging and discharging of load capacities. The importance of the load capacities can also be seen in the design of the POD output driver: A regular open drain output would rely solely on the resistor to V_{DDQ} to generate high output state, but the pseudo open drain output actively drives the output high to provide a faster recharging of the load and thus also a faster signal transitions than what could be achieved by the termination pull-up alone. Instead of reducing the number of transmitted zeros, the DBI signaling can also be used to reduce the number of signal transitions. In the DBI AC scheme, each transmitted byte is inverted, if the inversion reduces the number of signal transitions.

In this chapter, we present a novel DBI encoding scheme. It finds a minimum energy DBI encoding of a burst if given the ratio between the energy for transmitting a zero and the energy per transition. The chapter is organized as follows: Related work has already been presented in Chapter 3, Section 3.5. We start the chapter with introducing our optimal encoding algorithm and a simplified variant. Then in Section 8.2 we explain how power was modeled and explain a hardware design that is able to perform the new DBI encoding at the required data rates. In the following section, we present our experimental results. A Summary and some conclusions related to the chapter can be found in Section 8.4 as well as at the end of the thesis in Chapter 12.

8.1 OPTIMAL ENCODING

To reduce the power consumption, every burst should be transmitted using as little energy as possible. Each burst of 8 bytes can be encoded using 2^8 different DBI patterns. A naive algorithm would search through all possible encoding options and pick the cheapest one. But the cheapest encoding option can be found much more efficiently as we can reformulate the problem as a shortest path problem on a directed graph with non-negative weights. This is illustrated in Figure 8.2. The topology of the graph only depends on burst length. Two nodes exist for each byte, one node represents the transmission of the byte in its non-inverted representation, while the other node represents the inverted transmission of this byte. The cost of transmitting each byte depends only on the previous byte as well as the byte itself. Only two different previous bytes can exist, either the previous byte was inverted or not. The weight of the edges represents the cost of encoding each byte based on the previous byte. The shortest path from the start to the end node is the encoding with the minimum total energy. Three factors control the weights of the edges: The data that should be transmitted and the coefficients α and β. The α coefficient configures the cost of each signal transition, while the β coefficient sets the cost of each transmitted zero bit. As the shortest path does not change by a uniform scaling of the edge weights, we can freely scale the coefficients as long as the ratio $\frac{\alpha}{\beta}$ does not change. This allows us to use small integer coefficients without a significant loss of encoding efficiency. Our top example shows the shortest path and edge weights for $\alpha = \beta = 1$. This choice of α and β in the example implies that the energy cost of transmitting a zero is identical to the energy cost of a transition. If we vary the coefficients without changing the data, we find 5 other Pareto optimal encoding options. The DBI DC algorithm finds an encoding with 26 zeros, but 42 transitions. The DBI AC algorithm finds the encoding with 22 transitions but 43 zeros. But neither of these two previous algorithms are able to identify the three encodings with a more balanced trade-off between zeros and transitions. If we assume $\alpha = \beta = 1$, then the optimal encoding has energy cost of $28 + 24 = 52$, while

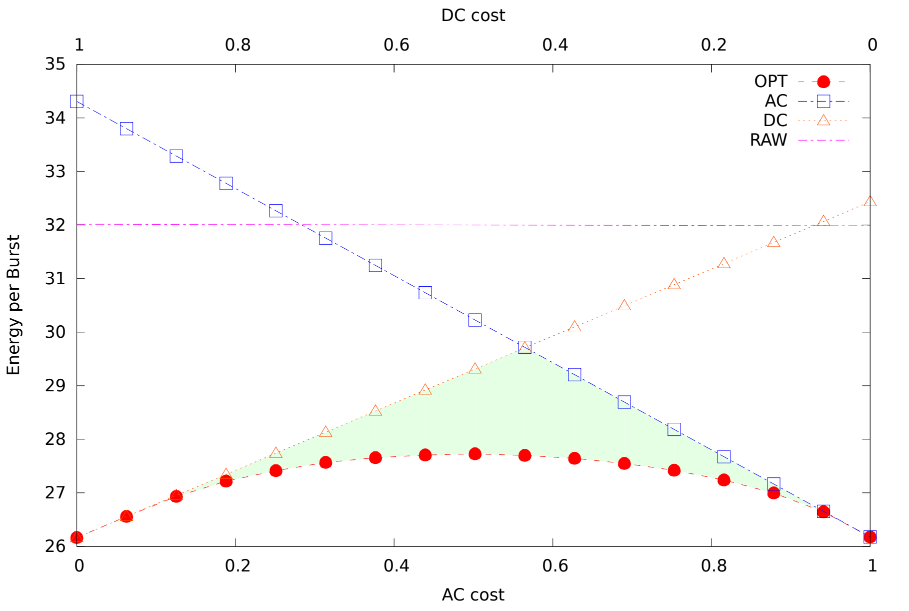


Figure 8.3: Energy per Burst using different DBI schemes

DBI DC choose an encoding with a cost of $26 + 42 = 68$ and DBI AC selects an encoding with a cost of $43 + 22 = 65$.

We simulated the different DBI encoding schemes on 10000 random bursts. We varied the cost α per signal transition from 0 to 1 and set the cost $\beta = 1 - \alpha$. The result is shown in Figure 8.3. DBI DC behaves identically to optimal DBI (DBI OPT) encoding when the AC cost is 0. This is no surprise as DBI OPT with $\alpha = 0$ and $\beta = 1$ is identical to DBI DC. DBI DC works almost as well as the optimum encoding until the AC cost reaches 0.15. Similar results can be seen for DBI AC. As expected DBI AC performs identically to DBI OPT when the DC cost is 0 and the performance stays close until the DC cost reaches 0.15. DBI DC finds the one encoding with the lowest number of zeros, every other encoding will use at least one zero more. At $\alpha = 0.10$, the cost of zeros is much higher than the cost of transitions. In order to gain an advantage over DBI DC, for each additional zero the number of transitions would need to be reduced by more than 9, this is rarely possible. The same concept applies to DBI AC and large values of α.

Both DBI AC and DBI DC perform worse than unencoded (RAW) data, when used together with high DC cost or AC cost, respectively. DBI AC encoding is cheaper than DBI DC encoding starting from $\alpha = 0.56$. The biggest advantage of optimal DBI encoding is also offered at this point, where the average cost per burst is 2 points or 6.75% lower than with DBI AC or DBI

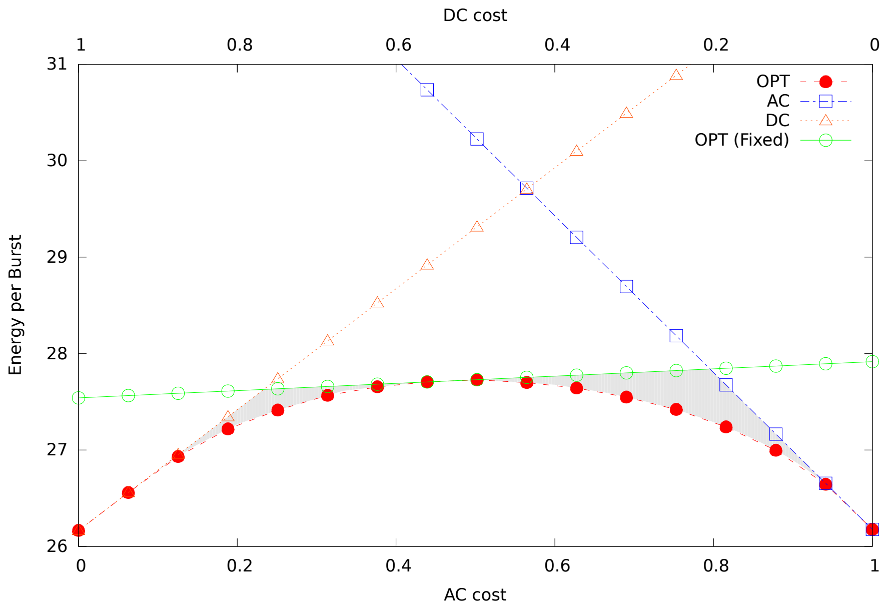


Figure 8.4: Energy per Burst for different DBI schemes, shaded area shows loss of efficiency from fixed coefficients

DC. The shaded area in Figure 8.3 shows the advantage of DBI OPT encoding compared to the best conventional encoding scheme (DBI DC or AC).

One problem with DBI OPT encoding is the accuracy required for the coefficients. However, as we already saw with DBI AC and DBI DC, the coefficients do not need to be very accurate to still enable almost perfect encoding results. We fixed $\alpha = \beta = 1$ and named this encoding scheme DBI OPT (Fixed). Figure 8.4 shows the results. The shaded area indicates the small reduction of performance due to the fixed coefficient. The encoding with fixed coefficients performs better than previous scheme from an AC cost of 0.23 to 0.79. The maximum energy reduction from this encoding is nearly identical at 6.58%.

8.2 EXPERIMENTAL SETUP

8.2.1 *Power Model*

We estimated the energy consumption with a model derived from the CACTI-IO model presented by Jouppi et al. [179], [182]. We unified all load capacities into a single load capacity and reformulated the equations from power to energy per activity.

E_{zero} is the energy consumed by transmitting a single zero.

$$E_{zero} = \frac{V_{DDQ}^2}{R_{pullup} + R_{pulldown}} \frac{1}{f} \tag{8.1}$$

$E_{transition}$ is the energy consumed by a single transition from zero to one or one to zero.

$$E_{transition} = \frac{1}{2} V_{DDQ} V_{swing} c_{load} \tag{8.2}$$

V_{swing} is the signal swing, it is calculated from the output resistance of the pulldown driver ($R_{pulldown}$) and on-die termination resistor. (R_{pullup})

$$V_{swing} = V_{DDQ} \frac{R_{pullup}}{R_{pullup} + R_{pulldown}} \tag{8.3}$$

The total interface energy per burst is calculated as follows:

$$E_{burst} = n_{zeros} E_{zero} + n_{transitions} E_{transition} \tag{8.4}$$

c_{load} is the total load capacity. We tested a wide range of values from 1 pF to 8 pF total load. It should be the sum of the effective capacities of the driver in the CPU or GPU, the capacities of the memory devices added to the DQ lines, the capacity of the transmission line connecting memory and CPU/GPU. If a system uses DIMM or similar sockets the extra load of those should also be considered. Amirkhany et al. state a 1.3 pF load for a GDDR5 output driver [183]. CACTI-IO assumes 2 pF for an DDR4 output driver and 1 pF per memory device [179]. Vuong lists a maximum capacity of 1.3 pF for DDR4 [184]. IBIS files from Micron also list similar values per DDR4 input. DIMM sockets and the PCB trace can add a few additional pF.

8.2.2 *Hardware*

To validate that the proposed DBI encoding can be done at the required data rates and add only a small overhead to a CPU or GPU using this scheme, we developed a hardware implementation. Our proposed hardware architecture is shown in Figure 8.5. Each byte of the burst is processed by one processing block. Each block receives two minimum costs: $cost(i)$ is the minimum cost of transmitting bytes 0 to $i-1$ with the last byte transmitted in non-inverted encoding, while $cost\_inv(i)$ is the minimum cost of transmitting those bytes with the last byte inverted. If we consider the problem as a shortest path problem, $cost(i)$ is the cost of the shortest path from the start to the node of the ith byte and $cost\_inv(i)$ is the cost of the shortest path to the corresponding inverted node. Each of the processing blocks receives the byte itself as well as the exclusive-or of this byte and the previous byte. Within each processing

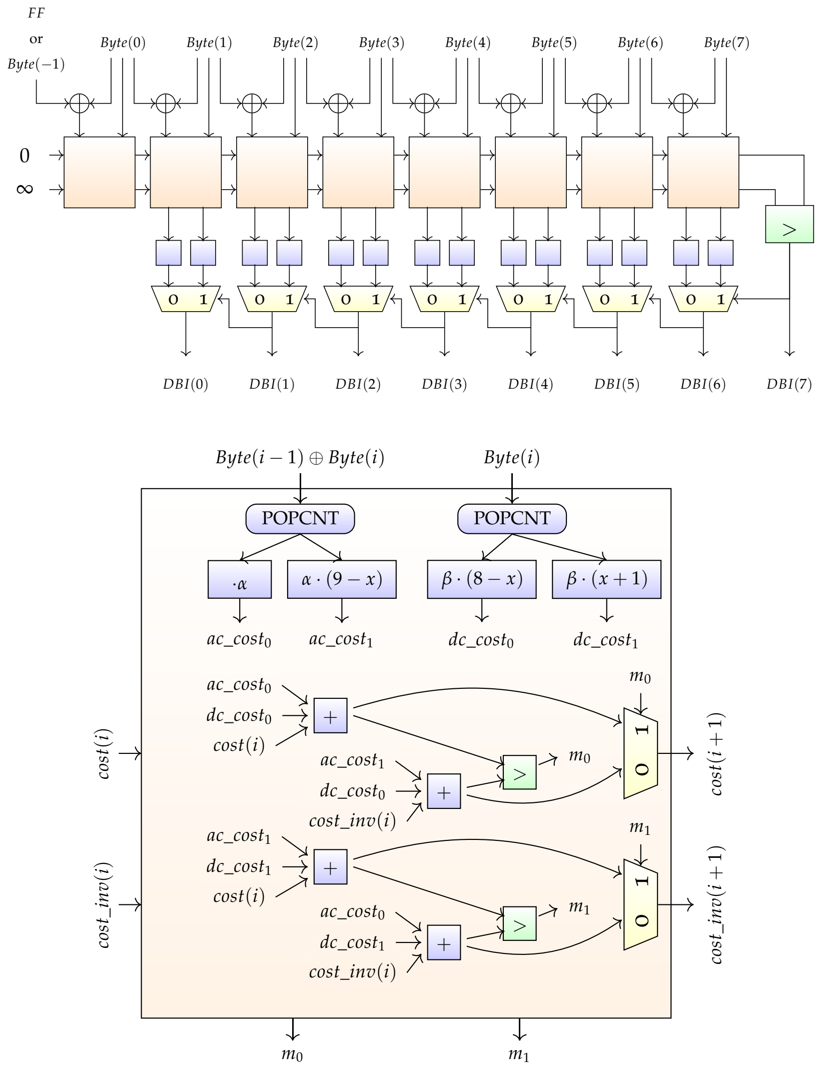


Figure 8.5: Hardware architecture of improved DBI encoder

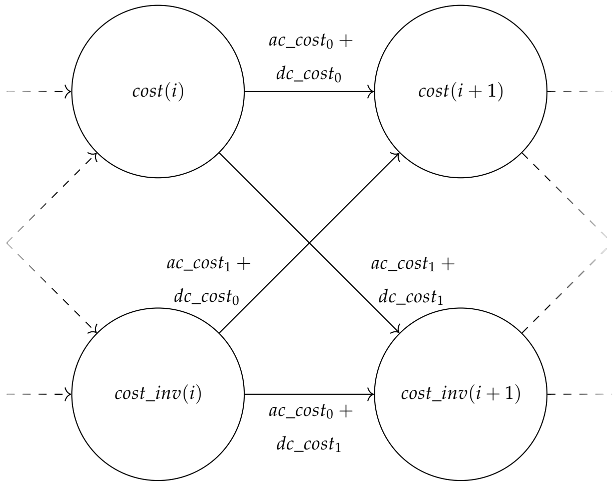


Figure 8.6: Mapping of shortest path search to signals

block, two population count units (POPCNT) count the number of set bits in each of the two inputs on the top. dc_cost_0 is the cost of transmitting the current byte without inversion, i.e., the number of zero bits multiplied with the cost β of each zero. dc_cost_1 is the DC cost of transmitting the current byte inverted. In this case, the extra zero transmitted on the DBI signal also needs to be considered, which results in the $+1$ term. Two options also exist for the number of signal transitions: Either both the previous byte and the current byte are transmitted in the same way (ac_cost_0) or the DBI bit changed between the two bytes (ac_cost_1). Now the cost of four different encoding options can be calculated (from the top to the bottom):

1. Previous byte was not inverted, current byte also not inverted.
2. Previous byte was inverted, current byte is not inverted.
3. Previous byte was not inverted, current byte is inverted.
4. Previous byte is inverted, current byte also inverted.

The relationship to the graph is also shown in Figure 8.6. To calculate the cost of reaching a node via one edge, we need to consider the cost of the edge as well as the minimum cost of reaching the source node of the edge. Two edges lead to each node and we compare their cost and store which of the edges provided the cheapest path. The cheapest path is then forwarded to the next block.

At the last block, we compare which of the two end nodes provides the shortest overall path. This path is backtracked to find the DBI pattern using

Table 8.1: Synthesis Results (32nm)

Scheme	Area (μm^2)	Static Power (μW)	Dynamic Power (μW)	Burst Rate (GHz)	Total (μW)	Energy per Burst (pJ)
DBI DC	275	105	111	1.5	216	0.14
DBI AC	578	170	250	1.5	420	0.28
DBI OPT (Fixed Coeff.)	3807	257	2233	1.5	2490	1.66
DBI OPT (3-Bit Coeff.)	16584	5200	3600	0.5	8800	17.6

the muxes below the blocks. This is the same technique that is also used in the Dijkstra's algorithm to reconstruct the shortest path.

We described our designs in VHDL and synthesized the designs using Synopsys Design Compiler Ultra K-2015.06-SP4 together with the Synopsys 32nm generic libraries in order to estimate the required die area, power and throughput. We synthesized two variants of our proposed design: One design used configurable 3-bit coefficients for α and β, while the other design fixes $\alpha = \beta = 1$. The fixed coefficients remove multipliers from the design and reduce the bit width of the data path. We added 8 pipeline stages to the output of our design and used the retime option of the synthesis tool to move the registers to an appropriate location. Current GDDR5X uses up to 12 Gbps data rate per pin. Our design encodes 8 bytes per clock cycle, thus a clock frequency of 1.5 GHz is required to meet the required throughput using a single encoding unit. Whether this design adds additional latency, depends on the design of the memory controller, often it should be possible to perform the encoding in parallel with other memory controller tasks. If extra latency is added, this can still be acceptable for GPUs: GPUs already have memory subsystems with hundreds of cycles of latency and their performance is relatively insensitive to additional latency [185].

8.3 RESULTS

Table 8.1 shows the results of our synthesis. DBI DC, DBI AC and DBI OPT with fixed coefficients could meet the 1.5 GHz timing, equivalent to a data rate of 12 Gbps. DBI OPT with 3-Bit configurable coefficients was significantly slower and could only run at 500 MHz (equivalent to 4 Gbps). It also required $4.5\times$ more area than the design with fixed coefficients and used $10.6\times$ more energy per encoded burst than the design with fixed coefficients. Due to the

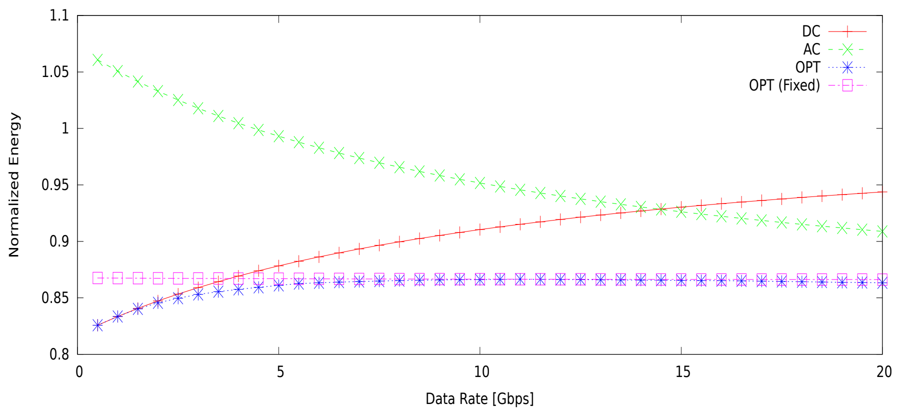


Figure 8.7: Interface energy per burst normalized to unencoded transmission for various DBI encoding schemes

lower frequency, 3 units are required to reach the same throughput, increasing the area requirements even further.

Figure 8.7 displays the interface energy per burst normalized to the cost of transmitting the data without any DBI encoding using POD135 (used by GDDR5X) and 3 pF load. However, results for DDR4 with POD12 are almost identical. DBI DC performs better than DBI OPT (Fixed) until 3.8 Gbps. DBI DC assumes that energy of zeros dominates, while DBI OPT (Fixed) assumes that energy per zero and energy per transition are identical, and until 3.8 Gbps, DBI DC assumptions are closer to the truth. DBI AC assumes transition energy dominates and DBI AC would require a significantly higher frequency than 20 Gbps to perform better than the DBI OPT (Fixed) scheme. The maximum gain from this optimized encoding can be found around 14 Gbps.

The previous Figure 8.7 does not include the energy required for encoding. If we also consider the energy for encoding, the picture changes. DBI OPT encoding with configurable coefficients encodes the data only slightly better than the fixed coefficient version, however, it uses significantly more energy for encoding each burst. For this reason, it always consumes more power than the DBI DC and DBI AC schemes. However, further optimization of the hardware might change this. We used a relatively old 32nm process node for estimating the power consumption and an optimized implementation in a more recent process could provide a significant power reduction, that could make configurable coefficients beneficial.

Figure 8.8 shows the energy per burst for DBI OPT with fixed coefficients normalized to the best conventional DBI encoding. Higher capacitive load reduces the frequency where the highest reduction of energy is achieved. At

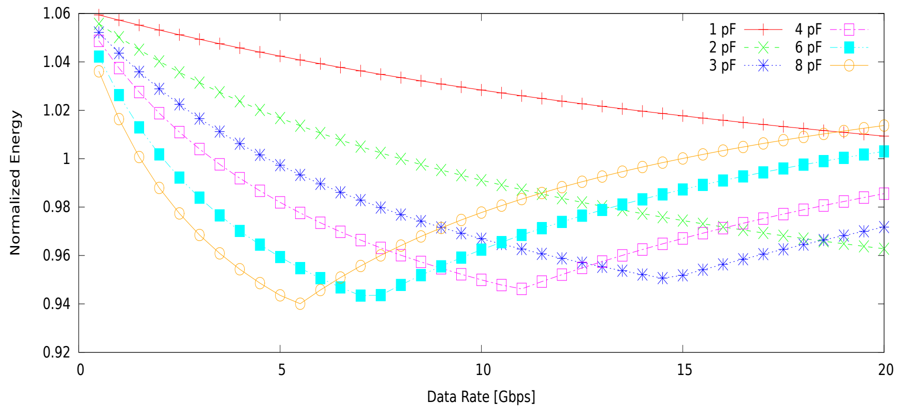


Figure 8.8: Energy per burst using optimized encoding, including encoding energy, normalized to best of DBI DC or AC

3 to 8 pF load, the energy is reduced between 5 – 6% at the operating points with the highest gains.

8.4 SUMMARY

This chapter presented optimal data bus inversion encoding. It showed that link power consumption can be reduced by up to 6% with an improved data bus inversion scheme. Optimal DBI encoding finds the lowest energy encoding for each transmitted burst by reducing both the number of zeros and the number of signal transitions. We have shown that finding this optimal encoding is identical to solving a specific shortest path problem. A hardware implementation of DBI encoding was presented that is able to encode at the required data rates. We have shown that optimal DBI encoding is beneficial for energy consumption, even when considering the energy consumed by the encoding. Optimal DBI encoding does not change the performance, but increases the energy efficiency. In the next chapter, we will present a technique called Sparkk. It uses the approximative computing paradigm to reduce the number of DRAM refresh cycles and thus also the energy consumed by refresh.

9 SAVING DRAM REFRESH POWER

The work presented in this chapter was previously published: J. Lucas, M. Alvarez-Mesa, M. Andersch, *et al.*, "Sparkk: Quality-scalable approximate storage in DRAM," in *The Memory Forum*, 2014.

DRAM memory stores its contents in leaky cells that require periodic refresh to prevent data loss. The refresh operation does not only degrade system performance but also consumes significant amounts of energy in mobile systems. Relaxed DRAM refresh has been proposed as one possible building block of approximate computing. Multiple authors have suggested techniques where programmers can specify which data is critical and cannot tolerate any bit errors and which data can be stored approximately. However, in these approaches, all bits in the approximate area are treated as equally important. We show that this produces suboptimal results and higher energy savings or better quality can be achieved if a more fine-grained approach is used. Our proposal is able to save more refresh power and enables a more effective storage of non-critical data by utilizing a non-uniform refresh of multiple DRAM chips and a permutation of the bits to the DRAM chips. In our proposal bits of high importance are stored in a high quality storage bits and bits of low importance are stored in low quality storage bits. The proposed technique works with commodity DRAMs.

Refresh is projected to consume almost 50% of total DRAM power only a few generations ahead [110]. For mobile devices, DRAM refresh is already a big concern. At the same time, the DRAM capacity of modern smartphones and tablets lags just slightly behind regular PCs. Complex multimedia application are now used on phones and often use large parts of their DRAM usage to store uncompressed audio or picture data. DRAM refresh must be performed even if the CPU is in sleep mode. This makes reducing refresh energy important for battery life. Short refresh periods such as 64 ms are required for error-free storage. Most bit cells can hold their data for many seconds, but to ensure error-free storage all cells are refreshed at a rate sufficient for even the most leaky cells. But not all data requires an error-free storage, some types of data can accept an approximate storage that introduces some errors.

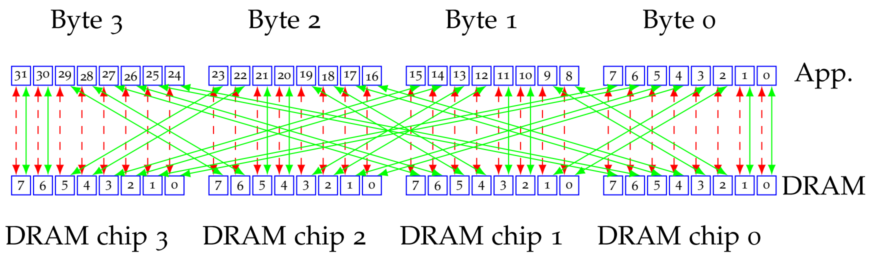


Figure 9.1: Mapping of bits to 4 DRAM chips for approximate byte storage

This provides an opportunity to reduce the refresh rate. Uncompressed media data is a good candidate for approximate storage. However not all media data suitable for approximate storage is equally tolerant to the errors caused by the storage.
In this chapter, we propose a lightweight modification called Sparkk to DRAM-based memory systems that provides the user with an approximate storage area in which accuracy can be traded for reduced power consumption. We show how data and refresh operations can be distributed over this storage area to reach better quality levels than previously proposed techniques with the same energy or the same quality with less energy. Our technique allows an analog-like scaling of energy and quality without requiring any change to the DRAM architecture. This technique could be a key part of an approximate computing system because it enlarges the region of operation where useful quality levels can be achieved.

This chapter is organized as follows: First we describe the concept of approximative storage in Section 9.1. In Section 9.2 we discuss the key ideas behind our proposed Sparkk storage and its integration into the DRAM controller. Prior to the evaluation, our theoretical model of Sparkk is described in Section 9.3. In Section 9.4 we evaluate the properties of Sparkk. Finally, we provide a short summary in Section 9.5. Additional conclusions related to Sparkk can be found in Chapter 12.

9.1 APPROXIMATIVE STORAGE

Reducing the energy consumption is a topic of ever increasing importance. By relaxing the normal correctness constraints, approximate computing opens many possibilities for energy reduction. Many applications can tolerate small errors and still provide a great user experience [187].
This does not only apply to the correctness of calculations but also applies to the storage of data values. Often a bit-exact storage is not required and small deviations do not hurt. The amount of variation applications can

tolerate depends on the application [187]. We, therefore, argue that a practical approximative storage system should be able to provide different quality levels. This allows programmers or tools to find a good trade-off between power consumption and quality. Even within a single application, many different storage areas with vastly different quality requirements can exist. Previous work often used binary classifications such as critical and non-critical data [187][112]. This binary classification, however, limits the usefulness of approximative storage. With only a single quality level for approximate storage, applications cannot choose the best accuracy and power consumption trade-off for each storage area, but are forced to pick a configuration that still provides acceptable quality for the data that is most sensitive to errors. One proposal for approximative storage is Flikker [112]. Flikker reduces the refresh rate for a part of the DRAM to provide a memory area with reduced energy consumption but also errors caused by the reduced refresh rate.

9.2 SPARKK

In this section, first, the key ideas behind the our proposed Sparkk storage are described. Then, it is explained how the DRAM controller manages the refresh of these storage areas.

9.2.1 *Sparkk Storage*

Our proposed extension to Flikker [112] and RAIDR [110] is based on two key observations:

1. Even within a single storage area, not all bits are equally critical.
2. Most memory systems require multiple DRAM chips (or multiple dies in a single package) to reach the required bandwidth and capacity.

Applications mostly use and store multi-bit symbols such as bytes. A bit error in the most significant bit of a byte changes the value of the byte by 128, while a bit error of the least significant bit will only change the value by one. Many applications can tolerate errors that change the least significant bit but will provide an unacceptable quality if the most significant bit fails often.
Regular DDR3 chips have 4, 8 or 16 bit wide interfaces, but most CPUs use wider interfaces. Building a 64-bit wide DRAM interface with regular DDR3 requires at least four 16-bit wide chips or eight 8-bit wide chips. Chip select signals are normally used to control multiple ranks that share the same data lines. The chip select signals for all DRAM chips of a single rank are usually connected together. Only the whole rank can be enabled or disabled. Thus a command is always executed on the whole rank. We propose that the memory controller provides separate CS signals for every chip of the DRAM.

This way commands can be issued to a subset of the DRAM chips of a rank. While many additional uses are possible [113], Sparkk uses this to gain finer grained control over the refresh. With this modification to the traditional interface, different DRAM chips of a rank do not need to share the same refresh profile, as refresh commands can be issued selectively to DRAM chips. Using manual refresh does not work together with self-refresh, as during self-refresh mode the interface to the memory controller is disabled and self-refresh is performed autonomously by the DRAM chip. Requiring one CS line per subrank can also be problematic in systems with many subranks. These problems can be solved by making changes to the DRAM refresh signaling. This, however, requires modifications to the DRAM chips. The exact details of such a scheme remain future work.

Different refresh periods for the same row of different DRAM chips of one rank can make multiple storage bits with different quality levels available simultaneously. But without additional modifications to the usual memory system, this does not solve the problem: Some high quality storage bits would be used to store low-importance bits and vice versa. It is, therefore, necessary to permute the data bits so that high-importance bits are stored in high-quality storage bits and low quality storage bits are only used for bits of low importance.
Figure 9.1 shows how four bytes can be mapped to a 32-bit wide memory interface built from four DRAM chips. The red dashed lines show a commonly used mapping between application bits and DRAM bits. The green lines show the proposed mapping: The two highest significance bits from all four bytes transmitted per transfer are mapped into DRAM chip 3. The two smallest significance bits are all mapped into DRAM chip 0. Bit errors in DRAM chip 3 now have a much higher impact on the stored values. They will change the stored values by 64, 128 or 192, while bit errors in DRAM chip 0 will only cause small errors in range of 1-3. Data types with more than 8 bits per value have even larger differences in significance between the bits. To find a good trade-off between the number of refresh operations and quality, more refresh cycles need to be allocated for the refresh of DRAM chip 3 than for DRAM chip 0. Which bits are important varies between data types; here, an example is shown for bytes. We propose to add a permutation stage to the memory controller that provides a few permutations selected for the most common data types. The used permutation could be controlled by the type of instruction used for access, additional page table flags or a few extra high address bits of the physical address. Permutations are not limited to blocks equal to the width of the memory interface: It is also possible to permute the bits within one burst or one cache line. With a 32-bit wide DDR3 interface one burst is 256-bit long, so it may be used to map four 64-bit values to DRAM chips in a way that minimizes errors.

9.2.2 *Controlling the refresh*

Aside from the permutation, an effective way for the memory controller to decide whether a row needs to be refreshed is also required. Unfortunately, the solution used by RAIDR cannot be adapted for this task. In RAIDR, the memory controller uses bloom filters to decide which rows need to be refreshed more often. These bloom filters are initialized with the set of rows that contain high leakage cells. A bloom filter can produce false positives, but refreshing some rows more often than needed does not hurt but only causes additional power usage. For schemes such as Sparkk or Flikker, a bloom filter cannot be used: Areas falsely binned as approximate would cause data corruption in critical memory locations and a bloom filter containing all non-approximate rows would use large amounts of memory.
A different solution is required. The authors of Flikker proposed a single boundary in memory space between critical and non-critical data. While this solution is simple, it does not offer enough flexibility as it does not allow for multiple approximate storage areas with different quality levels at the same time. It should be possible to run multiple applications each using multiple approximative storage areas at different quality levels. A practical approximate storage system should thus offer the ability to configure multiple memory areas with different quality levels. This way, a trade-off between power and accuracy can be chosen for each area. Using our proposal, it is also possible to turn off refresh completely for unused bits, e.g. if 16-bit data types are used to store 12-bit data. It is also possible to build an approximate storage that guarantees that errors directly caused by the memory stay within configured bounds: More significant bits can be configured to be refreshed at the rate required for error free operation. Only less significant bits are then stored approximately.

Fortunately, we can exploit the characteristics of the refresh counter to build a flexible mechanism that enables the handling of hundreds or thousands of memory areas with different refresh characteristics with a very simple hardware unit and low storage requirements. Every time the refresh counter switches to the next row, the memory controller must decide if a refresh operation should be triggered in the memory and, if so, on which subranks. The refresh counter in the memory controller counts through the rows in a monotonic and gap-less fashion. A mechanism that provides retrieval of the refresh characteristics of arbitrary addresses is therefore not required. At every point in time, it is only necessary to have access to the refresh properties of the current block and information about the end of the current block. We propose to store the refresh properties in an ordered list. Each entry contains the address of the end of the current area and information about the refresh properties of this block. When the refresh counter address and the end of the block address match, we advance to the next entry in the table. Figure 9.2 shows the proposed hardware unit. The unit uses a small single ported memory to store the list of memory areas. The exact storage requirements depend on the number of DRAM chips per rank, the maximum number of DRAM rows and the maximum refresh period at which a DRAM chip still provides at least some working storage bits. Normally less than 100 bits are required per memory area. Because of the list structure, each storage area can be composed of any number of consecutive rows. For each area, the refresh period of each DRAM chip can be freely configured to any multiple of the base refresh rate.

The refresh settings table stores phase and period information for each DRAM chip. Every time a new area is accessed, the controller determines which memory chips should be refreshed within this area: The phase counters for each memory chip are decremented and a counter reads zero, the chip is refreshed and the phase is reset to the value of the period information. A special period value can be used to indicate to omit refresh completely, this be can be another useful feature for some applications: Reading or writing data from a row also causes a refresh of the data. A GPU that renders a new image into a framebuffer every 16.6 ms does not need to refresh the framebuffer. The access pattern guarantees refresh, even without explicit refresh cycles.

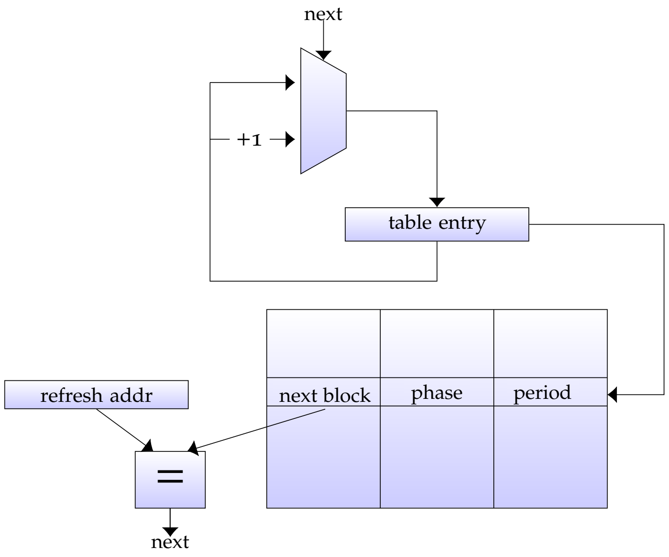


Figure 9.2: Refresh settings table

Different software interfaces to such a storage system are possible: It can be part of a system that is designed from top to bottom for approximate computing and uses specialized languages, specialized instruction sets and specialized hardware such as proposed by Esmaeilzade et al. [188]. In such a system specialized load and stored instructions could select different permutations for each access based on the data type. A compiler aware of these permutation rules could create data structures containing a mix of data types. Even with such a specialized instruction set, refresh settings can only be selected on a per-page granularity. Applications that organize data in arrays of structures thus often need to make problematic trade-offs if the structure members have different accuracy requirements. If applications operate on structures of arrays as it is popular with GPU or DSP applications or data such as pictures or audio, the interface to software can be much simpler: Regular languages such as C, C++ or OpenCL can be used and approximate memory areas may simply be allocated using a specialized malloc. The application must provide information about size, data type and quality requirements to the specialized malloc. The operating system then calculates refresh settings for meeting the quality requirements, reserves memory and inserts the appropriate entries into the refresh list. If enough space is not available in the refresh list to allocate a new block with a new refresh list entry, the operating system can always choose to provide better than requested storage quality. Adjacent list entries can be merged by choosing the maximum refresh rate from each of the two merged blocks. One important limitation of the memory management of approximate storage blocks should be noted: Approximately stored memory blocks should not be moved from one approximate location to another, as this will cause an accumulation of errors. In two different blocks, different bit cells will fail at a selected refresh rate and bit errors already added to the data will not disappear by moving these bits into new bit cells. In some cases, it might be required to reserve error headroom to allow for data movement. Another possibility to allow data reallocation is to ask the application to restore data to an error-free state by, for example, recalculating the data or reloading it from a compressed file.

9.3 MODELING OF SPARKK

We model the expected number of non-functional DRAM cells at a given refresh period using data for a 50 nm DRAM provided by Kim and Lee [111]. Our model assumes that non-functional cells will statically stick to either zero or one on readout. We model cells as equally likely to stick to zero or one. If we also assume that zeros and ones are equally common in the stored data, then half of the non-functional cells will still return correct results.

$$P_{bytechange}(k) = \prod_{i=0}^{7} \begin{cases} P_{bitflip}(i) & \text{if } bit_i \text{ in } k = 1 \\ 1 - P_{bitflip}(i) & \text{if } bit_i \text{ in } k = 0 \end{cases} \tag{9.1}$$

As shown in Equation 9.1, the peak signal to noise ratio(PSNR) can be estimated by first calculating the probabilities of the 255 possible changes to a byte, based on the probabilities of changes to the individual bits. $P_{bitflip}(i)$ is the probability of a bitflip in bit i. As already mentioned, this is estimated using the data from Kim and Lee [111]. $P_{bytechange}(k)$ is the probability of the change of a byte by mask k, e.g. $P_{bytechange}(129)$ is the probability of a byte with a bitflip in bit 7 and bit 0 and all other bits without storage errors.

$$MSE = \sum_{k=1}^{255} k^2 P_{bytechange}(k) \tag{9.2}$$

These probabilities are then weighted by their square error to calculate the mean square error (MSE) as presented in Equation 9.2. This is slightly simplified and assumes that additional bit flips always increase the error. This is true, if a single bit per byte flips, but not necessarily true if multiple bits in a single byte flip., e.g.: If 128 should be stored and bit 7 flips, the absolute error is 128, but if bit 5 flips as well, the error is reduced to 96. On the other hand, if 0 should be stored and bit 7 flips the error is 128 and if bit 5 flips as well, the error increases to 160. We found that this effect is negligible in practice. This simplification makes it possible to pick good refresh settings for a given quality without knowledge of the data statistics.

$$PSNR = 10 \log_{10} \left(\frac{255^2}{MSE} \right) \tag{9.3}$$

From the mean square error, the PSNR can be calculated, using the well known equation 9.3. To compare Sparkk with Flikker, the harmonic means of the per chip/subrank refresh rates are calculated. Refresh schemes with an identical harmonic mean trigger the same number of refresh operations per chip/subrank.

Before we can estimate the benefits from Sparkk, it is necessary to find suitable refresh settings that maximize quality at a given energy. With Sparkk, the rows of one approximate storage area within every DRAM subrank can be refreshed at multiples of the base refresh rate of 64 ms. Thus a suitable set of multiples of the base refresh rate must be found. We used hill climbing to find our refresh settings: Starting from the base refresh period, the refresh periods of single DRAM chips is gradually increased until a solution is found that meets the average refresh requirements. At each step of the process and for each DRAM subrank, we calculate how the error rate and average

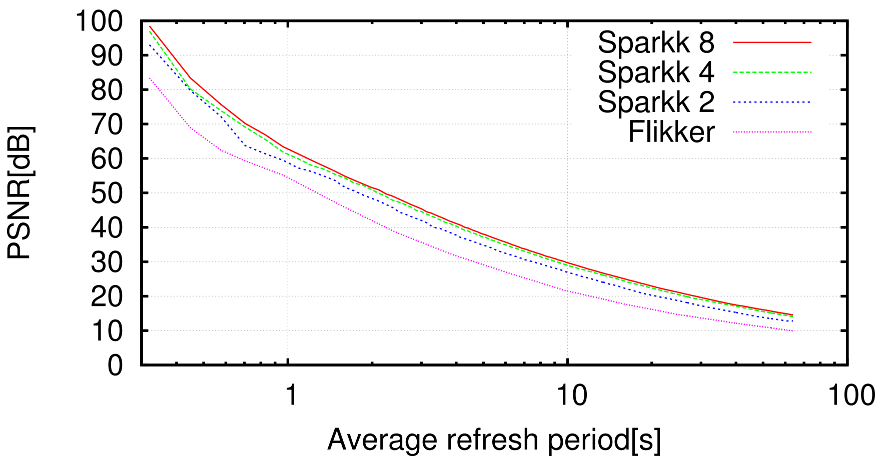


Figure 9.3: PSNR for Flikker and Sparkk with 2, 4, 8 DRAM chips

refresh energy would change, when the refresh period would increase by 64 *ms*. We select the DRAM subrank with the best ratio of newly introduced error and energy saved by the prolonged refresh and increase its refresh rate by 64 *ms*. We continue this process until our average refresh rate requirements are satisfied.

To enable a visual evaluation of Sparkk, we simulated the effect of the approximate storage on image data. Our model was used to estimate how many bits of each significance would flip at a given average refresh period. Then, this number of bits of each type was randomly selected and flipped. Kim and Lee assumed that the retentions times are randomly distributed over the cells [111]. Rahmati et al. verified this assumption experimentally and did not find any specific patterns in the retention times of the measured DRAM [189]. There might be patterns regarding which bits tend to stick to zero or one if their retention time is not met. We assume that there are no such patterns and that both stuck-at cases have the same likelihood. If a real DRAM shows patterns, an address based scrambler could be used to prevent visible patterns caused by the internal DRAM array architecture.

9.4 EVALUATION

Figure 9.3 shows the expected PSNR as predicted by our stochastic model for Sparkk and Flikker. This model predicts the mean PSNR over an infinite set of samples. The PSNR in single samples can be better or worse, depending on the exact distribution of weak bitcells within the used DRAM rows and stored data, but large derivations from the expected PSNR are unlikely. At all tested

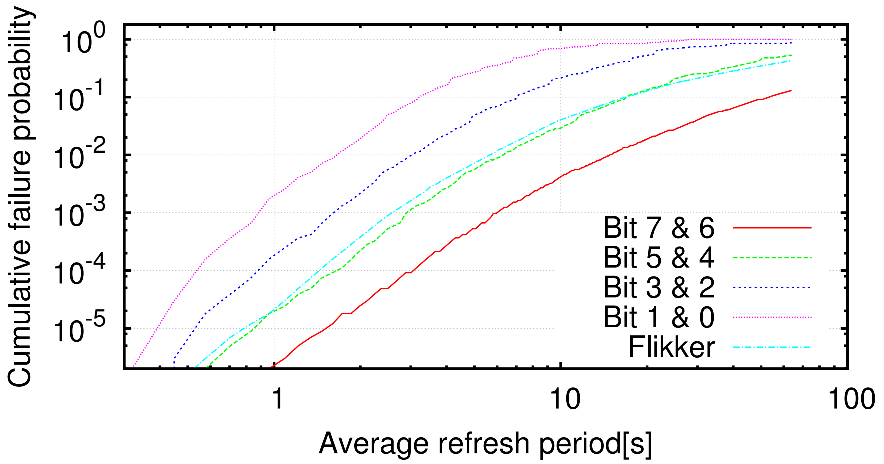


Figure 9.4: Effect of variable refresh on failure probability for individual bits for Sparkk with 4 DRAM chips

refresh rates, Sparkk performs better than Flikker. Even Sparkk with just two subranks provides benefits over Flikker. Sparkk with 4 subranks provides almost all the benefits of Sparkk with 8 subranks.

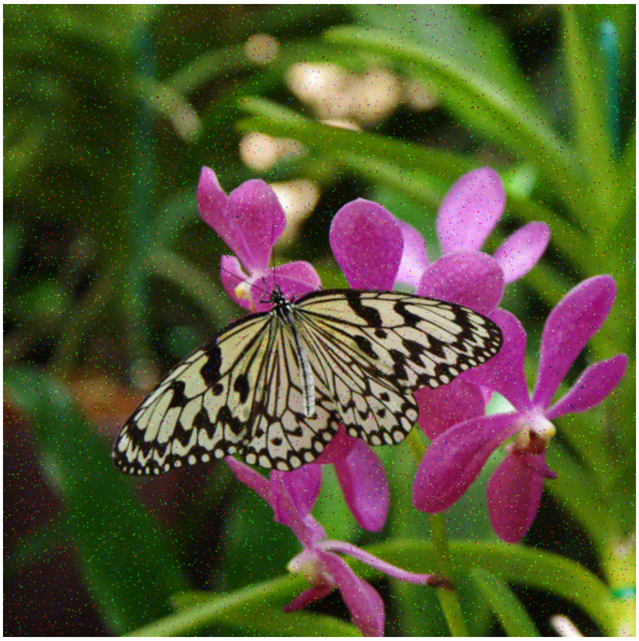
Figure 9.4 displays how Flikker and Sparkk distribute the errors over the bits on memory interface with 4 subranks. In Flikker, all bits have the same error rate. In Sparkk, the subrank storing Bits $0-3$ is refreshed at a lower rate than in Flikker, the subrank with bit 4 & 5 is refreshed at approximately the same rate as in Flikker and the subrank with the two most important bits is refreshed at a higher rate than in Flikker.

To test if the PSNR also provides a good estimate of subjective quality, we generated pictures simulating the effects of Sparkk and Flikker. These pictures can be seen in Figure 9.5. For Sparkk a configuration with 8 subranks was used. The average refresh rate in both cases was 8 seconds. The image saved in the Flikker storage shows many easy-to-spot bit errors of the most significant bit. The Sparkk storage shows only a few of those errors and despite the extremely long refresh period, the image still seems to have an acceptable subjective quality for some applications such as texturing. The background of the Sparkk stored picture looks grainy which is the result of the high number of bit errors in the less important bits.

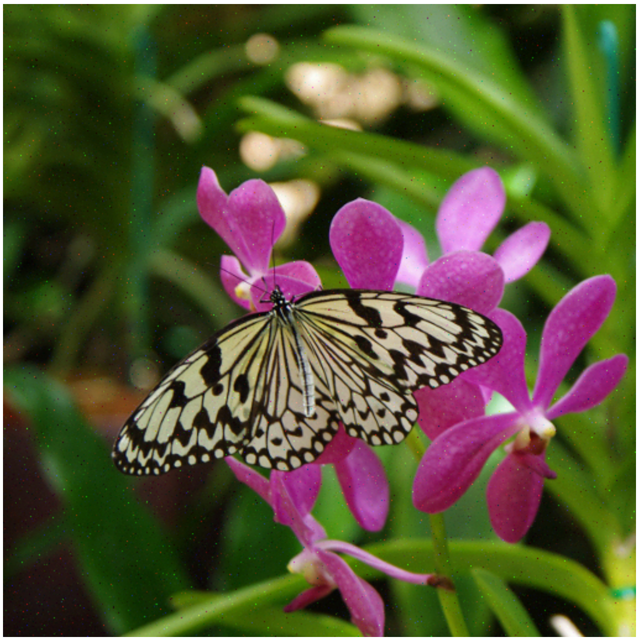
Sparkk is able to reduce the number of refresh operations on the DRAM arrays at a given quality level. This reduces the power required for the chip internal refresh operation. However, Sparkk requires a more complex refresh signaling and the additional energy consumed by this could potentially mitigate the energy advantage in some use cases. While we proposed one refresh signaling scheme that can be used using unmodified DRAMs, with modifications to the DRAM many other schemes are possible and likely more efficient. It remains an open research question how much energy could be saved exactly.

9.5 SUMMARY

In this chapter we presented Sparkk, a modification to conventional DRAM based storage. It uses the concept of approximative storage to reduce the refresh rate, while still enabling high storage quality. It allows a flexible configuration of the storage quality. Compared to previous work, we were able to increase the quality in terms of signal to noise ratio by 10 dB or reduce the required refresh rate by 50%. In this and the previous chapter we looked at two architectural enhancements for improving the energy efficiency of the memory subsystem, Optimal AC/DC DBI encoding and Sparkk, and in the next two chapters we will look at architectural enhancements of the GPU core to enhance both performance and energy efficiency.



(a) Flikker



(b) Sparkk 8

Figure 9.5: Pictures stored in DRAM with 8 seconds average refresh

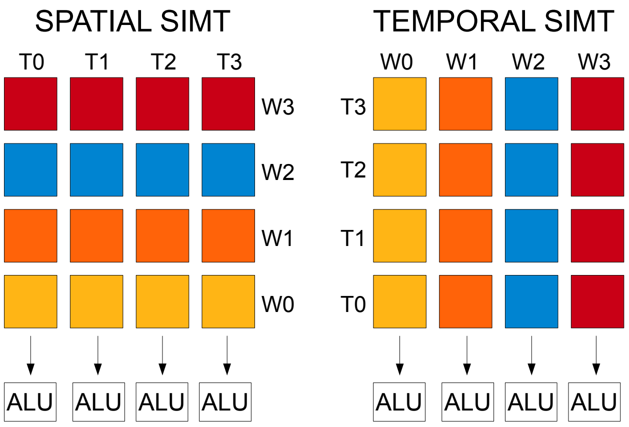
10 SPATIO-TEMPORAL SIMT

The work presented in this chapter was previously published: J. Lucas, M. Andersch, M. Alvarez-Mesa, *et al.*, "Spatiotemporal SIMT and Scalarization for improving GPU efficiency," *ACM Transactions on Architecture and Code Optimization*, Sep. 2015. DOI: 10.1145/2811402.

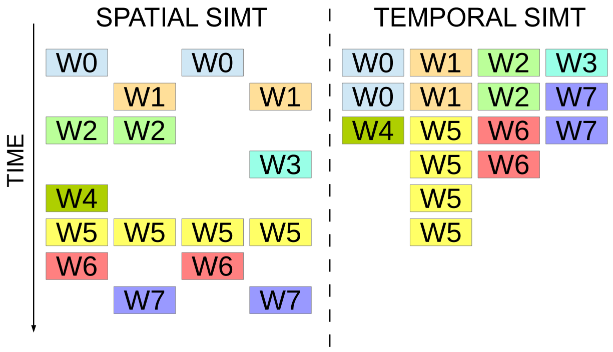
One of the main design decisions in GPUs has been the ganging of execution threads into batches named *warps* and in particular the sequencing of these warps onto an array of execution units in a single-instruction multiple-data (SIMD) fashion. This way of performing SIMD execution is supported by a hardware stack to manage divergent thread control flow, and the resulting execution paradigm has become widely known as single-instruction multiple-thread (SIMT) execution. While SIMT amortizes control hardware over many execution units, research over the past years has shown that this approach yields poor SIMD utilization under control divergence, i.e. when some or most of the threads in a warp are inactive, thus not performing any useful work [121].

In this chapter, we evaluate an alternative approach to the classical *spatial* SIMT. It has been proposed to base GPU cores on *temporal* SIMT (TSIMT), which is a different way of mapping individual threads to execution units [120]. Figure 10.1a compares the spatial and temporal approaches intuitively. In the figure it is shown how four warps (W0 - W3) consisting of 4 threads each are executed, once for spatial and once for temporal SIMT. On the left-hand side, spatial SIMT operates in a classical SIMD fashion: All threads in a warp execute the same instruction and thus the instruction word is broadcast to all execution units every clock cycle. On the right-hand side, temporal SIMT is shown as a *transpose* of the spatial SIMT mapping of warps and threads to datapath units. Here, each warp is executed on only one specific lane, and the threads corresponding to the warp are sequenced onto the scalar execution unit in that lane cycle by cycle. As such, TSIMT is reminiscent of a single lane vector processor [129].

Beside divergent branches another situation where SIMT architectures waste execution cycles is when all threads in a warp not only execute the same instruction, but do so on identical data as well. In this case, it would be



(a) Mapping of warps and threads to execution units in both spatial and temporal SIMT.



(b) Comparison of conventional and TSIMT execution for control-divergent code.

Figure 10.1: Conventional SIMT vs. TSIMT. Different colors are used to symbolize different warps on the core. In this comparison, warps are assumed to be 4-wide and cores are assumed to have 4 execution units / 4 TSIMT lanes.

more efficient to execute the instruction only once instead of once per thread, thereby freeing execution resources. By doing so the SIMD instruction is turned into a scalar instruction, and therefore this technique is known as *Scalarization*. Besides releasing execution resources, Scalarization also reduces the pressure on the register file. This requires, however, that scalar values are stored in such a way that all threads of a warp can access the value. In a conventional GPU this requires additional broadcast networks and often specialized dedicated scalar register files as well. In TSIMT GPUs, on the other hand, tiny modifications to the existing register file suffice and the register file space can be used for both scalar and vector values. We explore Scalarization on TSIMT GPUs in further detail in Chapter 11.

While the basic idea of TSIMT has been sketched in a patent [119] and has been briefly described as an additional idea in two papers [120], [129], a microarchitectural implementation and detailed performance analysis have not been presented before. To fill this gap, we introduce a GPU design based on TSIMT and perform a detailed analysis of its advantages and shortcomings.

Although TSIMT can improve the performance of control divergent workloads, it can, as we will show in the analysis of simulation results, suffer from load balancing issues. As a way to have both the control divergence mitigation of TSIMT and improve the load balancing, we also propose and evaluate an architecture that combines the traditional spatial SIMT with TSIMT, and we refer to it as spatio-temporal SIMT (STSIMT).

More concretely, this chapter makes the following contributions:

- We present a detailed microarchitecture design and implementation of temporal SIMT for GPGPUs.

- We evaluate the TSIMT approach in detail and show that it is able to provide large performance benefits for control divergent GPU codes in μ-benchmarks, but suffers from significant *load balancing* problems in real applications.

- We propose two optimizations to the basic TSIMT microarchitecture that reduce the load balancing problem.

- We introduce and evaluate an STSIMT architecture that combines spatial and temporal SIMT and demonstrate that it exhibits performance improvements compared to both spatial and temporal SIMT.

- We evaluate power, energy consumption and energy efficiency of the architectures presented.

In Chapter 11 we make additional contributions, related to (S)TSIMT combined with Scalarization.

This chapter is organized as follows. Section 10.1 introduces and describes our TSIMT-based GPU design. Performance evaluation results are discussed in Section 10.2. A short summary is provided in Section 10.3. Conclusions are drawn as part of Chapter 12.

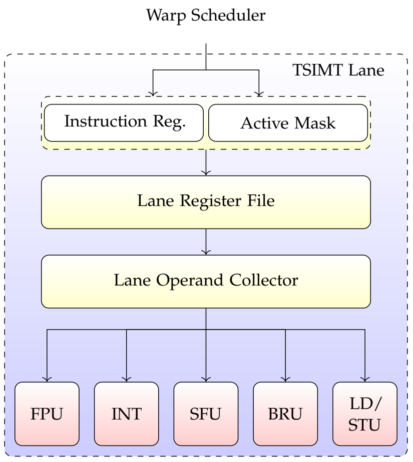
10.1 A TEMPORAL SIMT GPU ARCHITECTURE

These sections describe the proposed TSIMT μ-architectures. As TSIMT-based GPUs are still GPUs, most parts of the μ-architecture are identical to conventional GPUs. A overview of conventional GPU micro-architecture was provided in Chapter 2. Everything outside the GPU cores is unchanged: interconnect, memory controllers, PCIe interfaces and CTA scheduler. Many structures inside the GPU core are also unchanged: instruction fetch and decode, caches, scoreboards and the reconvergence stack. In the following sections, we will thus concentrate on the elements that are modified in TSIMT GPUs compared to conventional GPUs.

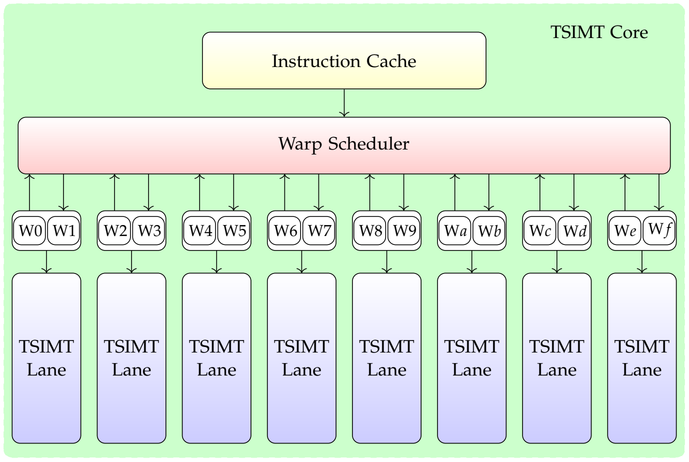
10.1.1 *TSIMT Cores and Lanes*

The basic building block of TSIMT GPU cores is the *TSIMT lane*. A block diagram is depicted in Figure 10.2a. Such a lane consists of four key components: An instruction register able to store a single warp instruction, a slice of the core's register file, an operand collector, private to the lane, and one-thread-wide execution resources for integer, floating point, and memory instructions. Operationally, a TSIMT lane receives a warp instruction along with the corresponding thread active mask from the core's warp scheduler and stores them into dedicated instruction and mask registers. The instruction register is used to hold the instruction in place while the lane back-end sequences through the warp's threads, thereby decoupling the lane from the scheduler while the lane is executing. Together with additional routing logic, a similar instruction register was employed by Braak et al. to build reconfigurable processing pipelines using the GPU execution units [191], [192].

Figure 10.2b indicates how TSIMT lanes are used to construct cores with arbitrary throughput. In a TSIMT core, multiple lanes operate independently and in parallel processing the instruction words stored in their instruction registers. The overall number of threads and warps held in the core are evenly divided over all TSIMT lanes, e.g. in a core with 8 lanes holding 64 warps at most, every lane will statically hold 8 of the warps. Warps cannot switch between lanes as the thread context associated with a warp is stored in the register file within the warp's lane. This subdivides the core's warp pool into separate pools for every lane. In the core's front-end, an instruction fetch unit accesses the instruction cache and fetches as well as decodes instructions into



(a) TSIMT lane



(b) TSIMT core

Figure 10.2: Block diagrams visualizing the organization of a TSIMT lane and the construction of TSIMT GPU cores from TSIMT lanes. In the figure, lanes are one-wide, and cores are constructed from 8 lanes and have an overall occupancy of 16 warps at maximum.

an instruction buffer (IB). The instruction buffer uses dedicated slots for each warp. A single warp scheduler (WS) for the whole core utilizes a scoreboard to monitor which instructions in the IBs have their dependencies fulfilled and are ready to issue. The WS also monitors when TSIMT lanes complete the execution of an instruction and, therefore, require a new instruction word to be sent to the lane's instruction register.

The execution resources are warp-wide (i.e. 32-thread-wide) in conventional spatial SIMT. In a TSIMT execution architecture, on the other hand, each TSIMT lane is one-wide, meaning that one thread instruction can be executed every cycle. There is, however, an entire spectrum of GPU architectures possible in between spatial and temporal SIMT, that combine both execution paradigms. Such *spatio-temporal SIMT* architectures operate like TSIMT architectures, but each lane contains sufficient execution resources to execute instructions of multiple threads in a single clock cycle. For example, with a warp size of 32, one can construct a 4-way-spatial 8-way-temporal SIMT architecture where each TSIMT lane contains 4-wide execution resources. In this microarchitecture, a TSIMT lane sequences the 32-wide warp onto its 4-wide execution resources in 8 consecutive clock cycles. The performance trade-offs of STSIMT as an evolution of basic TSIMT are discussed in Section 10.2.7.

STSIMT is not completely new, single lane implementations without compaction have been used in existing GPU architectures such as NVIDIA Tesla [193]. Tesla uses STSIMT to construct SIMT cores with lower throughput while maintaining a large warp size. For example, in Tesla, 32-wide warps are sequenced onto an 8-wide SIMD datapath over 4 clock cycles. One key difference to STSIMT as proposed here is that existing approaches do not implement compaction and are therefore unable to provide any benefit for the execution of divergent applications. As in both TSIMT and STSIMT lanes are usually busy for more than a single cycle, multiple lanes can share a single frontend for instruction issue and decode. Tesla, however, does not exploit this property of STSIMT. We use a baseline architecture similar to Tesla for the experimental evaluation in Section 10.2.

10.1.2 *Control Divergence*

The TSIMT concept can efficiently provide large performance benefits when executing control-divergent codes. Consider Figure 10.1b for example. It shows how a conventional GPU core with 4 execution units and a TSIMT-based core with 4 lanes execute instructions from 8 different warps, where each warp is coded with a different color. For the sake of conciseness, warps are assumed to consist of 4 threads each. The warp instructions executed are control divergent, i.e., some threads do not participate in the execution and are inactive. For explanatory purposes, each warp instruction is shown with a different active mask, although such situations are rare.

The figure shows that the conventional GPU architecture always requires one clock cycle on all 4 ALUs to execute a warp instruction, regardless of the instruction's active mask. Threads that are switched off and the respective ALUs are left unused for the clock cycle. In this example, the conventional GPU completes 15 thread instructions in 8 clock cycles for an overall IPC of $16/8 = 2.0$. On the TSIMT-based core, on the other hand, *compaction* is performed: Warp instructions with some inactive threads are executed in fewer cycles than the warp width. In fact they are executed in the minimum possible number of clock cycles, e.g. a warp with 2 active threads utilizes one ALU for exactly 2 clock cycles. As such, no execution resources are wasted. Overall, in this hypothetical example, the TSIMT core completes the 15 thread instructions in 6 clock cycles, corresponding an overall IPC of $16/6 = 2.67$. We remark that the IPC in the TSIMT case would approach the ideal IPC of 4 if more work was available, as TSIMT lanes 0 (on the left) and 3 (on the right) are ready to receive new instructions after 3 clock cycles.

In spatio-temporal SIMT architectures, some compaction ability of TSIMT is lost. As an example, consider an STSIMT architecture where each TSIMT lane contains 4-wide execution resources. In this case, compaction only works for warp active masks with aligned bundles of 4 consecutive inactive threads (e.g. 111100001111...). Warp instructions with active masks that switch more often between active and inactive threads (e.g. 101010...), irregular (e.g. 100111010...) or unaligned (e.g. 1000011110000...) cannot be compacted. Therefore, STSIMT architectures lose compaction ability compared to TSIMT architectures but exhibit larger latency hiding ability within each lane as the core's warp pool is partitioned over fewer lanes if the overall execution width of the core remains constant.

10.1.3 *Instruction Issue*

In essence, TSIMT *decouples instruction issue bandwidth from instruction execution bandwidth*. In conventional GPUs, the instruction issue bandwidth is coupled with the execution bandwidth. If a SIMT GPU needs four cycles to execute a warp, it will only need to supply one new instruction every four cycles. If some improvement made it possible to execute instructions with a smaller number of active threads faster in the execution units, speed would not improve, because instructions could not be issued faster. In this chapter, we assume a front end that is able to issue one instruction per clock cycle. This provides a 4 times higher instruction issue bandwidth than needed for perfectly convergent code. The SIMT GPU can use this additional issue bandwidth to execute instructions on SPs, SFUs and LDST units concurrently to exploit ILP. The total number of issued instructions is always the same in SIMT, TSIMT and STSIMT, only the peak issue rate increases in TSIMT. Each warp instruction needs to be issued only once, no matter how many threads are active. Executing the operation over multiple cycles on all active threads of the warp does not require additional issue cycles but is performed locally in the TSIMT lane. In conventional SIMT threads from each warp are executed in lock-step and explicit synchronization can be omitted when data is exchanged between threads of the same warp. Contrary to some other techniques for improving the performance of divergent applications such as thread block compaction [122], where threads can get reassigned to a different hardware warp and programmers cannot expect that threads from same initial warp keep executing in lockstep, in TSIMT lockstep execution of threads of the same warp is preserved and no modifications are necessary to kernels.

10.1.4 *Memory Access Coalescing*

With respect to *memory coalescing*, a TSIMT architecture is largely unchanged from a conventional GPU architecture. When a lane executes a warp-wide memory instruction, it generates one thread memory address per clock until all memory addresses requested by the warp instruction are known. Then, the entire bundle is passed onto the coalescing hardware in the load-store unit that reduces the requests to the minimum number of memory transactions. Finally, the transactions access the L1 cache and, potentially, the lower levels of the memory hierarchy. If the L1 cache or the load-store unit are stalled, then stall signals are propagated back to the warp scheduler which will, therefore, be unable to issue memory instructions until the stall is resolved. Using instruction replay the coalescing hardware can be simplified by reusing major parts of the core [194], which is also currently used in NVIDIA GPUs [195]. Our simulator is based on GPGPU-Sim which, however, currently does not model instruction replay. For this reason, we decided to model all architectures without instruction replay.

10.1.5 *Shared Memory*

While global memory request coalescing in TSIMT and SIMT is similar, shared memory instructions are handled differently by TSIMT GPUs. In a conventional GPU, threads within a warp must access different memory banks to prevent serialization due to bank conflicts. In TSIMT, on the other hand, threads within the same warp never produce bank conflicts as they are executed in consecutive cycles. Instead, warps on different lanes that try to access shared memory simultaneously may produce inter-warp bank-conflicts. Despite these differences the hardware needed for the shared memory is almost identical for TSIMT and SIMT. Each lane sends up to one address to the shared memory, the addresses are checked for conflicts and a crossbar connects several SRAM banks to the input and output ports of the lanes. In TSIMT some ports of the shared memory are often not used because the lane connected to that port is currently executing an arithmetic instruction, this can reduce the number of shared memory bank conflicts that occur. The lockstep execution of the warps in SIMT, on the other hand, often causes all lanes to execute a shared memory instruction at the same time, which makes conflicting accesses more likely.

10.1.6 *Latency Hiding*

In conventional GPU architectures, the multitude of active warps residing on a core or warp scheduler is used to hide the latency of currently executing instructions. A rather large number of warps is required to hide the latency of deep pipelines or long-latency memory operations [157]. In our TSIMT architecture, warps and the execution back-end of the core are partitioned into a number of TSIMT lanes. This means that while the pipeline depth and memory latency remain unchanged, the number of warps available for latency hiding within each lane decreases considerably (i.e. by a factor equal to the number of lanes per core). This does not necessarily impact performance negatively though, as a single instruction contributes significantly more latency hiding ability in a TSIMT GPU core than to a conventional one: In a conventional GPU, having one independent warp instruction available for execution corresponds to one clock cycle of latency hiding. In the TSIMT core, on the other hand, a single independent warp instruction keeps a TSIMT lane busy for up to 32 clock cycles, depending on its active mask.

10.1.7 *Register File*

TSIMT register files use the same basic design idea as the register files of conventional GPUs: Instead of using costly multiported memories, multiple single ported SRAM banks are used [48], [196]. These register banks are connected using a crossbar to an operand collector. The operand collector fetches the operants over multiple cycles. In TSIMT instead of using a single very wide register file with one 32-bit entry for each thread of a warp, each lane implements one small 32-bit wide register file. In a conventional SIMT register file only a whole warp wide register entry can be addressed. Even if just a single thread is active and we are only interested in the operand for that thread, a whole 1024-bit wide entry (32-bits for each of the 32 threads of warp) would be fetched. In TSIMT only the operands of active threads are fetched. Using individual register file lanes rather than a single monolithic register file gives more flexibility with the placement of the components and helps to keep distances between register file and execution units small. However, it also divides the register file into multiple parts. The register file in each lane stores the registers for warps assigned to that lane. Other lanes cannot execute instructions from these warps as there is no connection between the lanes that would allow operands from register file of one lane to be passed to an execution unit located in a different lane. The register file of each lane provides only enough bandwidth for executing instructions at full speed in one lane. For this reason, adding additional connections to allow the execution of warps on other lanes would not increase performance, as the execution units would stall because one register file lane cannot supply operands fast enough to keep multiple lanes running at full speed. All registers required by one warp must fit into a single lane as it is not possible to store some registers of the warp in one lane and some registers of the warp in the register file of a different lane. No additional warp can be allocated if all lanes together have sufficient free register resources for one or multiple additional warps, but no lane alone can provide enough space for an additional warp.

The operand collector reads and writes the operands in multiple cycles from multiple banks. Reads and writes of multiple instructions are overlapped. In TSIMT we can use this structure to fetch the operands for the different active threads. This multi-cycle operand fetch avoids the need for an area and power-hungry multiported register file. However, depending on the register allocation and divergence pattern, load balancing problems between the register banks can appear and cause stalls.

Two optimizations related to the register file allocation are explored in this chapter: First, register resources are freed on warp exit instead of block exit and second, partially filled warps only allocate registers for each active thread instead of the entire warp. We use TSIMT+ to refer to an optimized version of TSIMT that implements these two optimizations.

The first optimization makes it possible to launch new thread blocks sooner: In the conventional GPU, as modeled by GPGPU-Sim, register resources are managed at the thread block level. Registers allocated to a warp can only be reused after the entire thread block has finished executing. This potentially leaves many register resources unused for extended periods of time. With the optimization, warp resources are freed as soon as a warp finishes execution. Consequently, new blocks are launched as soon as sufficient resources are available. A similar approach has been described for conventional SIMT GPUs by Xiang et al.[197]. Other than their solution, however, our solution only permits the launch of a new block if sufficient resources are available to launch a full thread block.

The second optimization can increase occupancy if thread block sizes are not divisible by the warp size. For example, if a thread block size of 112 is requested in a regular GPU, registers for 128 threads are allocated. Our

Table 10.1: Area estimates for different possible register file implementations at 40nm

GPU Register File using a single 256-bit wide lane

Component	Width	Size/Ports	Area (mm^2)	Number	Total Area (mm^2)
SRAM	256-bit	8192 Byte	0.0296	8	0.2365
Crossbar	256-bit	8x8	2.0302	1	2.0302
Total					2.2667

GPU Register File using 8 independent 32-bit wide lanes

Component	Width	Size/Ports	Area (mm^2)	Number	Total Area (mm^2)
SRAM	32-Bit	1024 Byte	0.0036	64	0.2279
Crossbar	32-bit	8x8	0.0333	8	0.2664
Total					0.4943

optimization allocates only the registers for 112 threads, i.e. the restriction to allocate registers with warp size granularity is removed. In the case of 112 threads, three full warps of 32 threads and one half filled warp would be allocated. When the next block is allocated, another half filled warp is allocated to the lane where the half filled warp from the first block resides. This optimization is not possible in the register files of conventional GPUs as it is enabled by the ability of TSIMT register files to address registers with a per-thread granularity. The register files of conventional GPUs can only be addressed with a per-warp granularity, because of this limitation partially filled warps leave some register file space unusable in conventional GPUs.

10.1.8 *Area*

We expect that the die area required by a TSIMT GPU should be close to the area required by a SIMT GPU with an otherwise identical configuration. As already explained in Section 10.1, most structures are unchanged from a SIMT GPU. On die storage is almost unchanged: Only a few additional bits for the instruction register and the storage of the active mask are required per TSIMT lane.

The number of bits in the register file stays the same, however, they are distributed over a higher number of narrower banks. [198] estimated an 18.7% increase in register file area, but we think this is an overestimate. We used CACTI 6.5 to estimate the area of the SRAM banks and crossbars used in the GPU register file. We tested two potential designs: First, a monolithic 256-bit wide register file and second, a register file with 8 narrow 32-bit wide lanes with independent decoders and crossbars. We show the results of this estimation in Table 10.1. The second option is more flexible and smaller at the same time. Even the SRAM banks are slightly smaller, but especially the crossbar is reduced in area: the narrow input and output ports greatly reduce the distances between the different ports. Splitting the register file into lanes, results in an interleaved implementation of the register file that reduces the length of required wiring. The additional flexibility offered by the second design is required for TSIMT, but the second design can also be used to implement the register file of a conventional GPU. As a conventional GPU does not require all the flexibility offered by this design, slightly less area is likely needed as some parts of the address decoders could be shared by multiple lanes. The large area predicted for the first design is likely a result of a weakness of CACTI 6.5. Yifan He, Yu Pu, et al. found 128-bit wide dual port memory to be best in terms of energy efficiency and speed [199], [200].

10.2 EXPERIMENTAL EVALUATION

In this section, the proposed TSIMT GPUs are experimentally evaluated using a GPU simulator. It is organized as follows: Section 10.2.1 describes the experimental platform as well as the benchmarks employed. Section 10.2.2 evaluates the properties of the TSIMT core using a synthetic microbenchmark. In Section 10.2.3 TSIMT is evaluated using real benchmarks, afterwards Section 10.2.4 explores load balacing issues we discovered in TSIMT. Section 10.2.5 describes how optimization to the resource allocation can reduce these issues and in Section 10.2.6 the performance effects of different design tradeoffs are examined. Section 10.2.7 evaluates STSIMT.

10.2.1 *Experimental Platform and Benchmarks*

For microarchitecture simulations, we utilized the cycle-level GPU simulator GPGPU-Sim 3.2.1 [149] and extended it to support TSIMT. Table 10.2 shows our used GPU configuration. We selected a similar configuration as the configuration used in [122], however, these results are still not directly comparable, because Fung et al. used a much older version of GPGPU-Sim. For the evaluation, we selected a large set of benchmarks listed in Table 10.3 from multiple widely-known sources such as the popular Rodinia benchmark suite [162] and the GPGPU-Sim repository [149]. We also included a version of breadth-first search using the virtual warp-centric programming model [201].

The benchmarks are selected to contain both very control-divergent kernels as well as almost and fully coherent kernels to be able to see the performance impact of TSIMT on both types of applications. The SIMT bars in Figure 10.3 quantify the degree of divergence by showing the average SIMD efficiency for each benchmark without compaction. For each warp instruction, SIMD efficiency is defined as the ratio of active threads to the maximum number of threads per warp. The maximum SIMD efficiency that can be achieved is therefore 1.0. To arrive at the average SIMD efficiency for the entire kernel,

Table 10.2: GPU configuration used for experimental evaluation.

Parameter	Value	Parameter	Value
GPU cores	30	SP units / core	8
Memory channels	8x 64-bit	SFU units / core	2
Core clock	1300 Mhz	L1 D-cache / core	32 KB
Interconnect clock	650 Mhz	L1 I-cache / core	4 KB
Memory clock	800 Mhz	L2 cache	1 MB
Shared mem. / core	64 KB	Max. warps / core	32
Max. blocks / core	16	Process Node	40nm

Table 10.3: GPGPU benchmarks used for experimental evaluation.

Abbr.	Description	Kernels	Domain	Blocks per Grid	Threads per Block	Source
AES	AES Encryption	1	Cryptography	257	256	[149]
BFS	Breadth-first search	2	Graph Algorithms	1954	512	[162]
BWC	BFS warp centric	1	Graph Algorithms	977	128	[201]
BP	Back Propagation	2	Pattern Recognition	4096	256	[162]
CP	Coulombic potential calculation	1	Physics Simulation	256	128	[149]
DG	Discontinuous Galerkin solver	3	Physics Simulation	268, 268, 603	84, 112, 256	[149]
GAU	Gaussian Elimination	2	Linear Algebra	1,2704	512,16	[162]
HOTSP	HotSpot	1	Physics Simulation	1849	256	[162]
IDCT	H.264 IDCT	2	Video Compression	252,231, 100, 130	192	[202]
LIB	Libor stock option calculation	2	Computational Finance	64	64	[149]
LPS	3D Laplace Solver	1	Physics Simulation	100	128	[149]
LUD	LU decomposition	3	Linear Algebra	1-155	16,32,256	[162]
MC	H.264 Motion Compensation	2	Video Compression	8160	64	[203]
MGST	Merge sort	4	Sorting	256,4, 4,2048	512,256, 256,128	[204]
MUM	DNA sequence alignment	2	Bioinformatics	196, 316	256,256	[162]
NN	Nearest Neighbors	1	Data Mining	168	256	[162]
NW	Needleman wunsch	2	Bioinformatics	1-128	16	[162]
PR	Parallel reduction	2	Parallel Algorithm	64,1	256,32	[204]
RAY	Raytracing	1	Computer Graphics	512	128	[149]
STO	StoreGPU	1	Database	1-260	2-64	[205]
VIS	Visibility Calculation	1	Game AI	24	256	AiGD

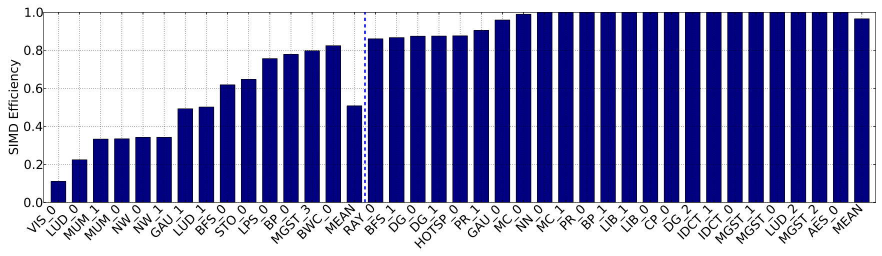


Figure 10.3: Effective SIMD efficiency for conventional SIMT.

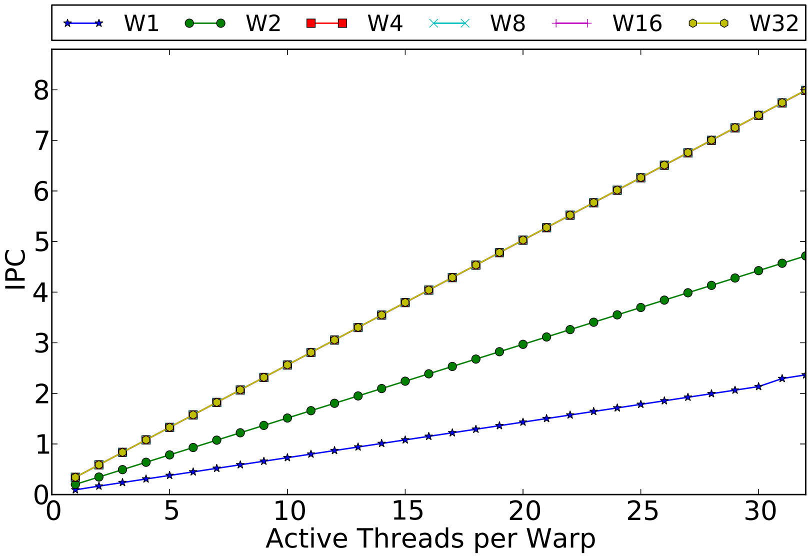
the per-instruction SIMD efficiency is averaged over all executed instructions. Figure 10.3 groups the benchmark kernels into two categories separated by the blue dashed line. *Divergent benchmarks* are shown left of the line, these kernels have an average SIMD efficiency of less than 85%. The remaining kernels to the right of the line are called *coherent* benchmark kernels.

10.2.2 *Synthetic Benchmark Analysis*

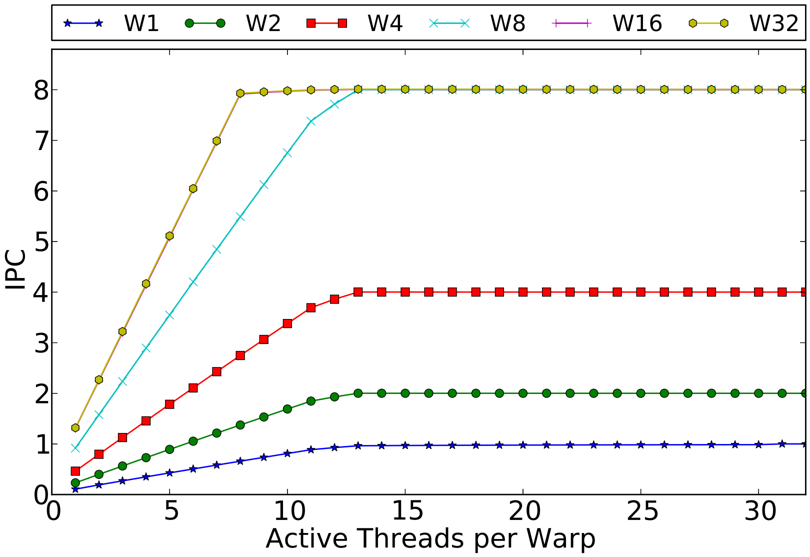
To demonstrate the performance of a TSIMT-based GPU architecture in an isolated fashion, we developed a microbenchmark that enables us to precisely control both the *warp active masks* and the *overall number of active warps* (i.e. occupancy). We executed this microbenchmark both on a conventional GPU core and on a TSIMT GPU core, while varying the number of active threads per warp and occupancy, and measured core IPC. For this experiment, the core's configuration is as described in Table 10.2 (execution throughput of 8 thread instructions, warp size of 32 threads). The results of these experiments are shown in Figures 10.4a and 10.4b. Both figures show the IPC achieved as a function of the number of active threads per warp for different numbers of active warps. (1 (W1) up to 32 (W32)).

In the conventional GPU (Figure 10.4a), the effect of reducing branch divergence is a corresponding linear increase in IPC (with an increase in active threads per warp). The maximum per-core IPC of 8 is only achieved when the execution is *coherent*, i.e. there is no control divergence. The number of active warps has no effect on performance, provided it exceeds 4. The measurements for 4, 8 and 16 active warps are hidden behind the results of 32 active warps, as 4 warps are sufficient to provide full performance. Additional warps are not needed to tolerate the latency of the arithmetic pipeline but help to tolerate memory latency. As having many warps is important for hiding instruction latency on GPUs, the effect of having a small number of available warps is directly linked to the type of instructions executed. As our microbenchmark utilizes math instructions with relatively short latency, only the pipeline latency must be hidden. This effect is observed in the figure, where configurations with 1 and 2 active warps are unable to fully hide the latency and cannot reach full performance.

The microbenchmarking results on the TSIMT architecture (Figure 10.4b) are vastly different. We begin by looking at the effect of the number of active threads per warp in the full-occupancy configuration with 32 active warps. The figure shows that the maximum IPC of 8 is reached much sooner than on the conventional GPU at only 8 active threads per warp. Below this number performance increases linearly with the number of threads. This behavior is caused by insufficient instruction issue bandwidth: On a GPU configuration with 8 TSIMT lanes and an instruction issue bandwidth equal to that of the baseline GPU (i.e. one warp instruction per clock), the scheduler will be busy for exactly 8 clock cycles before it can re-issue to the same lane again. Therefore, each lane must be able to *at least* hide a number of clock cycles equal to the number of lanes per core with execution. To hide 8 clock cycles, a lane requires a warp instruction with at least 8 active threads. If there are fewer than 8 threads active per instruction, the lane completes the instruction before the warp scheduler can issue a new instruction to it.



(a) Regular SIMT-based GPU



(b) TSIMT-based GPU

Figure 10.4: Synthetic Benchmark for performance characteristics of regular SIMT and TSIMT-based GPUs

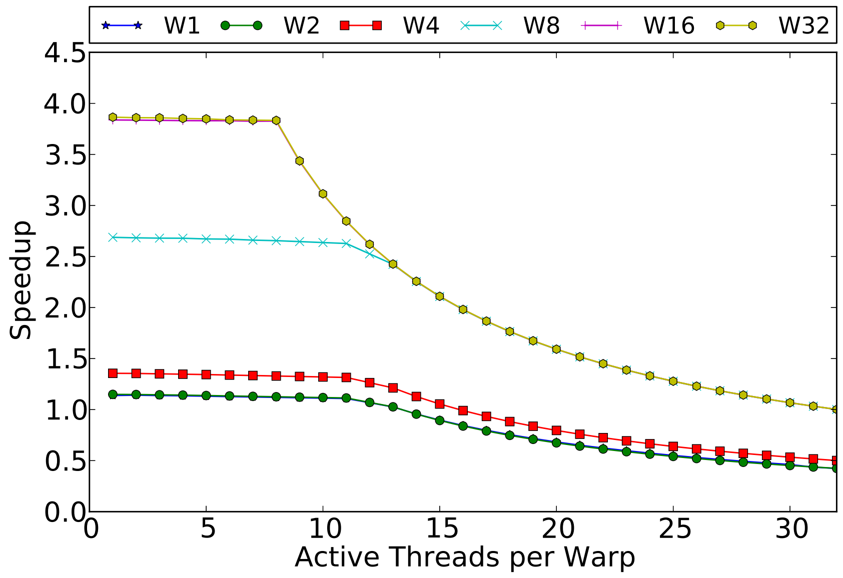


Figure 10.5: Microbenchmarking results showing the speedup of TSIMT over regular SIMT for different combinations of active warps and threads per warp.

Next, we consider the effect of the number of active warps on TSIMT GPU performance. Figure 10.4b shows that a small number of active warps has a stronger impact on the performance of the TSIMT-based GPU than on the conventional GPU. In fact, having only a few warps available on the core enforces an *upper bound* on the achievable performance within that core. The figure demonstrates that this upper bound is equal to the number of available warps, e.g. with 4 active warps, the maximum achievable core IPC is 4. This can be explained by the warp allocation scheme in the TSIMT architecture: As the warp set is statically partitioned across all TSIMT lanes, having fewer than 8 active warps on the core means that some TSIMT lanes will not have any warps allocated to them. As a result, TSIMT cores can never reach full performance if the number of warps is so small that some lanes remain empty.

Comparing the results for SIMT and TSIMT reveals that TSIMT provides only relatively small speedups, when the average number of active threads per warp is high. Even severe slowdowns by 50% are possible in case less than 8 warps are available. On the other hand, speedups between 2.5× and 4× are possible if 8 or more warps are available and 12 or fewer threads per warp are enabled.

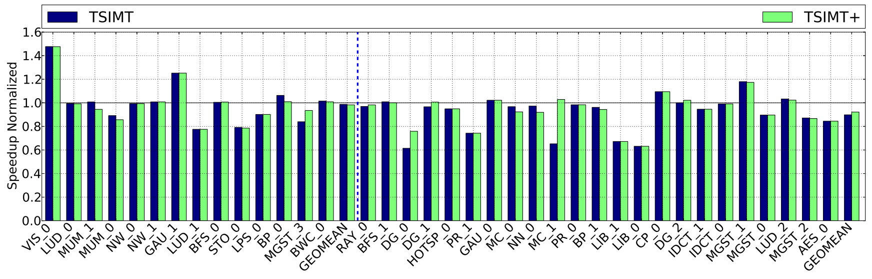


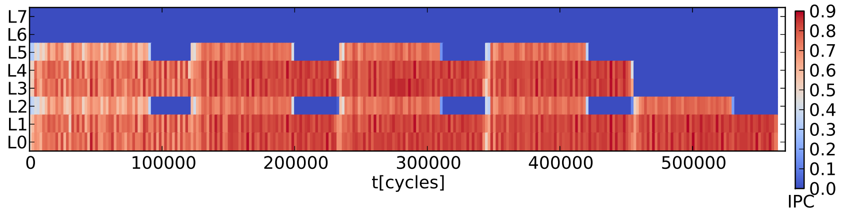
Figure 10.6: TSIMT GPU Speedup compared to the baseline

Additional insights can be gained by considering the speedup over regular SIMT. The speedup is shown in Figure 10.5. The speedup peaks close to four with 16 and 32 warps and 8 active threads per warp. For smaller numbers of active threads, TSIMT is unable to provide additional speedup as the warp scheduler cannot issue new instructions to the lanes any faster. In configurations with large numbers of active threads but small numbers of active warps, TSIMT exhibits slowdowns over regular SIMT. In the microbenchmark, the largest possible slowdown is equal to 0.5, as regular SIMT performance also decreases with only one or two active warps due to the inability to hide the pipeline latency. The slowdown of TSIMT over regular SIMT can be larger for real-world kernels, however, as such kernels normally contain some amount of ILP, which increases the ability of SIMT GPUs to hide the pipeline latency, even if only 1 or 2 warps are active. For TSIMT, on the other hand, ILP does not increase the performance when only a single warp is active. While ILP provides more independent instructions to the warp scheduler, these instructions can only be executed on the lane that is already busy. Lanes without any active warps remain idle. For this reason, benchmarks with only a few active warps per SM can experience drastic slowdowns with TSIMT. In the worst case (1 active warp, no control divergence, large amounts of ILP), TSIMT can never achieve more than one eighths of SIMT's performance.

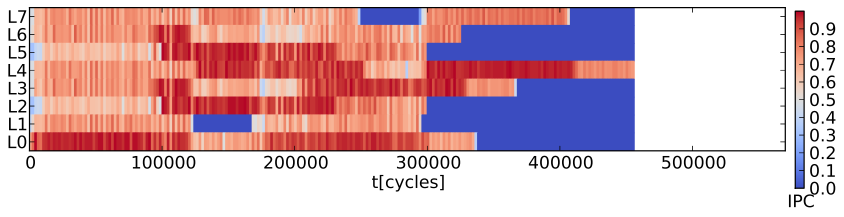
10.2.3 *Full Benchmark Analysis*

Figure 10.6 depicts the speedup of a TSIMT-based GPU over the conventional GPU. The benchmarks on the horizontal axis are sorted by increasing average SIMD efficiency. The dotted line separates the divergent (left-hand side) from the coherent (right-hand side) benchmarks. For both types of benchmarks, the geometric mean is shown as well.

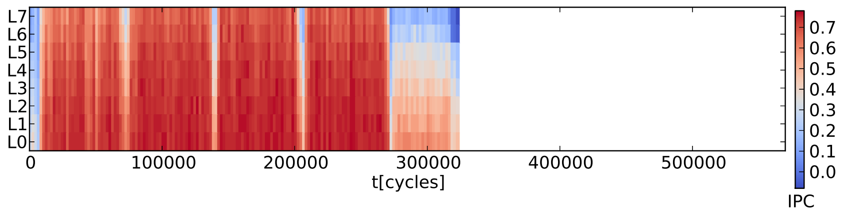
As the figure reveals, a straight implementation of TSIMT does not perform as well as one might expect. There are cases where TSIMT provides substantial performance benefits, but the overall effect from TSIMT is an average performance loss of 7.3%, both for the divergent as well as for the coherent benchmarks. A significant performance improvement is obtained, e.g., for the GAU_1 kernel, but a severe slowdown is incurred in other kernels such as LUD_1 or the (coherent) DG_1 kernel. Some coherent benchmarks show increased performance due to the changed shared memory handling. As explained in Section 10.1.5 TSIMT can reduce the number of shared memory bank conflicts. Interestingly, the five most divergent kernels (LUD_0, MUM_0, MUM_1, NW_0, NW_1) experience either no change or a performance loss on TSIMT. Looking at the SIMD efficiency provides a first hint of the possible performance improvements. As Figure 10.3 reveals, even most divergent benchmarks have SIMD efficiencies of more than 50%, and only LUD_0, MUM_1, MUM_0, NW_0 and NW_1 have SIMD efficiencies below this level. As discussed already in the last section, kernels with high SIMD efficiency usually cannot benefit from TSIMT.



(a) DG_0 benchmark on unoptimized TSIMT



(b) DG_0 benchmark on optimized TSIMT



(c) DG_0 benchmark on TSIMT without lane lock

Figure 10.7: IPC over time for each TSIMT lane.

10.2.4 *Load Balancing Issues*

While investigating the matter we discovered that the limited performance improvements of TSIMT were due to load balancing issues. To illustrate these issues, we developed a tool that generates graphs that show IPC over time for each lane of one core. One of these graphs is shown in Figure 10.7a for the DG_0 kernel. It can be seen that lanes 6 and 7 are completely empty, while lanes 2 and 5 frequently run out of work. DG_0's block size of 84 threads largely explains this behavior: A block of 84 threads is mapped to 2 full warps and 1 partially filled warp with 20 active threads. Furthermore, because of the high register requirements of this kernel, each core only holds at most 2 blocks simultaneously. The full warps are mapped to lanes 0, 1, 3, and 4 while lanes 2 and 5 execute the partially filled warps. Because the warps are only partially filled, they execute and finish much faster than the full warps. But this does not result in any performance advantage: The lanes must stay idle until the complete block is finished.

To gather even more insight into the effects of load balancing we recorded how many lanes on average had warps allocated to them and how many of these lanes had usable warps. Lanes can have warps allocated to them, but can still stall because all its warps are currently waiting for long latency memory operations. We recorded this information in the "lanes active" columns of Table 10.4. Static means that at least one warp is allocated to the lane. However, some of these lanes are still stalled, because all warps allocated to them are waiting for long latency memory operations. The dynamic column shows how many lanes on average have at least one warp available, that is not stalled by a long latency operation. This can also be considered to be the average effective width of the TSIMT core. In 19 out of 37 benchmarks more than 7 lanes on average have warps allocated to them, however, only 2 kernels have more than 7 lanes with usable warps. Some kernels such as BFS_0 or DG_2 have warps allocated to almost all lanes, but only a small number of lanes can be active because almost all warps are not available for scheduling since they are waiting for long latency memory operations.

10.2.5 *Register Allocation Optimizations*

As explained in Section 10.1.7 two optimizations of TSIMT register resource allocation can potentially improve the performance: First, resource deallocation on warp instead of block exit and second, allocating registers only for active threads instead of allocating register for the entire warp.

Figure 10.7b again shows IPC over time per TSIMT lane for an execution of DG_0 with these optimizations. Most lanes are now busy at the start of the kernel. But at the end of the kernel a strong tail effect is visible: only 1 of the 8 lanes are still busy.

The performance of these optimizations is shown in Figure 10.6. The overall effect of these optimizations is a slightly better performing version of TSIMT, called TSIMT+, with 6.0% performance loss compared to the SIMT baseline and a 1.4% improvement over unoptimized TSIMT. For most benchmarks, the optimizations have no effect, but some particularly problematic cases (MGST_3, DG_0 and DG_1) show speedups between 5% and 10%. Unfortunately, in some benchmarks, the optimizations lead to extended tail effects, thereby causing slight slowdowns compared to TSIMT.

Table 10.4: Benchmark Scheduling Statistics

Kernel	Issue Conflicts			Lanes active		Warps
	1	2	≥ 3	Dynamic	Static	
AES_0	9.34%	1.45%	1.23%	4.70	7.66	7.66
BFS_0	8.92%	1.79%	0.75%	1.14	7.69	26.68
BFS_1	15.44%	4.45%	3.12%	3.06	6.84	15.25
BP_0	7.59%	2.05%	3.06%	6.72	7.97	23.14
BP_1	7.62%	0.81%	0.40%	7.02	7.97	15.16
BWC_0	5.11%	0.29%	0.03%	2.50	7.81	25.14
CP_0	4.95%	0.07%	0.01%	6.55	6.71	17.68
DG_0	7.18%	0.33%	0.01%	3.89	4.95	4.95
DG_1	10.17%	0.73%	0.04%	4.76	5.93	10.15
DG_2	10.45%	1.61%	0.27%	1.85	7.82	27.81
GAU_0	12.46%	1.78%	0.37%	1.11	1.61	2.41
GAU_1	17.87%	3.58%	0.80%	3.80	7.55	14.29
HOTSP_0	10.24%	2.12%	1.34%	5.34	7.72	7.72
IDCT_0	5.14%	1.14%	1.07%	4.65	7.47	19.89
IDCT_1	3.14%	0.44%	0.32%	4.86	7.67	20.47
LIB_0	1.44%	0.01%	0.00%	3.24	4.08	4.08
LIB_1	1.86%	0.02%	0.00%	2.87	4.11	4.11
LPS_0	16.61%	3.92%	1.42%	4.47	6.43	10.09
LUD_0	0.00%	0.00%	0.00%	0.13	0.33	0.33
LUD_1	0.00%	0.00%	0.00%	0.23	0.38	0.38
LUD_2	13.58%	1.78%	0.31%	4.92	7.05	20.95
MC_0	11.47%	1.17%	0.13%	7.47	7.86	19.31
MC_1	6.94%	0.45%	0.04%	5.14	7.85	20.73
MGST_0	7.04%	0.69%	0.28%	6.62	7.60	15.18
MGST_1	5.42%	0.73%	0.57%	0.87	2.57	2.57
MGST_2	10.13%	1.93%	1.09%	1.95	2.59	2.59
MGST_3	13.59%	2.03%	0.46%	5.22	7.77	28.56
MUM_0	7.26%	0.97%	0.31%	2.37	7.36	20.80
MUM_1	9.01%	1.05%	0.47%	1.08	5.98	8.91
NN_0	9.51%	1.31%	0.67%	6.53	7.06	21.11
NW_0	2.70%	0.05%	0.00%	0.23	2.54	2.54
NW_1	2.91%	0.06%	0.00%	0.20	2.55	2.55
PR_0	3.87%	0.08%	0.02%	1.95	7.92	16.77
PR_1	0.00%	0.00%	0.00%	0.13	0.33	0.33
RAY_0	8.91%	1.24%	0.31%	6.14	7.50	7.50
STO_0	2.43%	0.99%	1.05%	2.24	4.89	4.89
VIS_0	23.60%	6.71%	1.86%	0.66	3.03	3.03

10.2.6 *TSIMT Design Tradeoffs*

Another potential bottleneck in TSIMT can be the instruction issue bandwidth. Multiple lanes can finish execution of their current warp instruction in the same cycle, but the frontend can only supply a new instruction to a single lane each cycle. If two or more lanes request new instructions at the same time, all but one lane will stall because the frontend cannot supply a new instruction fast enough. To discover how common these stall cycles are, we recorded how often they happen on average and show this information in the "Issue Conflicts" columns of Table 10.4. The table shows how often 1, 2 or 3 or more instructions could not be issued as soon as they were ready for issue and their lane was ready to accept them, but the instruction frontend was busy with issuing an instruction to a different lane. In on average 7.9% of the cycles two lanes are waiting for instructions. This can also be seen as a temporary reduction of the average effective number of lanes. In some benchmarks such as VIS_0, GAU_1, BFS_1 and LPS_1 in more than 20% of the cycles one or more lanes are stalled because the frontend cannot supply instructions fast enough.

In the previous section, we already noticed that load balancing issues between different lanes hurt the performance of TSIMT. Additionally, we cannot exploit ILP in TSIMT as all instructions, even if independent, need to be executed in the same lane. To determine how much these issues reduce the performance of TSIMT, we also simulated an unrealistic configuration of TSIMT, where all warps can issue instructions to all lanes. The overall effect of removing the locking of warps to a specific lane is a performance improvement of 7.6% compared to SIMT and of 16.4% over TSIMT.

A more realistic approach than removing the lane locking for improving the TSIMT load balancing is to reduce the warp size. We simulated a configuration with warp size of 16 and found that overall the performance improves by 1.4% over SIMT. Reducing the warp size improves the load balancing as more warps are available and the GPU can exploit ILP within a warp for additional performance. However, reducing the warp size increase the load of the frontend. With a warp size of 16, instruction fetches need to be amortized over a smaller number of threads. But the reduction of warp size can also have positive effects on divergent memory accesses. The results show that on average smaller warps are better than TSIMT with 32-wide warps.

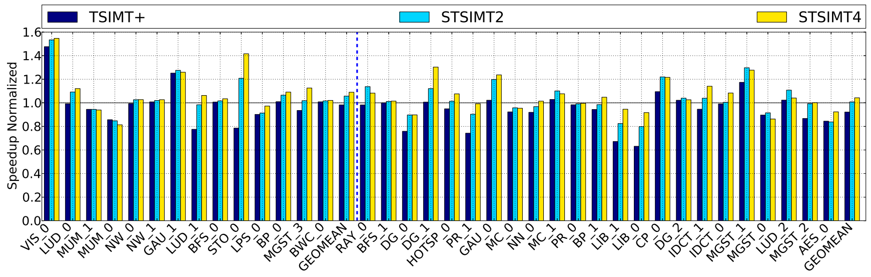


Figure 10.8: Speedup for STSIMT with different lane width

10.2.7 *Spatio-Temporal SIMT*

While the optimizations presented in Section 10.2.5 help some applications, they do not resolve the main problem of TSIMT: Severe performance reductions when only a few warps are available and/or load balancing issues between the lanes. As described in Section 10.1, spatio-temporal SIMT reduces the impact of these problems, as the warp pool is partitioned across a smaller number of lanes: With only four or two lanes, it becomes much more likely that each lane receives at least one active warp and that the work is distributed equally over all lanes. At the same time it also reduces the latency.

Figure 10.8 shows the performance of regular SIMT, TSIMT and STSIMT with two and four ALUs per lane. The number of lanes was adjusted to keep the number of functional units (FUs) identical in all configurations, i.e. TSIMT has 8 lanes with one FU each, STSIMT2 has 4 lanes with 2 FUs each, and STSIMT4 has 2 lanes with 4 FUs each. In the results, we observe improvements over the conventional SIMT architecture for both two and four wide STSIMT. The maximum speedup is observed in the STO benchmark where regular TSIMT experiences a 15% slowdown compared to the baseline while STSIMT4 shows a 51% speedup. Due to the low active warp count, the STO benchmark was not performing well on TSIMT. On average, STSIMT4 performs about 6% faster than the baseline on the divergent benchmark set and 10% faster than TSIMT. For the coherent benchmarks, STSIMT4 is 12.6% faster than TSIMT and 5.9% faster than the baseline. The very short GAU_0 benchmark exhibits an unusual behavior: It runs much faster despite having almost no divergence. This happens because a large part of the divergent instructions in this benchmark are very slow division instructions that are responsible for tail effects. The convergent instructions are mostly simple and fast instructions, with little influence on the total runtime of the kernel.

10.3 summary

In this chapter we presented TSIMT and the STSIMT extension. We proposed several optimizations: We showed how optimized instruction scheduling and better management of register file resources can significantly reduce tail effects. We explored the performance of TSIMT with a microbenchmark to show the relationship between TSIMT IPC, active warps and branch divergence. We show that while optimized TSIMT offers speedups on some benchmarks, on average TSIMT results in a lower performance than regular SIMT, as a low number of active warps often results in significant performance reductions for TSIMT and most benchmarks have a high SIMD efficiency and gain little from TSIMT. For these workloads, STSIMT4 provides better performance and performs about 6% faster on average than regular SIMT, and some benchmarks improve their performance by more than 40%. In the next chapter, we will look at how we can optimize performance and energy efficiency of TSIMT and STSIMT by removing redundant computations with a technique called Scalarization.

11 SCALARIZATION

The work presented in this chapter was previously published: J. Lucas, M. Andersch, M. Alvarez-Mesa, *et al.*, "Spatiotemporal SIMT and Scalarization for improving GPU efficiency," *ACM Transactions on Architecture and Code Optimization*, Sep. 2015. DOI: 10.1145/2811402.

Previous work [129], [130], [133] has shown that in SIMT architectures several threads often redundantly perform the same calculation on the same vector operands. Such situations are common because in many cases it is easier and faster to recalculate results in different threads than to calculate the results only once and to broadcast them to all threads. Redundant calculation not only wastes execution throughput, it also wastes storage as well as energy since copies of the calculated values have to be stored for every thread.

Redundant calculations can be removed by applying a technique called Scalarization [129], [133]. In this technique, a static compiler algorithm is used to identify instructions that always use the same operands in all active threads of a warp. Likewise, it also identifies registers that always store the same value in all threads of a warp. These instructions are then only executed once per warp instead of once per thread, also the identified registers are stored once per warp instead of once per thread. The hardware cost of integrating Scalarization in the proposed TSIMT architectures are much lower compared to regular GPUs, because since most of the execution resources can be reused for the scalar and vector datapaths. Furthermore, we improve upon the Scalarization algorithm proposed in [129] by allowing Scalarization, even with divergent control flow.

In this chapter, we present the following contributions, related to TSIMT and Scalarization:

- We show how Scalarization can be integrated in the proposed TSIMT architecture in a way that requires less hardware than its integration in conventional SIMT GPUs.
- We present an improved Scalarization algorithm.
- We show that STSIMT with Scalarization improves the energy-delay product (EDP) by more than 25% compared to the SIMT baseline.

The chapter is organized as follows: First in Section 11.1 we describe the hardware changes required to support Scalarization on (S)TSIMT based GPUs. Section 11.2 outlines the compiler algorithm used to identify Scalar instructions and the registers that contain them. Section 11.3 explains how the Scalarization algorithm was integrated into our simulation framework. In Section 11.4 we evaluate the Scalarization algorithm on a wide range of benchmark and discuss static properties of the kernels after Scalarization. In Section 11.5 Scalarization is combined with TSIMT and we discuss the performance benefits of Scalarization. We continue our evaluation of Scalarization in Section 11.6 by estimating the power usage and energy efficiency of TSIMT+Scalarization. Section 11.7 concludes the chapter with a short summary.

11.1 HARDWARE SUPPORT FOR SCALARIZATION

Vector processors usually combine a scalar execution units with wide vector units [206]. This combination can also be found in many accelerators, e.g.: [200], [207]–[209]. In these accelerators and in conventional SIMT GPU architectures, implementing support for Scalarization requires separate execution units and register files for scalar values and a broadcast network to transport values from the scalar register file to the vector execution units. AMD's GCN architecture is an example of such an architecture [78]. It combines scalarization and a spatial SIMT GPU. In a TSIMT based GPU, however, we may use the same ALU and register file for both scalar and vector operations. The additional logic required for scalarization is limited to small changes: An additional addressing mode in the register file is needed for scalar registers. Scalar registers can be packed more densely as we only need to store one value per warp instead of one value per thread. Beside this difference in register file addressing, execution of scalar instructions is handled just like execution of regular vector instructions with a single active thread. This reduces not only the additional hardware required for scalarization, but also enables more flexibility: As opposed to conventional GPU architectures, where the separate scalar ALUs stay idle when no scalar instructions are available from the currently active warps, in TSIMT GPUs with scalarization one type of ALU is used for both scalar and vector instructions. This enables flexible adjustment to any ratio of vector and scalar instructions.

11.2 COMPILER SCALARIZATION ALGORITHM

We present a new Scalarization algorithm for code analysis, that is able to identify instructions that are guaranteed to use the same inputs in all active threads of a warp. It also identifies which registers always store scalar

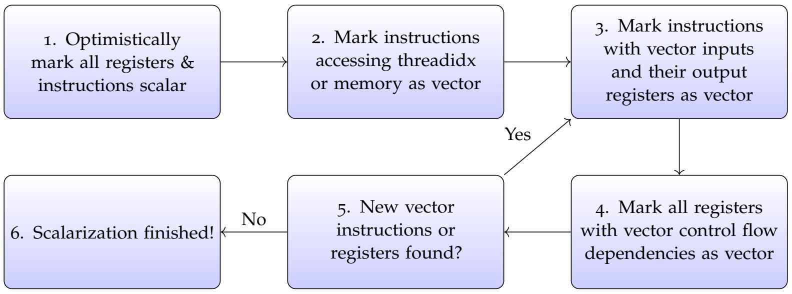

1. Optimistically mark all registers & instructions scalar
2. Mark instructions accessing threadidx or memory as vector
3. Mark instructions with vector inputs and their output registers as vector
Yes
4. Mark all registers with vector control flow dependencies as vector
5. New vector instructions or registers found?
No
6. Scalarization finished!

Figure 11.1: Scalarization Algorithm

values. The algorithm is shown in Figure 11.1. The algorithm starts by optimistically marking all registers and all instructions as scalar (Step 1). Then, instructions reading the thread local memory or the *threadid* register are marked as non-scalar (Step 2). Instructions reading non-scalar registers are also marked as non-scalar (Step 3). Registers written by non-scalar instructions are also marked non-scalar (also Step 3). In Step 4 registers with control flow dependencies on vector values are marked as vector. The main difference of our algorithm compared to [129] lies in this stage: In the algorithm from Lee et al. all registers and instructions are marked as non-scalar where convergent(=non-divergent) control flow cannot be guaranteed. We found that their criterion is safe but too strict. If convergent control flow cannot be guaranteed, we can still scalarize as long as the register goes dead before the reconvergence point. This way only a single version of the register per warp can exist at the same time and a scalar register can be used. If the register would be alive beyond the reconvergence point, its value in different thread could differ due to the divergent control flow of the threads.

After Step 4 we check if any new vector registers or instructions have been found, if yes we repeat steps 3 to 5, if nothing changes we have found all vector registers and instructions. All instructions and registers that are still marked scalar are guaranteed to be uniform for the whole warp and can benefit from the Scalarization capabilities of the hardware.

11.3 implementation of the scalarization algorithm

The experimental evaluation presented in Section 11.3 uses GPGPU-Sim 3.2.1 [149] extended with our enhancements. By default, this GPU simulator does not simulate a real instruction set of any GPU but simulates NVIDIA's PTX intermediate code instead. PTX uses a generic ISA with an unlimited number of virtual registers. In real systems, the PTX code is mapped by the driver to the actual ISA of the employed GPU. We implemented the presented Scalarization algorithm in the PTX loader of the simulator. After parsing the PTX code and identifying the basic blocks and recovergence points, the algorithm explained in Section 11.2 identifies scalar instructions and registers as well as vectors instructions and registers. In a real system, the driver would subsequently map the PTX code to the actual ISA of the GPU. This mapping includes register allocation. Unfortunately GPGPU-Sim, is not able to simulate this part of the mapping process but instead simulates an unlimited register file and queries NVIDIA's *ptxas* tool to inquire the number of registers required per thread after register allocation. This partial information about the results of register allocation is used by the simulator to restrict the maximum occupancy. This simulation shortcut of GPGPU-Sim is problematic as not only the number of registers needed per thread influences the performance but the register mapping also. Register allocation changes the timing of the register fetch and additional pipeline stalls may happen due to write-after-read hazards, that were not present in the PTX code prior to register allocation. GPGPU-Sim is also able to simulate PTXPlus code, that closely resembles the ISA of NVIDIA's Tesla architecture, however, as NVIDIA's Tesla architecture does not support Scalarization and scalarizing works with PTX code, all simulations in this chapter use PTX instead of PTXPlus.

To solve the issues with PTX we added a register allocator to GPGPU-Sim. A standard register allocator based on graph-coloring [210] was implemented. It first determines which virtual registers are alive at each instruction. Then an interference graph is constructed, in which every vertex represents a virtual register. The edges connect all virtual registers that are alive at the same time. Then all vertexes are colored so that no vertex is connected to another vertex of the same color. Each color represents a physical register. As GPUs support an adjustable number of registers per thread, we try to color using the smallest number of colors possible. As this is an NP-hard problem, we employ a heuristic [211]. To support Scalarization this algorithm is executed twice: Once for allocating vector registers and once for scalar registers.

An important change from the standard register allocation algorithm described above is required while constructing the interference graph: Additional edges need to be added to the graph to account for interferences between different threads from the same warp. Scalar registers are shared by all threads of a warp. For this reason, the control flow of warp and the effect of the reconvergence stack need to be considered. When threads execute a divergent branch a scalar register can be alive on one branch direction but dead on the other branch direction. When such a branch is executed, scalar registers that were considered dead from the perspective of a single thread can be "resurrected" when the control flow reaches the reconvergence point. This would, however, fail if the space occupied by the scalar register had been reused in the branch path, where it is dead. All scalar registers that are alive at the first instruction after a potentially divergent branch must also be considered alive at all instructions of the other side of the branch.

11.4 SCALARIZATION RESULTS

We executed our new Scalarization algorithm on the kernels described in Section 10.2.1. We verified that our algorithm works correctly and does not scalarize non-scalar registers or instructions by adding checks to the simulator. Many of the used GPU benchmarks also performed correctness checks on the output and did not detect errors. Figure 11.2 shows the number of registers required per warp, before and after Scalarization. Figure 11.3 shows how many executed instructions were scalarized by our algorithm. Without Scalarization, the kernels required an average of 24.2 vector registers per thread. After Scalarization 17.6 vector registers and 9.0 scalar registers are required on average per thread. While the sum of scalar+vector registers is slightly higher than the number of registers without Scalarization, each scalar register is allocated only once per warp instead of once per thread. This reduces the register file space needed per warp significantly: With 32 thread wide warps, the kernels require an average of 773.2 registers per warp without Scalarization, but with Scalarization, only 571.1 registers are needed per warp. With our new algorithm Scalarization reduces register file space required per warp by 26.1%. If we restrict the algorithm to scalarize only when convergent control flow can be guaranteed, as was proposed in [129], then less Scalarization is possible and 695.0 registers are required per warp. Likewise, the number of instructions that could be scalarized is also lower: With our algorithm 31.1% of the static instructions are classified as scalar and 30.3% of the executed instructions are scalar instructions, but with the restriction to convergent control flow only 12.9% of the instructions could be scalarized and 13.5% of executed instructions could be scalarized.

All benchmarks but one execute at least 6% scalar instructions. Only the STO benchmark executes almost no scalar instructions (0.4%). The GAU_0 kernel has the highest percentage of scalar instructions executed with 64.2%, the fraction of scalar instruction identified is slightly lower at 57.1%. LIB_1 has the highest number of static scalar instructions at 66.1%, but the fraction of executed scalar instructions is only 48.8%. Compared to the previous restricted Scalarization algorithm our algorithm scalarizes more than double the number of instructions and the number of registers is reduced by 17.8%. This Scalarization algorithm is well suited for code with complex control flow, as it makes Scalarization possible where convergent control flow cannot be guaranteed.

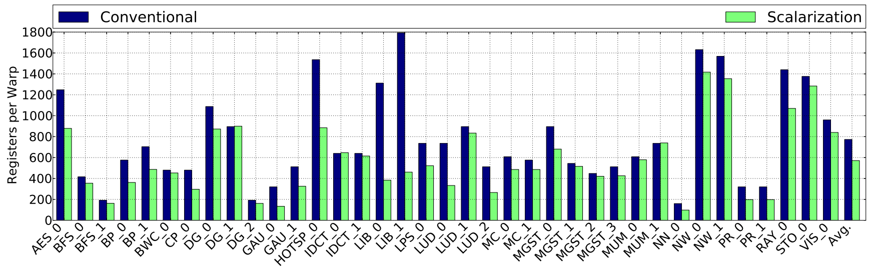


Figure 11.2: Registers per Warp

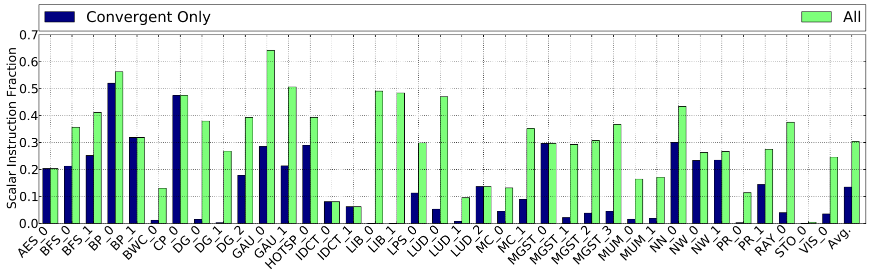


Figure 11.3: Scalar fraction of executed instruction for regular algorithm (all) and algorithm restricted to convergent control flow

11.5 TSIMT+SCALARIZATION

We combined Scalarization with TSIMT, and performed the experiments again. Figure 11.4 compares the performance achieved with SIMT with a GCN-style Scalarization (SIMT_SCALAR), TSIMT+, TSIMT+ with Scalarization (TSIMT_SCALAR) and the best spatio-temporal SIMT configuration without Scalarization (STSIMT4) and with Scalarization (STSIMT4_SCALAR). SIMT without Scalarization is employed as a baseline.

The geometric average of the speedup achieve by TSIMT+ due to Scalarization to the optimized TSIMT configuration is 16.1%. Gains from applying Scalarization to STSIMT4 are slightly lower with a 13.0% higher performance than in STSIMT4 without Scalarization. The highest Scalarization speedup of 4.2× over TSIMT+ can be seen in the BP_0 kernel. Three reasons explain the very high speedup of this kernel: First, a high ratio of 56% scalar instructions. Second, many scalar instructions are low throughput SFU instructions, while the vector instructions are mostly high throughput instructions. Third, the performance differences reorder the memory accesses and this improves the DRAM efficiency significantly from 12% to 42%. In MC_1, on other hand, Scalarization causes a slowdown of almost 40%. In this case, scalarization

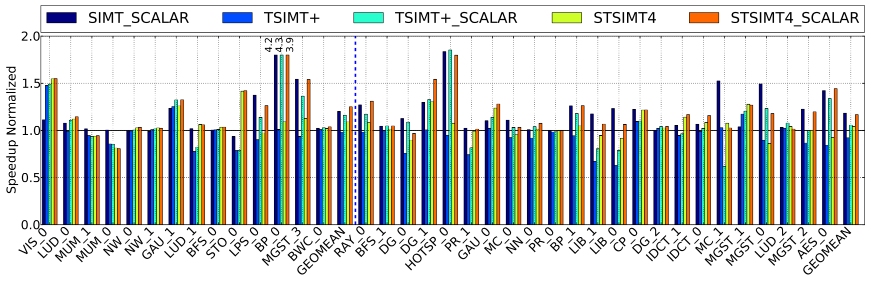


Figure 11.4: Performance with and without Scalarization for TSIMT and STSIMT4 Configurations, normalized to SIMT

allows placing 4 instead of 3 warps in a lane. This increased occupancy is normally beneficial, but in some rare cases such as in this kernel, the higher occupancy causes the number of shared memory bank conflicts to increase by more than 10× and also decreases locality, which results in 65% more read misses in the L1 data cache. Some kernels show almost no change in performance. In many cases this is connected to a low fraction of scalar instructions such as in STO_0, PR_0, IDCT_0 and IDCT_1. Some kernels such as BFS_1 use many scalar instructions but still do not profit from Scalarization. This happens if the performance of the GPU kernel is not limited by compute throughput but by another bottleneck. The BFS benchmark, for example, is a graph benchmark and is limited mostly by memory bandwidth and latency rather than compute throughput.

We also evaluated Scalarization on a conventional GPU. Similar to AMD's GCN architecture, an entire scalar datapath including an additional scalar register file and scalar execution units as well as a broadcast network to transmit scalar values back to the vector datapath was added to the simulated architecture. This additional hardware overhead is significant, which needs to be considered when comparing it to TSIMT+Scalarization. On kernels with little divergence, conventional SIMT+Scalarization performs slightly better (+1.3%) than STSIMT4+Scalarization. On benchmarks with higher divergence, however, STSIMT4 with Scalarization performs better than SIMT+Scalarization (+4.2%). Benchmarks that show high performance gains from Scalarization on one architecture such as BP_0, MGST_3, HOTSP_0 and AES_0 show high performance gains from Scalarization across all architectures. Benchmarks such as PR_0 or NW_1 that do not benefit from Scalarization on a conventional GPU do not profit from adding Scalarization to TSIMT either.

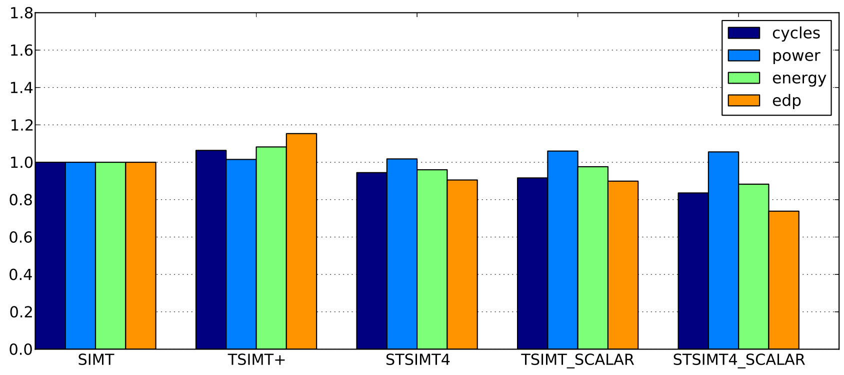


Figure 11.5: Geometric Mean of Runtime, Power, Energy and EDP

11.6 POWER AND ENERGY

We extended GPGPU-Sim to allow power modeling of TSIMT as well as SIMT GPUs. Figure 11.5 shows the power estimates of this power model for different variants of TSIMT. Despite lower performance on average the power consumption of TSIMT+ is approximately the same as that of regular SIMT (101% of SIMT power). These configurations are thus less energy efficient than regular SIMT. The architecture with STSIMT4 provides higher performance (5.6% shorter runtime) dissipating only slightly higher power (1.8% higher power), but lower energy consumption (−4.1%) and thus improve energy efficiency (EDP reduced by 9.5%). Scalarization results in significant performance gains (+9.3% over baseline TSIMT, +20% with STSIMT4). Because Scalarization improves the utilization of resources it increases the power consumption (5.9% for TSIMT, 5.6% for STSIMT4), but overall energy consumption is decreased because of the shorter execution time (−8.3% for TSIMT, −16.4% for STSIMT4). EDP is improved by 10.1% for regular TSIMT with Scalarization and by 26.2% for STSIMT4 with Scalarization. These results show that by combining STSIMT and Scalarization the energy efficiency can be improved significantly.

11.7 summary

In this chapter, we presented our Scalarization algorithm and the integration of Scalarization with TSIMT and STSIMT. We illustrated how accurate analysis of control flow and register lifetime can identify more scalar instructions and registers. We explained, how scalarized instructions can be executed using the same execution units also used for vector instructions and how scalar and vector registers can share the same register file. We showed how STSIMT4+Scalarization improves performance by 4.2% and improves the energy efficiency, as measured by the energy-delay-product, by 26.2%. With this chapter all our proposed and evaluated architectural enhancements have been presented and in our next and final chapter, we will draw conclusions and present ideas for future work.

12

CONCLUSIONS & FUTURE WORK

In this last chapter, we draw conclusions and present future work. We start with conclusions directly related to the individual chapters of this thesis in Section 12.1. The answers to our research questions from Chapter 1 are summarized in Section 12.2. We then conclude the chapter (and the thesis) in Section 12.3 with more general findings and directions for further research.

12.1 CONCLUSIONS

Chapter 4 presented our power measurement testbeds. They provided invaluable help for the research presented in this thesis. Their high sampling rate allowed us to measure the power consumption of short events. This allowed us to validate the accuracy of the GPUSimPow power simulator with regular short kernels. As an architectural simulator gpgpu-sim has a high overhead, simulating kernels is slow and feasible only for kernels that the real hardware can execute within several ms. Wallplug power meters would have required kernel runtimes of several seconds, and internal power meters as included on some GPUs would still only provide enough time resolution to measure kernels $150\,ms$ or longer accurately. The high time resolution was not only required for measuring short existing kernels with fixed length but also helped with the measurements of our microbenchmarks for empirical, data-dependent power models. These data-dependent power models presented in Chapters 6 and Chapters 7 required energy measurements for a huge set of testvectors. With the high time resolution of our measurement test beds, we could measure the energy consumption of several testvectors each second. Even at this speed our measurements were running for multiple weeks. Collecting the large set of measurements would not have been possible within a reasonable time frame without the high time resolution of the testbeds. The 16-bit or even 24-bit resolution provided by the testbeds was also important for this thesis. The calibration of many of our power models relied on difference measurements. Microbenchmarks were designed to only change the behaviour of one specific GPU component, while keeping all other components identical. This required accurately measuring small differences, well below 1 Watt in power consumption, while the GPU is consuming between 50 to 250 Watts. The high resolution of the testbeds allowed accurate measurements of these small differences. We discovered that when performing these measurements many small details matter: Leakage power changes significantly with temperature, difference measurements between similar kernels only produce meaningful results, when performed at nearly the same GPU die temperature. Twisting analog signal lines in the testbed reduces electrical noise from external electromagnetic interference in the measurements and improved the accuracy of the power model. Even the tiny differences of several parts per million in the clock sources of ADC and GPU can add up to large timing differences during longer measurements. These timing difference can cause large measurement errors in short kernels. Our software clock wander correction allowed us to correct the clockspeed differences and keep measurements accurate.

Modern GPGPU designs are pervading many areas of research and industry because of the massive compute power they offer. While the development of such designs is trying to drive performance further, GPU chips are increasingly limited by the power they consume and dissipate as heat. With this problem of the power wall, the design of new GPGPU architectures has become even more complex than before as consumed power is now an additional variable in the design space.

In Chapter 5, we have demonstrated a novel power simulation framework entitled GPUSimPow. With GPUSimPow, programmers and computer architects can accurately estimate the static and dynamic power consumed by a given GPGPU architecture when executing a particular kernel *without building the actual chip*. As our evaluations on a set of well-known benchmarks have shown, the average relative error of our power simulation results compared to measurements on real hardware is 11.7% for GT240 and 10.8% for GTX580. The simulator is also able to generate the distribution of power consumption over the hardware components of the GPU, and also of the different components of each core. These power profiles can be used to drive architecture or application power optimization. However, as a power breakdown for a selected benchmark revealed, a large fraction of the simulated power is currently attributed to components that are not modeled in detail, i.e. "undifferentiated transistors". Additional research could add accurate models of these components, however, as we have shown in Chapters 6 and 7, providing data dependent models for ALUs and memory is also very important for increasing the accuracy of the simulator.

The GPUSimPow simulator is a helpful tool for both processor architects and GPGPU programmers to gain valuable insights into where power is consumed in the GPU.

In Chapter 6 the design and calibration of ALUPower, an energy model for GPU ALUs, was presented and its accuracy was evaluated. The main contributions can be summarized as follows:

- We developed the ALUPower energy macro model for GPU ALUs based on measurements of commercial high performance GPUs.
- ALUPower improves the prediction accuracy by 85.6% over previous ALU energy models and exhibits an average correlation of 0.976 to measured results on real GPUs.
- We demonstrated the large ($\geq$ 30%) influence of data values on the energy consumption on both Fermi and Maxwell GPUs.
- We identified several potential energy optimizations for code running on GPUs, such as optimized register allocation or swapping operands to reduce energy consumption.
- ALUPower enables the development of new architectural optimizations to GPUs and similar architectures.

In the future, we intend to integrate the ALUPower energy model into GPUSimPow. This will enable the development of additional optimizations of the GPU architecture that cannot be evaluated properly using current GPU simulators, such as special warp or register fetch schedulers that are aware of values and try to execute instructions with similar inputs consecutively to reduce the power consumption. ALUPower can also be useful for modeling different GPU architectures as its coefficients can be scaled based on process node and voltage. Our LSFR benchmark can be used to calculate the scaling factor or, if an assembler is available, the microbenchmarks can be ported and new coefficients can be determined. DVFS and boost clocking schemes also benefit from more accurate energy predictions using hardware counters for input statistics such as average Hamming distances. Without a data dependent power model such as ALUPower, GPU architects aiming at reducing GPU ALU energy consumption are limited to techniques that reduce the number of executed instructions. ALUPower enables optimizations that reduce the energy consumption by reordering instructions to execute instructions with similar values consecutively on the same functional unit. It also enables a fair evaluation of techniques such as new warp schedulers that reorder instructions for different reasons, as reordering instructions can sometimes increase the energy consumption. Conventional power models leave GPU architects oblivious, while ALUPower makes them aware of these effects.

In Chapter 7, we continued the development of data-dependent GPU energy models with the MEMPower energy model for GPU memory transactions. Our contributions can be summarized as follows:

- We presented a novel technique to identify in which memory channel a specific memory address is located.
- Our microbenchmarks uncovered previously unknown architectural details of GF100-based GPUs.
- We show that memory channels are not completely identical, but differ in latency and energy consumption.
- The MEMPower model improves the energy predictions accuracy by on average 37.8% for loads compared to non-data dependent models and provides a 77.1% improvement on our validation set for stores.

At peak bandwidth data dependent changes to energy can influence the total power consumption of the GTX580 GPU by more than 25 Watt or around 10% of the total power. Future Work in this area includes software and hardware techniques to reduce the energy consumption. Common but expensive data patterns could be recoded to patterns with reduced energy consumption. As memory transactions are significantly more expensive than simple ALU operations, even software solutions could be beneficial. Programmer control over data allocation could allow rarely used data to be placed in memory channels with costlier memory access and often used data in memory channels with reduced energy consumption.

After modeling the power consumption of the memory, in Chapter 8 and Chapter 9 we looked at architectural changes to reduce the power consumption of the memory and its interface.

Chapter 8 presented a novel data bus inversion (DBI) encoding scheme. It reduces the link power consumption by up to 6%. It has been shown that the problem of finding a DBI encoding with the smallest link energy is equivalent to finding the shortest path in a graph. A hardware design that performs the encoding at the required data rates using an insignificant extra area and energy was presented. Additional optimization to reduce the hardware overhead including partially analog implementation are possible. A design with fixed coefficients provides a very good trade-off between the energy required for encoding and the saved link energy. It can be used without changing existing DDR4, GDDR5 and GDDR5X memories to reduce the interface energy during writes and could be integrated into future memories to also reduce read interface energy.

Chapter 9 proposed Sparkk, an effective approximate storage using commodity DRAMs. It achieves more than $10dB$ PSNR improvement over Flikker at the same average refresh rate or reaches the same quality at less than half the refresh rate of Flikker. We also proposed a simple, small and flexible hardware unit to control how the memory controller refreshes multiple configurable memory areas for approximate storage.

In Chapter 10, a microarchitecture implementation of temporal SIMT (TSIMT) was presented, as well as TSIMT optimizations, and a rigorous performance evaluation of temporal SIMT GPUs. TSIMT aims to improve the performance of control divergent GPGPU workloads by executing warps over time instead of over space as regular spatial SIMT GPUs do.

A microbenchmark analysis has shown that TSIMT offers significant performance benefits compared to spatial SIMT, provided there are sufficient warps are available. When evaluated with complete benchmarks, however, the basic TSIMT approach generally achieves lower performance compared to spatial SIMT. A detailed performance analysis has revealed that TSIMT suffers from lane load balancing and occupancy issues and microarchitecture optimizations have been presented to improve this for some benchmarks.

In addition, we have proposed and evaluated a more general solution, called spatio-temporal SIMT (STSIMT) that offers the control divergence mitigation of TSIMT while significantly reducing the high occupancy and load-balancing requirements of TSIMT. Using a particular configuration of STSIMT, an average speedup of 8% was achieved for control divergent benchmarks and 6% on average for all benchmarks.

In Chapter 11 Scalarization was combined with TSIMT. The hardware cost of this combination is much lower than a SIMT GPU with Scalarization, while being more flexible. It improves performance by 16% over regular SIMT. We also showed that a previously published scalarization algorithm employs overly restrictive rules, and presented a scalarization and register allocation algorithm, that is well suited for extracting scalar instructions from kernels with divergent control flow. By applying this algorithm double the number of instructions could be scalarized and 26.1% fewer registers were required per warp.

It has also been shown that several of the proposed designs provide significant power and energy advantages. The most energy-efficient design (STSIMT4 with Scalarization) improves the energy delay product by 26.4% on average.

Future work includes new methods for reducing lane load imbalance such as flexible warp sizes and GPUs that can dynamically switch between SIMT and TSIMT operation modes. Further code and microarchitecture optimizations are possible to increase the performance of TSIMT architectures. Moreover, current benchmarks are not targeted at and optimized for TSIMT and therefore often incur a performance reduction. If the programmer is targeting a TSIMT-based execution architecture, divergent code can be implemented in a more straightforward way and still be executed with high performance by the GPU. Future work could demonstrate the advantages of TSIMT-optimized code.

12.2 ANSWERS TO OUR RESEARCH QUESTIONS

In this Section, we summarize the answers to our research questions from Chapter 1. Our main questions were:

(A) How can we measure the power consumption of GPUs and kernels running on GPUs?
(B) How can we estimate the power consumption using an architectural simulator?
(C) Can architectural enhancements improve the energy efficiency of GPUs?

Question (A) was answered by the LPGPU and LPGPU2 power measurement testbeds presented in Chapter 4. Chapter 5 explained the design of an architectural power simulator for GPUs and Chapters 6 and 7 explained how more accurate, data-dependent models can be built. Together, these chapters answer Question (B). Chapters 8 to 11 explained various architectural enhancements that can improve the energy efficiency of GPUs and answer Question (C).

Additional questions regarding power measurements were:

(A_1) How can we acquire high quality GPU power measurements?
(A_2) How can power measurements be combined with application level event information?

In Chapter 4 we explained how power measurements using a shunt together with analog signal conditioning and an ADC with a high sampling rate and resolutions leads to high quality power measurements, answering Question (A_1). In the same chapter, we also explained our synchronization scheme and how it can be used to combine profiling results with our measurements. This generates combined power profiles, that list energy consumption for every executed kernel. This scheme provides the answer to Question (A_2).

We also asked the following questions regarding power modeling:

(B_1) Can an architectural simulator predict the power consumption of a GPU from its architectural level configuration?
(B_2) How to design microbenchmarks to measure the power consumption of individual GPU components?
(B_3) How can microbenchmarks and power measurements be used to discover unpublished architectural details?

Our power simulator GPUSimPow presented in Chapter 5 answers Question (B_1). It enables us to predict the power consumption of new GPUs by changing a configuration file. Many GPU components can be configured via the configuration file and their power is predicted using analytical power models. This allows us to predict how the power consumption would change, if e.g.: we change the number of cores, the size of the register file or change the memory interface. Chapters 5 to 7 used custom microbenchmarks to measure the power consumption of different GPU components and describe their design. A common theme of these microbenchmarks is the use of differential measurements and the use of specific architectural details to allow targeted changes to the activity of specific GPU components. This answers Question (B_2). Chapter 6 showed how power measurements were used to discover the number of register banks. In Chapter 7 we discovered the number of memory channels using a special microbenchmark. This answers our Question (B_3).

Architectural enhancements also generated more questions:

(C_1) Which GPU components can we change to improve the power consumption?
(C_2) Can enhancements that improve performance, but also increase power consumption still result in gains in energy efficiency?
(C_3) What kind of architectural enhancements will increase the applicability of GPUs for new applications and still improve energy efficiency?

In response to Question (C_1) we developed four different techniques presented in Chapters 8 to 11. The technique described in Chapter 8 improves the power consumption of the memory interface and Chapter 9 targets the DRAM refresh. Chapters 10 and 11 show how we can reorganize the GPU core to reach higher performance and better energy efficiency. Our results from Chapters 6 and 7 indicate that additional power savings might be possible, if we can optimize the warp scheduling to reduce the data dependent power consumption of the ALUs or develop a special data encoding to reduce the power consumption of the memory interface. Our Question (C_2) was answered in Chapter 11. In this chapter we have shown how STSIMT4 together with Scalarization introduces a slight increase in power consumption, but provides an even higher increase in performance. Together this results in an gain in power efficiency. Our power simulator developed in Chapter 5 allowed us to perform this evaluation and showed the EDP improvement of 26.2%. The TSIMT architecture from Chapter 10 and its improvements are also the answer to Question (C_3). These architectures allow GPUs to be used efficiently for executing algorithms that require many divergent branches and thus increase the applicability of GPUs.

12.3 FUTURE WORK

Future GPUs could add additional performance counters or even on-chip power measurement capabilities to enable highly accurate power estimation or measurement without lengthy architectural simulations and allow programmers to easily improve the energy efficiency of their applications.

Even further improvements to energy efficiency of GPUs might be found using techniques such as approximation, compression, data-value aware transforms such as special instruction scheduling or register allocation. And while these techniques could lead to large improvements in performance and efficiency, they all require additional information about the processed data that is not available in common currently available programming interfaces for GPUs. Part of this information might be gained using profiling, other parts likely require annotation by the programmer, e.g.: profiling cannot declare values to be safe to approximate or provide strict upper and lower value bounds, but profiling might be used to gather hard to annotate information about average value distribution, covariances and similar measures useful for value-aware instruction scheduling and compression.

Recently some GPUs have used 3D-stacked memory such as HBM and HBM2 [63], [64], [212]. These memory technologies are expensive but are able to provide a large increase in memory bandwidth and reduce the interface energy. While currently used to replace regular GDDR5/5X DRAM without changing the GPU core architecture, these new memory technologies will likely influence core design and programming models in the future. Due to Little's law and the latency of the memory interface, higher bandwidth at a similar latency means that even more requests need to be in-flight at a time, or the high memory bandwidth cannot be used. With the current execution model, this would require even more warps active at the same and thus even larger register files.

We expect that techniques such as decoupled access/execute [213], [214] and dynamic management of the register file will be implemented in future GPUs and allow more in-flight memory transactions without requiring even larger register files [215]. GPUs aimed at HPC and AI application might also offer DMA capabilities to shared memory or locked cache blocks to trigger large memory transactions from a single warp and reduce the number of concurrent warps required to fully utilize the memory throughput.

The idea of Scalarization can also be extended to not just cover instructions within the same warp but efficiency can be improved further by also scalarizing instructions that are identical in the whole thread block or the current grid. Currently Scalarization removes redundant computations in instructions where all active threads in the warp operate on exactly the same input values. In these cases the instruction of only one of the threads needs to be executed because it is representative for the whole warp. This concept could be extended to cover cases where a few threads can represent the whole warp, e.g.: instructions that perform different computations in odd and even threads, but all odd threads perform exactly the same calculation and all even threads perform the same calculation. In this example, an extension to Scalarization could allow the GPU to only perform two calculations per Warp, instead of one calculation per active thread in the warp.

The current programming model that creates a one, two, or three-dimensional array of threads at kernel launch without allowing dependencies between thread blocks is useful for applications such as linear algebra or image processing but can be problematic for more irregular applications such as video decoding or graph processing. Here we also see room for future improvement. Relatively simple dependency patterns exist between different image parts during video decoding and a programmable thread block scheduler could wait with starting new threads blocks until other blocks that calculate dependencies are finished.

While recent versions of CUDA and OpenCL support kernel launches from within another kernel, this feature is currently often not beneficial due to the high overhead of the current implementations [216], [217]. Especially CUDA dynamic parallelism can require the GPU to swap out the current execution context to DRAM to free resources to allow the kernel launch. More limited APIs for dynamic creation of warps might improve the situation by adding additional constraints such as local creation on the same GPU core and pre-reserved resources. Wang et al. propose the addition of lightweight thread block launches [218]. This proposal could also be used as an alternative to the programmable thread scheduler mentioned above.

With these and similar architectural changes, the energy efficiency can be improved and GPUs could be used efficiently for an even wider range of computations. Being able to move more computations from energy-hungry Out-Of-Order CPUs to low-power GPUs is likely going to improve the energy-efficiency and performance of the whole system. New applications will become possible, even in energy constrained mobile environments.

BIBLIOGRAPHY

[1] Evans and S. C. Corporation, *Picture System: The interactive, dynamic, 3-D line-drawing system*, 1974.

[2] J. Clark, "Special feature a VLSI geometry processor for graphics," *Computer*, vol. 13, no. 7, Jul. 1980.

[3] *NAMCO system 21 hardware*, http://www.system16.com/hardware.php?id=536.

[4] E. Wu and Y. Liu, "Emerging technology about GPGPU," in *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, IEEE, 2008.

[5] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, Mar. 2010.

[6] J. Andrews and N. Baker, "Xbox 360 system architecture," *IEEE micro*, 2006.

[7] V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa, "Shader performance analysis on a modern GPU architecture," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, IEEE, 2005.

[8] D. Luebke and M. Harris, "General-purpose computation on graphics hardware," in *Workshop, SIGGRAPH*, 2004.

[9] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," in *ACM Transactions on Graphics (TOG)*, ACM, 2004.

[10] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, 2012.

[11] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science & Engineering*, 2011.

[12] T. Scogland, B. Subramaniam, and W.-c. Feng, "The Green500 list: Escapades to exascale," *Computer Science-Research and Development*, 2013.

[13] L. N. Huynh, R. K. Balan, and Y. Lee, "Deepsense: A GPU-based deep convolutional neural network framework on commodity mobile devices," in *Proceedings of the 2016 Workshop on Wearable Systems and Applications*, ACM, 2016.

[14] M. Alzantot, Y. Wang, Z. Ren, and M. B. Srivastava, "RSTensorFlow: GPU enabled TensorFlow for deep learning on commodity Android devices," in *Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications*, ACM, 2017.

[15] K.-T. Cheng and Y.-C. Wang, "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, IEEE, 2011.

[16] N. Singhal, J. W. Yoo, H. Y. Choi, and I. K. Park, "Implementation and optimization of image processing algorithms on embedded GPU," *IEICE TRANSACTIONS on Information and Systems*, 2012.

[17] F. Tschorsch and B. Scheuermann, "Bitcoin and beyond: A technical survey on decentralized digital currencies," *IEEE Communications Surveys & Tutorials*, 2016.

[18] *Ethash design rationale*, https://github.com/ethereum/wiki/wiki/Ethash-Design-Rationale.

[19] Noel Randewich, *AMD rallies as cryptocurrency miners snap up graphics chips*, https://www.reuters.com/article/amd-stocks/amd-rallies-as-cryptocurrency-miners-snap-up-graphics-chips-idUSL1N1J3179, 2017.

[20] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, ACM, 2009.

[21] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, "Theano: A CPU and GPU math compiler in Python," in *Proc. 9th Python in Science Conf*, 2010.

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[23] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, 2011.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, ACM, 2014.

[25] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with COTS HPC systems," in *International Conference on Machine Learning*, 2013.

[26] Amazon Web Services, *Elastic GPUs*, https://aws.amazon.com/ec2/elastic-gpus/.

[27] Google Cloud Platform, *Graphics processing units (GPU)*, https://cloud.google.com/gpu/.

[28] Microsoft Azure, *Azure N-Series*, http://gpu.azure.com/.

[29] AMD, *Radeon Instinct MI25*, https://instinct.radeon.com/en/product/mi/radeon-instinct-mi25/.

[30] N. Brunie, "Modified fused multiply and add for exact low precision product accumulation," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2017.

[31] S. P. Gurrum, D. R. Edwards, T. Marchand-Golder, J. Akiyama, S. Yokoya, J.-F. Drouard, and F. Dahan, "Generic thermal analysis for phone and tablet systems," in *2012 IEEE 62nd Electronic Components and Technology Conference*, IEEE, 2012.

[32] J. Leng, Y. Zu, M. Rhu, M. Gupta, and V. J. Reddi, "GPUVolt: Modeling and characterizing voltage noise in gpu architectures," in *Proceedings of the 2014 international symposium on Low power electronics and design*, ACM, 2014.

[33] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, IEEE, 2010.

[34] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE micro*, 2008.

[35] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, ACM, 1970.

[36] AMD, *AMD GCN3 ISA architecture manual*, https://gpuopen.com/compute-product/amd-gcn3-isa-architecture-manual/.

[37] W. W. L. Fung and T. Aamodt, *GPGPU-sim 3.x manual*, http://gpgpu-sim.org/manual/index.php/Main_Page, 2012.

[38] S. Pai, *How the Fermi thread block scheduler works*, http://cs.rochester.edu/~sree/fermi-tbs/fermi-tbs.html.

[39] J. Lucas and B. Juurlink, "ALUPower: Data Dependent Power Consumption in GPUs," in *IEEE Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, ©2016 IEEE, 2016.

[40] M. Sadrosadati, A. Mirhosseini, S. B. Ehsani, H. Sarbazi-Azad, M. Drumond, B. Falsafi, R. Ausavarungnirun, and O. Mutlu, “Ltrf: Enabling high-capacity register files for GPUs via hardware/software cooperative register prefetching,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2018.

[41] V. Volkov, “Better performance at lower occupancy,” in *Proceedings of the GPU technology conference*, 2010.

[42] A. Li, S. L. Song, A. Kumar, E. Z. Zhang, D. Chavarria-Miranda, and H. Corporaal, “Critical points based register-concurrency autotuning for GPUs,” in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, EDA Consortium, 2016.

[43] B. W. Coon, J. E. Lindholm, G. Tarolli, S. D. Tzvetkov, J. R. Nickolls, and M. Y. Siu, “Register file allocation,” Patent US 7634621 B1, 2006.

[44] S. Y. Wing-Kei, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Suh, “SRAM-DRAM hybrid memory with applications to efficient register files in fine-grained multi-threading,” in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, IEEE, 2011.

[45] X. Liu, M. Mao, X. Bi, H. Li, and Y. Chen, “An efficient STT-RAM-based register file in GPU architectures,” in *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, IEEE, 2015.

[46] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. H. Li, “Exploration of GPGPU register file architecture using domain-wall-shift-write based racetrack memory,” in *Proceedings of the 51st Annual Design Automation Conference*, ACM, 2014.

[47] M. Gebhart, S. W. Keckler, and W. J. Dally, “A compile-time managed multi-level register file hierarchy,” in *Proceedings of the 44th annual IEEE/ACM international symposium on microarchitecture*, ACM, 2011.

[48] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, “Unifying primary cache, scratch, and register file memories in a throughput processor,” in *Proc. 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*, IEEE Computer Society, 2012.

[49] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking,” *ArXiv e-prints*, 2018. arXiv: 1804.06826 `[cs.DC]`.

[50] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, “Warped-compression: Enabling power efficient GPUs through register compression,” in *ACM SIGARCH Computer Architecture News*, ACM, 2015.

[51] J. D. Little and S. C. Graves, "Little's law," in *Building intuition*, Springer, 2008.

[52] V. Volkov, *Understanding Latency Hiding on GPUs*. University of California, Berkeley, 2016.

[53] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2012.

[54] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *ACM SIGARCH Computer Architecture News*, ACM, 2013.

[55] S.-Y. Lee and C.-J. Wu, "Caws: Criticality-aware warp scheduling for gpgpu workloads," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, ACM, 2014.

[56] Q. Xu and M. Annavaram, "Pats: Pattern aware scheduling and power gating for gpgpus," in *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, IEEE, 2014.

[57] NVIDIA, *Kepler GK110 white paper*, https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[58] JEDEC Standard, "Graphics Double Data Rate (GDDR5) SGRAM standard," *JESD212C, February*, 2016.

[59] ——, "Graphics Double Data Rate (GDDR5X) SGRAM standard," *JESD232A, August*, 2016.

[60] ——, "DDR4 SDRAM standard," *JESD79-4B, June*, 2017.

[61] H. W. Johnson, M. Graham, *et al.*, *High-speed digital design: a handbook of black magic*. Prentice Hall Upper Saddle River, NJ, 1993, vol. 1.

[62] M. O'Connor, "Highlights of the high-bandwidth memory (HBM) standard," in *Memory Forum Workshop*, 2014.

[63] J. Macri, "Amd's next generation GPU and high bandwidth memory architecture: FURY," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*, IEEE, 2015.

[64] *NVIDIA Tesla V100 datasheet*, https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf.

[65] I. Antochi, B. Juurlink, S. Vassiliadis, and P. Liuha, "Memory bandwidth requirements of tile-based rendering," in *International Workshop on Embedded Computer Systems*, Springer, 2004.

[66] A. C. Beers, M. Agrawala, and N. Chaddha, "Rendering from compressed textures," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, 1996.

[67] J. Ström and T. Akenine-Möller, "i PACKMAN: High-quality, low-complexity texture compression for mobile phones," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, 2005.

[68] S. Morein, *ATI Radeon HyperZ technology*, http://www.graphicshardware.org/previous/www_2000/presentations/ATIHot3D.pdf, 2000.

[69] N. Greene, M. Kass, and G. Miller, "Hierarchical z-buffer visibility," in *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '93, 1993.

[70] S. Lal, J. Lucas, and B. Juurlink, "E^2mc: Entropy encoding based memory compression for GPUs," in *Proc. 31st IEEE Int. Parallel and Distributed Processing Symposium (IPDPS)*, May 2017.

[71] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in gpus: Enabling flexible data compression with assist warps," in *ACM SIGARCH Computer Architecture News*, ACM, 2015.

[72] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for GPU systems," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, IEEE, 2016.

[73] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing DMA engine: Leveraging activation sparsity for training deep neural networks," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, IEEE, 2018.

[74] D. B. Kirk and W.-M. W. Hwu, *Programming massively parallel processors: A hands-on approach*. Morgan Kaufmann, 2016.

[75] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, 5th ed. Elsevier, 2012, ch. 4.

[76] N. Leischner, V. Osipov, and P. Sanders, *Fermi architecture white paper*, http://www.nvidia.de/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[77] *NVIDIA GeForce GTX 750 Ti whitepaper*, http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.

[78] AMD, *AMD Graphics Core Next GCN architecture white paper*, 2012.

[79] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA, 2010.

[80] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures," in *Parallel Processing (ICPP), 2012 41st International Conference on*, IEEE, 2012.

[81] Y. Wang and N. Ranganathan, "An instruction-level energy estimation and optimization methodology for GPU," in *Proceedings of the IEEE 11th International Conference on Computer and Information Technology*, ser. CIT '11, 2011.

[82] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical Power Modeling of GPU Kernels Using Performance Counters," in *Proceedings of the International Green Computing Conference*, 2010.

[83] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for GPU-based computing," in *Proceedings of the Workshop on Power Aware Computing and Systems, HotPower*, 2009.

[84] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7, 2014.

[85] K. Ramani, A. Ibrahim, and D. Shimizu, "PowerRed: A flexible power modeling framework for power efficiency exploration in GPUs," in *Workshop on General Purpose Processing on Graphics Processing Units, GPGPU*, 2007.

[86] G. Wang, "Power analysis and optimizations for GPU architecture using a power simulator," in *Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering, ICACTE*, 2010.

[87] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2009.

[88] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *Proc. 40th Annual Int. Symp. on Computer Architecture (ISCA)*, 2013.

[89] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "Architectural simulators considered harmful," *IEEE Micro*, 2015.

[90] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for GPU architectures using McPAT," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2014.

[91] T. Diop, N. E. Jerger, and J. Anderson, "Power modeling for heterogeneous processors," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ACM, 2014.

[92] P. Libuschewski, D. Kaulbars, B. Dusza, D. Siedhoff, F. Weichert, H. Müller, C. Wietfeld, and P. Marwedel, "Multi-objective computation offloading for mobile biosensors via LTE," in *Wireless Mobile Communication and Healthcare (Mobihealth), 2014 EAI 4th International Conference on*, IEEE, 2014.

[93] P. Libuschewski, P. Marwedel, D. Siedhoff, and H. Müller, "Multi-objective, energy-aware GPGPU design space exploration for medical or industrial applications," in *Signal-Image Technology and Internet-Based Systems (SITIS), 2014 Tenth International Conference on*, IEEE, 2014.

[94] A. Sankaranarayanan, E. K. Ardestani, J. L. Briz, and J. Renau, "An energy efficient GPGPU memory hierarchy with tiny incoherent caches," in *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, IEEE, 2013.

[95] R. Nath and D. Tullsen, "The CRISP performance model for dynamic voltage and frequency scaling in a GPGPU," in *Proceedings of the 48th International Symposium on Microarchitecture*, ACM, 2015.

[96] A. Dhar and D. Chen, "Efficient GPGPU computing with cross-core resource sharing and core reconfiguration," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, IEEE, 2017.

[97] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills GPUSimPow: A GPGPU power simulator," in *Proc. IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, ©2013 IEEE, 2013.

[98] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. Int. Symp. on Computer Architecture, ISCA*, ACM, 2000.

[99] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald, "Challenges for Architectural Level Power Modeling," in *Power aware computing*, 2002.

[100] V. Adhinarayanan, I. Paul, J. L. Greathouse, W. Huang, A. Pattnaik, and W.-c. Feng, "Measuring and modeling on-chip interconnect power on real hardware," in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, IEEE, 2016.

[101] D. Sarta, D. Trifone, and G. Ascia, "A Data Dependent Approach to Instruction Level Power Estimation," in *Low-Power Design, 1999. Proc. Alessandro Volta Memorial Workshop on*, IEEE, Mar. 1999.

[102] S. Kerrison and K. Eder, "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor," *ACM Trans. Embed. Comput. Syst.*, 2015.

[103] T. M. Hollis, "Data bus inversion in high-speed memory applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2009.

[104] N. Chang, K. Kim, and J. Cho, "Bus encoding for low-power high-performance memory systems," in *Design Automation Conference (DAC)*, ACM, 2000.

[105] T. M. Hollis, *Devices and methods for facilitating data inversion to limit both instantaneous current and signal transitions*, US Patent 9,270,417, 2016.

[106] J. D. Ihm, S. J. Bae, K. I. Park, H. Y. Song, W. J. Lee, H. J. Kim, K. H. Kim, *et al.*, "An 80nm 4Gb/s/pin 32b 512Mb GDDR4 graphics DRAM with low-power and low-noise data-bus inversion," in *IEEE ISSCC Digest of Technical Papers*, 2007.

[107] M. R. Stan and W. P. Burleson, "Bus-invert coding for low-power I/O," *IEEE Transactions on VLSI systems*, 1995.

[108] U. Narayanan, K.-S. Chung, and T. Kim, "Enhanced bus invert encodings for low-power," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2002.

[109] J.-H. Kim, W. Kim, D. Oh, R. Schmitt, J. Feng, C. Yuan, L. Luo, and J. Wilson, "Performance impact of simultaneous switching output noise on graphic memory systems," in *Electrical Performance of Electronic Packaging*, IEEE, 2007.

[110] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-Aware intelligent DRAM refresh," in *Proc. of the International Symposium on Computer Architecture, ISCA*, 2012.

[111] K. Kim and J. Lee, "A new investigation of data retention time in truly nanoscaled DRAMs," *IEEE Electron Device Letters*, 2009.

[112] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *Proc. of the Conference on Programming Language Design and Implementation, PLDI*, 2011.

[113] F. Ware and C. Hampel, "Improving Power and Data Efficiency with Threaded Memory Modules," in *Proceedings of the International Conference on Computer Design, ICCD*, 2006.

[114] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate Storage in Solid-State Memories," in *Proc. of the International Symposium on Microarchitecture, MICRO*, 2013.

[115] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan, "Quality-aware data allocation in approximate DRAM," in *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, IEEE Press, 2015.

[116] A. Ranjan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Approximate storage for energy efficient spintronic memories," in *Proceedings of the 52nd Annual Design Automation Conference*, ACM, 2015.

[117] M. Jung, É. Zulian, D. M. Mathew, M. Herrmann, C. Brugger, C. Weis, and N. Wehn, "Omitting refresh: A case study for commodity and Wide I/O DRAMs," in *Proceedings of the 2015 International Symposium on Memory Systems*, ACM, 2015.

[118] Y. Chen, X. Yang, F. Qiao, J. Han, Q. Wei, and H. Yang, "A multi-accuracy-level approximate memory architecture based on data significance analysis," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, IEEE, 2016, pp. 385–390.

[119] R. M. Krashinsky, *Temporal SIMT execution optimization*, Patent No. US 2013/0042090 A1, Filed August 2011, Issued Februar 2013.

[120] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, 2011.

[121] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. 40th Annual IEEE/ACM Int. Symp. on Microarchitecture, MICRO*, 2007.

[122] W. W. L. Fung and T. Aamodt, "Thread block compaction for efficient SIMT control flow," in *Proc. 17th Int. Symp. on High Performance Computer Architecture, HPCA*, 2011.

[123] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proc. 44th Annual IEEE/ACM Int. Symp. on Microarchitecture, MICRO*, 2011.

[124] N. Brunie, S. Collange, and G. Diamos, "Simultaneous branch and warp interweaving for sustained GPU performance," in *Proc. 39th Int. Symp. on Computer Architecture, ISCA*, 2012.

[125] N. Jayasena, M. Erez, J. H. Ahn, and W. J. Dally, "Stream Register Files with Indexed Access," in *Proc. 10th Int. Symp. on High Performance Computer Architecture, HPCA*, 2004.

[126] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi, "SIMD divergence optimization through intra-warp compaction," in *Proc. 40th Annual Int. Symp. on Computer Architecture, ISCA*, 2013.

[127] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the Tradeoffs Between Programmability and Efficiency in Data-parallel Accelerators," in *Proc. 38th Annual Int. Symp. on Computer Architecture, ISCA*, 2011.

[128] J. E. Smith, G. Faanes, and R. Sugumar, "Vector Instruction Set Support for Conditional Operations," in *Proc. 27th Annual Int. Symp. on Computer Architecture, ISCA*, 2000.

[129] Y. Lee, R. Krashinsky, V. Grover, S. Keckler, and K. Asanovic, "Convergence and Scalarization for Data-Parallel Architectures," in *Proc. Int. Symp. on Code Generation and Optimization (CGO)*, 2013.

[130] S. Collange, "Identifying scalar behavior in CUDA kernels," 2011.

[131] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proc. 19th Int. Conference on Parallel Architectures and Compilation Techniques, PACT*, 2010.

[132] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. Meira, "Divergence analysis and optimizations," in *Proc. Int. Conference on Parallel Architectures and Compilation Techniques (PACT' 11)*, IEEE, 2011.

[133] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou, "Exploiting uniform vector instructions for GPGPU performance, energy efficiency, and opportunistic reliability enhancement," in *Proc. 27th Int. ACM Conference on Int. Conference on Supercomputing*, 2013.

[134] J. Kim, C. Torng, S. Srinath, D. Lockhart, and C. Batten, "Microarchitectural Mechanisms to Exploit Value Structure in SIMT Architectures," in *Proc. 40th Annual Int. Symp. on Computer Architecture*, 2013.

[135] T. Regan, J. Munson, G. Zimmer, and M. Stokowski, "Current sense circuit collection," *Linear Technology Application Note 105*, 2005.

[136] P. A. Bode, *AN39 Current measurement applications handbook*, https://www.diodes.com/assets/App-Note-Files/an39.pdf, 2008.

[137] Analog Devices, *AD8210 data sheet*, http://www.analog.com/media/en/technical-documentation/data-sheets/AD8210.pdf, 2006.

[138] ——, *AD8218 data sheet*, http://www.analog.com/media/en/technical-documentation/data-sheets/AD8218.pdf, 2013.

[139] National Instruments, *USB-6210 specifications*, http://www.ni.com/pdf/manuals/375194d.pdf, 2013.

[140] ——, *NI-DAQmx base software*, http://sine.ni.com/nips/cds/view/p/lang/en/nid/14480.

[141] Adex, *PEXP16-EX-CSR*, http://www.adexelec.com/pciexp.htm.

[142] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," *University of Virginia Dept of Computer Science Tech Report CS-2003*, vol. 5, 2003.

[143] STMicroelectronics, *STM32F103xB data sheet*, http://www.st.com/resource/en/datasheet/cd00161566.pdf, 2015.

[144] Microchip Technology, *MCP3912 data sheet*, http://ww1.microchip.com/downloads/en/DeviceDoc/20005348A.pdf, 2014.

[145] *libusb - A cross-platform user-mode library, for generic access to USB devices*, https://github.com/libusb/libusb/wiki.

[146] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies," in *Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA*, 2008.

[147] NVIDIA, *CUDA: Compute unified device architecture*, http://developer.nvidia.com/object/gpucomputing.html, 2007.

[148] Khronos Group, *OpenCL - The open standard for parallel programming of heterogeneous systems*, http://www.khronos.org/opencl/, 2009.

[149] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. Int. Symp. on Performance Analysis of Systems and Software, ISPASS*, 2009.

[150] C. Kun, S. Quan, and A. Mason, "A Power-Optimized 64-Bit Priority Encoder Utilizing Parallel Priority Look-Ahead," in *Proceedings of the 2004 International Symposium on Circuits and Systems, ISCAS*, 2004.

[151] B. W. Coon and J. E. Lindholm, "System and Method for Managing Divergent Threads Using Synchronization Tokens and Program Instructions that Include Set-Synchronization Bits," Patent US 7543136 B1, 2009.

[152] B. W. Coon, P. C. Mills, S. F. Oberman, and M. Y. Sui, "Tracking Register Usage During Multithreaded Processing Using a Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators," Patent US 7434032 B1, 2008.

[153] J. E. Lindholm, M. Y. Siu, S. S. Moy, L. S., and J. Nickolls, "Simulating Multiported Memories Using Lower Port Count Memories," Patent US 7339592, 2008.

[154] S. Galal and M. Horowitz, "Energy-Efficient Floating-Point Unit Design," *IEEE Transactions on Computers*, 2011.

[155] D. De Caro, N. Petra, and A. Strollo, "A High Performance Floating-Point Special Function Unit Using Constrained Piecewise Quadratic Approximation," in *Proceedings of the IEEE International Symposium on Circuits and Systems. ISCAS*, 2008.

[156] C. Galuzzi, C. Gou, H. Calderón, G. N. Gaydadjiev, and S. Vassiliadis, "High-bandwidth Address Generation Unit," *Journal of Signal Processing Systems*, 2009.

[157] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems Software, ISPASS*, 2010.

[158] L. Nyland, J. Nickolls, G. Hirota, and T. Mandal, "Systems and Methods for Coalescing Memory Accesses of Parallel Threads," Patent US 8086806 B2, 2011.

[159] B. W. Coon, M. Y. Siu, W. Xu, S. F. Oberman, J. R. Nickolls, and P. C. Mills, "Shared Memory with Parallel Access and Access Conflict Resolution Mechanism," Patent US 8108625 B1, 2012.

[160] Micron, *Calculating memory system power for DDR3*, http://www.micron.com/products/dram/ddr3-sdram, 2007.

[161] Hynix, *GDDR5 datasheet*, http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf, 2010.

[162] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. Int. Symp. on Workload Characterization, IISWC*, IEEE, 2009.

[163] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proc. 38th Annual Int. Symp. on Computer Architecture, ISCA*, 2011.

[164] S. Lal, J. Lucas, M. Andersch, M. Alvarez-Mesa, A. Elhossini, and B. Juurlink, "GPGPU workload characteristics and performance analysis," in *Proc. 14th Int. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014.

[165] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski, "The design and implementation of PowerMill," in *Proc. Int. Symp. on Low Power Design, ISPLED*, ACM, 1995.

[166] S. Gupta and F. N. Najm, "Power Macromodeling for High Level Power Estimation," in *Proc. Design Automation Conference, DAC*, ACM, 1997.

[167] F. Klein, G. Araujo, R. Azevedo, R. Leao, and L. C. Dos Santos, "On the Limitations of Power Macromodeling Techniques," in *Proc. Symp. on VLSI, ISVLSI*, IEEE, 2007.

[168] E. Macii, M. Pedram, and F. Somenzi, "High-level Power Modeling, Estimation, and Optimization," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Trans. on*, vol. 17, no. 11, 1998.

[169] H. Yunqing, *Asfermi*, https://github.com/hyqneuron/asfermi.

[170] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, 2008.

[171] K. Fatahalian and M. Houston, "A closer look at GPUs," *Communications of the ACM*, 2008.

[172] JEDEC Standard, "POD15 - 1.5 v pseudo open drain I/O," *JESD8-20A*, 2009.

[173] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, 2011.

[174] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 1979.

[175] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, "Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[176] J. Sell, "The Xbox One X Scorpio engine," *IEEE Micro*, vol. 38, no. 2, 2018.

[177] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, 1996.

[178] J. Lucas, S. Lal, and B. Juurlink, "Optimal DC/AC data bus inversion coding," in *Design, Automation and Test in Europe, DATE*, EDAA, 2018.

[179] N. P. Jouppi, A. B. Kahng, N. Muralimanohar, and V. Srinivas, "CACTI-IO: CACTI with off-chip power-area-timing models," *IEEE Transactions on VLSI Systems*, 2015.

[180] JEDEC Standard, "Graphics Double Data Rate (GDDR4) SGRAM standard," *SDRAM3.11.5.8, May*, 2006.

[181] S. J. Bae, Y. S. Sohn, K. I. Park, K. H. Kim, D. H. Chung, J. G. Kim, S. H. Kim, *et al.*, "A 60nm 6Gb/s/pin GDDR5 graphics DRAM with multifaceted clocking and ISI/SSN-reduction techniques," in *IEEE ISSCC Digest of Technical Papers*, 2008.

[182] N. P. Jouppi, A. B. Kahng, N. Muralimanohar, and V. Srinivas, *CACTI-IO Technical Report*. Department of Computer Science and Engineering, University of California, San Diego, 2012.

[183] A. Amirkhany, J. Wei, N. K. Mishra, J. Shen, W. T. Beyene, C. Chen, T. Chin, D. Dressler, C. Huang, V. P. Gadde, *et al.*, "A 12.8-Gb/s/link tri-modal single-ended memory interface," *IEEE Journal of Solid-State Circuits*, 2012.

[184] H. Vuong, "Mobile memory technology roadmap," in *JEDEC's Mobile Forum*, 2013.

[185] M. Andersch, J. Lucas, M. Álvarez-Mesa, and B. Juurlink, "On latency in GPU throughput microarchitectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[186] J. Lucas, M. Alvarez-Mesa, M. Andersch, and B. Juurlink, "Sparkk: Quality-scalable approximate storage in DRAM," in *The Memory Forum*, 2014.

[187] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. of the Conference on Programming Language Design and Implementation, PLDI*, 2011.

[188] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture Support for Disciplined Approximate Programming," in *Proc. of the international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2012.

[189] A. Rahmati, M. Hicks, D. E. Holcomb, and K. Fu, "Refreshing Thoughts on DRAM: Power Saving vs. Data Integrity," in *Workshop on Approximate Computing Across the System Stack, WACAS*, 2014.

[190] J. Lucas, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "Spatiotemporal SIMT and Scalarization for improving GPU efficiency," *ACM Transactions on Architecture and Code Optimization*, Sep. 2015. DOI: 10.1145/2811402.

[191] G.-J. V. D. Braak and H. Corporaal, "GPU-CC: A reconfigurable GPU architecture with communicating cores," in *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, ACM, 2013, pp. 86–89.

[192] ——, "R-GPU: A reconfigurable GPU architecture," *ACM Trans. Archit. Code Optim.*, 2016.

[193] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, Mar. 2008.

[194] J. R. Diamond, D. S. Fussell, and S. W. Keckler, "Arbitrary modulus indexing," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM Int. Symp. on*, IEEE, 2014.

[195] G. Ziegler, *Analysis-Driven Optimization*, http://www.nvidia.de/content/PDF/isc-2011/Ziegler.pdf, 2011.

[196] J. Lindholm, M. Siu, S. Moy, S. Liu, and J. Nickolls, *Simulating multi-ported memories using lower port count memories*, Patent No. US 7339592 B2, Filed July 2004, Issued March 2008.

[197] P. Xiang, Y. Yang, and H. Zhou, "Warp-Level Divergence in GPUs: Characterization, Impact, and Mitigation," in *Proc. 20th Int. Symp. on High Performance Computer Architecture, HPCA*, 2014.

[198] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware," *ACM Trans. Archit. Code Optim.*, vol. 6, no. 2, Jul. 2009, ISSN: 1544-3566.

[199] Y. He, Y. Pu, R. Kleihorst, Z. Ye, A. A. Abbo, S. M. Londono, and H. Corporaal, "Xetal-Pro: An ultra-low energy and high throughput SIMD processor," in *Proceedings of the 47th Design Automation Conference*, ACM, 2010.

[200] Y. Pu, Y. He, Z. Ye, S. M. Londono, A. A. Abbo, R. Kleihorst, and H. Corporaal, "From Xetal-II to Xetal-Pro: On the road toward an ultralow-energy and high-throughput SIMD processor," *IEEE Transactions on Circuits and Systems for Video Technology*, 2011.

[201] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *ACM SIGPLAN Notices*, ACM, 2011.

[202] B. Wang, M. Alvarez-Mesa, C. C. Chi, and B. Juurlink, "An optimized parallel IDCT on graphics processing units," in *Euro-Par 2012: Parallel Processing Workshops*, ser. Leture Notes in Computer Science, Springer Berlin Heidelberg, 2013.

[203] ——, "Parallel H.264/AVC motion compensation for GPUs using OpenCL," *Cir. and Sys. for Video Technology, IEEE Trans. on*, 2015.

[204] NVIDIA, *NVIDIA GPU computing SDK 3.1*, 2011.

[205] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems," in *Proc. 17th Int. Symp. on High Performance Distributed Computing*, 2008.

[206] R. R. M. Krashinsky, "Vector-thread architecture and implementation," PhD thesis, Massachusetts Institute of Technology, 2007.

[207] S. Kyo, T. Koga, and S. Okazaki, "IMAP-CE: A 51.2 GOPS video rate image processor with 128 VLIW processing elements," in *Proceedings 2001 International Conference on Image Processing*, 2001.

[208] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "Soda: A low-power architecture for software radio," *ACM SIGARCH Computer Architecture News*, 2006.

[209] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "AnySP: Anytime anywhere anyway signal processing," in *ACM SIGARCH Computer Architecture News*, 2009.

[210] S. S. Muchnick, "Advanced compiler design and implementation," in. 1997, ISBN: 1-55860-320-4.

[211] A. Lumsdaine and D. Gregor, *Boost graph library: Sequential vertex coloring*, http://www.boost.org/doc/libs/1_57_0/libs/graph/doc/sequential_vertex_coloring.html, 2004.

[212] *NVIDIA Tesla P100 datasheet*, https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf.

[213] J. E. Smith, "Decoupled access/execute computer architectures," in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society Press, 1982.

[214] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*, IEEE, 2016.

[215] H. Jeon, G. S. Ravi, N. S. Kim, and M. Annavaram, "GPU register file virtualization," in *Proceedings of the 48th International Symposium on Microarchitecture*, ACM, 2015.

[216] J. DiMarco and M. Taufer, "Performance impact of dynamic parallelism on different clustering algorithms," in *Modeling and Simulation for Defense Systems and Applications VIII*, International Society for Optics and Photonics, vol. 8752, 2013.

[217] J. Wang and S. Yalamanchili, "Characterization and analysis of dynamic parallelism in unstructured GPU applications," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, IEEE, 2014.

[218] J. Wang, N. Rubin, A. Sidelnik, and S. Yalamanchili, "Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on GPUs," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, 2016.