Power Modeling and Architectural Techniques for Energy-Efficient GPUs

vorgelegt von
M.Sc.
Sohan Lal
ORCID: 0000-0002-2325-1705

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

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Sabine Glesner
Gutachter:	Prof. Dr. Ben Juurlink
Gutachter:	Prof. Dr. H.A.G. Wijshoff
Gutachter:	Prof. Dr. Akash Kumar

Tag der wissenschaftlichen Aussprache: 1. August 2019

Berlin 2019

Abstract

Graphics Processing Units (GPUs) have evolved from fixed function graphics processors to programmable general-purpose compute accelerators in a short time. The high theoretical performance and energy efficiency of GPUs compared to CPUs have made them indispensable for mainstream computing. However, their high power consumption and limited energy efficiency under low utilization is a challenge that still needs to be tackled.

This thesis investigates bottlenecks that cause low performance and low energy efficiency in GPUs and proposes architectural techniques to address them. To conduct energy efficiency research for GPUs, we first develop a flexible and accurate *power simulator* called GPUSimPow. We use a hybrid approach for power modeling that improves flexibility and accuracy compared to previous approaches. Our evaluation shows an average relative error of 11.7% and 10.8% between simulated and measured power for the NVIDIA GT240 and GTX580, respectively. We then use GPUSimPow to study the energy efficiency of a wide range of kernels and categorize them into *high performance* and *low performance*. We further investigate the bottlenecks of low-performance kernels by analyzing their occupancy. We quantify the gain in performance and energy efficiency when occupancy is increased. For instance, the average increase in instructions per cycle, the average reduction in energy consumption and energy-delay-product is 11%, 9%, and 23%, respectively, when occupancy is increased for a sub-category of low occupancy kernels. The full occupancy kernels have low performance despite having the maximum number of threads. Further investigation shows that several of these kernels are memory-bound and can gain significantly from an increase in memory bandwidth.

The traditional ways of increasing memory bandwidth by widening interfaces and increasing frequency have issues such as high power consumption, large form factor, and difficulty in the scaling of pin count. Memory compression is a promising alternative to increase the effective memory bandwidth of GPUs, however, we find that the existing memory compression techniques for GPUs exploit simple patterns for compression and trade low compression ratios for low decompression latency. Based on the evidence that GPUs are less sensitive to latency than CPUs, we propose the more complex *Entropy Encoding Based Memory Compression* (E^2MC) technique for GPUs. On average, E^2MC delivers 53% higher compression ratio and 8% higher speedup than the state of the art. E^2MC reduces energy consumption and energy-delay-product by 13% and 27%, respectively. While designing E^2MC, we observe that lossless memory compression techniques including E^2MC often have a low effective compression ratio due to the large memory access granularity (MAG) exhibited by GPUs. Our study of the distribution of compressed blocks reveals that a significant percentage of compressed blocks have only a few

bytes above a multiple of MAG but a whole burst is fetched from memory. With the goal of reducing the compressed size by these extra bytes, we propose the novel *MAG Aware Selective Lossy Compression* (SLC) technique for GPUs which increases the effective compression ratio and drives the performance and energy efficiency further. For a lossy threshold of 16B and 32B MAG, SLC provides a speedup of up to 17% on top of E^2MC with a maximum error of 0.64%. SLC reduces energy consumption and energy-delay-product by 8.4% and 18.2%, respectively.

Zusammenfassung

Grafikprozessoren (graphics processing units, GPUs) haben sich innerhalb relativ kurzer Zeit von auf Ausgabe spezialisierten Einheiten zu programmierbaren Universalbeschleunigern entwickelt. Durch die hohe theoretische Leistung und Energieeffizienz von GPUs im Gegensatz zu CPUs wurden GPUs unverzichtbar für heutige Rechenzentren. Allerdings ist die hohe Leistungsaufnahme und schlechte Energieeffizienz unter geringer Auslastung eine Herausforderung, die es noch zu meistern gilt.

Diese Arbeit untersucht die Engpässe, die für eine geringe Leistung und für die geringe Energieeffizienz verantwortlich sind und schlägt Lösungen vor, die auf einer verbesserten Hardwarearchitektur basieren. Um die Energieeffizienz von GPUs zu untersuchen, entwickeln wir zuerst einen flexiblen und genauen *elektrischen Leistungssimulator* GPUSimPow. Wir verwenden einen hybriden Ansatz für die Modellierung der Leistung, der die Flexibilität und Genauigkeit im Vergleich zu bisherigen Ansätzen verbessert. Unsere Untersuchung zeigt einen durchschnittlichen relativen Fehler zwischen 11,7 % und 10,8 % zwischen der simulierten und der gemessenen Leistungsaufnahme einer NVIDIA GT240 und einer GTX580. Wir benutzen den Simulator um die Energieeffizienz einer Vielzahl von Kerneln zu untersuchen. Dabei kategorisieren wir die Kernel in hoch-performante Kernel und Kernel mit geringer Leistung. Außerdem untersuchen wir die Engpässe der Kernel mit geringer Leistung und teilen sie in die Gruppen *geringe Auslastung* und *volle Auslastung*. Wir quantifizieren den Zugewinn an Rechenleistung und Energieeffizienz bei einer Erhöhung der Auslastung. Beispielsweise ist die durchschnittliche Erhöhung der Zyklen pro Instruktion, die durchschnittliche Verringerung der Leistungsaufnahme und das Energy-delay-product bei voller Auslastung jeweils 11 %, 9 % und 23 %, wenn die Auslastung für die Gruppe der Kernel mit geringer Auslastung erhöht wird. Die Kernel mit hoher Auslastung haben eine geringe Leistung, obwohl sie die maximale Anzahl von Threads ausführen. Weitere Untersuchen zeigen, dass einige dieser Kernel durch die Speicherbandbreite begrenzt werden (memory-bound) und durch eine Erhöhung der Speicherbandbreite signifikant beschleunigt werden können.

Traditionelle Techniken zur Vergrößerung der Speicherbandbreite etwa durch breitere Schnittstellen und höhere Taktraten haben Nachteile wie zum Beispiel hohe Leistungsaufnahme, große Bauweise und Probleme bei der Erhöhung der Anzahl der elektrischen Kontakte. Speicherkompression ist ein vielversprechender Ansatz die effektive Speicherbandbreite zu erhöhen. Allerdings stellen wir fest, dass bestehende Speicherkompressionstechniken für GPUs einfache Muster zur Kompression ausnutzen und bei der Dekompression niedrigere Kompressionsraten zum Erreichen geringer Verzögerungen in Kauf nehmen. Basierend auf der Tatsache, dass GPUs weniger als CPUs von einer höheren La-

tenz beeinflusst werden, schlagen wir das verbesserte Verfahren *Entropy Encoding Based Memory Compression* (E^2MC) für GPUs vor. Dieses Verfahren erreicht durchschnittlich eine 53 % höhere Kompressionsrate und 8 % höhere Leistung als die modernsten Techniken. E^2MC reduziert die Energieaufnahme um 13 % und das Energy-delay-product um 27 %. Allerdings zeigt sich, dass die verlustfreie Kompression in vielen Fällen zu einer geringen effektiven Kompressionsrate führt, da die Granularität der Speicherzugriffe (Memory Access Granularity, MAG) auf GPUs groß ist. Unsere Untersuchung der Verteilung der komprimierten Blöcke zeigt, dass ein signifikanter Anteil der komprimierten Blöcke eine Größe haben, sodass ein paar Bytes mehr als die MAG geladen werden müssen und dadurch trotzdem eine Speichertransaktion ausgeführt wird. Mit dem Ziel die Größe der komprimierten Blocks um diese extra Bytes zu reduzieren, schlagen wir unseren neuartigen Ansatz *MAG Aware Selective Lossy Compression* (SLC) vor. Damit erhöhen wir die effektive Kompressionsrate und verbessern damit die Leistung und Energieeffizienz weiter. Für eine Schwelle von 16 B und 32 B MAG kann mit SLC eine Beschleunigung von 17 % im Vergleich zu modernsten verlustfreien Ansätzen erreicht werden, während der Fehler nur 0,64 % beträgt. SLC reduziert die Energieaufnahme um 8,4 % und das Energy-delay-product um 18,2 %.

Contents

Acronyms

AFBC	ARM Frame Buffer Compression
AGU	Address Generation Unit
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BDI	Base Delta Immediate
BL	Burst Length
BPC	Bit-plane Compression
BW	Bandwidth
C-PACK	Compact Cache
CE	Coalescing Efficiency
CNN	Convolution Neural Network
CPU	Central Processing Units
CR	Compression Ratio
CTA	Cooperative Thread Arrays
CUDA	Compute Unified Device Architecture
DDR	Double Data Rate
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
E^2MC	Entropy Encoding Based Memory Compression
ECC	Error Checking and Correction
EDP	Energy Delay Product
EI	Energy Per Instruction
FP	Floating Point
FPC	Frequent Pattern Compression
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit

FWC	Fixed Width Coding
GB/s	Giga Bytes Per Second
GDDR	Graphics Double Data Rate
GFLOP/J	Giga Floating Point Instructions Per Joule
GFLOP/W	Giga Floating Point Instructions Per Watt
GFLOPS	Giga Floating Point Instructions Per Second
GPGPU	General Purpose Computation on Graphics Processing Units
GPU	Graphics Processing Units
HBM	High Bandwidth Memory
HMC	Hybrid Memory Cube
HP	High Performance
HPC	High Performance Computing
HyComp	Hybrid Compression
INT	Integer
IPC	Instructions Per Cycle
LCP	Linearly Compressed Pages
LLVM	Low Level Virtual Machine
LP	Low Performance
LSTU	Load Store Unit
LZ	Lempel-Ziv
MAG	Memory Access Granularity
MC	Memory Controller
MXT	Memory Expansion Technology
NoC	Network on Chip
OpenCL	Open Computing Language
PCIe	Peripheral Component Interconnect Express
PDW	Parallel Decoding Ways
PTX	Parallel Thread Execution
QSLC	Quantization-based Selective Lossy Compression
RF	Register File
RLE	Run Length Encoding
RTL	Register Transfer Language

SC^2	Statistical Cache Compression
SFU	Special Function Unit
SGRAM	Synchronous Graphics Random Access Memory
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
SLC	Selective Lossy Compression
SM	Streaming Multiprocessor
SP	Single Precision
SPMD	Single Program Multiple Data
SU	SIMD Utilization
TLB	Translation Look-aside Buffer
TOQ	Target Output Quality
TPC	Thread Processing Cluster
TSLC	Tree-based Selective Lossy Compression
VWC	Variable Width Coding
WCU	Warp Control Unit
WST	Warp Status Table

List of Tables

List of Figures

Listings

1 Introduction

Since the release of the first microprocessor by Intel in 1971 until early 2000, performance has been the main metric that has driven microprocessors design [19, 117]. However, since early 2000, energy efficiency has gained significance as the main metric for microprocessors design [48, 117]. The change of focus from performance to energy efficiency is related to Moore's law [111] and the end of Dennard Scaling [43, 117]. Dennard [39] observed that as transistor size reduces with the reduction in process node, the power density of a chip remains constant because both voltage and current scale downwards with transistor size. In other words, the power usage of a chip stays in proportion to the chip area. During the first 30 years of microprocessors design, total chip power has stayed constant for a given area, and at the same time, the number of transistors doubled and frequency increased by 40% almost every two years. However, with process nodes below 65 nm, these rules are no longer applicable due to exponential growth in leakage current, and it is commonly recognized that Dennard Scaling broke between 2005-2007 [114, 18]. With the end of Dennard Scaling, it is no longer possible to increase the performance of a chip by simply increasing the frequency and simultaneously keep the power envelope constant [114].

To continue the gain of performance improvement, microprocessors design has shifted from single core to homogeneous multi-core and of late even from homogeneous to heterogeneous. In recent years, heterogeneous computing has been adopted as a way to achieve higher absolute performance and better performance per watt [109, 28]. This is achieved by mapping different parts of an application to the best-suited accelerator for the task. Hardware accelerators such as Graphics Processing Units (GPUs), Field-programmable Gate Arrays (FPGAs), many-core CPUs such as Xeon Phi are used as co-processors resulting in heterogeneous architectures. These accelerators have specialized processing capabilities to handle various processing tasks.

Among these hardware accelerators, GPUs have become an indispensable component of mainstream computing. GPUs are being used in almost all forms of computing from mobile phones to supercomputers, thanks to their tremendous computing power which is increasing with every generation. GPUs are massively multithreaded processors, designed for high throughput computing. They have a large number of cores, capable of performing a large number of floating point operations per second. However, a GPU core is much simpler than a modern superscalar CPU core which has complex logic to support branch prediction, out-of-order and speculative execution [80]. GPUs use a single instruction multiple thread (SIMT) execution model to execute a group of threads concurrently. Each thread executes the same instruction but operates on different data. Therefore, SIMT execution model simplifies the front-end of a GPU by sharing the instruction fetch, decode

and issue logic among several cores. As GPUs have simpler front-end and control logic, a large portion of chip area is dedicated to many cores, enabling much higher throughput than CPUs. Because of their massive computational power, GPUs are being harnessed to accelerate compute-intensive applications from various domains such as scientific, social media, and finance [98, 35, 149, 64, 60, 94, 164].

The high-performance demands have influenced the design of GPUs to be optimized for higher performance per watt, even at the cost of relatively large power consumption. Although GPUs have high theoretical performance per watt, not all applications can utilize all available resources. We observe that due to various bottlenecks such as low occupancy, high bandwidth requirements, branch divergence, memory divergence, the achieved performance per watt is quite low (see Chapter 5). This thesis investigates key performance bottlenecks that lead to low performance and low energy efficiency in GPUs and proposes architectural techniques to address them.

This chapter is organized as follows. Section 1.1 elaborates the motivation for the thesis. Section 1.2 lists the objectives of the thesis. Section 1.3 highlights the main contributions of the thesis. Finally, Section 1.4 presents the organization of the thesis.

1.1 Motivation

Although GPUs were initially designed as accelerators for graphics applications, their massive compute power and high bandwidth made them attractive for general-purpose computing tasks such as scientific simulations. Scientific researchers were the first users of the general-purpose computing on GPUs due to their limited programmability. The high computational power of GPUs together with the recent explosion of data that need to be processed by applications, led to rapid evolution of general-purpose computing on GPUs, making them a key computing device in embedded systems, desktop computers, supercomputers, data centers and of late in cloud-based systems. There are several GPU vendors such as NVIDIA, AMD, Intel, and ARM. Among them, NVIDIA GPUs are most popular with a market share of about 69.1% [112].

The first general-purpose capable NVIDIA GPU, GTX-8800, was released towards the end of 2006 [122]. It has been only a decade, however, GPUs have already become an indispensable part of the mainstream computing. Since the release of the first general-purpose capable GPU, NVIDIA has released six main architectures: *Tesla, Fermi, Kepler, Maxwell, Pascal*, and *Volta* with several architectural improvements from *Tesla* to *Volta* [121, 119, 120]. We collect data from data sheets for different generations of NVIDIA GPUs to study their performance scaling. Figure 1.1 shows the scaling of NVIDIA Desktop GPUs in terms of the number of CUDA cores, performance as giga floating point operations per second (GFLOPS), and off-chip memory bandwidth as gigabytes per second (GB/s) from Tesla to Volta architecture. The key observations from the figure are:

- The number of CUDA cores and performance in terms of GFLOPS scaled in accordance with Moore's law, even better. The number of CUDA cores increased by

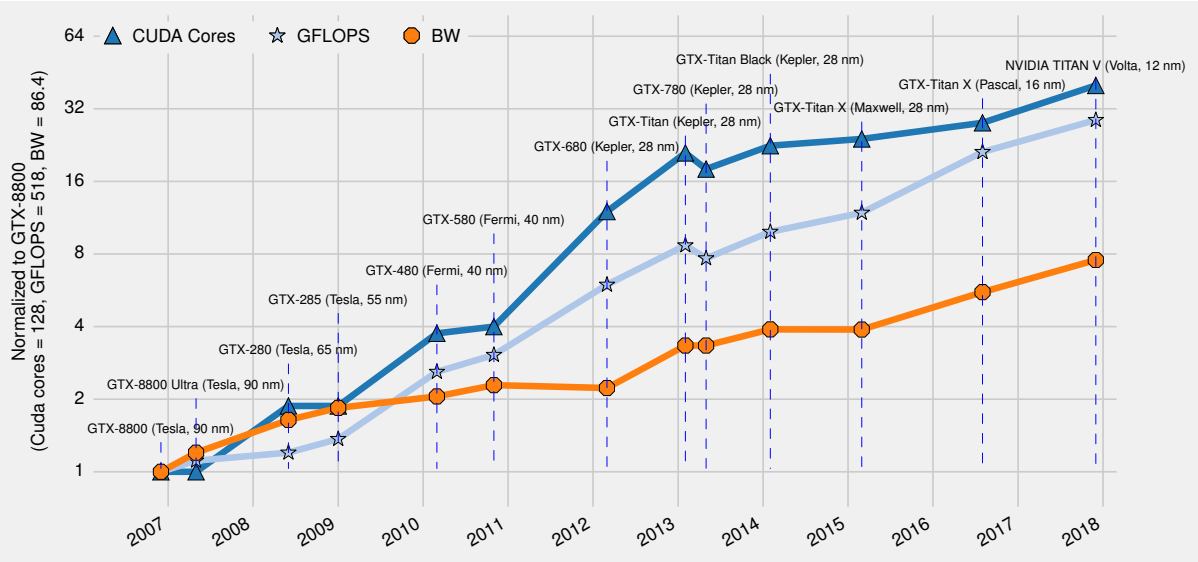


Figure 1.1: Scaling of NVIDIA Desktop GPUs normalized to first CUDA capable GTX-8800. The first number in the bracket shows architecture, while the second number shows the fabrication process in nanometer (nm).

about 40× and GFLOPS increased by about 28× in the last decade.

- The performance has increased significantly from GTX-8800 (Tesla architecture) which has 518 GFLOPS (10^9) to NVIDIA Titan V (Volta architecture) which has 14.9 TFLOPS (10^{12}).

- Memory bandwidth continues to scale as well, however, with respect to performance scaling, it is *far lagging* behind and thus, memory bandwidth scaling continues to be a major challenge for increasing GPU performance further.

The performance scaling of GPUs needs to continue to meet their ever-increasing demands, especially to attain exascale computing (10^{18}) challenge which can be considered a next big achievement in computer science. High bandwidth provided by GDDR (Graphics Double Data Rate) memory has been a key enabler of the continuous bandwidth and performance scaling of GPUs. Successive generations of GDDR (GDDR/2/3/5/5X/6) memories have increased overall bandwidth primarily by using wider interfaces and increasing frequency of off-chip signaling. However, the later generations of GDDR have issues such as large form factor, difficulty in the scaling of pin count. Moreover, research has shown that memory is a significant power consumer [90, 99, 84] and the traditional ways of increasing memory bandwidth by increasing the number of memory channels and/or frequency elevate the power consumption problem. Therefore, further scaling of GDDR bandwidth is not possible without significantly adding to system energy [113, 1]. Clearly, alternative ways to tackle the memory bandwidth problem are required.

To mitigate the issues of GDDR, recently, 3D stacked DRAM (Dynamic Random Access Memory) technologies such as Hybrid Memory Cubes (HMC) and High Bandwidth Memory (HBM, HBM2) have been introduced, offering much higher bandwidth and energy efficiency in a much small form factor compared to GDDR. However, future GPUs will demand even higher (multiple TB/s) DRAM bandwidth requiring further improvements in the bandwidth. Moreover, 3D stacked memories have their own problems such as much higher cost, significantly different interfaces, leading to adoption only in the high-end GPUs. For instance, HBM2 is about 4× more expensive compared to GDDR5 and it requires a different memory controller design. Therefore, the initial use of HBM is only limited to high-end graphics cards in HPC (High Performance Computing) domain.

This thesis proposes memory compression as an alternative to increase memory bandwidth. We note that existing memory compression techniques for GPUs exploit simple patterns for compression and trade low compression ratios for low decompression latency [6, 130, 26]. As GPUs are less sensitive to latency than CPUs, we explore the feasibility of a relatively complex memory compression technique which offers high compression ratio and performance gain. This thesis shows that more aggressive entropy encoding based memory compression technique delivers higher compression ratio and performance gain, provided its key challenges are addressed properly. Memory bandwidth compression techniques proposed in this thesis are orthogonal to technological improvements such as HBM, HMC and can be employed on top of them to meet the bandwidth and energy efficiency requirements of future high-throughput accelerators.

While continuous performance scaling of GPUs is desired to meet exascale computing challenge, seamless user experience, their high power consumption issue needs to be tackled as well. The high power consumption of GPUs has a significant impact on their reliability, performance scaling, economic viability, and deployment in a wide range of applications domains. A significant amount of work has been done by the research community as well as by the industry, both from the software and hardware perspective, to increase the energy efficiency of GPUs while continuing its performance scaling [69, 70, 89, 72, 53, 64]. We collect data from data sheets for different generations of NVIDIA GPUs to study their power and energy efficiency scaling. Figure 1.2 shows the power and energy efficiency scaling of NVIDIA Desktop GPUs in the last decade. The figure shows the peak power in watts and energy efficiency as GFLOP/Joule. The key observations are:

- The peak power of GPUs is in the range of 150-250 W and starting from GTX-780 (Kepler architecture), 250 W seems to be a norm for high-end desktop GPUs.
- While the initial focus of GPU design has been on performance, starting from the Kepler architecture, GPU design has also focused on energy efficiency. As can be seen from the figure, the energy efficiency of the Kepler architecture jumped compared to the energy efficiency of its predecessors.
- The energy efficiency of GPUs improved significantly starting from Kepler due to enhancements in the architecture such as simplified data-hazard detection in the

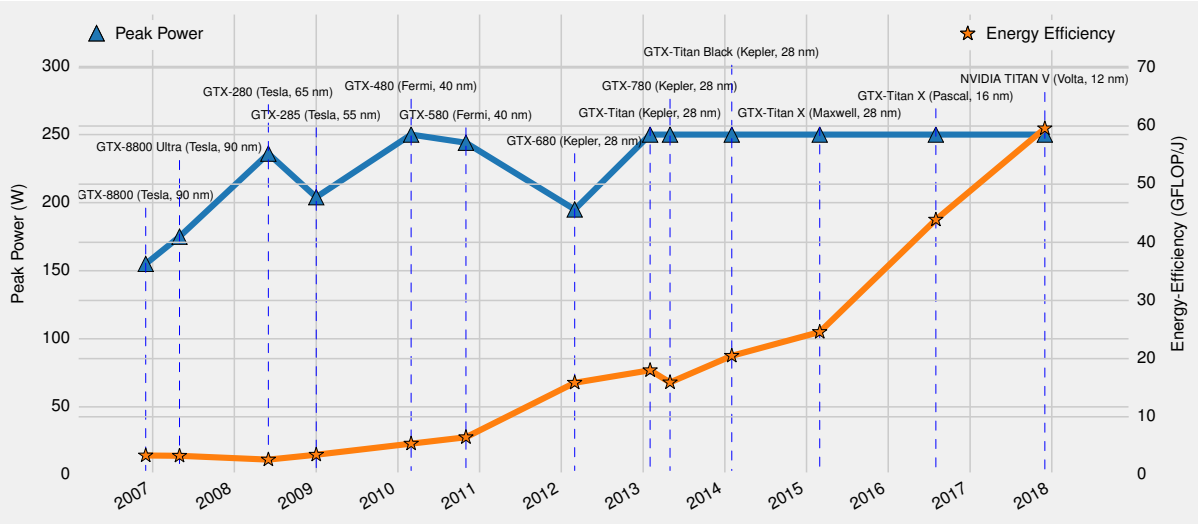


Figure 1.2: Scaling of power and energy efficiency of NVIDIA Desktop GPUs. The left y-axis shows peak power in watts, while the right y-axis shows energy efficiency in GFLOP/J. The first number in the bracket shows architecture, while the second number shows the fabrication process in nanometer (nm).

math pipeline, improved warp scheduler, primary clock for cores instead of 2× the shader clock. These improvements allowed Kepler to replace several complex and power-expensive hardware blocks with simpler blocks. Moreover, Kepler used 28 nm process node compared to 40 nm for Fermi.

- In general, recent generations of GPUs have better energy efficiency. For example, GTX-Titan X (Pascal architecture) is 1.8× more energy efficient compared to GTX-Titan X (Maxwell architecture). GDDR5X is supposed to be the main contributing factor besides process scaling and architectural improvements. The latest release of GTX-Titan X uses GDDR5X, which has 2× more memory bandwidth compared to its predecessor (GDDR5) and significantly higher energy efficiency. NVIDIA Titan V (Volta architecture) further improved the energy efficiency by 1.36× compared to GTX-Titan X (Pascal architecture). HBM2 which has higher bandwidth and energy efficiency compared to GDDR5X is one of the main contributing factors.

In addition to the theoretical energy efficiency shown in Figure 1.2, we conduct an energy efficiency study of a large number of GPU kernels (see Chapter 5) and observe a wide gap between the energy efficiency of the high performance and low performance kernels. The average energy per instruction for the high performance and low performance kernels is 0.27 nJ (10^{-9}) and 2.01 nJ for NVIDIA GTX580, respectively. The later is 7.5× less energy efficient compared to the former, a huge difference which is not favorable for the future growth of high-performance computing and far away from the exascale aim of 10 pJ (10^{-12}) per instruction [36]. Moreover, with the benefits of process scaling coming to an end, achieving higher energy efficiency is now even greater responsibility

of the GPU architects and programmers. Inherently, GPUs are more energy-efficient devices [64] compared to CPUs as they are optimized for throughput and performance per watt. Indeed, theoretical performance per watt of GPUs is much higher than CPUs, for example, NVIDIA's GTX 690 has 18.7 GFLOPS/W while Intel's Haswell i7 4770K has 5.3 GFLOPS/W. However, due to various performance bottlenecks which result in under-utilization of GPU resources, the achieved performance per watt is often much lower than the theoretical peak performance. There are several bottlenecks that contribute to low performance and low energy efficiency such as low occupancy, high memory bandwidth requirements, control flow divergence, and memory divergence [85, 84, 145, 135, 106, 50].

To enhance the energy efficiency of GPUs, we first need to discern their power consumption at a fine-grained level, understand the bottlenecks which lead to low performance and low energy efficiency [43, 72, 84] and then propose novel techniques to alleviate these bottlenecks. Therefore, in this thesis, we study the power consumption of GPUs at the component level, investigate the bottlenecks which cause low performance and energy efficiency and then propose architectural techniques to enhance performance and energy efficiency. To conduct architectural and energy efficiency research, cycle-accurate architectural and power simulators are a necessity as they help to evaluate alternative design choices and make early high-level design decisions. However, at the start of the thesis, there was no suitable GPU power simulator capable of conducting this research. Thus, we first develop a flexible and accurate GPU power simulator, enabling the research conducted in this thesis as well for the wider research community.

In essence, this thesis explores the following research questions:

- How can we develop a power simulator for GPUs that is both flexible as well as accurate to conduct energy efficiency research?
- What are the bottlenecks that cause low performance and low energy efficiency in GPUs? And what is the effect of eliminating bottlenecks on performance and energy consumption?
- What architectural techniques can be employed to improve the performance and energy efficiency of GPUs?

1.2 Objectives

We seek to tackle the problem of understanding bottlenecks which lead to low performance and low energy efficiency in GPUs and then improve their performance and energy efficiency. To enable energy efficiency research, we aim to develop a parameterizable power simulator for GPUs which can accurately estimate the power consumption and empower design space exploration and architectural research. Concretely, the objectives are:

Objective 1: To develop a flexible and accurate power simulator for GPUs, enabling estimation of power consumption, analysis of energy efficiency, and evaluation of new/existing

architectural techniques from the energy perspective.

Objective 2: To understand the bottlenecks causing low performance and low energy efficiency in GPUs, despite having high theoretical peak performance and energy efficiency.

Objective 3: To design new architectural techniques to enhance the performance and energy efficiency of GPUs.

1.3 Main Contributions

- The *first* contribution of the thesis is the modeling and estimation of GPU power consumption by developing a power simulator for GPUs. The power model is based on a combination of analytical and empirical approaches. The hybrid approach for power modeling improved flexibility compared to previous approaches which are mostly empirical and accuracy compared to pure analytical approaches. Our evaluation on a set of well-known benchmarks shows an average relative error of 11.7% and 10.8% between simulated and measured power for GT240 and GTX580, respectively. This contribution addresses the first objective of the thesis and results are published in [99].

- The *second* contribution is the investigation of bottlenecks that result in low performance and low energy efficiency in GPUs. We use the power simulator developed as a part of the first objective to study the energy efficiency of a wide range of kernels and show that 69% of the kernels have low performance and low energy efficiency. The average IPC of the low-performance category kernels is less than 25% of the peak IPC and average energy per instruction is 7.5× less than the average energy per instruction of the high-performance category kernels. To investigate the bottlenecks that lead to low performance and low energy efficiency, we divide the low-performance kernels into two categories: low occupancy and full occupancy. For the low occupancy kernels, we find architectural resources that cause low occupancy and study if increasing occupancy helps in increasing the performance. The results show that most of the kernels with low occupancy gain in performance and energy efficiency when occupancy is increased. For the kernels having full occupancy but still performing low, we show that these kernels are either limited by memory bandwidth, low coalescing efficiency or low SIMD utilization. This contribution addresses the second objective of the thesis and results are published in [84, 88].

- In the *third* contribution, we use memory-bound benchmarks from bottlenecks investigation performed in the second contribution and improve their performance and energy efficiency by increasing the effective off-chip memory bandwidth. We design a lossless memory compression technique which increases memory bandwidth by transferring data to/from the off-chip memory in a compressed form. We find that

the existing memory compression techniques for GPUs exploit simple patterns for compression and trade low compression ratios for low decompression latency. These techniques originally targeted CPUs and hence, traded low compression ratios for lower latency. Based on the evidence that GPUs are less sensitive to latency than CPUs, we propose the more complex entropy encoding based memory compression (E^2MC) technique for GPUs. We show that entropy encoding based memory compression is feasible for GPUs and delivers higher compression ratio and performance gain than state-of-the-art compression techniques, provided the key challenges of probability estimation, appropriate symbol length for encoding, and low decompression latency are addressed properly. The average compression ratio of E^2MC is 53% higher than the state of the art. This translates into an average speedup of 20% compared to no compression and 8% higher compared to the state of the art. Energy consumption and energy-delay-product are reduced by 13% and 27%, respectively. This contribution addresses the third objective of the thesis and results are published in [85].

- In the previous contribution, we show that memory compression is a promising approach for reducing memory bandwidth requirements and increasing performance and energy efficiency, however, we also observe that lossless memory compression techniques often result in a low effective compression ratio. We analyze reasons for the low effective compression ratio of several state-of-the-art lossless memory compression techniques and show the low effective compression ratio is caused by the large memory access granularity (MAG) exhibited by GPUs. Our analysis of the distribution of compressed blocks shows that a significant percentage of blocks are compressed to a size that is only a few bytes above a multiple of MAG, but due to the restrictions of MAG, a whole burst is fetched from memory. These few extra bytes significantly reduce the compression ratio and the performance gain that otherwise could result from a higher raw compression ratio. The *fourth* contribution of the thesis is the novel MAG aware selective lossy compression (SLC) technique for GPUs which increases the effective compression ratio and drives the performance and energy efficiency further. This is the first study that highlights the importance of MAG aware approximation. We propose two techniques to implement SLC and compare their advantages and disadvantages. For a lossy threshold of 16B and 32 MAG, we show an average speedup of 10% normalized to a state-of-the-art lossless compression technique with a maximum error of 0.64%. The energy consumption and energy-delay-product are reduced by 8.4% and 18.2%, respectively. We conduct sensitivity analysis to different MAGs and show an even higher significance of SLC at a larger MAG. For 64B MAG, we achieve a speedup of up to 35% with a maximum error of 1.8%. This contribution addresses the third objective of the thesis and results are published in [87, 83, 86].

The author of the thesis, together with others, published the following papers during the course of his dissertation:

1. Sohan Lal, Jan Lucas, Ben Juurlink, "SLC: Memory Access Granularity Aware Selective Lossy Compression for GPUs", *IEEE International Conference on Design Automation, and Test in Europe* ***(DATE)****, 2019, France.*

2. Sohan Lal, Ben Juurlink, "A Case for Memory Access Granularity Aware Selective Lossy Compression for GPUs", *ACM Student Research Competition Poster and Extended Abstract at IEEE/ACM International Symposium on Microarchitecture* ***(MICRO)****, 2018, Japan* ***(Semifinalist)****.*

3. Sohan Lal, Jan Lucas, Ben Juurlink, "SLC: Memory Access Granularity Aware Lossy Compression for GPUs", *14^{th} International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* ***(ACACES)****, 2018, Italy.*

4. Jan Lucas, Sohan Lal, Ben Juurlink, "Optimal DC/AC Data Bus Inversion Coding", *International Conference on Design Automation, and Test in Europe* ***(DATE)****, 2018, Germany.*

5. Sohan Lal, Jan Lucas, Ben Juurlink, "E^2MC: Entropy Encoding Based Memory Compression for GPUs", *IEEE International Conference on Parallel and Distributed Processing Symposium* ***(IPDPS)****, 2017, USA.*

6. Maurice Peemen, Runbin Shi, Sohan Lal, Ben Juurlink, Bart Mesman, and Henk Corporaal, "The Neuro Vector Engine: Flexibility to Improve Convolutional Net Efficiency for Wearable Vision", *International Conference on Design Automation, and Test in Europe* ***(DATE)****, 2016, Germany.*

7. Sohan Lal, Jan Lucas, Michael Andersch, Mauricio Alvarez Mesa, Ahmed Elhossini, and Ben Juurlink, "GPGPU Workload Characteristics and Performance Analysis", *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* ***(SAMOS)****, 2014, Greece.*

8. Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez Mesa, and Ben Juurlink, "How a Single Chip Causes Massive Power Bills - GPUSimPow: A GPGPU Power Simulator", *IEEE International Symposium on Performance Analysis of Systems and Software* ***(ISPASS)****, 2013, USA.*

9. Sohan Lal, Jan Lucas, Mauricio Alvarez Mesa, Ahmed Elhossini, and Ben Juurlink, "Exploring GPGPUs Workload Characteristics and Power Consumption", *9^{th} International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* ***(ACACES)****, 2013, Italy.*

10. Jan Lucas, Sohan Lal, Mauricio Alvarez Mesa, Ahmed Elhossini, and Ben Juurlink, "DART: A GPU Architecture Exploiting Temporal SIMD for Divergent Workloads", *9^{th} International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* ***(ACACES)****, 2013, Italy.*

Table 1.1: Author's contributions to published papers.

Publication No.	Contribution of the author of this thesis
1	Analyzed reasons for the low effective compression ratio of several state-of-the-art lossless memory compression techniques, quantitatively studied the distribution of the compressed blocks and designed a novel memory access granularity aware selective lossy compression technique and estimated its area and power overhead.
2	Showed that low effective compression ratio exists in several state-of-the-art memory compression techniques and made a case for memory access granularity aware selective lossy compression.
3	Analyzed raw and effective compression ratios of several state-of-the-art memory compression techniques, showed that they suffer due to the large memory access granularity and discussed the possibility of a memory access granularity aware lossy compression.
4	Synthesized hardware implementations of different data bus inversion (DBI) schemes and analyzed their area and power trade-offs.
5	Developed an entropy encoding based memory compression technique for GPUs, addressed its key challenges and estimated area and power overhead.
6	Optimized face detection and speed sign detection algorithms for easy auto-vectorization on ARM processors, simulated optimized algorithms using GEM5 and McPAT simulators to estimate performance, area and power overhead. Analyzed and compared results with proposed neuro vector engine.
7	Investigated bottlenecks causing low performance and low energy efficiency in GPUs, analyzed the effect of bottlenecks elimination and studied correlation between GPU components power consumption and workload metrics.
8	Developed power models for several GPU components such as shared memory, off-chip memory, Special Function Units (SFU), caches, and modified gpgpu-sim to generate activity factors for various components.
9	Studied GPU power consumption at components level and investigated their correlation with workload metrics.
10	Investigated alternative memory coalescing possibilities for the DART architecture compared to a regular GPU architecture.

The publication numbered 8 is a collaborative work between Jan Lucas, the author

of the thesis, Michael Andersch, Mauricio Alvarez Mesa, and Ben Juurlink. The work done by Jan Lucas, Michael Andersch, Mauricio Alvarez Mesa, and Ben Juurlink is also included in the thesis with their permission to provide a full overview and evaluation of the power simulator. The effort of the author of the thesis in the above-mentioned publications including the collaborative work is summarized in Table 1.1. The publications numbered 4, 6, and 10 are not referred in the thesis. The complete list of publications is also included in Chapter 9.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 surveys the related work and highlights the advances made over the state of the art. Chapter 3 reviews GPU architecture and provides an overview of data compression in general and memory compression techniques used as baselines in the thesis in particular. Chapter 4 presents a power simulator for GPUs. Chapter 5 uses the power simulator to analyze GPU performance bottlenecks. Chapter 6 proposes an entropy encoding based memory compression technique to reduce memory bandwidth requirements. Chapter 7 proposes a memory access granularity aware approximation technique to address the issue of low effective compression ratio. Finally, Chapter 8 summarizes the key contributions, draw conclusions and suggest future work.

2 Related Work

In this chapter, we review the work related to this thesis. The related work is divided into four main categories aligning with the four main contributions of the thesis which also determines the organization of this chapter. First, Section 2.1 covers the work related to GPU power modeling and estimation which ranges from pure empirical, pure analytical to hybrid modeling where empirical and analytical approaches are combined. Second, the work related to GPU performance bottlenecks and components power consumption is analyzed in Section 2.2. Third, Section 2.3 surveys the work related to lossless memory compression techniques for GPUs, highlighting the opportunities and challenges for a relatively complex memory compression technique. Fourth and last, we review the work related to approximate computing techniques for GPUs in Section 2.4. Finally, we sum up the advances made over the state of the art in Section 2.5 and summarize the chapter in Section 2.6.

2.1 GPU Power Modeling and Estimation

We compare our power modeling work from the perspective of power modeling approach and methodology employed for its validation. As briefly described above, GPU power modeling approaches range from pure empirical, pure analytical to hybrid modeling which combines the bests of both the approaches. The first approach in literature to model power is the empirical approach which is entirely based on the measurement data obtained from a particular device. There are several works which empirically model GPU power consumption [61, 100, 166, 176]. Hong and Kim [61] propose an integrated performance and power prediction model for a GPU architecture to predict the optimal number of active processors for a given application. Unlike most previous empirical power models which require measured execution time, hardware performance counters, their model uses predicted execution time to estimate dynamic power events. The geometric mean of the error between predicted and measured power is 2.5% for microbenchmarks and 9.2% for GPU kernels. Ma et al. [100] present a statistical scheme for analyzing and modeling the power consumption of GPUs. Based on the measured power consumption, runtime workload signals such as the percentage of the time when a texture unit is busy, and performance data, they build a statistical regression model which is capable of dynamically estimating the power consumption of a GPU based on a subset of workload signals. The geometric mean of the prediction error is 8.9%. Nagasaka et al. [115] also develop a statistical power prediction model for GPUs by using performance counters and report an average error of 4.7%. Wang and Chen [166] also develop a statistical power model for

GPUs and show that the power consumption is directly proportional to the computational intensity and the number of active SMs. The average relative error is less than 6%. A similar work is done by Zhang et al. [176]. While the empirical power models deliver higher accuracy for the architecture they are designed for, they lack the flexibility to make accurate predictions for GPUs with different architectural parameters and designs.

The second approach to model power is the analytical approach which is generic and parameterized. Wang [165] builds a high level, purely analytical power model for the main functional units of GPUs by integrating gpgpu-sim [15], Wattch [20], and Orion [71]. However, the power model is highly coarse-grained without any details about how GPU-specific components such as warp control unit, special functions units, coalescing unit are modeled. Moreover, no discussion of power model validation is presented in the paper. The power model is used to show that additional cores only decrease the energy efficiency of GPUs when the gap between the shaders and memory speed increases and the traditional dynamic voltage frequency scaling schemes could also be applied to GPUs. While analytical approach generally shows 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 data. Moreover, it is not possible to design analytical models for irregular units such as functional units.

The third approach to model power consumption is the hybrid approach which combines empirical and analytical approaches to deliver reasonably high accuracy and also offers flexibility for wider applicability. Ramani et al. [133] use hybrid approach to model GPU power consumption. The limitations of their work are similar to Wang [165]. The paper lacks information on power models for GPU-specific components and there is no discussion of the validation of power model and its accuracy. Moreover, they use an in-house performance simulator. They use the power model to demonstrate the effectiveness of the framework in exploring how encoding on long buses can save as much as 15-30% power at medium to high activity levels. Another hybrid and well known power model is McPAT [92], however, it is used to model CPU power consumption. While the general design philosophy of McPAT and some structures can be reused for GPU power modeling, a significant amount of work is needed to add GPU-specific components to it.

A common methodology used to validate power simulators is usually the usage of a custom designed power measurement setup that can measure the actual power consumed by the real hardware and then compare the measured power with the estimated power reported by the power estimation tool. However, in terms of GPU power measurements, many previous approaches have made strong assumptions about the hardware they measure, leading to inaccurate measurement methodologies. For example, Hong and Kim [61] assume a GPU power can be calculated by measuring the power of the entire PC under load and subtracting it from the power of the PC in idle state. This assumption is highly naive since the power used by the remaining PC components is usually not constant and the measurement results will include ATX power supply losses. Large bypass capacitors inside the ATX power supply prevent the accurate measurement of power for kernels which run fewer than 50 ms. Other papers [165, 167, 115, 100] use improved measure-

Table 2.1: Key features of GPUSimPow compared to the state of the art.

Work	Hybrid approach?	Modeled GPU-specific components?	Power model validated?	Measured all sources?
[61, 100, 166, 176, 165, 167, 115]	✗	✓	✓	✗
Ma et al. [100]	✗	✓	✓	✓
G. Wang [165]	✗	✗	✗	✗
PowerRed [133]	✓	✗	✗	✗
McPAT [92]	✓	N/A	✓	✗
GPUWattch [90]	✓	✓	✓	✓
GPUSimPow [99]	✓	✓	✓	✓

ment methodologies, but still exhibit multiple weaknesses. To the best of our knowledge, all these published methodologies either fail to measure all power sources, e.g. they do not measure the power provided via the graphics card slot (PCI-E) [165, 167] but only measure the power from the external source, measure only the current and assume constant voltages [115], or use low sampling frequencies that prevent them from measuring short-term power variations [100, 167].

We develop a GPU power simulator called GPUSimPow [99] by integrating an architectural simulator called gpgu-sim [15] and heavily modified CPU power simulator called McPAT [92]. GPUSimPow is not only highly parameterizable but also provides an accurate estimate of static and dynamic power. GPUSimPow has an average relative error of 11.7% and 10.8% between simulated and measured power for GT240 and GTX580, respectively. Our GPU power simulator improves state of the art in several ways. First, we use hybrid approach that is both architecturally flexible and reasonably accurate. Second, we develop detailed power models for GPU-specific components. Third, we rigorously validate our power model by using a custom power measurement setup. Our power measurement setup measures power from all sources and uses high sampling frequency so that even the short power events can be measured accurately. Table 2.1 lists the key features of GPUSimPow compared to the state of the art. GPUSimPow and GPUWattch [90] have the same key features. We published GPUSimPow [99] in April 2013. GPUWattch [90], a very closely related GPU power simulator, was published in June 2013. A detailed comparison of these simulators is presented in Section 4.4.3.

2.2 GPU Performance Bottlenecks and Analysis

The work related to GPU performance bottlenecks and analysis is divided into two categories. First, we review the work done for estimating the power consumption of GPUs

at components level and their correlation with workload metrics. Second, we survey the work related to analysis of GPU performance bottlenecks.

There are previous works which estimate GPU power consumption, but they estimate power consumption at a very coarse-grained level. For example, Ma et al. [100] and Zhang et al. [176] use statistical analysis to develop a power model for GPUs and report power consumption of the entire GPU. Gebhart et al. [53] use a high level power model to estimate the total core power. According to their work, cores consume up to 70% of total power, but this is not enough to optimize the power consumption as we need to understand the power consumption at much fine-grained level. Wang [165] reports power consumption for some components, however, his power model lacks GPU-specific components. Moreover, the power model is not validated against measured power. Thus, the accuracy of the reported results is unknown. Abe et al. [4] also raise the issue of high power consumption by GPUs and need for energy-efficient GPUs. They provide power and performance analysis of GPU-accelerated system and show system energy can be reduced by 28% at performance decrease of within 1% by controlling the voltage and frequency levels of GPUs. In contrast to this, we study GPU power consumption at detailed component level by using GPUSimPow [99]. The in-depth exploration of GPU power consumption is enabled by the recent release of GPU power simulators (GPUSimPow [99] and GPUWattch [90]) which use hierarchical approach to have detailed power models for GPU components.

Using metrics to understand workload characteristics is well known [73, 25, 57, 22]. Goswami et al. [57] propose a set of microarchitecture agnostic workload metrics to characterize them. They provide workload categorization based on workload subspaces such as branch and memory divergence. Burtscher et al. [22] study the performance of irregular programs on GPUs. They define two metrics for irregularity called control-flow irregularity and memory-access irregularity and use these metrics to study how irregular kernels differ from regular kernels. Che et al. [25] characterize GPU workloads in terms of architectural, parallelization, and synchronization characteristics. Like Che et al. [25], Kerr et al. [73] also use several metrics to characterize GPU workloads, however, former run benchmarks on real hardware while later develop an emulator for NVIDIA parallel thread execution virtual machine (PTX). All of the above mentioned works use metrics to study workload characteristics, but none of them study the correlation between workload metrics and components power. We compute the correlation between workload metrics and components power to understand the power characteristics of various workloads. In addition to this, we also quantify the change in components power consumption with the change in workload characteristics.

Some researchers have used micro-benchmarks to understand the performance as well as power characteristics of GPUs [176, 105]. Zhang et al. [176] design a set of micro-benchmarks to study the power consumption of different functional units of a GPU. Based on those results, they derive instructive principles that can guide the design of power-efficient high performance computing systems. Mei and Chu [105] study the characteristics of the memory hierarchy using micro-benchmarks. Specifically, they investigate

Table 2.2: Key features of our approach and state of the art to study GPU components power consumption, identify performance bottlenecks and performance prediction after bottlenecks elimination.

	Components power and correlation with metrics		GPU performance bottlenecks			
Work	Component level?	Correlation with metrics?	Identified bottlenecks?	Performance simulator for identification?	Performance simulator for prediction?	Energy analysis?
[100, 176, 4]	✗	✗	N/A	N/A	N/A	N/A
Blem et al. [17]	N/A	N/A	✓	✓	✗	✗
Our [84]	✓	✓	✓	✓	✓	✓

the structures of GPU cache systems, such as the data cache, the texture cache and the translation look-aside buffer (TLB) and also investigate the throughput and access latency of GPU global and shared memory.

Blem et al. [17] characterize a set of benchmarks to find performance bottlenecks and then predict the performance improvements after mitigating those bottlenecks. We also investigate performance bottlenecks that cause low performance, but there are key differences both in the methodology and performance metrics. First, we use a performance simulator not only for bottlenecks identification but also for performance prediction. Blem et al. [17] use analytical model to predict performance, which according to their work has error in the range -70% to 2×, which is very high and a limitation of their work. Second, we also analyze low performance workloads from the energy efficiency perspective and provide the energy reduction when a bottleneck is eliminated. Third, they do not consider the case that low occupancy can lead to low performance. We show that a large number of kernels have low occupancy and how increasing the occupancy helps in increasing the performance and energy efficiency. Table 2.2 summarizes the key features of our approach and state of the art to study GPU components power consumption, identify performance bottlenecks and quantify the effect of bottlenecks elimination on performance.

There are some GPU performance analysis studies that are conducted after our work [84]. We summarize the most important ones related to this thesis. Qiumin et al. [171] study the performance and utilization statistics of the core components of GPUs for graph processing applications. They show that graph applications tend to execute kernel and data transfer more frequently than non-graph applications and suggest several approaches to optimize GPU hardware for enhancing the performance of graph applications. Madougou and Nieuwpoort [101] propose a tool for bottlenecks analysis and performance prediction for GPU-accelerated applications. Based on random forest modeling and hardware performance counters, the proposed tool is used to quickly and accurately evaluate ap-

plication performance on GPU-based systems for different problem characteristics and different hardware generations. However, as the tool is based on a statistical approach, the applicability of the tool is restricted to applications where enough training data is available. Recently, deep learning models such as convolutional neural networks (CNNs) have achieved great success in a number of applications such as image classification, speech recognition and natural language understanding, primarily due to significant increase in the computational capability of hardware devices. The computational complexity of training CNNs on large data sets has led to several open-source parallel implementations on GPUs. Li et al. [93] compare these implementations over a wide range of parameter configurations, investigate potential performance bottlenecks and point out a number of opportunities for further optimization. Kim et al. [76] analyze GPU performance characteristics for five popular deep learning frameworks: Caffe, CNTK, TensorFlow, Theano, and Torch in the perspective of a representative CNN model, AlexNet. They suggest criteria to choose convolution algorithms, study scaling of deep neural networks (DNNs) in a multi-GPU context and show speedup of up to 2× by just tuning the options provided by the frameworks.

2.3 Lossless Memory Compression for GPUs

There is a significant amount of prior work for memory compression. Memory compression has been used to increase cache capacity, reduce memory bandwidth requirement with/without increasing effective memory capacity. We first review memory compression work for GPUs and then report more work done for CPUs. Finally, we review the use of compression for reducing error checking and correcting (ECC) overhead.

GPUs employ compression for color and texture data. ARM Frame Buffer Compression (AFBC) is a lossless image compression technique and is available on Mali GPUs [11]. AFBC is claimed to reduce the required memory bandwidth by up to 50% for graphics workloads. AMD and NVIDIA use lossless Delta Color Compression to store image data as delta from the previous pixel and save bandwidth [68]. Above mentioned techniques are only applicable for image data and not for general purpose data and to the best of our knowledge, the micro-architectural details of these techniques are also proprietary.

Recent research has shown that compression can be used for general purpose workloads [146, 89, 6, 160, 77]. Sathish et al. [146] use Cache-Packer (C-Pack) [26], a dictionary based compression technique to compress data transferred through memory I/O links and show performance gain for memory-bound applications. However, they do not report compression ratios and their primary focus is to show that compression can be applied to GPUs and not how much can be compressed. Lee et al. [89] use Base-Delta-Immediate (BDI) compression [130] to compress data in the register file. They observe that the computations that depend on the thread-indices operate on register data that exhibit strong value similarity and there is a small arithmetic difference between two consecutive thread registers. Compression of registers enables power saving due to activation of fewer registers banks. Vijaykumar et al. [160] use underutilized resources to create assist warps that

can be used to accelerate the work of regular warps. These assist warps are scheduled and executed in hardware. They use assist warps to compress cache block before writing to memory and decompress it before placing to cache. In contrast to this, our compression technique is hardware based and is completely transparent to the warps. The assist warps may compete with the regular warps and can potentially affect the performance. Kim et al. [77] propose Bit-plane Compression (BPC) that first uses transformation to increase the compressibility of the data and then uses either run-length or Frequent Pattern Compression (FPC) [6] to compress the transformed data. The authors show that transformation results in higher compression ratio.

Most existing memory compression techniques exploit simple patterns for compression and trade low compression ratios for low decompression latency. For example, FPC [6] replaces predefined frequent data patterns, such as consecutive zeros, with shorter fixed-width codes. Frequent patterns are differentiated by a 3-bit prefix. C-Pack [26] exploits fixed frequently occurring static patterns like FPC and also uses dynamic dictionary to adapt to other frequently appearing words. BDI compression [130] is based on the observation that values stored in a cache line have high value similarity and low dynamic range and therefore, stores one or more values as base and the relative difference between the values as deltas. While these techniques can decompress with a few cycles, their compression ratio is low, typically only 1.5×. All these techniques originally targeted CPUs and hence, traded low compression for lower latency.

As GPUs are less sensitive to latency, we propose a relatively more complex Entropy Encoding Based Memory Compression (E^2MC) technique for GPUs. We address the key challenges of probability estimation, appropriate symbol length for encoding, and low decompression latency. We use Huffman coding [150] for entropy encoding. Huffman-based Statistical Cache Compression (SC^2) has been used for CPUs [10], but to the best of our knowledge no work has used Huffman-based memory compression for GPUs. Moreover, the differences in CPU and GPU architectures offer new challenges which need to be tackled as well as new opportunities which can be harnessed. For example, GPUs use longer cache lines (128B is a typical value) than CPUs (64B is a typical value) which has implications on compression and decompression latency. SC^2 [10] has the highest compression ratio for 32-bit symbols. In contrast to CPUs, GPUs have been optimized for FP operations and 32-bit granularity does not work well for GPUs. We show higher compression ratio and performance gain for 16-bit symbols instead of 32-bit symbols for E^2MC. Due to the differences in cache line size and symbol length for encoding, the latency requirements of SC^2 and E^2MC are significantly different. SC^2 uses 64B cache lines and 32-bit symbols, therefore, it only needs to (de)compress 16 symbols whereas E^2MC uses 128B cache lines and 16-bit symbols, it needs to (de)compress 64 symbols (4×). The higher number of symbols means higher compression and decompression latency for GPUs. Although GPUs can hide latency to some extent, we show too much increase can also degrade their performance. Therefore, we reduce the decompression latency of E^2MC by parallel decoding. E^2MC results in 53% higher compression ratio and 8% increase in speedup compared to state of the art and saves 13% energy and 27% EDP compared to

Table 2.3: Key features of E^{2}MC and state-of-the-art lossless memory compression techniques for GPUs.

Work	Designed for GPUs?	Employed on GPUs?	Exploited complex patterns?	Traded low compression for low latency?
BDI [130],FPC [6], C-PACK [26]	✗	✓	✗	✓
HyComp [9]	✗	✗	✓	✗
BPC [77]	✓	✓	✓	✓
SC^2 [10]	✗	✗	✓	✗
E^{2}MC [85]	✓	✓	✓	✗

no compression. Table 2.3 summarizes the key features of E^{2}MC [85] and state-of-the-art lossless memory compression techniques for GPUs.

There is more work for cache compression for CPUs [172, 177, 59]. Yang et al. [172] develop a data compression scheme for use in a first level cache which exploits frequently accessed values. The authors show that on an average nearly 40%, 52% and 51% of cache lines of sizes 4, 8 and 16 words respectively can be compressed to at least half of their sizes by exploiting top 2, 4 and 8 frequent values respectively. An earlier work from the same authors presents profiling techniques for identifying frequent values [177] where they found that 10 distinct values occupy over 50% of all memory locations for 6 out of the 8 SPECint95 benchmarks. Hallnor and Reinhardt [59] propose a uniform compressed memory hierarchy that increases last-level on-chip cache capacity, off-chip bandwidth, and main memory size, while avoiding compression and decompression overheads between levels. The main-memory compressed organization and the compression algorithm are taken from IBM's Memory Expansion Technology (MXT) system [3]. Arelakis et al. [9] propose hybrid cache compression (HyComp) technique for CPUs which improves the compression ratio by selecting a suitable compression method based on the specific data-type. HyComp selectively uses either BDI, C-Pack, Huffman encoding [10] or FP-H method based on type prediction during runtime. FP-H divides a floating-point number into three fields: Exponent, Mantissa-High, and Mantissa-Low and then employs Huffman encoding [10] to compress each of these fields in isolation.

Some works also use memory compression to increase the effective capacity along with reducing memory bandwidth requirements [3, 42, 129, 27, 79, 132]. The main challenge is to find the starting address of a compressed cache block in the main memory which depends on the compressed size. The additional overhead to compute the starting address of the cache block increases complexity, system cost and latency. IBM MXT [3] uses a large (32MB) on-chip translation table to store the mapping between original and compressed addresses, however, this requires significant area and power cost. Ekman and Stenstrom [42] reduce the latency of calculating the starting address of a cache block in main memory by speculatively computing the main memory address of every last-level

cache access in parallel to cache access. However, this also wastes significant amount of energy as many accesses to last-level cache do not result in a miss. Pekhimenko et al. [129] propose a framework called linearly compressed pages (LCP) to compute the starting address. LCP fixes a target compression ratio for each block so that the starting address can be located with a simple linear equation. All blocks which cannot be compressed to a fixed ratio are stored at a reserved location in the same compressed page. Choukse et al. [27] propose a pragmatic memory compression called Compresso that reduces compressed data movement in a hardware compressed memory. The authors show Compresso exhibits only 15% compressed data movement accesses, as compared to 63% in an enhanced LCP-based baseline. Kim et al. [79] propose dual memory compression architecture which uses two kinds of compression: Lempel-Ziv at 1KB granularity for cold pages and LCP with BDI for hot pages. It decides between the 2 compression mechanisms in an OS-transparent fashion. Qian et al. [132] propose compression to increase the capacity of Hybrid Memory Cubes by having compressed and uncompressed regions per vault.

Since the main memory is prone to errors and failures, large scale systems and critical servers utilize error checking and correcting (ECC) mechanisms to meet their reliability requirements. However, such systems need extra space in memory to store ECC metadata and an extra access to read the metadata which reduces the available space and bandwidth for actual data. Recent work has shown that by employing simple compression at block granularity enough space can be created to fetch the ECC metadata along with the data [78, 127]. The goal of the compression is not to achieve the highest compression ratio but to create just enough space for metadata. For example, Palframan et al. [127] show that a block size reduction by 32-bit is enough to fetch the ECC metadata for free.

Hong et al. [62] propose a framework called Attaché that mitigates metadata accesses to provide a near-ideal speedup. Attaché stores metadata along with the cache block in memory and uses a predictor in the memory controller to predict the value of metadata. Attaché reduces 99.997% bandwidth overheads of Metadata accesses.

2.4 MAG Aware Approximation for GPUs

Traditionally, approximate computing has been mainly limited to CPUs and several approximate computing techniques such as memoization [30], loop perforation [151], task skipping [108] have been proposed to trade some accuracy for higher performance and energy efficiency. Recently, approximate computing usage for GPUs has increased as GPUs have become an essential computational unit in the mainstream computing. To the best of our knowledge, there is no work directly related to memory access granularity (MAG) aware approximation, however, approximation has been used for GPUs at software, hardware, and hybrid levels [141, 140, 145, 12, 175, 152, 91, 168, 102]. Samadi et al. [141, 140] use static compilers to automatically generate different versions of GPU kernels with different aggressiveness. Depending upon the target output quality (TOQ), a run-time system selects the appropriate version of an approximate kernel and trade-off accuracy with performance. Arnau et al. [12] propose hardware based memoization tech-

nique to remove redundant fragment computation for graphical applications in low-power mobile GPUs. Yazdanbakhsh et al. [175] propose rollback-free value prediction scheme to reduce long memory latency and memory bandwidth of GPUs. When a safe to approximate loads misses in the cache, the load value is predicted instead of actually fetching from the off-chip memory. Sutherland et al. [152] use texture cache approximation as a method to eliminate costly global memory accesses. Sathish et al. [146] use lossless and lossy memory I/O link compression for GPUs, however, for the lossy compression they always truncate the least significant bits. Sartori and Kumar [145] use approximate computing to eliminate the control and memory divergence on GPUs. They use branch herding to eliminate control divergence by forcing all threads in a warp to take the same control path and data herding to eliminate memory divergence by forcing each thread in a warp to load from the same memory block. They use static analysis and compiler framework to prevent exceptions when control and data errors are introduced, a profiling framework that aims to maximize performance while maintaining acceptable output quality, and hardware optimizations to improve the performance benefits of exploiting error tolerance through branch and data herding. They have both software and hardware based implementation of branch herding. Maier et al. [102] propose a local memory-aware kernel perforation technique that first skips the loading of parts of the input data from global memory, and later uses reconstruction techniques on local memory to reach higher accuracy while having performance similar to state-of-the-art techniques. Traditionally, approximate computing has been used for applications with high error resilience, however, recently, approximate computing usage has also started in scientific applications which have lower error tolerance [91].

In contrast to the above mentioned techniques, we propose a novel MAG aware Selective Lossy Compression (SLC) technique for GPUs which increases effective compression ratio and performance gain. SLC exploits the observation that several state-of-the-art lossless memory compression techniques result in a low effective compression ratio because a significant percentage of blocks are compressed to a size that is only a few bytes above a multiple of MAG, but a whole burst is fetched from the memory. SLC selectively approximates these blocks to increase the effective compression ratio and performance gain with a very low loss in accuracy.

Approximate computing has some similarities with approximate storage, for example, both trade-offs accuracy for higher performance. However, there are important differences, for example, approximate computing deterministically approximates data while approximate storage retains data with some probability. Some of the important works in approximate storage can be found here [138, 97, 157, 143], however, a complete discussion of them is out of the scope of this thesis.

2.4.1 Qualitative Comparison with Lossy Compression Techniques

Although, SLC only selectively approximates blocks, at the end, it is a hardware-based approximation technique. Warped Approximation [168], RFVP [175], Truffle [44], NPU [45]

Table 2.4: Key features of SLC and state-of-the-art approximation techniques for GPUs.

Work	Designed for GPUs?	MAG Aware approximation?	Toggleable on top of lossless compression?
[30, 151, 108, 44, 45]	✗	✗	✗
[141, 140, 145, 12, 175, 152, 91, 168, 102]	✓	✗	✗
SLC [87]	✓	✓	✓

are other existing hardware-based approximation techniques. While Warped Approximation has only been studied for GPUs as it exploits intra-warp value similarity prevalent in GPUs, RFVP has been studied for both CPUs and GPUs. Truffle and NPU have been evaluated for CPUs. We think a direct comparison of these techniques with SLC is not viable, however, we present our qualitative perspective.

Warp Approximation [168] dynamically detects intra-warp value similarity using hardware and executes only a single representative thread and uses its value to approximate the values of other threads in the same warp. Warp Approximation can reduce execution unit energy by 37% and register file energy by 28%, improving overall GPU energy consumption by 26% with minimal quality degradation (typically 1%).

RFVP [175] is a rollback-free value prediction scheme which reduces long memory latency and memory bandwidth by predicting the value of a load instead of actually fetching from the off-chip memory when a safe to approximate loads misses in the cache. For GPUs, RFVP improves performance and reduces energy consumption on average by about 6% with 1% loss in quality. For out-of-order modern CPUs, authors show on average 8% performance improvement with 0.8% average quality loss on an approximable subset of SPEC CPU 2000/2006.

Truffle [44] uses disciplined approximate programming to declare which parts of a program can be computed approximately and then uses an efficient mapping of disciplined approximate programming onto hardware. Authors extend ISA that provides approximate operations and storage and propose microarchitecture design called Truffle that efficiently supports the ISA extension. Truffle results in 43% reduction in energy consumption for out-of-order design.

NPU [45] uses program transformation to select and train a neural network to mimic a region of imperative code. After the learning phase, the compiler replaces the original code with an invocation of a low-power accelerator called a neural processing unit (NPU) which provides speedup of 2.3× with quality loss of about 7%.

SLC is orthogonal to above mentioned techniques, for example, Warp Approximation reduces execution unit and register file energy by power gating, NPU improves performance by skipping a region of code and approximating it with a neural network model whereas SLC reduces GPU energy consumption by reducing memory bandwidth require-

ments. Although, RFVP also reduces memory bandwidth, it does so by predicting the values of some of the loads that miss in a cache. SLC can be used on other loads from memory. In terms of performance, SLC reduces energy consumption by 8.4% and EDP by 18.2% with mere 0.2% loss in quality normalized to the state-of-the-art lossless compression technique. Table 2.4 highlights the key features of SLC and state-of-the-art approximation techniques for GPUs.

2.5 Summary of Advances Over State of the Art

This thesis made four main contributions in the direction of GPU power modeling and estimation, performance bottlenecks analysis, memory compression, and MAG aware approximation. Below we summarize the advances over state of the art in these directions.

We developed a power simulator for GPUs called *GPUSimPow* [99]. Our GPU power simulator advanced state of the art in several ways. First, in contrast to previous pure empirical approaches [61, 100, 166, 176] and pure analytical approach [165], we used a hybrid power modeling approach. The hybrid approach improved flexibility compared to the pure empirical approaches and accuracy compared to the pure analytical approach. While hybrid power modeling has been used before by some researchers [133, 92], they either have limitations compared to our approach or not applicable in our case. Ramani et al. [133] used the hybrid approach to model GPU power, however, they do not provide information on power models for GPU-specific components and there is no discussion of the validation of power model and its accuracy. Moreover, they use an in-house architectural simulator. Li et al. [92] also used the hybrid approach, however, they developed a power simulator for CPUs called McPAT. While our power simulator is also based on the general design philosophy of McPAT and reused some structures for GPU power modeling, a significant amount of work was performed to develop GPU-specific components. Second, we rigorously validated our power simulator by using a custom power measurement setup and our validation methodology is more accurate compared to previous approaches that made strong assumptions about the hardware they measure, leading to inaccurate measurement methodologies [61]. Compared to other power measurement methodologies [165, 167, 115, 100], we measured power from all sources and used high sampling frequency so that even the short power events can be measured accurately.

We studied the energy efficiency of a large number of GPU kernels and investigated bottlenecks leading to low performance and low energy efficiency. We showed that about 70% of the kernels have low performance and low energy efficiency. We also predicted the performance improvement when a particular bottleneck is mitigated. Blem et al. [17] also characterized benchmarks for performance bottlenecks and predicted the performance improvements after mitigating those bottlenecks, however, there are key differences both in the methodology and performance metrics. Unlike Blem et al. [17], we used performance simulator not only for bottlenecks identification but also for performance prediction. Blem et al. [17] used an analytical model to predict performance that has an error in the range -70% to 200%, which is very high and a strong limitation of their work. We used gpgpu-

sim for performance prediction that has an average absolute error of 35% and 62% for Tesla and Fermi architectures, respectively. The average absolute error is high due to outliers [49]. The IPC correlation is 98.3% and 97.3% for Tesla and Fermi architectures, respectively. In addition to performance, we also provided an estimate of the energy reduction when a bottleneck is eliminated. Furthermore, Blem et al. [17] observed that low available parallelism is a bottleneck but did not consider the case that even with high available parallelism, actual parallelism (occupancy) could still be very low that can lead to low performance. We studied GPU power consumption at detailed components level while previous work estimated GPU power consumption at very coarse-grained level [100, 176, 53]. For instance, Ma et al. [100] and Zhang et al. [176] reported power consumption for an entire GPU. Gebhart et al. [53] estimated total core power. While several works used workload metrics to understand their characteristics [73, 25, 57, 22], none of them studied the correlation between workload metrics and components power. We showed the existence of a correlation between workload metrics and components power and quantified the change in components power consumption with the change in workload characteristics.

State-of-the-art memory compression techniques for GPUs such as FPC [6], C-Pack [26], BDI [130] were originally targeted for CPUs and hence they exploited simple patterns for compression to keep the decompression latency low. Based on the observation that GPUs are less sensitive to latency than CPUs, we proposed a relatively more complex Entropy Encoding Based Memory Compression (E^2MC) technique which can also exploit complex patterns. We showed E^2MC delivers higher compression and performance gain than the state-of-the-art provided the key challenges of probability estimation, appropriate symbol length for encoding, and low decompression latency are addressed properly. E^2MC delivered 53% higher compression ratio and 8% increase in speedup compared to the state of the art and saved 13% energy and 27% EDP compared to no compression. While Huffman-based cache compression (SC^2) has been used for CPUs by Arelakis and Stenstrom [10], to the best of our knowledge no work has used Huffman-based memory compression for GPUs. Compared to SC^2, we showed a higher compression ratio and performance gain for 16-bit symbols instead of 32-bit symbols. Due to differences in the CPU and GPU architectures, the decompression latency of E^2MC is 4× more. Although GPUs can hide latency to some extent, we showed too much increase can also degrade their performance. Therefore, we reduced the decompression latency of E^2MC by parallel decoding with small loss of compression ratio ($< 4\%$).

We showed that memory compression is a promising alternative to increase memory bandwidth, however, we also observed that state-of-the-art lossless memory compression techniques including E^2MC often have a low effective compression ratio due to large memory access granularity (MAG) exhibited by GPUs. We further analyzed the MAG problem and quantitatively showed that low effective compression ratio due to MAG exists in four state-of-the-art techniques and qualitatively in three more techniques. We proposed a novel MAG aware Selective Lossy Compression (SLC) technique for GPUs to increase the effective compression ratio and performance gain. While approximate computing has been used for GPUs at different levels [141, 140, 145, 12, 175, 152, 91, 168,

Table 2.5: Summary of advances over the state of the art.

Field	State of the art	Advances over the state of the art
Power modeling and estimation	[61, 100, 166, 176, 165, 133, 92]	1. Hybrid power modeling improved flexibility compared to pure empirical approaches [61, 100, 166, 176] and accuracy compared to pure analytical approach [165] 2. No reference to power models for GPU-specific components and no discussion of power model validation in the existing hybrid approach [133] 3. Accurate power model validation methodology
Performance bottlenecks	[17, 100, 176, 53, 73, 25, 57, 22]	1. Identified key performance bottlenecks and quantified the effect of eliminating bottlenecks on performance and energy consumption 2. Used performance simulator for bottlenecks identification as well as for performance prediction unlike [17] 3. Previous work used workload metrics to understand their characteristics [73, 25, 57, 22], but none of them studied the correlation between workload metrics and components power consumption
Memory compression	[6, 26, 130, 10, 146, 89, 160, 77]	1. Proposed relatively more complex entropy encoding based memory compression technique (E^2MC) 2. E^2MC delivered 53% higher compression ratio and 8% increase in speedup compared to the state of the art 3. Addressed key challenges of entropy encoding for GPUs
MAG aware approximation	[141, 140, 145, 12, 175, 152, 91, 168, 102, 108, 45, 44]	1. Quantitatively [6, 26, 130, 85] and qualitatively [10, 9, 77] showed that state-of-the-art lossless memory compression techniques suffer due to the large MAG 2. Proposed a novel MAG aware approximation technique 3. First study that highlighted the importance of MAG aware approximation 4. Speedup of up to 35% with a maximum error of only 1.8% (state of the art treats 10% as acceptable error [108, 45, 44])

102], to the best of our knowledge no work has used MAG aware approximation. Moreover, this is the first study that highlighted the importance of MAG aware approximation for GPUs. SLC provided a speedup of up to 35% with a maximum error of only 1.8%. While 10% is treated as an acceptable average error in several related works [108, 45, 44], the

average error of SLC is less than 1%.

Table 2.5 succinctly summarized advances made by this thesis in the field of power modeling and estimation, performance bottlenecks analysis, memory compression and MAG aware approximation over the state of the art.

2.6 Summary

In this chapter, we reviewed the work related to this thesis and highlighted the advances made over the state of the art. We first provided a thorough review of GPU power modeling and estimation, including power model validation methodologies used by different researchers. We then reviewed the work related to GPU performance bottlenecks, components power consumption and their correlation with workload metrics. We surveyed lossless memory compression and approximate computing techniques for GPUs which are gaining importance due to i) difficulty in the traditional scaling of GDDR bandwidth ii) tremendous increase in the amount of data that needs to be processed in an energy efficient way. Finally, we summarized the advances made over the state of the art.

In the next chapter, we will provide an overview of a GPU architecture and data compression techniques for GPUs. As most of the research in this thesis is conducted using architectural simulator, we will also briefly discuss and list available GPU simulators.

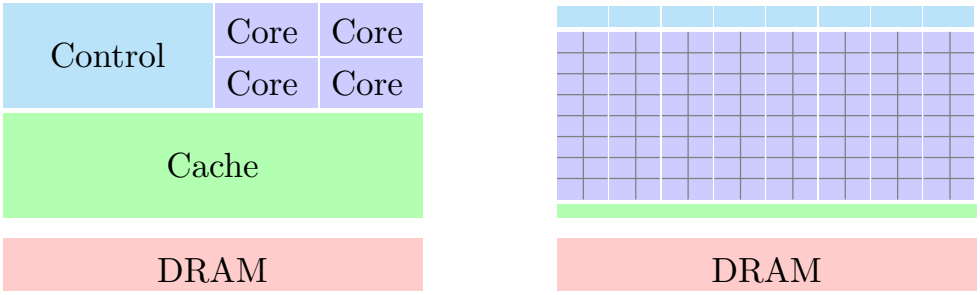
3 GPU Architecture and Compression Overview

In this chapter, we provide an overview of a GPU architecture and common terminology that will pave the foundation to understand the scientific results in the subsequent chapters. We also provide an overview of data compression which will be useful to understand the contributions in Chapter 6 and Chapter 7.

This chapter is organized as follows. Section 3.1 briefly discusses the evolution of general-purpose computing on GPUs. Section 3.2 presents high-level differences between a multicore CPU and a manycore GPU. Section 3.3 provides an overview of a GPU architecture including programming models. Section 3.4 gives a short overview of data compression. Finally, we summarize the chapter in Section 3.5.

3.1 Evolution of General-Purpose Computation on GPUs

Graphics Processing Units (GPUs) were initially designed to accelerate only graphics applications. The use of GPUs to process general-purpose applications that we traditionally executed on a central processing unit (CPU) became attractive due to their massive computational power. General-purpose computing on GPUs, commonly known as GPGPU, started in 2001 with the design of programmable shaders and support for floating-point operations. However, programming GPUs was not easy and required high effort as computational problems need to be reformulated in terms of graphics primitives. This means the initial use of GPUs was limited to expert GPU scientists. Programming GPUs became relatively easy with the removal of complex reformulation effort by the onset of general-purpose programming languages and APIs (Application Programming Interface) such as Sh [104] and Brook [21] in 2004. General-purpose computing on GPUs became further easy after the release of vendor-specific CUDA (Compute Unified Device Architecture) by NVIDIA in 2006, vendor-independent OpenCL (Open Computing Language) by Khronos Group, and DirectCompute by Microsoft in 2009. CUDA is proprietary and only supports NVIDIA GPUs, whereas OpenCL is an open standard. Microsoft's DirectCompute is also proprietary and bound to a single operating system. These parallel computing frameworks and APIs allowed programmers to ignore the underlying graphics concepts by offering more commonly used general-purpose notions and thus, made GPGPU much simpler. Since then GPGPU has truly taken-off with its ubiquitous usage in fields such as digital image processing, scientific computing, bioinformatics, computational finance, medical imaging, machine learning.



(a) High level CPU architecture (b) High level GPU architecture

Figure 3.1: CPU vs. GPU architecture. The figure is based on [80].

3.2 Multicore CPU vs. Manycore GPU

The two main computing styles in the heterogeneous computing are latency-oriented multicore CPUs and throughput-oriented manycore GPUs. While CPUs have been optimized for executing serial programs faster, GPUs have been optimized to deliver high throughput for parallel programs. For example, the peak performance of Intel's Core i7 8086K (Coffee Lake-S) is 177.94 GFLOPS, while the peak performance of NVIDIA's GTX-1080 (Pascal) is 8227 GFLOPS. Both are in the similar price range. The former launch price was 429$, while the later launch price was 549$.

The main reason for the large performance gap between a manycore GPU and a general-purpose multicore CPU is the fundamental difference in their design philosophies as shown in Figure 3.1. A contemporary multicore CPU consists of a few complex cores, with intricate control logic, multi-level large caches, and off-chip DRAM, while a contemporary GPU consists of a many simple cores, with simple control logic, one or two-level small caches, and a high bandwidth graphics DRAM. The complex cores and control logic of multicore CPUs improve single thread performance by supporting long pipelines with optimizations such as out-of-order execution, and instruction level parallelism. Moreover, the large caches ensure that most of the time the data is found close to the cores and very rarely data needs to be fetched from the off-chip DRAM. GPUs, on the other hand, have very simple control logic that is shared among several cores. For example, they have shorter pipelines, no out-of-order execution and speculation. GPUs employ Single Instruction Multiple Data (SIMD) execution model which means the instruction fetch, decode, and the issue units are shared among a SIMD unit. As GPUs have simpler front end and control logic, a large portion of chip area is dedicated to many cores. Thus, the peak performance of GPUs is far higher than CPUs. GPUs depend upon massive multithreading to keep the cores busy and deliver higher performance. The massive multithreading in GPUs makes it very hard to exploit the data locality and caches often suffer from thrashing [137]. Moreover, the caches in GPUs are much smaller than CPUs. For instance, L1 data cache per core in Intel's Core i7 8086K is 32KB, while L1 data cache per core in NVIDIA's GTX-1080 is only 384B. Because GPUs are massively multithreaded

processors and caches often have very low hit rates, they employ high bandwidth GDDR for off-chip memory instead of DDR to support a large number of threads. A GDDR is optimized for high throughput, thanks to wide buses, while a DDR is optimized for latency. Despite massive multithreading and high bandwidth off-chip memory, GPUs still run slower on some tasks for which CPUs have been optimized. Therefore, CPU-GPU heterogeneous computing has evolved in the last decade where a computationally complex part of the execution is offloaded on the GPU and the serial part is executed on the CPU [164].

3.3 GPU Architecture Overview

Figure 3.2 shows an overview of a typical NVIDIA GPU architecture. It consists of an array of highly threaded streaming multiprocessors (SM). Each SM consists of a set of simple cores. For example, Fermi architecture has 32 cores. Each core has a fully pipelined integer arithmetic unit (ALU) and floating point unit (FPU). In addition, each SM has four special function units (SFUs) that execute transcendental instructions such as sine, cosine, reciprocal. The number of cores is fixed in an SM in a given architecture, however, the number of SMs vary from a few in a low-end GPU to several for a high-end GPU. All cores in an SM share the same front-end which is responsible for instruction fetch, decode, and issue. Each SM has a shared memory and L1 cache. All SMs are connected to a shared L2 cache via an interconnection network. The L2 cache is partitioned into several banks, typically one bank per memory partition. Finally, the L2 cache is connected to the off-chip DRAM.

The basic organization is same across different generations of GPU architectures, however, there are significant changes in the number of cores in an SM and number of SMs in a GPU. For example, the Fermi architecture has 32 cores in an SM and 16 SMs in total, while the Kepler architecture has 192 cores and 15 SMs.

3.3.1 SIMT Execution

In a contemporary CPU, a significant amount of area and power are spent to support the executions of instructions. Each core in a CPU has the overhead of fetching, decoding, controlling and issuing instructions. This overhead is unnecessary when an application has significant data level parallelism because the same instruction is executed on different data. This style of computation is called Single Instruction Multiple Data (SIMD). A typical GPU architecture exploits data level parallelism to increase the area and power spent on actual computations than supporting instructions execution. However, instead of using vector instructions to exploit data level parallelism, a GPU employs multiple threads where each thread works on different data. Since the pipeline only has a single instruction, the same instruction is executed by all threads and this style of computation is called Single Instruction Multiple Thread (SIMT). Threads are typically grouped into units called warps in NVIDIA terminology and wave front in AMD terminology. A typical

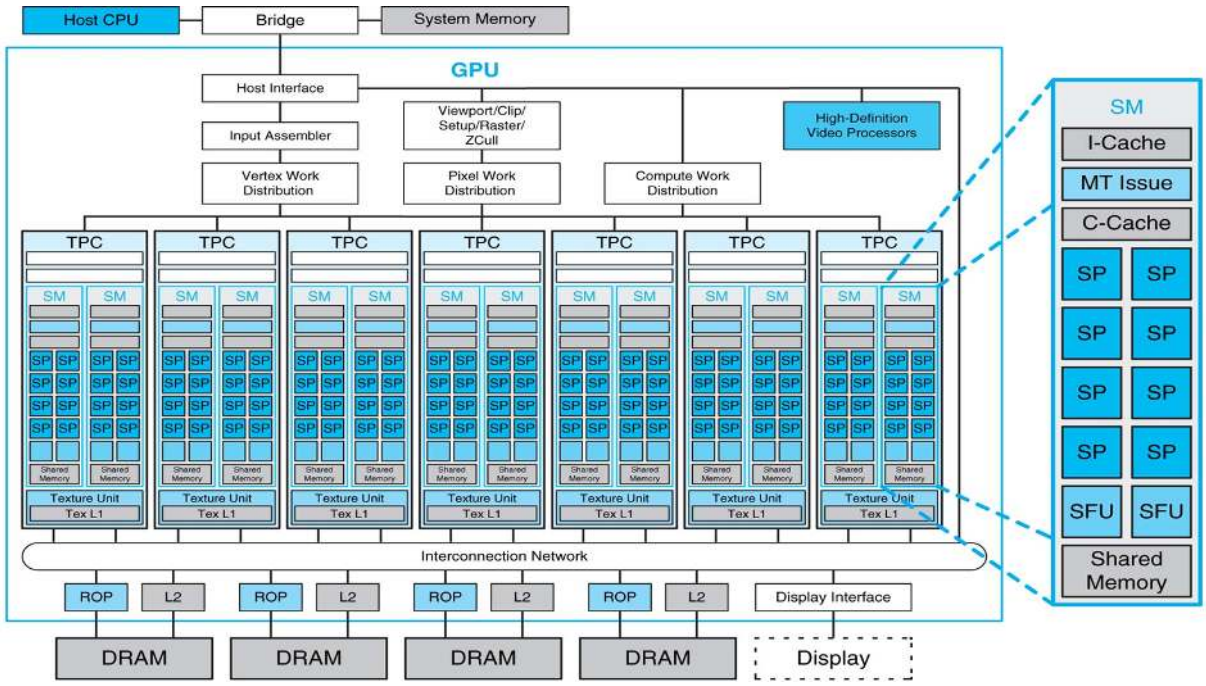


Figure 3.2: Overview of a contemporary NVIDIA GPU architecture [116]. © Elsevier 2012

warp consists of 32 threads. The size of a warp is implementation specific and it is not even part of the CUDA specifications. However, understanding of warps can be helpful in optimizing the performance of applications. A warp is a unit of scheduling in an SM and SIMT model is implemented at warp level. A warp executes the same instruction on different data. However, if needed different threads of a warp can also execute different instructions, leading to thread divergence. Because of SIMT execution, different instructions cannot be executed at the same time. Therefore, a thread masking mechanism is used to mask the threads that are not part of an instruction. Thread masking leads to idle cores corresponding to the masked threads. Therefore, a programmer optimizes applications to avoid a situation where different threads need to take different paths. However, thread divergence cannot be avoided always due to the irregular control flow of an application. For example, in breadth-first-search, thread divergence may occur.

3.3.2 GPU Memory Hierarchy

GPU memory hierarchy typically consists of a large register file, an on-chip programmer-managed scratchpad, known as shared memory in CUDA, hardware-managed caches (typically two levels), and high-bandwidth off-chip DRAM, also known as global memory. In addition, GPUs employ constant and texture memories for special purposes. Figure 3.3 shows an overview of memory hierarchy of Fermi architecture.

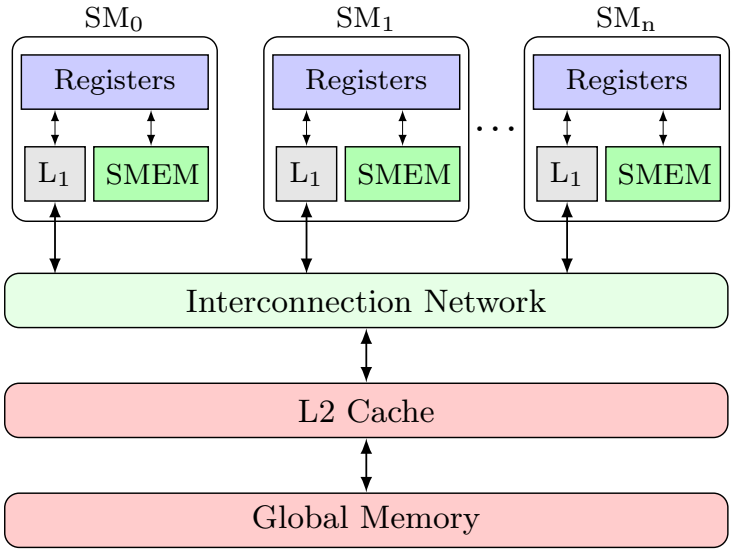


Figure 3.3: Overview of Fermi architecture memory hierarchy.

Register File: GPUs employ large register files to accommodate a large number of active threads and support fast context switching. For example, the maximum number of active threads in a Fermi architecture is 1536 and each thread can use a maximum of 63 registers. A GPU register file size is even larger than its L1 cache (see Table 3.1). The register file is multi-banked to provide high-bandwidth and low latency. Because of the large size and high-bandwidth oriented design, a register file contributes significantly to a GPU power consumption. For example, the register file power consumption is about 13.4% in GTX 480 [90]. Due to the high significance of the register file on a GPU performance, a lot of research has been done on the architecture and efficient management of the register file [103, 67, 110, 154, 153].

Shared Memory: Shared memory is allocated per thread block, so all threads in a thread block have access to the same memory. A thread block is a programming abstraction that represents a group of threads that can execute in parallel on different SMs. Shared memory is managed by a programmer and its efficient utilization depends upon the programmer and the nature of applications. It is suitable for applications for which data can be shared between threads of a thread block, for example, parallel reduction, matrix multiplication. In essence, shared memory acts as a programmer-managed cache as the data loaded by threads can be accessed by other threads of the same thread block. Similar to the register file, the shared memory is also banked and the maximum throughput is achieved when all threads access different banks. However, a conflict happens when two or more threads access different words in the same bank. The bank conflicts may lead to stall and performance degradation. Therefore, several techniques have been proposed to avoid bank conflicts [159, 58]. Shared memory is much faster than the global memory. The latency is about 100× lower than the latency of accessing the global memory, provided

there are no bank conflicts between the threads. On devices of compute capability 2.x and 3.x, each SM has 64KB of on-chip memory that can be partitioned between L1 cache and shared memory. For devices of compute capability 2.x, there are two settings, 48KB shared memory / 16KB L1 cache, and 16KB shared memory / 48KB L1 cache. By default the 48KB shared memory setting is used. Devices of compute capability 3.x allow a third setting of 32KB shared memory / 32KB L1 cache. The configuration of choice can be set using a runtime API.

L1 and L2 Data Cache: Traditionally, GPUs only have programmer-managed caches, however, with the advent of Fermi architecture, GPUs started using hardware-managed caches. In fact, the hardware-managed caches have accelerated the use of GPUs for general purpose computing. There are several works which compared the performance of Tesla and Fermi GPUs and reported that hardware-managed caches play an important role in higher performance of Fermi GPUs over Tesla GPUs [170, 63, 23]. However, GPU caches face different design challenges due to their different characteristics than CPU caches. For instance, write and allocation policies are quite different from CPU caches. L1 data cache is only write back for local accesses[1] and write evict for global accesses whereas in a CPU we have either write back or write through caches. Accordingly, allocation policy is usually no write-allocate for global accesses and write-allocate for only local accesses. The deviation in write and allocation policies is to cater to the different requirements of GPU workloads and smaller caches. GPU caches are shared by thousands of threads which make them a scarce resource and a victim of lot of contention. Furthermore, due to the streaming nature of many GPU applications and smaller cache size, caches can suffer from thrashing and high miss rate. Therefore, exploiting temporal locality is hard due to a large number of active threads and smaller caches. In fact, there are some works that reported negative performance results with caches [66, 134].

Thread-blocks are independent units of scheduling on SMs and a thread-block can be scheduled on any SM. This feature allows transparent scalability as simply more thread-blocks can be scheduled in parallel when more SMs are available and vice-versa. As thread-blocks can be scheduled on any SM, it is very hard to exploit inter-thread block locality at L1 caches because L1 cache is private to an SM. For example, when two thread-blocks have inter-thread block locality but they are scheduled on different SMs, there is no way to exploit inter-thread block locality at L1. L2 cache is useful for exploiting inter-thread block locality in this case as L2 cache is shared among all SMs. Therefore, we have a relatively large L2 cache with higher hit rate which helps to filter requests to off-chip memory. L1 and L2 caches on GPUs are smaller than L1 and L2 caches in CPUs, but they have high bandwidth.

Global Memory: The off-chip memory, also known as global memory, is the main memory of a GPU. A high-end GPU has several times the compute performance of a typical

[1] Local memory is private to each thread and is not visible to other threads.

CPU, but lacks deep cache hierarchy to support the memory requests. Therefore, compared to a CPU, a much larger fraction of memory requests are serviced by the off-chip memory. The off-chip memory in GPUs uses wider buses, several memory controllers to provide high-bandwidth. For example, Fermi has bus width of 384-bit, 6 memory controllers, and peak memory bandwidth of about 190GB/s.

Instead of standard Double Data Rate (DDR), GPUs deploy Graphic Double Data Rate (GDDR) for main memory. Although, both DDR and GDDR are designed from the Synchronous Dynamic-access Memory (SGRAM) and share some features such as double date rate, they differ significantly from each other. For example, GDDR is designed for high-bandwidth and DDR is optimized for latency. High bandwidth provided by successive generations of GDDR (GDDR/2/3/4/5/5X/6) memories has been a key factor in the performance scaling of GPUs. However, the later generations of GDDR have issues such as high power consumption, large form factor, difficulty in the scaling of pin count. The form factor plays an important role in determining the actual size of a GPU and vendors are struggling to keep GPU size small due to the large form factor of the GDDR. To mitigate the issues of GDDR, recently, 3D stacked DRAM technologies such as HMC (Hybrid Memory Cubes), HBM (High Bandwidth Memory) have been developed which have much smaller form factor and offer higher bandwidth and energy efficiency compared to GDDR. 3D stacked memories have their own problems such as much higher cost, significantly different interface, leading to adoption only in the high-end GPUs. However, these problems are expected to come down once the technology matures.

Local Memory: Local memory is private to a thread and is generally used for temporal spilling when there are not enough registers to hold all variables of a thread. It is also used when arrays are declared in a kernel but the compiler is unable to figure out the right indexing for them. Local memory is not a physical memory but a part of the global memory and it is cached in L1 and L2 in Fermi and Kepler architectures and only in L2 in Pascal and Maxwell architectures.

Constant Memory: As the name indicates, constant memory is used to hold the constant data of a kernel. Constant memory also resides in the global memory, like the local memory. However, unlike the local memory, it is not cached in L1 and L2 but in a separate cache known as constant cache. The constant cache is optimized to broadcast the data of a single memory address to all threads of a warp at the same time. Constant memory is fast as long as all threads of a warp access the same address, otherwise the accesses are serialized. Constant memory is a read-only cache for global memory and is managed by the compiler.

Texture Memory: Similar to local memory and constant memory, texture memory is also a part of a global memory, however, it is also buffered in a texture cache which is specially optimized for 2D spatial locality. Moreover, a GPU provides hardware support for address calculation for texture accesses, thus, adding extra compute power. Texture

Table 3.1: Summary of memory hierarchy of NVIDIA's different architectures.

Arch.	Representative GPU	RF (KB)	Shared memory (KB)	L1 (KB)	L2 (KB)	Const (KB)	Texture (KB)	BW (GB/s)
Tesla	GTX-8800	64	16	N/A	N/A	8	12	86.4
Fermi	GTX-580	128	16/48	16/48	768	8	12	197.6
Kepler	GTX-780	256	16/32/48	16/32/48	1536	8	48	288.4
Maxwell	GTX-Titan X	256	64	24	2058	10	Unified	336.0
Pascal	GTX-1080	256	64	48	4096	10	Unified	352.0
Volta	TITAN V	256	0/8/16/32/64/96	128	6144	64	Unified	652.8

memory is designed for streaming fetches with a constant latency. A texture cache hit reduces off-chip memory bandwidth, but not fetch latency [118]. Texture cache is also a read-only cache and starting from Maxwell, it is unified with L1 data cache.

In summary, GPU memory hierarchy is not so deep but wide and is designed to support high throughput rather than low latency. GPU design philosophy to hide long latency is to use fine-grained multi-threading and switch to a different group of threads whenever a long latency access occurs. In general, efficient use of shared memory, hardware-managed caches can reduce pressure on the high-bandwidth off-chip memory by reducing traffic. However, if the memory access rate is too high, the memory system cannot service the memory requests at the required rate, leading to GPU stalls. In general, efficient utilization of memory bandwidth is a challenge for application developers of throughput-oriented systems. Moreover, due to high power consumption and physical limitations of chip-packaging, increasing memory bandwidth by traditional ways of increasing clock-frequency or pin-count is extremely challenging. Therefore, techniques which can reduce off-chip memory traffic such as efficient utilization of shared memory, better cache management policies, and alternative techniques to increase memory bandwidth such as memory compression, approximate computing will play an important role to full fill the memory bandwidth demands of future high throughput systems.

Table 3.1 summarizes the size of register file (RF), shared memory, L1 data cache, L2 data cache, constant cache, texture cache and global memory bandwidth for six generations of NVIDIA architectures. In Maxwell and Pascal architectures, the texture cache is unified with L1 data cache. Volta architecture has unified shared memory, L1 data cache and texture cache. The combined size is 128KB. The size of L1 data cache and texture cache depends on shared memory configured size.

3.3.3 Programming GPUs

To program massively multi-threaded GPUs, new programming models have been developed. The two most famous programming models are CUDA [123] and OpenCL [74]. Both CUDA and OpenCL are extensions to C programming language which provide APIs

and runtime to allow programmers to harness parallelism to achieve higher performance. There are alternatives to CUDA and OpenCL such as directive based languages, e.g., pragmas with C/C++, OpenACC and OpenMP. CUDA is vendor specific and is only supported by NVIDIA GPUs, while OpenCL is vendor neutral and is supported by all GPUs, typically including mobile GPUs. OpenCL is a standard programming model developed by Khronos Group. Thus, an application developed in OpenCL can run without modification on all processors that support the OpenCL language extensions and API.

```
CPU serial code

GPU parallel kernel
kernel1 <<< num_blocks, num_threads >>> (args);

CPU serial code

GPU parallel kernel
kernel2 <<<  num_blocks, num_threads >>> (args);
```

Listing 3.1: Execution of a CUDA program.

Since GPUs are used as co-processors to accelerate the computationally intensive part of applications, a typical GPU application has a host code and a device code. The device code is called kernel and is executed on a GPU. Usually the serial part of an application is executed on the host and parallel part of the application is executed on the device. The host code runs on a CPU and is also responsible for allocating memory, transferring data to the GPU before the start of a kernel execution, and transferring the data back to the host at the end of a kernel execution. A typical CUDA application execution flow is shown in Listing 3.1. The execution starts from the host. The execution moves to the device when a kernel function is launched. When all threads in a kernel finish, the execution again moves back to the host. The execution continues on the host until another kernel function is launched. This process continues until the end of an application.

In CUDA, a kernel function specifies the code to be executed by all threads. As all threads execute the same code, CUDA programming is an example of a popular Single Program Multiple Data (SPMD) style of programming massively parallel systems. Listing 3.3 shows a simple kernel for adding two vectors. A kernel function starts with the keyword *__global__*. The keyword means the function is a CUDA kernel and it can be called from the host to generate a grid of threads to be executed on the device. There are three other keywords: *threadId.x, blockId.x* and *blockDim.x* shown in the listing. These keywords are built-in variables and refers to hardware registers which help a thread to identify its coordinate at runtime. *threadId.x* and *blockId.x* gives the thread index and block index, while *blockDim.x* gives the block dimension. A CUDA programmer only needs to specify the computation, while threads generation and parallel execution is handled by the runtime and underlying hardware. This helps simplifying the programming of a GPU. For example, although the vector addition is done in parallel by many threads, the computation is specified by a simple addition of two vectors as shown in Listing 3.3.

```
int call_kernel(int *A, int *B, int *C, int num_blocks,
int num_threads) {
   vector_add<<<num_blocks, num_threads>>>(A, B, C);
   return 0;
}
```

Listing 3.2: Kernel call from host.

```
__global__ void vector_add(int *A, int *B, int *C) {
   int tid = blockIdx.x*blockDim.x + threadIdx.x;
   C[tid] = A[tid] + B[tid];
}
```

Listing 3.3: Kernel definition.

As briefly discussed before, a kernel needs to be launched from the host to execute it on the device. A simple example to launch a kernel is shown in Listing 3.2. The key aspect of kernel launch is the thread organization that is specified at the kernel launch time using the syntax $<<< num\_blocks, num\_threads >>>$. For data mapping and better processing such as scalability, scheduling, threads are grouped into thread blocks. In the Listing 3.2, *num_threads* is the number of threads in each thread block. The number of threads in a thread block is restricted, for example, for Fermi architecture, the number of threads in a thread block is 512. Threads in the same thread block run on the same SM and they can communicate via shared memory. As the number of threads in a thread block is restricted, multiple thread blocks are created to map the computation that require a large number of threads. The array of thread blocks is called *grid*. In the Listing 3.2, *num_blocks* represents the number of blocks in a grid. Both *num_blocks* and *num_threads* are 3-D variables, enabling different organization of threads according to an application suitability. The x, y, z co-ordinates of these variables can be accessed using the built-in variables, for example, the co-ordinates of a thread block are accesses using *threadId.x, threadId.y,* and *threadId.z*.

In this thesis, we use CUDA terminology, however, we briefly discuss OpenCL equivalent terminology of CUDA to help a reader familiar with OpenCL terminology. Table 3.2 shows the CUDA and OpenCL terminology for commonly used terms. A *thread*, *warp*, *thread block* and *grid* is called *work item*, *wave front*, *work group* and *computation domain* in OpenCL. A reader may get confused with the *local memory* and *shared memory* terms if he/she is only familiar with one of the terminologies. In CUDA, *shared memory* refers to a small on-chip memory that is shared by threads in a thread block whereas its equivalent is known as *local memory* in OpenCL. In CUDA, *local memory* is the per-thread memory and its equivalent is known as *private memory* in OpenCL. Off-chip memory is known as *global memory* in both CUDA and OpenCL. A *scalar core* is equivalent to *processing element* while a *streaming multiprocessor* is equivalent to *compute unit* in OpenCL.

Table 3.2: CUDA vs. OpenCL terminology.

CUDA	OpenCL	CUDA	OpenCL
Thread	Work item	Local memory	Private memory
Warp	Wave front	Shared memory	Local memory
Thread block	Work group	Global memory	Global memory
Grid	Computation domain	Scalar core	Processing element
Streaming multiprocessor	Compute unit	-	-

3.3.4 GPU Simulators

An architectural simulator is a piece of software that is used to model and predict the performance of computer devices. gpgpu-sim [15] is the most widely used and accepted general-purpose computing on GPUs (gpgpu) simulator. gpgpu-sim is a cycle-accurate simulator which means it simulates a microarchitecture on a cycle-by-cycle basis. A cycle accurate simulator is much slower than a functional simulator. However, a functional simulator can only simulate the functional correctness and it has no notion of performance such as execution time. Barra [31], Ocelot [40], and Multi2Sim [158] are alternative simulators to gpgpu-sim. Barra is a GPU functional simulator that simulates the native instruction set of the Tesla architecture. Ocelot [40] is a dynamic compilation framework designed to map the NVIDIA Parallel Thread eXecution ISA (PTX), an intermediate language onto diverse multithreaded platforms. Ocelot includes a dynamic binary translator for PTX to many-core processors that leverages the Low Level Virtual Machine (LLVM) code generator to target x86 and other ISAs. The dynamic compiler is able to execute existing CUDA binaries without recompilation. Multi2Sim is an heterogeneous simulator that could simulate an x86 CPU and an AMD Evergreen GPU in the initial release. However, it was later extended to support the simulation of ARM, and MIPS CPUs. Recently, Multi2Sim was again extended to support the simulation of NVIDIA's Kepler architecture [56].

Most of the GPU microarchitecture ideas in the last decade have been implemented and evaluated using gpgpu-sim. It has more than 1100 references to this day which is an indicator of its popularity. However, gpgpu-sim is also getting out of date because it has been validated using NVIDIA's Tesla and Fermi architectures and NVIDIA has released four more architectures after Fermi and gpgpu-sim is not modified to support them. Recently, an open source RTL implementation of the AMD Southern Islands GPGPU ISA, capable of running unmodified OpenCL-based applications was published [16]. However, not much research has been reported using it until now. Table 3.3 summarizes the key features of the GPU simulators.

Table 3.3: Summary of GPU simulators.

	gpgpu-sim	Barra	Ocelot	Multi2sim	MIAOW
Release date	2009	2010	2010	2012	2015
Platforms	NVIDIA	NVIDIA	NVIDIA	AMD	AMD
Architectures	Tesla, Fermi	Tesla ISA	PTX ISA	Evergreen	Southern Islands
Functional only	No	Yes	Yes	No	No
Cycle accurate	Yes	No	No	Yes	Yes (RTL level)

3.4 Data Compression Overview

In this section, we provide an overview of data compression and compression techniques that we use as baselines to compare our proposed memory compression techniques in Chapter 6 and Chapter 7. Data compression encodes information using fewer bits than the original representation. Compression can be either lossless or lossy. Lossless compression identifies the redundant information to reduce the bits required to represent the original information. As lossless compression only removes the redundant information, the original information can be retrieved without any loss after decompression. In contrast to lossless compression, lossy compression also removes less important information and hence, there is some loss of information.

Most of the real-world data has statistical redundancy, which is exploited by the lossless compression techniques to represent data with fewer bits and without any loss of information. For example, images usually have areas where several pixels have the same color. For such areas, instead of coding the same color repeatedly n times, the color may be encoded once along with the count n. Another example of common redundancy is the bulk initialization of many applications with initial values such as zeros. There are many techniques to exploit such redundancy present in the data to reduce the number of bits to store them efficiently.

Lossless compression techniques can be classified into two types: *fixed-width coding* (FWC) and *variable-width coding* (VWC) [77]. FWC compresses *fixed-length* symbols by using *variable-length* output codes. The most frequent symbols are encoded with the shortest codes to reduce the average number of bits required to store all symbols. For example, Huffman coding assigns code words depending on the probability of symbols. The length of an output code is approximately equal to the negative logarithm of its probability [150]. The maximum compression ratio is bounded by the entropy of input data. The Shannon entropy [150] of an input set of symbols A = (a_1, a_2, ..., a_n) with a probability of occurrence P = (p_1, p_2, ..., p_n) is defined as:

$$H(A) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{3.1}$$

Data with lower entropy has better compressibility because a small number of symbols occurs frequently, whereas data with higher entropy is hard to compress. Compression also depends on the symbol size. Larger symbols can provide higher compression ratio because larger symbols can be encoded with fewer bits. Larger symbols are better when the input data has low entropy, however, larger symbols may also not work when the entropy is high as it becomes difficult to find redundancy with larger input symbols. For instance, floating point values have higher entropy compared to integers values. That is why several compression techniques break the floating point number into different sub-fields such as mantissa, exponent and compress them separately [9, 29, 55, 156]. In addition to compressibility, symbol size also determines codewords table size. In general, the number of entries in a codeword table is equal to 2^n, where n is the number of bits in a symbol. For a large symbol size, for instance 32-bit symbols, the number of entries is very large and storing all of them is not practical. Therefore, larger symbols present the challenge to keep the codewords table size reasonable and feasible, making the selection of an appropriate symbol size an important task in a FWC [85].

VWC uses input symbols of different width to achieve higher compression ratio while limiting the table size. The *run length encoding* (RLE) and *Lempel-Ziv* (LZ) are two well-known compression methods that use variable width encoding. Typically, VWC uses either static or dynamic dictionary or a combination of both to store the frequently occurring patterns and encode them with fewer bits. For example, C-PACK [26] uses a combination of static patterns and a small dynamic dictionary. Usually, the anticipated frequent patterns such as consecutive zeros, one byte zero-extended words get static codes, while a dynamic dictionary helps to compress the not so frequent patterns. A dynamic dictionary plays an important role in VWC as it captures the spatial locality which is very difficult to exploit statically. In the next subsections, we provide an overview of compression techniques used as baseline in this thesis.

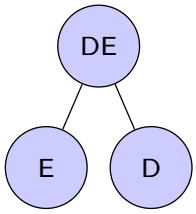
3.4.1 Huffman Encoding

Huffman encoding, also known as Huffman compression, is based on the evidence that not all symbols have same probability. Therefore, instead of using fixed-length codes, Huffman encoding uses variable-length codes based on the relative frequency of symbols. A fixed-length code assumes equal probability for all symbols and hence assigns same length codes to all symbols. For example, ASCII encoding uses 8-bit to encode each symbol. In contrast to fixed-length codes, Huffman encoding is based on the principle to use fewer bits to represent frequent symbols and more bits to represent infrequent symbols. Huffman encoding derives the length of codewords by constructing a Huffman tree based on the frequency of input symbols.

Figure 3.4 shows a step by step example of constructing a Huffman tree. The input is a list of symbols sorted in increasing order of their probabilities as shown in Figure 3.4a. In each step, two nodes with the least probabilities are removed from the list and inserted into the tree connected to an internal node with frequency equal to the sum of the frequencies

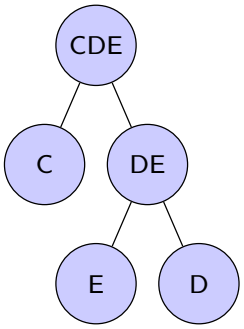
Symb.	Prob.
A	0.40
B	0.25
C	0.20
D	0.10
E	0.05

(a) Input symbols with probability



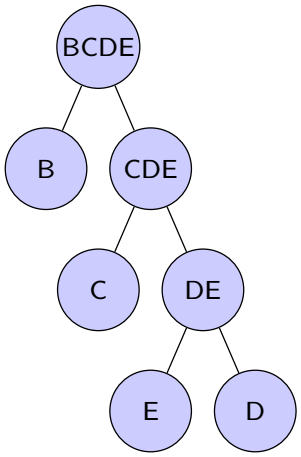
Symb.	Prob.
A	0.40
B	0.25
C	0.20
DE	0.15

(b) First step



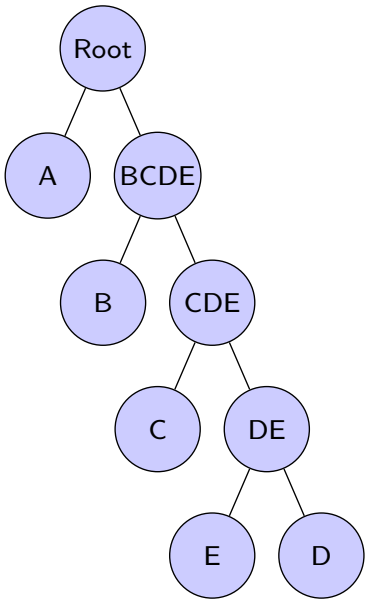
Symb.	Prob.
A	0.40
CDE	0.35
B	0.25

(c) Second step



Symb.	Prob.
BCDE	0.60
A	0.40

(d) Third step



(e) Huffman tree

Symb.	Code
A	0
B	10
C	110
D	1110
E	1111

(f) Codewords

Figure 3.4: A stepwise example of constructing a Huffman tree.

of two symbols removed from the list. The internal node is also inserted in the list at its sorted position. For example, E (0.05) and D (0.10) are the two nodes with least probabilities as shown in Figure 3.4b. These two nodes are removed and connected to an internal node DE with frequency 0.15 and the internal node DE is inserted into the list as shown in Figure 3.4c. The process is repeated until there is only one node left in the list with frequency equal to the sum of the frequencies of all input symbols. This is the root node of the Huffman tree. The input symbols are the leaf nodes of the Huffman tree as shown in Figure 3.4e.

Once a Huffman tree is constructed, a codeword for each symbol is derived by traversing the tree. To assign codewords, a zero or one is assigned to each branch of the tree. Usually, 0 is assigned to each left branch of the tree and 1 is assigned to each right branch of the tree. A codeword of a symbol is derived by concatenating the 0's and 1's while traversing a Huffman tree starting from the root node to the leaf node representing the symbol. For example, the symbol B gets the codeword 10 as we need to take a right branch and then left branch to reach the leaf node representing symbol B from the root node. The output of traversing the whole tree gives the codewords for all input symbols as shown in Figure 3.4f. A Huffman code has a prefix property which means that there is no codeword which is a prefix of any other other codeword. Moreover, a Huffman code is optimal which means that there is no other way of assigning codewords with minimal average length.

Shannon's source coding theorem showed that the smallest possible codeword length is given by Shannon entropy for a given input symbols with probabilities. For the symbols and codewords shown in Figure 3.4, the Shannon entropy is 2.04 bits per symbol, whereas the weighted average codeword length is 2.10 bits per symbol. The entropy is calculated using Equation 3.1. The weighted average codeword length is calculated using Equation 3.2. l_i is the length of a Huffman codeword assigned to a symbol a_i with probability p_i. We see that the weighted average codeword length is only slightly larger than the calculated entropy. Therefore, the Huffman code is not only optimal prefix code, but also very close to the theoretical limit established by Shannon.

$$L(A) = \sum_{i=1}^{n} l_i p_i \tag{3.2}$$

In general, compression techniques using variable-length codes can provide high compression ratio, but they have certain challenges which need to be addressed [10]. For instance, symbol length plays an important role as discussed before. To generate Huffman codes, we need to find the probability of input symbols. Huffman encoding estimates the probability of input symbols either statically or dynamically. In static probability estimation, an extra pass over the input data is needed to find the probability and the encoding is done in the second pass. In dynamic probability estimation, the probability is estimated by sampling the part of input for some time and then encoding is performed on the remaining input. The trade-offs of static versus dynamic probability estimation and required sampling duration are discussed in detail in Chapter 6.

Table 3.4: Frequent Pattern Encoding [6].

Prefix	Pattern Encoded	Data Size
000	Zero run	3 bits (runs up to 8 zeros)
001	4-bit sign-extended	4 bits
010	One byte sign-extended	8 bits
011	Halfword sign-extended	16 bits
100	Halfword padded with a zero halfword	16 bits (the non-zero halfword)
101	Two halfwords, each a byte sign-extended	16 bits (the two bytes)
110	Word consisting of repeated bytes	8 bits
111	Uncompressed word	32 bits (original word)

3.4.2 Frequent Pattern Compression (FPC)

FPC [6] is a significance-based compression that uses the observation that some data patterns occur more frequently and they can be compressed using fewer bits. A significance-based compression exploits the observation that most values (e.g., 32-bit integers) can be stored in a fewer number of bits and instead of storing all 32-bits, it only stores the required number of least significant bits. For example, many narrow-value integers can be stored in 4, 8, or 16 bits, but they are normally stored in a full 32-bit word (or 64-bit for 64-bit architecture). The research shows that these values occur frequently, thus, storing them efficiently can save cache capacity, memory bandwidth etc. Moreover, FPC also exploits the observation that runs of zeros are very common and stores them only once along with their count.

FPC compresses data on a word-by-word (32-bit) basis. It stores most frequent patterns using a 3-bit prefix code along with the least significant bits required to represent the data. Table 3.4 shows the 3-bit prefix for the most frequent patterns. FPC can compress a 32-bit word that is 4-bit sign-extended to 4 bits, one byte sign-extended or word consisting of repeated bytes to 8 bits, halfword sign-extended or halfword padded with a zero halfword or two halfwords (each a byte sign-extended) to 16 bits. In essence, FPC is a static dictionary based scheme that uses prefix and pre-determined patterns to achieve compression. An advantage of FPC is low compression and decompression latency because it exploits simple patterns for compression.

Figure 3.5 shows a simple example of compressing an input of 4 words using FPC. The first word is a 4-bit sign extended, the second word is a one byte sign-extended, the third word is a halfword padded with a zero halfword, and the last word consists of repeated bytes. FPC will store them in a compressed form as shown in Figure 3.5. For example, the first word is stored as 3-bit prefix and 4 bits of data ((0b001)0xC). The second word is also stored with a 3-bit prefix and 8-bits of data ((0b010)0x8E) and so on. For this

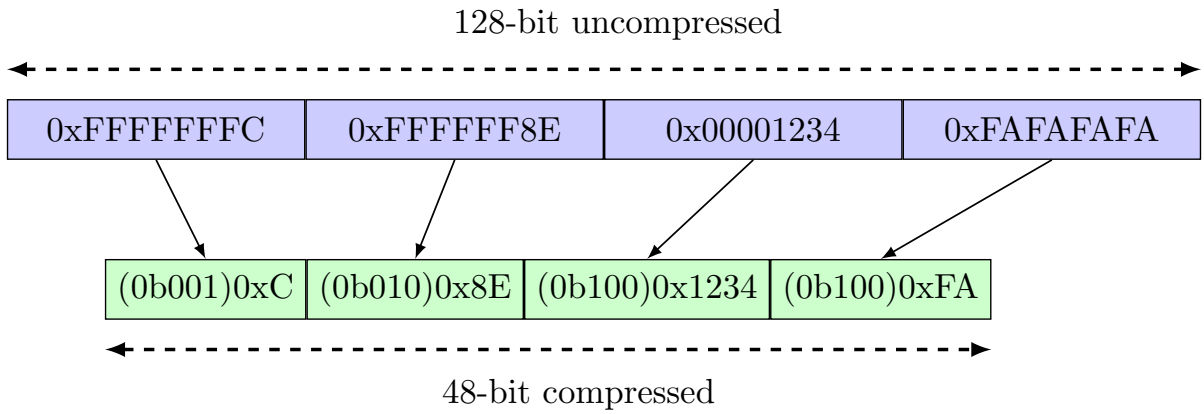


Figure 3.5: An example of FPC technique [144].

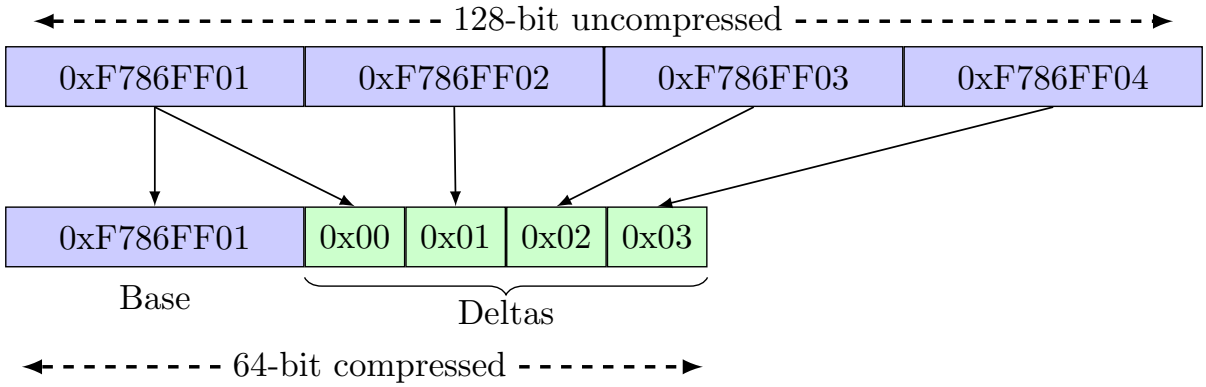


Figure 3.6: An example of BDI technique with one base.

simple example, the data can be stored in 48 bits instead of 128 bits.

3.4.3 Base-Delta-Immediate Compression (BDI)

BDI compression [130] is a simple and low latency compression technique for on-chip caches and memory [160]. BDI compression is based on the observation that values stored in a cache line have high value similarity and low dynamic range i.e., the relative difference between the values is small. In such a case, a cache line can be represented using a common base and small differences as deltas. Instead of a single base, multiple bases can be used to increase the compression ratio. For example, BDI with a single base will fail to compress a cache line even if only one of the value cannot fit in the delta. Using different base values, BDI compression scheme can compress data with different dynamic ranges. The authors show that two base values work best.

Figure 3.6 shows a simple example of BDI compression using a single base. Although, the uncompressed values are large, the relative difference is very small. The first value

Table 3.5: C-PACK Pattern Encoding [26].

Prefix	Pattern Encoded	Output Codeword	Length
00	zzzz (zero word)	(00)	2b
01	xxxx (uncompressed word)	(01)BBBB	34b
10	mmmm (matched a dictionary entry)	(10)bbbb	6b
1100	mmxx (matched 2 bytes with a dictionary entry)	(1100)bbbbBB	24b
1101	zzzx (one byte zero-extended)	(1100)B	12 b
1110	mmmx (matched 3 bytes with a dictionary entry)	(1110)bbbbB	16 b

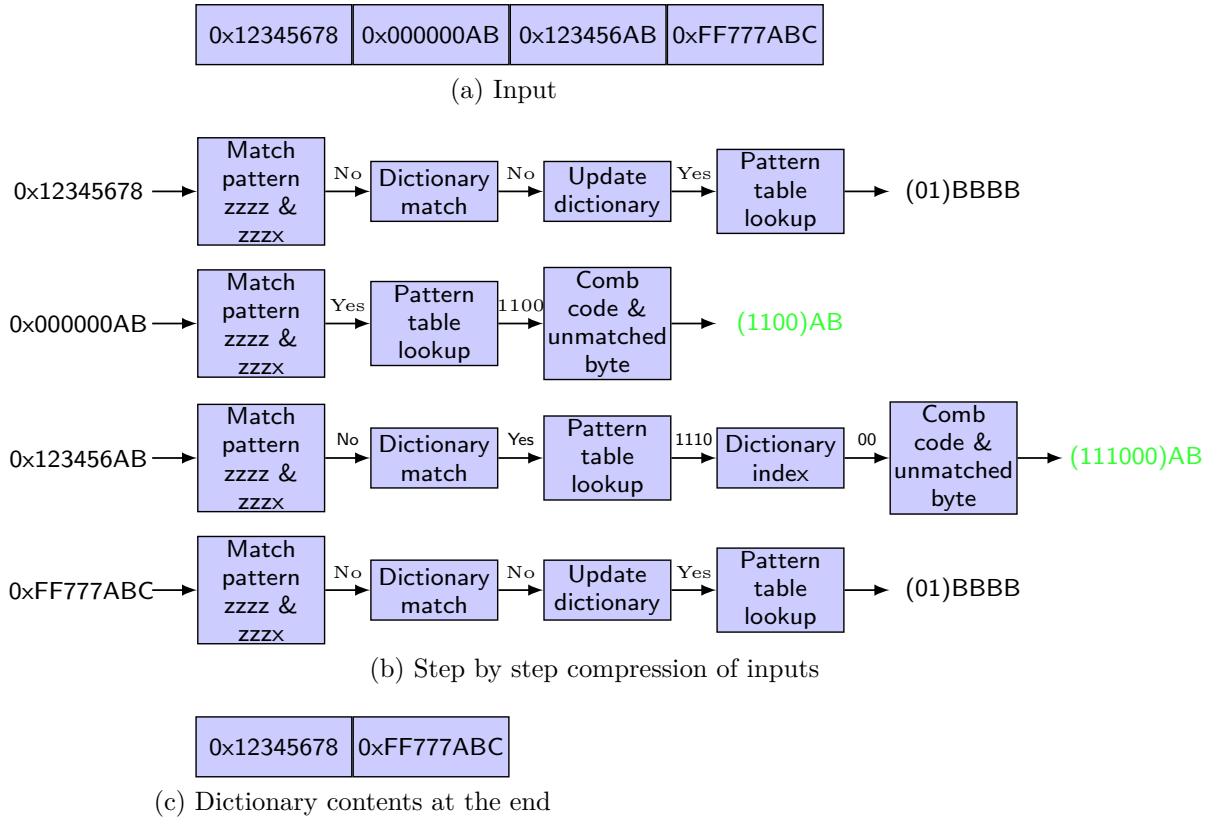
is chosen as the base and the differences are stored as deltas. As deltas require less space, we can achieve significant compression. Determining the optimal number of bases is complicated. A typical implementation of BDI uses two base values: zero and first non zero value in the input. This simple selection of base values allows parallel decompression by just adding deltas to base values, resulting in a very low decompression latency.

3.4.4 Cache Packer (C-PACK)

C-PACK [26] is a compression technique that has been used for compression at different levels in memory hierarchy [146]. C-PACK is a mixture of significance based compression and dictionary based compression. Like FPC, C-PACK also operates on 4-byte word-by-word basis and compresses the frequently occurring patterns such as zeros, one byte sign-extended by fewer bits. In addition to encoding frequently occurring patterns with fewer bits, it also uses a small dynamic dictionary to compress repeated words including partial matches. Table 3.5 shows the prefix, the encoded pattern, output codeword and output length in bits. *zzzz*, *zzzx*, and *xxxx* represent a zero word, a word with one-byte sign extended, and uncompressed word, respectively. *mmmm* means a complete match with a dictionary entry, while *mmmx* and *mmxx* mean 3 bytes and 2 bytes matched with a dictionary entry. C-PACK [26] used 16 entries for the dictionary, thus, an output codeword has 4-bit for the dictionary index. C-PACK initializes the dictionary to zero and builds it up during compression.

Figure 3.7 shows a simple example of compression using C-PACK. Figure 3.7a shows the 4 input words. Figure 3.7c shows the contents of the dictionary at the end of compressing all inputs. The dictionary is initially empty. A value is inserted into the dictionary when there is no match with the fixed patterns as well as no match with any existing values in the dictionary and the dictionary is not full. C-PACK uses dictionary size of 16.

Figure 3.7b shows stepwise processing of four input words. The C-PACK works as follows. The input (0x12345678) is first checked for a frequent pattern match and then for a dictionary match. As there is neither a pattern match nor a dictionary match, the word is stored uncompressed with its prefix from the pattern table. The dictionary is



(a) Input

(b) Step by step compression of inputs

(c) Dictionary contents at the end

Figure 3.7: An example of C-PACK [26].

updated with the input word (0x12345678) as shown in Figure 3.7c.

The second input (0x000000AB) matches the pattern *zzzx*. Thus, the pattern table is checked for its code. The code along with the unmatched byte is stored. The third input (0x123456AB) has no match with the static patterns, but it has a 3-byte match with a dictionary entry. The pattern table is looked for the prefix code. Thus, C-PACK outputs a combined prefix code, a dictionary index, and an unmatched byte. As C-PACK used dictionary size of 16, we use 4-bit index in the output codeword whenever a dictionary match happens. The last input (0xFF777ABC) does not match with any static pattern as well as any entry in the dictionary. The dictionary is updated with the input word as shown in Figure 3.7c and the prefix for uncompressed word along with the word is produced as output.

For decompression, C-PACK needs to rebuild the dictionary. This is done by starting with a dictionary that is initialized to zero values and then constructing the dictionary backward as the codewords are processed. An advantage of C-PACK is that it uses a combination of static patterns and dynamic dictionary, enabling the compression of frequent patterns found in whole input data as well as the patterns which only repeat in a single block. In general, a dynamic dictionary based compression technique can exploit redundancy that is hard to determine statically. However, dictionary based algorithms

usually have higher compression and decompression latency compared to non-dictionary based algorithms such as BDI. More detailed information on data compression can be found here [144].

3.5 Summary

In this chapter, we gave an overview of GPU architecture and data compression for GPUs. We highlighted the differences between CPU and GPU architectures, briefly discussed the SIMT execution model and wide memory hierarchy of GPUs. We also provided a short introduction to GPU programming models and briefly highlighted the key features of GPU architectural simulators. Finally, we discussed data compression, in particular, we described the memory compression techniques that we use as baseline techniques to compare our work presented in Chapter 6 and Chapter 7.

In the next chapter, we will present a novel power simulator for GPUs which is the first main contribution of the thesis. This work is done in collaboration with others as described in Section 4.1.

4 GPU Power Modeling

The work presented in this chapter was previously published: 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 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), © 2013 IEEE.

In this chapter, we present a novel power simulator for GPUs called *GPUSimPow*. The power simulator serves multiple purposes towards conducting the research presented in the thesis. First, we use GPUSimPow to study the energy efficiency of GPU workloads and understand bottlenecks that lead to low performance and low energy efficiency. Second, we use GPUSimPow to estimate the energy benefits of architectural techniques proposed in the thesis to improve the energy efficiency of GPUs.

4.1 Introduction

With processor design becoming more and more complex and chip manufacturing processes getting smaller and smaller, the inability of computer architects to produce working prototypes of their designs for testing is a more pressing problem than ever before. As chips are rapidly approaching (and nowadays touching) the power wall, the conventional design space of a processor architecture has been extended by another dimension: energy efficiency. To optimize a processor for energy efficiency requires a detailed study of energy-performance trade-offs in all aspects of the processor design space, including both architectural and circuit design choices.

Over the past years, it has become apparent that the chips consuming the most energy are modern GPUs. With GPUs turning into major devices for general-purpose computing, also known as general-purpose computing on GPUs (GPGPU), more and more vendors are striving to drive GPUs performance up. The inability to manufacture chips to evaluate architectural design choices, however, remains a problem as does the looming power wall.

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. GPU architects rely on building cycle-accurate architectural simulators [15, 158] in high-level languages and evaluate novel designs using these simulators. On the other hand, if we only consider *CPUs*, there are several accepted tools and frameworks to model and estimate power consumption such as Wattch [20]. To the best of our knowledge, however, no one

ever before combined an architectural GPU simulator with a power model to create a *GPU power simulator*.

In this work, we seek to mitigate this issue by designing *GPUSimPow*, a power simulator for GPGPU architectures. GPUSimPow is a highly parameterizable simulator that is able to provide an accurate estimate of area, static power and dynamic power of GPGPU workloads. With this power simulator, computer architects can evaluate their design choices early from a power perspective, and programmers can gain significant insights of their programs (*kernels*) to optimize power consumption from a software perspective. To make GPUSimPow flexible enough such that alternative architectural design choices can be explored while still maintaining a reasonably high accuracy, we model the components of a GPU architecture in two ways. Regular components such as memories are modeled *analytically* using the well-known McPAT [92] tool which is based on CACTI 6.5 [155]. Irregular components such as address generation units (AGUs), special-function units (SFUs) are modeled *empirically* by acquiring measurement data from real hardware. We propose a custom *power measurement testbed* to validate our proposed power simulator and derive empirical power models of some GPU components.

In summary, this chapter makes the following main contributions:

- We develop a GPU power simulator that is able to provide an accurate estimate of the area, static power and dynamic power for GPGPU microarchitectures and GPGPU kernels. Our evaluation on a set of well-known benchmarks shows an average relative error of 11.7% and 10.8% between simulated and measured power for GT240 and GTX580, respectively.

- We use hybrid approach to model power which provides both flexibility and high accuracy to conduct architectural research from power and energy perspective.

- We propose a novel power measurement testbed to accurately measure GPU power consumption on real hardware down to the individual kernel.

GPUSimPow was developed jointly by Jan Lucas, the author of this thesis, Michael Andersch, Mauricio Alvarez Mesa, and Ben Juurlink. Text and figures by Jan Lucas and Michael Andersch are also included in this thesis with their permission to provide a full overview and evaluation of the GPUSimPow simulator. Jan Lucas developed a power model for register file, empirical power models for execution units, core and cluster power and adapted network-on-chip power model from McPAT. He also developed power measurement testbed used for the validation of the power simulator and development of the empirical power models. Michael Andersch developed power models for load-store unit, texture cache, and warp control unit. The author of this thesis developed power models for shared memory, off-chip memory, special-function units, coalescing unit, and caches. In addition, the author also modified architectural simulator, gpgpu-sim [15], to generate activity factors for various components. Ben Juurlink and Mauricio Alvarez Mesa supervised the work and helped to improve the quality of the writing.

This chapter is organized as follows. Section 4.2 presents the detailed design of the power simulator. Section 4.3 describes the power measurement setup used to validate the power simulator and derive empirical power models for some GPU components. In Section 4.4, we present results including a quantitative comparison with GPUWattch [90]. Finally, we summarize the contributions of this chapter in Section 4.5.

4.2 Design of GPUSimPow

GPUSimPow is a power simulator for GPGPU workloads, i.e., given a configuration of a GPU architecture and a GPGPU kernel written in CUDA [123] or OpenCL [74], GPUSimPow is capable of producing both architectural information such as area, static power and dynamic power for the executed kernel. The simulator is designed to be flexible regarding the architecture that is simulated to allow architects to use the simulator as a high-level tool to explore the design space of a GPU architecture. 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 streaming multiprocessors (SMs).

4.2.1 Power Modeling Approach

In general, the power of switching circuits is described by the well-known Equation 4.1 [92]. The first term is the dynamic power that is spent charging and discharging of the capacitive loads when the circuit switches state. An important factor for estimating the dynamic power is the *activity factor* α that describes the percentage of the circuit's capacitance being charged during switching. The second term in Equation 4.1 is the short-circuit power that is 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 powers. Finally, the third term in Equation 4.1 is the *static* power that is consumed due to the leakage of current in the transistors. The leakage consists of two types: subthreshold leakage, where a transistor that is switched off leaks current between its source and drain, and gate leakage, where current leaks through a transistor's gate terminal.

$$P_{\text{total}} = \alpha C V_{dd} \Delta V f_{\text{clk}} + V_{dd} I_{\text{short-circuit}} + V_{dd} I_{\text{leakage}} \tag{4.1}$$

4.2.2 Overview of GPUSimPow

Figure 4.1 shows an overview of GPUSimPow. Structurally, GPUSimPow consists of two main parts. First, a cycle-accurate GPGPU architectural simulator that simulates a given kernel and generates activity factors α (utilization information) for all components of a GPU architecture. Second, a chip representation with a power model for each component that uses the activity information from the architectural simulator to produce power

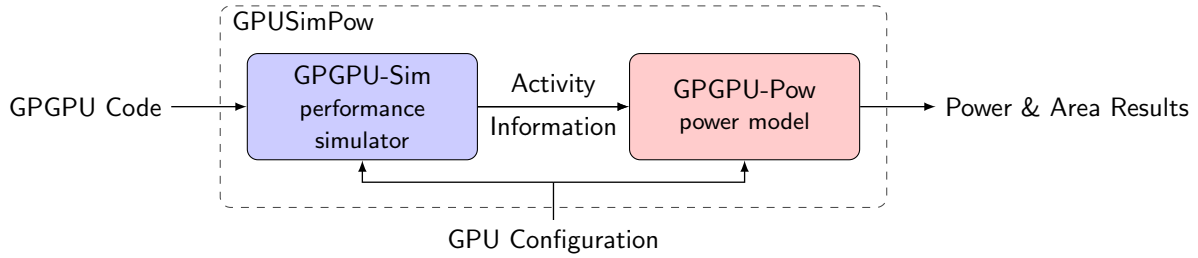


Figure 4.1: Overview of GPUSimPow [99]. © 2013 IEEE

numbers for a given kernel. From the chip representation, we derive statistics about area, peak power, leakage power, and short-circuit power.

For the cycle-accurate GPGPU simulator, we employ gpgpu-sim [15] that is modified to produce activity information for all components of the simulated architecture. gpgpu-sim has been developed for an architecture that is not equal but comparable to many present GPUs such as NVIDIA's Fermi [124] or AMD's GCN [8]. Further details about the architecture are given in the next Section 4.2.3.

The chip representation and power model are provided by a heavily modified variant of McPAT [92] and we call it *GPGPU-Pow*. McPAT uses three tiers *hierarchically* modeling to provide a flexible and highly accurate power model for *CPUs*. The *architectural tier* breaks down a processor into major components such as cores, caches, and memory controllers. The *circuit tier* maps the architectural components to basic circuit structures such as arrays or clocking networks and the *technology tier* provides the physical parameters such as current densities and capacitances of the circuits. Besides hierarchal modeling, another advantage of McPAT is its *combination* of analytical and empirical models for the individual components. We embrace both the hierarchical as well as the hybrid nature of McPAT and develop a McPAT-based power model for GPUs. On the one hand, this requires many modifications to McPAT, as many components that are present in the CPU architecture 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, McPAT enables us to utilize all the integrated low-level technological information, e.g., to scale a GPU power model for a specific manufacturing process node, we can use the ITRS (International Technology Roadmap for Semiconductors) roadmap scaling techniques already present in McPAT.

4.2.3 Modeled Architecture

GPU microarchitecture modeled in our power simulator is comparable to the one modeled in gpgpu-sim to ensure a good "fit" between the performance simulator and power model. On a high level, it models a single instruction multiple threads (SIMT) architecture that uses a stack-based divergence handling mechanism that is a well representative of contemporary GPUs.

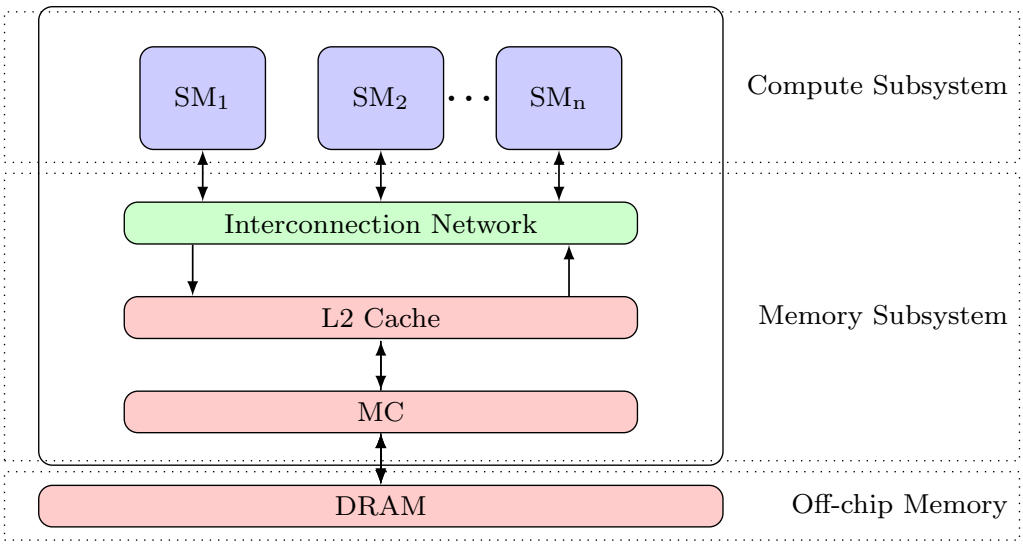


Figure 4.2: High-level overview of the modeled architecture.

Figure 4.2 shows a high level overview of the modeled GPU architecture. On a high level, a GPU chip in our model consists of a Compute Subsystem, Memory Subsystem, and Off-chip Memory. The compute subsystem contains a set of Streaming Multiprocessors (SMs) which are the compute horses of a GPU. The main components of the memory subsystem are Interconnection Network (NoC), L2 Cache, and Memory Controller (MC). Besides the on-chip components, we also model the Off-chip Memory. GPUs employ graphics double data rate (GDDR), a special type of DRAM designed for high bandwidth as Off-chip memory. For NoC, MC, and PCIe, we reuse highly configurable models already present in McPAT and adjust their parameters to fit the requirements of our model. The internal structure of the compute subsystem consists of a Warp Control Unit (WCU), a highly banked register file, a set of SIMD execution units (Integer Units, Floating Point Units, Special Function Units), and a load/store unit (LDSTU). More details of these components are presented in the following subsections.

Warp Control Unit

A Warp Control Unit (WCU) represents the front end of a single SM and it is responsible for keeping the execution back end, i.e., the functional units and the load/store unit busy with instructions at all times. Thus, a WCU handles thread management (e.g., 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 the WCU model is depicted in Figure 4.3.

The information needed for each warp to fetch instructions and manage threads of a warp is contained in a single multi-ported RAM table called Warp Status Table (WST). The WST contains one entry for each in-flight warp an SM can handle. A round-robin warp scheduler is modeled to fetch instructions. Such schedulers consist of a set of invert-

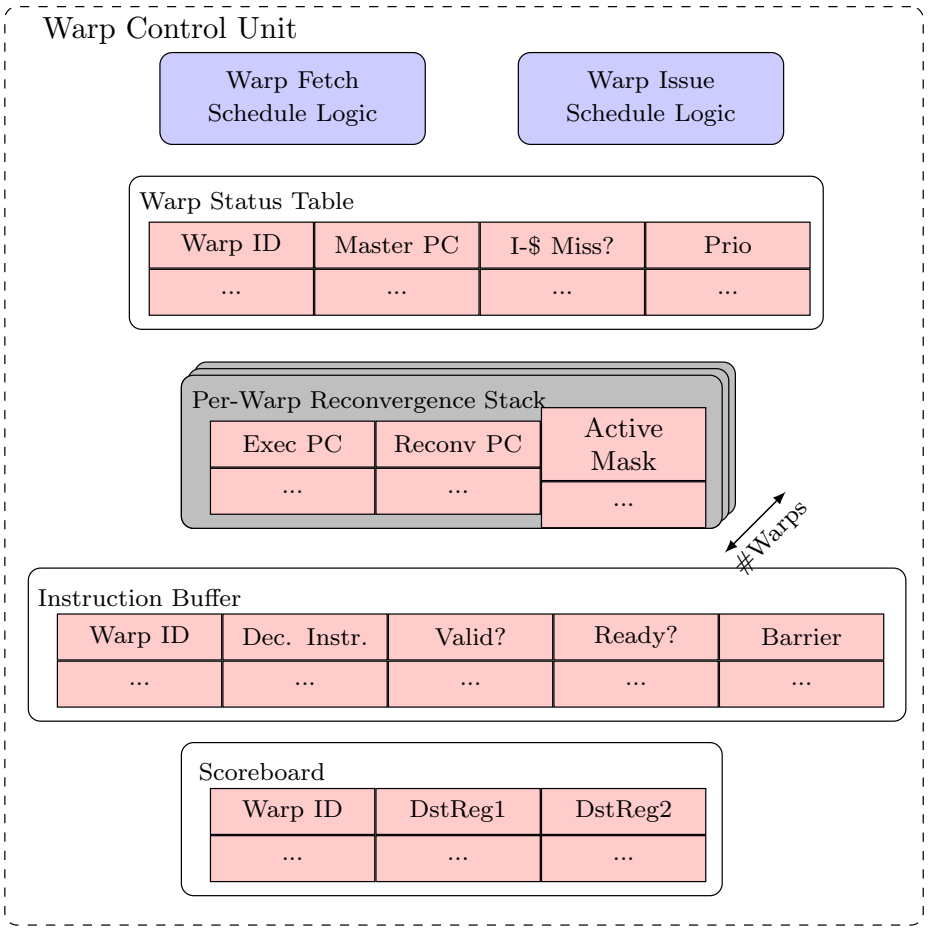


Figure 4.3: Overview of the front end of a GPU called warp control unit in our model [99]. © 2013 IEEE

ers, a wide priority encoder, and a phase counter. These components have been modeled from appropriate circuit plans [82] using McPAT's circuit and technology layers. After instructions have been fetched from the I-Cache, they are decoded. For this, we reuse the instruction decoder hardware models already present in McPAT.

GPUs use Single Instruction Multiple Thread (SIMT) execution model which allows the execution of only single instruction at a time for a warp. As the individual in-flight threads in a warp can execute different dynamic instruction paths due to branching, the execution of threads with different program counters (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 [32]. 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 as shown in Figure 4.3.

Once an instruction has been decoded, a WCU places the instruction into an instruction buffer (IB) slot. An instruction resides in its IB slot until it is ready to execute. An

instruction is ready to execute 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 the several slots tagged by its parent warp ID. For resolving register dependencies, GPUs (e.g., Fermi) use simple approaches based on scoreboarding [33]. In our model, a scoreboard is a cache-like table tagged by a warp ID.

Register File

The GPU register file model is based on the NVIDIA patent [95] 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 in a SIMT unit is a warp which is a group of threads. A GPU has a set of SIMT units which execute the threads of a warp in lock step. For example, a SIMT unit in the NVIDIA GT240 has eight floating point units (FPUs), eight integer units (IUs) and two special function units (SFUs). SFUs execute transcendental instructions such as sine, cosine, reciprocal, and square root. In our power model, we use the area numbers published by Sameh et al. [51] for FPUs. We use numbers published by Caro et al. [38] for power and area of the SFUs with scaling for the desired process technology. We develop our own measurement based empirical power models for integer units and floating point units (see Section 4.2.4).

Load/Store Unit

A load-store unit (LDSTU) is functionally responsible for handling instructions that read or write any kind of memory. In our power model, the LDSTU encapsulates the top-tier memory structures of an SM, i.e., L1 cache, shared memory (SMEM), constant cache and L2 cache. In the future variant of the model, the LDSTU will contain the texture caching subsystem, i.e., texture cache and texture mapping units, as well.

An high-level overview of the LDSTU is depicted in Figure 4.4. As the figure shows, a memory 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),

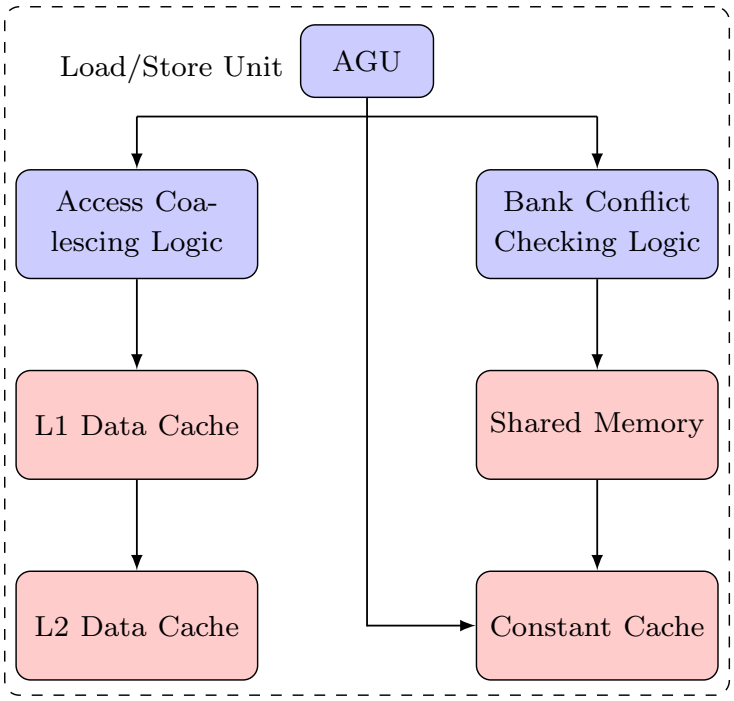


Figure 4.4: High-level overview of the internals of a GPU load/store unit [99]. © 2013 IEEE

each of which is able to generate 8 memory addresses per cycle [52]. Given the memory address bundle for all threads in a warp, the address bundle is further analyzed depending on the type of memory an instruction accesses.

If an 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 is cached in the memory hierarchy [169].

If an instruction accesses global memory, it is first coalesced before passing to the L1/L2 cache/DRAM. The memory coalescing logic is modeled based on a NVIDIA patent [125] 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. We find that CACTI cannot be used to model buffers with a few but very large entries such as the pending request table and input queue of the coalescer. Therefore, we compute the total amount of bits which must be held in the coalescing system at any time and then model the required storage using D Flip Flops.

In several modern GPUs, shared memory and L1 data cache are portions of the same physical memory structure. The distribution of physical memory between shared memory and L1 data cache is configurable. Therefore, we model shared memory and L1 data cache as an integrated structure and convert accesses to shared memory and L1 hits to

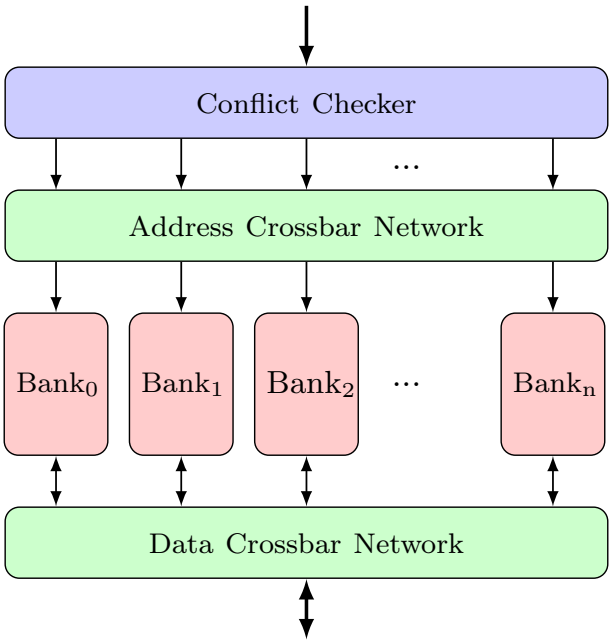


Figure 4.5: High-level overview of the shared memory model.

accesses to same integrated memory structure. Shared memory is used for inter-thread communication in a thread block. To provide high bandwidth, the shared memory is multi banked like register file. Figure 4.5 shows the high-level overview of the shared memory model. Besides the physical memory banks, the shared memory consists of interconnects for addresses and data, both modeled as crossbars, and a bank conflict checking unit [34]. In Fermi architecture, shared memory has 32 banks with the same number of inputs and outputs for the crossbars. Shared memory and register file have very similar structure.

L2 cache is shared by all streaming multiprocessors (SMs) and is connected to all SMs through an interconnection network. We use CACTI to derive an analytical model for the L2 cache.

Global Memory

The global memory in GPUs has high bandwidth but long latency. The current generation of GPUs such as Fermi use special high bandwidth Graphics Double Data Rate Synchronous Graphics Random Access Memory (GDDR5 SGRAM) to implement the global memory. The power consumed by a typical DDR or GDDR chip is the sum of background, activate, read/write, termination, and refresh power [107, 65]. We extract numbers for each of these components from industry data sheets [107, 65], however, data sheet numbers need to be scaled according to actual usage. For example, the actual read-/write power depends on the command scheduling, i.e., the percentage of read and write commands. We extract such activity factors from gpgpu-sim.

4.2.4 Deriving Power Empirically

We build empirical power models for the Integer Units (IUs) and Floating Point Units (FPUs) by stressing these units using microbenchmarks and measuring the power using our custom power measurement setup described in Section 4.3.1. As described earlier in Section 4.2.3, GPUs employ SIMT execution model to execute single instruction for multiple threads. 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 consumption of the execution units with reasonably high accuracy. For deriving the power models for both IUs and FPUs, we launch one thread block for each SM and use 512 threads per block to ensure all streaming multiprocessors (SMs) and targeted execution units are busy. For stressing the IUs, we use Linear Shift Feedback Register microbenchmark, while for stressing the FPUs we use Mandelbrot set iterations. We use loop unrolling to make the loop overhead of our microbenchmarks negligible. In both the cases, we alternately configure the test kernels to use 31 enabled threads per warp (config_1) and 1 enabled thread per warp (config_2). Both configurations have the same execution time. The energy per instruction (EPI) can be calculated by using a simple calculation shown in Equation 4.2. We calculate the power difference between these two kernel launches and divide the result by the number of SMs (#SM), difference in execution units enabled in an SM (#coresEnabled), number of executed instructions (#executed_ins) and SMs frequency (f_{clk}) to arrive at an estimate for the energy used by a single execution unit executing a single instruction. Our measurements show that integer instructions use approximately 40 pJ per instruction while floating point instructions use about 75 pJ per instruction. NVIDIA reports 50 pJ per floating point instruction [72].

$$EPI = (power\_config_1 - power\_config_2)/(\#SM * \#coresEnabled * \#executed\_ins * f_{\text{clk}}) \tag{4.2}$$

We develop power models for all components to the best of our knowledge where publicly available information is sufficient. However, there are some components of GPU architecture such as the raster operations pipelines (ROPs), fixed-function video decoder where publicly available information is especially scarce. These components also consume leakage power even if they are not used in GPGPU applications. While we cannot model such components accurately because of the lack of information, nonetheless, we know that these components are part of the chip. To be able to account for the amount of power they contribute, we use our measurement setup to build empirical models of "base power" for SMs and SMs clusters (called Thread Processing Clusters (TPCs) or, more recently, Graphics Processing Clusters (GPCs) in NVIDIA terminology). These base power numbers are derived by measuring SM/cluster power and subtracting the power of all components we know about. Figure 4.6 shows an example of the measurement used to estimate the cluster power. We execute the same kernel 12 times with the increasing number of thread blocks and measure the power consumption. We start with one thread block

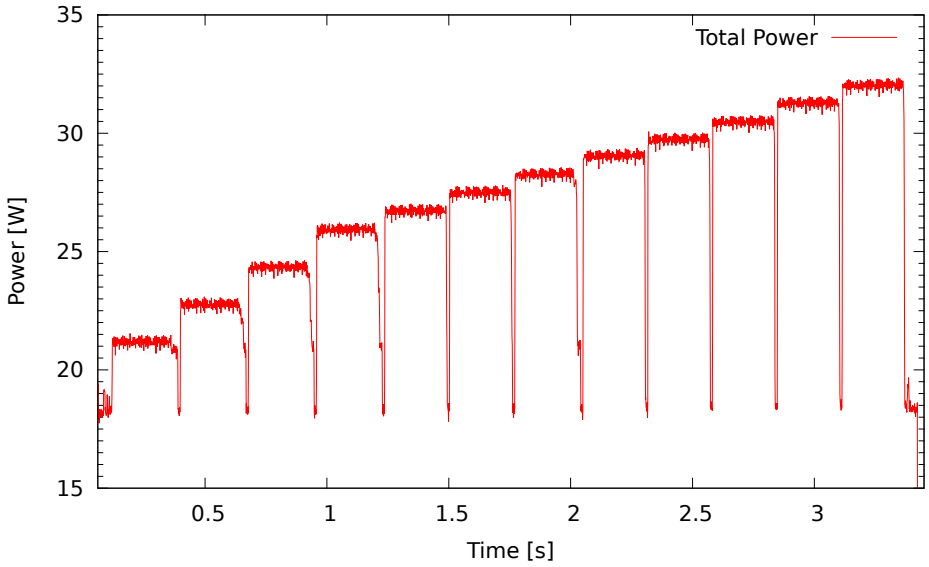


Figure 4.6: Power measurement results of GT240 running the same kernel 12 times with increasing number of thread blocks. GT240 features 12 SMs distributed equally over 4 SMs clusters [99]. © 2013 IEEE

and increase the number of thread blocks by one for the next iteration. The figure shows that increasing the number of thread blocks gradually increases the power consumption. More interestingly, the figure shows that up to 4 blocks, adding another thread block increases power by a larger margin than beyond 4 blocks. The reason for this is the way the global scheduler distributes thread blocks among SMs. The global scheduler first assigns the thread blocks *to unoccupied clusters* in round robin fashion, i.e., when a second thread block is added, it is scheduled on an SM in a different cluster than the first one. As we see from the figure, the activation of such an SMs cluster consumes about 0.692W power. Once all clusters are activated, in this case four, adding more thread blocks increases power by a relatively smaller margin. This power is attributed to the activation of an additional SM. We also notice that the activation of the very first SM/cluster consumes even more power than the other clusters. This extra power (3.34W) can be attributed to the activation of the global scheduler which distributes thread blocks to SMs.

4.3 Experimental Setup

To validate GPUSimPow, we compare the power estimated by the simulator with the real power consumed by the hardware for various GPGPU workloads. In this section, we describe our experimental setup to estimate and measure power. The custom power measurement testbed to measure power consumption of real GPUs as well as to derive empirical power models is explained in Section 4.3.1. The test system configuration is described in Section 4.3.2 and the benchmarks are presented in Section 4.3.3

Figure 4.7: Power measurement testbed with GT240 graphics card [99]. © 2013 IEEE

4.3.1 Power Measurement Testbed

We develop our custom power measurement testbed to validate the power simulator against real GPUs. The power measurement testbed consists of hardware and software components. Figure 4.7 shows the hardware part of the setup with NVIDIA GT240 graphics card. The hardware consists of a riser card with $20m\Omega$ probing resistors on the 12V and 3.3V rails to probe voltages and currents going to the GPU via the PCIe slot. NVIDIA GTX580 graphics card also has two external PCIe power connectors. To measure the power transmitted via these connectors we inserted $10m\Omega$ probing resistors into the PCIe power cables going to card. We design a custom signal conditioning board to convert the voltages into signals that can be easily measured by off-the-shelf data acquisition (DAQ) hardware. A resistive divider is used to scale the voltages into $0-5V$ range. The voltage drops over the sensing resistors are amplified and shifted into a usable common mode range using Analog Devices AD8210 Current Shunt Monitors. After signal conditioning, the signals are sampled using a NI USB-6210 USB DAQ at a rate of $31.2kHz$. Our resistive voltage divider is built from 1% resistors and has a gain accuracy of $\pm 1.7\%$ and no offset error. The AD8210 has a gain accuracy of $\pm 0.5\%$ and an offset error of $\pm 1mV$ at its output. At $12V$, this offset error translates to an error of up to $60mW$ in power measurements. The error range of signal conditioning and measurement is thus $\pm 1.5\%$ for currents and $\pm 1.7\%$ for voltages. In the relevant -5 to $5V$ range, the DAQ has a specified gain accuracy of 0.0085% and an offset error of $0.1mV$. Not taking the small offset errors into account, overall, our system thus measures power within $\pm 3.2\%$. We developed a custom measurement tool that controls the DAQ and calculates power and energy from the measured voltages and currents. This tool is capable of using a GPU profiler to get start and end timestamps of the kernels running on the GPU. Using this information and the measured power waveform, the average power and amount of consumed energy can be calculated for each kernel execution.

Table 4.1: Key features of GT240 and GTX580 used for experimental evaluation [99]. © 2013 IEEE

Feature	GT240	GTX580
#SMs	12	16
#Threads per SM	768	1536
#FUs per SM	8	32
Uncore clock	550 MHz	822 MHz
Shader-to-Uncore	2.47×	2×
#Warps in-flight	24	48
Scoreboard	×	✓
L2-$ size	×	768KByte
Process node	40nm	40nm

4.3.2 System Configuration

For evaluating GPUSimPow, we choose two different NVIDIA GPU architectures to show our power simulator configurability. Table 4.1 shows the key parameters of both the GPUs. GT240 graphics card is based on Tesla architecture and provides a good insight of many key features of modern GPUs. An initial advantage of using Tesla architecture is that gpgpu-sim simulator shows the highest correlation to real hardware for such architectures. GT580 is based on Fermi architecture. GT240 is a low-end graphics card while GTX580 is a high-end enthusiastic market graphics card.

We perform both measurements and simulations for a set of kernels selected from recent GPGPU benchmark suites (see next Section 4.3.3) for each of the two GPUs presented in Table 4.1. For each kernel and GPU, we record measured and simulated static and dynamic power as well as measured and simulated execution time. Table 4.2 shows the key parameters of the experimental environment used to acquire the results. To measure the static power, we run the same benchmark at stock frequency and at a 20% lower frequency. Then, we perform linear extrapolation from the two data points to estimate the power the chip would consume at a frequency of 0 Hz. As Equation 4.1 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, this methodology to measure static power is only possible for the GT240 card because NVIDIA Linux drivers do not yet support changing the clock speed for the GTX580. Therefore, we measure static power of 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.

Table 4.2: Summary of our experimental setup [99]. © 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

4.3.3 Benchmarks

Table 4.3 shows the benchmarks used for the experimental evaluation. The benchmarks are taken from the popular Rodinia benchmark suite [25], CUDA SDK [123] and gpgpu-sim [15], covering a wide variety of application domains. As our analysis focuses on the power consumed by a GPU, we are only interested in the *GPGPU kernels* present in each benchmark. The second column of Table 4.3 shows the number of different kernels in each benchmark. In some benchmarks, there are kernels with very short execution time (less than 500 μs). Because these kernels are too short for reliable measurements, we modify such benchmarks to execute the same kernel 100 times. The lower part of Table 4.3 shows 5 more benchmarks included from gpgpu-sim [15]. These benchmarks are included to expand the comparison space in Section 4.4.3.

4.4 Results

In this section, we present our experimental results. Section 4.4.1 compares simulated and measured power. Section 4.4.2 shows power profiling capabilities of GPUSimPow. Section 4.4.3 quantitatively compares GPUSimPow and GPUWattch [90].

4.4.1 Simulated and Measured Power

For each benchmark, we estimate and measure the total power consumed by a GPU for the execution of each of its kernels. For kernels that are executed multiple times during one benchmark run, we calculate arithmetic average. In the end, we report simulated and measured dynamic power 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 same for each kernel.

Table 4.4 shows the estimated static power and area of both GPUs. Using the static power measurement technique explained in Section 4.3.2, we estimate the static power of GT240 to be 17.6 W. The card seems to do some power gating to reduce the static power

Table 4.3: GPGPU benchmarks used for experimental evaluation [99]. © 2013 IEEE for the upper half of the table.

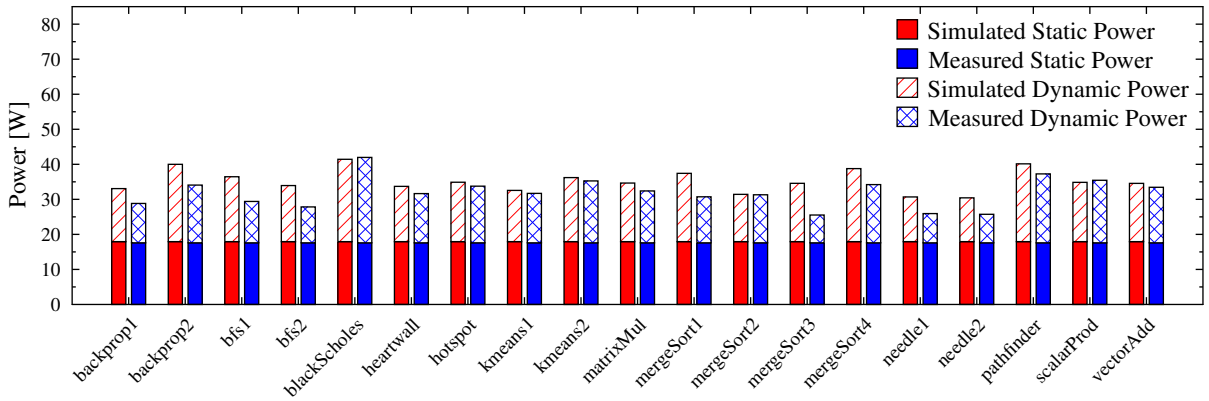
Name	#Kernels	Description	Origin
backprop	2	Multi-layer perceptron training	Rodinia
heartwall	1	Ultrasound image tracking	Rodinia
kmeans	2	k-means clustering	Rodinia
pathfinder	1	Dynamic programming path search	Rodinia
bfs	2	Breadth-first search	Rodinia
hotspot	1	Processor temperature estimation	Rodinia
nw	2	DNA sequence alignments	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
libor	2	LIBOR swaption portfolio pricing	gpgpu-sim
lps	1	3D Laplace solver	gpgpu-sim
ray	1	Ray tracing	gpgpu-sim
sto	1	Distributed storage systems	gpgpu-sim
nn	4	Neural network	gpgpu-sim

when no kernel is running on it. When no kernel is running, the card consumes about 15 W. While for some milliseconds before and after the execution of a kernel, the card consumes about 19.5 W. About 90% of the power consumed by the card in this state thus seems to be static power. GTX580 is using 90 W in the same state, so we estimate its static power to be 80 W.

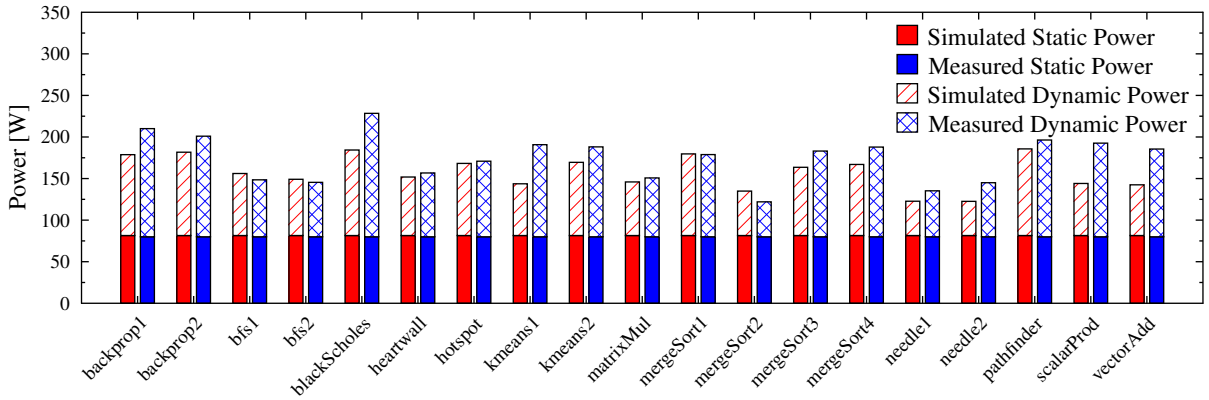
Figure 4.8a shows the total simulated and measured power results for GT240. Each bar in the figure is divided into two parts. The first part is the static power that is same for all kernels and the second part is dynamic power that is kernel specific. In general, the figure shows strong similarity between the measurement and simulation results for the most benchmarks. For all benchmarks except `blackscholes` and `scalarProd` the simulator overestimates the power consumed by the card. When averaging errors, we always average the absolute value of errors, so that under- and overestimates can not cancel out. The average relative error is 11.7% for all kernels. The maximum relative error of 35.4% occurs in the `mergeSort3`. This is likely a measurement artifact. The

Table 4.4: Simulated and measured static power and area of GT240 and GTX580 [99]. © 2013 IEEE

		Static [W]	Area [mm^2]
GT240	Simulated	17.9	105
	Measured	17.6	133
GTX580	Simulated	81.5	306
	Measured	80.0	520



(a) GT240



(b) GTX580

Figure 4.8: 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 power, i.e., the sum of static and dynamic power [99]. © 2013 IEEE

execution time of the kernel is short (1 ms) and the benchmark could not be changed to call it multiple times because the kernel does in-place processing of its data.

The simulator slightly overestimates the power consumed by the chip. This trend is mostly caused by the overestimation of simulated dynamic power as the simulated static power is just 0.3W (1.7%) larger than the measured static power. Not considering static power, the average relative error for dynamic power is 28.3%. The estimated chip area is smaller than the actual chip area (105.0mm^2 vs. 133mm^2).

Figure 4.8b shows the results of our experiments for GTX580. We again notice strong similarity between simulated and measured power. This also shows that our empirically derived models also work well on GTX580 even though they are obtained using the GT240. The average relative error for GTX580 is 10.8%. `scalarProd` is the kernel with the largest relative error (25.2%) for GTX580. The average relative error for the dynamic power on GTX580 is 20.9%. The estimated static power of GTX580 is 81.5 W which is very close to the measured static power from the real hardware (80 W). As explained in Section 4.3, we could not use measurements at different clock speeds for GTX580 to measure its static power. As a result, we use a different method to measure the static power of GTX580. The better match between the measured static power and the simulated static power could be a result of the more accurate static power estimate by GPUSimPow or it could be caused by the different methodology we used for GTX580.

4.4.2 Power Profiling

While an accurate estimate of the total power consumption is useful, the *distribution* of power consumption over the individual components even matters more. As GPUSimPow contains power models for the internals of each SM, interface and controller on a GPU, it automatically produces detailed power statistics for these internals. Therefore, it is possible to generate a *power profile* for a particular benchmark that breaks the overall power down to individual components. Table 4.5 shows such a power profile for the `blackscholes` benchmark. Please note that the table does not include the power consumed by the external DRAM (4.3 W).

The top part of the table shows both static and dynamic power for the top-level components of GT240. It can be seen that by far the largest fraction of the total power is, as one would expect, consumed by the SMs (82.2%). Previous researchers have reported similarly high power consumption by SMs [53]. Gebhart et al. [53] estimate the total SMs power to be 70% of the entire chip. According to the output of GPUSimPow, the next most power consuming component is the network on chip (7.3%), followed by the memory controller (6.1%) and PCIe controller (4.1%). Note that some of the GPU components such as the global scheduler, video decoder are not modeled in detail due to lack of publicly available information. Therefore, the power consumed by these components is included in the (per-core) undifferentiated core and base power.

The bottom part of Table 4.5 shows the power consumed by the individual components of a single SM. Overall, an SM consumes 2.31 W. As the table shows, surprisingly, a large fraction of the total power is attributed to the SM base power (8.6%) and undifferentiated core (38.3%). While the former includes all the *per-core* components we could only model

Table 4.5: `blackscholes` power breakdown on the entire GT240 GPU (top) and on a single SM (bottom) [99]. © 2013 IEEE

		Static [W]	Dynamic [W]	Percent
GPU	Overall	17.934	19.207	100.0
	SMs	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
SM	Overall	1.283	1.031	100.0
	Base Power	0.000	0.199	8.6
	WCU	0.042	0.089	5.7
	Register File	0.112	0.173	12.3
	Execution Units	0.009	0.556	24.4
	LDSTU	0.234	0.014	10.7
	Undifferentiated Core	0.886	0.000	38.3

empirically due to the lack of information (see Section 4.2.4), the latter includes a per-core fraction of the *global* GPU components that could only be modeled empirically. As we have no detailed power models for components included in the undifferentiated core, we cannot generate any activity factors for them in gpgpu-sim and thus the entire power consumption of the undifferentiated core is attributed to static power. Taking base power and undifferentiated core aside, the most power is consumed by the execution units (24.4%). After the execution units, the next most power is used in the register file (about 12.3%). This number has been confirmed by previous research [53]. As one would expect from a SIMD architecture, the smallest part of the SM power is consumed by the front end of the GPU, i.e, the warp control unit (5.7%). 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. For reasons of conciseness, however, no such investigations are presented in this chapter.

4.4.3 GPUSimPow vs. GPUWattch

GPUSimPow [99] was published in April 2013. A very closely related GPU power simulator called GPUWattch [90] was published in June 2013. Like GPUSimPow, GPUWattch is also designed by integrating gpgpu-sim [15] with a heavily modified variant of Mc-

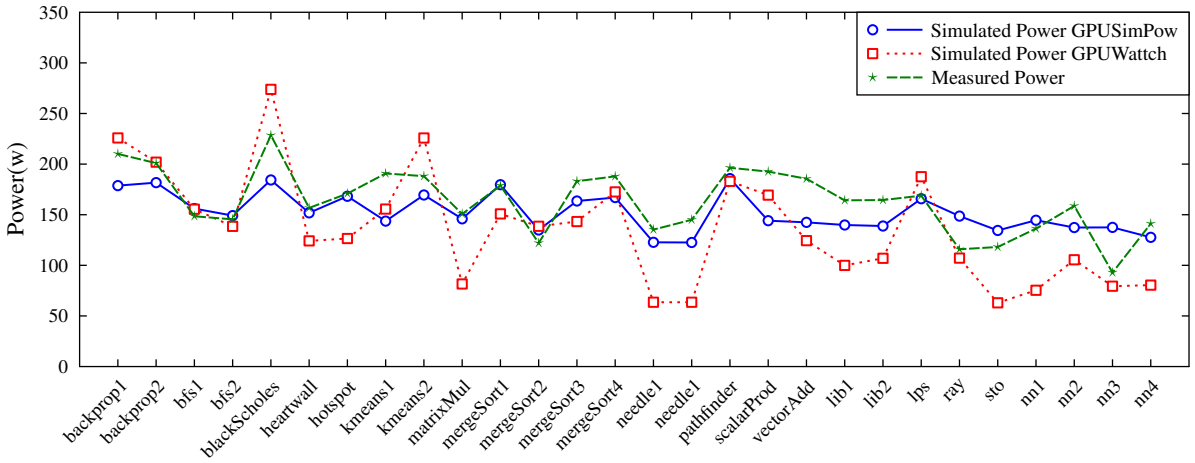


Figure 4.9: GPUSimPow vs. GPUWattch.

PAT [92]. GPUWattch also adds GPU-specific components such as streaming multiprocessor pipeline, register file, shared memory to McPAT and adapts a few components already present in McPAT such as memory controller, NoC to GPU requirements. Thus, both GPUSimPow and GPUWattch have a similar design philosophy, warranting a comparison between them.

In order to quantitatively compare GPUSimPow and GPUWattch, we simulate a set of benchmarks using both the simulators and note simulated power. We also measure power consumed on real hardware by using our custom power measurement testbed. Figure 4.9 shows the simulated power reported by GPUSimPow and GPUWattch as well as the measured power for GTX580 for a set of kernels. Figure 4.9 shows that in general GPUSimPow follows the measured power more closely than GPUWattch. GPUSimPow has an average relative error of 10.8%, while GPUWattch has an average relative error of 20.5%. Figure 4.9 uses more benchmarks than the results presented in Section 4.4.1 because we pick additional benchmarks to expand the comparison space. From the results shown in Figure 4.9, we conclude that in general, both the simulators follow the measured power trend, however, GPUSimPow is more accurate than GPUWattch. Unfortunately, it is very hard to pinpoint which GPU components are modeled more accurately in GPUSimPow than GPUWattch or vice versa because it is tough to accurately validate component-level power. However, both the simulators validate static and dynamic power consumption of GPUs under test and GPUSimPow estimates static power more accurately than GPUWattch. GPUSimPow is validated against GTX580 which is a Fermi architecture, while GPUWattch is validated against GTX480 which is also a Fermi architecture. GPUSimPow estimates the static power of GTX580 to be 81.5 W, while the measured power is 80 W. GPUWattch estimates the static power of GTX480 to be 41.9 W, while the measured power is 59 W. The relative error is only about 2% for GPUSimPow, while it is about 29% for GPUWattch.

4.5 Summary

In this chapter, we presented a novel power simulator for GPUs called GPUSimPow that can provide an accurate estimate of the area, static power and dynamic power. We designed GPUSimPow by integrating a modified version of gpgpu-sim, a cycle-accurate architectural simulator for GPUs, and a heavily modified McPAT, a power simulator for CPUs. We used hybrid approach for power modeling that improved flexibility compared to empirical approaches and accuracy compared to analytical approach. We validated the power simulator by using a highly accurate custom power measurement testbed. Our evaluations on a set of well-known benchmarks showed that GPUSimPow has an average relative error of 11.7% and 10.8% between the simulated power and measured power for GT240 and GTX580, respectively. As GPUSimPow is highly configurable, programmers and computer architects can accurately estimate the power consumed by different designs *without building an actual chip*, thereby enabling architectural design space exploration from power and energy perspective.

The component level power profiling capabilities of GPUSimPow demonstrated its usability not only for estimating the total power consumption but also down to the detailed individual components. These power profiles can be used to drive power optimizations. In its current state, GPUSimPow is a very useful tool to gain valuable insights into where power is consumed in a GPU. However, as the power breakdown revealed, a large fraction of the simulated power is currently attributed to components that are not modeled in detail, i.e., the "undifferentiated core" due to the lack of available information. More research needs to be done for creating accurate models of these components.

In the next chapter, we will use GPUSimPow to study the performance and energy efficiency of GPU kernels. We will identify bottlenecks that lead to low energy efficiency in GPUs. We will also study the power consumption of a GPU at components level and study the correlation between components power consumption and workload metrics.

5 GPU Energy Efficiency and Performance Bottlenecks

The work presented in this chapter was previously published: S. Lal, J. Lucas, M. Andersch, M. Alvarez-Mesa, A. Elhossini, and B. Juurlink, "GPGPU Workload Characteristics and Performance Analysis," in Proceedings of the IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), © 2014 IEEE.

In this chapter, we study the energy efficiency of a wide range of GPGPU kernels and investigate the bottlenecks that lead to low performance and low energy efficiency. We use the power simulator presented in the last chapter to study energy efficiency. We also study GPU power consumption at the components level and investigate their correlation with workload metrics.

5.1 Introduction

GPUs are massively multi-threaded, throughput oriented devices that employ a large number of parallel threads to achieve high throughput. The peak throughput of GPUs is a magnitude higher than CPUs. The higher throughput also comes with higher power consumption. However, GPUs are more energy-efficient devices [64] compared to CPUs, as performance per watt of GPUs is much higher than CPUs. For example, NVIDIA's GTX 690 has 18.7 GFLOPS/W (single precision) while Intel's Haswell i7 4770K has 5.3 SP GFLOPS/W (single precision). However, due to various performance bottlenecks which result in under-utilization of resources, the achieved performance per watt of GPUs is often much lower than what could be gained theoretically. There are several factors that contribute to low performance such as low occupancy, limited memory bandwidth, control flow divergence, and memory divergence [85, 145, 135, 106, 50].

To create energy-efficient techniques at the architectural level, we need to gain GPU power consumption knowledge at a fine-grained level and understand the bottlenecks that lead to low performance and low energy efficiency [43, 72]. Therefore, in this chapter we study GPU power consumption at the components level for a diverse set of workloads and investigate the bottlenecks which cause low performance and low energy efficiency. We also explore the correlation between workload metrics such as IPC, SIMD utilization and components power consumption to understand how workload characteristics affect power consumption. Moreover, we study workload characteristics at a kernel level instead of a benchmark level because kernels from the same benchmark might have completely

different characteristics.

To investigate the bottlenecks for low performance, we divide the low-performance kernels into two categories: low occupancy and full occupancy. The low occupancy kernels are further divided into different categories depending on the resources their occupancy is limited by. We increase the occupancy of each category by increasing the corresponding resources and study if high occupancy helps in achieving higher performance and energy efficiency. We show that increasing the occupancy helps in increasing performance and energy efficiency for most of the kernels, but just increasing occupancy is not enough to achieve desired performance. We also analyze full occupancy kernels and study if they are limited by memory bandwidth, low coalescing efficiency, or low SIMD utilization.

In summary, we make the following contributions in this chapter:

- We study the energy efficiency of a wide range of kernels and show that most kernels have low performance and low energy efficiency.
- We divide the kernels into high performance and low performance categories and further investigate the bottlenecks of low-performance category.
- We show that increasing occupancy does help to increase performance and energy efficiency of low occupancy category.
- We also analyze kernels having full occupancy but still performing low and study if these kernels are limited by memory bandwidth, low coalescing efficiency or low SIMD utilization.
- We study power consumption at the components level and show the most power consuming components in each category.
- We also show the existence of a correlation between workload metrics and components power consumption.

The chapter is organized as follows. Section 5.2 presents GPU energy efficiency and our bottlenecks investigation methodology. In Section 5.3, we explain experimental setup. Section 5.4 presents investigation results. Finally, we summarize the contributions of this chapter in Section 5.5.

5.2 Performance Bottlenecks Investigation Methodology

In this section, we first study the energy efficiency of a large number of kernels and then provide an overview of our methodology to investigate bottlenecks that cause low performance and low energy efficiency.

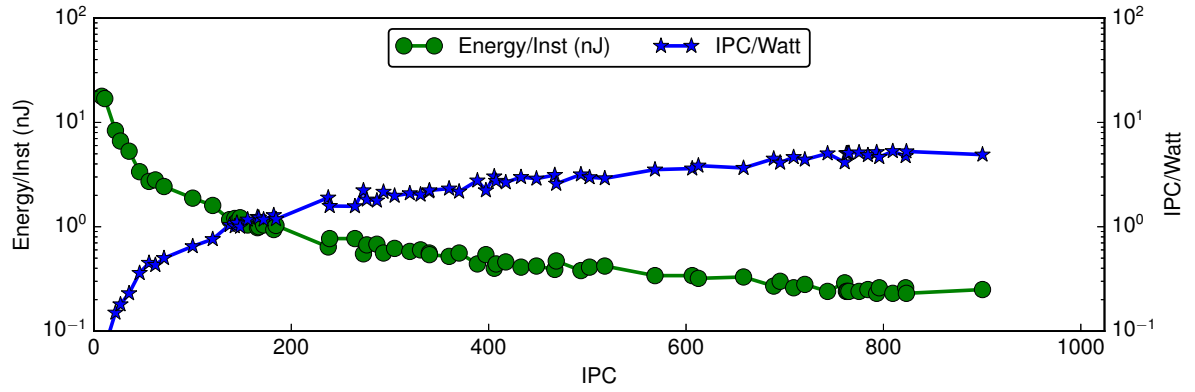


Figure 5.1: Energy efficiency for a set of kernels on GTX580 [84]. © IEEE 2014

5.2.1 GPU Energy Efficiency

GPUs are energy-efficient devices at full utilization. This is demonstrated by the inclusion of GPUs in all the top ten supercomputers in the green 500 list [2]. However, due to various bottlenecks which result in under-utilization of resources, the performance per watt of GPUs is much lower than what could be gained at full utilization. Energy per instruction (E/I) and IPC per watt (IPC/W) are metrics often used to study the energy efficiency of GPU workloads. Figure 5.1 shows the E/I (nJ) and IPC/W against IPC for several kernels. Each point represents a kernel. For a description of benchmarks please refer to Section 5.3.3. The figure shows that the E/I increases with the decrease in IPC which results in low energy efficiency for kernels with low IPC. The exact cut for low IPC may be a point of open discussion, but the trend shows that the lower IPC results in lower energy efficiency. For this study, we classify the kernels with IPC >50.5% of peak IPC into *high performance* (HP) category and kernels with IPC <= 50.5% of peak IPC into *low performance* (LP) category. We choose the cut at 50.5% instead of 50% because there is one kernel with IPC 50.5% and it lies closer to LP category than HP category. The peak IPC is 1024 (#SM×#FU per SM×2 = 16×32×2 = 1024) for the simulated GPU configuration described in Table 5.2. There is a factor of 2 in the formula because gpgpu-sim simulates a full warp at half frequency [15].

The average E/I for the HP and LP category is 0.27 nJ and 2.01 nJ, respectively. The later is 7.5× less energy efficient compared to former, a huge difference which is not good for the future growth of high performance computing. For example, to build an exascale machine (a billion billion calculations per second) in a power budget of 20MW requires an energy per instruction of 10 pJ [36].

The HP and LP categories have 21 and 47 kernels, respectively and the average IPC for the former is 741 and 250 for the later, which is less than 25% of the peak IPC. Surprisingly, more than 69% of the kernels belong to the LP category. Blem et al. [17] also notice that over half of the benchmarks they study have IPC less than 40% of the peak IPC for Tesla C1060. The figure also shows that the IPC/W decreases with the decrease

Table 5.1: Resource constraint for full occupancy [84]. © IEEE 2014

Resource	Max	Required for full occupancy
CTA limit	8	Min 192 threads/CTA
Registers	32K	Max 21/thread
Shared memory	48KB	Max 6KB/CTA

in IPC. The average IPC/W for HP and LP category is 4.61 and 1.65, respectively. Thus, the LP category has low performance and energy efficiency. In the following section, we describe our methodology to investigate bottlenecks for low performance.

5.2.2 Methodology

GPUs are high throughput devices and use a large number of threads to hide the long latency of operations. The number of threads allocated to a streaming multiprocessor (SM) of a GPU is noted by a metric called occupancy. It is defined as a ratio of threads allocated to an SM and the maximum number of threads that can be allocated to an SM. A certain minimum occupancy, which may vary from kernel to kernel depending on ILP, the ratio of arithmetic to memory operations, etc., is necessary to hide latency and to achieve high throughput. The occupancy depends on parallelism in a kernel, the resources requested by the kernel and the resources available on the GPU. We do not consider those kernels for low-performance analysis which do not have enough parallelism and hence also do not have enough threads to fill all the SMs. We argue that although it might be possible to get higher performance with lower occupancy [161], for this study, we consider the case where parallelism is not an issue but other architectural resources are the bottleneck to higher performance and energy efficiency. The resources requested by a kernel are allocated at CTA (Cooperative Thread Array in NVIDIA terminology. A CTA is also known as a thread block.) granularity and at least one CTA needs to be allocated for a GPU to work. A CTA is a group of concurrent threads that execute the same program and may cooperate via shared memory to compute results. A GPU may have low occupancy because it does not have the required resources to allocate enough CTAs to fully occupy the GPU. Table 5.1 shows the resource constraint for full occupancy on NVIDIA's GTX580 which can hold a maximum of 1536 threads per SM. Any kernel which has less than 192 threads/CTA or requires more than 21 registers/thread or more than 6KB shared memory/CTA cannot have full occupancy. For example, assume a kernel has CTA size of 64 threads. The maximum number of threads that can be assigned to an SM, in this case, is 512 (CTA Size $\times$ the hardware limit for the maximum number of CTAs that can be allocated), resulting in occupancy of only 0.33 (512/1536). Thus, the occupancy can be limited by CTAs limit, registers usage, and shared memory usage.

Figure 5.2 shows an overview of the methodology used to investigate bottlenecks. As

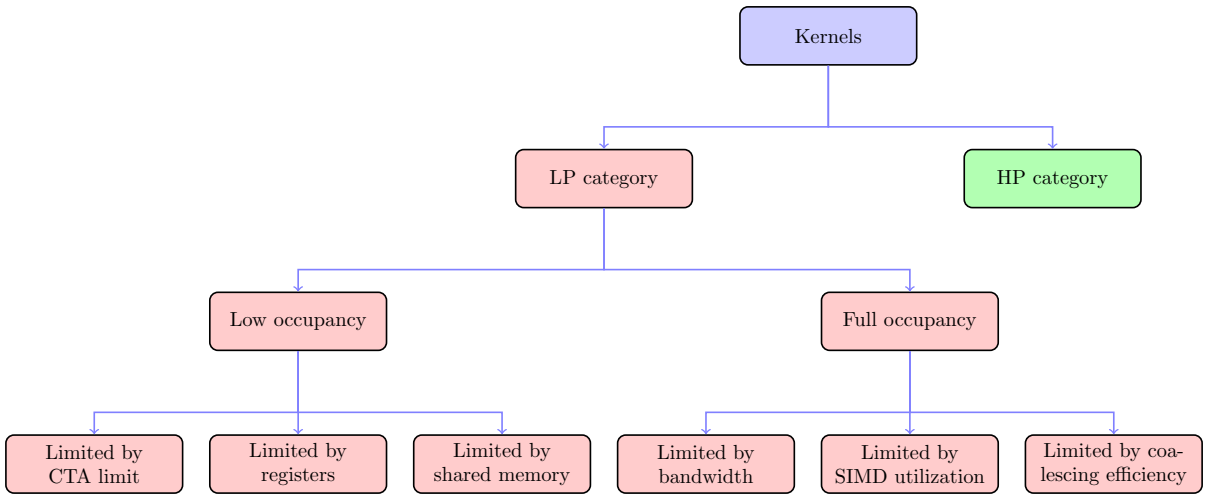


Figure 5.2: Bottlenecks investigation methodology.

described in Section 5.2, we study the energy efficiency of a large number of kernels and divide them into LP and HP categories. We do not further investigate HP category because we only focus on the LP category. As shown in Figure 5.2, we divide the LP category kernels into two categories for further analysis: *low occupancy* and *full occupancy*. The low occupancy category kernels have occupancy <1 and full occupancy category kernels have occupancy = 1. Low occupancy can restrict the latency hiding capabilities of a GPU and hence can restrict the performance as well. We investigate the effect of increasing the occupancy on performance and energy in Section 5.4.3. We further classify the low occupancy kernels depending on the resources they are limited by. As the occupancy can be limited by CTAs limit, registers usage, and shared memory usage, the low occupancy category kernels are further classified into three categories: 1) limited by CTA limit, 2) limited by registers, and 3) limited by shared memory.

The full occupancy category kernels have maximum number of threads that can be assigned to an SM, but they are still performing low. In this case, the most likely bottlenecks are high bandwidth utilization, low SIMD utilization, and low coalescing efficiency. Therefore, we further classify the full occupancy category kernels into three categories: 1) limited by bandwidth, 2) limited by SIMD utilization, and 3) limited by coalescing efficiency. We investigate if any of them is a bottleneck for high performance in Section 5.4.4.

5.3 Experimental Setup

5.3.1 Simulator

We use GPUSimPow [99] for simulating different benchmarks. The simulator has an average relative error of 11.7% and 10.8% between simulated and measured power for GT240 and GTX580, respectively. For more information regarding the power simulator,

Table 5.2: Baseline simulator configuration [84]. © IEEE 2014

Parameter	Value	Parameter	Value
#SMs	16	Shared memory/SM	48KB
SM freq (MHz)	822	L1 $ size/SM	16KB
Max #Threads per SM	1536	L2 $ size	768KB
Max #CTA per SM	8	# Memory controllers	6
Max CTA size	512	Memory type	GDDR5
#FUs per SM	32	Memory clock	2004 MHz
#Registers/SM	32K	Memory bandwidth	192.4 GB/s

please refer to [99]. We use GPUSimPow to simulate a GPU similar to NVIDIA GTX580. Table 5.2 summarizes the baseline configuration for the simulator.

5.3.2 Evaluated GPU Components

Table 5.3 shows the list of GPU components evaluated for the power consumption. The table also shows a short description of each component. We divide the GPU power consumption into components such that it is neither too coarse grained nor too fine grained. For more details of the components, their subcomponents and power models please refer to Chapter 4.

5.3.3 Benchmarks

Table 5.3.3 shows the benchmarks used for evaluation. The benchmarks selection includes benchmarks from the popular Rodinia benchmark suite [25] and CUDA SDK [123]. Our benchmarks selection also covers benchmarks recommended by Goswami et al. [57] and an internally developed motion compensation kernel for H264 [163].

5.3.4 Workload Metrics

We use several workload metrics to study the performance characteristics as in [57, 73]. Table 5.5 shows the set of metrics selected for studying the performance characteristics of workloads. A short description of each metric is also given in the table.

The SIMD Utilization is calculated by the equation given below.

$$\text{SIMD Utilization} = \sum_{i=1}^{n} W_{\text{i}}.\text{i} / \sum_{i=1}^{n} W_{\text{i}} \tag{5.1}$$

where, n is the warp size, W_{i} = Number of cycles when a warp with i active threads is

Table 5.3: GPU Components evaluated for power consumption and correlation [84]. © IEEE 2014

Component name	Abbrev.	Description and subcomponents, if any
Register file	RF	A GPU register file contains multiple SRAM banks, crossbar, and operand collectors.
Execution units	EU	Execution units consist of integer units, floating point units, and special function units.
Warp control unit	WCU	It is the front end of a GPU and contains warp status table, instruction buffer, reconvergence stack, and scoreboard as subcomponents.
Load store unit	LSU	It handles load and store requests to memory subsystem and it contains memory coalescer, bank conflict checker, shared memory, L2 cache, constant cache, and texture cache.
Base power	BP	It is the power consumed when an SM is activated.
Clusters power	CP	It is the power consumed when a cluster is activated.
Network on chip	NoC	It connects SMs to global memory.
Memory controller	MC	The current generation of GPUs such as Fermi use 64-bit memory controllers.
Global memory	GM	It is the off-chip memory and current generation of GPUs such as Fermi use GDDR5.
Total Power	TP	It is the power consumed by all GPU components.

scheduled on a SIMD unit.

5.4 Results

We first present results for the correlation between components power consumption and workloads metrics in Section 5.4.1. We then discuss components power consumption for the low performance and high performance categories in Section 5.4.2. The bottlenecks investigation results for the low and full occupancy categories are presented in Sections 5.4.3 and 5.4.4, respectively.

Table 5.4: GPGPU benchmarks used for experimental evaluation [84]. © IEEE 2014

Name	Abbrev.	#Kernels	Description	Origin
backprop	BP	2	Multi-layer perceptron training	Rodinia
bfs	BFS	2	Breadth-first search	Rodinia
b+tree	BT	2	Graph search	Rodinia
cfd	CFD	4	Computational fluid dynamics	Rodinia
heartwall	HW	1	Ultrasound image tracking	Rodinia
hotspot	HS	1	Processor temperature estimation	Rodinia
kmeans	KM	2	k-means clustering	Rodinia
lavaMD	MD	1	Molecular dynamics	Rodinia
leukocyte	LC	3	Microscopy video tracking	Rodinia
mummergpu	MUM	2	Pairwise local sequence alignment	Rodinia
pathfinder	PF	1	Dynamic programming path search	Rodinia
srad_v1	SRAD1	6	Speckle reducing anisotropic diffusion	Rodinia
srad_v2	SRAD2	2	Speckle reducing anisotropic diffusion	Rodinia
similarityScore	SS	17	Similarity score calculation	Rodinia
blackscholes	BS	1	Black-Scholes PDE solver	CUDA SDK
binomialOptions	BN	1	Binomial options pricing	CUDA SDK
convolutionSep	CS	2	Convolution	CUDA SDK
fastWalshTransform	FWT	3	Fourier transform	CUDA SDK
histogram	HG	4	Histograms for analysis	CUDA SDK
mergesort	MS	4	Parallel merge-sort	CUDA SDK
monteCarlo	MC	2	Monte carlo numerical solver	CUDA SDK
scalarprod	SP	1	Scalar product of two vectors	CUDA SDK
scan	SCAN	3	Parallel prefix sum	CUDA SDK
transpose	MT	8	Computation of matrix transpose	CUDA SDK
vectoradd	VA	1	Addition of two vectors	CUDA SDK
storegpu	STO	1	Distributed storage systems	Third party [5]
H264	MCO	2	H264 video decoding	Wang et al. [163]

5.4.1 Correlation

We calculated the Pearson correlation coefficient between the workload metrics and components power consumption for all kernels. The Pearson correlation coefficient is a measure of linear dependence between the two variables and it varies between -1 and 1. The higher absolute value of correlation coefficient means strong linear dependence between the metric and the corresponding component. The negative value means there is an inverse dependence. Since the static power is caused by leakage currents and it does not depend on the workload characteristics, but only on an architecture where the workload

Table 5.5: Workload metrics with short description [84]. © IEEE 2014

Workload metric	Abbrev.	Description
Instructions per cycle	IPC	Instructions per cycle.
Arithmetic instructions	AI	Ratio of arithmetic instructions to total instructions.
Branch instructions	BI	Ratio of branch instructions to total instructions.
Memory instructions	MI	Ratio of memory instructions to total instructions.
Bandwidth utilization	BW	Ratio of bandwidth utilized and bandwidth available.
Coalescing efficiency	CE	Ratio of global memory instructions and global memory transactions.
SIMD utilization	SU	Average utilization of an SM's core for issued cycles. It does not include the cycles for which pipeline is stalled and cannot issue instructions.
Pipeline stalled	PS	The fraction of total cycles for which pipeline is stalled and cannot not issue instructions.
Active warps	AW	Number of active warps per SM.

Table 5.6: Pearson correlation coefficient between workload metrics and components power consumption.

	RF	EU	WCU	BP	LSU	CP	NoC	MC	GM	TP
IPC	0.95	0.92	0.80	0.46	0.26	0.46	-0.06	-0.12	-0.13	0.69
AI	0.29	0.42	0.15	0.07	-0.18	0.07	0.02	-0.02	-0.01	0.23
BI	-0.29	-0.32	-0.25	-0.41	-0.16	-0.41	-0.18	-0.20	-0.21	-0.40
MI	-0.14	-0.25	-0.02	0.17	0.29	0.17	0.07	0.12	0.12	-0.02
BW	-0.09	-0.08	0.27	0.44	0.33	0.44	0.47	0.98	1.00	0.54
CE	0.22	0.29	0.24	0.17	0.29	0.17	-0.10	0.28	0.26	0.31
SU	0.30	0.35	0.41	0.46	0.32	0.46	0.27	0.15	0.15	0.45
PS	-0.92	-0.86	-0.72	-0.29	-0.18	-0.29	0.16	0.23	0.24	-0.56
AW	0.54	0.52	0.86	0.63	0.36	0.63	0.32	0.36	0.37	0.77

is executed, therefore, we only consider dynamic power for studying the correlation in this section and components power consumption in Section 5.4.2.

Table 5.6 shows the correlation coefficient between workload metrics and components power consumption. The table shows that IPC has strong correlation with RF (0.95), EU (0.92), and WCU (0.80) which means changes in IPC are strongly correlated to changes in the power consumption of these components. The metrics related to types of instructions

Table 5.7: Components dynamic power consumption (W) [84]. © IEEE 2014

	RF	EU	WCU	BP	LSU	CP	NoC	MC	GM
HP	11.3	20.2	16.2	3.8	0.6	13.1	2.3	4.2	8.3
LP	4.3	7.0	11.2	3.8	0.9	13.0	4.8	7.8	14.4

(AI, BI, and MI) do not have strong correlation with any of the components, but shows some expected trends. For example, AI has positive correlation with RF (0.29), EU (0.42), WCU (0.15), but it has negative correlation with MC (-0.02). BW utilization has very strong correlation with MC (0.98) and GM (1.0) which means the power consumption of MC and GM is strongly depended on BW utilization. PS is almost inverse of IPC for all components. This means we can choose just one of them. AW has strong correlation to WCU (0.86) which shows more active warps will consume more power in this unit. In general, the strong value of correlation coefficient between a workload metric and a component power means it is possible to predict the value of one from the other.

5.4.2 Components Power Consumption

Table 5.7 shows components average dynamic power consumption in watts for the HP and LP categories. The average dynamic power consumption of HP and LP categories is 80.0 W and 67.2 W, respectively.

The table shows a significant change in components power consumption across the two categories. The EU (25.3%), WCU (20.3%), and CP (16.3%) are the three most power consuming components for HP category and together consume about 62% of the total power. The next most power consuming component is the RF (14.0%). Since these components have a higher usage for kernels with high IPC, these components consume more power. It is interesting to know that the power consumed by the EU (10.4%), WCU (16.6%), and RF (6.4%) is far less for LP category compared to HP category. The largest fraction of power is consumed by the GM (21.4%) in the LP category. The CP and BP power consumption is the same in both categories because the activation power is always consumed in both categories. The NoC and MC consume more power in LP category because of increased activity of these units. In summary, we see that the power distribution is different across the two categories and the most power consuming components change across the two categories of the workloads.

5.4.3 Low Occupancy

In this section, we present bottlenecks investigation results for low occupancy category.

Table 5.8: Kernels limited by CTA limit [84]. © IEEE 2014

Kernel	IPC	Power (W)	Energy (mJ)	CTA size	Occupancy
BS	387.5	167.9	188.1	128	0.67
CS1	339.8	147.3	261.7	64	0.33
CS2	339.8	152.9	260.7	128	0.67
MS1	448.5	154.5	297.8	128	0.67
MC1	502.2	167.1	3.4	128	0.67
SS1	134.2	129.7	2.0	128	0.67
MCO1	446.9	141.2	52.5	64	0.33

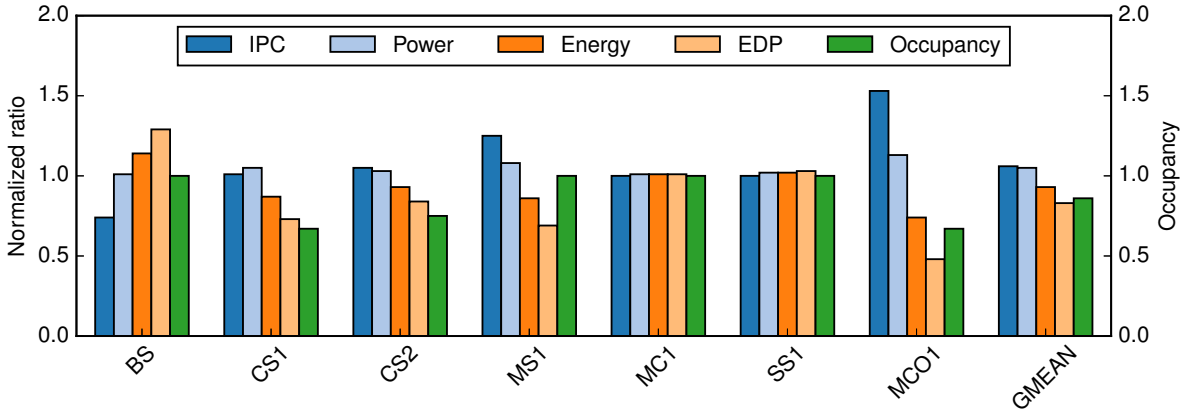
Limited by CTA Limit

Table 5.8 shows kernels whose occupancy is limited by the maximum limit of CTAs. The table shows the kernel, IPC, power, energy consumption, CTA size, and occupancy. The IPC for this category varies from 134.2 to 502.2 and the average IPC is 371.3, which is even less than 37% of the peak IPC. The table also shows that the occupancy varies from 0.33 to 0.67.

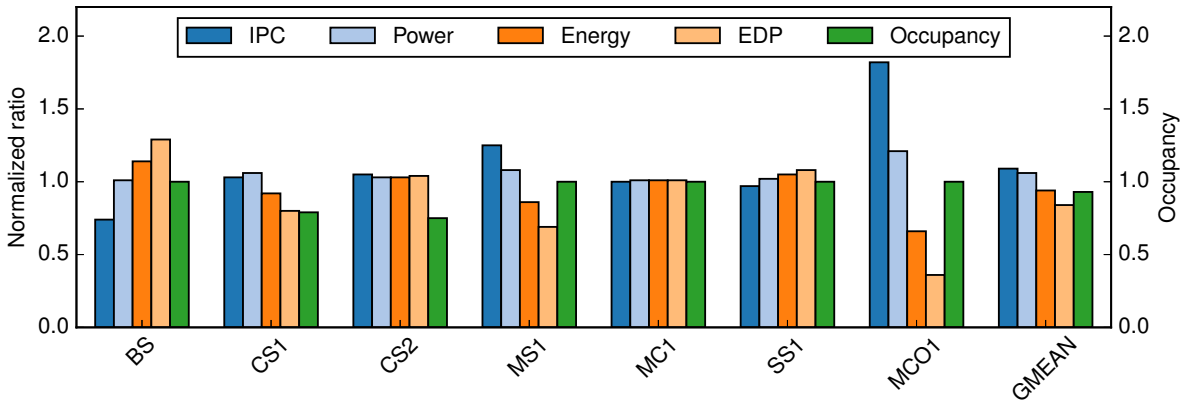
Table 5.8 shows that the smallest CTA size is 64 threads for the convolution kernel (CS1), and the chroma (MCO1) kernel of H264. These kernels require 24 CTAs per SM to have full occupancy ($24 \times 64 = 1536$ which is the maximum number of threads that can be allocated to an SM of NVIDIA GTX580). However, GTX580 has maximum limit of 8 CTAs. Thus, we increase the maximum number of CTAs from 8 to 24 in two steps to increase the occupancy.

Figure 5.3 shows IPC, power, energy, EDP normalized to baseline, and occupancy when the CTA limit is increased to 16 and 24. The figure also shows the geometric mean (GMEAN) of all kernels in the category. The average increase in IPC and power is 6% and 4%, respectively, when the CTA limit is increased to 16. The average energy consumption and EDP is decreased by 5% and 16%, respectively. The largest gain is 53% increase in IPC and 26% decrease in energy consumption for the MCO1 kernel. All kernels except BS either gain in IPC or have the same IPC. The reason for the decrease in IPC of BS kernel is that BS has high bandwidth utilization (74%) and the increase in occupancy adds to the already existing high bandwidth contention, and hence, IPC decreases. Figure 5.3a shows that the kernels CS1, CS2, and MCO1 still have occupancy <1, and thus, these kernels can gain from further increase in CTA limit. However, CS2 is now limited by shared memory and just increasing the CTA limit further will not help increasing the occupancy. We call such a kind of bottleneck as *second-order* bottleneck. Second order bottlenecks may occur after the elimination of first-order bottlenecks.

Figure 5.3b shows the IPC, power, energy, EDP, and occupancy when the CTA limit is increased to 24. Only the occupancy of the MCO1 and CS1 kernels increases further



(a) Maximum number of CTAs = 16



(b) Maximum number of CTAs = 24

Figure 5.3: IPC, power, energy, EDP, and occupancy of kernels limited by CTA limit [84]. © IEEE 2014

because all other kernels either already have full occupancy or show second-order bottlenecks after the CTA limit was increased to 16. The average increase in IPC and power is 9% and 6% over the baseline while the average decrease in energy consumption and EDP is 6% and 16%, respectively. The kernel CS1 now also has a second-order bottleneck of shared memory. The shared memory per SM is increased to 96KB to eliminate the second-order bottleneck of CS1, CS2.

Figure 5.4 shows that the kernels CS1 and CS2 also have full occupancy after the elimination of second-order bottleneck. Figure 5.4 does not show the kernels which already have full occupancy after the CTA limit is increased to 24. The average increase in IPC of CS1 and CS2 is 13% while the average reduction in energy consumption is 12% after the elimination of second-order bottleneck. At full occupancy, the average increase in IPC and power for the category is 11% and 7%, respectively. The average reduction in energy consumption and EDP is 9% and 23% compared to the baseline. The kernels MC1 and SS1 does not gain in performance even at full occupancy. MC1 is limited by memory

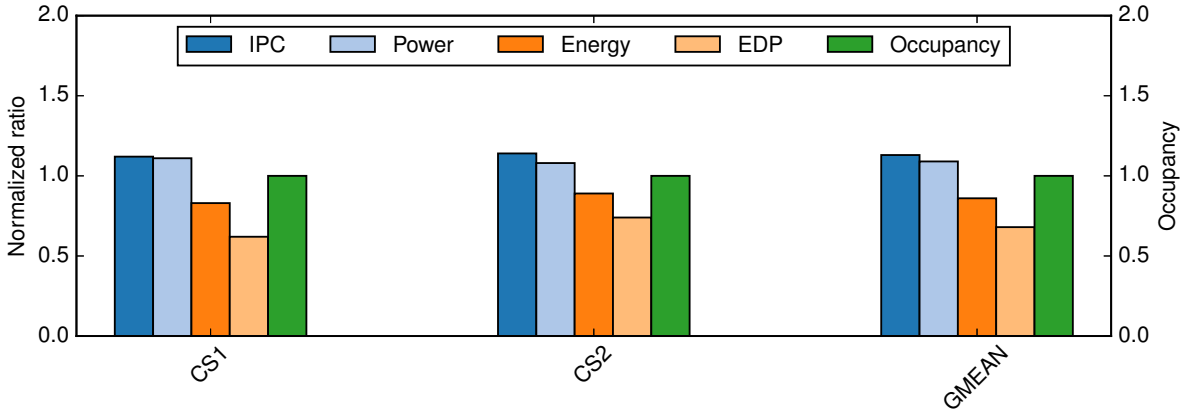


Figure 5.4: IPC, power, energy, EDP, and occupancy after the elimination of second-order bottleneck. These kernels were originally limited by CTA limit [84]. © IEEE 2014

Table 5.9: Kernels limited by registers [84]. © IEEE 2014

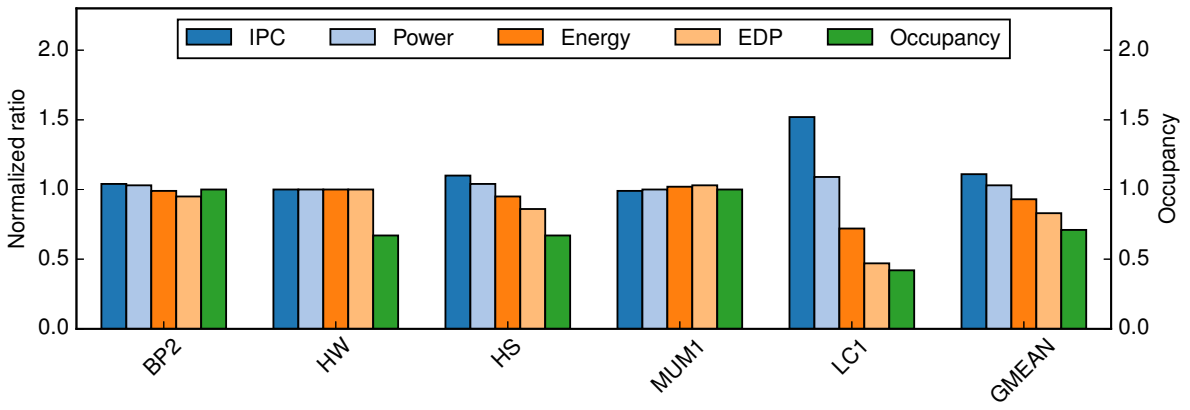
Kernel	IPC	Power(W)	Energy(mJ)	Registers/CTA	Occupancy
BP2	517.7	176.9	30.0	5.5K	0.83
HW	407.6	148.5	3124.4	14.0K	0.67
HS	493.5	154.5	41.9	8.5K	0.50
LC1	271.6	129.8	13283.5	16.0K	0.21
MUM1	26.9	146.8	714.0	6.0K	0.83

bandwidth and SS1 has a very low coalescing efficiency (6.7%) and hence, these kernels do not gain from the increased occupancy. MC1 gains from increase in memory bandwidth as shown in Section 5.4.5.

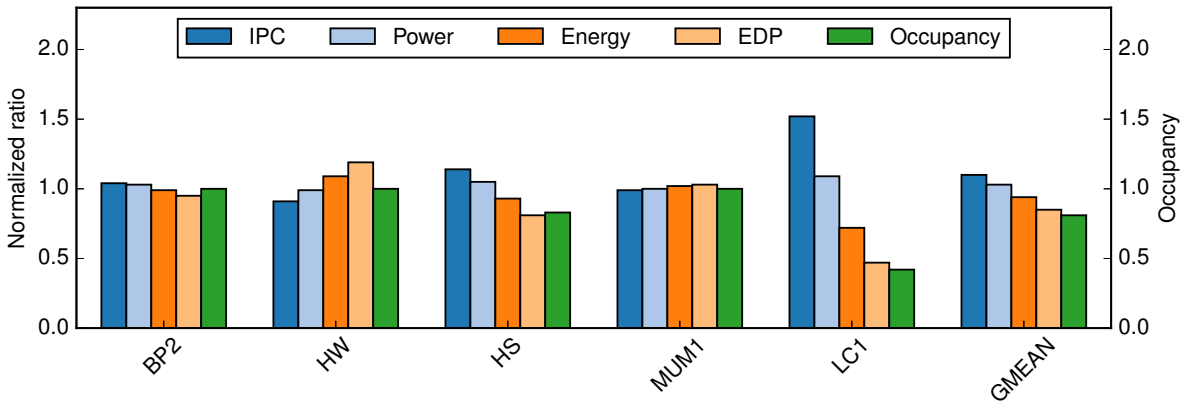
Limited by Registers

Table 5.9 shows the kernels whose occupancy is limited by registers. The table shows the kernel, IPC, power, energy consumption, registers used per CTA, and occupancy. The IPC in this category ranges from 26.9 to 517.7 and the average IPC is 343.4 which is 33.5% of the peak IPC. The table shows that the occupancy of these kernels varies from 0.21 to 0.83.

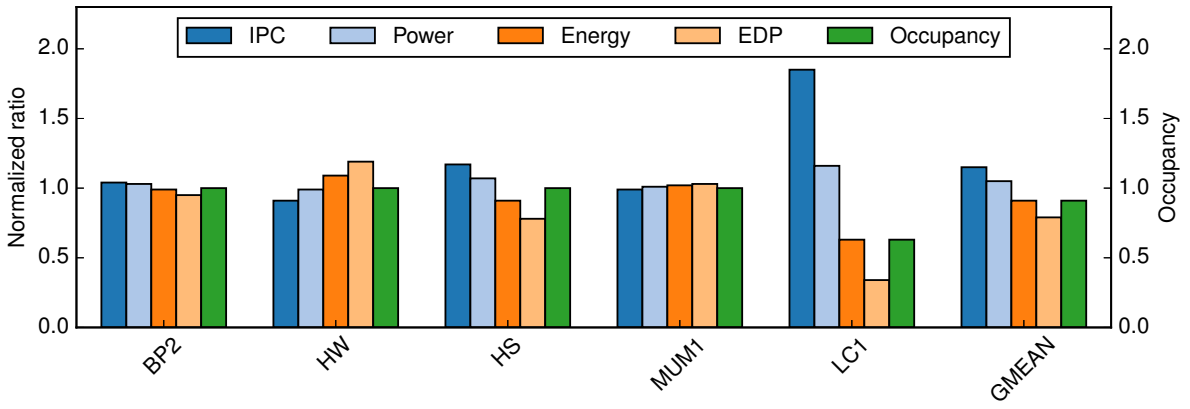
The kernel LC1 has the lowest occupancy and it requires 16K registers per CTA. The LC1 has 320 threads per CTA and requires 4.8 (1536/320) CTAs to reach full occupancy. However, the allocation of threads is done at CTA granularity, thus, the SM can hold a maximum of 4 CTAs in this case. The total number of registers required for 4 CTAs is 64K. However, we only present the results upto 56K registers because at this point all



(a) Registers = 40K



(b) Registers = 48K



(c) Registers = 56K

Figure 5.5: IPC, power, energy, EDP, and occupancy of kernels limited by registers [84]. © IEEE 2014

kernels either have full occupancy or a second-order bottleneck.

Figure 5.5 shows IPC, power, energy, EDP and occupancy when the number of registers per SM is increased to 40K, 48K, and 56K. The baseline configuration has 32K registers. Figure 5.5a shows that the kernels BP2 and MUM1 reach full occupancy after increasing the number of registers to 40K. The largest increase in IPC is 52% for the LC1 kernel, with the corresponding 28% decrease in energy consumption. The average increase in IPC and power is 11% and 3%, respectively, while the average decrease in energy consumption and EDP is 7% and 17%, respectively. The kernel HW reaches full occupancy when the number of registers is further increased to 48K, but HS and LC1 kernels are still limited by registers and have occupancy less than 1 as shown in Figure 5.5b. The average increase in IPC and power is 10% and 3%, respectively, while the average decrease in energy consumption and EDP is 6% and 15%, respectively.

The average increase in IPC and power consumption is 15% and 5%, respectively when the number of registers is further increased to 56K as shown in Figure 5.5c. The average decrease in energy consumption and EDP is 9% and 21%, respectively. The largest gain is 85% increase in IPC and 37% decrease in energy consumption for the LC1. Since the kernels BP2, MUM1, and HW already have full occupancy at 40K registers, these kernels do not gain from the increase in registers. The figure shows that all kernels except LC1 have full occupancy. At this point, the occupancy of LC1 is 0.63 and is also limited by a second-order bottleneck of shared memory. Hence, we further increase registers size to 64K and also shared memory to 64KB to eliminate the second-order bottleneck of LC1. The LC1 reaches its maximum achievable occupancy of 0.83 and continues to gain from increased occupancy. At full occupancy, the average increase in IPC and power for the category is 15% and 5%, respectively and the average reduction in energy consumption and EDP is 9% and 21% compared to the baseline. The kernels MUM1 and HW does not gain in performance even at full occupancy. MUM1 has high BW utilization (77.2%), low CE (14.9%) and low SU (52.2%) and it gains from increase in memory bandwidth as shown in Section 5.4.5. HW also has low CE (48.4%) and SU (79.6%).

Limited by Shared Memory

There is only one kernel (STO) which is limited by shared memory. STO is used to accelerate a set of hashing functions used in distributed storage systems. The IPC, power, energy, CTA size, shared memory per CTA, registers per CTA, and occupancy is 405.7, 133.8 (W), 51.6 (mJ), 128, 15.9KB, 4.2K, and 0.25, respectively. The IPC is well below the peak IPC and the occupancy is only 25%.

The reason for the low occupancy is that STO is using almost 16KB shared memory per CTA. Since the baseline GPU has 48KB shared memory, no more than 3 CTAs can be allocated simultaneously. Moreover, the CTA size is only 128 which means STO also needs 12 CTAs to achieve full occupancy. Also, STO needs 4.2K registers per CTA. Therefore, at some point, STO will be limited by both CTA and registers limit when the shared memory is increased.

Figure 5.6 shows the change in performance of STO when the shared memory is increased to 96KB and 144KB, respectively. There is a 49% increase in IPC and a 26%

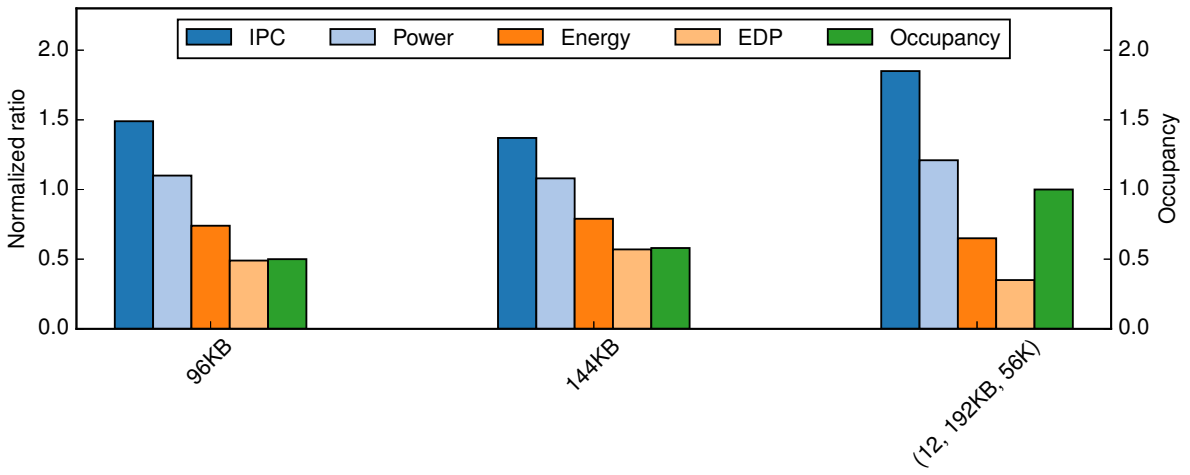


Figure 5.6: IPC, power, energy, and occupancy of STO kernel limited by shared memory. The first two groups of bars show the results after the elimination of first-order bottleneck, while the third group of bars shows the results after the elimination of second-order bottlenecks [84]. © IEEE 2014

reduction in energy consumption when the shared memory size is increased to 96KB. The occupancy is doubled to 0.5. There is only a slight increase in occupancy (0.58) when the shared memory is further increased to 144KB because STO is now limited by second-order bottleneck of registers. To eliminate the second-order bottlenecks, the number of CTAs is increased to 12, shared memory to 192KB, and registers to 56K (12, 192KB, 56K). Figure 5.6 also shows the performance of STO kernel when all second-order bottlenecks are eliminated to achieve full occupancy. At full occupancy, the STO kernel gained 85% increase in IPC with just 21% more power consumption. Moreover, we have 35% reduction in energy consumption and 65% less EDP compared to the baseline.

Multiple Bottlenecks

There are two kernels (MCO2 and MD) which are limited by multiple bottlenecks to begin with. MCO2 is a luma kernel of motion compensation part of H264 decoder and it is limited by CTA limit and shared memory. The IPC, power, energy, CTA size, shared memory per CTA and occupancy of MCO2 is 365.6, 135.6 (W), 183.0 (mJ), 64, 6KB, and 0.33, respectively. The kernel has low IPC as well as low occupancy and needs 24 CTAs and 144KB shared memory to have full occupancy. Figure 5.7 shows the results of the MCO2 kernel when the number of the CTAs is increased to 16 and shared memory is increased to 96KB (16, 96K). The number of the CTAs is further increased to 24 and shared memory to 144KB (24, 144K) to have full occupancy. At full occupancy, the IPC is increased by 39% with 8% more power consumption and energy consumption and EDP is decreased by 22% and 44%, respectively.

MD is used to calculate the physical movements of molecules and atoms and is limited

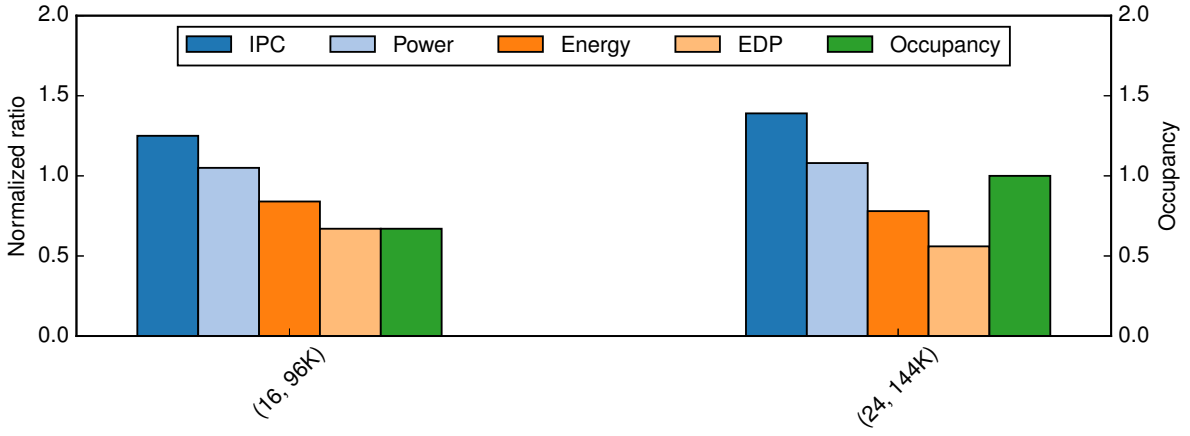


Figure 5.7: IPC, power, energy, EDP and occupancy of the MCO2 kernel limited by CTA limit and shared memory [84]. © IEEE 2014

by shared memory and registers. The IPC, power, energy, CTA size, shared memory per CTA, registers per CTA, and occupancy of MD is 142.7, 138.9 (W), 24937.0 (mJ), 128, 7.1KB, 4.62K, and 0.5, respectively. We increase the shared memory to 64KB and registers to 48K to increase the occupancy. The occupancy increases to 0.67, but there is no gain in IPC. MD is now limited by second-order bottleneck of CTA limit. We further increase shared memory to 96KB, registers to 56K, and number of CTAs to 12 to have full occupancy. At full occupancy, the IPC and power consumption is increased by 2% and 5%, respectively. The energy and EDP also increase by 3% and 1%, respectively, which shows MD does not gain from the increased occupancy. Further investigation reveals MD has very low coalescing efficiency (13%) which could limit its performance.

5.4.4 Full Occupancy

Table 5.10 shows the kernels having full occupancy but low performance. The table shows kernel, IPC, power, and energy consumption of these kernels. The IPC in this category ranges from 8.0 to 468.5. The average IPC is 208.2 which is less than 21% of the peak IPC. Since all kernels in this category have full occupancy, increasing occupancy is not a solution. We analyze if bandwidth utilization (BW), coalescing efficiency (CE), or SIMD utilization (SU) is a bottleneck for low performance of these kernels.

Figure 5.8 shows the percentage of BW, CE and SU for the full occupancy kernels. The high BW utilization, low CE, or low SU can severely limit the performance of GPU kernels [85, 106, 50, 145]. We see that most of the kernels have at least one problem. Figure 5.8 shows that the kernels CFD1, CFD2, CFD3, FWT1, FWT2, KM1, KM2, MUM2, SP, SCAN1, SCAN2, SRAD1_1, SRAD1_2, SRAD1_4, and VA have high BW utilization and these kernels could be performing low due to high bandwidth demands.

Figure 5.8 shows that the kernels HG3 (3%), KM2 (6%), MUM2 (4%), SCAN3 (5%) and MT2 (11%) have very low CE. Also kernels BT1, BT2, BFS1, SCAN1, SCAN2,

Table 5.10: Kernels with full occupancy and low performance [84]. © IEEE 2014

Kernel	IPC	Power (W)	Energy (mJ)	Kernel	IPC	Power (W)	Energy (mJ)
BT1	432.8	144.7	133.5	SCAN2	286.7	161.1	96.3
BT2	467.0	149.4	123.6	SCAN3	8.0	116.6	10.10
BFS1	21.8	149.9	195.2	SRAD1_1	331.0	163.0	17.1
BFS2	276.5	151.8	10.0	SRAD1_2	370.1	170.3	9.3
CFD1	62.3	144.1	16.7	SRAD1_3	273.1	122.6	9.0
CFD2	184.5	156.6	4.7	SRAD1_4	148.1	148.6	5.9
CFD3	71.0	141.9	5.9	SRAD2_1	304.7	154.0	467.7
FWT1	100.1	154.4	189.7	SRAD2_2	167.0	137.2	452.8
FWT2	264.5	168.4	79.6	MT1	359.9	154.8	3.6
HG3	55.9	125.1	9.4	MT2	46.3	128.2	16.2
KM1	468.5	181.7	741.0	MT3	417.3	156.8	3.5
KM2	11.0	153.2	2216.5	MT4	165.8	134.1	6.6
MUM2	35.6	152.0	681.4	MT5	320.1	152.9	3.5
SP	182.3	141.5	20.7	MT6	144.9	132.5	6.8
VA	171.9	147.4	11.5	MT7	155.8	133.1	6.9
SCAN1	120.4	158.0	57.2	MT8	238.7	151.1	3.1

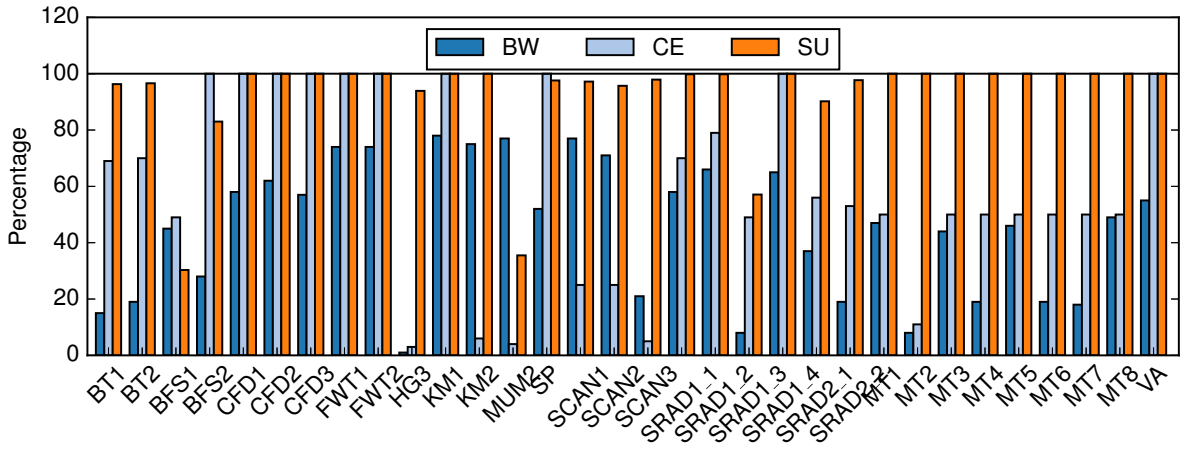


Figure 5.8: Bandwidth utilization (BW), Coalescing efficiency (CE), and SIMD utilization (SU) of full occupancy kernels [84]. © IEEE 2014

SRAD1_1, SRAD1_2, SRAD1_3, SRAD2_1, SRAD2_2, MT1, MT3, MT4, MT5, MT6, MT7, and MT8 have less than 100% CE. We think low CE could be a reason for their low performance. As low CE results in more than one memory transaction for each memory access from a warp, resulting in higher pressure on the memory subsystem that could lead to contention and longer latency, and thus, can limit the performance. For

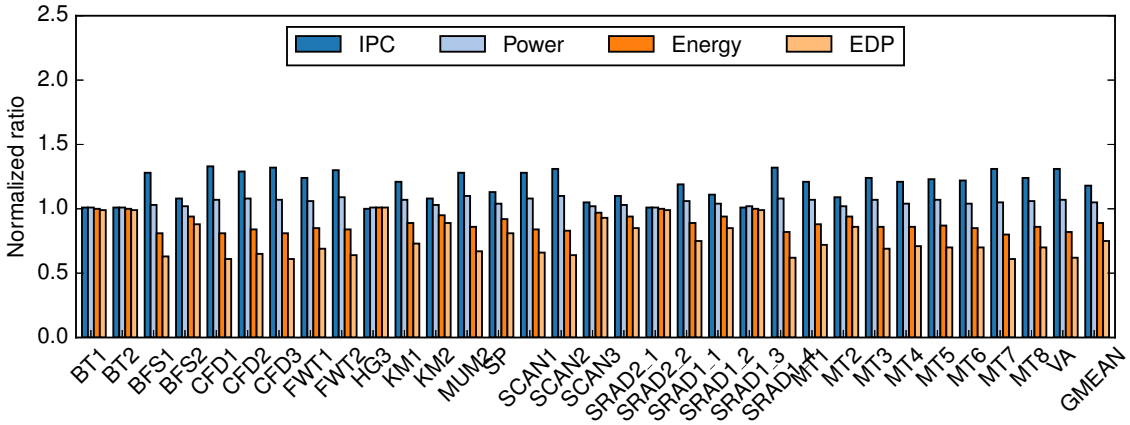
example, a coalescing efficiency of 3% means that each memory access from a warp is fully divergent and results in 32 memory transactions, injecting 32× more memory requests in the memory subsystem.

Another factor that could also effect the performance of full occupancy kernels is low SU. The low SU is caused by branch divergence that leads to underutilized SIMD cores and low performance. Unlike high BW and low CE, which are indicators that these factors could limit the performance, SU is a more direct measure of bottleneck because low SU directly implies low performance. For example, a kernel having 50% SU can never have IPC more than 50% of the peak IPC. Figure 5.8 shows that kernels BFS1 (30.3%) and MUM2 (35.5%) have very low SU and kernels BT1, BT2, BFS2, HG3, SP, SCAN1, SCAN2, SCAN3, SRAD1_3, SRAD2_1, SRAD2_2 have less than 100% SU. The figure also shows that some kernels such as BFS1, MUM2, SRAD1_3 have multiple bottlenecks (high BW, low CE, or low SU) and hence, these kernels could have low performance due to the combined effect.

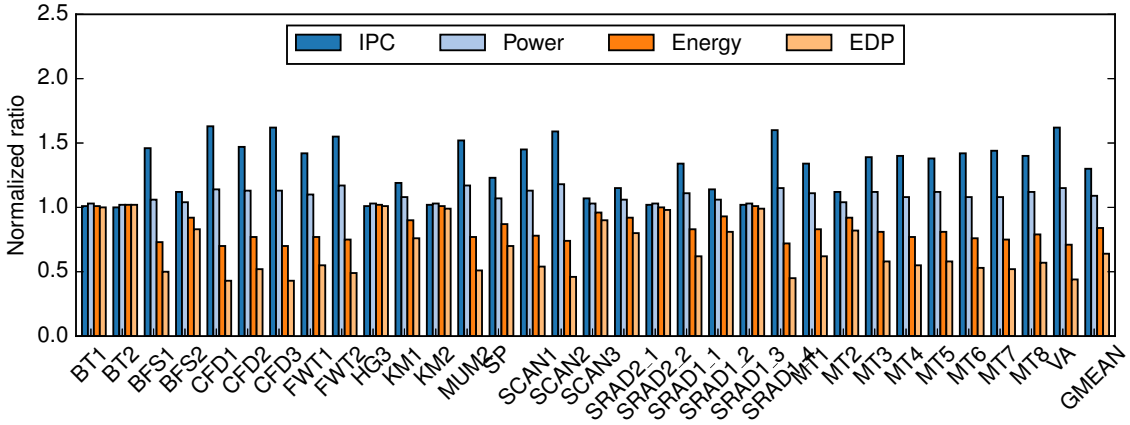
To determine if high BW or low CE is actually limiting the performance of full occupancy kernels or not, we perform following experiments. For testing if memory bandwidth is a bottleneck or not, we increase the memory bandwidth and study its effect on performance. For testing if low CE is a bottleneck or not, we simulate a system with perfect coalescing and study its effect on performance.

Limited by Memory Bandwidth

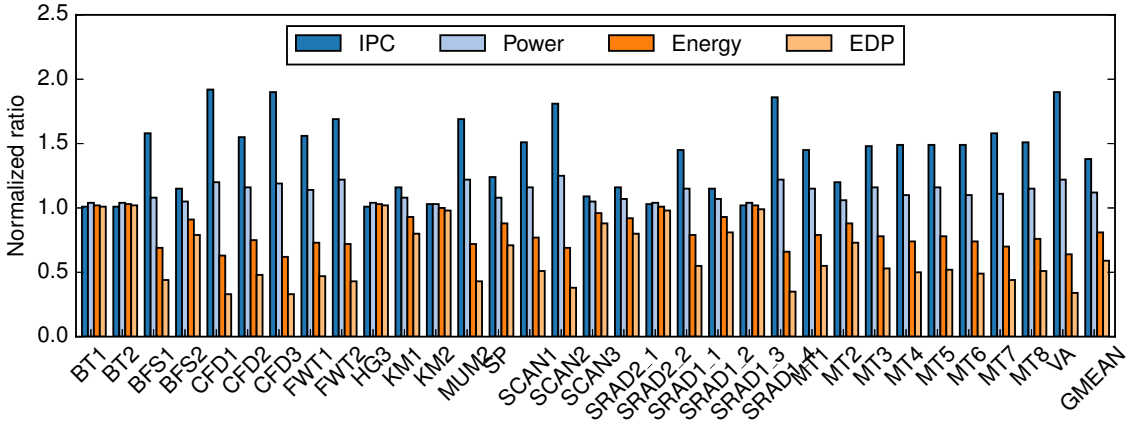
Figure 5.9 shows IPC, power, energy and EDP of the full occupancy category kernels normalized to the baseline when memory bandwidth is increased by 2×. We double the memory bandwidth by doubling the DRAM frequency, incrementing 33.3% at a time and study the change in performance at each increment. The baseline configuration has memory bandwidth of 192.4 GB/s. Thus, we increase memory bandwidth to 255.9 GB/s, 319.4 GB/s, and 384.8 GB/s in three steps. Figure 5.9a shows IPC, power, energy, and EDP of full occupancy kernels when the memory bandwidth is increased to 255.9 GB/s. The average increase in IPC is 18% with 5% more power consumption. Moreover, we see a 11% average reduction in energy consumption and 25% less EDP compared to the baseline. Figure 5.9b shows that kernels gain performance from further increase in memory bandwidth. The average increase in IPC is 30% with only 9% increase in power consumption, while the average decrease in energy consumption and EDP is 16% and 36% compared to the baseline. Figure 5.9c shows that most of the kernels continue to gain from increase in memory bandwidth. The average increase in IPC and power consumption is 38% and 12%, respectively, while the average decrease in energy consumption and EDP is 19% and 41%, respectively at 384.8 GB/s. The kernels BT1, BT2, HG3, SCAN3, and SRAD1_3 gain very low (average 1.3%) from the increase in memory bandwidth because these kernels have low BW utilization (average 13%), low CE (average 39%), and low SU (average 88%).



(a) Bandwidth = 255.9 GB/s



(b) Bandwidth = 319.4 GB/s



(c) Bandwidth = 384.8 GB/s

Figure 5.9: IPC, power, energy, and EDP of full occupancy kernels when memory bandwidth is increased by 2× in three increments [84]. © IEEE 2014

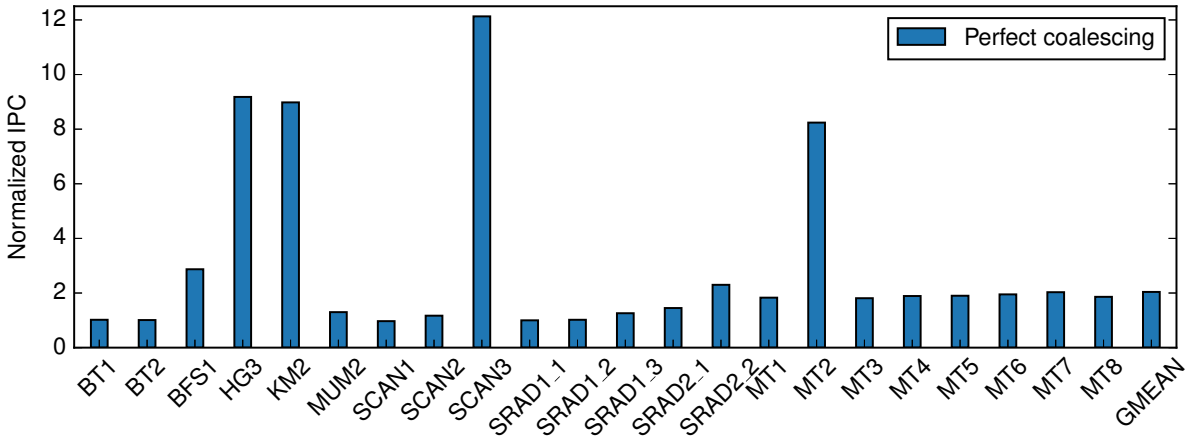


Figure 5.10: IPC of low CE kernels with perfect coalescing.

Limited by Coalescing Efficiency

To get an estimate of the performance loss due to low CE, we modify gpgpu-sim to have perfect coalescing so that one memory transaction is generated for each warp when the data size is less than or equal to 32-bit or half-warp when the data size is 64-bit. This gives us an estimate of the possible performance gain in case all memory accesses were perfectly coalesced under the assumption that this can be done at the hardware or application level. For instance, recent work has shown that coalescing efficiency can be improved by using compiler-assisted tools, kernel fusion, optimized data layout [47, 7, 96, 162]. Figure 5.10 shows normalized IPC of the low CE kernels with perfect coalescing over the baseline configuration. The figure shows that the kernels HG3 (9.2×), KM2 (9.0×), SCAN3 (12.1×), and MT2 (8.2×) can gain very high from perfect coalescing. MUM2 gains less because it also has very low SU (35.5%) and high BW utilization (77%). The average estimated increase in IPC is 2× for all kernels, which shows that low CE is a serious performance bottleneck. As low CE effects different levels of the memory hierarchy and system resources, the overall estimated performance gain could be a cumulative effect of, for example, decrease in L1 and L2 cache miss rate, reduce NoC traffic, less memory bandwidth requirements, less queuing etc. It remains a future work to quantify the exact gain from different sources.

5.4.5 Performance at the Combined Configuration

In Section 5.4.3 and Section 5.4.4, we presented bottleneck investigation category-wise. Ideally, we would build a GPU with enough resources so that all kernels achieve optimal performance. Practically, however, it is impossible to build such a GPU due to the area and power demands of the combined resources. Thus, we need to find a design point which provides benefits to most of the kernels. In this section, we evaluate such a design point. We use the following greedy approach: For each category of kernels, we find an optimal

Table 5.11: EDP optimal point for each category [84]. © IEEE 2014

Category	Optimal Point
Limited by CTA limit	CTA = 16
Limited by registers	Registers = 56K
Limited by shared memory	Shared memory = 96KB
Multiple bottlenecks	CTA = 24, shared memory = 144KB
Full occupancy	Memory bandwidth = 384.8 GB/s

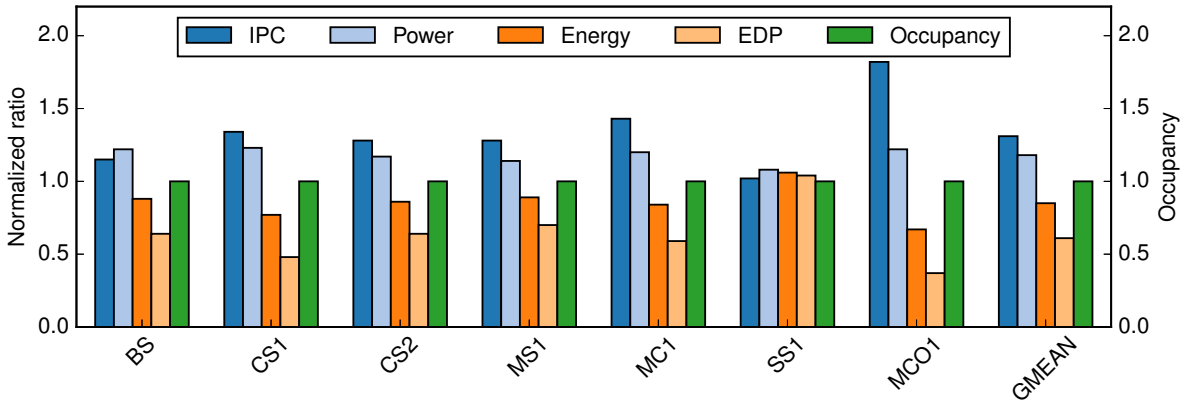


Figure 5.11: IPC, power, energy, EDP, and occupancy of kernels limited by CTA limit at the combined configuration [84]. © IEEE 2014

point using maximum reduction in EDP as the selection criterion. We consider category-wise results up to the first-order bottlenecks. Table 5.11 shows the EDP optimal point for each category. Then, we combine category EDP optimal points to derive the combined configuration (CTA = 24, registers = 56K, shared memory = 144KB, memory bandwidth = 384.8 GB/s). We choose a larger value of a resource when the resource is common but has different values in two categories to keep the category-wise gains unaffected. For example, the number of CTAs is 16 in CTA limited category and 24 in multiple bottlenecks category and we choose CTAs to be 24 for the combined configuration. This approach may result in a suboptimal solution, however, it allows us to evaluate the effect of all modifications on various categories.

Figure 5.11 shows the performance of kernels limited by CTA at the combined configuration. The average increase in IPC and power is 31% and 18%, respectively, while the average reduction in energy consumption and EDP is 15% and 39%, respectively. The increase in performance and energy reduction is higher than the category level and is mainly due to the elimination of second-order bottlenecks and increased bandwidth.

Figure 5.12 shows the performance of kernels limited by registers at the combined

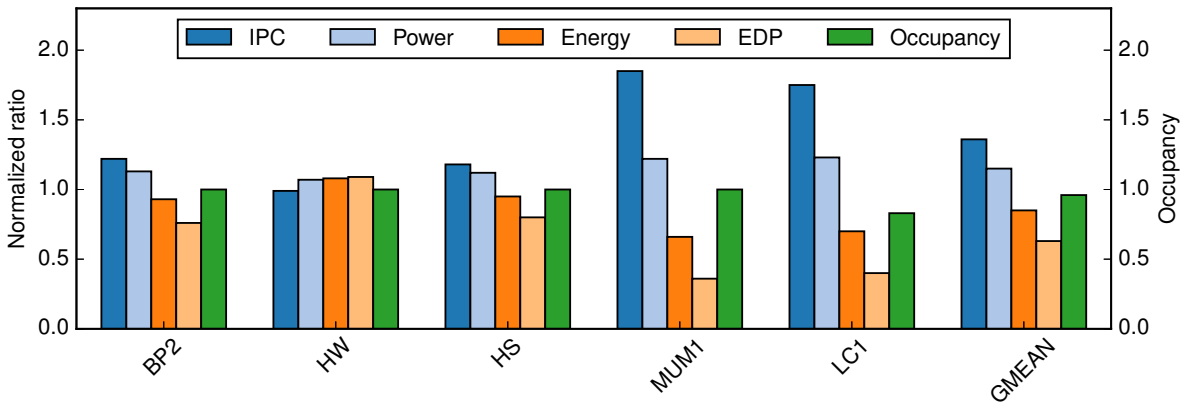


Figure 5.12: IPC, power, energy, EDP, and occupancy of kernels limited by registers at the combined configuration [84]. © IEEE 2014

configuration. The average gain in IPC is 36% which is higher than the average gain at the category level (15%). The average reduction in energy consumption and EDP is 15% and 37% which is also higher than the category-wise gain. The higher gain in performance and energy reduction at the combined configuration shows the registers limited kernels also gain from the increased bandwidth.

In limited shared memory category, STO kernel has 56% increase in IPC and 26% decrease in energy consumption compared to the baseline at the combined configuration. In the multiple bottlenecks category, MD kernel does not gain in IPC even at the combined configuration due to low CE and the performance of MCO2 kernel is the same as at the category level because it has low BW utilization (3.5%) and hence does not benefit from increased bandwidth at the combined configuration.

We also conducted experiments for full occupancy category kernels at the combined configuration. The average gain in IPC for the full occupancy kernels at the combined configuration is 38% which is identical to the gain at the category level. This shows that kernels in this category do not gain from other architectural changes made to increase the occupancy. This is expected as this category already had enough threads due to full occupancy. The average reduction in energy consumption and EDP is 18% and 40%, respectively, which is slightly less than the reduction at the category level. This is caused by the increase in static power due to the increased size of other architectural components such as registers, shared memory.

indicated by higher average IPC (405.1). peak IPC.

Table 5.12 shows components dynamic power consumption for LP kernels at the baseline (LP old), combined configuration (LP new) and the ratio between them. The component power consumption of RF (48%), EU (35.0%), WCU (13%), LSU (48%), NoC (39%), MC (39%), and GM (57%) increased compared to the baseline. This is because the bottlenecks elimination resulted in better utilization of resources which is indicated by higher average IPC (35.5%) compared to the baseline. The power consumption of BP and CP remains

Table 5.12: Components dynamic power consumption (W) for LP category kernels at the baseline (LP old), combined configuration (LP new) and their ratio [84]. © IEEE 2014

	RF	EU	WCU	BP	LSU	CP	NoC	MC	GM
LP old	4.3	7.0	11.2	3.8	0.9	13.0	4.8	7.8	14.4
LP new	6.4	9.5	12.6	3.7	1.4	13.0	6.7	10.8	22.6
Ratio	1.5	1.4	1.1	1.0	1.5	1.0	1.4	1.4	1.6

almost the same because the activation power consumption remains the same.

5.5 Summary

In this chapter, we studied the energy efficiency of several kernels (68) and classified them into HP and LP categories depending on the IPC. The HP and LP categories have 21 and 47 kernels, respectively. The average IPC for the former is 741 and 250 for the later, which is less than 25% of the peak IPC of the simulated GPU. Surprisingly, more than 69% of the kernels belong to LP category. The average energy per instruction for the HP and LP category is 0.27 nJ and 2.01 nJ, respectively. The later is 7.5× less energy efficient compared to the former, a huge difference that is an obstacle for the future growth of high performance computing and far away from the exascale aim of 10 pJ per instruction.

We also studied the power consumption of GPUs at the component level for the two categories. The distribution of the power consumption is different across the two categories. For example, for kernels in the HP category, EU (25.3%), WCU (20.3%), and CP (16.0%) are the three most power consuming components and together consume about 62% power. For kernels in the LP category, GM (21.4%), CP (19.3%), and WCU (16.6%) are the three most power consuming components.

To investigate the performance bottlenecks of LP category, we divided the LP category kernels into low occupancy (15 kernels) and full occupancy (32 kernels) categories. We further categorized the low occupancy category kernels into different categories depending on the resource their occupancy is limited by. We increased the occupancy by increasing the resource limiting the occupancy. The results showed that most of the kernels with low occupancy gain in performance and energy efficiency when the occupancy increased. For example, at full occupancy, the average increase in IPC, the average reduction in energy consumption and EDP is 11%, 9% and 23%, respectively, for the CTA limited kernels. The average increase in IPC, the average reduction in energy consumption and EDP is 15%, 9% and 21%, respectively, for the registers limited kernels. The results showed that high occupancy is an important factor for both high performance and energy efficiency but occupancy alone is not sufficient to achieve the desired performance.

We further showed that full occupancy kernels have low performance either due to high

BW utilization or low CE or low SU. The full occupancy kernels on an average have 38% increase in IPC, 19% decrease in energy consumption and 41% reduction in EDP at 2× the bandwidth compared to the baseline. The results showed that bandwidth optimization techniques will significantly increase the performance. We also showed that many kernels in the full occupancy category are severely limited by low CE and perfect coalescing could increase the average performance by 2×. The kernels with low CE can also gain from the memory bandwidth optimizations techniques because low CE exerts high pressure on the memory subsystem by injecting several memory requests per warp.

Ideally, we would build a GPU with enough resources so that all kernels achieve optimal performance, however, practically it is impossible to build such a GPU. Thus, we derived an architectural configuration that benefits most of the kernels by combining the EDP optimal point for each category. The results showed further gain in performance and reduction in energy consumption at this configuration compared to the baseline. At the architectural point, 12 kernels achieved IPC greater than 50.5% of the peak IPC.

In the next chapter, we will propose an entropy encoding based memory compression technique for GPUs to reduce memory bandwidth requirements. We will discuss and address the challenges of an entropy encoding based memory compression for GPUs.

6 Entropy Encoding Based Memory Compression Technique

The work presented in this chapter was previously published: S. Lal, J. Lucas, and B. Juurlink, "E²MC: Entropy Encoding Based Memory Compression for GPUs," in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), © 2017 IEEE.

In the last chapter, we studied the energy efficiency of GPU workloads and showed that there are several workloads with low performance and low energy efficiency that can gain from the increase in memory bandwidth. In this chapter, we propose an entropy encoding based memory compression technique to increase the memory bandwidth. We discuss and address the key challenges of an entropy encoding based memory compression for GPUs.

6.1 Introduction

GPUs are high throughput devices which use fine-grained multi-threading to hide the long latency of accessing off-chip memory [72]. GPUs use single instruction multiple thread (SIMT) execution model to execute a group of threads concurrently. The grouping of threads into fixed size batch is called a warp in NVIDIA terminology or a wavefront in AMD terminology. A GPU warp scheduler chooses a new warp from a pool of ready warps if the current warp is waiting for data from memory. This is effective for hiding the memory latency and keeping the cores busy for compute-bound benchmarks. However, for memory-bound benchmarks, all warps are usually waiting for data from memory and performance is limited by off-chip memory bandwidth. Performance of memory-bound benchmarks increases when additional memory bandwidth is provided. Figure 6.1 shows the speedup of memory-bound benchmarks when the off-chip memory bandwidth is increased by 2×, 4×, and 8×. The average speedup is close to 2×, when the bandwidth is increased by 8×. An obvious way to increase memory bandwidth is to increase the number of memory channels and/or their speed. However, technological challenges, cost, and other limits restrict the number of memory channels and/or their speed [113, 1]. Moreover, research has already shown that memory is a significant power consumer [90, 99, 84] and increasing the number of memory channels and/or frequency elevates this problem. Clearly, alternative ways to tackle the memory bandwidth problem are required.

A promising technique to increase the effective memory bandwidth is memory compression. However, compression incurs overheads such as (de)compression latency which need to be addressed, otherwise the benefits of compression could be offset by its overhead.

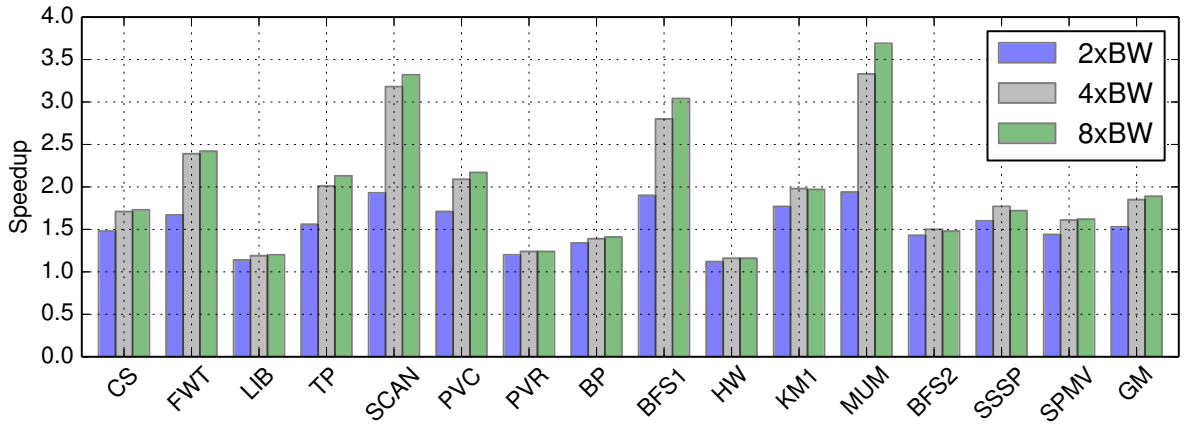


Figure 6.1: Speedup with increased memory bandwidth [85]. © 2017 IEEE

Fortunately, GPUs are not as latency sensitive as CPUs and they can tolerate latency increases to some extent. Moreover, GPUs often use streaming workloads with large working sets that cannot fit into any reasonably sized cache. The higher throughput of GPUs and streaming workloads mean bandwidth compression techniques are even more important for GPUs than CPUs. The differences in the GPU and CPU architecture offer new challenges which need to be tackled and new opportunities which can be harnessed.

Most existing memory compression techniques exploit simple patterns for compression and trade low compression ratios for low decompression latency. For example, Frequent-Pattern Compression (FPC) [6] replaces predefined frequent data patterns, such as consecutive zeros, with shorter fixed-width codes. C-Pack [26] utilizes fixed static patterns and dynamic dictionaries. Base-Delta-Immediate (BDI) compression [130] exploits value similarity. While these techniques can decompress with few cycles, their compression ratio is low, typically only 1.5×. All these techniques originally targeted CPUs and hence, traded low compression for lower latency.

As GPUs can hide latency to a certain extent, more aggressive entropy encoding based data compression techniques such as Huffman compression seems feasible. While entropy encoding could offer higher compression ratios, these techniques also have inherent challenges which need to be addressed. The main research questions are 1) How to estimate probability? 2) What is an appropriate symbol length for encoding? And 3) How to keep the decompression latency low? In this chapter, we address these key challenges and propose to use the Entropy Encoding based Memory Compression (E^2MC) for GPUs.

We use both offline and online sampling to estimate probabilities and show that small online sampling results in compression comparable to offline sampling. While GPUs can hide a few tens cycles of additional latency, too many can still degrade their performance [126]. We reduce the decompression latency by decoding multiple codewords in parallel. Although parallel decoding reduces the compression ratio because additional information needs to be stored, we show that the reduction is not much as it is mostly hidden by the memory access granularity (MAG). MAG is the amount of data read from

or written to a memory by a single read or write command and it is a multiple of burst length and bus width. For example, MAG for GDDR5 is 32B resulting from burst length of 8 and bus width of 32-bit.

As GPUs are high throughput devices, the compressor and decompressor should also provide high throughput. Therefore, we also estimate the area and power needed to meet the high throughput requirements of GPUs.

In summary, we make the following contributions in this chapter:

- We propose an entropy encoding based memory compression technique for GPUs that delivers higher compression ratio and performance gain than state-of-the-art techniques.
- We address the key challenges of probability estimation, choosing a suitable symbol length for encoding, and low decompression latency by parallel decoding with a small loss of compression ratio.
- We provide a detailed analysis of the effects of memory access granularity on the compression ratio.
- We analyze the high throughput requirements of GPUs and provide an estimate of area and power needed to support such high throughput.

This chapter is organized as follows. In Section 6.2, we present E^{2}MC in detail. Section 6.3 explains the experimental setup and experimental results are presented in Section 6.4. Finally, we summarize the contributions of this chapter in Section 6.5.

6.2 Huffman-based Memory Bandwidth Compression

First, we provide an overview of a system with entropy encoding based memory compression (E^{2}MC) for GPUs and its key challenges and then in the subsequent sections address these challenges in detail.

6.2.1 Overview

Figure 6.2 shows an overview of a system with main components of the E^{2}MC technique. The memory controller (MC) is modified to integrate the compressor, decompressor and metadata cache (MDC). Depending on the memory request type, either it needs to pass through the compressor or it can directly access the memory. A memory write request passes through the compressor while a read request can bypass the compressor. The MDC is updated with the size of the compressed block and finally the compressed block is written to memory. A memory read request first accesses the MDC to determine the memory request size and then fetches that much data from memory. (De)compression takes place in the MC and is completely transparent to the L2 cache and the streaming multiprocessors. The compressed data is stored in the DRAM. However, the goal is not to increase

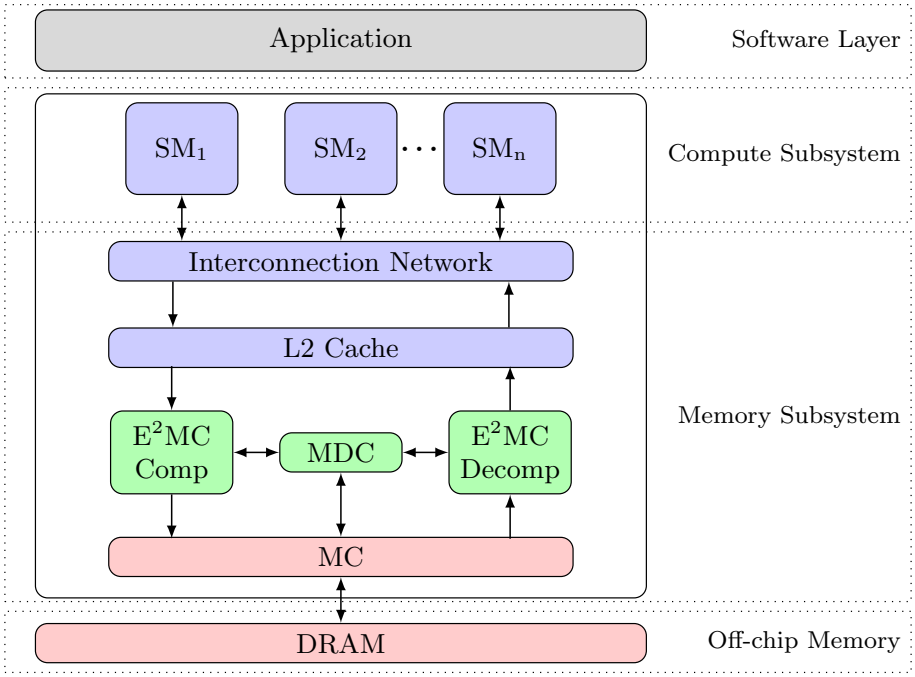


Figure 6.2: System overview with compression components.

the effective capacity of the DRAM (Dynamic Random Access Memory) but to increase the effective off-chip memory bandwidth similar to [146]. Hence, a compressed block is still allocated the same size in DRAM, although it may require less space. Like [160], E^{2}MC requires compression when the data is transferred from CPU to GPU to initialize the MDC. As the (de)compressors are integrated in the MC, data is compressed while transferring from CPU to GPU and vice versa.

6.2.2 Huffman Compression and Key Challenges

Huffman compression is based on the evidence that not all symbols have same probability. Instead of using fixed-length codes, Huffman compression uses variable-length codes based on the relative frequency of different symbols. A fixed-length code assumes equal probability for all symbols and hence assigns same length codes to all symbols. Fixed length codes are not the best choice when a data stream contains symbols with highly variable frequency. In contrast to fixed-length codes, Huffman compression is based on the principle to use fewer bits to represent frequent symbols and more bits to represent infrequent symbols. On average, a variable length code needs less number of bits and hence are quite useful to achieve compression. In general, compression techniques based on variable-length codes can provide high compression ratio, but they also have high overhead in terms of latency, area, and power [10]. Moreover, to achieve high compression and performance, certain key challenges should be addressed.

The first challenge is to find an appropriate *symbol length* (SL). The choice of SL is a very

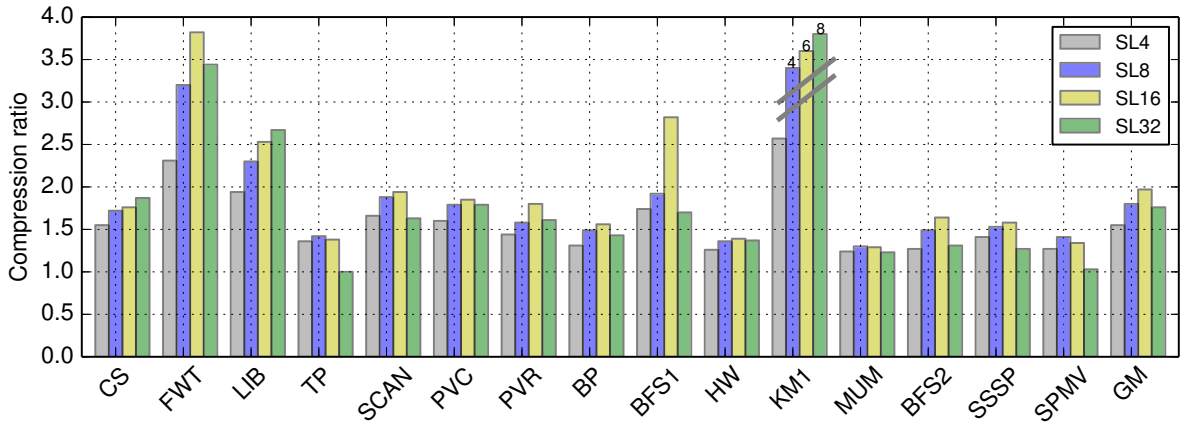


Figure 6.3: Compression ratio for different symbol lengths [85]. © 2017 IEEE

important factor as it affects compression ratio, decompression latency and hardware cost. We evaluate the trade-offs of using different SLs (4, 8, 16, 32-bit). The second challenge is *accurate probability estimation*. We perform both offline and online sampling of data to estimate the probability of symbols and show that it is possible to achieve compression ratio comparable to offline sampling with small online sampling. The third important factor is *low decompression latency*, which affects the performance gain. We reduce the decompression latency by decoding in parallel. In the following sections, we discuss these challenges in detail.

6.2.3 Choice of Symbol Length

We encode with different SLs of 4, 8, 16, and 32-bit and evaluate their trade-offs to make sure that not only the compression ratio but other aspects are also compared. Figure 6.3 shows the compression ratio for different SLs (see Section 6.3 for details of benchmarks). It can be seen that 16-bit encoding yields the highest compression ratio for GPUs. This result is in contrast to [10] where it was shown that 16 and 32-bit encodings yield almost same compression ratio for CPUs. The next highest compression ratio is provided by 8-bit symbols. For some benchmarks (TP, MUM, SPMV), 8-bit encoding offers the highest compression ratio. GPUs often operate on FP values, but INT values are also not uncommon. Most of the benchmarks in MARS [46] and Lonestar [81] suites are INT. Often smaller symbols are more effective for FP, while longer symbols are more effective for INT and on average 16-bit symbols provide good trade-off for both.

6.2.4 Probability Estimation

Figure 6.4 shows the different phases of an entropy encoding based compression technique. In the probability estimation phase, frequencies of the different symbols are collected. Based on frequencies, variable-length codes are assigned to different symbols. In the

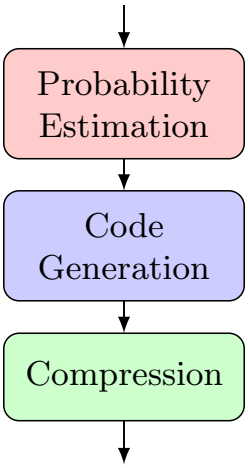


Figure 6.4: Phases of entropy encoding based compression [85]. © 2017 IEEE

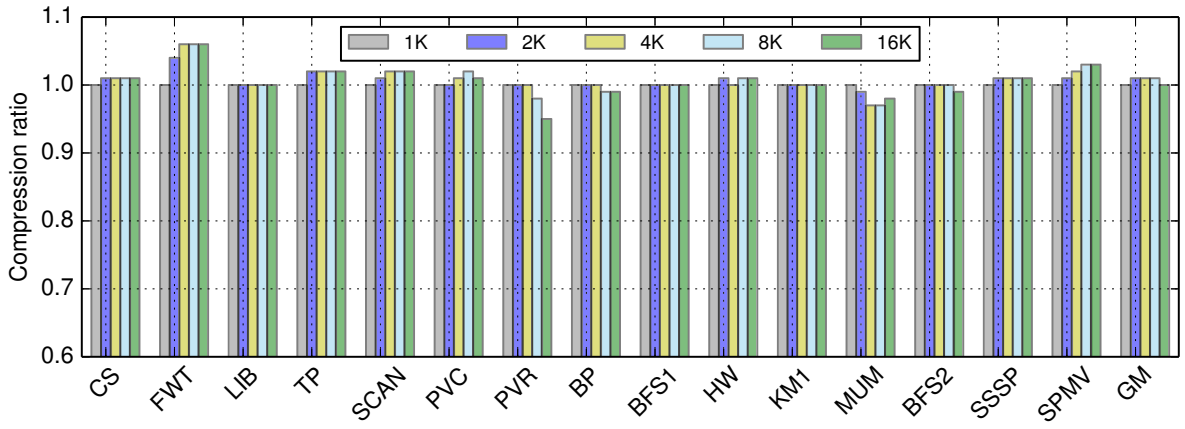
final phase, compression takes place. Accurate probability estimation is one of the key components of entropy based compression. Therefore, in our proposed E^{2}MC technique we use both *offline probability estimation* where we estimate the probability of symbols offline, and *online probability estimation* where we sample the probability of symbols during runtime.

Offline Probability Estimation We simulate all benchmarks and store their load and store data in a database. Then we profile the database offline to find the probability of symbols. Offline probability estimate is the best estimate we can have. In Section 6.4 it is shown that offline probability yields the highest compression ratio. However, offline probability can only be used if approximate entropy characteristics of a data stream are known in advance.

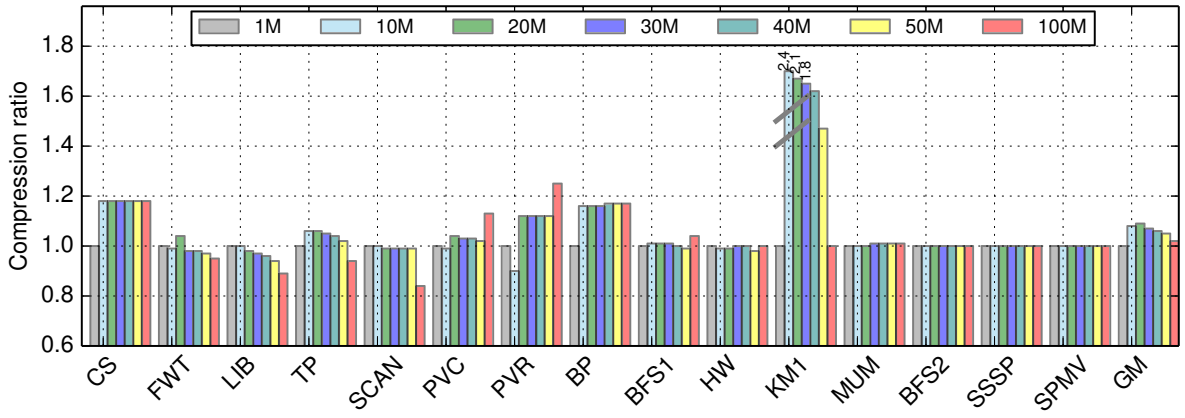
Online Probability Estimation One of the drawbacks of an entropy based compression techniques is that they may require online sampling to estimate the frequency of symbols if entropy characteristics are not known in advance. The sampling phase is an additional overhead as during sampling no compression is performed. Fortunately, our experiments show that it is possible to achieve compression ratio comparable to offline probability with a very short online sampling phase at the start of the benchmarks.

During the sampling phase every memory request is monitored at the memory controller to record unique values and their count. We use a value frequency table (VFT) similar to [10] to store unique values and their count. For 4 and 8-bit SLs, we store all values as there are only 16 and 256 possible values. However, for SLs of 16 and 32-bit we only store the most frequent values (MFVs) and use them to generate codewords as the total number of values is very large and storing all of them is not practical.

There are two key decisions to make regarding online sampling. First, what is the suitable number of MFVs? More MFVs may help to encode better for some benchmarks, however, more MFVs increase the cost of the hardware. Figure 6.5a shows the compression



(a) Compression ratio with different number of MFVs.



(b) Compression ratio with different sampling durations.

Figure 6.5: Online sampling decisions [85]. © 2017 IEEE

ratio for different numbers of MFVs relative to 1K MFVs. It shows that the compression ratio does not increase much with the number of MFVs, only 1% on average. Only for some benchmarks (FWT, TP, SCAN, SPMV), more MFVs give slightly higher compression ratio, FWT being the highest gainer (6%). Hence, we choose 1K MFVs to construct encoding. In some cases (PVR, MUM), the compression ratio can even decrease with the increase in MFVs as the length of the prefix which is attached to each uncompressed symbol can increase. For example, for the MUM benchmark, the prefix is 3-bit long for 2K MFVs and 4-bit long for 4K MFVs. Thus, the compression ratio is lower for 4K MFVs compared to 2K MFVs. Please refer to Section 6.2.4 for more details.

The second decision is: what is the best sampling duration? Figure 6.5b shows the compression ratio for sampling durations of 1M, 10M, 20M, 30M, 40M, 50M and 100M instructions relative to sampling duration of 1M instructions. The unit of sampling is millions of instructions. It shows that sampling for 20M instructions yields the highest compression ratio. Hence, we choose 20M instructions for online sampling. The longer

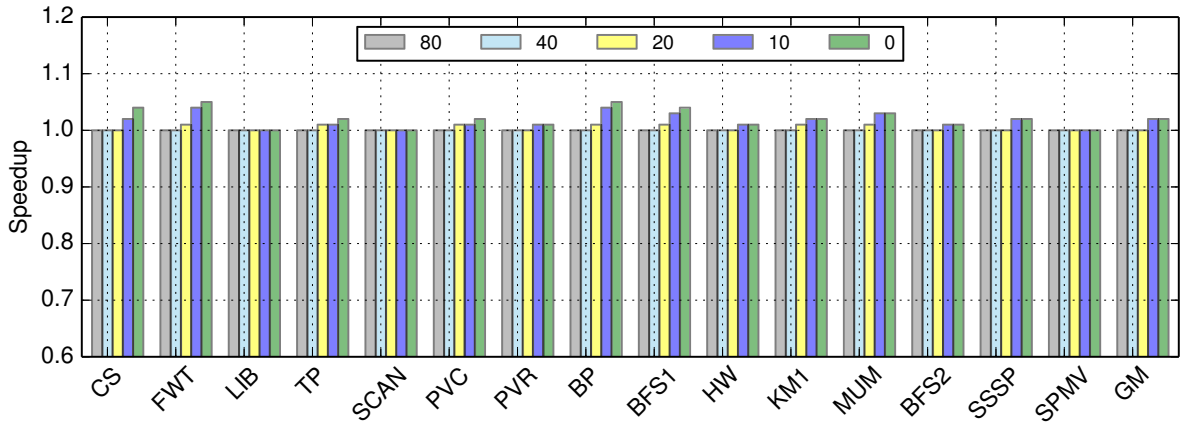
sampling duration can improve probability estimation, however, there is a trade-off between compression ratio and improved probability estimation as no compression is done during sampling duration. Thus, we notice low compression ratio for sampling durations longer than 20M instructions.

Codeword Generation After the sampling phase, code generation takes place. Instead of classic Huffman coding, canonical Huffman coding [148] is used because it is fast to decode as the codeword length is enough to decode a codeword. In canonical Huffman coding, codewords of the same length are consecutive binary numbers. To generate canonical Huffman codewords (CHCs), first classic codewords are generated by building a Huffman tree using a minimum heap data structure as described in [139] and then symbols are sorted according to their code lengths. The first symbol is assigned a codeword of all zeros of length equal to the original SL. The next codeword is just the consecutive binary number if the SL is the same. When a longer codeword is encountered, the last canonical codeword is incremented by one and left shifted until its length is equal to the original SL. The procedure is repeated for all symbols. For example, assume we have three symbols (A = $(11)_2$, B = $(0)_2$, C = $(101)_2$) with classic Huffman codewords. To convert them to canonical, we first sort symbols according to code length (B = $(0)_2$, A = $(11)_2$, C = $(101)_2$) and then the first symbol (B) is assigned code $(0)_2$. The CHC for the next symbol A is $(10)_2$ which is obtained by incrementing the last codeword by one and then left shifting to match the original code length of A. Similarly, CHC for C is $(110)_2$.

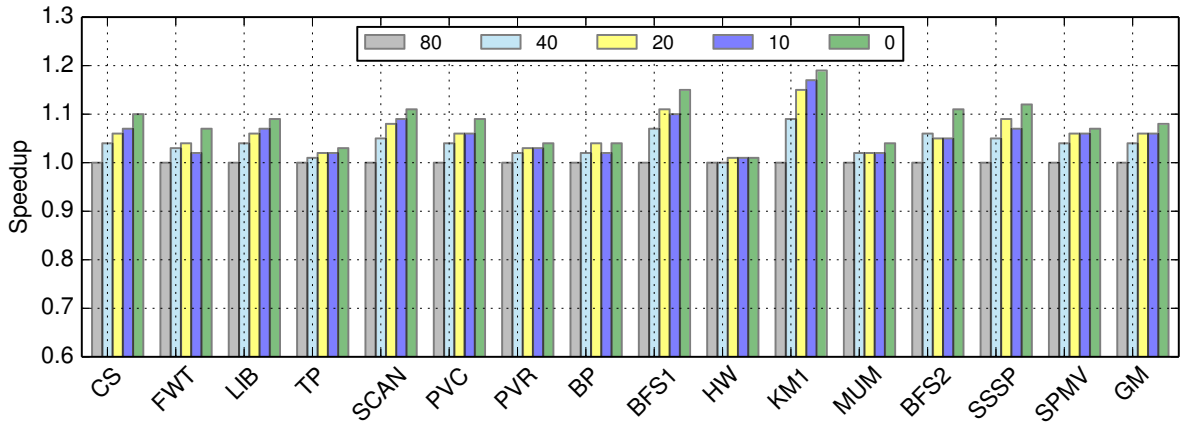
To ensure that unnecessarily long codewords are not generated, we assign a minimum probability to each symbol such that no codeword is longer than the predetermined maximum length. Our experiments show that there are only a few symbols which have codewords longer than 20-bit for SL 16 and 32-bit. Hence, we adjust the frequency so that no symbol is assigned a code longer than 20-bit. Similarly, for SL 4 and 8-bit we fix the maximum codeword length to 8 and 16-bit, respectively. Finally, all symbols that are not in MFVs are assigned a single codeword based on their combined frequency and this codeword is attached as a prefix to store all such symbols uncompressed. Since we only need probability estimation and code generation in the beginning, both of these steps can be done in software.

6.2.5 Low Decompression Latency

As already explained, GPUs are high throughput devices and less sensitive to latency than CPUs. However, large latency increases can also affect GPU performance. Figure 6.6a and 6.6b shows the speedup when compression and decompression latency is decreased from 80 cycles to 0 cycles, respectively. It can be seen that there is a small speedup (geometric mean of 1%) when the compression latency is decreased to 0 cycles. However, there is a significant speedup (geometric mean of 9%) when the decompression latency is decreased to 0 cycles. The speedup is more sensitive to decompression latency because warps have to stall for loads from memory, while stores can be done without stalling.



(a) Effect of compression latency on speedup.



(b) Effect of decompression latency on speedup.

Figure 6.6: Effect of latency [85]. © 2017 IEEE

The results clearly show the importance of low decompression latency. Thus, we perform parallel decoding to decrease the decompression latency as explained in Section 6.2.10.

6.2.6 Memory Access Granularity and Compression

GPUs employ GDDR (Graphics Double Data Rate) as main memory because GDDR provides higher bandwidth due to large memory access granularity (MAG). MAG is the amount of data read from or written to a memory by a single read or write command. MAG is a product of the burst length and bus width. The burst length is decided by DRAM technology and it directly determines the MAG size. Table 6.1 shows the burst length for different generations of GDDR. The burst length has increased over the generations to support high data transfer rates.

MAG is an important factor for memory compression as it affects the minimum amount of data that can be fetched from memory. Since it is only possible to fetch in the multiple

Table 6.1: Burst length across generations of GDDR [85]. © 2017 IEEE

GDDR Generation	Burst Length	GDDR Generation	Burst Length
GDDR1	2	GDDR5X	8/16
GDDR2	4	GDDR6	16
GDDR3/4/5	8	-	-

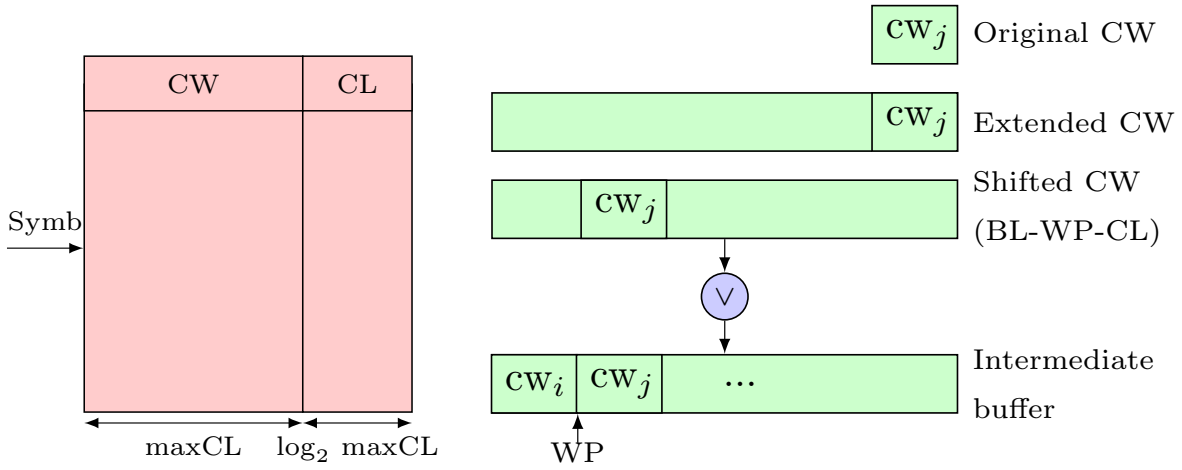
of a MAG, for a compression ratio (CR) that is not an exact multiple of a MAG, more data is fetched from memory than what is actually needed. Assuming a MAG of 32B which is the case for GDDR5/5X/6, for a block that is compressed from 128B to 65B (raw CR of 1.97), the actual amount of fetched data is 96B (3× 32B). Thus, while the raw CR looks very close to 2, the effective CR is only 1.33. Therefore, due to low effective compression ratio, the performance gain could be significantly less than otherwise possible from high raw compression ratio. We assume MAG is 32B for this study.

6.2.7 Compression Overhead: Metadata Cache

To save memory bandwidth as a result of compression, we need to only fetch the compressed 32B bursts from a DRAM. Therefore, the memory controller needs to know how many bursts to fetch for every memory block. For GDDR5, the number of bursts varies from 1 to 4. Similar to previous work [146, 160], we store 2- bit for every memory block as metadata. For a 32-bit, 4GB DRAM with block size of 128B, we need 8MB of DRAM for storing the metadata. However, we cannot afford to access the metadata first from DRAM and then issue a memory request for the required number of bursts. This requires two accesses to DRAM and defeats the purpose of compression to save memory bandwidth. Therefore, like previous work [146, 160], we cache the most recently used metadata in a cache. We use a small 8KB 4-way set associate cache to store the metadata. The 2-bit stored in the metadata are also used to determine if the block is stored (un)compressed. The value $(11)_2$ means the block is stored uncompressed.

6.2.8 Huffman Compressor

Figure 6.7 shows an overview of the Huffman compressor. It can be implemented as a small lookup table (c-LUT) that stores codewords (CWs) and their code lengths (CLs) as shown in Figure 6.7a. The maximum number of CWs is 2^N, where N is the SL. As the maximum number of CWs is only 16 and 256 for SLs 4 and 8-bit, we store all CWs in a c-LUT and index it directly using symbol bits. However, for SLs 16 and 32-bit such a lookup table is not practical and instead we store 1K MFVs as discussed in Section 6.2.4. We use an 8-way set associative cache to implement c-LUT for 1K MFVs. The cache is indexed by lower 7-bit of a symbol.



(a) c-LUT. (b) Placing codewords together.

Figure 6.7: Huffman compressor [85]. © 2017 IEEE

For SLs 4 and 8-bit, we build different Huffman trees corresponding to different symbols in a word and thus, also use multiple c-LUTs, one corresponding to each tree. For example, for SL 8-bit, we use 4 c-LUTs for the four different symbols in a word. We assume a word size of 32-bit for our study.

Once we obtain a CW from c-LUT, we need to place it together with other CWs. Figure 6.7b shows how the CWs are placed together. We use an intermediate buffer of 2× maxCL as buffer length (BL), where maxCL is the maximum code length. To place a CW at its right position, first the CW is extended to match BL and then the extended CW is left shifted by $BL - WP - CL$ using a barrel shifter, where WP is the current write position in the buffer. Finally, the shifted CW is bitwise ORed with the intermediate buffer and WP is incremented by CL. When WP $\geq$ maxCL, compressed data, equal to maxCL, is moved from the intermediate buffer to the final buffer. The intermediate buffer is left shifted by maxCL and WP is decremented by maxCL. Our RTL synthesis shows that placing the CWs together takes more time than getting CW and CL from c-LUT. The sum of the lengths of all CWs of a block determines if a block is stored (un)compressed. When the sum is $\leq$ 96B, a block is stored compressed, otherwise uncompressed. Please refer to Section 6.2.6 to understand the reason to choose compressed size $\leq$ 96B to decide if a block is stored (un)compressed.

6.2.9 Huffman Decompressor

Figure 6.8 shows an overview of the Huffman decompressor. Our design is based on the Huffman decompressor as proposed in [10], which mainly consists of a barrel shifter, comparators, and a priority encoder to find the CW and CL. We use a buffer of length 2× maxCL to store part of the compressed block. We use buffer length of 2× maxCL instead of maxCL as in [10] for two reasons. First, we can continue decoding without shifting

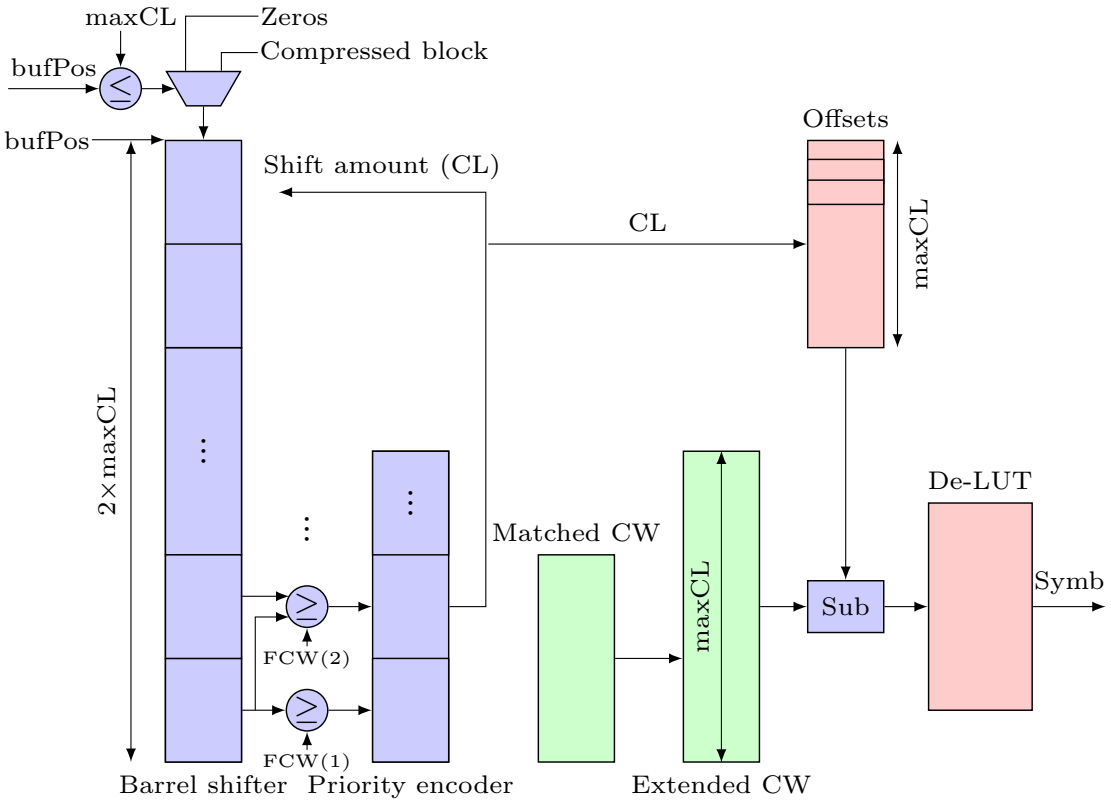


Figure 6.8: Huffman decompressor [10, 85]. © 2017 IEEE

data every cycle from the compressed block to the buffer. Second, it helps to define a fixed-width interface (maxCL in this case) to input compressed data instead of every cycle shifting a different number of bits, equal to the matched CL. A pointer (bufPos) which initially points to the end of the buffer is used to track the filled size of the buffer. To find a CW, comparison is done in parallel between all potential CWs and the First Codewords (FCWs) of all lengths. The FCWs of all lengths are stored in a table. A priority encoder selects the first CW which is $\geq$ FCW of length l and $\leq$ FCW of length $l+1$. The selected CW is extended by padding zeros to match the maxCL. An offset which depends on the CL and is calculated during code generation is subtracted from CW to obtain the index for the De-LUT. The barrel shifter shifts the buffer by CL and the bufPos is decremented by CL. When bufPos $\leq$ maxCL, the remaining compressed data of length maxCL is shifted into the buffer from the compressed block and bufPos is incremented by maxCL.

Although symbols are stored in consecutive locations in the De-LUT, canonical CWs of different CLs are not consecutive binary numbers. Therefore, the De-LUT cannot be indexed directly using the CW to obtain the decoded symbol. We need to subtract an offset from the CW to find the index. These offsets are calculated during code generation. For example, assume we have three symbols with canonical Huffman codewords (CHCs) (B = $(0)_2$, A = $(10)_2$, C = $(110)_2$). The symbol B will be stored at index 0 in the De-LUT,

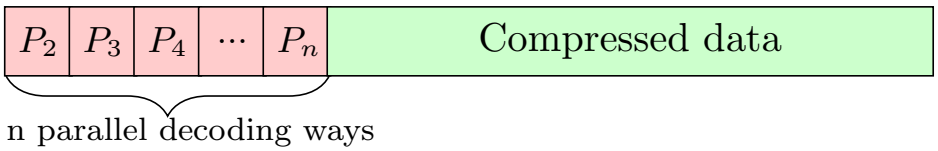


Figure 6.9: Structure of a compressed block [85]. © 2017 IEEE

so it has offset 0. However, A will be stored at index 1, so it has offset 1 ($(10)_2$-$(01)_2$). Similarly, C will be stored at index 2 and it has offset 4 ($(110)_2$-$(010)_2$). The maximum number of offsets is equal to maximum number of CWs of different lengths. Both offset and FCW tables are read and writable so that they can be changed for different benchmarks. We use multiple decompressor units for different symbols in a word for SLs 4 and 8-bit.

6.2.10 Parallel Decoding and Memory Access Granularity

As the compressed data needs to be decompressed before it can be used and decompression incurs high latency which can degrade the performance. Thus, we need to decode in parallel to reduce the decompression latency. Unfortunately, Huffman decoding is serial as the start of the next CW is only known once the previous CW has been found. Serial decoding requires high latency which can limit the performance gain. One way to parallelize Huffman decoding is to explicitly store pointers to CWs where we want to start decoding in parallel in the compressed block itself. The number of pointers depends on the number of required parallel decoding ways (PDWs).

Figure 6.9 shows the structure of a compressed block. It consists of $n-1$ pointers (P_2, P_3, ..., P_n) for n PDWs, and the compressed data. Each pointer consists of N bits where 2^N is the block size in bytes. For example, for a 128B block, 7-bit are needed for each PDW. The starting codewords for parallel decoding are byte-aligned while compressing them. These pointers are overhead and hence will reduce compression ratio. However, the effective loss in compression ratio is usually much lower due to the aforementioned memory access granularity (MAG). Most blocks are compressed to a size that allows adding extra bits for parallel decoding without reducing their compression ratio at the MAG. Our experiments show that even with 4 PDWs where we store 21 extra bits (3∗7) in the compressed block, there is either no or small loss in compression ratio. We analyze the loss in compression and the number of PDWs needed in Section 6.4.2 and 6.4.4.

6.2.11 GPU High Throughput Requirements

To gain from memory compression, the throughput (bytes (de)compressed per second) of the compressor and decompressor should match the full compressed memory bandwidth (BW). Suppose we can obtain a maximum compression of 4×, then the compressor and decompressor throughput has to be 4 times the GPU BW to fully utilize the compressed BW. Unfortunately, a single (de)compressor unit cannot meet such high throughput.

Table 6.2: Frequency, bandwidth, area, and power of a single unit of compressor and decompressor [85]. © 2017 IEEE

	Compressor				Decompresor			
SL (Bits)	Freq (GHz)	BW (GB/s)	Area (mm^2)	Power (mW)	Freq (GHz)	BW (GB/s)	Area (mm^2)	Power (mW)
4	1.67	0.84	0.02	1.43	1.11	0.56	0.01	5.12
8	1.54	1.54	0.08	4.01	0.91	0.91	0.04	11.91
16	1.43	2.86	0.11	5.47	0.80	1.60	0.07	11.89
32	1.43	5.72	0.17	9.34	0.80	3.20	0.12	14.30

Table 6.3: #units, area, and power to support 4× 192.4 GB/s [85]. © 2017 IEEE

	Compressor			Decompresor			GTX580	
SL (Bits)	Units (#)	Area (mm^2)	Power (W)	Units (#)	Area (mm^2)	Power (W)	Area (%)	Power (%)
4	464	7.9	0.7	692	10.3	3.6	3.4	1.7
8	252	20.3	1.0	424	18.1	5.0	7.3	2.5
16	136	14.6	0.7	240	16.4	2.8	5.8	1.5
32	68	11.5	0.6	120	14.3	1.7	4.9	0.9

Table 6.2 shows the frequency, BW, area, and power of a single compressor and decompressor unit as reported by Synopsis design compiler for different SLs. The BW of NVIDIA GTX580 is 192.4 GB/s (32.1 GB/s per memory controller). However, the combined throughput of the compressor and decompresor for any SL is far less than 4× 32.1 GB/s. For example, the combined throughput of the SL 16-bit is only 4.46 GB/s. Clearly, a single (de)compressor unit is not enough and we need multiple units.

Table 6.3 shows the total number of compressor and decompressor units needed to support 4× 192.4 GB/s and the corresponding area and power. The n parallel decoding ways (n PDWs) utilize n decompressors from these total numbers of units to decode a single block in parallel. This only decreases decompression latency and does not add up to the total number of required units. Thus, no further multiplication of the numbers shown in Table 6.3 is required for n PDWs.

Table 6.3 also shows the total area and power needed as percentage of the area and peak power of the GTX580. A single (de)compressor unit requires less area and power for smaller SLs. However, the total area and power needed to support the GTX580 BW is much higher for smaller SLs as more units are required to meet the BW. We have

Table 6.4: Baseline simulator configuration [85]. © 2017 IEEE

Parameter	Value	Parameter	Value
#SMs	16	L1 $ size/SM	16KB
SM freq (MHz)	822	L2 $ size	768KB
Max #Threads per SM	1536	# Memory controllers	6
Max #CTA per SM	8	Memory type	GDDR5
Max CTA size	512	Memory clock	1002 MHz
#FUs per SM	32	Memory bandwidth	192.4 GB/s
#Registers/SM	32K	Burst length	8
Shared memory/SM	48KB	Bus width	32-bit

smaller area and power for SL 4-bit because the c-LUT and De-LUT have very small number of entries (16). In general, the area numbers are likely higher than expected because the memory design library does not have exact memory designs needed to design (de)compressor and we have to combine smaller designs to get the required size. We believe that a custom design will be denser and will need less area. We find that none of the related work discussed throughput requirements of GPUs.

6.3 Experimental Setup

6.3.1 Simulator

We use gpgpu-sim v3.2.1 [15] and modify it to integrate BDI, FPC and E^{2}MC. We configure gpgpu-sim to simulate a GPU similar to NVIDIA's GF110 on the GTX580 card. The baseline simulator configuration is summarized in Table 6.4. For more information regarding the simulator, please refer to [15].

Table 6.5 shows the (de)compressor latencies used to evaluate BDI, FPC, and E^{2}MC. For E^{2}MC, we evaluate four designs of SLs 4, 8, 16, and 32-bit denoted by E^{2}MC4, E^{2}MC8, E^{2}MC16, and E^{2}MC32, respectively. The latencies of BDI and FPC are obtained from published papers [130] and [6]. For E^{2}MC, we write RTL for the compressor and decompressor designs and then synthesize using Synopsis design compiler version K-2015.06-SP4 at 32nm process node to accurately estimate the frequency, area, and power. The compressor is pipelined using two stages. The first stage fetches the CW and CL from the c-LUT, while the second stage combines CWs together. We find that the critical path delay is in the second stage of the compressor. The decompressor is pipelined using three stages. The first stage finds a CW, the second stage calculates the index for the De-LUT using CW and offsets, and the De-LUT is accessed in the third stage to get the decoded symbol. We find that the critical path delay is in the first stage of the decompressor. In

Table 6.5: Compressor and decompressor latency in cycles [85]. © 2017 IEEE

	BDI	FPC	E^{2}MC4	E^{2}MC8	E^{2}MC16	E^{2}MC32
Compressor latency	1	6	154	84	46	24
Decompressor latency	1	10	233	143	82	42

E^{2}MC, one symbol can be (de)compressed in a single cycle of frequency listed in Table 6.2. The frequency is calculated using critical path delay. However, the DRAM frequency is 1002 MHz for GTX580. We scale the (de)compressor frequency and then count the number of cycles needed to (de)compress a memory block of size 128B. Furthermore, for each PDW, we assume the decompressor latency is decreased by the same factor.

For estimating the energy consumption of the different benchmarks, GPUSimPow [99] is modified to integrate the power model of the compressor and decompressor. The power numbers obtained by RTL synthesis are used to derive power models for the compressor and decompressor. Unfortunately, we do not have power models for BDI and FPC and therefore, we cannot estimate their energy consumption. Thus, we only provide energy estimate for E^{2}MC.

6.3.2 Benchmarks

Table 6.6 shows the benchmarks used for evaluation. We include benchmarks from the popular CUDA SDK [123], Rodinia [25], Mars [46], Lonestar [81], SHOC [37], gpgpu-sim [15]. The lower part of Table 6.6 shows 7 compute-bound benchmarks that we use for sensitivity analysis in Section 6.4.5. The benchmarks either belong to single-precision/double-precision floating point (FP), integer (INT), unsigned character (U8), or mixed (Mixed) category depending upon their data types. We modified the inputs of SCAN and FWT benchmarks as the original inputs were random which are not suitable for any compression technique. We use SCAN for stream compaction which is an important application and FWT to transform Walsh functions.

6.4 Experimental Results

To evaluate the effectiveness of E^{2}MC, we compare compression ratio (CR) and performance of E^{2}MC for SLs 4, 8, 16, and 32-bit with BDI and FPC. We provide two kinds of compression ratios, raw CR and CR at memory access granularity (MAG). The raw CR is the ratio of the total uncompressed size to total compressed size. For CR at MAG, the total compressed size is calculated by scaling up the compressed size of each block to the nearest multiple of MAG and then adding all the scaled block sizes.

First, we present CR results using offline and online probability estimation and discuss CR and parallel decoding trade-off. We then compare the speedup of E^{2}MC with BDI and

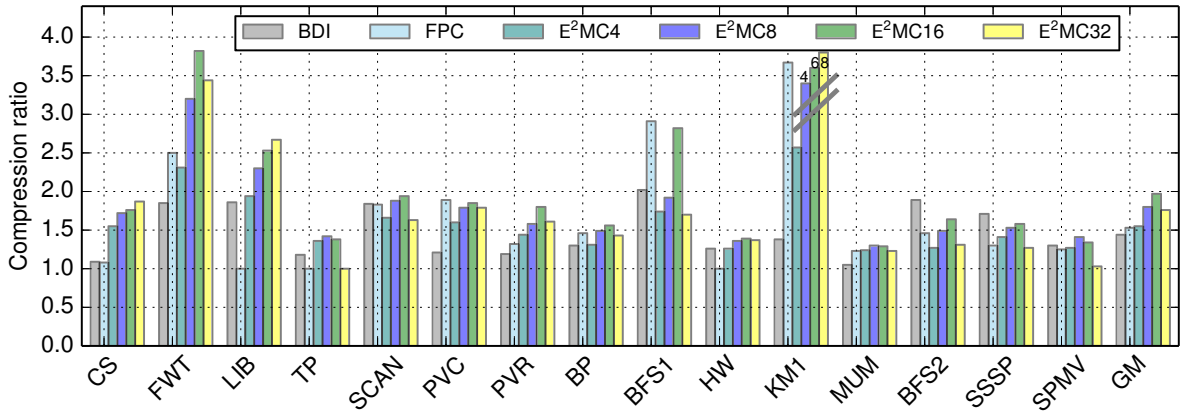
Table 6.6: Benchmarks used for experimental evaluation [85]. © 2017 IEEE for the upper half of the table.

Name	Abbreviation	Data Type	Origin
convolSeparable	CS	Single-precision FP	CUDA SDK
fastWalshTrans	FWT	Single-precision FP	CUDA SDK
libor	LIB	Single-precision FP	CUDA SDK
transpose	TP	Single-precision FP	CUDA SDK
scan	SCAN	INT	CUDA SDK
PageViewCount	PVC	INT	MARS
PageViewRank	PVR	INT	MARS
backprop	BP	Single-precision FP	Rodinia
bfs	BFS1	INT	Rodinia
heartwall	HW	Mixed	Rodinia
kmeans	KM1	Mixed	Rodinia
mummergpu	MUM	INT	Rodinia
bfs	BFS2	INT	Lonestar
sssp	SSSP	INT	Lonestar
spmv	SPMV	Mixed	SHOC
storegpu	STO	INT	[5]
StringMatch	SM	Mixed	MARS
WordCount	WC	Mixed	MARS
lavaMD	MD	Mixed	Rodinia
pathfinder	PF	INT	Rodinia
nn	NN	Single-precision FP	gpgpu-sim
ray	RAY	Mixed	gpgpu-sim

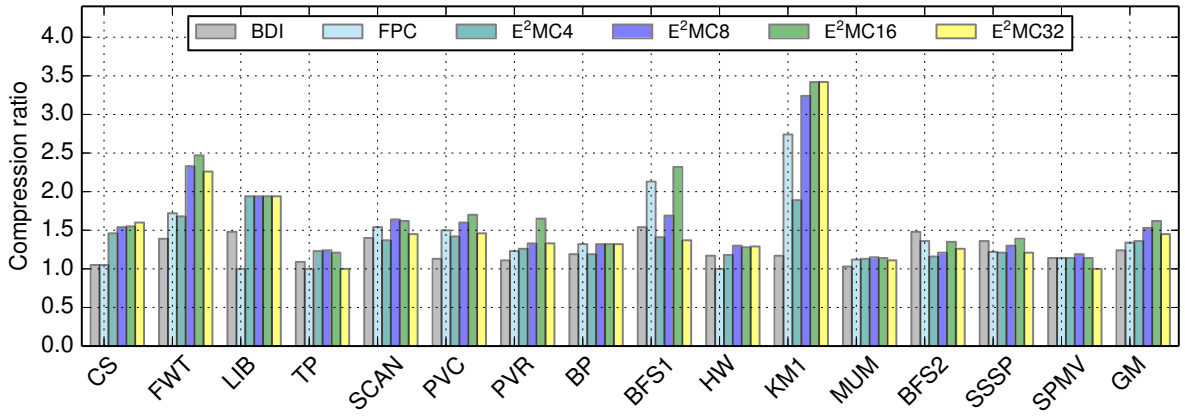
FPC and show the importance of decoding in parallel. Finally, we present the sensitivity analysis of compute-bound benchmarks to E^{2}MC and study the energy efficiency of E^{2}MC.

6.4.1 Compression Ratio using Offline Probability

Figure 6.10a depicts the raw CR of BDI, FPC, and E^{2}MC with offline probability. It shows that on average E^{2}MC provides higher CR than BDI and FPC for all SLs. The geometric mean of the CR of E^{2}MC for SLs 4, 8, 16 and 32-bit is 1.55×, 1.80×, 1.97×,



(a) Raw compression ratio.



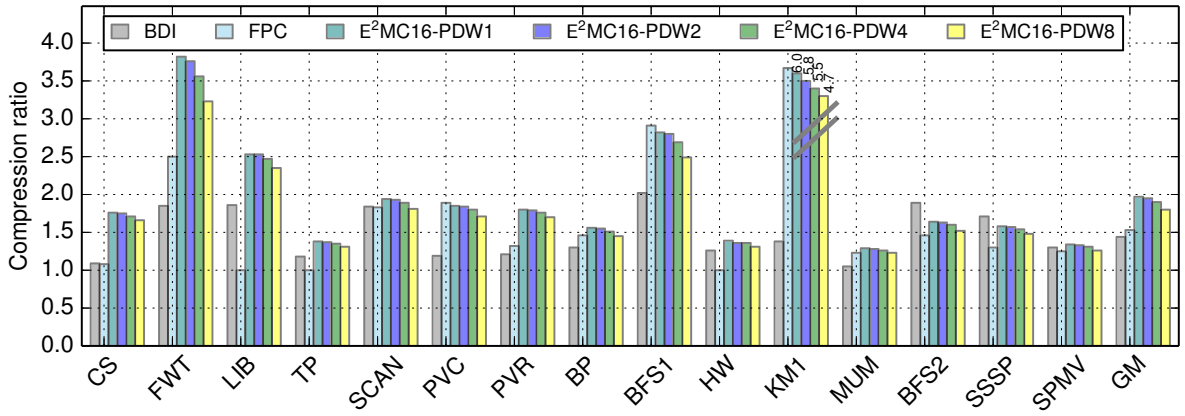
(b) Compression ratio at MAG.

Figure 6.10: Compression ratio of BDI, FPC and E²MC [85]. © 2017 IEEE

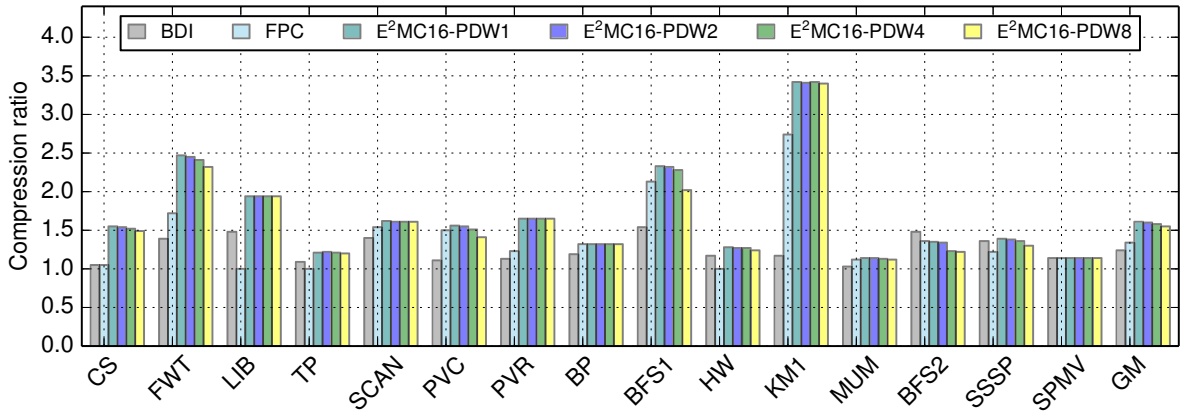
and 1.76×, respectively, while that of BDI is only 1.44× and FPC is 1.53×. This shows that entropy based compression techniques provide higher CR than simple compression techniques such as BDI and FPC whose CR is limited. E²MC16 yields the highest CR which is 53% and 42% higher than the CR of BDI and FPC respectively.

As discussed in Section 6.2.6, data from memory is fetched in the multiple of MAG. Figure 6.10b shows the CR of BDI, FPC and E²MC when this is taken into account. We see that the CR of all three techniques at MAG is less than the raw CR. However, the CR of E²MC is still higher compared to BDI and FPC. The geometric mean of the CR of E²MC for SLs 4, 8, 16 and 32-bit is 1.36×, 1.53×, 1.62×, and 1.45×, respectively, while the CR of BDI is only 1.24× and the CR of FPC is 1.34×. We see that E²MC16 also yields the highest CR at MAG. So, we select E²MC16 for further analysis.

To obtain an estimate how close E²MC16 is to the optimal CR, we calculate upper bound on CR using Shannon's source coding theorem [150]. The average optimal CR for SL 16-bit is 2.61×, which means there is a gap of 64% that could be further exploited.



(a) Raw compression ratio.



(b) Compression ratio at MAG.

Figure 6.11: Compression ratio of E²MC16 with parallel decoding [85]. © 2017 IEEE

However, we think further improving the compression ratio of E²MC16 to narrow the gap with optimal CR is difficult as compression is data dependent.

6.4.2 Compression Ratio and Parallel Decoding Trade-off

As shown in Section 6.2.5, low decompression latency is important for achieving high performance. To reduce the decompression latency we decode in parallel as discussed in Section 6.2.10. However, parallel decoding is not for free as it decreases the CR. Hence, the number of parallel decoding ways (PDWs) needs to be selected in a way such that the performance gain is maximal and the loss in CR is minimal.

Figure 6.11a shows that the CR decreases slightly when the number of PDWs increases. However, we will see in Section 6.4.4, the performance increases with the number of PDWs as the decompression latency decreases. Moreover, we see from Figure 6.11a that the CR of E²MC16 even for 8 PDWs is still much higher than the CR of BDI and FPC.

We have seen that parallel decoding causes loss in CR. However, the loss in CR at MAG is much lower than the loss in raw CR as shown in Figure 6.11b than Figure 6.11a. For example, there is 9% loss in raw CR with 4 PDWs, while at MAG it is only 4%. The reason for lower CR loss at MAG is that at MAG we usually need to fetch some extra bits to meet the MAG requirements. Using these extra bits to store offsets for parallel decoding does not cause loss in CR. This is not always true, therefore, we see some loss in CR even at MAG.

6.4.3 Compression Ratio using Online Sampling

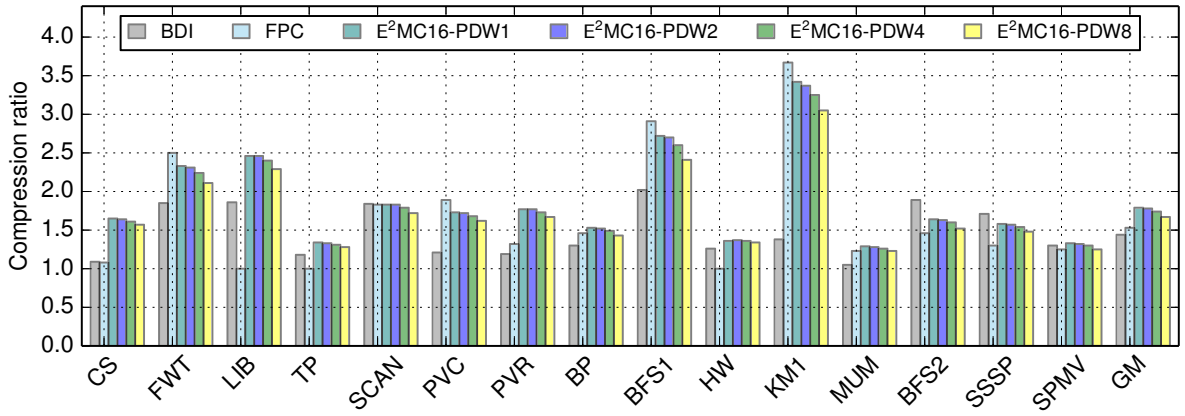
As discussed in Section 6.2.4 online sampling is needed to estimate probability if entropy characteristics are not known in advance. Figure 6.12a shows the CR for online sampling size of 20M instructions. We choose 20M instructions for online sampling as it gives the highest CR as shown in Figure 6.5b and we only sample at the beginning of each benchmark. The CR of E^{2}MC16 is 1.79×, which is 35% and 26% higher than the CR of BDI and FPC, respectively. However, as expected the CR with online sampling is lower by 18% on average than that of offline sampling.

Figure 6.12b shows the CR with online sampling at MAG. The CR of E^{2}MC16 at MAG is 1.52×, which is still 28% and 18% higher than BDI and FPC, respectively. The CR results show that it is possible to achieve reasonably higher CR with small online sampling. However, the CR at MAG with online sampling is 10% lower than offline sampling.

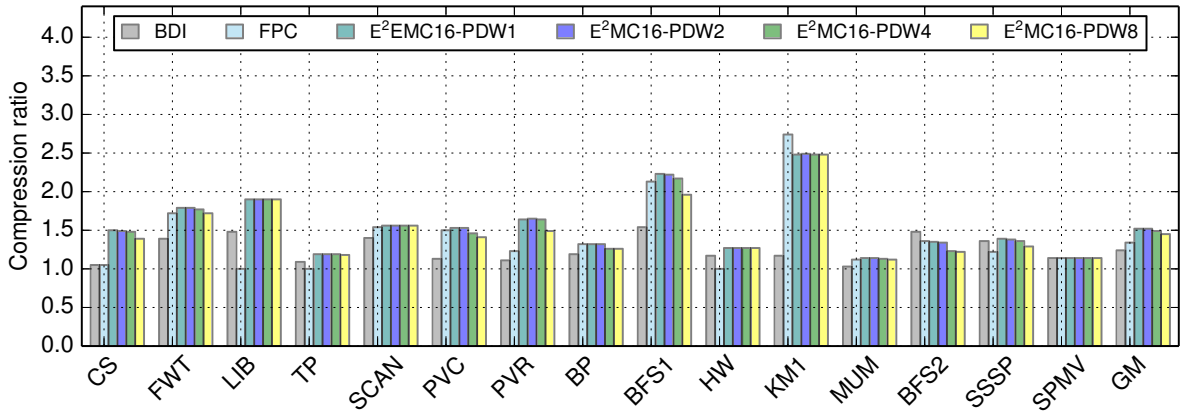
6.4.4 Speedup

We first establish an upper bound on the speedup assuming favorable conditions and then use realistic conditions to study the actual gain. Figure 6.13a shows the speedup of BDI, FPC, and E^{2}MC16 when offline probability is used and the (de)compression latency is only one cycle for all techniques. BDI and FPC achieve an average speedup of 12% and 16%, respectively, while the average speedup of E^{2}MC is 16%, 21%, 23%, and 16% for SLs 4, 8, 16, and 32-bit, respectively. The speedup is due to the decrease in DRAM bandwidth requirement which is reciprocal of the achieved compression ratio.

Figure 6.13b and Figure 6.13c shows the speedup of BDI, FPC and E^{2}MC16 for offline and online sampling with realistic latencies as shown in Table 6.5. For E^{2}MC we only show speedup in detail for SL 16-bit with 1 to 8 PDWs. A brief discussion of the speedup for SL 4, 8, and 32-bit is presented later in the section. We see that the speedup is less for FPC and E^{2}MC16 using realistic latencies. However, the speedup of BDI does not change as actual latency is also single cycle. The speedup of E^{2}MC16 with offline probability (13%) is equal to the speedup of FPC (13%) and even slightly less with online probability (11%) when no parallel decoding is used, even though the CR of E^{2}MC16 is much higher than that of FPC. This is because without parallel decoding the decompression latency of E^{2}MC16 is 82 cycles, which is much higher than the decompression latency of FPC which is 10 cycles. The speedup increases when we increase the PDWs from 1 to 4 because each



(a) Raw compression ratio.

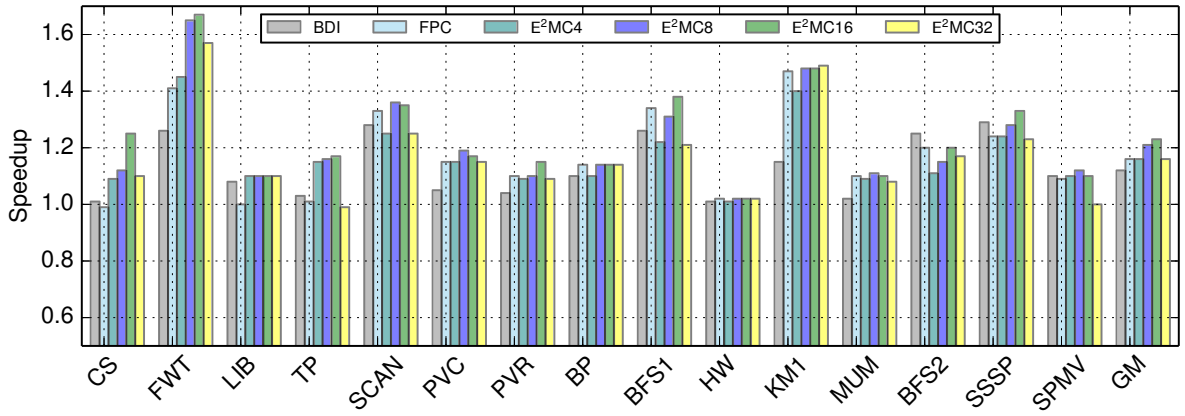


(b) Compression ratio at MAG.

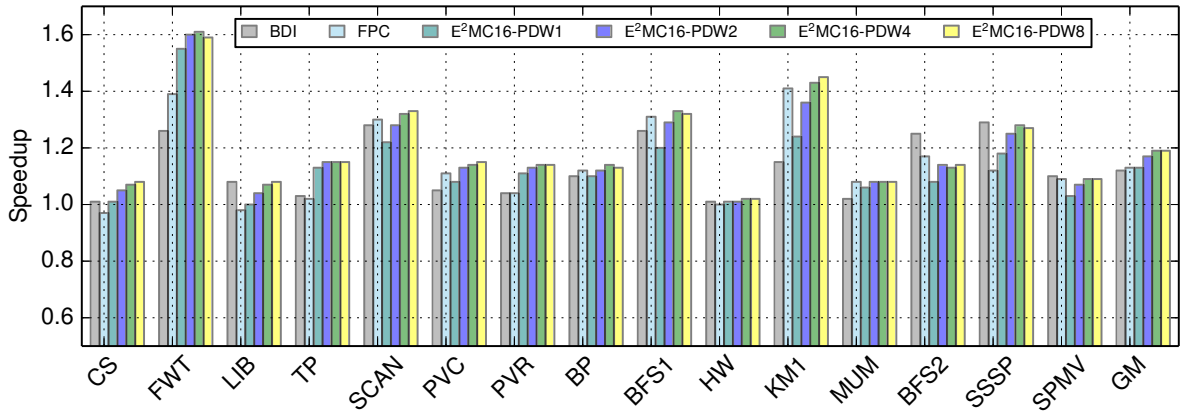
Figure 6.12: Compression ratio of BDI, FPC and E²MC16 for online sampling size of 20M instructions [85]. © 2017 IEEE

PDW decreases the decompression latency by half. However, each PDW also decreases the CR as we need to store the offsets for parallel decoding and hence there is a trade-off between the CR and performance gain. Figure 6.13b shows that there is no further gain in performance from 4 PDWs to 8 PDWs. This is because the increase in performance due to further decrease in latency is nullified by the decrease in CR. Hence, to achieve higher speedup for E²MC16 we not only need higher CR but also the decompression latency has to be reasonably low.

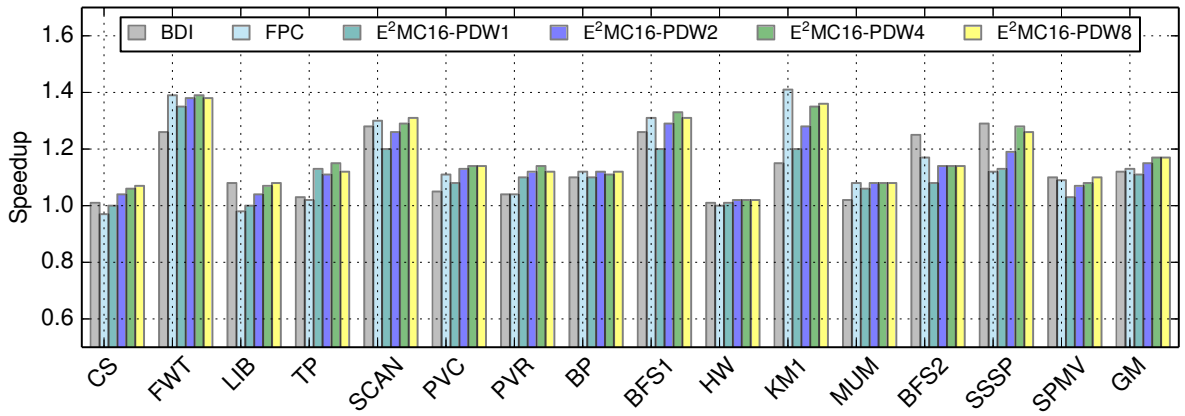
Figure 6.13a shows that the average speedup of E²MC for SL 8-bit with single cycle latency is much higher than the speedup of BDI and FPC and close to the speedup for SL 16-bit. The speedup for SLs 4 and 32-bit is also higher or equal to BDI and FPC. However, when actual latency is used the average speedup is much lower. The average speedup of E²MC for SLs 4, 8-bit with 8 PDWs and for SL 32-bit with 4 PDWs is 2%, 14% and 15% respectively. The reason for low speedup for SLs 4, and 8-bit is their high



(a) Offline sampling with single cycle latency.



(b) Offline sampling with realistic latency.



(c) Online sampling with realistic latency.

Figure 6.13: Speedup of BDI, FPC and E^{2}MC [85]. © 2017 IEEE

decompression latency.

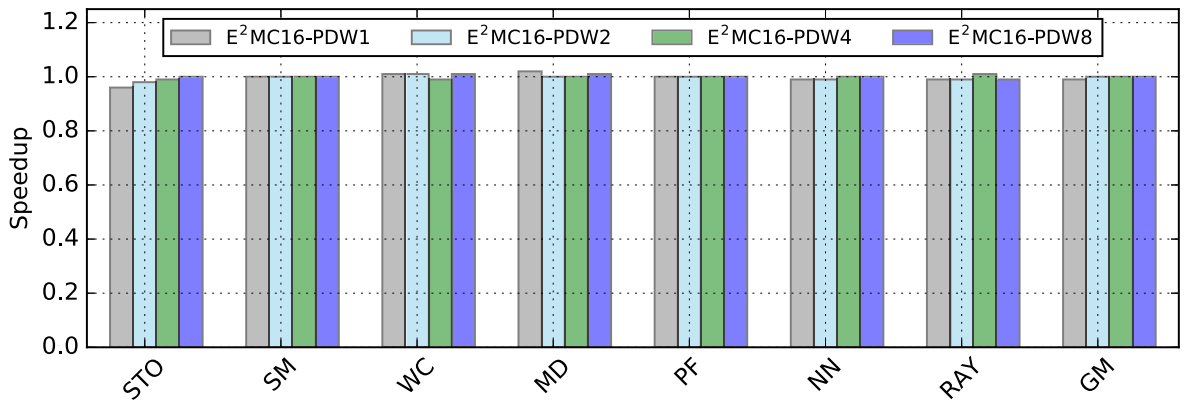


Figure 6.14: Sensitivity analysis of compute-bound benchmarks.

We see that both offline and online sampling results in higher performance gain for E^{2}MC16 than BDI and FPC, provided decompression latency is reduced by parallel decoding. The geometric mean of the speedup of E^{2}MC16 with 4 PDWs is about 20% with offline probability and 17% with online probability. The average speedup is 8% higher than the state of the art with offline and 5% higher with online sampling.

6.4.5 Sensitivity Analysis to Compute-bound Benchmarks

We conduct sensitivity analysis to verify that E^{2}MC increases performance of the memory-bound benchmarks without hurting the performance of the compute-bound benchmarks. The compute-bound benchmarks have low bandwidth utilization and they either do not gain or gain very small from the increase in bandwidth. We select such compute-bound benchmarks and compress them using E^{2}MC16. Figure 6.14 shows the results of compressing compute-bound benchmarks for E^{2}MC16 for different PDWs. Figure shows that these benchmarks do not gain from E^{2}MC16 and also their performance do not degrade. Only the performance of STO reduces by 4% for 1 PDW as decompression latency is high for 1PDW, however, on average there is only 1% reduction in performance for 1 PDW. Moreover, we propose to use 4 PDWs and we notice no decrease in performance of the compute-bound benchmarks at 4 PDWs.

6.4.6 Effect on Energy

Figure 6.15 shows the reduction in energy consumption and energy-delay-product (EDP) over no compression for E^{2}MC16 for offline and online sampling. On average there is 13% and 27% reduction in energy consumption (E-OFF) and EDP (EDP-OFF) respectively, for offline sampling and 11% and 24% reduction in energy consumption (E-ON) and EDP (EDP-ON) respectively, for online sampling. E^{2}MC16 reduces the energy consumption by reducing the off-chip memory traffic and total execution time. We show energy and EDP of E^{2}MC only over no compression due to lack of power models for BDI and FPC.

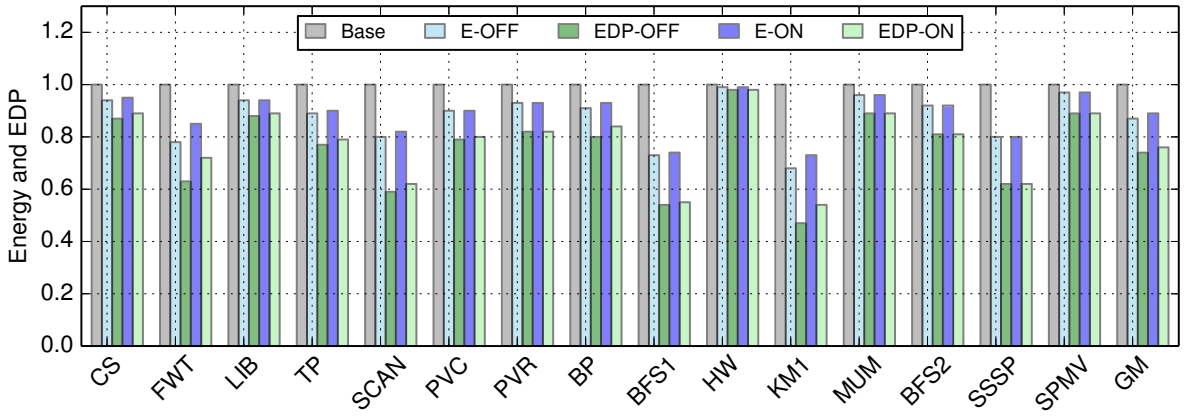


Figure 6.15: Energy and EDP of E^{2}MC16 for offline and online sampling over baseline [85]. © 2017 IEEE

6.5 Summary

In this chapter, we proposed an entropy encoding based memory compression (E^{2}MC) technique for GPUs that delivered higher performance and energy efficiency compared to the state-of-the-art. Most of the previous compression techniques were originally proposed for CPUs and hence, they traded low compression ratio for low decompression latency by exploiting simple patterns for compression. As GPUs are less sensitive to latency than CPUs, we studied the feasibility of a relatively more complex entropy encoding based memory compression technique that has higher potential for compression, but also higher latency. We showed that E^{2}MC delivered higher compression ratio and performance gain than state-of-the-art, provided the key challenges of probability estimation, appropriate symbol length for encoding and decompression with reasonably low latency are addressed properly. E^{2}MC with offline sampling resulted in 53% higher compression ratio and 8% increase in speedup compared to the state-of-the-art techniques and saved 13% energy and 27% EDP compared to no compression. Online sampling resulted in 35% higher compression ratio and 5% increase in speedup compared to the state of the art and saved 11% energy and 24% EDP compared to no compression.

We also provided an estimate of the area and power needed to meet the high throughput requirements of GPUs. We showed that area and power overhead of the E^{2}MC is acceptable with respect to GTX580. For E^{2}MC16, the area and power overhead is 5.8% and 1.5% of the area and power of GTX580, respectively. We think the area numbers are likely higher than expected and a custom design will be denser and will need less area.

A problem that we observed with all the three lossless compression techniques (FPC, BDI, and E^{2}MC) is that their compression ratio at memory access granularity (MAG), which is the effective compression ratio by which the memory bandwidth effectively improves, is much lower than the raw compression ratio. The effective compression ratio is low because the data can only be fetched in a multiple of MAG, however, a memory block

is not always compressed to an exact multiple of MAG. Therefore, we end up fetching more data than needed in order to meet the MAG restrictions.

In the next chapter, we will propose a MAG aware selective lossy compression technique to increase the effective compression and performance gain further. The selective approximation can be toggled on top of a lossless compression depending on an application performance requirement and error tolerance.

7 MAG Aware Selective Lossy Compression Technique

The work presented in this chapter was partially published: S. Lal, J. Lucas, and B. Juurlink, "SLC: Memory Access Granularity Aware Selective Lossy Compression for GPUs," in Design, Automation and Test in Europe, DATE, EDAA, 2019.

In the last chapter, we observed that lossless memory compression techniques have low effective compression ratio due to large memory access granularity (MAG). The effective low compression ratio reduces performance gain that otherwise could result from a higher raw compression ratio. In this chapter, we propose the novel MAG aware selective lossy compression (SLC) technique to increase the effective compression ratio and performance gain. We propose two techniques to implement SLC and present their trade-offs.

7.1 Introduction

Memory compression has been demonstrated as a promising alternative to increase memory bandwidth [26, 146, 160, 77, 85], however, memory compression techniques often exhibit a low effective compression ratio. The main reason for the low effective compression ratio is the large memory access granularity (MAG) exhibited by GPUs due to wide bus width and large burst length. For example, MAG of GDDR5/5X/6 is 32B resulting from 32-bit bus width and 8 burst length. MAG is the amount of data read from or written to a memory by a single read or write command. MAG reduces the compression ratio as data can only be fetched in a multiple of MAG but a compressed block is often not a multiple of a MAG. For example, for a compressed size of 36B, we fetch 64B. Thus, a compression ratio that seems close to 4× (3.6×, assuming a typical block size of 128B in current GPUs) is actually only 2×. This leads to a significant difference between the raw and effective compression ratio actually gained by a system. The raw compression ratio is calculated without considering MAG, while the effective compression ratio is calculated by scaling up the compressed size to the nearest multiple of a MAG.

Figure 7.1 shows the raw and effective compression ratios of Base Delta-Immediate (BDI) [130], Frequent-Pattern Compression (FPC) [6], C-PACK [26], and Entropy Encoding Based Memory Compression (E^2MC) [85] techniques for memory-bound benchmarks included from CUDA SDK [123], Rodinia [25], and AxBench [173]. More details about the benchmarks and experimental setup can be found in Section 7.4. The geometric mean (GM) of the effective compression ratio of BDI, FPC, C-PACK and E^2MC is 22%, 19%,

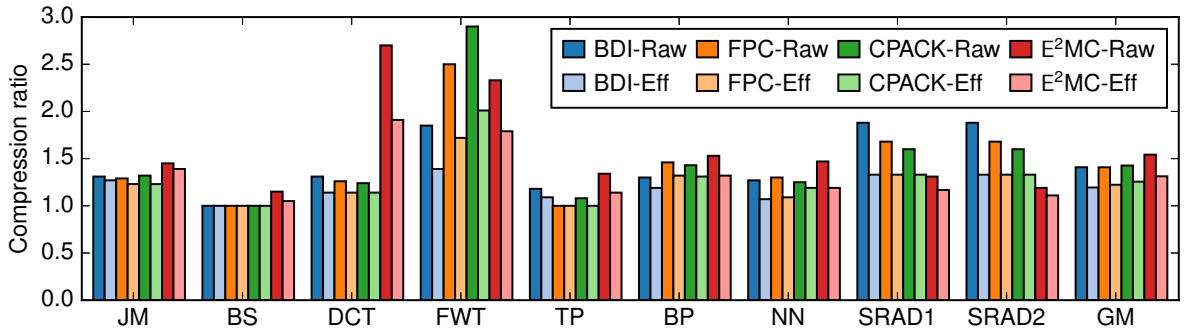


Figure 7.1: Raw and effective compression ratio of BDI, FPC, C-PACK and E^{2}MC using MAG of 32B.

18%, and 23% less than the GM of the raw compression ratio, respectively. The low effective compression ratio reduces performance benefits, otherwise possible from a higher raw compression ratio.

Interestingly, our study of the distribution of compressed blocks (presented in Section 7.2.2) shows that a significant percentage of compressed blocks have only a few bytes above a multiple of MAG. With the goal to reduce the compressed size by these extra bytes, we propose the novel MAG aware Selective Lossy Compression (SLC) technique for GPUs. The key idea of SLC is that when a lossless compression yields a compressed size with a few bytes above MAG, we use lossy compression to approximate these few bytes such that the compressed size is a multiple of MAG. This way, we selectively introduce a small approximation error, however, we significantly increase the compression ratio. Fortunately, there are several GPU applications that are inherently error-resilient to some extent and small approximation will not degrade their output quality to an unacceptable level [141, 45]. Considering that E^{2}MC provides the highest compression ratio and performance gain, we choose E^{2}MC as the baseline lossless compression technique for SLC. However, SLC is not limited to E^{2}MC, but can also be applied to other lossless memory compression techniques.

We propose two techniques for implementing SLC and compare their advantages and disadvantages. A key challenge of SLC is to find the number of symbols needed to be approximated to decrease the compressed size to a multiple of MAG. Since it is not trivial to find the symbols that contribute extra bytes, one simple way is to approximate the whole block. Our first method approximates the whole block using quantization, and we call this technique as *Quantization-based SLC* (QSLC). An advantage of QSLC is that quantization error is uniformly distributed across all symbols of a block. However, a disadvantage of QSLC is that it does not guarantee that the quantized block will be an exact multiple of MAG after the quantization. Our second method determines the number of symbols needed to be approximated to decrease the compressed size by extra bytes and then only approximate these symbols and not the whole block. The second method uses a tree structure to select the symbols for approximation and we call this

technique as *Tree-based SLC* (TSLC). TSLC guarantees that approximated blocks are an exact multiple of MAG, however, this method may result in a slightly higher error as it uses truncation for approximation. Fortunately, we show that the error is not much as we only need to approximate a few symbols and value similarity-based prediction to reconstruct the approximated symbols works quite well. For a lossy threshold of 16B, SLC provides a speedup of up to 17% with $< 1\%$ average error.

In summary, we make the following contributions in this chapter:

- We quantitatively show that low effective compression ratio due to MAG exists in four state-of-the-art techniques and qualitatively in three more.
- We propose the novel MAG aware Selective Lossy Compression (SLC) technique and show a significant performance gain with an acceptable and low accuracy loss.
- We propose two techniques to implement SLC and present their trade-offs.
- This is the first study that highlights the importance of MAG aware compression by quantitatively studying the distribution of compressed blocks above MAG.
- A sensitivity analysis to different MAGs shows an even higher significance of SLC at larger MAG.
- We implement hardware and show the area and power cost of SLC is only 0.0015% and 0.0008% of GTX580.

This chapter is organized as follows. In Section 7.2, we further motivate the problem. In Section 7.3, we present SLC in detail. Section 7.4 explains the experimental setup and experimental results are presented in Section 7.5. In Section 7.6, we discuss the relevance of SLC with emerging DRAM technologies and sector cache. In Section 7.6.2, we conduct sensitivity analysis to different MAG sizes. Finally, we summarize the contributions of this chapter in Section 7.7.

7.2 Motivation

We first show that more memory compression techniques suffer due to large MAG and then study the distribution of compressed blocks to make a case for MAG aware approximation.

7.2.1 Qualitative Analysis of More Compression Techniques

Figure 7.1 quantitatively showed that four state-of-the-art memory compression techniques suffer due to MAG. There are three other techniques: SC^2 [10], HyComp [9] and BPC [77] that can also be applied for memory compression. SC^2 [10] is a statistical cache compression technique and is similar to E^2MC [85] because both are based on Huffman encoding. The former is proposed for CPUs, while the later is proposed for GPUs.

Table 7.1: Summary of qualitative analysis of memory compression techniques.

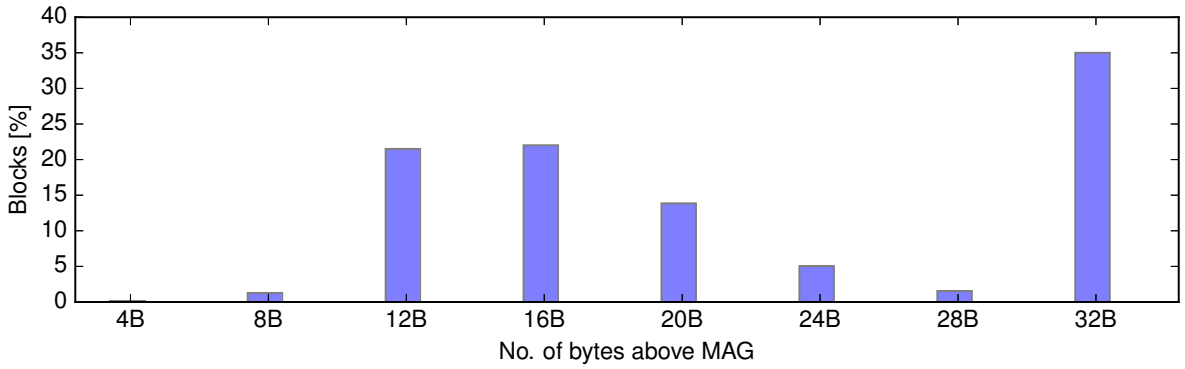
Work	Qualitative analysis	Suffers due to MAG?
SC^2 [10]	Similar to E^2MC [85], already shown to suffer	✓
HyComp [9]	Based on SC^2 [10] and BDI [130], already shown to suffer	✓
BPC [77]	Based on FPC [6] and C-PACK [26], already shown to suffer	✓

Therefore, SC^2 will suffer due to MAG. HyComp is a hybrid compression method which improves the compression ratio by selecting a suitable compression method based on the specific data-type. HyComp will also suffer from MAG as two (BDI and SC^2) out of the four compression methods that HyComp selectively uses are already shown to suffer. The third method called FP-H divides a floating-point number into three fields: Exponent, Mantissa-High, and Mantissa-Low and then employs SC^2 to compress each of these fields in isolation that means FP-H will also suffer from MAG. BPC stands for bit plane compression that uses transformation to increase the compressibility and then uses either run-length or frequent pattern encoding to compress the transformed data. While transformation increases compressibility, BPC will still suffer from MAG as both the run length and frequent pattern encodings exploit patterns similar to FPC and C-PACK which are already shown to suffer in Figure 7.1. Therefore, several memory compression techniques suffer from MAG. Table 7.1 summarizes the qualitative analysis.

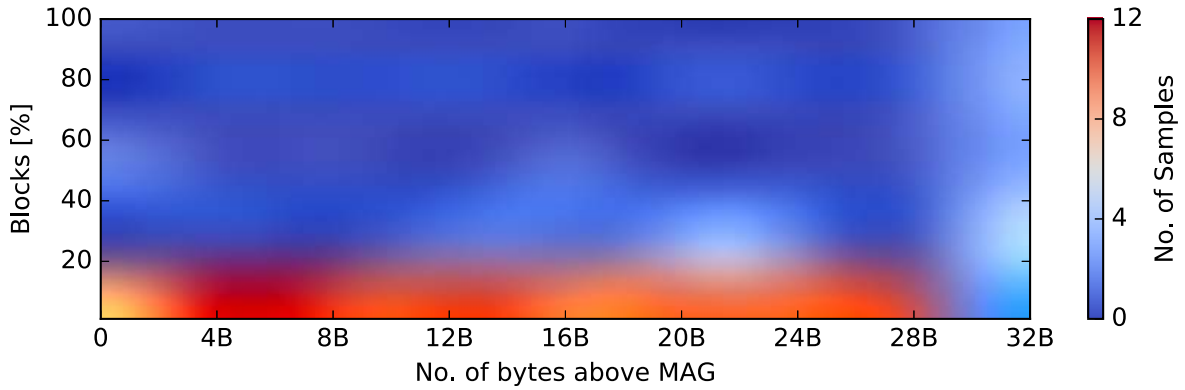
7.2.2 Distribution of Compressed Blocks at MAG

Figure 7.2 shows the distribution of compressed blocks at MAG when E^2MC [85] is used for compression of different benchmarks detailed in Section 7.4.2. We assume a MAG of 32B and a block size of 128B, which are typical values in current GPUs. The x-axis shows the number of bytes above a multiple of MAG. 0B on the x-axis means a compressed block size is a multiple of MAG i.e. 32B, 64B, or 96B. For simplification, all blocks with a compressed size $<$ 32B are also included in the 0B origin. 32B on the x-axis represents the percentage of uncompressed blocks. The left y-axis shows the percentage of blocks and the right y-axis shows the number of samples. The number of samples shows the number of times a certain percentage of blocks e.g. 20% are compressed with a certain number of bytes e.g. 4B above a multiple of MAG for all benchmarks.

Figure 7.2a shows the distribution of compressed blocks for a single benchmark *nn*. There are about 22% of blocks with a compressed size $\leqslant$ 12B above a multiple of MAG and about 45% of blocks with a compressed size $\leqslant$ 16B above a multiple of MAG. Figure 7.2b shows a heat map plot of the distribution of compressed blocks for all benchmarks. We see that all blocks are not compressed to a multiple of MAG and the distribution does not favor any specific size above a multiple of MAG. This is also expected as the probability



(a) Compressed blocks distribution for *nn* benchmark.



(b) Heat map plot showing distribution of compressed blocks.

Figure 7.2: Distribution of compressed blocks above MAG.

that a compressed block will be an exact multiple of a MAG is far less than not an exact multiple of MAG. Ideally, for a high effective compression ratio, all blocks should be compressed to 0B above a multiple of MAG. However, we see there is a significant percentage of blocks that are not compressed to an exact multiple of MAG, but a few bytes above a multiple of MAG. As explained before, there is no way to just fetch these extra bytes, but we have to fetch a whole 32B burst, causing a low effective compression ratio. Nevertheless, these few extra bytes present an opportunity to achieve high effective compression ratio at low accuracy loss by selective approximation.

7.3 Selective Lossy Compression

In this section, we first provide an overview of a system employing Selective Lossy Compression (SLC) and then present architectural details of SLC.

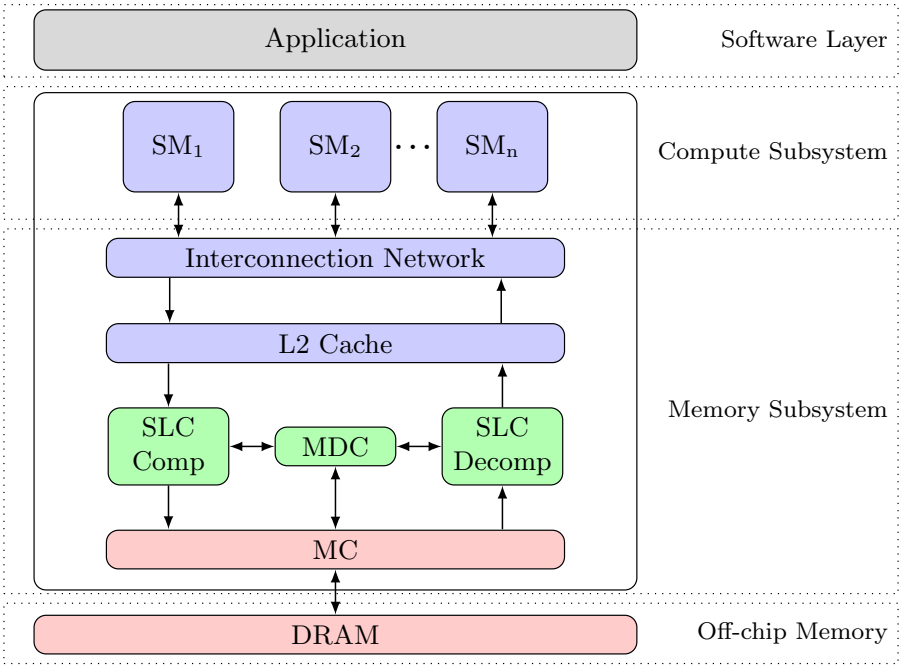


Figure 7.3: Overview of a system with compression components.

7.3.1 Overview of a System with SLC Components

Figure 7.3 shows an overview of a system with main components of SLC technique. The system overview is similar to E^{2}MC as described in Chapter 6. The main difference is that the compressor and decompressor shown in Figure 7.3 also implement SLC on top of E^{2}MC. As indicated by the green color, the memory controller (MC) is modified to integrate the compressor, decompressor, and metadata cache (MDC). As the memory controller needs to fetch only the required number of bursts which can vary from 1 to 4 for every compressed block, we store 2-bit in MDC similar to previous work [146, 85, 160]. Data transfer to and from DRAM is in compressed form with (de)compression taking place in the MC and compression is completely transparent to the L2 cache and the streaming multiprocessors (SM). Similar to E^2 MC, the data is stored in the compressed format in the DRAM and goal also remains the same to increase the effective off-chip memory bandwidth and system efficiency and not to increase the effective capacity of the DRAM, similar to [146, 85]. Hence, a compressed block is still allocated the same size in DRAM similar to E^{2}MC, although it may require less space. Moreover, a block is marked decompressed as soon as we decompress the required number of symbols to recover the original block and the extra data that is fetched due to MAG is meaningless and not interpreted. To decrease latency and reduce performance degradations normally resulting from the overhead of compression and decompression techniques, SLC inherits several novel techniques such as parallel decoding and pipelined design from E^{2}MC as explained in Chapter 6.

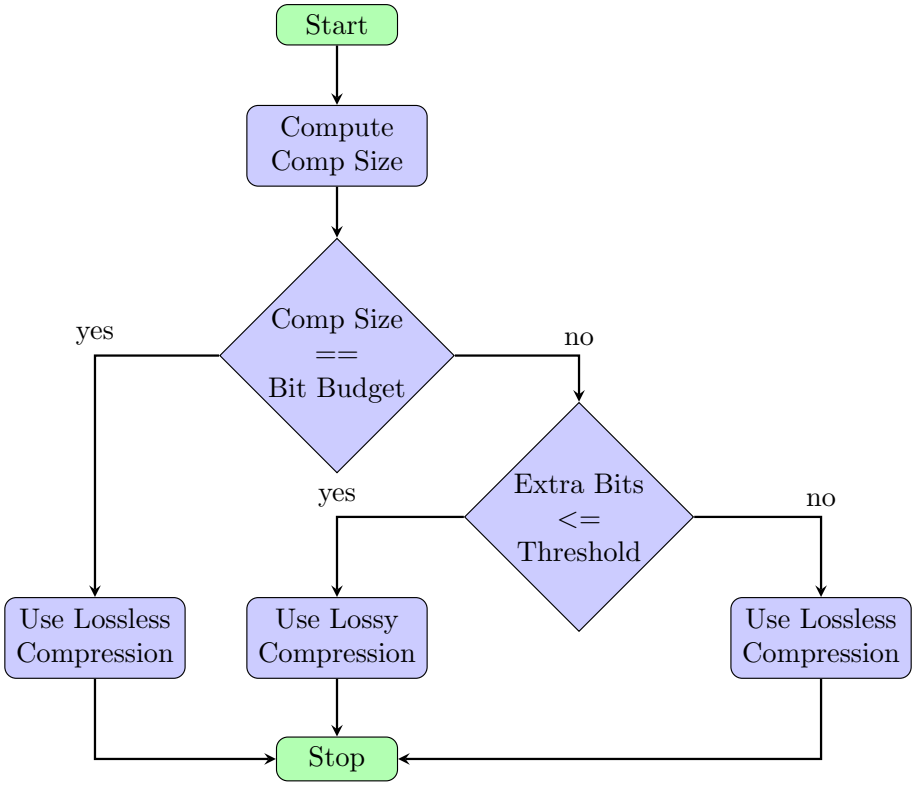


Figure 7.4: Overview of selective lossy compression.

7.3.2 SLC Architecture

Figure 7.4 shows an overview of the SLC technique. Basically, SLC is a budget-based compression technique which allows selection between different compression modes depending upon *comp size*, *bit budget*, *extra bits*, and a *threshold*. SLC selectively uses a combination of lossless and lossy compression modes. The key idea of SLC is that when lossless compression yields a compressed size (*comp size*) that is a few bytes (*extra bits*[1]) above a multiple of MAG, lossy compression is used to approximate these *extra bits* such that the compressed size is a multiple of MAG. Thus, SLC retains the quality of a lossless compression when the compressed size is a multiple of MAG or the *extra bits* are above MAG, however, when the compressed size is not a multiple of MAG and *extra bits* are within the *threshold*, lossy compression is used to achieve the desired compression.

The *bit budget* is a multiple of MAG and in our case, it is either 32B, 64B, 96B, or 128B. When the *comp size* of a block is more than its uncompressed size, the block is always stored uncompressed and in this case, the *bit budget* is 128B. Since it is not possible to fetch less than 32B from memory, we also use lossless compression when the compressed size is less than 32B and in this case, the *bit budget* is also 32B. The *extra bits* are the number of bits above the *bit budget* and the *threshold* is the number of bits defined by

[1]We refer to size above MAG as extra bits instead of extra bytes while explaining SLC in detail as a compressed size is not always a multiple of a byte. Extra bytes is used in other places for readability.

the user that can be safely approximated.

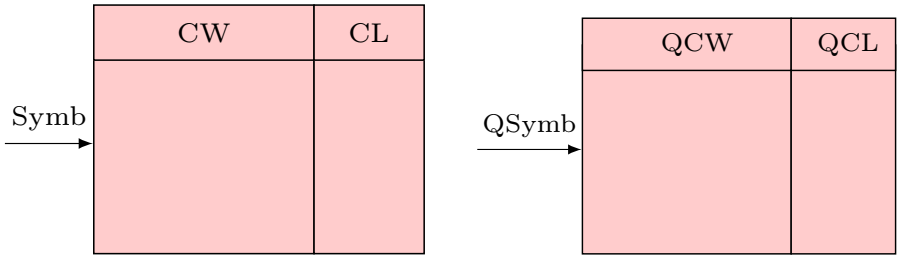
Once we know the *comp size*, we check if it is equal to *bit budget*. We use lossless compression when the *comp size* is equal to *bit budget*. When the *comp size* is not equal to *bit budget*, we use lossy compression if the *extra bits* $\leqslant$ *threshold* and lossless compression if the *extra bits* $>$ *threshold*. A block may be stored uncompressed when the *extra bits* $>$ *threshold*. In nutshell, based on a *bit budget*, *extra bits*, and a *threshold*, SLC selects an appropriate compression mode.

We know how many bytes (*extra bits*) are above a MAG, but the problem is that these *extra bits* are codewords and not symbols. The challenge here is to find the number of symbols that need to be approximated to decrease the compressed size by *extra bits* such that the new compressed size is a multiple of MAG. We propose two techniques to approximate *extra bits* and evaluate their trade-offs. As it is not trivial to find the symbols that contribute these *extra bits*, one simple way is to approximate a whole block when lossy compression mode is selected. Our first method does that using quantization and we call this technique as Quantization-based SLC (QSLC). In our second method, we first determine the number of symbols needed to be approximated to decrease the compressed size by *extra bits* and then only approximate these symbols and not the whole block. The second method uses a tree structure to select the symbols for approximation and we call this technique as Tree-based SLC (TSLC). We first describe how we compute *comp size*, *bit budget*, *extra bits*, and *threshold* in the next Section 7.3.3 and then explain the two methods in detail in Section 7.3.5 and Section 7.3.4, respectively.

7.3.3 Compressed Block Size, Bit Budget, and Extra Bits

To use SLC, the first thing that we require is the compressed block size (*comp size*) that would result if only lossless compression is used. However, we cannot wait for a lossless compression to compress a block and then decide which compression mode to use as compression incurs long latency. Although GPUs can hide compression latency, too much increase can also start degrading their performance [85]. Fortunately, we only need to know the *comp size* and not the compressed block to choose a compression mode and the *comp size* can be easily calculated by just adding all code lengths. As explained by Lal et al. [85], obtaining codewords and code lengths from Huffman compressor table is much faster than placing the codewords together. There are different ways to sum these code lengths such as using an accumulator or a parallel tree adder. We use an accumulator for QSLC and a parallel tree adder for TSLC for the reasons described in Section 7.3.5. RTL synthesis details to obtain the *comp size* are described in Section 7.4.1.

Once the *comp size* is known, *bit budget* of a block can be computed. The *bit budget* is the closest multiple of MAG that is less than or equal to the *comp size*. The possible values of bit budget are 32B, 64B, 96B, or 128B. The *bit budget* is dynamically calculated for each block from its compressed size. Since it is not possible to fetch less than 32B from memory, *bit budget* is also 32B when the compressed size is less than 32B. The *extra bits* are simply calculated by subtracting the *bit budget* from the *comp size*.



(a) Normal Huffman table. (b) Quantized Huffman table.

Figure 7.5: Huffman compressor with extra table for quantization.

7.3.4 Quantization-based SLC

Quantization-based SLC (QSLC) uses quantization to implement the lossy compression mode. We use the compressor and decompressor proposed by Lal et al. [85] as base designs and adapt them to implement QSLC. QSLC uses two Huffman tables for compression. A Huffman table is a small lookup table that stores codewords (CWs) and their code lengths (CLs). The table is indexed by a symbol to get its CW and CL. The first table is a normal Huffman table as used by Lal et al. [85] for lossless compression. The second table is a quantized Huffman table that stores quantized codewords (QCWs) and their code lengths (QCLs). This table is built in a similar way to the first table except that the symbols are quantized before generating the codewords. More details about the generation of Huffman codewords can be found in [85]. We quantize the least significant bits as they introduce the least error and vary the number of quantization bits (4, 8, 12, and 16-bit). Our baseline E^{2}MC uses 16-bit symbols, thus, we only quantize the first symbol of a 32-bit word. The quantized Huffman table is used for the lossy compression mode and the normal Huffman table is used for the lossless compression mode. The quantized symbols are expected to generate smaller codewords compared to no quantization as several unquantized symbols will be quantized to a single symbol with higher probability and thus, a quantized symbol gets smaller codeword.

The base Huffman decompressor is also modified to implement QSLC. The main modification is the addition of second decompressor lookup table (De-LUT) to store the quantized symbols along with unquantized symbols and some arbitration logic to select the appropriate De-LUT.

An advantage of QSLC is that we do not need to find the number of symbols needed to be approximated to reduce the compressed size to a multiple of MAG and quantization error is uniformly distributed across an approximated block. A disadvantage of QSLC is that it does not guarantee that a quantized block will be an exact multiple of MAG and a slightly higher hardware overhead if 4 or 8-bit QSLC is used as detailed in Section 7.3.9.

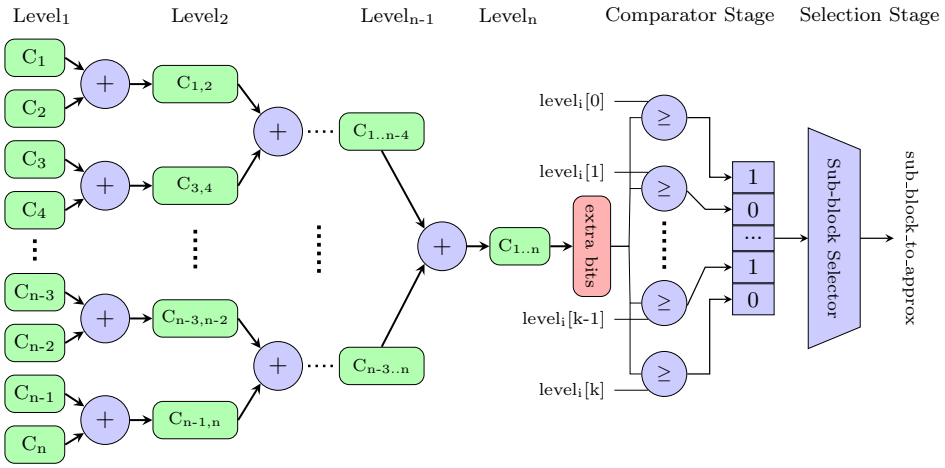


Figure 7.6: Tree-based SLC.

7.3.5 Tree-based SLC

In this method, we first determine the number of symbols needed to be approximated to reduce the compressed size to a multiple of MAG and then only approximate these symbols and not the whole block. We use a parallel tree adder to add all code lengths of a block to find the compressed size (*comp size*) and intermediate sums of the code lengths to select the symbols for approximation as shown in Figure 7.6. The last node of the tree contains the *comp size* that is used to find the *bit budget* and *extra bits* as described in Section 7.3.3. A parallel tree adder could also be used for QSLC to know the compressed size of a block, however, an accumulator is sufficient and cheaper in hardware as QSLC does not need intermediate sums of code lengths. The use of intermediate sums of the code lengths to select the symbols for approximation is explained below.

When the lossy compression mode is selected, the *extra bits* are compared with the intermediate sums at all levels in parallel as shown in Figure 7.6. The output of these comparisons is written to a bit vector. It may happen that we do not find any sub-block with compressed size (intermediate sum) $\geq$ *extra bits* at some levels. The output of the comparison stage is all zeros for these levels. In the sub-block selection stage, priority encoders are used to output the indices of the first sub-block with sum $\geq$ *extra bits* for each level of the tree. Finally, a sub-block (*sub_block_to_approx*) with compressed size $\geq$ *extra bits* from the lowest level (*approx_level*) is selected for approximation as at this level we need to approximate the fewest symbols. As the *sub_block_to_approx* is selected in parallel, the latency is fixed regardless of the approximated level. The latency overhead is described in Section 7.4. Once *sub_block_to_approx* is selected, the start symbol for approximation is obtained by: $sub\_block\_to\_approx \times 2^{approx\_level}$.

An optimization that will likely result in slightly less error is to find all the sub-blocks with compressed size $\geqslant$ *extra bits* and then select the smallest, however, that would require more comparisons. A further optimization that can save comparisons at the sub-block selection stage is to first estimate the approximation level (*est_approx_level*) using the

Algorithm 1 Estimated approximation level

```
1: procedure ApproxLevel(bit_budget, extra_bits, num_symb)
2:     avg_comp_size_per_symb = (bit_budget + extra_bits)/num_symb
3:     num_symb_to_approx = extra_bits/avg_comp_size_per_symb
4:     if num_symb_to_approx < 1 then
5:         num_symb_to_approx = 1
6:     end if
7:     est_approx_level = log2(num_symb_to_approx)
8: end procedure
```

pseudo Algorithm 1 and then only compare the *extra bits* at levels $\geq$ *est_approx_level* in parallel. As shown in pseudo algorithm, first the average compressed size per symbol (*avg_comp_size_per_symb*) is computed as shown in line 2 and then the estimated number of symbols needed to be approximated (*num_symb_to_approx*) as shown in line 3. The log of the number of symbols to approximate (*num_symb_to_approx*) gives the estimated approximation level (*est_approx_level*) as shown in line 7.

During the decompression of a block, we check if the symbol being decoded is approximated or not. When the symbol being decoded is not approximated, we decompress it normally as in [85], otherwise, we predict its value as described in Section 7.3.6.

7.3.6 Value Similarity-based Prediction

In TSLC, we simply truncate the symbols selected for approximation during compression that guarantees the desired compression, however, this method may result in a higher error due to truncation. To reduce the error resulting from truncation, we predict the value of approximated symbols during decompression. The challenge in designing a predictor for GPUs is that a single memory request is a SIMD load that produces values for all threads in a warp and a multi-value predictor will be quite expensive accounting that there are many active warps in a GPU. Fortunately, the research has shown that SIMD loads in GPUs exhibit high value similarity [141, 175] and our experimental results also reveal the same. While a SIMD load fetches several data elements for a warp, adjacent threads in a warp operate on values that have significant value similarity, for example, adjacent pixels of an image. This implies that even a very simple value replication could also provide a reasonable accuracy for the predicted symbols. Considering that we only need to predict a few symbols of a block and adjacent threads have significant value similarity, we decide to use a simple value similarity-based prediction scheme. The prediction works as follows. While decoding an approximated block, we use the value of the first symbol of the block being decoded as its predicted value when the first symbol itself is not approximated and the value of the first non-truncated symbol when the first symbol itself is approximated. In terms of decompressor hardware change, we only need to generate the index of the predicted value when an approximated symbol is decoded.

While there are exact value predictors [54, 131] and stride predictors [147, 41] with

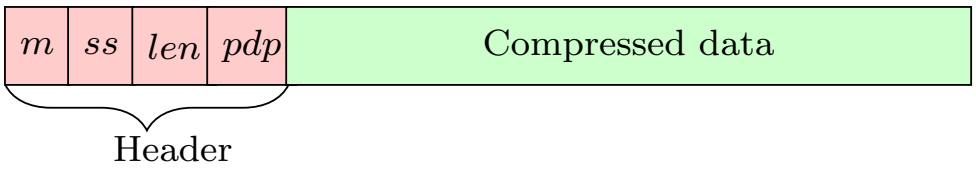


Figure 7.7: Structure of a compressed block.

trade-offs in terms of accuracy and complexity, we opt for the very simple prediction scheme due to its negligible hardware cost and reasonable accuracy for our use case.

7.3.7 TSLC Optimization

TSLC may approximate significantly more bits than needed to reduce the compressed size to a multiple of MAG due to coarse intermediate sums at middle levels in the tree. This can happen as a node at level *l+1* has a sum of two times the nodes at level *l* as shown in Figure 7.6 and when we cannot find a sub-block with compressed size $\geq$ *extra bits* at level *l*, we move to level *l+1* and it may be the case that the largest sub-block at level *l* is only a few bits less than the *extra bits*. The experiments show that a significant unneeded approximation may happen at the middle levels (3 and 4). The high unneeded approximation does not happen at lower levels (< 3) as the intermediate sums are smaller and it also does not occur at higher levels (> 4) because we can mostly find a sub-block to approximate at the middle levels. To reduce the unneeded approximation, we further optimize TSLC by adding a few extra intermediate nodes at the middle levels. We add 8 and 4 extra nodes to have less coarse sums at levels 3 and 4, respectively, which originally have 16 and 8 nodes. We can further optimize by having even fine-grained sums, however, that will require more hardware resources.

7.3.8 Structure of a Compressed Block

Figure 7.7 shows the structure of a compressed block which consists of a header and compressed data. The header information is needed to decompress a block. The header consists of 1-bit (m) to indicate the compression mode that could be either lossless or lossy, 6-bit to store the first approximated symbol index (ss), 4-bit to store the number of approximated symbols (len), and 3 parallel decoding pointers (pdp) for 4 parallel decoding ways (PDWs). Our experiments show that the maximum number of the approximated symbols is 16, thus we need 4-bit to store the *len*. SLC uses 4 PDWs to reduce the decompression latency as we show the highest performance gain for 4 PDWs in Chapter 6. Each pointer consists of N bits, where 2^N is the block size in bytes. The *ss* and *len* fields are only needed for TSLC and not for QSLC as later approximates the whole block. No header is needed as in the baseline [85] when a block cannot be compressed.

Table 7.2: Frequency, area, and power of TSLC and QSLC.

	Compressor			Decompresor		
Technique	Freq (GHz)	Area (mm^2)	Power (mW)	Freq (GHz)	Area (mm^2)	Power (mW)
TSLC	1.43	0.00830	1.620	0.80	0.00030	0.210
QSLC-4b	1.43	0.15000	4.200	0.80	0.05200	1.230
QSLC-8b	1.43	0.09000	2.800	0.80	0.02600	0.910
QSLC-12b	1.43	0.00350	1.150	0.80	0.00370	0.600
QSLC-16b	1.43	0.00070	0.240	0.80	0.00001	0.001

7.3.9 Hardware Implementation and Overhead

To estimate the frequency, area and power overhead of the proposed SLC techniques, we implement TSLC and QSLC in RTL and then synthesize the designs using Synopsis design compiler version K-2015.06-SP4 targeting 32 nm technology node. Table 7.2 shows the frequency and the additional hardware overhead of extending E^{2}MC with TSLC and QSLC with the different number of quantization bits. We only present synthesis results for one variation of TSLC (optimized TSLC with prediction) as the differences are insignificant. The area and estimated power of QSLC-4b and QSLC-8b are higher than the TSLC as QSLC requires additional SRAM tables to store the quantized codewords and the quantized symbols for decompression lookup to implement lossy compression mode. The overhead decreases for higher quantization as the number of quantized symbols and codewords decreases. More importantly, both the area and power overhead of TSLC and QSLC are very small with respect to the GTX580, which has 540 mm^2 area and 240 W power consumption. The area and power cost of the TSLC is about 0.0015% and 0.0008% of GTX580, while the area and power cost of the QSLC-12b is about 0.0013% and 0.0007% of GTX580. TSLC adds 5.6% of the area of E^{2}MC, while QSLC-12b and QSLC-16b only add 4.7% and 0.5% of the area of E^{2}MC, respectively.

7.3.10 Safe to Approximate Loads and Approximation Threshold

Figuring out which loads and stores can be approximated is an important initial step for any approximation technique as not all variables and operations can be approximated. For example, loads that affect critical data segments, pointer addresses, array indices, or branch conditions cannot be safely approximated as they may result in segment faults or executions that are completely unacceptable. As the previous research has shown that safety is a semantic property of the program [128, 142] and to identify whether a load or a region of a program is safe to approximate or not, it requires programming language support. Moreover, it is a common practice that a programmer annotates the

loads or code regions with the support of programming language [174, 44, 175, 168, 146]. Similar to previous work, SLC also requires programmer annotations to find the loads that can be safely approximated. However, instead of burdening a programmer with the task of identifying individual loads, we propose a model that is much easier to use for a programmer, cheaper to implement in the hardware and well suited for programming GPUs. In our model, a programmer specifies if a memory region is safe to approximate during memory allocation by using an extended cudaMalloc() API as shown in Listing 7.1. We extend gpgpu-sim by implementing the extended version of cudaMalloc(). The address returned by the extended cudaMalloc() and size of the memory allocation is used to determine if a load is safe to approximate or not. Therefore, we model a solution similar to a memory type range register. For the benchmarks used in this chapter, we exclude memory regions which can cause a catastrophic failure such as segmentation fault etc. Table 7.4 also lists the number of approximated memory regions for each benchmark.

In a real GPU, the safe to approximate status of a given address can also be determined by using an extra status bit in a page table or an extra bit of a physical address. All the three methods have their own advantages and disadvantages in terms of ease of implementation, cost, and flexibility. However, in terms of performance, we only expect minor differences. Since the regular gpgpu-sim does not model a GPU MMU or TLBs, the memory type range register solution was easier to implement in gpgpu-sim by using internal data structures. However, a GPU architect can choose the method that is most suitable for the given design. In general, the most flexible solutions such as a status bit in a page table or an extra address bit within the physical address have a higher overhead. For example, to set a bit in the page table requires some help from the OS and passing the information to the memory controller. A single memory type range register is sufficient when the memory is allocated contiguously, otherwise, it requires multiple memory type range registers. For simplicity, we assume contiguous memory allocation for modeling memory type range register solution in gpgpu-sim. Moreover, we think that the memory is often allocated contiguously in GPUs. For example, as shown by Ausavarungnirun et al. [13], GPU applications perform most of their memory allocation en masse (i.e., they allocate a large number of base pages at once) and an intelligent memory allocation policy can ensure that base pages that are contiguous in virtual memory are allocated to contiguous physical memory pages. We think that such smart policies can further simplify approximation control integration.

```
cudaMalloc(void** devPtr, size_t size, bool safeToApproximate,
size_t threshold)
```

Listing 7.1: Extended API

Likewise, a programmer needs to specify a lossy threshold that satisfies the target output quality and maximizes the benefits of SLC. From the programmer's perspective, the lossy threshold is a knob that enables the evaluation of trade-off between performance

Table 7.3: Baseline simulator configuration.

Parameter	Value	Parameter	Value	Parameter	Value
#SMs	16	L1 $ size/SM	16 KB	Memory type	GDDR5
SM freq (MHz)	822	L2 $ size	768 KB	Memory clock	1002 MHz
Max #Threads/SM	1536	#Registers/SM	32 K	Memory bandwidth	192.4 GB/s
Max #CTA/SM	8	Shared memory/SM	48 KB	Burst length	8
Max CTA size	512	# Memory controllers	6	Bus width	32-bit

gain and quality. The lossy threshold can be specified by the programmer or statically determined by profiling. We experiment with different lossy thresholds of 8B, 16B, 24B, representing extra bytes above MAG.

The goal of this work is to show the benefits of MAG aware selective approximation and to achieve this goal, we use well-known coarse-grain annotation of loads. However, a more fine-grained annotation with the support of programming language can be used. For example, Truffle [44] uses disciplined approximate programming for annotation, proposes ISA extensions which allow a compiler to convey what can be approximated and then maps ISA to approximation-aware micro-architecture. Moreover, other approximation frameworks such as Green [14] which provides statistical guarantees on the quality of service, Rumba [75] which dynamically detects and corrects large errors can be integrated into SLC. However, this is beyond the scope of this work.

7.4 Experimental Setup

7.4.1 Simulator

We use gpgpu-sim v3.2.1 [15] and modify it to integrate BDI, FPC, C-PACK, E^{2}MC and SLC techniques. We configure gpgpu-sim to simulate a GPU similar to NVIDIA's GTX580. The baseline simulator configuration is summarized in Table 7.3, which is similar to Chapter 6. More details of the simulator can be found in [15].

For our baseline lossless compression E^{2}MC, we use 16-bit symbols, 4 PDWs to reduce the decompression latency and online sampling size of 20 million instructions as they provide the highest compression ratio and performance gain [85]. We synthesize RTL designs of E^{2}MC, TSLC, and QSLC to accurately estimate their frequency as described in Section 7.3.9. It takes 46 cycles to compress and 20 cycles to decompress a block for the baseline lossless compression. As we only need to know the compressed block size to select a compression mode, we read all code lengths of a block from the compressor table and add them. We assume a block size of 128B and 16-bit symbols, therefore, a total of 64 code lengths need to be read. RTL synthesis shows that it only takes 0.18 ns to get a code length from the compressor table, which means that all code lengths of a block can be fetched in about 12 cycles at 1002 MHz and it requires another 2 cycles

Table 7.4: Benchmarks used for experimental evaluation.

Name	Short Description	Input	Error Metric	#Approx. Regions
JM	Intersection of two tri. [173]	400 K tri. pairs	Miss rate	6
BS	Options pricing [123]	4 M options	MRE	4
DCT	Discrete cosine transform [123]	1024×1024 image	Image difference	2
FWT	Fast walsh transform [123]	8 M elements	NRMSE	2
TP	Matrix transpose [123]	1024×1024 matrix	NRMSE	2
BP	Perceptron training [25]	64 K elements	MRE	6
NN	Nearest neighbors [25]	20 M records	MRE	2
SRAD1	Anisotropic diffusion [25]	1024×1024 image	Image difference	8
SRAD2	Anisotropic diffusion [25]	1024×1024 image	Image difference	6

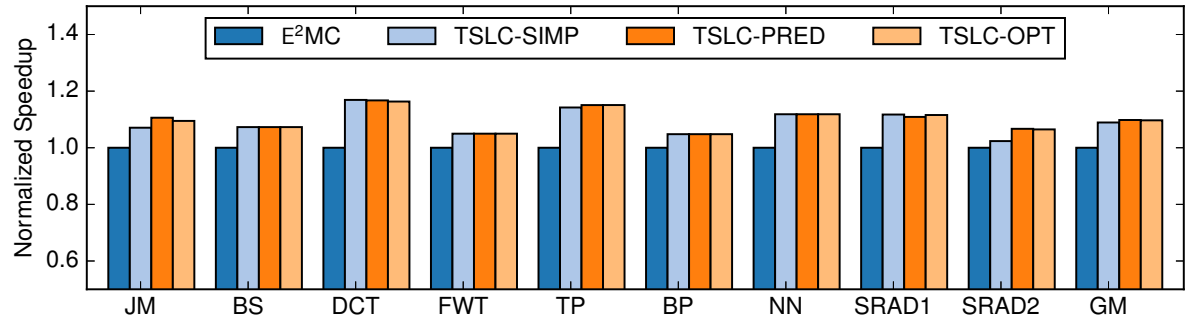
to add all the code lengths to obtain the compressed block size and select a sub-block for approximation. Thus, TSLC and QSLC need 60 cycles to compress a block. Due to very simple additional decompression logic, a block in SLC can be decompressed in the same number of cycles as in E^{2}MC. For estimating the energy consumption of different benchmarks, GPUSimPow [99] is modified to integrate the power models of the compressor and decompressor for E^{2}MC, TSLC, and QSLC. The power numbers obtained from RTL synthesis are used to derive the power models for the compressors and decompressors.

7.4.2 Benchmarks

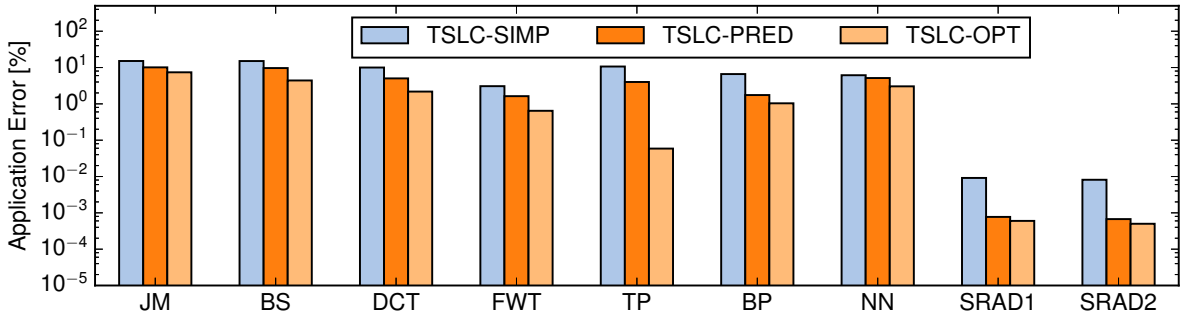
Table 7.4 shows the benchmarks used for the evaluation of SLC technique. We include memory-bound and amenable to approximation benchmarks from CUDA SDK [123], Rodinia [25], and AxBench [173], covering a diverse range of applications. We use speedup and application specific error metrics to explore the trade-off between performance and accuracy which commensurates with previous work [175, 44, 168, 45]. We use mean relative error (MRE) for applications that produce numeric outputs and Normalized Root Mean Square Error (NRMSE) that process images or belong to a signal processing domain. JM finds the intersection between triangles and we use miss rate to report the fraction of incorrect decisions.

7.5 Experimental Results

First, we evaluate the speedup and error of TSLC and QSLC normalized to E^{2}MC and then present the results for bandwidth, energy, and energy-delay-product (EDP) reductions due to TSLC and QSLC. We conduct the above mentioned experiments with a 16B lossy threshold. Finally, we study the sensitivity of TSLC and QSLC to different lossy thresholds, MAG sizes, and block sizes.



(a) Speedup



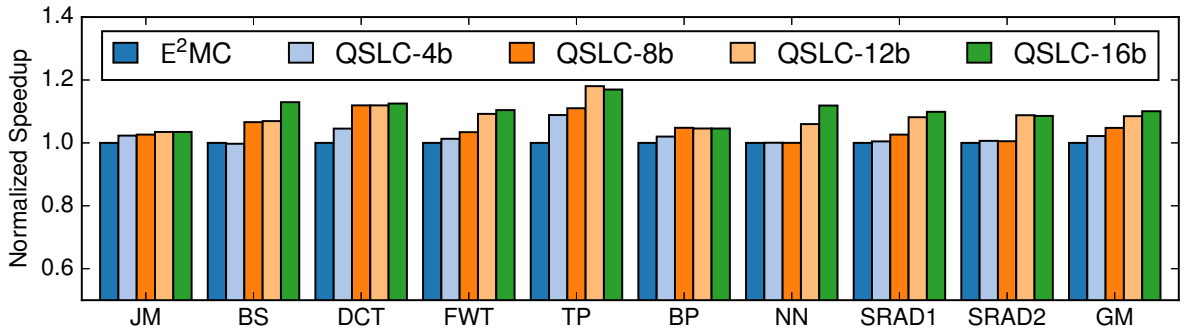
(b) Error

Figure 7.8: Speedup and error for TSLC.

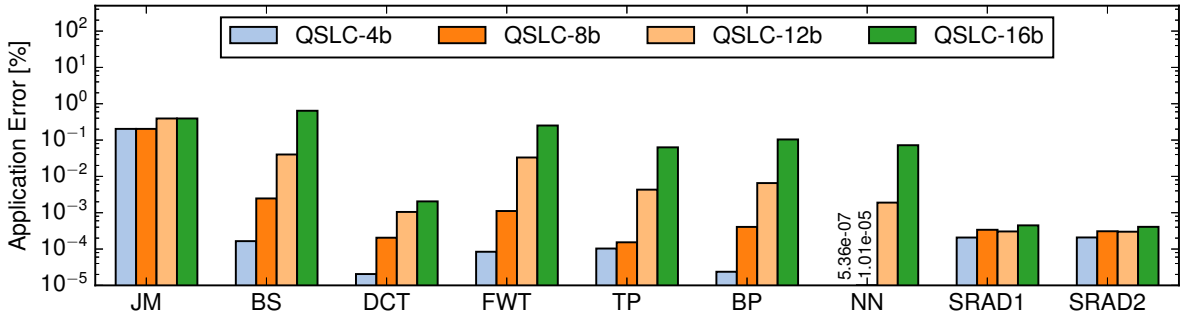
We present results for three variations of TSLC. The first variation is called TSLC-SIMP, where we simply truncate the selected symbols during compression and replace them with zero values during decompression. The second variation is called TSLC-PRED; in this variation, we truncate the selected symbols during compression and use value similarity-based prediction during decompression. The third variation is called TSLC-OPT, where in addition to prediction, we further optimize TSLC-PRED by adding few extra nodes at middle levels of the tree. This helps to reduce the error by decreasing unneeded approximation. We present four variations of QSLC for four quantization levels (4, 8, 12 and 16-bit). We provide speedup and application specific error (see Table 7.4) for individual benchmark and geometric mean (GM) of the speedup of all benchmarks.

7.5.1 Speedup

Figure 7.8 and Figure 7.9 show speedup and error for TSLC and QSLC respectively, normalized to E^{2}MC [85]. Figure 7.8a shows that all three variations of TSLC provide higher speedups compared to E^{2}MC. The maximum speedup is about 17% for *DCT* and the minimum speedup is about 5% for *FWT* and *BP*. The geometric mean of the speedup is 9%, 9.8%, 9.7% for TSLC-SIMP, TSLC-PRED, and TSLC-OPT, respectively. There is only slight variation in the average speedup of the three schemes which is expected



(a) Speedup



(b) Error

Figure 7.9: Speedup and error for QSLC.

because all of them roughly approximate the same number of blocks by the same amount. However, a slight variation in speedup occurs due to the differences in the way the three schemes perform compression and decompression. All the three schemes truncate during compression, however, TSLC-OPT may truncate slightly less number of symbols. During decompression, TSLC-SIMP uses zeros for approximated symbols, TSLC-PRED and TSLC-OPT employ prediction, but the number of predicted symbols may differ. Thus, due to above mentioned differences, a decompressed block may differ from one scheme to another. Because of the differences in the decompressed blocks, the further compressibility of these blocks and the blocks which depend on them may differ and hence we see slight variations in the speedups of the three schemes.

Figure 7.8b shows error for different variations of TSLC. The error reduces from TSLC-SIMP to TSLC-PRED. TSLC-SIMP has the highest error among the TSLC schemes due to truncation of the selected symbols. We see that the error reduces significantly for TSLC-PRED which shows that the value similarity-based prediction works quite well. Our results commensurate with previous work on value similarity-based prediction [24, 54]. The error reduces further for TSLC-OPT because it reduces unneeded approximation by adding a few extra nodes in the tree. The error is $< 3\%$ except for *JM* (7.3%) and *BS* (4.4%) when TSLC-OPT is used and for TP (0.05%), SRAD1 (0.001%), SRAD2 (0.001%) is extremely low. The reason for slightly high error (miss rate) for JM is that

it finds an intersection between two triangles and outputs a boolean that may flip due to approximation. In general, the error is much lower than 10% that is treated as an acceptable error in several related works [108, 45, 44]. In addition to application-specific error, we also calculated mean relative error for individual benchmark which enables averaging the error across all benchmarks. For TSLC-OPT, the geometric mean of the mean relative error for all benchmarks is 0.99%.

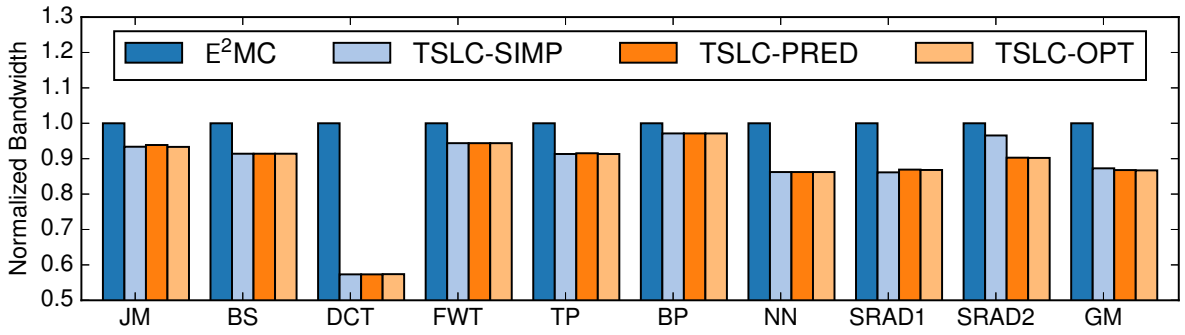
Figure 7.9a and Figure 7.9b show speedup and error of QSLC for 4, 8, 12, and 16-bit quantization. The geometric mean of the speedup is 2%, 4.8%, 8.5%, and 10.1% for 4, 8, 12, and 16-bit quantization, respectively. In general, the error is quite low which ranges from a minimum of 10^{-7}% for *nn* with 4-bit quantization to the maximum of 0.64% for *BS* with 16-bit quantization. Both the speedup and error increase with the increase in the number of quantization bits which is expected because more blocks get compressed to a multiple of MAG when the number of quantized bits is increased. The results are interesting as it is possible to achieve a maximum speedup of up to 17% and an average speedup of more than 10% with 0.64% maximum loss in the accuracy. Similar to TSLC, we also calculated mean relative error for individual benchmark to calculate average error across all benchmarks. The geometric mean of the mean relative error for all benchmarks is 0.0006%, 0.004%, 0.04%, and 0.2% for 4, 8, 12, and 16-bit quantization, respectively.

The average speedup of QSLC with 12-bit quantization (8.5%) and 16-bit quantization (10.1%) is close to the average speedup of TSLC (9.5%). As discussed before, QSLC does not guarantee that a compressed block will be a multiple of MAG after approximation, however, from the comparison of TSLC and QSLC speedups, we can infer that 12-bit and 16-bit quantizations are roughly approximating the same number of blocks to a multiple of MAG as TSLC.

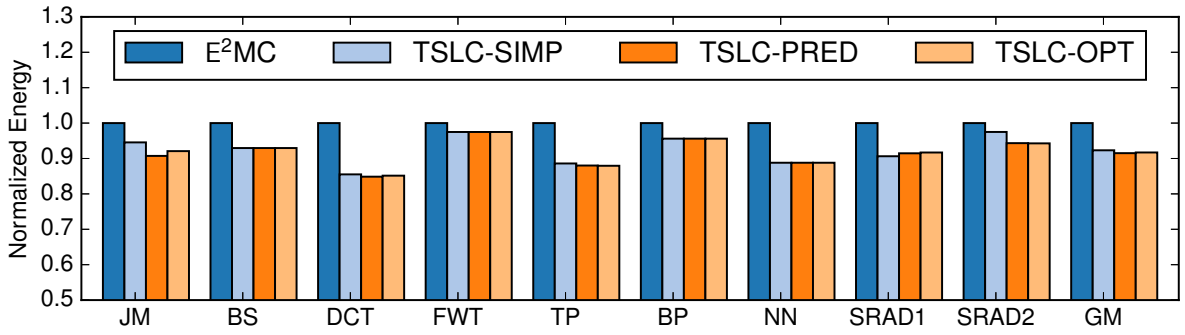
The main reason for differences in speedup and error between TSLC and QSLC is that both techniques compress blocks differently. TSLC only approximates the selected symbols while QSLC approximates the whole block. While TSLC introduces relatively high error due to truncation which is significantly reduced with a simple value similarity-based prediction, QSLC results in relatively less error as only the least significant bits are quantized. In summary, we show that both TSLC and QSLC provide significant performance gain with a very small and acceptable loss in accuracy.

7.5.2 Bandwidth and Energy Reduction

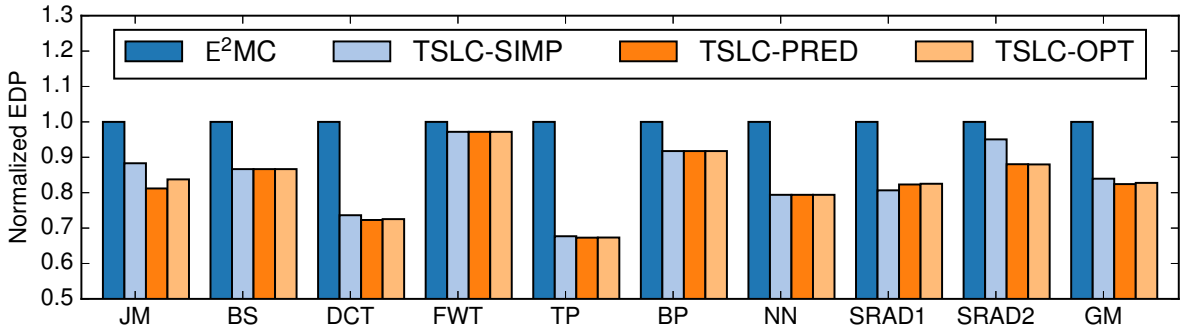
Figure 7.10 and Figure 7.11 show bandwidth, energy, and energy-delay-product (EDP) reductions normalized to E^{2}MC for TSLC and QSLC, respectively. The reduction in off-chip memory bandwidth, energy, and EDP results from the increased effective compression ratio due to the use of SLC. Figure 7.10a and Figure 7.11a show the reduction in memory bandwidth normalized to E^{2}MC for TSLC and QSLC, respectively. The geometric mean of the reduction in memory bandwidth is about 14% for all three variations of TSLC. For QSLC, the geometric mean of the reduction in memory bandwidth is 2%, 8.6%, 14.2%, and 15.8% for 4, 8, 12, and 16-bit quantization, respectively. The reduction



(a) Bandwidth reduction



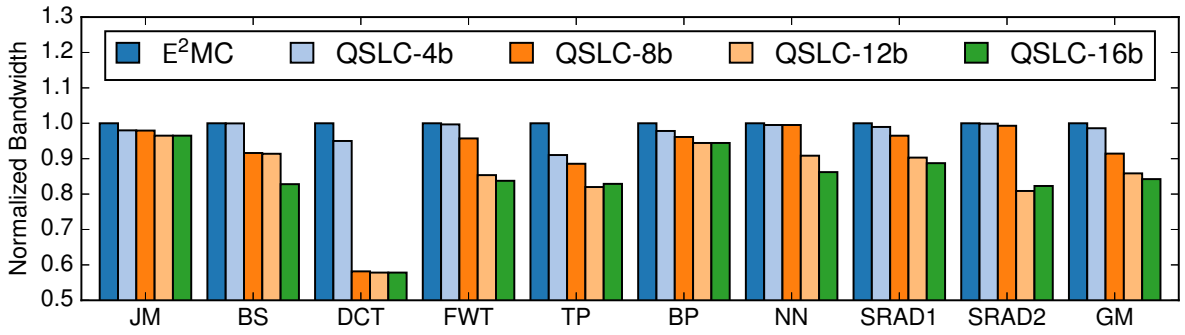
(b) Energy reduction



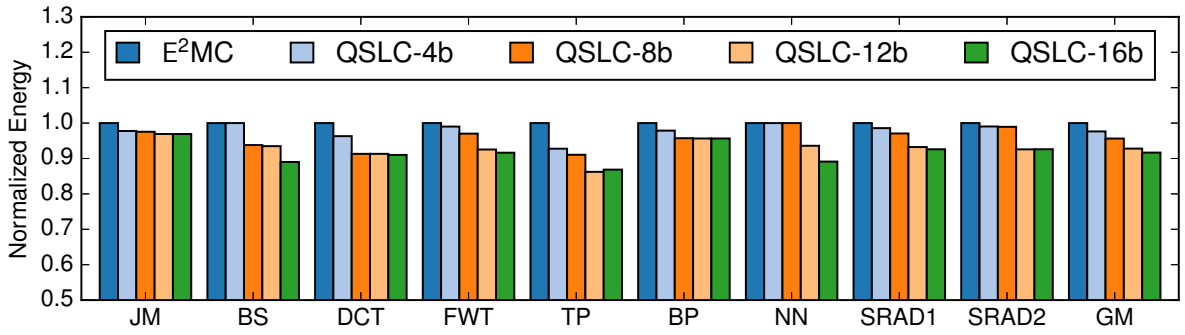
(c) EDP reduction

Figure 7.10: Bandwidth, energy, and EDP reduction for TSLC.

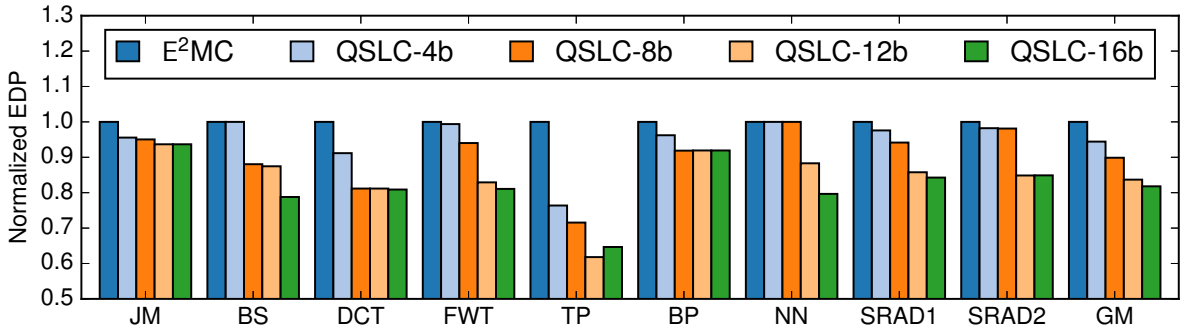
in bandwidth, which is a reciprocal of the compression ratio, is due to the increase in the effective compression ratio resulting from saving 32B burst transactions. Again, we observe that TSLC and QSLC with 12-bit quantization have nearly the same bandwidth reductions which commensurate with the speedup results presented in Section 7.5.1. The results show a relatively higher decrease in bandwidth for some benchmarks such as DCT, however, there is not that much increase in speedup. This is because some loads can be more performance critical than others [175]. Thus, we can have lower gain in performance in some cases, although there is a higher reduction in bandwidth that still can be useful.



(a) Bandwidth reduction



(b) Energy reduction



(c) EDP reduction

Figure 7.11: Bandwidth, energy, and EDP reduction for QSLC.

For example, modern GPUs support concurrent execution of kernels and the reduction in bandwidth requirements of one kernel offers a higher share of bandwidth to other concurrently running kernels. Moreover, the bandwidth reduction can also decrease the memory power consumption even when there is no increase in speedup.

Figure 7.10b and Figure 7.10c show the reduction in energy and EDP normalized to E^{2}MC for TSLC. The geometric mean of the reduction in energy consumption and EDP is about 8.3% and 17.5% for TSLC. Figure 7.11b and Figure 7.11c show the reduction in energy and EDP normalized to E^{2}MC for QSLC. The geometric mean of the energy and EDP reductions is 2.0%, 6.0% for 4-bit, 4.0%, 10.0% for 8-bit, 7.2%, 16% for 12-bit, and

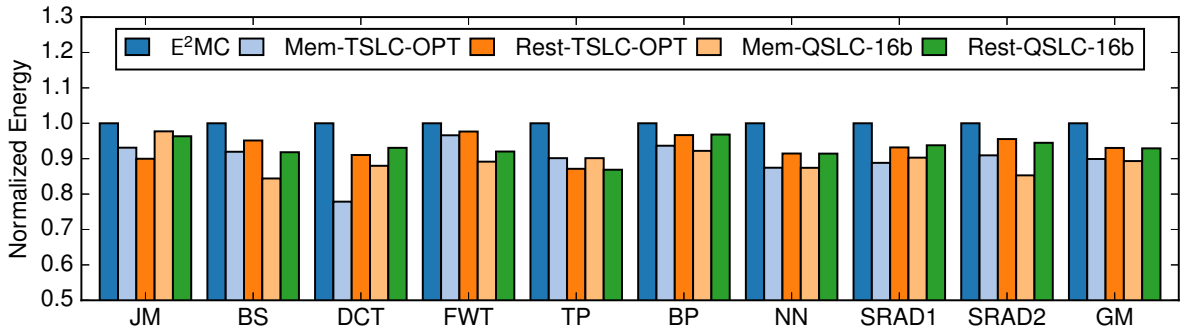


Figure 7.12: Energy reduction of the memory system and rest of the GPU.

8.4%, 18.2% for 16-bit quantizations, respectively. The reduction in energy and EDP is nearly equal to TSLC for 16-bit quantization. SLC reduces the energy consumption by reducing the off-chip memory traffic and total execution time.

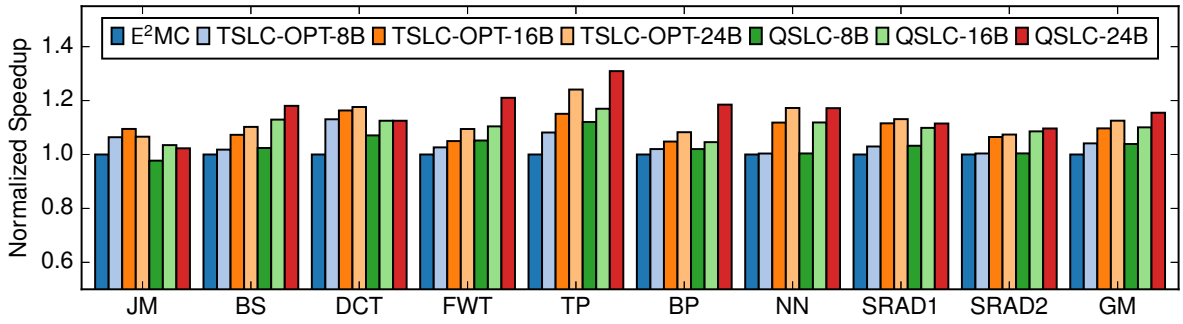
7.5.3 Energy Breakdown

Figure 7.12 shows the separate energy reduction of the memory system (off-chip memory, memory controller along with the integrated compressor and decompressor) and rest of the GPU normalized to E^{2}MC. We only show results for TSLC-OPT and QSLC-16b for brevity. For TSLC-OPT, the average energy reduction of the memory system (Mem-TSLC-OPT) and rest of the GPU (Rest-TSLC-OPT) is 10.1% and 7%, respectively, while it is 10.6% and 7% for QSLC-16b. The memory system has higher energy reduction compared to the rest of the GPU due to the direct affect of the reduced off-chip memory traffic, while the rest of the GPU gains from the overall performance improvement.

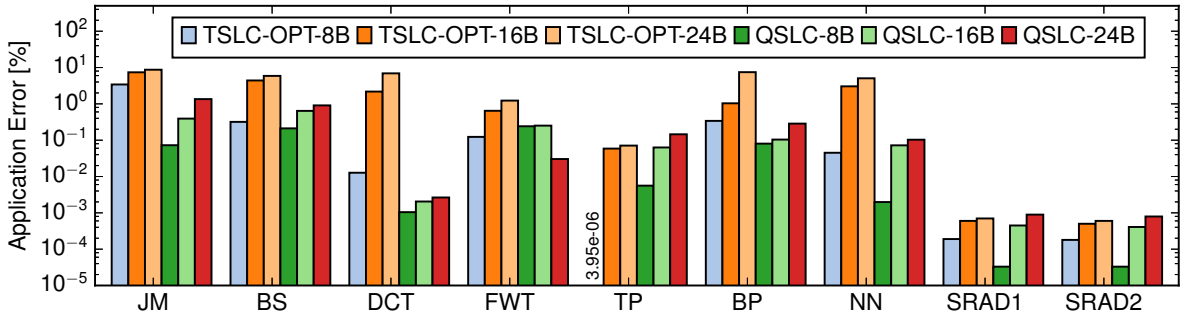
7.6 Sensitivity Analysis and Discussion

7.6.1 Performance and Accuracy Trade-off with Lossy Threshold

Lossy threshold is a knob to trade-off accuracy for performance. Figure 7.13 shows the change in speedup and error for lossy thresholds of 8B, 16B, and 24B. We only show results for TSLC-OPT and QSLC with 16-bit quantization as they have similar performance with a 16B threshold. The numbers in the legends of Figure 7.13 indicate the lossy threshold in bytes. The speedup and error increase with the increase in threshold which is expected as more number of blocks get compressed to a multiple of MAG. The results show that even for a small lossy threshold (8B), we can get an average speedup of 4% (and up to 13%) with $< 0.3\%$ loss in accuracy (and maximum of 3.4%) for TSLC-OPT and 3.8% (and up to 12%) for maximum of 0.24% loss in accuracy for QSLC. In general, QSLC has almost equal or better speedup at lower error compared to TSLC-OPT. For instance, for a lossy threshold of 24B, the average speedup is 15.5% (and up to 30%) with a maximum



(a) Speedup



(b) Error

Figure 7.13: Accuracy and performance trade-off.

of only 1.35% loss in accuracy for QSLC. For the same threshold, the average speedup is 12.5% (and up to 24%) with a maximum of only 8.7% loss in accuracy for TSLC-OPT. Thus, performance can be increased by increasing the lossy threshold, apparently at relatively higher error. However, as different applications have different error tolerance, an appropriate threshold can be selected depending upon the performance requirements and error tolerance.

7.6.2 SLC Sensitivity to MAG

Figure 7.14 shows the raw and effective compression ratio for different MAGs when E^{2}MC is used. The geometric mean (GM) of the effective compression ratio is 1.41, 1.31, 1.16 for MAG of 16B, 32B, and 64B, respectively. The effective compression ratio decreases with the increase in MAG because the probability decreases where a compressed block size can be an exact multiple of a MAG, thereby increasing the importance of SLC technique even more. Figure 7.14 shows only one bar for the raw compression ratio as it remains the same across different MAGs.

Figure 7.15 shows the speedup and error when SLC is used for different MAGs of 16B, 32B, and 64B. We only show the results for QSLC-16b for brevity. We set the lossy threshold to half of the corresponding MAG for this experiment as one threshold across

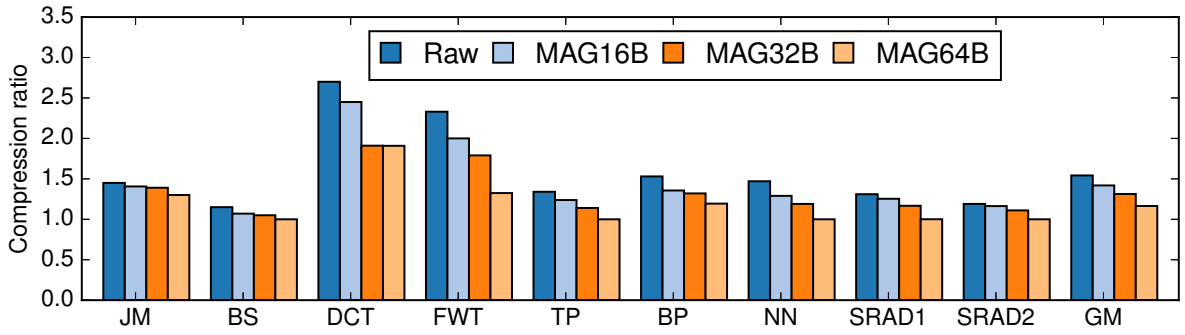
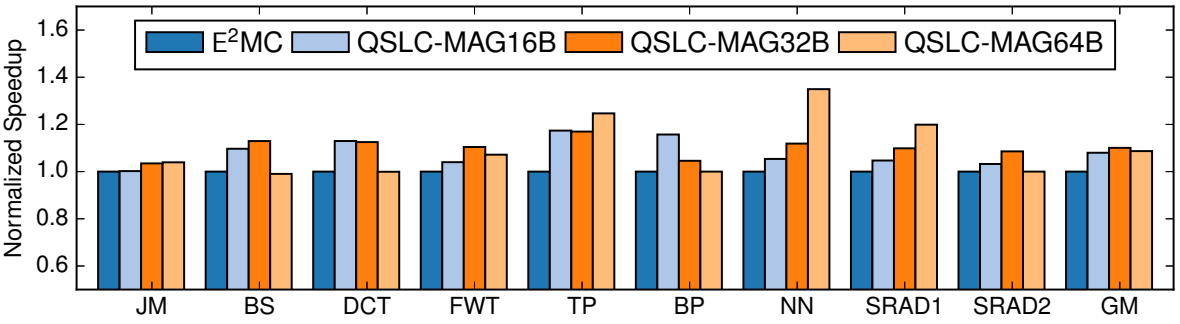
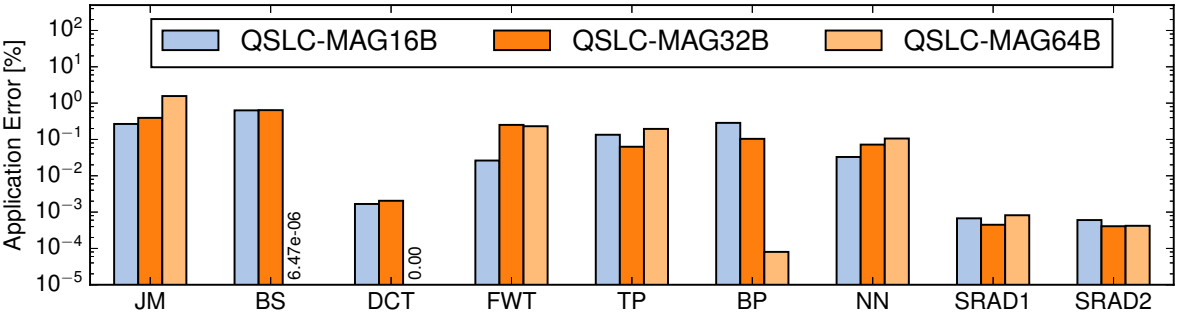


Figure 7.14: Raw and effective compression ratio for different MAGs for E^{2}MC.



(a) Speedup



(b) Error

Figure 7.15: Speedup and error for different MAGs when SLC is used.

different MAGs is not suitable. For example, a lossy threshold of 16B for MAG of 16B will always approximate blocks to a lower multiple of a MAG. Moreover, a 16B threshold might be small for a 64B MAG. Therefore, we set the lossy threshold to half of the corresponding MAG.

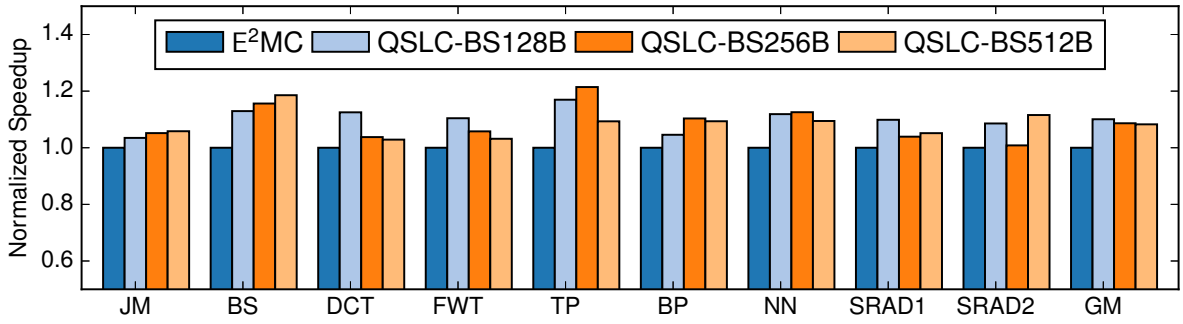
The geometric mean of the speedup is 8%, 10%, and 9% for MAG of 16B, 32B, and 64B, respectively. There are few interesting observations from the results. First, we observe that SLC provides speedup across different MAGs and we can achieve speedup as high as 35%. Second, we observe large variations in the speedup for 64B MAG compared to

16B and 32B. We have a high speedup of 35% for *NN*, 20% for *SRAD1*, 25% for *TP*, while there is no speedup for *BS, DCT, BP* for 64B MAG. This is mainly because there is low probability where the effective compression ratio can be greater than 1.0 ($\leq$ 64B) for MAG of 64B, while there is much higher probability for MAG of 32B and 16B. For example, for *NN, SRAD1, TP*, the losslessly compressed size of most of the blocks is above 64B and SLC exploits that to increase the effective ratio and performance gain. In contrast, the compressed size of most of the blocks is already below 64B for *DCT, BP* and above 96B for *BS*, thus, there is no opportunity for SLC to improve the performance further. Like speedup, we have higher variations in the error for 64B MAG, for example, relatively higher error for *NN*, *SRAD1*, and *TP*. *DCT* has zero error for 64B MAG because no block is approximated. The error ranges from zero to maximum of 1.8%. We observe that, on one hand, a larger MAG results in a higher difference between the raw and effective compression which presents an opportunity for SLC, on the other hand, a larger MAG also has lower probability where a compressed block can be a multiple of MAG.

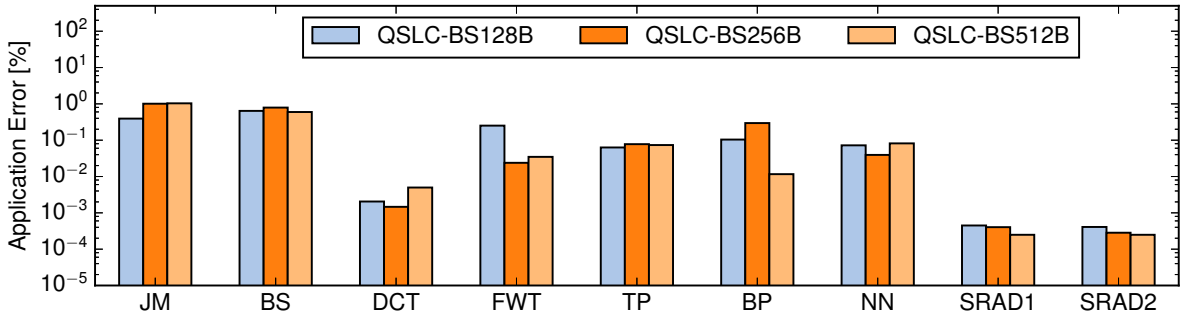
7.6.3 SLC Sensitivity to Block Size

To conduct sensitivity analysis to different block sizes, we perform experiments for the baseline E^{2}MC and SLC with block sizes of 128B, 256B, 512B using MAG of 32B. The GM of the raw compression ratio for E^{2}MC is 1.54, 1.57, 1.58 and the effective compression ratio is 1.31, 1.45, 1.47 for the block size of 128B, 256B, and 512B, respectively. We notice that for a larger block size the difference between the raw and effective compression ratio shrinks, leading to relatively higher compression ratio. The reason for the smaller difference with a larger block size is two-fold, first, there is a smaller number of blocks that need to be compressed and second, a larger block has a higher probability to be a multiple of MAG compared to a smaller block size. For example, assume there are two compressed blocks with a size of 36B each. This will result in four MAG fetches for a block size of 128B, while only 3 for a block size of 256B, assuming the same compressibility for the two consecutive smaller blocks and one larger block.

Our compression technique operates at the granularity of L2 cache line size as that is the size of memory requests processed by the memory controller. Thus, for conducting the block size sensitivity analysis, we increase L2 cache line size and reduce the number of sets to keep the L2 cache size the same for different block sizes. The compression and decompression latency is also adjusted accordingly to the block size. Figure 7.16 shows the change in speedup and error for different block sizes for SLC using 32B MAG. The average speedup slightly decreases due to the reduced gap between effective and raw compression ratio, however, the decrease in speedup is rather small. Moreover, for some benchmarks, the speedup is even higher compared to a smaller block size. This is likely caused by the change in cache configuration. For example, performance even without compression for the block size of 256B and 512B is on an average 15% and 70% slower compared to the block size of 128B. We also conduct experiments with 4$\times$ and 8$\times$ L2 cache size, however, the trend remains the same as it is not completely possible to isolate the effects of cache



(a) Speedup



(b) Error

Figure 7.16: Speedup and error for different block sizes for SLC using 32B MAG.

configuration change. In general, a bigger block size can reduce the difference between raw and effective compression ratio, however, as long as a large MAG exists we can use SLC to gain performance. Figure 7.16b shows that, in general, the error for a larger block size is slightly lower which is kind of expected due to less approximation.

7.6.4 SLC in the Presence of a Sectored L2 Cache

Traditionally, GPU memory hierarchy used coarse-grained memory accesses to exploit spatial locality, maximize peak bandwidth, simplify control, and reduce cache meta-data storage. However, coarse-grained memory accesses waste off-chip memory bandwidth and limit the energy efficiency of GPUs for irregular applications by over-fetching unnecessary data [136]. The size of a coarse-grained memory access is normally equal to a cache line size. To curtail the over-fetch, contemporary GPUs use sector caches, for example, NVIDIA introduced sectored L2 cache for Fermi architecture with a sector size of 32B. In a sector cache, a cache line consists of multiple sectors with provision to fetch individual sectors. For example, Fermi has 128B cache line size for L1 and L2 caches, however, it is possible to have 32B transactions when the L1 caching is disabled. Taking a step further, NVIDIA's Maxwell and Pascal architectures use sector cache for both L1 and L2 caches. The cache line size is still 128B, however, now it consists of four 32B sectors. Assuming

Table 7.5: Memory access granularity.

Generation	Bus Width	Burst Length	MAG	Mode
GDDR5	32b	8	32B	-
	32b	16	64B	QDR mode
GDDR5X	32b	8	32B	DDR mode (like GDDR5)
	2×16b	16	32B	2×32B (independent) in pseudo mode
GDDR6	16b	16	32B	-
HBM1	128b	2	32B	-
HBM2	128b	2	32B	Legacy mode
	2×64b	4	32B	2×32B (independent) pseudo channel mode

a coalesced access with each thread operating on 4B data, 128B are fetched from off-chip memory, apparently as 4 sectors. On an uncoalesced L1 or L2 miss or when each thread is operating on less than 4B data, only requested 32B sectors are loaded from L2 or off-chip memory, thus, reducing waste of memory bandwidth. The point here is that even with a sectored L2 cache, memory requests mostly consist of multiple sectors and a lossless memory compression technique reduces the number of 32B sectors fetched from memory by compressing them into fewer sectors. However, when compressed size is not an exact multiple of sectors, whole last sector is fetched even when only a few bytes are actually needed. The goal of our proposed SLC technique is to selectively approximate such memory requests to save a whole 32B sector fetch. Thus, even in the presence of sector caches, SLC saves 32B transactions from memory. An optimization that can be incorporated in SLC is the bypassing of memory requests whose size is only one sector as in this case it is not possible to fetch less than a sector from off-chip memory. We assume a sector size is equal to MAG.

7.6.5 SLC and 3D Stacked DRAM

High bandwidth provided by Graphics Double Data Rate (GDDR) memories has been a key enabler of the continuous performance scaling of GPUs. Successive generations of GDDR memories have increased overall memory bandwidth primarily by using wider memory interfaces and increasing frequency of off-chip signaling. However, further scaling of GDDR bandwidth is not possible without adding to system energy. Recently, 3D stacked DRAM technologies such as Hybrid Memory Cubes (HMC) and High Bandwidth Memory (HBM, HBM2) have been introduced, offering higher bandwidth and energy efficiency in a small form factor compared to GDDR. However, future GPUs will demand even higher (multiple TB/s) DRAM bandwidth requiring further improvements in the bandwidth. Memory bandwidth compression techniques including SLC, E^2MC are orthogonal to these technological improvements and can be employed on top of them to meet the

bandwidth and energy efficiency requirements of future high-throughput accelerators.

Table 7.5 shows the MAG for contemporary GDDR and HBM. Table 7.5 shows that 32B is the most widely supported MAG. Only GDDR5X provided 64B MAG, however, it also supports 32B MAG as well. GDDR6 again fully supports 32B MAG. The reasons for keeping the 32B MAG seems to be that some parts of the micro-architecture have been optimized for it and keeping the same access granularity enables GPU architectures that are optimized for one GDDR generation to transition to next generation with minimal effort. With HBM also supporting 32B MAG shows that 32B MAG is going to be the most widely used access granularity even in near future. However, SLC is not tied to any MAG and its significance increases with the increase in MAG size.

7.7 Summary

In this chapter, we showed that lossless memory compression techniques often exhibit a low effective compression ratio due to large MAG. We studied the distribution of compressed blocks and observed that a significant percentage of blocks are compressed to a size that is only a few bytes above a multiple of MAG, but a whole burst is fetched from memory, leading to low effective compression ratio and reduced performance gain. With the goal to increase the effective compression ratio and keep the approximation error low, we proposed a novel MAG aware Selective Lossy Compression (SLC) technique for GPUs. SLC appropriately selected between lossless and lossy compression modes, mostly retained the quality of a lossless compression and intelligently traded small accuracy for higher performance. We proposed TSLC and QSLC to implement SLC and compared their advantages and disadvantages. For a lossy threshold of 16B and 32 MAG, TSLC-OPT resulted in an average speedup of about 10% (up to 17%) normalized to a state-of-the-art lossless compression technique with an average error of $< 1\%$, while QSLC with 16-bit quantization also achieved an average speedup of 10% with an average error of only 0.2% (maximum error of 0.64%). Energy consumption and EDP are reduced by 8.3% and 17.5% using TSLC and 8.4% and 18.2% using QSLC, respectively. Furthermore, an energy breakdown analysis showed memory system has higher energy reduction compared to the rest of the GPU due to the direct affect of the reduced off-chip memory traffic.

We also conducted sensitivity analysis to different MAGs and showed an even higher significance of SLC at larger MAG. For 64B MAG, we showed a speedup of up to 35% with maximum error of 1.8%. In general we observed that, on the one hand, a larger MAG results in a higher difference between the raw and effective compression which presents an opportunity for SLC, on the other hand, a larger MAG also has less probability where a compressed block can be a multiple of MAG. We also conducted sensitivity analysis to different block sizes and showed that a bigger block size can reduce the difference between raw and effective compression ratio, however, as long as a large MAG exists we can use SLC to gain performance.

We estimated the area and power overhead of SLC and showed that it is feasible and very low with respect to GTX580. For example, the area and power cost of the TSLC is

about 0.0015% and 0.0008% of the total area and power consumption of GTX580, while the area and power cost of the QSLC-12b is about 0.0013% and 0.0007% of GTX580, respectively. TSLC added 5.6% of the area of E^2MC, while QSLC-12b and QSLC-16b only added 4.7% and 0.5% of the area of E^2MC, respectively.

In the next chapter, we will summarize the key contributions of the thesis and draw conclusions. We will also present possible directions to extend the work in the future.

8 Conclusions and Future Work

In this chapter, we summarize the contributions of each chapter, conclude the thesis and suggest possible directions for future work.

8.1 Summary of Contributions

This thesis addressed GPU power modeling and energy efficiency problems from different perspectives. In Chapter 1, we stated motivation, articulated objectives and briefly described the main contributions of the thesis. In Chapter 2, we thoroughly reviewed the work related to this thesis and presented advances over the state of the art. In Chapter 3, we provided an overview of a GPU architecture including programming models and simulators. We also presented an overview of data compression. In particular, we described state-of-the-art memory compression techniques that were used as baselines in Chapter 6 and Chapter 7.

In Chapter 4, we designed GPUSimPow, a power simulator for GPU architectures. GPUSimPow is a highly parameterizable simulator that is able to provide an accurate estimate of area, static power as well as dynamic power consumption of GPGPU workloads running on GPUs. The simulator not only provides an estimate of the total power consumption but also provides an estimate of the power consumption down to the individual component level. To develop GPUSimPow, we integrated a modified version of gpgpu-sim, a cycle-accurate architectural simulator for GPUs and a heavily modified McPAT, a CPU power simulator. Although McPAT includes detailed power models for several microarchitectural components, many GPU components were either not present or considerably different compared to CPUs. Therefore, we added and adapted several important components in McPAT to more accurately represent the underlying GPU microarchitecture. We used a hybrid power modeling approach improving flexibility compared to pure empirical approaches and accuracy compared to pure analytical approaches. We also developed a custom power measurement testbed to validate the power simulator and derive measurement based power models for some components.

In summary, we made the following key contributions in Chapter 4:

- We developed a GPU power simulator that is able to provide an accurate estimate of the area, static power as well as dynamic power for GPU architectures. Our evaluation of the power simulator on a set of well-known benchmarks showed an average relative error of 11.7% and 10.8% between simulated and measured power for GT240 and GTX580, respectively.

- We used a hybrid power modeling approach which provides both flexibility and high accuracy to conduct architectural research from power and energy perspective.
- We designed a novel power measurement testbed to validate the power simulator by accurately measuring GPU power consumption.

In Chapter 5, we used GPUSimPow to study the energy efficiency of a wide range of kernels. We divided the kernels into high performance and low performance categories. The kernels with IPC greater than 50.5% of the peak IPC were classified as high performance, otherwise low performance. We showed that 69% (47 out of 68 kernels) of the kernels have low performance and low energy efficiency. The average IPC of the low-performance category kernels is less than 25% of the peak IPC and average energy per instruction is 7.5× less than the average energy per instruction of the high-performance category kernels. We investigated the bottlenecks that lead to low performance and low energy efficiency by dividing the low performance kernels into two categories: low occupancy (15 kernels) and full occupancy (32 kernels). For the low occupancy category, we showed a significant gain in performance and energy efficiency when occupancy is increased. At full occupancy, the average increase in IPC, the average reduction in energy consumption and EDP is 11%, 9%, and 23%, respectively, for the CTA limited kernels. The average increase in IPC, the average reduction in energy consumption and EDP is 15%, 9%, and 21%, respectively, for the registers limited kernels. The results showed that high occupancy is an important factor for both high performance and energy efficiency. For the kernels having full occupancy but still performing low, we showed that these kernels are either limited by memory bandwidth, low coalescing efficiency or low SIMD utilization. Moreover, we showed a variation in the distribution of power consumption across the two categories. For example, for kernels in the high-performance category, EU (25.3%), WCU (20.3%), and CP (16.0%) are the three most power consuming components and together consume about 62% of the power. For kernels in the low-performance category, GM (21.4%), CP (19.3%), and WCU (16.6%) are the three most power consuming components. Furthermore, we also investigated the correlation between components power consumption and workload metrics and showed the existence of the correlation.

In summary, we made the following contributions in Chapter 5:

- We studied the energy efficiency of a wide range of kernels and exposed that most kernels have low performance and low energy efficiency.
- We studied GPU power consumption at the component level for the high performance and low performance categories and reported the most power consuming components. We also investigated the correlation between components power consumption and workload metrics and showed the existence of the same.
- We showed that increasing occupancy does help to increase performance and energy efficiency of low occupancy category, however, occupancy alone is not sufficient to achieve the desired performance.

- We also analyzed kernels having full occupancy but still performing low and showed that several of these kernels are limited by memory bandwidth, low coalescing efficiency or low SIMD utilization. We further showed that full occupancy kernels can benefit from increased memory bandwidth and improved coalescing efficiency.

One of the main conclusions from the performance bottlenecks study in Chapter 5 is that despite GPU possessing much higher memory bandwidth compared to CPU, several kernels are limited by memory bandwidth and optimizations in the memory hierarchy are needed to drive the performance and energy efficiency further. Moreover, the kernels with low coalescing efficiency will also gain from the memory bandwidth optimizations because low coalescing efficiency exerts high pressure on the memory subsystem by injecting several memory requests per warp. Therefore, in Chapter 6, we proposed a memory compression technique to increase the effective memory bandwidth of GPUs. We observed that the existing memory compression techniques for GPUs (FPC, C-Pack, BDI) exploit simple patterns for compression and trade low compression ratio for low decompression latency. As GPUs can hide latency to a certain extent, we proposed the more aggressive Entropy Encoding Based Memory Compression (E^2MC) technique for GPUs. We addressed the key challenges of probability estimation, appropriate symbol length for encoding, and reasonably low decompression latency. One of the drawbacks of entropy-based compression techniques is that they may require online sampling to estimate the frequency of symbols if entropy characteristics are not known in advance. Fortunately, our experiments showed that it is possible to achieve a compression ratio comparable to the one achieved by offline probability with a very short online sampling phase at the start of the benchmarks. We experimented with different symbol lengths and showed that 16-bit symbols yield the highest compression ratios for GPUs. Furthermore, to reduce the decompression latency, we used parallel decoding to decode multiple codewords in parallel. Although parallel decoding reduced the compression ratio due to additional metadata, we showed that the loss in compression ratio is not much as it is mostly hidden by the memory access granularity (MAG). We also estimated the area and power needed to meet the high throughput requirements of GPUs.

In summary, we made the following contributions in Chapter 6:

- We proposed an entropy encoding based memory compression technique for GPUs that delivered higher compression ratio and performance gain than state-of-the-art techniques.
- We addressed the key challenges of probability estimation, appropriate symbol length for encoding, and low decompression latency.
- E^2MC with offline sampling resulted in 53% higher compression ratio and 8% increase in speedup compared to the state of the art and saved 13% energy and 27% EDP compared to no compression. Online sampling resulted in 35% higher compression ratio and 5% increase in speedup compared to the state of the art and saved 11% energy and 24% EDP compared to no compression.

- We analyzed the high throughput requirements of GPUs and provided an estimate of area and power needed to support such high throughput. For E^2MC16, the area and power overhead is 5.8% and 1.5% of the area and power of GTX580, respectively. We think the area numbers are likely higher than expected because the available memory design library does not have exact components needed to design the (de)compressor and we have to combine smaller components to get the required size. We think that a custom design will be denser and will need less area.

- We also provided a detailed analysis of the effects of memory access granularity on the compression ratio.

In Chapter 6, we observed that lossless memory compression techniques often result in a low effective compression ratio due to the large memory access granularity (MAG) exhibited by GPUs. In Chapter 7, we further analyzed the reasons for the low effective compression ratio by quantitatively studying the distribution of compressed blocks above MAG. We showed that a significant percentage of blocks are compressed to a size that is only a few bytes above a multiple of MAG, but due to the restrictions of MAG, a whole burst is fetched from memory. With the goal to reduce the compressed size by these extra bytes, we proposed the novel MAG aware Selective Lossy Compression (SLC) technique for GPUs which uses approximation techniques to increase the effective compression ratio and drive the performance and energy efficiency further. SLC appropriately selects between lossless and lossy compression modes, mostly retains the quality of a lossless compression and intelligently trades small accuracy for higher performance. We proposed TSLC and QSLC to implement SLC and presented their trade-offs. We conducted sensitivity analysis to different MAGs and showed an even higher significance of SLC at larger MAG. We concluded that, on one hand, a larger MAG results in a higher difference between the raw and effective compression which is an opportunity for SLC, on the other hand, a larger MAG also has less probability where a compressed block can be a multiple of MAG. We also conducted sensitivity analysis to different block sizes and showed that a bigger block size can reduce the difference between raw and effective compression ratio, however, as long as a large MAG exists we can use SLC to gain performance.

In summary, we made the following contributions in Chapter 7:

- We analyzed reasons for the low effective compression ratio of several state-of-the-art memory compression techniques and quantitatively showed that the low effective compression ratio due to large MAG exists in four techniques and qualitatively showed in three more.

- We proposed a novel MAG aware selective lossy compression technique for GPUs and designed two techniques to implement SLC and presented their trade-offs. For a lossy threshold of 16B and 32 MAG, TSLC resulted in an average speedup of 10% (up to 17%) normalized to a state-of-the-art lossless compression technique with an average error of $< 1\%$, while QSLC with 16-bit quantization also achieved an

average speedup of 10% with an average error of only 0.2%. Energy consumption and EDP are reduced by 8.3% and 17.5% using TSLC and 8.4% and 18.2% using QSLC with 16-bit quantization, respectively

- This is the first study that highlights the importance of MAG aware compression by quantitatively studying the distribution of compressed blocks above MAG.

- We also conducted a sensitivity analysis to different MAGs and showed an even higher significance of SLC at a larger MAG. For 64B MAG, we achieved a speedup of up to 35% with a maximum error of 1.8%.

- We implemented hardware and provided an estimate of the area and power overhead of SLC compared to GTX580. The area and power cost of the TSLC is about 0.0015% and 0.0008% of the total area and power consumption of GTX580, while the area and power cost of the QSLC-12b is about 0.0013% and 0.0007% of GTX580, respectively. TSLC adds 5.6% of the area of E^{2}MC, while QSLC-12b and QSLC-16b only add 4.7% and 0.5% of the area of E^{2}MC, respectively.

8.2 Conclusions

The tremendous computing power of GPUs have made them popular accelerators but their high power consumption and low energy efficiency under low utilization is a challenge. Indeed, with the end of Denard scaling and energy efficiency taking over as the main design metric, specialized hardware accelerators such as GPUs are key to achieve higher energy efficiency by mapping different parts of an application to the best-suited accelerator. The high-performance demands have influenced the design of GPUs to be optimized for higher performance per watt, even at the cost of large power consumption. Not all applications, however, can utilize all available resources due to various bottlenecks such as data dependencies, low occupancy, high bandwidth requirements, thus, reducing the achieved performance per watt.

This thesis investigated bottlenecks causing low performance and low energy efficiency in GPUs and then proposed architectural techniques that significantly improved the performance and energy efficiency. To enable the energy efficiency research for GPUs, we first developed a flexible and accurate power simulator. The power simulator (presented in Chapter 4) in its current state is a very useful tool for GPU architects to explore the design space from a power perspective and GPU programmers to gain valuable insights into where power is consumed from a software perspective. The component level power profiling capabilities of the power simulator demonstrated its usability not only for estimating total power consumption but also down to the detailed individual components. As the power breakdown revealed, however, a large fraction of the estimated power is currently attributed to components that are not modeled in detail, named “undifferentiated core” in the thesis, due to the lack of the available information. More research needs to

be done for creating accurate models of these components to further increase the usability and accuracy of the simulator.

The energy efficiency study in Chapter 5 revealed that a large number of GPU kernels have low performance and low energy efficiency. The average energy per instruction for the high performance and low performance category is 0.27 nJ and 2.01 nJ, respectively. The later is 7.5× less energy efficient compared to the former, a huge difference which is not favorable for the future growth of high performance computing and far away from the exascale aim of 10 pJ per instruction. The bottlenecks investigation revealed that there are several bottlenecks such as low occupancy, high memory bandwidth utilization, low coalescing efficiency that lead to low performance and energy efficiency. We conclude that increasing the occupancy helps in increasing the performance and energy efficiency for most of the kernels, but just increasing occupancy is not enough to achieve the maximum performance. For example, the average increase in instructions per cycle, the average reduction in energy consumption and energy-delay-product is 11%, 9%, and 23%, respectively, for a sub-category of low occupancy kernels when its occupancy is increased. This work also showed that many kernels despite having full occupancy have low performance. Further investigation revealed that several of these kernels are memory-bound and can gain significantly from the increase in memory bandwidth.

To provide high bandwidth, traditionally, GPUs deploy GDDR memory which has been a key enabler of the continuous bandwidth and performance scaling, however, the later generations of GDDR have issues such as high power consumption, large form factor, and difficulty in the scaling of pin count, thereby limiting the traditional scaling of GDDR bandwidth. The latter part of the thesis showed that memory compression is a promising alternative to traditional ways of increasing memory bandwidth. Alternative ways of increasing memory bandwidth will play a significant role in scaling the performance and energy efficiency of GPUs, especially based on the evidence that further scaling of GDDR bandwidth is not possible without adding significantly to system energy. Based on the observation that GPUs are less sensitive to latency compared to CPUs, this thesis proposed an entropy encoding based memory compression technique (E^2MC) for GPUs which can exploit relatively complex patterns compared to existing techniques for GPUs (presented in Chapter 6). We showed that entropy encoding based memory compression technique delivers higher compression ratio and performance gain than state-of-the-art techniques, provided the key challenges of probability estimation, appropriate symbol length for encoding, and reasonably low decompression latency are addressed properly. On average, E^2MC delivered 53% higher compression ratio and 8% higher speedup than the state of the art and reduced energy consumption and energy-delay-product by 13% and 27%, respectively, compared to no compression. We also observed that lossless memory compression techniques often result in a low effective compression ratio due to the large memory access granularity (MAG) exhibited by GPUs.

In Chapter 7, we explored the use of approximate computing techniques to mitigate the MAG problem. Our analysis of the distribution of compressed blocks showed that a significant percentage of blocks are compressed to a size that is only a few bytes above

a multiple of MAG, leading to low effective compression ratio. To increase the effective compression ratio, we proposed a novel MAG aware selective lossy compression (SLC) technique for GPUs. SLC delivered a significant gain in performance (up to 17%) normalized to a state-of-the-art lossless compression technique with a maximum error of 0.64% (< 1% on average) and reduced energy consumption and EDP by 8.4% and 18.2%, respectively. The sensitivity analysis to different MAGs showed an even higher significance of SLC at a larger MAG. For 64B MAG, we achieved a speedup of up to 35% with a maximum error of 1.8%. Our work is the first detailed study that illustrated the effects of the large MAG on effective compression ratio and demonstrated the importance of MAG aware approximation techniques. We think that this work will spark more research in this direction. A disadvantage of SLC is that it introduces approximation errors, however, not all applications are error-tolerant. Therefore, we designed SLC in such a way that it can be easily toggled on top of a lossless compression technique depending on an application performance requirements and error tolerance.

Although this thesis focused on GPUs, some of the techniques presented are generic and applicable to CPUs as well. The memory compression technique presented in Chapter 6 and MAG aware approximation presented in Chapter 7 are equally important for CPUs and can also be applied to them.

8.3 Future Work

This thesis contributed significantly to GPU power modeling, bottlenecks investigation and alternative techniques to increase effective memory bandwidth, leading to improved performance and energy efficiency. However, there are several possible directions to extend the research further.

- First, the GPU power simulator can be improved by adding more detailed power models for some components. The power simulator reports power distribution over different components of the GPU, however, as power breakdown revealed, a large fraction of the estimated power is currently attributed to components that are not modeled in detail, called "undifferentiated core" in the thesis, due to the lack of the available information. Further research needs to be done for creating accurate models of these components. Moreover, the simulator was tested using real power measurements for NVIDIA GT240 and NVIDIA GTX580 which are based on Tesla and Fermi architectures. The simulator can be extended with new components to accurately model the architecture of newer GPUs such as Maxwell, Pascal, Volta.

- Second, GPU performance bottlenecks analysis showed that low coalescing efficiency can severely limit the performance and energy efficiency of GPUs. Further research is needed to ascertain the exact bottlenecks that low coalescing efficiency creates at different levels in memory hierarchy and any opportunities for micro-architectural changes that could benefit the kernels suffering from very low coalescing efficiency

can be explored. In this regard, micro-benchmarking can be used to pinpoint the bottlenecks. Also, architectural techniques such as dynamic granularity memory system which supports coarse-grained and fine-grained memory accesses could be explored to improve the performance.

- Third, E^{2}MC used single encoding at the beginning of a benchmark. This assumes that compressibility of the data does not change over the execution of an application which is not always true. Thus, it would be interesting to study the effect of multiple encodings on compression in the future. Furthermore, we showed that E^{2}MC is feasible for off-chip memory compression, however, caches have tighter latency requirements than off-chip memory. It would also be intriguing to see if E^{2}MC can be extended to other levels of the memory hierarchy.

- Fourth, SLC helped to tackle the MAG problem, but it introduced approximation errors. There are several applications, for instance, graph-based applications which cannot tolerate even small errors, but they also suffer due to MAG. Further possibilities to reconstruct the approximated bytes losslessly or determining when these extra bytes are not required at all will significantly increase the value of the proposed technique. For instance, even for a memory divergence benchmark, a full cache line is fetched, however, mostly a full cache line is not used. Furthermore, the possibilities of adopting existing lossless memory compression techniques to MAG size can be investigated.

- MAG also offers other opportunities for optimizations such as MAG aware error correcting code (ECC) support. ECC provides higher reliability for applications that are sensitive to data corruption, however, ECC bits use bandwidth otherwise available to user data. As we have seen that losslessly compressed size is not always an exact multiple of MAG and extra bytes are fetched as data can only be fetched in the multiple of MAG. ECC overhead can be reduced by piggybacking ECC bits along with compressed data.

Bibliography

[1] International Technology Roadmap for Semiconductors, ITRS, 2011. http://www.itrs.net/.

[2] The Green 500, 2018. www.green500.org.

[3] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith. Memory Expansion Technology (MXT): Software Support and Performance. *IBM Journal of Research and Development*, 45(2):287–301, 2001.

[4] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato. Power and Performance Analysis of GPU-accelerated Systems. In *Proceedings of the USENIX Conference on Power-Aware Computing and Systems, HotPower*, 2012.

[5] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing, HPDC*, 2008.

[6] A. Alameldeen and D. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. In *Technical Report, University of Wisconsin-Madison*, 2004.

[7] R. Alur, J. Devietti, O. S. Navarro Leija, and N. Singhania. GPUDrano: Detecting Uncoalesced Accesses in GPU Programs. In *Proceedings of the International Conference on Computer Aided Verification, CAV*, 2017.

[8] AMD. AMD Graphics Core Next GCN Architecture White Paper, 2012. https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[9] A. Arelakis, F. Dahlgren, and P. Stenstrom. HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods. In *Proceedings of the 48th International Symposium on Microarchitecture, MICRO*, 2015.

[10] A. Arelakis and P. Stenstrom. SC^2: A Statistical Compression Cache Scheme. In *Proceddings of 41st International Symposium on Computer Architecture, ISCA*, 2014.

[11] ARM. ARM Frame Buffer Compression Protocol. https://www.arm.com/products/multimedia/mali-technologies/arm-frame-buffer-compression.php.

[12] J. M Arnau, J. M Parcerisa, and P. Xekalakis. Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization. In *Proceedings of the 41st International Symposium on Computer Architecture, ISCA*, 2014.

[13] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu. Mosaic: A GPU Memory Manager with Application-transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2017.

[14] W. Baek and T. M. Chilimbi. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2010.

[15] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2009.

[16] R. Balasubramanian, V. Gangadhar, Z. Guo, C. H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam. MIAOW - An Open Source RTL Implementation of a GPGPU. In *IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*, 2015.

[17] E. Blem, M. Sinclair, and K. Sankaralingam. Challenge Benchmarks that Must be Conquered to Sustain the GPU Revolution. In *Proceedings of the 4th Workshop on Emerging Applications for Manycore Architecture*, 2011.

[18] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[19] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[20] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture, ISCA*, 2000.

[21] 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 SIGGRAPH Papers, SIGGRAPH*, 2004.

[22] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC*, 2012.

[23] D. Cederman, B. Chatterjee, and P. Tsigas. Understanding the Performance of Concurrent Data Structures on Graphics Processors. In *Proceedings of the 18th International Conference on Parallel Processing*, 2012.

[24] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau. CAVA: Using Checkpoint-assisted Value Prediction to Hide L2 Misses. *ACM Transactions on Architecture and Code Optimization*, 3(2):182–208, 2006.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC*, 2009.

[26] X. Chen, L. Yang, R.P. Dick, L. Shang, and H. Lekatsas. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. *IEEE Transactions on VLSI Systems*, 18(8):1196–1208, 2010.

[27] E. Choukse, M. Erez, and A. R. Alameldeen. Compresso: Pragmatic Main Memory Compression. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2018.

[28] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2010.

[29] D. Citron. Exploiting Low Entropy to Reduce Wire Delay. *IEEE Computer Architecture Letters*, 3(01):1, 2004.

[30] D. Citron, D. Feitelson, and L. Rudolph. Accelerating Multi-media Processing by Implementing Memoing in Multiplication and Division Units. In *Proceedings of the 18th*

International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 1998.

[31] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[32] 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, 2009.

[33] 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, 2008.

[34] 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, 2012.

[35] A. C. Crespo, J. M. Dominguez, A. Barreiro, M. Gómez-Gesteira, and B. D. Rogers. GPUs, a New Tool of Acceleration in CFD: Efficiency and Reliability on Smoothed Particle Hydrodynamics Methods. *PLOS ONE*, 6:1–13, 2011.

[36] B. Dally. Power, Programmability, and Granularity: The Challenges of ExaScale Computing. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2011.

[37] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[38] D. De Caro, N. Petra, and A. G. M. 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.

[39] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[40] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[41] R. J. Eickemeyer and S. Vassiliadis. A Load-instruction Unit for Pipelined Processors. *IBM Journal of Research and Development*, 37(4):547–564, 1993.

[42] M. Ekman and P. Stenstrom. A Robust Main-Memory Compression Scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture, ISCA*, 2005.

[43] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA*, 2011.

[44] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2012.

[45] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2012.

[46] W. Fang, B. He, Q. Luo, and N. K. Govindaraju. Mars: Accelerating MapReduce with Graphics Processors. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):608–620, 2011.

[47] N. Fauzia, L. N. Pouchet, and P. Sadayappan. Characterizing and Enhancing Global Memory Data Coalescing on GPUs. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO*, 2015.

[48] S. H. Fuller and L. I. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011.

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

[50] W. W. L. Fung, I Sham, G. Yuan, and T. M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2007.

[51] S. Galal and M. Horowitz. Energy-Efficient Floating-Point Unit Design. *IEEE Transactions on Computers*, 60(7):913–922, 2011.

[52] C. Galuzzi, C. Gou, H. Calderón, G. N. Gaydadjiev, and S. Vassiliadis. High-bandwidth Address Generation Unit. *Journal of Signal Processing Systems*, 57(1):33–44, 2009.

[53] 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 *Proceedings of the 38th Annual International Symposium on Computer Architecture, ISCA*, 2011.

[54] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture, HPCA*, 2001.

[55] L. A. B. Gomez and F. Cappello. Improving Floating Point Compression Through Binary Masks. In *IEEE International Conference on Big Data*, 2013.

[56] X. Gong, R. Ubal, and D. Kaeli. Multi2Sim Kepler: A Detailed Architectural GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2017.

[57] N. Goswami, R. Shankar, M. Joshi, and T. Li. Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications. In *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC*, 2010.

[58] C. Gou and G. N. Gaydadjiev. Addressing GPU On-Chip Shared Memory Bank Conflicts Using Elastic Pipeline. *International Journal of Parallel Programming*, 41(3):400–429, 2013.

[59] E. G. Hallnor and S. K. Reinhardt. A Unified Compressed Memory Hierarchy. In *11th International Symposium on High-Performance Computer Architecture, HPCA*, 2005.

[60] P. Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *High Performance Computing, HiPC*, 2007.

[61] S. Hong and H. Kim. An Integrated GPU Power and Performance Model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture, ISCA*, 2010.

[62] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K. H. Kim, and M. B. Healy. Attache: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2018.

[63] S. Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2011.

[64] S. Huang, S. Xiao, and W. Feng. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In *Proceedings of the IEEE International Symposium on Parallel&Distributed Processing, IPDPS*, 2009.

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

[66] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and Improving the Use of Demand-fetched Caches in GPUs. In *Proceedings of the 26th ACM International Conference on Supercomputing, ICS*, 2012.

[67] N. Jing, S. Chen, S. Jiang, L. Jiang, C. Li, and X. Liang. Bank Stealing for Conflict Mitigation in GPGPU Register File. In *IEEE/ACM International Symposium on Low Power Electronics and Design, ISLPED*, 2015.

[68] B. Justice. NVIDIA Maxwell GPU GeForce GTX 980 Video Card Review. http://goo.gl/y2i5Tm.

[69] B. Juurlink, J. Lucas, N. Mammeri, M. Bliss, G. Keramidas, C. Kokkala, and A. Richards. The LPGPU2 Project: Low-Power Parallel Computing on GPUs: Extended Abstract. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPES*, 2017.

[70] B. Juurlink, J. Lucas, N. Mammeri, G. Keramidas, K. Pontzolkova, I. Aransay, C. Kokkala, M. Bliss, and A. Richards. Enabling GPU Software Developers to Optimize their Applications - The LPGPU^2 Approach. In *Proceedings of the International Conference on Design and Architectures for Signal and Image Processing, DASIP*, 2017.

[71] A. B. Kahng, B. Li, L. S. Peh, and K. Samadi. ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration. In *Design, Automation Test in Europe Conference Exhibition, DATE*, 2009.

[72] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[73] A. Kerr, G. Diamos, and S. Yalamanchili. A Characterization and Analysis of PTX Kernels. In *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC*, 2009.

[74] Khronos Group. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems, 2009. http://www.khronos.org/opencl/.

[75] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: An Online Quality Management System for Approximate Computing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA*, 2015.

[76] H. Kim, H. Nam, W. Jung, and J. Lee. Performance Analysis of CNN Frameworks for GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2017.

[77] J. Kim, M. Sullivan, E. Choukse, and M. Erez. Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA*, 2016.

[78] J. Kim, M. Sullivan, S. L. Gong, and M. Erez. Frugal ECC: Efficient and Versatile Memory Error Protection Through Fine-grained Compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2015.

[79] S. Kim, S. Lee, T. Kim, and J. Huh. Transparent Dual Memory Compression Architecture. In *26th International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2017.

[80] D. Kirk and W.-M. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2nd Edition, Chapter 1, Page 4, 2012.

[81] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A Suite of Parallel Irregular Programs. In *International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2009.

[82] 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.

[83] S. Lal and B. Juurlink. A Case for Memory Access Granularity Aware Selective Lossy Compression for GPUs. In *ACM Student Research Competition Poster and Extended Abstract at IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2018. https://www.microarch.org/micro51/SRC/posters/14_lal.pdf.

[84] S. Lal, J. Lucas, M. Andersch, M. Alvarez-Mesa, A Elhossini, and B. Juurlink. GPGPU Workload Characteristics and Performance Analysis. In *Proceedings of the 14th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS*, 2014.

[85] S. Lal, J. Lucas, and B. Juurlink. E^2MC: Entropy Encoding Based Memory Compression for GPUs. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2017.

[86] S. Lal, J. Lucas, and B. Juurlink. SLC: Memory Access Granularity Aware Lossy Compression for GPUs. In *Proceedings of the 14th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, ACACES*, 2018.

[87] S. Lal, J. Lucas, and B. Juurlink. SLC: Memory Access Granularity Aware Selective Lossy Compression for GPUs. In *Proceedings of the International Conference on Design Automation, and Test in Europe, DATE*, 2019.

[88] S. Lal, J. Lucas, M. A. Mesa, A. Elhossini, and B. Juurlink. Exploring GPGPUs Workload Characteristics and Power Consumption. In *Proceedings of the 9th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, ACACES*, 2013.

[89] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram. Warped-compression: Enabling Power Efficient GPUs Through Register Compression. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA*, 2015.

[90] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. Sung Kim, T. M. Aamodt, and V. J. Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA*, 2013.

[91] A. Li, S. L. Song, M. Wijtvliet, A. Kumar, and H. Corporaal. SFU-Driven Transparent Approximation Acceleration on GPUs. In *Proceedings of the International Conference on Supercomputing, ICS*, 2016.

[92] 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.

[93] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng. Performance Analysis of GPU-Based Convolutional Neural Networks. In *45th International Conference on Parallel Processing, ICPP*, 2016.

[94] Y. Li, X. Dai, F. Zhao, and H. Shang. GPU-based Acceleration for Monte Carlo Ray-Tracing of Complex 3D Scene. In *IEEE International Geoscience and Remote Sensing Symposium*, 2012.

[95] J. E. Lindholm, M. Y. Siu, S. S. Moy, S. Liu, and J. R. Nickolls. Simulating Multiported Memories Using Lower Port Count Memories, 2008.

[96] J. Liu, W. Ding, O. Jang, and M. Kandemir. Data Layout Optimization for GPGPU Architectures. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, 2013.

[97] R. S. Liu, C. L. Yang, and W. Wu. Optimizing NAND Flash-based SSDs via Retention Relaxation. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST*, 2012.

[98] Q. Lu and J. Amundson. Synergia CUDA: GPU-accelerated Accelerator Modeling Package. *Journal of Physics: Conference Series*, 513(5):052021, 2014.

[99] 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 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2013.

[100] 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.

[101] S. Madougou, A. L. Varbanescu, C. D. Laat, and R. V. Nieuwpoort. A Tool for Bottleneck Analysis and Performance Prediction for GPU-Accelerated Applications. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW*, 2016.

[102] D. Maier, B. Cosenza, and B. Juurlink. Local Memory-Aware Kernel Perforation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO*, 2018.

[103] M. Mao, W. Wen, Y. Zhang, Y. Chen, and H. Li. Exploration of GPGPU Register File Architecture Using Domain-wall-shift-write Based Racetrack Memory. In *Proceedings of the 51st ACM/EDAC/IEEE Design Automation Conference, DAC*, 2014.

[104] M. D. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule. Shader Algebra. *ACM Transactions on Graphics*, 23(3):787–795, 2004.

[105] X. Mei and X. Chu. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2017.

[106] J. Meng, D. Tarjan, and K. Skadron. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *Proceedings of the 37th annual international symposium on Computer architecture, ISCA*, 2010.

[107] Micron. Calculating Memory System Power for DDR3, 2007. http://http://www.micron.com/products/dram/ddr3-sdram.

[108] J. S. Miguel, M. Badr, and N. E. Jerger. Load Value Approximation. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2014.

[109] S. Mittal and J. Vetter. A Survey of CPU-GPU Heterogeneous Computing Techniques. *ACM Computing Surveys*, 47, 2015.

[110] M. Moeng, H. Xu, R. Melhem, and A. K. Jones. ContextPreRF: Enhancing the Performance and Energy of GPUs With Nonuniform Register Access. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(1):343–347, 2016.

[111] G. E. Moore. Cramming More Components onto Integrated Circuits. *IEEE Journal of Circuits*, 11(3):33–35, 1965.

[112] H. Mujtaba. NVIDIA Gained Major Discrete GPU Market Share With GeForce 10 Series GPUs Over AMD Radeon RX Series in Q2 2018. https://wccftech.com/nvidia-amd-discrete-gpu-market-share-q2-2018/.

[113] O. Mutlu. Memory Scaling: A Systems Architecture Perspective. In *Proceedings of the 5th IEEE Annual International Memory Workshop, IMW*, 2013.

[114] C. Märtin. Post-Dennard Scaling and the final Years of Moore's Law. In *Technical Report, Hochschule Augsburg University of Applied Sciences*, 2014.

[115] 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.

[116] J. Nickolls and D. Kirk. *Graphics and Computing GPUs. Computer Organization and Design (Patterson and Hennessy)*. Morgan Kaufmann, 4th edition, Chapter A, Page A 47, 2012.

[117] C. Nugteren. *Improving the Programmability of GPU Architectures*. Technical University of Eindhoven, 2014.

[118] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[119] NVIDIA. NVIDIA Pascal Architecture White Paper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[120] NVIDIA. NVIDIA Volta Architecture White Paper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[121] NVIDIA. Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview. https://www.nvidia.com/page/8800_tech_briefs.html.

[122] NVIDIA. NVIDIA Unveils CUDA^{TM}-The GPU Computing Revolution Begins, 2006. https://www.nvidia.com/object/IO_37226.html.

[123] NVIDIA. CUDA: Compute Unified Device Architecture, 2007. http://developer.nvidia.com/object/gpucomputing.html.

[124] NVIDIA. Fermi Architecture White Paper, 2009. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf.

[125] L. Nyland, J. R. Nickolls, G. Hirota, and T. Mandal. Systems and Methods for Coalescing Memory Accesses of Parallel Threads, 2011.

[126] M. A. O'Neil and M. Burtscher. Microarchitectural Performance Characterization of Irregular GPU Kernels. In *Proceedings of the IEEE International Symposium on Workloads Characterization, IISWC*, 2014.

[127] D. J. Palframan, N. S. Kim, and M. H. Lipasti. COP: To Compress and Protect Main Memory. In *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture, ISCA*, 2015.

[128] J. Park, H. Esmaeilzadeh, X. Zhang, M. Naik, and W. Harris. FlexJava: Language Support for Safe and Modular Approximate Programming. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, 2015.

[129] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2013.

[130] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. Base-Delta-Immediate Compression: Practical Data Compression for On-chip Caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2012.

[131] A. Perais and A. Seznec. Practical Data Value Speculation for Future High-end Processors. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture, HPCA*, 2014.

[132] C. Qian, L. Huang, Q. Yu, Z. Wang, and B. Childers. CMH: Compression Management for Improving Capacity in the Hybrid Memory Cube. In *Proceedings of the 15th ACM International Conference on Computing Frontiers, CF*, 2018.

[133] 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.

[134] I. Z. Reguly and M. Giles. Efficient Sparse Matrix-vector Multiplication on Cache-based GPUs. In *Proceedings of the Innovative Parallel Computing, InPar*, 2012.

[135] M. Rhu and M. Erez. Maximizing SIMD Resource Utilization in GPGPUs with SIMD Lane Permutation. In *Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA*, 2013.

[136] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2013.

[137] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-Conscious Wavefront Scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2012.

[138] M. Salajegheh, Y. Wang, K. Fu, A. Jiang, and E. Learned-Miller. Exploiting Half-wits: Smarter Storage for Low-power Devices. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies, FAST*, 2011.

[139] D. Salomon. *Variable-Length Codes for Data Compression*. Springer-Verlag, Berlin, Heidelberg, 2007.

[140] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2014.

[141] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning Approximation for Graphics Engines. In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2013.

[142] 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 *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2011.

[143] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate Storage in Solid-State Memories. *ACM Transactions on Computer Systems*, 32(3):9:1–9:23, 2014.

[144] S. Sardashti, A. Arelakis, P. Stenström, and D. A. Wood. *A Primer on Compression in the Memory Hierarchy.* Morgan and Claypool, 2015.

[145] J. Sartori and R. Kumar. Branch and Data Herding: Reducing Control and Memory Divergence for Error-tolerant GPU Applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2012.

[146] V. Sathish, M. J. Schulte, and N. S. Kim. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2012.

[147] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO*, 1997.

[148] E. S. Schwartz and B. Kallick. Generating a Canonical Prefix Encoding. *ACM Communications*, 7(3):166–169, 1964.

[149] M. Sha, Y. Li, B. He, and K. L. Tan. Accelerating Dynamic Graph Analytics on GPUs. *Proceedings of the 44th International Conference on Very Large Data Bases*, 11(1):107–120, 2017.

[150] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[151] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 9th ACM SIGSOFT Symposium and 13th European Conference on Foundations of Software Engineering*, 2011.

[152] M. Sutherland, J. San, M. Natalie, and E. Jerger. Texture Cache Approximation on GPUs, 2015.

[153] J. Tan and X. Fu. Mitigating the Susceptibility of GPGPUs Register File to Process Variations. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2015.

[154] J. Tan, Z. Li, and X. Fu. Soft-error Reliability and Power Co-optimization for GPGPUs Register File Using Resistive Memory. In *Design, Automation Test in Europe Conference Exhibition, DATE*, 2015.

[155] 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.

[156] K. R. Townsend and J. Zambreno. A Multi-phase Approach to Floating-point Compression. In *IEEE International Conference on Electro/Information Technology, EIT*, 2015.

[157] H. W. Tseng, L. M. Grupp, and S. Swanson. Underpowering NAND Flash: Profits and Perils. In *Proceedings of the 50th Annual Design Automation Conference, DAC*, 2013.

[158] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *21st International Conference on Parallel Architectures and Compilation Techniques, PACT*, 2012.

[159] G. J. van den Braak, J. Gómez-Luna, J. M. González-Linares, H. Corporaal, and N. Guil. Configurable XOR Hash Functions for Banked Scratchpad Memories in GPUs. *IEEE Transactions on Computers*, 65(7):2045–2058, 2016.

[160] 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 *Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA*, 2015.

[161] V. Volkov. Better Performance at Lower Occupancy. In *GPU Technology Conference*, 2010. http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf.

[162] M. Wahib and N. Maruyama. Scalable Kernel Fusion for Memory-Bound GPU Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2014.

[163] B. Wang, M. Alvarez Mesa, C. C. Chi, and B Juurlink. Parallel H.264/AVC Motion Compensation for GPUs using OpenCL. *IEEE Transactions on Circuits and Systems for Video Technology, TCSVT*, 25(3):525–531, 2014.

[164] B. Wang, D. F. de Souza, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, A. Ilić, N. Roma, and L. Sousa. Highly Parallel HEVC Decoding for Heterogeneous Systems with CPU and GPU. *Signal Processing: Image Communication*, 62:93–105, 2018.

[165] 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.

[166] H. Wang and Qingkui. Power Estimating Model and Analysis of General Programming on GPU. *Journal of Software*, 7(5):1164–1170, 2012.

[167] 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, CIT*, 2011.

[168] D. Wong, N. S. Kim, and M. Annavaram. Approximating warps with intra-warp operand value similarity. In *IEEE International Symposium on High Performance Computer Architecture, HPCA*, 2016.

[169] H. Wong, 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.

[170] S. Xiao, H. Lin, and W. C. Feng. Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, 2011.

[171] Q. Xu, H. Jeon, and M. Annavaram. Graph Processing on GPUs: Where are the Bottlenecks? In *IEEE International Symposium on Workload Characterization, IISWC*, 2014.

[172] J. Yang, Y. Zhang, and R. Gupta. Frequent Value Compression in Data Caches. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2000.

[173] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *Proceedings of the International Conference on Design, Automation & Test in Europe, DATE*, 2017.

[174] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. Axilog: Language Support for Approximate Hardware Design. In *Proceedings of the International Conference on Design, Automation & Test in Europe, DATE*, 2015.

[175] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. *ACM Transactions on Architecture and Code Optimization*, 12(4):62:1–62:26, 2016.

[176] Y. Zhang, Y. Hu, B. Li, and L. Peng. Performance and Power Analysis of ATI GPU: A Statistical Approach. *Proceedings of the 6th International Conference on Networking, Architecture, and Storage, NAS*, 2011.

[177] Y. Zhang, J. Yang, and R. Gupta. Frequent Value Locality and Value-centric Data Cache Design. *SIGPLAN Not.*, 35(11):150–159, 2000.

9 List of Publications

1. Sohan Lal, Jan Lucas, Ben Juurlink, "SLC: Memory Access Granularity Aware Selective Lossy Compression for GPUs", *IEEE International Conference on Design Automation, and Test in Europe* ***(DATE)****, 2019, France.*

2. Sohan Lal, Ben Juurlink, "A Case for Memory Access Granularity Aware Selective Lossy Compression for GPUs", *ACM Student Research Competition Poster and Extended Abstract at IEEE/ACM International Symposium on Microarchitecture* ***(MICRO)****, 2018, Japan* ***(Semifinalist)****.*

3. Sohan Lal, Jan Lucas, and Ben Juurlink, "Memory Access Granularity Aware Lossy Compression for GPUs", *14^{th} International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* ***(ACACES)****, 2018, Italy.*

4. Jan Lucas, Sohan Lal, Ben Juurlink, "Optimal DC/AC Data Bus Inversion Coding", *International Conference on Design Automation, and Test in Europe* ***(DATE)****, 2018, Germany.*

5. Sohan Lal, Jan Lucas, Ben Juurlink, "E^{2}MC: Entropy Encoding Based Memory Compression for GPUs", *IEEE International Conference on Parallel and Distributed Processing Symposium* ***(IPDPS)****, 2017, USA.*

6. Maurice Peemen, Runbin Shi, Sohan Lal, Ben Juurlink, Bart Mesman, and Henk Corporaal, "The Neuro Vector Engine: Flexibility to Improve Convolutional Net Efficiency for Wearable Vision", *International Conference on Design Automation, and Test in Europe* ***(DATE)****, 2016, Germany.*

7. Sohan Lal, Jan Lucas, Michael Andersch, Mauricio Alvarez Mesa, Ahmed Elhossini, and Ben Juurlink, "GPGPU Workload Characteristics and Performance Analysis", *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* ***(SAMOS)****, 2014, Greece.*

8. Jan Lucas, Sohan Lal, Michael Andersch, Mauricio Alvarez Mesa, and Ben Juurlink, "How a Single Chip Causes Massive Power Bills - GPUSimPow: A GPGPU Power Simulator", *IEEE International Symposium on Performance Analysis of Systems and Software* ***(ISPASS)****, 2013, USA.*

9. Sohan Lal, Jan Lucas, Mauricio Alvarez Mesa, Ahmed Elhossini, and Ben Juurlink, "Exploring GPGPUs Workload Characteristics and Power Consumption", *9th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* ***(ACACES)****, 2013, Italy.*

10. Jan Lucas, Sohan Lal, Mauricio Alvarez Mesa, Ahmed Elhossini, and Ben Juurlink, "DART: A GPU Architecture Exploiting Temporal SIMD for Divergent Workloads", *9th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* ***(ACACES)****, 2013, Italy.*

Acknowledgment

First and foremost, I thank my supervisor Prof. Dr. Ben Juurlink. His supervision style encouraged independent work that, both consciously and subconsciously, taught me how to nourish an idea, do its early evaluation down to experimental results and scientific dissemination. This thesis would not have been possible without his guidance and motivation. I especially thank him for supporting and motivating me when our TACO paper was rejected. I also thank him for supporting me financially through all these years.

I thank all others who directly contributed to this thesis. In particular, I thank Jan Lucas, Micheal Andersch, Mauricio Alvarez-Mesa, and Ahmed Elhossini for being a co-author on papers and for our numerous scientific discussions. I also thank Biao Wang for our discussions on GPU programming and optimizations and for inviting me to China new year party in 2013.

My thanks to everyone at the AES group at TU Berlin. Thanks to my office mates Tamer Dallou, Stefan Hauser, Nico Moser, Nicolas Schier, Gervin Thomas, Philipp Habermann, Matthias Göbel, Angela Pohl, Nadjib Mammeri, Daniel Maier, Biagio Cosenza, Kaijie Fan, Tareq Alawaneh, Farzaneh Salehiminapour, Anastasiia Dolinina, Robert Drehmel who were always very helpful and kind. In particular, I thank Daniel Maier and Philipp Habermann for translating the abstract to German. I also thank Daniel Maier for our daily scientific discussions as we shared the office.

I and Nadjib traveled together for several project meetings. I thank him for scientific discussions and for all his support.

I visited PARsE group led by Prof. Henk Corporaal at TU/e. I thank him for hosting me for three months and for the collaboration opportunity which yielded a joint publication. I also thank Gert-Jan van den Braak for facilitating my stay at TU/e.

I also thank HiPEAC, an excellent network encouraging collaboration between researchers, who sponsored my visit to TU/e. HiPEAC also funded my two visits to ACACES summer school which gave me the opportunity to meet with my peers from across the globe.

Many thanks to Adela Westedt for all technical support to run my experiments. Special thanks to Jana Pilz, Sara Tennsted for all the help and support in paperwork.

I also thank my family and friends for being supportive. Thanks to my parents who always did their best. The journey from a small village in India to TU Berlin would not have been possible without their support. Finally, I thank my wife Pooja for her love, advice, and unending support, especially through the tough phases of my Ph.D. Love to my son Advik, his sweet smile in the evening, after long days at the office, rejuvenated me and inspires me to achieve new heights.