Power-Aware Caches for GPGPUs

by

Ahsan Saghir

Department of Electrical & Computer Engineering
Lakehead University

Thunder Bay, Ontario, Canada
September, 2015

Abstract

General Purpose Graphics Processing Units (GPGPUs) have evolved from fixed function graphics pipeline to massively parallel general purpose processors capable of running thousands of threads on hundreds of cores. However, for running parallel workloads from diverse computing domains, GPGPUs need to be able to provide large volume of data from memory sub-systems. This places a great demand on GPGPU manufacturers to include low-latency memory units, such as caches and shared memory, in order to provide huge amount of data to cores without suffering access delays. The criticality of memory hierarchy is expected to increase as the number of cores in GPGPUs is rising. To address this challenge, GPGPU designers have increased the size of caches in each generation of GPGPUs. However, increasing the size of these memory units also adds to power dissipation which has recently become a major design constraint in the design of microprocessor systems.

In this thesis, we propose two optimization techniques to reduce power consumption in L1 caches (data, texture and constant), shared memory and L2 cache. The first optimization technique targets static power. Evaluation of GPGPU applications shows that once a cache block is accessed by a thread, it takes several hundreds of clock cycles until the same block is accessed again. The long inter-access cycle can be used to put cache cells into drowsy mode and reduce static power. While drowsy cells reduce static power, they increase access time as voltage of a cache cell in drowsy mode should be raised before the block can be accessed. To mitigate performance impact of drowsy cells, we propose a novel technique called coarse grained drowsy mode. In coarse grained drowsy mode, we partition each cache into regions of consecutive cache blocks and wake up a region upon cache access. Due to temporal and spatial locality of cache accesses, this method dramatically reduces performance impact caused by drowsy cells. The second optimization technique relies on branch divergence in GPGPUs. The execution model in GPGPUs is Single Instruction Multiple Thread (SIMT) which means processing cores execute the same instruction with different data for GPGPU threads. The SIMT execution model may result in divergence of threads when a control instruction is executed. GPGPUs execute branch instructions in two phases. In the first phase, threads in the taken path are active and the rest are idle. In the second phase, threads in the

not-taken path are executed and the rest are idle. Contemporary GPGPUs access all portions of cache blocks, although some threads are idle due to branch divergence. We propose accessing only portions of cache blocks corresponding to active threads. By disabling unnecessary sections of cache blocks, we are able to reduce dynamic power of caches. Our results show that on average, the two optimization techniques together reduce power of caches by up to 98% and 15% for static and dynamic power, respectively.

Acknowledgements

I would like to thank my advisor, Dr. Ehsan Atoofian, for his guidance and support, without which this work could not have been possible. I would also like to thank Dr. Ali Manzak for his help with the power estimation of caches. I am grateful to the reviewers for their valuable comments that helped to improve this work. I am indebted to my family for their support that enabled me to complete this work.

Table of Contents

Abstract ii
Acknowledgements iv
Table of Contents v
List of Figures vii
List of Tables viii
List of Abbreviations ix
Chapter 1 Introduction 1
Chapter 2 Background and Literature Review 8
2.1 Background 8
2.1.1 Architecture of GPGPUs 8
2.1.2 Compute Unified Device Architecture (CUDA) 12
2.1.3 Thread Scheduling 13
2.1.4 Parallel Execution Model 14
2.1.5 Memory Spaces 14
2.2 Literature Review 17
2.2.1 Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor 17
2.2.2 An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches 19
2.2.3 Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration 20
2.2.4 Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs 21
2.2.5 Power Gating Strategies on GPUs 22
2.2.6 SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading 23
2.2.7 Drowsy Caches: Simple Techniques for Reducing Leakage Power 24
2.2.8 Warped Register File: A Power Efficient Register File for GPGPUs 25
2.2.9 Warped-Compression: Enabling Power Efficient GPUs through Register Compression ... 26
2.2.10 A Small, Fast and Low-Power Register File by Bit-Partitioning 27
2.2.11 Register File Partitioning and Recompilation for Register File Power Reduction 28
2.2.12 Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm 29
2.2.13 Characterizing and improving the use of demand-fetched caches in GPUs 30

2.2.14 A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures 30
2.2.15 Power-efficient Computing for Compute-intensive GPGPU Applications 31
2.2.16 PATS: Pattern Aware Scheduling and Power Gating for GPGPUs 33
2.2.17 Power-Aware L1 and L2 Caches for GPGPUs... 34
2.3 Overview of previous works and our contribution 34
Chapter 3 Motivation and Optimization Techniques.. 36
3.1 Problem Definition 36
3.2 Cache Access Patterns 38
3.3 Power Reduction Techniques 45
3.3.1 Static Power Reduction Using Drowsy Cells 46
3.3.2 Reducing Dynamic Power Using Active Mask 48
Chapter 4 Methodology and Results 51
4.1 Experimental Results... 51
Chapter 5 Summary and Future Work... 59
5.1 Contributions 59
5.2 Future Work 60
References 61

List of Figures

Figure 2-1: GPGPU Architecture. 9
Figure 2-2: Fermi Architecture (courtesy: NVIDIA Fermi 2009 [9]). 10
Figure 2-3: Fermi Streaming Multiprocessor (SM) (courtesy: NVIDIA Fermi 2009 [9]) 11
Figure 2-4: GPGPU Application hierarchy 12
Figure 2-5: Texture Prefetching Architecture (courtesy of Prefetching in a Texture Cache Architecture [32]) 16
Figure 2-6: Unified Memory Architecture (courtesy of Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor [2]) 18
Figure 2-7: SM with tinyCache (courtesy of An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches [7]) 20
Figure 2-8: GPU system with hybrid memory (courtesy of Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration [30]) 21
Figure 2-9: Compression in register banks (courtesy of Warped Register File: A Power Efficient Register File for GPGPUs [33]) 26
Figure 3-1: Breakdown of static power for different memory spaces. 37
Figure 3-2: Breakdown of cache dynamic power as percentage of total GPU dynamic power. 37
Figure 3-3: Breakdown of total (static + dynamic) power consumed by cache memories. 38
Figure 3-4: Breakdown of accesses to cache blocks. 41
Figure 3-5: Breakdown of inter-access cycles to cache blocks. 44
Figure 3-6: Different granularities for a cache with 8 rows. 48
Figure 3-7: Structure of DWL. 50
Figure 4-1: Performance impact with region size of 16 for one and two cycles wake-up delay. 52
Figure 4-2: Performance impact when region size changes for one and two cycles wake-up delay. 54
Figure 4-3: Static, dynamic and total power saving. 56

List of Tables

Table 3-1: GPGPU Benchmarks and warp utilization 45
Table 4-1: GPGPU-Sim Configuration 51
Table 4-2: Static Power of Caches 55

List of Abbreviations

Abbreviation	Meaning
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
API	Application Programming Interface
PE	Processing Element
GDDR	Graphics Double Data Rate
DRAM	Dynamic Random Access Memory
ALU	Arithmetic Logic Unit
FPU	Floating Point Unit
SFU	Special Function Unit
nvcc	NVIDIA C Compiler
CTA	Cooperative Thread Array
Texel	Texture Element
3-D	Three-dimensional
2-D	Two-dimensional
FIFO	First-In-First-Out
DWL	Divided Word Line
STT-RAM	Spin Transfer Torque-Random Access Memory
RRAM	Resistive Random Access Memory
NVM	Non-volatile Memory
GATES	Gating-Aware Two-Level Scheduler
PSS	Predictive Shader Scheme
DGP	Deferred Geometry Pipeline
eDRAM	Embedded Dynamic Random Access Memory
DVS	Dynamic Voltage Scaling
BDI	Base Delta Immediate
BPRF	Bit-Partitioned Register File
IPC	Instruction per Cycle
LAMAR	Locality-Aware Memory
FMA	Fused Multiply Add
SRF	Scalar Register File
PATS	Pattern-Aware Two-level Scheduler
PC	Program Counter
WL	Word-Line
BL	Bit-Line
BLB	Bit-Line-Bar
SWL	Small Word-Line
PTM	Predictive Technology Model

OpenGL	Open Graphics Library

Chapter 1
Introduction

Computer industry has seen unprecedented growth in the last two decades. At the heart of this growth, is the advancement in microprocessor design. With the advancement in semiconductor technology, the performance of microprocessors has increased at a very fast rate. The performance increase and the cost reduction in manufacturing processes brought a new age in the computer industry. The computers went from being bulky machines occupying huge rooms to being portable devices that fit in small bags. The performance increase in microprocessors enabled software industry to provide applications that run faster and provide more functionality. During this time, the software industry relied on the advances in hardware to provide improved performance for their applications. The same set of instructions (computer programs) ran faster on a newer generation of hardware.

This cycle of advancement in hardware continued for a number of years but has slowed down since 2003 [31]. A few issues contributing to the slow-down in the performance growth of microprocessors are outlined below:

1. Energy consumption has become significant and the heat dissipation has become a major design constraint.

2. There is a growing gap between memory and processor performance. Due to this, the processor has to wait a significant amount of time before the data is available for processing.

3. Various techniques such as instruction pipelining and out-of-order execution were developed to overlap the execution of multiple instructions. However, applications exhibit limited instruction level parallelism and, therefore, the performance gain due to these techniques is limited.

Due to these reasons, microprocessor vendors have adopted multiple processing units in place of single processing units. These multiple processing units are commonly referred to as processor cores. Now, the microprocessor industry produces chips that have multiple cores on a single die. Traditionally, most computer programs are written as sequential programs

and in the past the performance of the sequential programs improved with every new generation of computers. However, this premise does not strictly hold true today. A sequential program is run on only one of the cores. The current trend in microprocessor design is to increase the number of cores while maintain the processing speed of individual cores. Therefore, cores in a microprocessor will not become significantly faster than the ones that are in use today. To improve performance, software developers have to resort to parallel programming paradigm to continue to enjoy performance improvements for their applications.

Parallel programming is not new to the computer industry. Computer programs for high-performance computing applications have been under development for decades. These programs were run on high performance computer devices that are large-scale and expensive. Only a few applications could justify the use of expensive computers and, therefore, parallel programming remained limited to a small community of software developers. Modern day microprocessors are, mostly, parallel computers and parallel programming is critical to harness the full potential of their processing power. This has fueled a need in the programming community to adopt parallel programming, and multiple approaches have been taken to enable programmers to write parallel code.

Microprocessors (or simply processors) are at the heart of processing in a computer, but are not the only processing chips in the system. Another notable processing element in the system is the Graphics Processing Unit (GPU). Graphics Processing Units (GPUs) were originally designed for graphics applications. They had to process millions of pixels and perform massive number of numerical calculations for graphical workloads. This pushed the GPU vendors to adopt designs that were optimized for throughput and were able to keep up with the demands of the ever growing gaming industry. As a result, GPUs took the many-core trajectory to process huge number of pixels. With the advent of parallel programming, GPU vendors took the opportunity to introduce general purpose programming capabilities for their many core GPUs in an effort to increase their share of the processor industry by offering to run compute intensive tasks on their GPUs. This resulted in GPUs evolving from fixed-function graphics pipeline to massively parallel general purpose processors capable of

running workloads from a wide variety of applications including graphics processing. The ability of the GPUs to process workloads from, not only graphics but, other domains have earned them the title of General Purpose Graphics Processing Units (GPGPUs). The GPUs have followed a many-core trajectory for their design for a long time and modern day GPGPUs have hundreds of cores capable of running thousands of threads. The computing power of modern day GPGPUs far exceeds that of general-purpose microprocessors. General-purpose microprocessors are optimized for low-latency for sequential computations. Therefore, the manufacturers designed the processors with large amount of cache memories and complex circuitry, such as branch predictors, in a bid to process sequential instructions as fast as possible. On the other hand the design philosophy of GPUs focused on maximizing execution throughput of parallel applications. The GPUs, thus, employed a huge number of cores to maximize throughput. Each core in a GPU is much simpler than a core in a general-purpose microprocessor, which makes it possible to have hundreds of cores on a GPU in contrast to a few complex cores in a microprocessor.

In this new era of parallel programming, GPUs have become even more important for the software industry in their drive to provide users with applications that are faster than ever before and that can keep up with the growing demand of compute intensive applications in diverse fields such as genome analysis, protein folding, weather prediction, computational finance, molecular simulation, etc.

Applications, generally, have sequential and parallel blocks of code. Instructions within a sequential block have more dependency on other (neighbouring) instructions, whereas, instructions in a block of a parallel program have very little or no dependency on other block instructions. The sequential portion of the code needs to be run on the CPU which is optimized for low-latency. The parallel portion, however, can benefit from being offloaded to a GPGPU for execution. This strategy yields performance improvements for a multitude of applications that exhibit parallelism. To harness the increasing computing power of GPGPUs, a trend of heterogeneous computing has evolved where the sequential parts of an application are run on a CPU while the parallel parts are offloaded onto a GPU. This approach results in reduction of execution time of programs.

Besides performance, ease of programming also plays an important role in the decision to use a particular parallel programming technique. Until 2006, parallel programming using graphics chips was very difficult as programmers had to use graphics application programming interface (API) functions to access the cores in the GPU. This meant that OpenGL® or Direct3D® techniques were needed to program these GPUs. Even though the programming environment was high level, but still, the code was limited in its application due to the necessary use of graphics APIs. The applications that could be programmed using these APIs were limited and this prevented GPUs to be widely used for parallel programming. This, however, changed in 2007 when NVIDIA released CUDA [10]. CUDA stands for Compute Unified Device Architecture. It is a parallel computing platform consisting of general-purpose Application Programming Interface (API) model created by NVIDIA to allow direct access to the GPGPU's virtual instruction set and parallel computational elements. To facilitate the ease of parallel programming, NVIDIA added additional hardware to its GPUs to serve requests of CUDA programs without having to go through the graphics interface at all [31].

The well suited architecture of GPUs for parallel computation coupled with relative ease of programming opened new avenues for software developers to use GPUs in their applications. Today, a number of applications from diverse computing domains use GPUs as numeric computing engines. However, for their use as general purpose GPUs, vendors have to keep modifying the architecture of their designs to make them better suited to the requirements of general-purpose applications. The earlier designs of GPUs employed software-managed local memories instead of caches as a large amount of streaming data was difficult to cache. However, with the use of GPUs in general-purpose computing domains, various workloads exhibit high level of data locality. For this, the GPU vendors have adopted caches in their designs. NVIDIA and AMD both have added caches in their GPUs. For instance, Fermi [9] architecture from NVIDIA offers up to 48KB L1 cache per core, whereas AMD's Fusion GPU [13] offers 16KB L1 cache per core. These models also include global coherent L2 caches. The size of cache keeps increasing with every generation. In case of NVIDIA, the size of L2 cache has been increased from 768 KB in Fermi [9] to 1536 KB in Kepler GK110

[11], and to 2048 KB in Maxwell GM107 [24]. This trend is likely to continue in future generations of GPGPUs.

Large cache memories are power hungry components in GPGPUs and consume significant amount of static and dynamic power. This problem is going to intensify in future as dynamic power reduction has slowed down in recent years due to limited reduction in threshold voltage. Lowering threshold voltage has also resulted in an increase in static power. These factors urge to find ways to reduce power consumption in caches. A number of architectural and circuit level techniques have been proposed to optimize power of caches in processors [6, 14]. However, GPGPUs have distinct cache access patterns and thus provide unique opportunities to reduce power of caches. GPGPUs use round-robin scheduling policy [5] to assign work to cores. So, a thread has to wait to be executed again, until all the other threads have been scheduled for execution. This means that once a cache block has been accessed by a thread, it has to wait hundreds of clock cycles before it can be accessed again. The long inter-access delay of cache blocks presents an opportunity to optimize leakage power as the cache blocks are not being accessed during this time interval. Two techniques [25, 8] can be used to reduce leakage power in caches. In the first technique [25], the cache blocks can be switched off to reduce leakage power. However, this results in the data in the cache block to be lost, which would have to be requested again from the main memory. This poses a set of problems: first, the accesses to the main memory consume an order of magnitude more energy than accesses to on-chip caches and fetching the same data from the memory will erode any power saving achieved by switching off the cache blocks. Second, this will incur a delay in utilizing the data, similar to the delay in the case of a cache miss which is unacceptable. Third, this will put more demand on the bandwidth of the inter-connection network which may pose to be a bottleneck for performance. The second technique [8] puts cache blocks in drowsy state during the long inter-access intervals. After a cache block is accessed, it is immediately put into drowsy mode to reduce leakage power and switched to an ON state whenever it is accessed again. This results in data still being in the cache during the time when it is in drowsy mode. To access a cache block in drowsy cell, its voltage first should be changed to the nominal voltage. This process takes a few cycles which may hurt

performance of GPGPU applications. To mitigate the impact of wakeup latency on performance, we introduce coarse grained drowsy mode scheme. In this scheme, we partition the cache blocks into cache regions of contiguous memory locations and keep the most recently accessed region ON, while keeping other regions in drowsy mode. This exploits the spatial and temporal locality in cache regions, so contiguous memory locations in the same region can be accessed without any wake-up delay. To reduce dynamic power, we exploit the underutilization of cache blocks in GPGPUs. The architecture of GPGPUs is based on Single Instruction Multiple Thread (SIMT) model in which all the threads execute instructions in a lock-step manner. However, due to branch divergence, not all the threads are active at the same time. This prevents a number of applications from fully utilizing warp slots each cycle. This property can be exploited to reduce dynamic power by disabling inactive portions of cache blocks dynamically.

In summary, our work makes the following contributions:

(i). We exploit long inter-access cycle in GPGPUs and propose to save leakage power by putting cache blocks into drowsy mode. To overcome the latency overhead of drowsy cells, we use coarse granularity and partition a cache into regions of cache blocks. We switch all regions of cache blocks to drowsy mode except the region that was most recently accessed. By dynamically switching between drowsy and ON state for cache regions, we are able to reduce leakage power with negligible impact on performance. We call this technique, coarse grained drowsy mode.

(ii). We perform sensitivity analysis on the granularity of cache blocks in a cache region and evaluate the optimal region size to maximum power saving and minimize performance degrading.

(iii). We utilize GPGPU active-mask feature to detect inactive portions of cache blocks before an instruction is scheduled for execution, and reduce dynamic power by disabling bit-lines, word-lines and sense amplifiers of inactive cache cells.

(iv). We perform a detailed experimental evaluation of all L1 caches (data, texture, constant, and instruction), shared memory and L2 cache for the above proposed techniques. We evaluate the power saving achieved for each type of cache and report performance impact for enabling our optimization techniques for all caches concurrently. Our results demonstrate that we achieve up to 98% and 15% static and dynamic power saving, respectively, in caches using our proposed techniques.

The rest of the thesis is organized as follows. Chapter 2 provides the necessary background for our baseline GPGPU architecture model and related work. Chapter 3 explains the motivation behind our work, analyzes cache access patterns for all types of caches in GPGPUs, and discusses the details of our optimization techniques. Chapter 4 provides details of the experimental methodology used to evaluate our proposed techniques and the results obtained for power saving and performance. Finally, in chapter 5, we provide a summary of our work and conclude the thesis.

Chapter 2
Background and Literature Review

In this chapter, the architecture of a GPGPU is explained in detail. These details are important to understand the related work that has been done to reduce power consumption in GPGPUs. In section 2.1, the architecture and programming model of GPGPUs is described. An overview of different techniques that have been proposed to reduce power consumption in GPGPUs is given in section 2.2.

2.1 Background

Modern day GPGPUs are massively parallel processors capable of running thousands of threads on hundreds of cores. To explain the architecture of GPGPUs with consistent terminology, we describe it using NVIDIA and CUDA terminologies; however, our techniques are not vendor specific and can be applied to GPGPUs from other vendors too.

In this section, we use NVIDIA Fermi series [9] GPGPU to describe the architecture of GPGPUs. This provides a basis to understand the GPGPU architecture. Even though other models of GPGPUs may have slight variations from the Fermi architecture, most of the architectural aspects are similar.

2.1.1 Architecture of GPGPUs

A GPGPU consists of a number of Streaming Multiprocessors (SMs) and each SM typically contains 8 to 32 Processing Elements (PEs) or cores. In the case of NVIDIA Fermi series [9], there are a total of 512 cores organized in 16 SMs of 32 cores each. Figure 2-1 shows architecture of a GPGPU. Each SM features a number of CUDA cores and memory spaces optimized for low latency. Theses memory spaces are private to each SM and are categorized into L1 data cache, shared memory and read only constant, instruction and texture caches. The global memory space is divided into six 64-bit memory partitions, providing a 384-bit memory interface. The architecture supports up to a total of 6 GB of GDDR5 (Graphics Double Data Rate 5) DRAM (Dynamic Random Access Memory) memory.

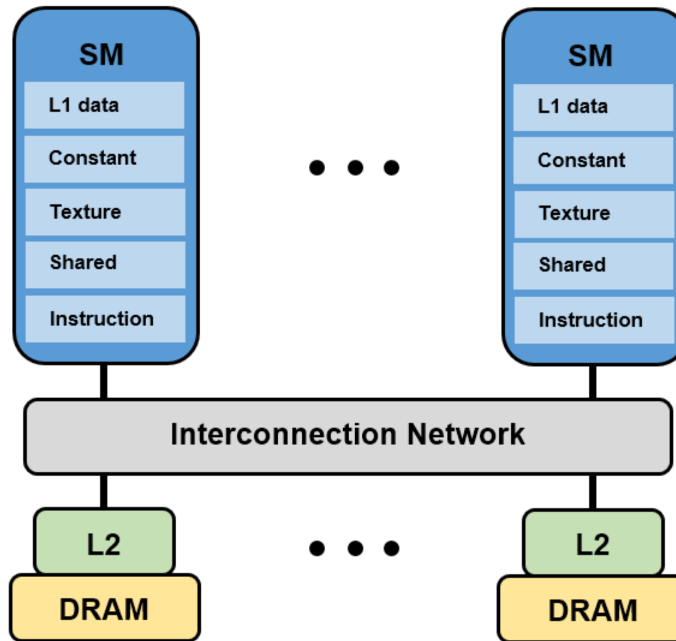


Figure 2-1: GPGPU Architecture.

Figure 2-2 shows the architecture of Fermi series. A vertical rectangular strip represents an SM. It contains an orange portion (scheduler and dispatch). The execution units are represented by the green portion. The blue portion represents register file and L1 cache. The SMs are positioned adjacent to a shared L2 cache. The host interface provides connection to the CPU through PCI-Express. The GigaThread global scheduler is used to distribute thread blocks to thread schedulers in an SM.

Figure 2-3 shows the architectural details of a Fermi Streaming Multiprocessor (SM). Each SM contains 32 CUDA processors or cores. Each CUDA core is composed of a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Every SM also contains 16 load/store units. These load/store units are used to calculate source and destination addresses for sixteen threads in one clock cycle. Supporting units are used to load and store data, for each of the address calculated, to cache or DRAM. Each SM also has 4 Special Functions Units (SFUs) to execute transcendental instructions. The transcendental instructions are used to calculate functions such as sin, cosine, square root and reciprocal. The threads are executed in groups of 32 threads called warps. Each SM has two warp

schedulers and two instruction dispatch units. These units support two warps to be issued and executed concurrently.

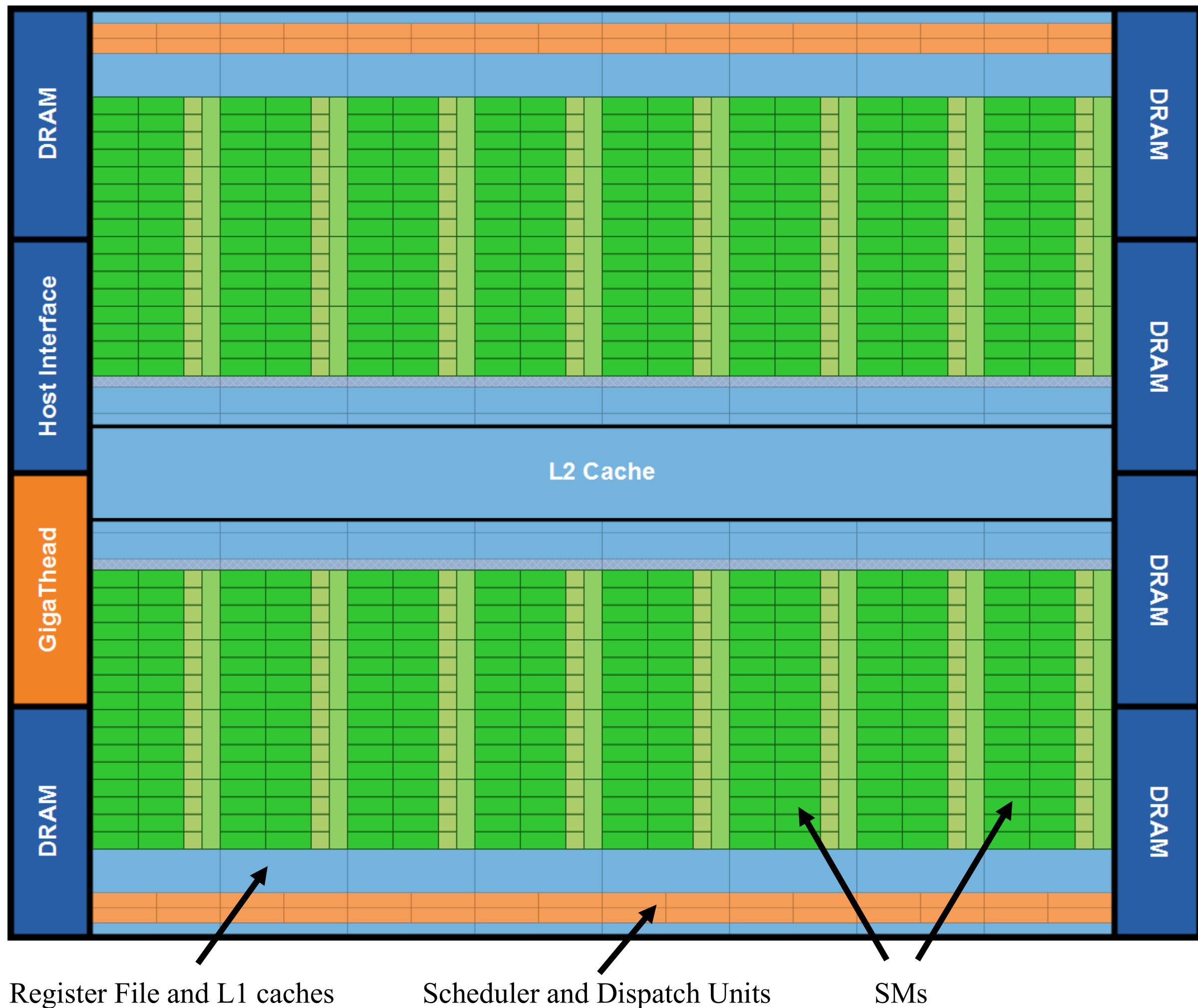


Figure 2-2: Fermi Architecture (courtesy: NVIDIA Fermi 2009 [9]).

The L1 data cache and shared memory share 64 KB of space in an SM and can be configured either as 48 KB of shared memory with 16 KB of L1 cache, or as 16 KB of shared memory and 48 KB of L1 cache. This feature, of providing programmers the choice over the partitioning of memory space between shared memory and L1 data cache, was offered to benefit different types of workloads. Applications that make extensive use of shared memory can benefit from more space allocated to shared memory while for other applications, whose memory accesses are not known beforehand, a larger space for L1 cache is likely to improve performance. This also suggests that the structure of L1 cache and shared memory data lines

is similar, since they are configurable to share the same space. The memory is banked to offer greater bandwidth and each bank is associated with a slice of shared L2 cache. An interconnection network provides the connection between SMs and L2 cache. For this work, we use a 2D mesh topology for the inter-connection network as it is simple to implement and is throughput-effective [4].

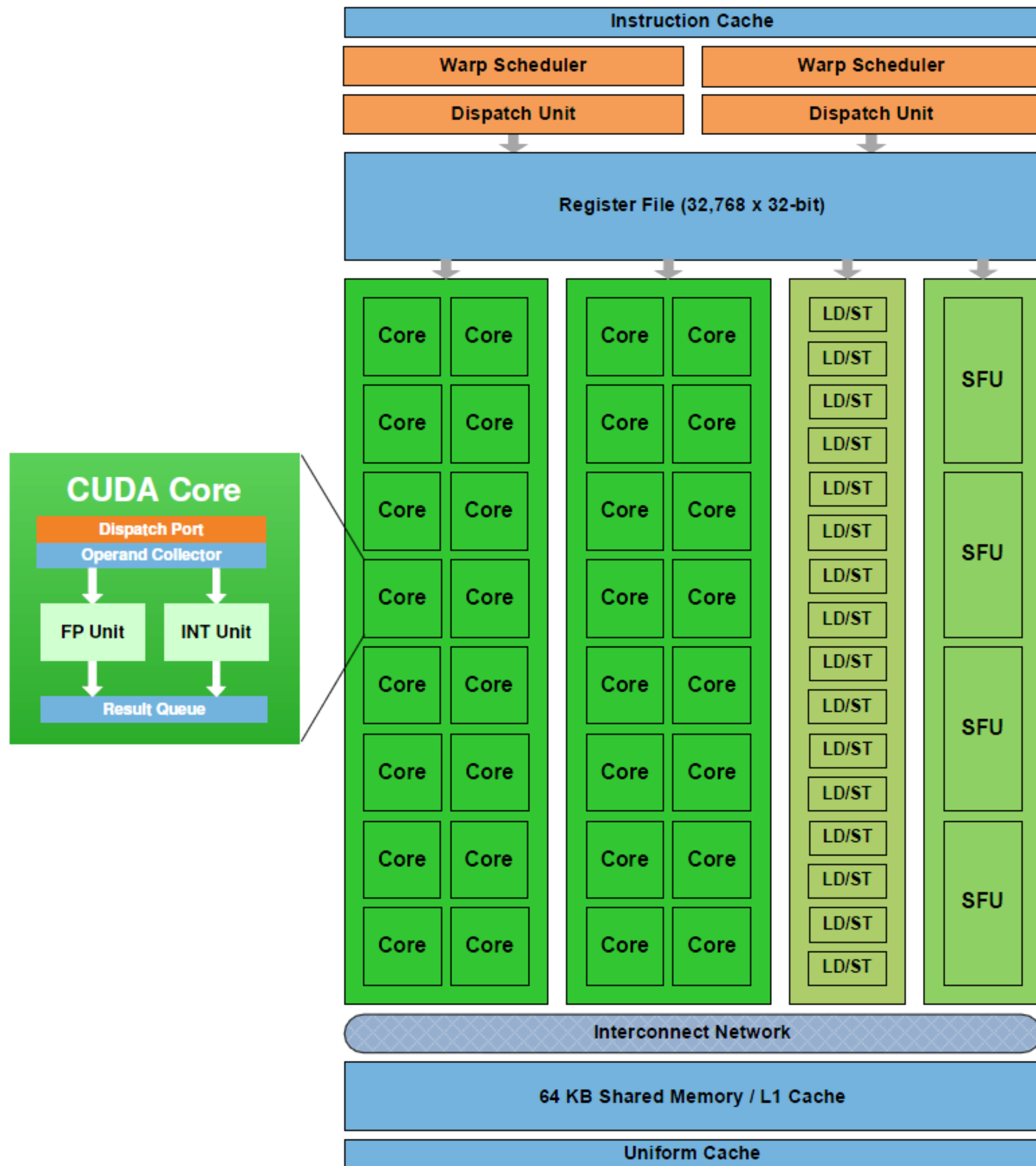


Figure 2-3: Fermi Streaming Multiprocessor (SM) (courtesy: NVIDIA Fermi 2009 [9])

2.1.2 Compute Unified Device Architecture (CUDA)

A program that runs on the GPGPU is called a CUDA program. CUDA is a parallel computing platform consisting of Application Programming Interface (API) model created by NVIDIA to allow direct access to the GPGPU's virtual instruction set and parallel computational elements. A CUDA program consists of blocks of code that are executed on either the host (CPU) or the device (GPGPU). The blocks that exhibit high level of data parallelism are implemented in the device code whereas blocks that exhibit little or no data parallelism are implemented in the host code. The NVIDIA C compiler (nvcc) separates the host code and the device code. The host code is further compiled by the C compiler on the host to run as a CPU process. The device code is typically further compiled by the nvcc to execute on the GPGPU. The device code is composed of data parallel functions, called kernels (figure 2-4).

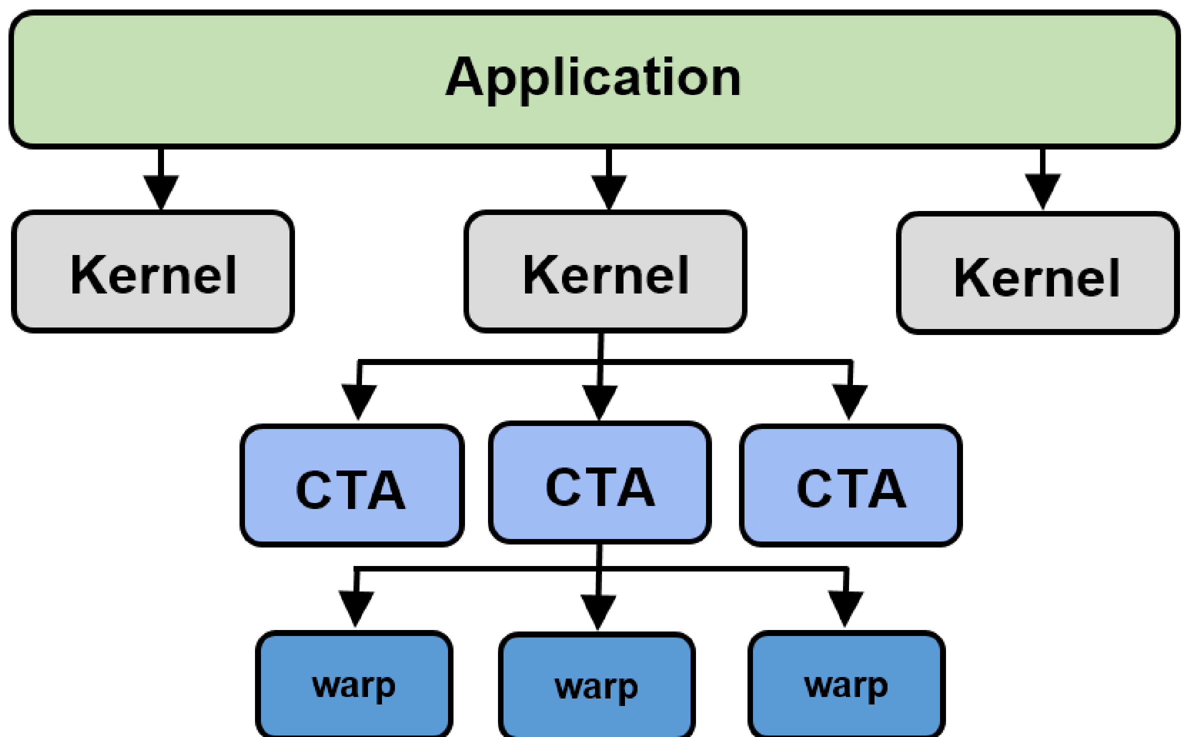


Figure 2-4: GPGPU Application hierarchy

The kernel divides the work over a large number of threads to exploit data parallelism. All the threads generated during a kernel invocation are, collectively, called a grid. A grid is

divided into identically sized Cooperative Thread Arrays (CTAs). Within each CTA, threads are grouped together to form warps. Each warp is composed of 32 threads on current NVIDIA GPUs. Size of the warps is implementation specific and is not part of the CUDA specification. The programmer does not need to know the size of the warp to program in CUDA, but it can help in certain cases to optimize programs. The kernel function, on invocation, assigns CTAs to SMs for execution. To hide latency of long global memory operations and to effectively utilize resources in an SM, more than one CTA can be assigned to an SM. However, the number of CTAs that can be assigned to an SM is limited by the resources of the SM such as register file, size of shared memory, number of threads, etc. [10]. For example, if a CTA requires 12 KB of shared memory and the baseline SM has 48 KB available, then only 4 CTAs can be launched simultaneously on each SM.

2.1.3 Thread Scheduling

The unit of scheduling on SMs is a warp. The SM executes one warp at a time. The GPGPUs use scheduling of warps to hide latency of long-latency instructions by selecting a warp ready for execution over a warp that is stalled due to long-latency operations. The scheduling does not introduce any delay to the execution and is, therefore, referred to as zero overhead thread scheduling. With enough warps, the hardware is able to find warps ready for execution, in spite of having some warps waiting for long-latency instructions. This approach makes full use of the execution hardware. Different types of schedulers can be used to schedule warps for execution. A one level round-robin scheduler [5] schedules warps in a round robin fashion. This makes the warps progress at the same speed. Since all the warps are typically executing the same instructions, this can increase the probability of warps arriving at the long-latency instructions at the same time. This can stall all warps in a kernel and can adversely affect performance. In a two-level scheduler [17], the scheduler partitions warps into two groups. An active group holds warps eligible for execution and an inactive group holds pending warps. Warps waiting for long-latency operations such as global memory load are placed in the pending set. Once a warp becomes ready for execution, it is removed from the pending list and is inserted into the active list of warps. This approach can avoid stall

cycles encountered in the case of one level round-robin based scheduler [5] since the execution of warps progresses at different speeds. Therefore, the same long-latency instruction, in different warps, is spread out over time. This results in an increased probability of the scheduler to find ready warps for execution. In this work, we employ a two-level scheduler [17].

2.1.4 Parallel Execution Model

The programming model of GPGPUs is Single Instruction Multiple Thread (SIMT). This means that all the threads within a warp execute the same instruction. This architecture reduces hardware cost since it allows the cost of fetching and processing an instruction to be amortized over a large number of threads. This behaviour, of all the 32 threads executing the same instruction, is followed as long as all the threads within a warp have the same control path. However, if a GPGPU executes a control dependent instruction, then the set of instructions executed by threads may vary. For example, in the case of an if-then-else construct, the program is executed in two phases. In the first phase, the taken path is executed and in the second phase the not-taken path is executed. During these two phases, threads corresponding to each path are enabled and the rest are disabled. This phenomenon, of threads in the same warp following different control flow paths, is called branch divergence. In normal programming practice, it is recommended to avoid branch divergence as much as possible since it results in partial utilization of execution resources. However, there are scenarios where it is not possible to avoid branch divergence and programs use only a subset of the total threads within a warp during the course of execution.

2.1.5 Memory Spaces

The global memory space is shared by all the threads. To access data in the global memory, the accesses have to go through a two-level cache hierarchy. L2 cache is shared by all the SMs whereas L1 caches are private to an SM. The L1 caches are not coherent and follow a write-evict, write-no-allocate policy [10]. The L2 caches, on the other hand, are coherent and use write-back with write-allocate policy [10]. The cache blocks in GPGPUs are typically

wider than cache blocks in general-purpose microprocessors. In Fermi family of NVIDIA GPGPUs, the L1 and L2 cache blocks are 128 bytes with the exception of L1 texture caches. The wide cache lines allow load/store instructions of all threads within a warp to complete in a single instruction if the load/store addresses map to the same cache line. There are different types of L1 caches namely, L1 data cache, constant cache, texture cache and instruction cache. These caches are private to each SM. In addition to these L1 caches, every SM also has a shared memory space which is managed by the programmer to improve program performance by storing data that is frequently accessed. L1 data cache is used by a GPGPU to improve performance of applications that exhibit data locality. The L1 data cache is a read/write cache.

Constant cache, on the other hand, is a read only cache that is used to store data that does not change over the course of kernel execution. The programmer can, explicitly, select data that can be stored in the constant cache for faster access during kernel execution. This can be particularly useful for look-up tables and other constant data.

Instruction cache is used to provide faster access to future instructions by caching instructions, speculated, to be executed in future. In Fermi, each SM has 64 KB of on-chip memory space that can be partitioned between L1 cache and shared memory. Since the shared memory is on-chip, it is much faster than global memory. Shared memory is a programmer-managed memory space and is shared by all the threads within a CTA. Since shared memory is shared by all threads (in a CTA), this can be leveraged to share data between threads of the same CTA without having to refer data in the global memory. This helps to reduce utilization of memory bandwidth and also improves performance by reducing latency of memory accesses.

Texture cache is a read only cache that is used to improve performance of graphics applications. Texture caches are optimized for multi-dimensional data locality, particularly, useful for processing data associated to texels (texture element) in texture filtering. Figure 2-5 shows the prefetching in a texture cache architecture. To process fragments the architecture follows a number of stages. A fragment corresponds to a single pixel that includes colour,

depth, and texture coordinates. The first stage of the architecture involves the rasterizer looking up the cache tags for texel addresses of each of the fragments. A rasterizer is a graphics hardware system used to convert a 3-D (three-dimensional) world into a 2-D (two-dimensional) image. Other data related to the fragment is forwarded to a fragment FIFO (First-In-First-Out).

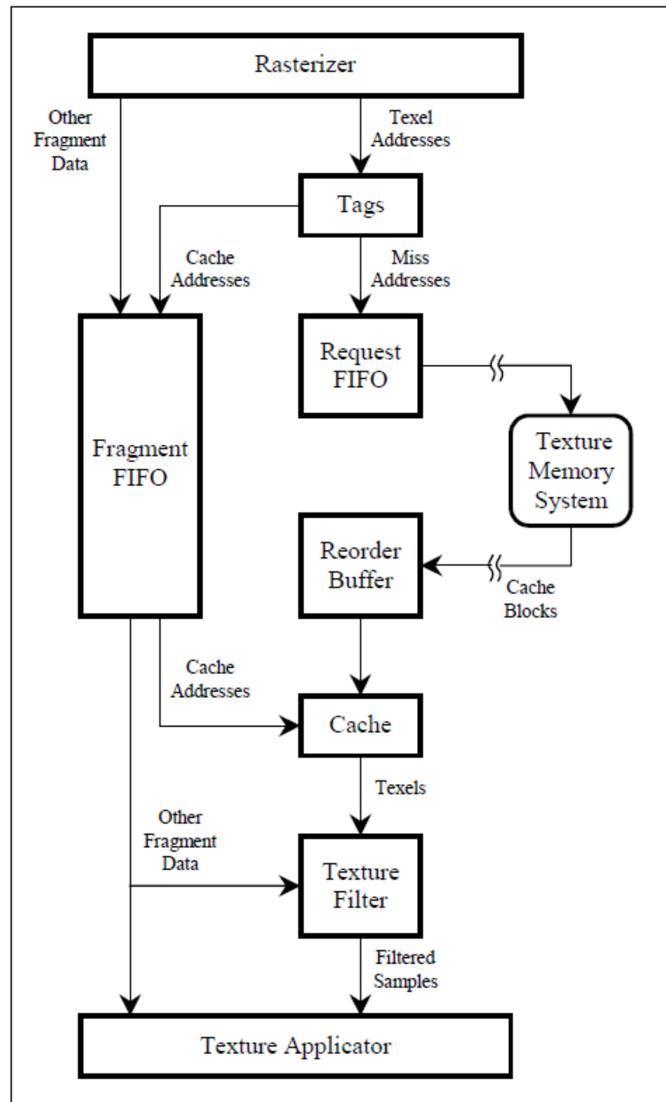


Figure 2-5: Texture Prefetching Architecture (courtesy of Prefetching in a Texture Cache Architecture [32])

If the texel addresses are not found in the cache tags, indicating a cache miss, the cache tags are updated with the texel addresses of the fragment. The address of the texels is forwarded

to the memory request FIFO. The cache addresses for the texels are also forwarded to fragment FIFO to be stored with the other data that is needed to process the fragment. The request FIFO sends requests to the texture memory system for the texel data. At the same time, space is reserved in the reorder buffer for the returning data. This allocation of space ensures that the system does not run into dead-lock if an out-of-order memory system is used. If the memory system is in-order, a FIFO can be used instead of the reorder buffer because the data from the memory would arrive in the same order as the requests sent to the memory. When a fragment reaches the head of the fragment FIFO, it can be processed if the data needed for all of its texels is present in the cache. This means that fragments that did not generate any misses can be processed immediately. The fragments, for which the texel data is not present in the cache, must first wait for the corresponding texel data to arrive from the memory system. It is important to avoid a scenario where the older cache blocks are not over-written prematurely by new cache blocks. This is ensured by committing the new cache blocks only when their corresponding fragment data is available at the head of the fragment FIFO. After all the texel data corresponding to the fragment is available, the fragment is removed from the head of the FIFO and is processed by the texture filter and applicator.

2.2 Literature Review

Extensive work has been done in GPGPUs to reduce power consumption without affecting performance. In this section, we review various techniques proposed for reducing power in processors and provide a brief explanation of each technique and explain how our work differs from the work done by others.

2.2.1 Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor

GPGPUs are increasingly being used for high performance computing applications. This puts more demand on caches. Low-latency memory spaces are required to exploit significant data locality available in general purpose applications and reduce execution time of programs. Existing implementations of GPGPUs use a one-size-fit-all policy and hard-partition local

storage of an SM during the design time. However, GPGPU applications are diverse in the nature of memory requirements and a single memory unit is often most critical to performance of a given application. NVIDIA's Fermi [9] offers flexibility to programmers over the choice of size between shared memory and L1 data cache at a coarse granularity of 16/48 KB. This helps certain benchmarks that make use of shared memory to benefit from an increased shared memory size. Other benchmarks that do not make much use of shared memory can benefit from a larger L1 data cache. However, this feature is only reserved for shared memory and L1 data cache. This does not include the register file in GPGPUs, which is also critical to the performance of certain applications.

Gebhart et al. [2] proposed a unified local memory which can dynamically change the capacity of register, shared memory, and cache on a per application basis. This can help applications to benefit from an increased low-latency space according to the needs of the application. For example, if an application requires more shared memory but does not make much use of register file, then the register file space can also be used for storing shared memory data. This helps in improving the performance of the application. Gebhart et al. [2] proposed a unified memory architecture that aggregates different memory units and allows a flexible allocation of memory customized for each application. Figure 2-6 shows the proposed unified memory architecture.

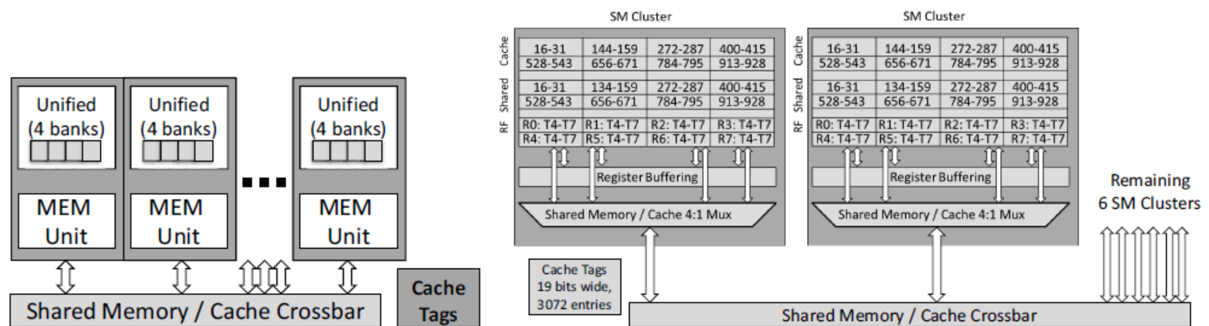


Figure 2-6: Unified Memory Architecture (courtesy of Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor [2])

The accesses to global memory are reduced in the unified memory architecture. This customized allocation of memory offers tuning that improves both performance and energy of GPGPUs.

Our work focuses on reducing leakage and dynamic power of caches by using drowsy caches and Divided Word Line (DWL). Therefore, our work differs from the work of Gebhart et al. [2] that unifies on-chip memory spaces to reduce global memory accesses. This technique [2] can be used along with our proposed optimization techniques to further reduce power consumption in GPGPUs.

2.2.2 An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches

Sankaranarayanan et al. [7] observed that L1 data cache and scratchpad memory consume a significant portion of dynamic energy as they have to service a large number of cores. They proposed adding tinyCache to reduce power of L1 data cache. A tinyCache is a small filter inserted between a Processing Element (PE) and an L1 data cache and intercepts accesses to the shared L1 cache. This technique reduces number of expensive requests to L1 data cache by replacing these requests with energy-efficient accesses to the tinyCache. The inclusion of tinyCache for each PE presents a problem of addressing cache coherency. Since each PE has a private tinyCache, it is necessary to maintain coherency across tinyCaches of an SM. To reduce coherence overhead, Sankaranarayanan et al. proposed to either evict content of tinyCache into L1 cache (e.g. for barriers) or bypass tinyCache (e.g. for atomic operations).

Figure 2-7 shows the architecture of a Streaming Multiprocessor containing tinyCaches. TinyCache filters out a sizable portion of memory accesses to the L1 cache and is able to reduce power of L1 data cache and scratchpad memory.

This work [7] focuses on reducing dynamic power of L1 cache, whereas, our work reduces both static and dynamic power for L1 as well as L2 caches. The technique proposed by Sankaranarayanan et al. [7] can be combined with our proposed techniques to further reduce power consumption in caches.

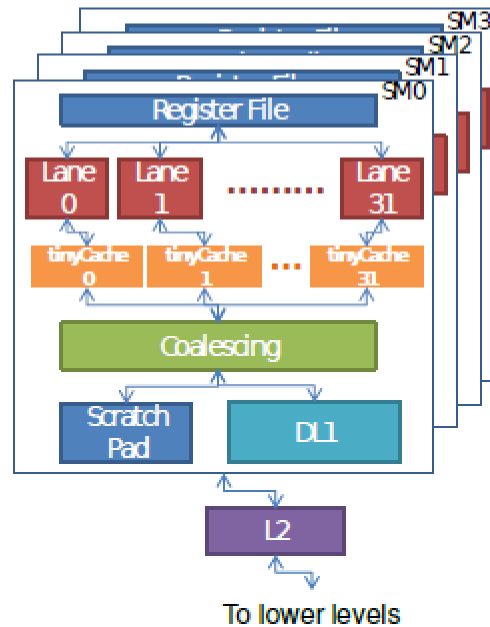


Figure 2-7: SM with tinyCache (courtesy of An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches [7])

2.2.3 Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration

J. Zhao et al. [30] proposed a hybrid graphics memory system that integrates different memory technologies such as DRAM (Dynamic Random-Access Memory), spin-transfer torque memory (STT-RAM) and resistive memory (RRAM). Non-volatile memory (NVM) such as STT-RAM and RRAM do not require refresh operations and have very little standby power. However, they have longer write latency and high write energy. Their design [30] of memory replaces part of the DRAM with NVM (STT-RAM and RRAM). By moving the read-only and infrequently-accessed data to the NVM partition, the hybrid memory system is able to consume less power than GDDR5 (Graphics Double Data Rate 5) memory in GPGPUs. Figure 2-8 shows the architecture of GPU with hybrid memory. On the left is the conventional GPU with off-chip DRAM. On the right is the GPU architecture with hybrid memory.

The work done by J. Zhao et al. [30] focuses on reducing power consumption of main memory in GPGPUs. Our work is different as we focus on power of caches. Our proposed

techniques and the techniques proposed by J. Zhao et al. [30] can be combined to further reduce power in GPGPUs.

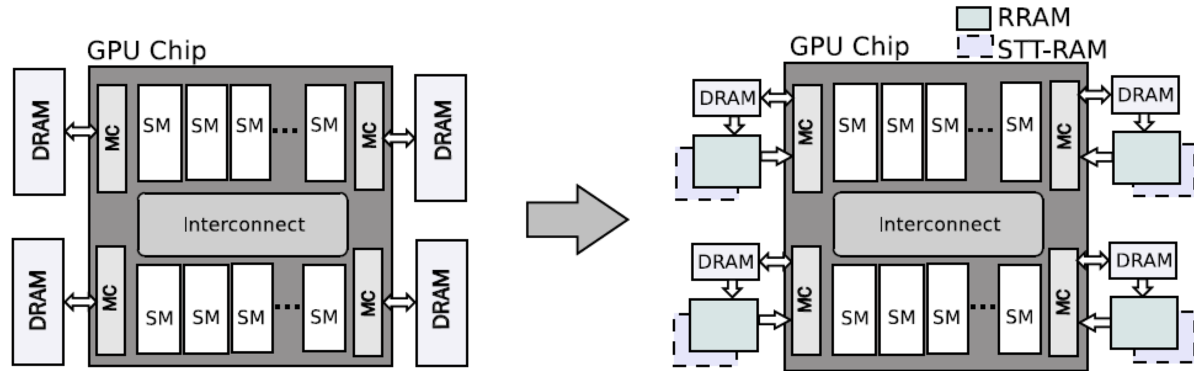


Figure 2-8: GPU system with hybrid memory (courtesy of Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration [30])

2.2.4 Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs

Abdel Majeed et al. [27] proposed warped gates to switch off execution units within a Streaming Multiprocessor (SM) to reduce leakage power. GPGPU execution units are idle only for very short periods of time. Conventional power-gating techniques are not able to switch these execution units to save power. They showed that the two-level warp scheduler [17] greedily schedules instructions to execution units, resulting in short switching cycles between different types of execution units. This prevents the execution units from being idle for longer periods of time. They proposed a gating-aware two-level warp scheduler (GATES) over the two-level scheduler [17] proposed by Gebhart et al. The gating aware scheduler optimizes idle periods in execution units. GATES priorities issuing groups of instructions that run on the same type of execution unit for longer intervals before switching to the other group of instructions that use another type of execution unit. This approach increases the idle time and provides opportunity to power gate idle execution units. While, the proposed scheduler increases the idle periods for execution units, there are still many idle periods that are not long enough to justify switching off these execution units. The idle time has to be so long that the power saved by gating is more than the overhead of gating. To ensure this, they also proposed a new power gating scheme, called Blackout power gating. This power gating

scheme forces execution units to be gated for at least as many cycles that are necessary to recover the power gating overhead. The power gating scheme is enforced even when there are instructions waiting to use the execution unit that has been gated. This can, however, reduce performance as the instructions have to wait for the execution unit to become ready for execution. To reduce performance loss using Blackout, they use a runtime approach to adaptively adjust the amount of time a unit is idle before it can be gated. This runtime approach is referred to as Adaptive Idle Detect. The two techniques, GATES and Blackout are combined together to create a power gating scheme called Warped Gates. Warped Gates is able to save the static power in execution units.

In our work, we propose to reduce power in caches as opposed to execution units, proposed by Abdel Majeed et al. [27]. The techniques proposed [27] by Abdel Majeed et al. are compatible with our proposed techniques and can be combined with our techniques to further reduce power in GPPGUs.

2.2.5 Power Gating Strategies on GPUs

P.-H. Wang et al. [28] proposed architectural-level power gating to reduce leakage power. Their proposed technique targets hardware in a GPU such as shader-clusters, fixed-function geometry units, and non-shader execution units. Their scheme turns off units at a coarser granularity than warped gates [27]. The quality of visual perception is, often, measured by frames per second. They observed that the shader resources required to satisfy the quality of visual perception vary across frames, depending on the complexity of the scene. They proposed a Predictive Shader Scheme (PSS) technique to predict the required shader resources for the next frame, and turn off the extra shader clusters that are not required. This reduces leakage power in shader clusters. They also observed that there is an imbalance between geometry and fragment computation. The fixed-function geometry units often need to wait for fragment computation to complete. This results in long stall time of the fixed-function geometry units. They proposed Deferred Geometry Pipeline (DGP) mechanism to turn off execution and memory circuits in the fixed-function geometry units during the time for which these units are idle. In their work, they also analyzed idle periods for non-shader

execution units. A significant percentage of non-shader execution units remain idle for a long number of cycles. They use time-out power gating for non-shader execution units with request batching. Time-out power gating is used to turn off a circuit after observing a streak of idle cycles ($T_{idledetect}$) for the circuit. Some pipeline stages have shorter idle periods. For these stages, request batching can be employed to achieve more reduction in leakage power. When a new request is seen, a sleeping unit is not immediately activated. Instead, its wake-up action is delayed until the input buffer of the next pipeline stage remains under a pre-defined threshold. Thus, request batching gathers short idle periods into a longer period.

Using these techniques, P.-H. Wang et al. [28] are able to reduce leakage power of shader clusters, fixed-function geometry units, and non-shader execution units. In our work, we focus on reducing power in caches, and therefore, our techniques can be combined with the techniques proposed by P.-H. Wang et al. [28] to further reduce power in GPGPUs.

2.2.6 SRAM-DRAM Hybrid Memory with Applications to Efficient Register Files in Fine-Grained Multi-Threading

Wing-kei S. et al. [29] proposed a SRAM-DRAM hybrid register file for GPGPUs. In their memory cell design they integrate embedded DRAM (eDRAM) into SRAM cells. Each SRAM cell is supplemented by multiple DRAM cells. This way, each memory cell is able to store multiple bits. The configuration is such that a value can be locally copied between the SRAM cell and one of the DRAM branches within that cell. The hybrid cell design allows external access to the SRAM cell, but does not allow direct access to DRAM branches. They use the term, multi-context memory, to refer to their memory organization where cells are partitioned into multiple contexts. Only one active context is accessible at a time. To access other dormant contexts, the active context needs to be explicitly switched. The external accesses only see the low-latency SRAM cell for the active context and, thus, the memory cell is able to hide the DRAM latency when accessing the active context. To access dormant contexts, they must first be made active by copying from DRAM to SRAM. This process of making a dormant context active is called context switching. Context switching involves accessing DRAM branches and incurs delays. To address the issues of context-switching

latencies and DRAM refresh, they propose a scheduling algorithm to do context-aware warp scheduling. Their design allows less energy consumption due to a reduction in silicon chip area and reduced capacitance in word-lines and bit-lines in hybrid memory. Hybrid memory uses DRAM capacitors as storage elements for dormant contexts. Since DRAM leaks charge over time, it must be refreshed periodically to retain data. This implies that dormant contexts need to be made active and then stored back as dormant contexts. This cycle is similar to a refresh for the DRAM capacitor. A refresh timer is added to each context to ensure that it is refreshed within retention time, in order to maintain consistency of data in the context. If a refresh timer for a context is close to the retention limit, the context is forcefully made active. This means that the active context also needs to be copied to DRAM. This can incur delays. However, the authors observe that in the multi-threading architecture, each warp is re-scheduled within a reasonably short period of time. This means that each dormant context is made active, most of the time, before its refresh timer reaches the retention limit. Thus, DRAM refresh does not affect the normal pipeline operations.

Their work uses this hybrid memory cell design for the register file in GPPGUs. This work [29] is different from our work as we focus on power reduction in caches. Their work [29] can be used along with our power-optimization techniques to further reduce power consumption in GPGPUs.

2.2.7 Drowsy Caches: Simple Techniques for Reducing Leakage Power

K. Flautner et al. [8] proposed to put cache lines into a state-preserving, low-power drowsy mode for SRAM caches in general-purpose microprocessors to reduce static power consumption. The information in the cache line is preserved in the drowsy mode, but the cache line needs to be switched to a high-power mode before the contents of the line can be accessed. The technique of Dynamic Voltage Scaling (DVS) is used to implement drowsy caches. Two different supply voltages can be selected for each cache line, one for drowsy mode and one for active mode. The changes that need to be made to the standard cache line include a drowsy bit, a mechanism to control the voltage to the cache line, and a word line gating circuit. The voltage controller controls the operating voltage of an array of memory

cells and the voltage level is determined by the state of the drowsy bit. For access to the drowsy line, the drowsy bit is cleared which switches the supply voltage to high V_{DD}. If the cache line is in drowsy mode, accesses to it must be prevented, before the voltage level is raised. This is done in order to prevent contents of the cache line, upon access, from being lost when the supply voltage of the cache line is low. The wordline gating circuit is used to ensure that the cache line is only accessed when the supply voltage is high. Upon a cache access, the cache controller reads the drowsy bit to determine the condition of the voltage of the cache line. The contents of the cache line are accessed without any penalty in performance provided the line is in normal mode. However, if the line is in drowsy mode, its voltage first must be raised to high voltage before its contents can be read.

This work [8] uses drowsy cache for general-purpose microprocessors. We discuss this work here as it provides the basis for the drowsy cache cells in SRAM. However, in our work, we use drowsy cache cells for caches in GPGPUs since the structure of SRAM cells for caches is the same in GPGPUs. Our technique uses coarse grained drowsy scheme which is different from the technique adopted in this work [8]. The technique [8] proposed in this work only targets static power in caches. We also target dynamic power of caches in our work.

2.2.8 Warped Register File: A Power Efficient Register File for GPGPUs

Warped register file [16] presents two techniques to reduce static and dynamic power consumption of GPGPU register files. The first technique involves a tri-modal register access control unit to reduce static power. This control unit enables a register file to switch between three modes, namely, ON, OFF and drowsy state. The proposed technique uses compiler to turn off unallocated registers and places the rest into drowsy mode to reduce leakage power. The registers are switched to ON, upon access, and are aggressively put into drowsy mode immediately after each access. The work also proposes a technique that prevents charging bit-lines and word-lines of registers associated with inactive threads to reduce dynamic power. This technique involves using active mask to determine inactive threads.

Our work is different from this work as we focus on optimizing power in caches while this work [16] targets power optimization in register file. Our work and warped register file [16] can be combined to achieve even greater power savings.

2.2.9 Warped-Compression: Enabling Power Efficient GPUs through Register Compression

Sangpil Lee et al. [33] proposed a warp-level compression scheme for reducing power consumption of register files in GPGPUs. Their work suggests that register values of threads within a warp exhibit arithmetic similarity, meaning the values for successive threads have very small arithmetic differences. The register values can, thus, be compressed to remove data redundancy. This reduces the effective register width and presents opportunities to reduce power consumption. Figure 2-9 shows how compression can help to reduce register file occupancy.

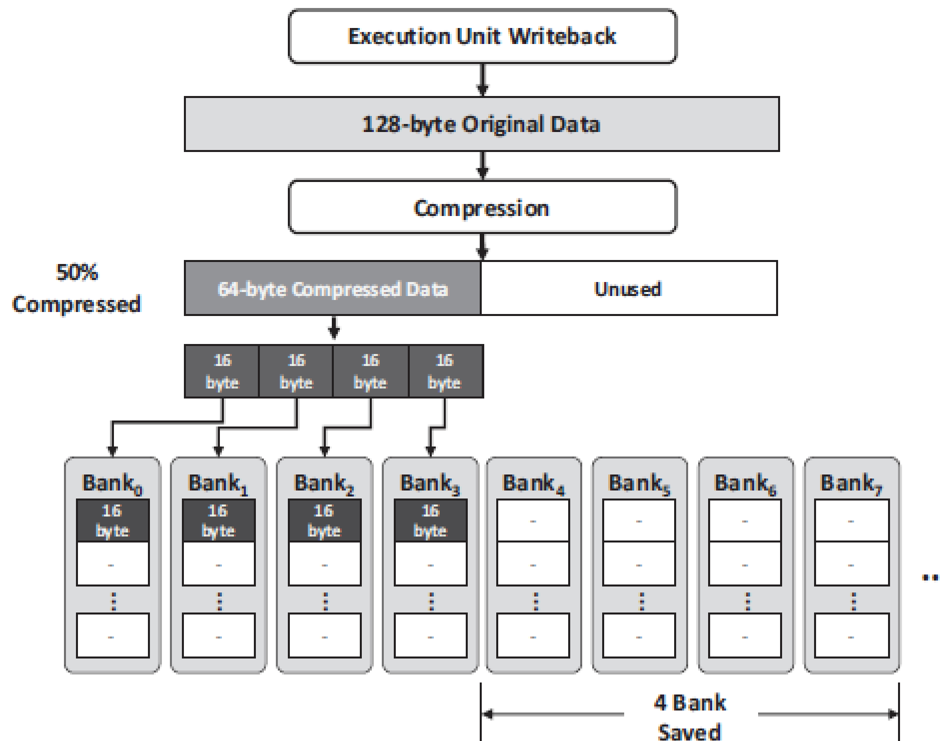


Figure 2-9: Compression in register banks (courtesy of Warped Register File: A Power Efficient Register File for GPGPUs [33])

The original data of 128-byte can be compressed to 64-byte. This compressed data can be saved in 4 register banks instead of 8 register banks needed for the uncompressed data. Since the register width is reduced by compression, it can help to save dynamic power of register file as less number of bits are needed. The work [33] proposes a low-cost and implementation-efficient base-delta-immediate (BDI) compression scheme to compress register file data in GPGPUs. The BDI compression scheme works by selecting one of the thread register values as the primary base value and then computing difference (or delta) values of all other thread registers with respect to the base value. The BDI scheme can be used for quick compression and decompression of data and is suitable for use in this architecture since its impact on performance is marginal.

Sangpil Lee et al. [33] focus on power reduction in register file and our work targets caches in GPGPUs. Our work can be combined with their work [33] to reduce the power consumption even further.

2.2.10 A Small, Fast and Low-Power Register File by Bit-Partitioning

As the issue width and instruction size window increases in today's dynamically scheduled superscalar processors, the number of ports and the size of the register file must be increased to exploit instruction level parallelism. The problem with increasing the size of the register file is the increase in power consumption and access delays. Masaaki Kondo et al. [34] proposed a Bit-Partitioned Register File (BPRF) to address these issues. Typical bit-width of registers is 32-bit or 64-bit. However, many operands do not need the full bit-width. Since for many operations, the effective bit-width of operands is shorter than the full bit-width, the upper bits are redundant. Storing these redundant bits wastes chip area, and access to these bits wastes unnecessary power. A Bit-Partitioned Register File (BPRF) helps to use these redundant upper bits for other operands by partitioning the register entries. Partitioning the register file reduces the bit-width of a register entry and increases effective size of the register file. In the case of operands whose bit-width exceeds the bit-width of one entry, multiple entries are allocated in the register file. For operands that fit within the register bit-width, only one entry is reserved. This efficiently manages the limited register space and

increases the number of register entries. This technique helps to achieve higher Instruction per Cycle (IPC) in a smaller chip area and, thus, uses less power by efficient utilization of register storage space.

Although this technique is for CPUs, but it provides an insight into data patterns for different applications. GPGPUs are widely used for running general-purpose programs ported from CPUs. This means that the data patterns exhibit a great deal of similarity for CPUs and GPGPUs. We discuss this technique [34] here to provide the reader with more information on the type of optimizations done for register files. These techniques can also be applied to GPGPUs. Since our work focuses on reducing power consumption in caches, the techniques used for power reduction in register files can complement our work in achieving greater power savings.

2.2.11 Register File Partitioning and Recompilation for Register File Power Reduction

A significant portion of energy consumption in processors is related to register files due to their large switching capacitance and long working time. However, not all registers contribute evenly to power consumption. Some registers contribute to a major portion of the total power consumption in register files, while others contribute only a fraction. Xuan Guan et al. [35] proposed to partition the register file into hot and cold regions. The registers that are accessed most frequently are placed in the hot region, whereas the registers that are less frequently accessed are put in the cold region. The registers that are most frequently accessed, i.e. the ones in the hot region, can be put in a special small section of the file that consumes much less power than the whole register file. The problem, however, in this approach is that the registers that are most frequently accessed may not necessarily have consecutive addresses in the register file. Since only contiguous memory locations in a physical register file can be partitioned into regions, the register file partitioning is accomplished by swapping the registers. Swapping of the registers is done by reallocating the registers after code generation. Their work employs a graph partitioning algorithm to select the candidates for hot register file region, i.e. the registers that are most frequently accessed. The variables are then re-mapped with a register reallocation process, which results in putting

the hot registers and cold registers in separate regions. For register reallocation, the assembly code has to be modified and, thus, this approach requires recompilation of program code. They use two techniques to reduce static and dynamic power. The first technique involves bit-line splitting to reduce dynamic power. The power dissipation of bit-lines is directly proportional to the number of registers on the bit-line. Splitting the bit-line into multiple segments helps to reduce dynamic power and the hot registers are mapped into the short bit-line partition. This reduces the dynamic power for accesses to the registers in the hot region. The unused register file region can be put into drowsy state to reduce static power.

This work targets register file power optimization for embedded processors [35]. GPGPUs, nowadays, are used for a wide variety of applications. The pattern of data stored in register file for different types of processors exhibits strong similarity with data used in applications run on GPGPUs. Therefore, this technique may be used for GPGPUs as well. This work [35] provides another technique for reducing power of register file. Since our work focuses on reducing power consumption in caches, this technique can work well with our proposed techniques if this work [35] is ported to GPGPUs.

2.2.12 Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm

Lashgar et al. [36] proposed using a filter-cache to reduce power consumption in instruction caches. The architecture of GPGPUs is based on SIMT (Single Instruction Multiple Thread) model. This means that threads, grouped in warps, execute the same instruction. Since all threads execute the same instruction, therefore, the same instruction is fetched for every warp. Due to this, an instruction that has recently been fetched, has a high probability to be fetched again. This is referred to as “inter-warp instruction temporal locality”. This technique [36] aims to exploit this property to reduce number of accesses to the instruction cache. A small filter-cache is used to cache the instructions that have been fetched before, and helps to reduce number of accesses to the instruction cache. This increases the energy efficiency of the fetch engine by reducing number of accesses to the instruction cache.

This work [36] targets instruction cache for power optimization, while our work focuses on all caches except instruction cache. Therefore, this work [36] is well suited to be integrated with our work to further reduce power consumption of caches in GPGPUs.

2.2.13 Characterizing and improving the use of demand-fetched caches in GPUs

Jia et al. [37] characterized the performance of GPU applications with L1 caches and explored the access patterns and locality of L1 caches. They explore the degree to which performance may either improve or reduce, by turning L1 cache on and off. Their results indicate that, in some cases, L1 caches may hurt performance opposed to common intuition that caches always help to improve performance. In NVIDIA GPUs, L1 caches are not coherent across SMs. This means that global memory store instructions bypass L1 caches. This suggests that turning off caches can help, for applications, where the performance might be reduced by the use of L1 caches. Also, since GPGPUs have inherent latency tolerance mechanism due to massive multi-threading, the primary function of cache is more related to memory bandwidth than latency. They propose a compile-time algorithm to speculate the impact of the cache on performance by using traffic estimates. Based on the speculated performance impact, L1 cache can be enabled or disabled. Although their work targets performance, the proposed technique can also be used to reduce power consumption when L1 caches are not needed.

This work involves optimization techniques that use recompilation of applications for gaining leakage power savings by turning off L1 caches. Our work, however, does not need recompilation of codes and targets both leakage and dynamic power optimization. We also provide power optimizations for all caches including L1 and L2, except instruction cache.

2.2.14 A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures

Since cache block size in GPGPUs is 128 bytes, blocks of 128 bytes are fetched from memory. Applications with high spatial locality can benefit from this coarse grained memory hierarchy. However, since GPGPUs run applications from different domains, not all applications exhibit regular control flow and memory access patterns. For such applications,

coarse grained memory accesses waste energy by fetching unnecessary data. The massive multithreaded architecture of GPGPUs provides little cache capacity per thread. This results in high cache miss rate which may lead to eviction of a cache block before it can be accessed again. This behaviour limits the degree of temporal locality that can be exploited for certain applications. This phenomenon combined with coarse grained only memory hierarchy results in significant over-fetching of off-chip data for irregular applications. This, not only, wastes DRAM power, but also wastes memory bandwidth and on-chip storage. Rhu et al. [38] proposed a Locality-Aware Memory (LAMAR) hierarchy technique to select appropriate memory access granularity for improving energy efficiency. They use a hardware predictor to adaptively adjust the memory access granularity without requiring any intervention from the programmer.

The work of Rhu et al. [38] targets optimization of fetches from global memory to reduce power consumption. Our work is different from their work [38] as we focus on caches for optimizing power consumption. This work [38] can be used along with our techniques to further reduce power in memory hierarchy of GPGPUs.

2.2.15 Power-efficient Computing for Compute-intensive GPGPU Applications

Most of the energy optimization techniques focus on reducing power of register file or memory hierarchy to achieve energy efficiency. S. Zohaib Gilani et al. [39] proposed techniques to optimize power consumption of compute-intensive GPGPU applications. GPGPUs are, typically, optimized for floating-point intensive applications. Due to the diverse nature of applications that can be run on GPGPUs, there are many integer-intensive applications that are also run on GPGPUs. Such applications include, but are not limited to, those for data compression and encryption, medical image processing etc. To utilize the same hardware for both integer and floating point instructions, the floating point fused multiply-add (FMA) units are enhanced to perform integer arithmetic, bitwise, and logical operations. Of the three techniques proposed in the work [39], the first technique involves combining a pair of dependent integer instructions into a composite instruction which can be efficiently executed by an enhanced fused multiply-add unit. The compiler forms these composite

instructions, and consequently, reduces the number of fetched/executed instructions. This improves performance and energy efficiency. The second technique exploits the computational redundancy exhibited by many applications. Computational redundancy occurs when a number of instructions are duplicated across multiple threads and produce the same results. This redundancy attributes to applications having duplicated control instructions across threads in SIMT groups and manipulation of constant values. Memory address calculations can also contribute to such computational redundancy. To exploit this behaviour of redundant computations, the authors suggest executing such instructions on a separate scalar unit. The technique involves dynamically detecting an instruction that produces the same result across all the threads in a warp. This detection is done at runtime, and such an instruction is then issued to a separate scalar pipeline. The source and destination registers for the instruction are kept in a separate Scalar Register File (SRF). The scalar pipeline and scalar register file form the scalar unit. This approach improves power efficiency by avoiding redundant computations. The scalar unit also relieves the SIMT pipeline from executing such redundant instructions and allows it to execute other instructions. This helps to improve the performance of the application. The third technique involves slicing the 32-bit data-path into two 16-bit data-paths. Many instructions do not require the full 32-bit data length to represent their operands, and 16-bit data-path suffices to hold the operands for accurate representation. Thus, using a single 16-bit data-path, reduces the register file access energy. On the other hand, two instructions that need only 16-bit data-path to represent their operands can be issued simultaneously to improve performance.

S. Zohaib Gilani et al. [39] proposed several optimization techniques to reduce power consumption in GPGPUs. Their techniques involve reducing redundancy in computation, and combining a pair of instructions to provide energy-efficient execution. They also propose slicing the data-path to optimize power consumption. Their work [39] focuses on optimizing power for execution pipeline. Our work differs from their work as we target on-chip memory spaces (caches and shared memory) to reduce power consumption. Their work [39] can be used along with our optimization techniques. The two techniques combined can help to achieve greater power savings.

2.2.16 PATS: Pattern Aware Scheduling and Power Gating for GPGPUs

For the SIMT (Single Instruction Multiple Thread) model of GPGPUs, a single instruction is executed concurrently on several execution lanes, sometimes, referred to as SIMT lanes. However, due to branch divergence not all threads are active within a warp during execution. An active mask vector provides the basis for controlling which threads are active in case of branch divergence. For a control structure, the threads having the active-mask bit *set* execute, and the remaining threads stay idle. The threads in the taken and not-taken paths are executed sequentially based on which threads are active in the corresponding path. This leaves the hardware resources, corresponding to inactive threads, to be un-utilized. Qiumin Xu et al. [40] explored the branch divergence patterns for different benchmarks and observed that each benchmark exhibits very few divergence patterns. This means that many warps branch in the same way as other warps in the application. Thus, the hardware resources that are not utilized for these warps remain idle for most of the warps in the program. This can be exploited to gate these hardware units to save leakage power. Their work power gates the SIMT lanes that are not being used during branch divergence. Since there is a power overhead associated with gating of hardware units, the optimum power saving is achieved if the warps exhibiting similar divergence pattern are executed consecutively. The order in which the warps having similar divergence patterns are executed depends on the type of scheduler used. The authors [40] proposed a Pattern Aware Two-level warp Scheduler (PATS) that issues cluster of warps having similar active mask patterns. The pattern aware two-level warp scheduler increases the idle period of SIMT lanes, resulting in an improved power gating efficiency. The authors also proposed a runtime technique to dynamically track the active masks of warps, waiting in the operand collector buffer, which are soon going to be executed. This deterministic look-ahead technique can be used to proactively wakeup an idle execution unit or stop power gating any execution resource that may be needed in the near future. The two techniques are used, in tandem, to improve power gating efficiency.

This work [40] proposes to reduce power consumption for execution units based on branch divergence. We also use branch divergence in our work, but to reduce dynamic power in caches. Our work targets caches for leakage and dynamic power and, inherently, differs from

the work of Qiumin Xu et al. [40]. However, our work and techniques proposed by Qiumin Xu et al. [40] can be used together to further improve energy efficiency in GPGPUs.

2.2.17 Power-Aware L1 and L2 Caches for GPGPUs

This paper is an extension of the previous work [26] on L1 data cache and L2 cache in GPGPUs. In this work [26], static and dynamic power in L1 data cache and L2 cache is reduced by using drowsy mode and divided word line approach. The drowsy mode scheme [26] greedily switches a block to drowsy mode immediately after it is accessed. In this thesis, we propose a novel technique of coarse grained drowsy mode over the drowsy mode technique used in the previous work [26]. We also performed design space exploration to determine the granularity for our new coarse grained drowsy mode. We performed sensitivity analysis in our work to determine the optimal granularity for the region size. In our previous work [26], the performance impact was explored for L1 data cache and L2 cache individually. In this work we enable the optimization techniques for all caches concurrently and report performance for all the caches combined.

We have proposed improvements in our techniques in this work over our previous work [26]. We have included shared memory, texture and constant caches along with L1 data and L2 cache for our proposed techniques in this work. We have also performed more comprehensive analysis of performance as opposed to our previous work [26]. This work builds on our previous work and makes significant contribution in improving the techniques presented in our previous work [26]. Our new technique helps to further improve performance over our previous work [26].

2.3 Overview of previous works and our contribution

Most of the techniques proposed to reduce energy consumption target register file energy [2, 29, 16, 33, 34, 35]. Some of the works [2, 7, 8, 36, 37] target caches to reduce power, while other works [27, 28, 40] target execution units to optimize power consumption. Memory hierarchy has also been optimized [30, 38] to reduce power consumption. Architectural optimizations and optimizing redundant instructions have also been proposed to gain

performance and energy efficiency [39]. Some of the techniques proposed earlier can be used along with our proposed techniques to further optimize energy consumption. Our work targets all levels and types of caches, except instruction cache, in GPPGUs and we apply our proposed power saving techniques to reduce leakage as well as dynamic power. We propose a novel technique of using coarse grained drowsy mode. To the best of our knowledge, this technique has not been used before. Our work adds to previous works to optimize energy consumption and can also be used to gain insight into access patterns of caches in GPGPUs and latency tolerance of caches.

Chapter 3
Motivation and Optimization Techniques

In this chapter, we explain motivation behind our work and demonstrate power saving opportunities in GPGPUs using workloads from different domains. We also present our proposed optimization techniques in this chapter. We use applications from Rodinia [20], Parboil [21] and NVIDIA Computing SDK [18] benchmark suites for characterizing cache access patterns that can be exploited to save power in GPGPU caches. The detail of the experimental framework is discussed in chapter 4.

3.1 Problem Definition

Power dissipation used to be an issue in portable devices, until recently, but is now becoming a significant design constraint in the design of many systems. There are two sources of power dissipation in microprocessor design: static and dynamic. Static power is the leakage power dissipated when a component is idle, whereas, dynamic power is the power consumed whenever a component is accessed. Dynamic power used to be a major contributor to power dissipation in comparison to static power, but the trend of reducing geometrics of semiconductor devices has aggravated the problem of leakage currents. Now static power is rapidly becoming a major source of power dissipation in newer system designs. Voltage scaling has reduced over the years and is no longer contributing significantly in reducing dynamic power. Our work focuses on both sources of power dissipation in caches for GPGPUs and we propose techniques to optimize static and dynamic power consumption. There are different types of L1 caches in GPGPUs namely, L1 data cache, constant cache, texture cache, and instruction cache. The L1 caches are backed by a L2 cache. Shared memory, although technically not a cache, shares a great deal of similarity in architecture with L1 data cache, since the memory space for implementing shared memory is configurable to be partitioned between L1 data cache and shared memory.

In this section, we look at the share of static and dynamic power consumed by different caches in the context of GPGPUs. For static power, we model row-decoder, SRAM cells, and sense amplifier to estimate the static power consumed by each type of cache. Figure 3-1 shows the breakdown of static power consumed by different caches and shared memory.

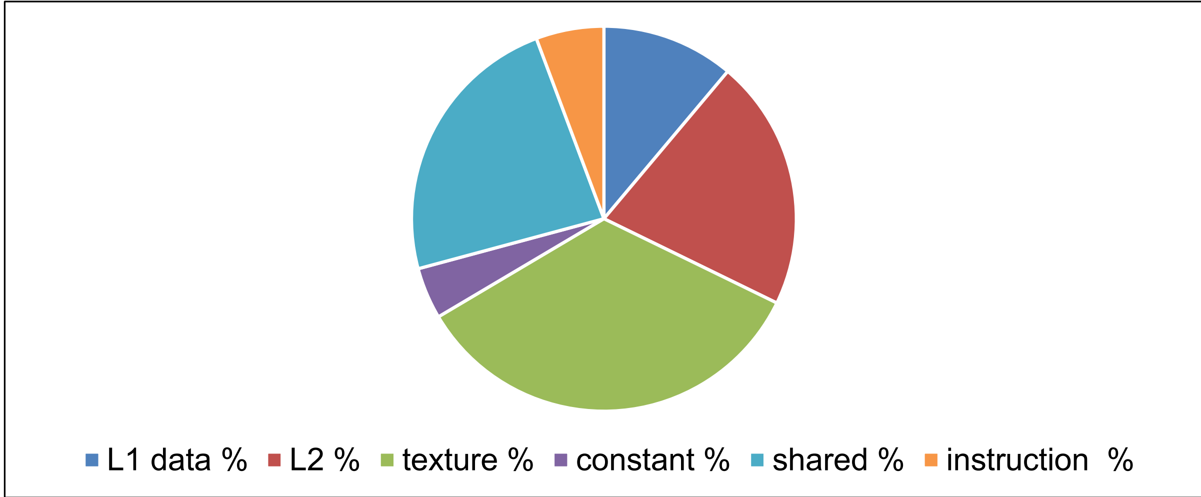


Figure 3-1: Breakdown of static power for different memory spaces.

Texture cache consumes the most static power at 34.3%, followed by shared memory consuming more than 23%. L1 data cache and L2 cache consume 11% and 21% of the static power, respectively. Constant and instruction cache consume significantly less power as compared to other caches, with constant cache consuming 4% and instruction cache consuming 5% of the total static power consumed by all caches. We used GPUWattch [41] to estimate dynamic power of caches. Figure 3-2 illustrates the percentage of total GPU dynamic power consumed by caches for different benchmarks. The last bar shows the average dynamic power share of all caches and shared memory for all the benchmarks.

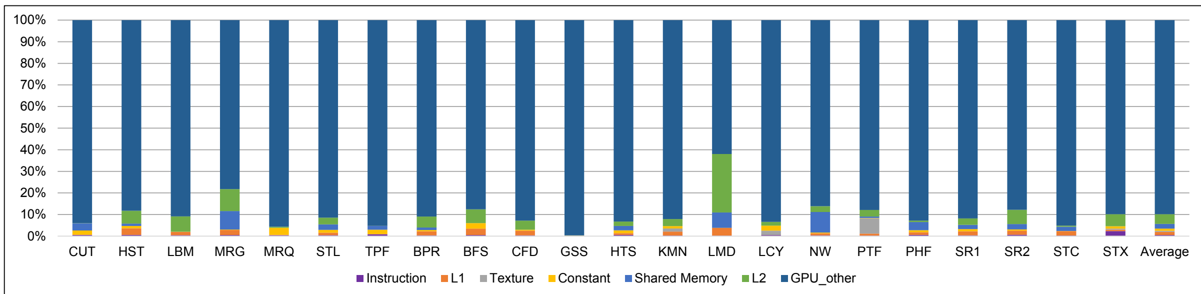


Figure 3-2: Breakdown of cache dynamic power as percentage of total GPU dynamic power.

On average, more than 10% of total dynamic power is consumed by caches with some benchmarks, i.e. mri-gridding and lavaMD, consuming more than 21% and 38%, respectively. On average, L2 cache has the largest share of dynamic power of 4.5%. L1 data cache and shared memory, on average, consume 1.4% and 2.2%, respectively. Constant and instruction caches consume 1.02% and 0.53% of total GPU dynamic power, respectively. We also evaluate the breakdown of dynamic power for individual caches. On average, L2 cache contributes 45.5% to dynamic power consumed by all caches. L1 data cache contributes 14%, shared memory 20.5%, while constant and instruction caches contribute 9.5% and 5.1%, respectively, of total dynamic power consumed by caches. Figure 3-3 shows the breakdown of total power, static and dynamic, consumed by individual caches. The last bar shows the average total power share of all caches and shared memory for all the benchmarks.

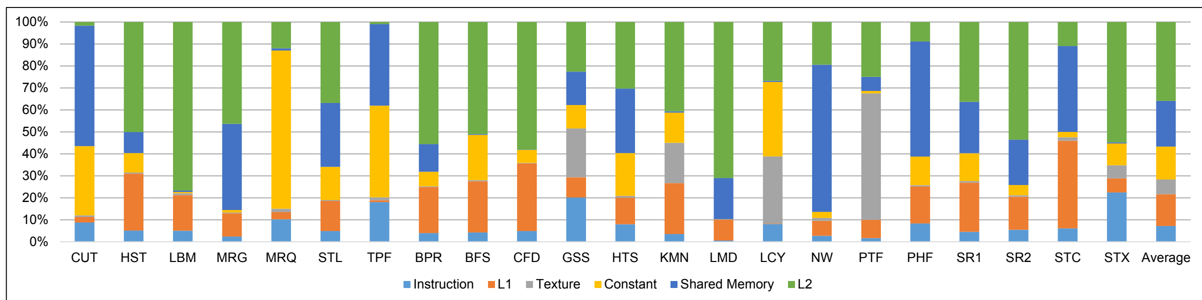


Figure 3-3: Breakdown of total (static + dynamic) power consumed by cache memories.

On average, L2 cache consumes more than 35% of total cache power while L1 data cache consumes more than 14%. Shared memory, on average, consumes around 20.8% of total cache power, whereas texture cache consumes 6.7%. Constant and instruction caches, on average, consume 14.9% and 7.3%, respectively. Our analysis shows that cache memories consume a significant percentage of power in GPGPUs and optimizing static and dynamic power in caches can help to achieve significant power savings in GPGPU architectures.

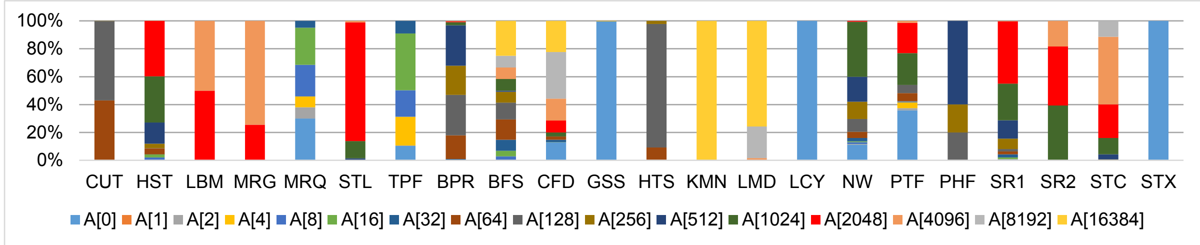
3.2 Cache Access Patterns

Next, we look at the opportunities in caches for optimizing static and dynamic power. The pattern of cache accesses in GPGPUs is unique and can be exploited to save static and dynamic power in cache blocks. For opportunities to reduce static power, we analyze access

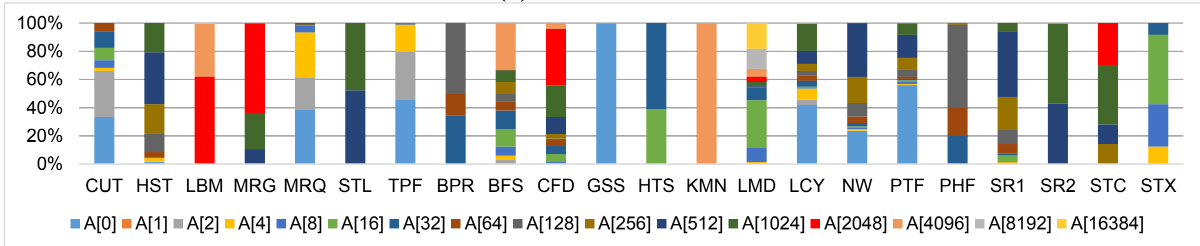
patterns of cache blocks by analyzing two parameters: number of accesses to cache blocks and the inter-access delay for cache blocks. These two parameters play an important role in defining the access pattern of caches.

Graphs in figure 3-4 show breakdown of accesses to the cache blocks in L1 data cache, L2 cache, constant cache, texture cache, shared memory and instruction cache. Each bar in the graph is divided into 16 sections by using a logarithmic scale to categorize the number of accesses to cache blocks. The bottom most component labelled A0 shows number of blocks that are not accessed by any SM, A1 component shows the number of blocks accessed one time. Similarly, the last component A16384 shows the number of blocks that are accessed 16384 times or more. From the graphs in figure 3-4, we observe that in L1 data cache, more than 25% of cache blocks are accessed less than 32 times. In mri-q, gaussian, leukocyte and tpacf, more than 90% of cache blocks in L1 data cache are accessed less than 32 times. In gaussian and leukocyte, more than 99% of cache blocks in L1 data cache are never used for execution. In L2 cache, more than 26% of cache blocks are accessed less than 16 times. Since most of memory requests are serviced by L1 caches, cache blocks in L2 are idle more often. In mri-q, tpacf and gaussian, more than 96% of cache blocks in L2 cache are accessed less than 32 times. In cutcp, mri-q, tpacf, gaussian, leukocyte and particlefilter, more than 33% of cache blocks in L2 cache are never used for execution. In constant caches, 89% of cache blocks are accessed less than 2 times. In all benchmarks except cutcp, mri-q and leukocyte, more than 92% of cache blocks in constant cache are never used for execution. Since texture cache is only used for graphical applications, therefore, not all benchmarks use texture cache. Only kmeans, leukocyte, particlefilter and simpleTexture use texture cache. In kmeans and particlefilter, 50% and 94% of cache blocks in texture cache are never used for execution. The cache blocks in other benchmarks that do not use texture cache are never accessed. Some benchmarks, namely lbm, mri-q, bfs, cfd, gaussian, kmeans, leukocyte and simpleTexture, do not use shared memory and the memory blocks in shared memory are never accessed for these benchmarks. For those benchmarks that use shared memory, more than 89% of memory blocks in shared memory are accessed less than 4 times. For all the benchmarks, more than 93% of memory blocks in shared memory are never accessed. In

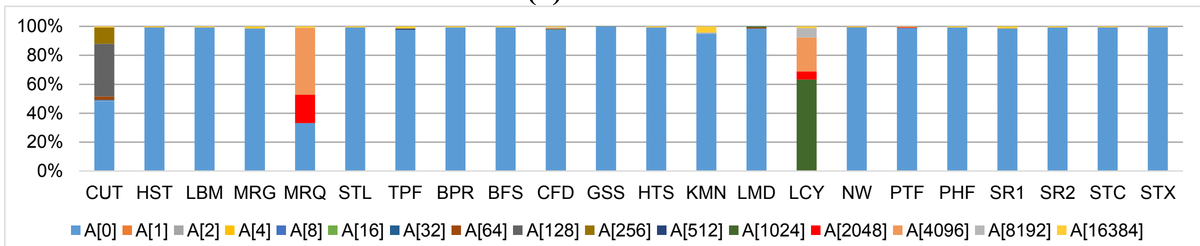
instruction cache, more than 27% of cache blocks are accessed less than 16 times. Around 27% of cache blocks in instruction cache are never used for execution.



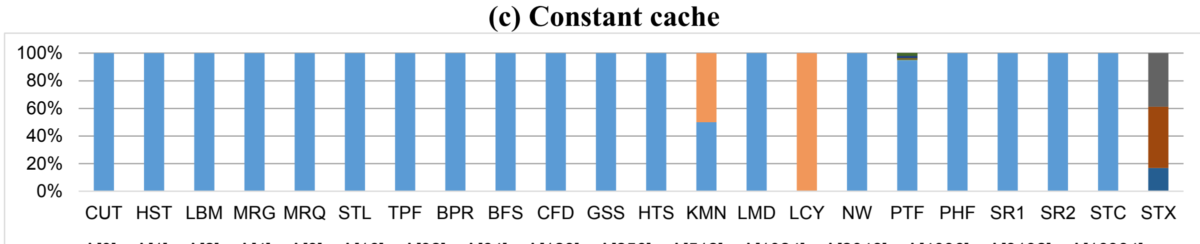
(a) L1 data cache



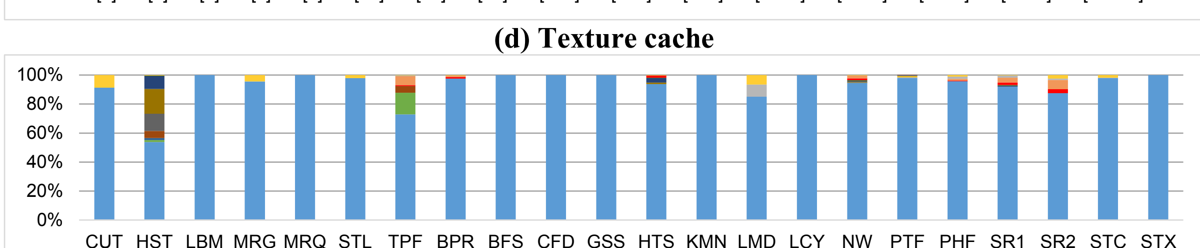
(b) L2 cache



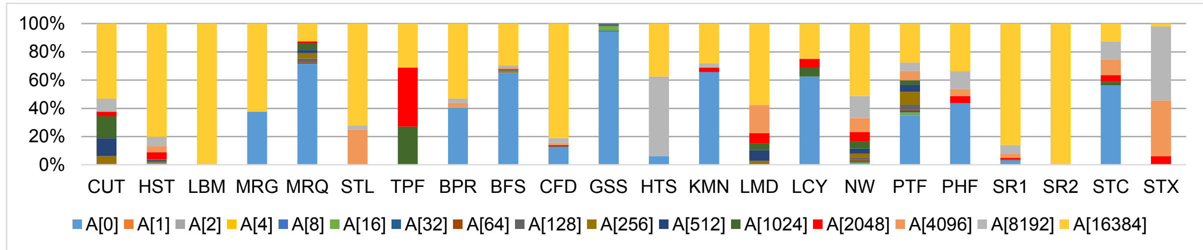
(c) Constant cache



(d) Texture cache



(e) Shared memory



(f) Instruction cache

Figure 3-4: Breakdown of accesses to cache blocks.

This shows that for different types of caches and shared memory in GPGPUs, there is a significant percentage of cache blocks that are never accessed. This means that these blocks remain idle throughout the execution of the program and can be put into drowsy mode to reduce leakage power. Other than the cache blocks that are never accessed during execution, a large fraction of the cache blocks are accessed only a few times, providing an opportunity to put these cache blocks in drowsy state to gain power saving.

Number of accesses to cache blocks play an important role in evaluating the access pattern of cache blocks for putting them into drowsy mode. However, the access pattern is not entirely dependent on number of cache accesses. It is also, greatly, impacted by inter-access cycle of cache blocks. Inter-access cycle of a cache block is the number of cycles between two consecutive accesses to the cache block. If the inter-access time is very small then the cache blocks cannot be put into drowsy mode as they would have to be switched back to ON state immediately after being put into drowsy state. In such a scenario, there would not be any considerable power saving because the only time leakage power can be reduced using drowsy mode is when the cache blocks are idle. It is, thus, necessary to have a significant inter-access delay for the cache blocks to be able to save power by putting the cache blocks in drowsy mode.

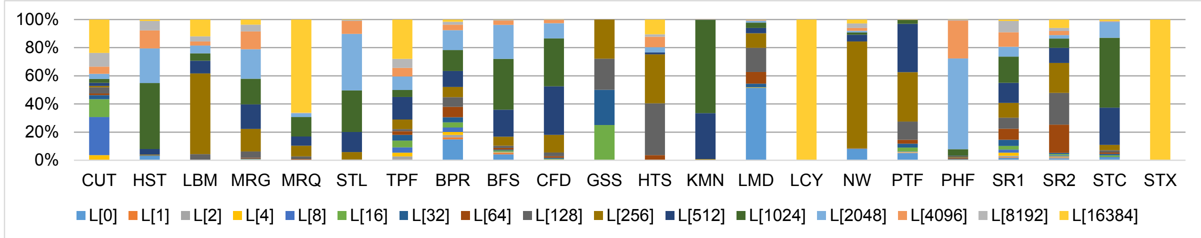
In the two-level scheduler, a warp, after being executed has to wait for all the other warps in the active list to be scheduled, before it can be executed again. The only time when the scheduler schedules the same warp to be executed, consecutively, is when there are no other warps in the active list. This scenario, although possible, occurs quite rarely and hence there is a gap between two executions of the same warp. This inter-access delay provides the

opportunity to put the cache blocks into drowsy mode immediately after they have been accessed. This can be used to reduce leakage power.

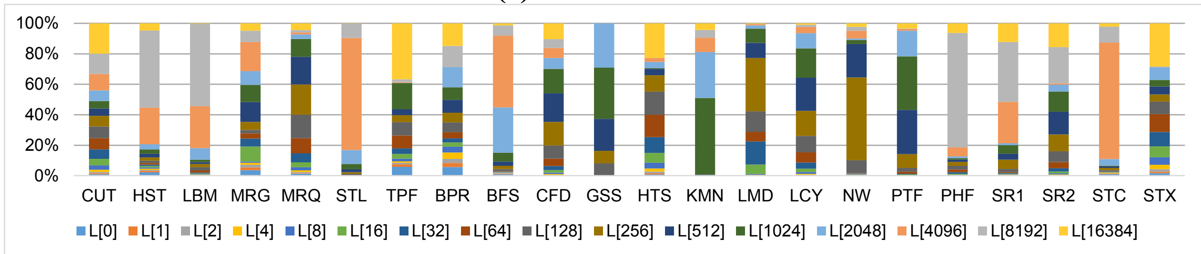
We measure the number of cycles elapsed between two consecutive accesses to the same cache block to analyze inter-access delay. Graphs in figure 3-5 show the breakdown of inter-access cycles for cache blocks in L1 data, L2, constant, texture, shared and instruction caches. In L1 caches, more than 63% of cache blocks have inter-access cycles of 512 or more. The average inter-access cycle for L1 caches is 3578-cycles. More than 69% of cache blocks in L2 cache have inter-access cycles of 512 or more, and the average inter-access cycle is 3702-cycles. In constant caches, more than 26% of cache blocks have average inter-access cycles of 32 or more. The average inter-access cycle, in constant caches, is 161 cycles. Amongst the benchmarks that access texture cache, kmeans has 99%, leukocyte has 98%, particlefilter has 19% and simpleTexture has 74%, of cache blocks that have inter-access cycles of 32 or more. On an average, in benchmarks that use texture cache, 73% of cache blocks have inter-access cycles of 32 or more. The average inter-access cycles for these benchmarks, in texture caches, is 3249-cycles. For benchmarks that access shared memory, more than 51% of shared memory blocks have inter-access cycles of 128 or more. For these benchmarks, the average inter-access cycle is 1054-cycles in shared memory. In instruction caches, more than 12% of cache blocks have inter-access cycles of 16 or more. The average inter-access cycles for cache blocks, in instruction caches, is 23-cycles. The breakdown of inter-access cycles for different type of caches demonstrates that the cache blocks in these caches remain idle for a significant number of cycles before they are accessed again. This behaviour of access pattern in cache blocks can be exploited to put them into drowsy mode to reduce static (or leakage) power.

Next, we look at opportunities to optimize dynamic power in cache blocks. GPGPUs execute threads in granularity of warps. Instructions are executed in a lock-step manner for the 32 threads in a warp. A fully utilized warp has all the 32 threads active and executes the same instruction at any given time. Most of the graphics applications make use of all the 32 threads in a warp to execute instructions. However, general purpose applications exhibit more complex control flow behaviour due to frequent branch instructions. Conditional

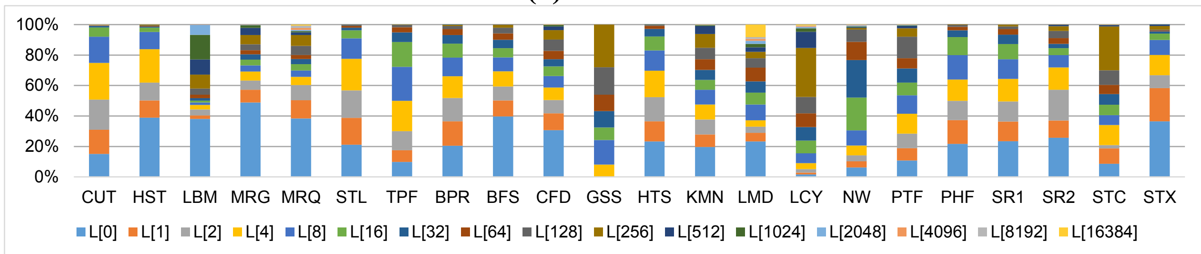
branch instructions may cause threads, within a warp, to take different paths. This phenomenon is known as branch divergence.



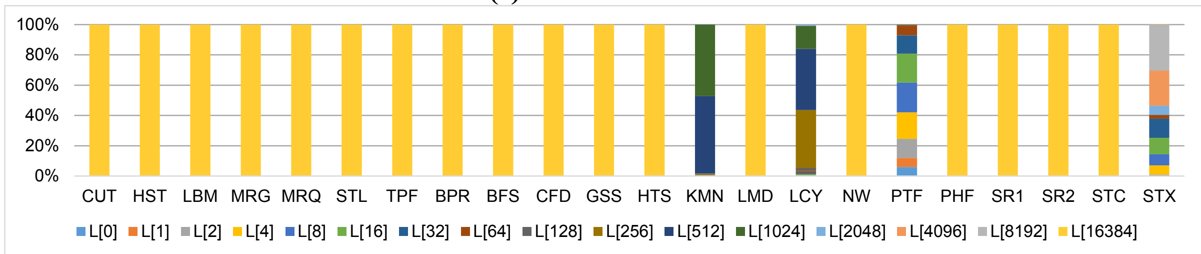
(a) L1 data cache



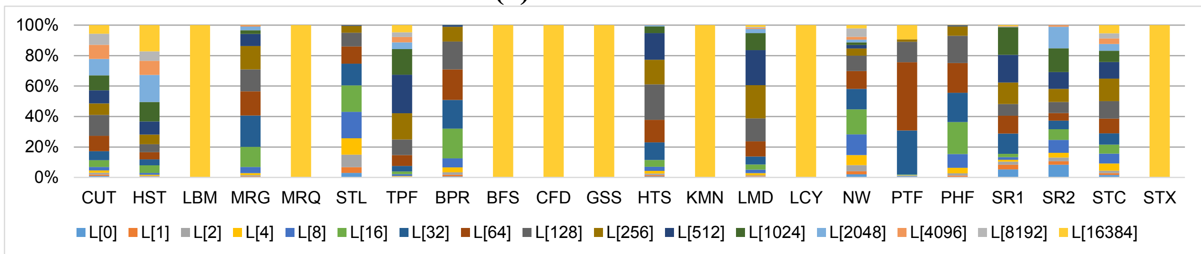
(b) L2 cache



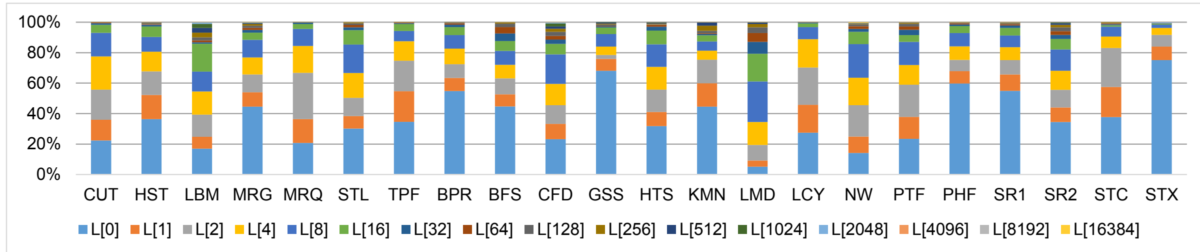
(c) Constant cache



(d) Texture cache



(e) Shared memory



(f) Instruction cache

Figure 3-5: Breakdown of inter-access cycles to cache blocks.

The architecture of GPGPUs provides only one active program counter (PC) for a warp during execution. In case of control instructions, for example an if-then-else construct, there are usually different instructions to be executed for the taken and not-taken paths. To execute these instructions, a warp has to make two passes over the divergent paths. The two passes are sequential to each other, and only the threads corresponding to that particular path are active during each pass. The other threads remain idle during the pass. However, in existing GPGPU implementations, cache blocks for all 32 threads are accessed even though we may have only a few active threads and the remaining threads, in a warp, may be idle. Different programs exhibit different level of branch divergence due to the nature of each program. We investigated branch divergence occurring in benchmarks that we used for our analysis of cache access patterns. For our analysis, we recorded the number of active threads whenever accesses to cache blocks are requested. Table 3-1 shows the percentage of active threads within the warps that access corresponding cache blocks for different type of caches and shared memory. Block utilization value of ‘0’ for a cache signifies that the benchmark does not make use of that particular type of cache. Since architecture of GPGPUs is based on SIMT programming model, all threads within a warp execute the same instruction. This means that the whole cache block in instruction cache is accessed for fetching each instruction, regardless of the branch divergence that may occur during execution of that instruction. Hence, block utilization for instruction cache is not reported in Table 3-1 as it is always 100%. It is important to note that L2 cache is accessed when a miss occurs in any of the L1 caches including data and texture caches. This is the main reason that block utilization in L2 cache is lower than block utilization in L1 data caches. While some benchmarks, for example kmeans and mri-q, have 32 active threads in all warps throughout the entire

execution, other benchmarks such as gaussian and bfs, have very low cache block utilizations. Since, in the baseline architecture, the whole cache block is accessed even for programs with low warp utilization, significant dynamic power is wasted. We can reduce dynamic power in caches by avoiding these unnecessary accesses to cache blocks.

Table 3-1: GPGPU Benchmarks and warp utilization

Benchmarks	Abbreviation	Block Utilization (%)				
		L1	L2	Shared	Constant	Texture
Cutcp	CUT	99.9%	99.5%	94.1%	93.7%	0.0%
Histo	HST	99.9%	98.2%	68.6%	99.9%	0.0%
Lbm	LBM	100.0%	90.9%	0.0%	93.8%	0.0%
mri-gridding	MRG	100.0%	99.2%	90.5%	99.5%	0.0%
mri-q	MRQ	100.0%	100.0%	0.0%	100.0%	0.0%
stencil	STL	100.0%	91.3%	99.3%	84.3%	0.0%
tpacf	TPF	97.2%	78.5%	96.6%	100.0%	0.0%
backprop	BPR	96.8%	89.8%	62.9%	71.3%	0.0%
Bfs	BFS	81.1%	67.7%	0.0%	48.0%	0.0%
cfd	CFD	99.7%	92.9%	0.0%	98.5%	0.0%
gaussian	GSS	41.8%	36.3%	0.0%	14.0%	0.0%
hotspot	HTS	100.0%	96.2%	78.1%	96.6%	0.0%
kmeans	KMN	100.0%	100.0%	0.0%	100.0%	100.0%
lavaMD	LMD	98.6%	96.6%	78.5%	80.6%	0.0%
leukocyte	LCY	0.0%	93.4%	0.0%	92.2%	94.3%
nw	NW	95.6%	97.2%	30.0%	28.7%	0.0%
particlefilter	PTF	98.2%	98.4%	86.5%	88.4%	92.9%
pathfinder	PHF	99.8%	97.8%	91.9%	92.8%	0.0%
srad_v1	SR1	99.4%	99.2%	21.1%	99.1%	0.0%
srad_v2	SR2	100.0%	95.5%	66.0%	100.0%	0.0%
streamcluster	STC	98.3%	95.8%	94.3%	30.8%	0.0%
simpleTexture	STX	0.0%	84.0%	0.0%	100.0%	100.0%

3.3 Power Reduction Techniques

The unique access patterns of caches in GPGPUs present opportunities for power reduction. We describe the techniques to reduce static and dynamic power in this section.

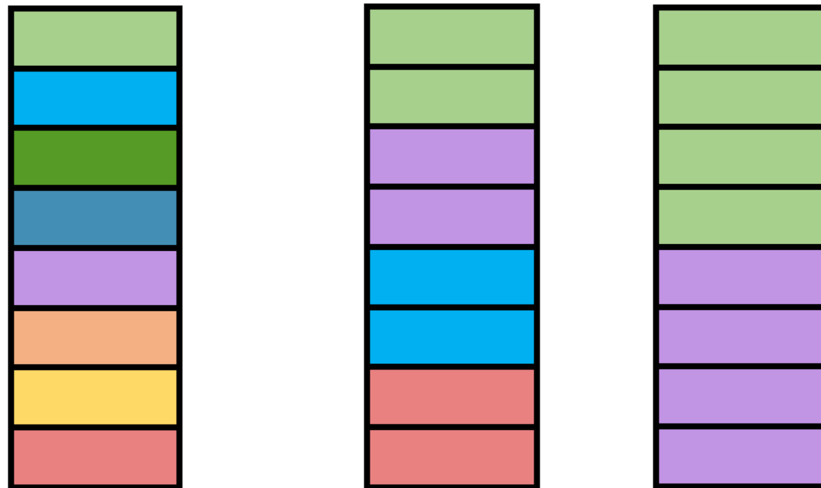
3.3.1 Static Power Reduction Using Drowsy Cells

In the last section, we discussed the inter-access pattern of cache blocks. Due to the scheduling policy [17], a warp has to wait after execution of all the other warps in the active list before it can be executed again. During this long inter-access cycle delay, the cache blocks are idle. This property of GPGPUs presents an opportunity to reduce leakage power for cache cells when they are idle. Several techniques have been proposed to reduce leakage power of cache cells by turning off cache blocks when they are not needed [1, 22]. The downside of these techniques is that the data in the cache blocks are lost when they are turned off and the extra power needed to access interconnection network and lower level cache or global memory erodes any power saving achieved by turning off cache cells. In addition, extra power is needed to re-load the data in the cache cells. This also results in increased latency to access the requested data, hence, reducing overall performance. These techniques are not suitable to our problem and, therefore, we use another technique of putting cache blocks into drowsy mode [8]. Dynamic voltage scaling is employed in a drowsy cell to reduce leakage power. Each cache block can be switched between high and low (drowsy) supply voltages. During access to the cache block, the voltage of SRAM cells is raised to nominal voltage. Following each access, the voltage is reduced to save leakage power. Due to short-channel effects in deep-submicron processes, leakage current reduces considerably in idle cache blocks. Since power is the product of voltage and current, reduced leakage current and low voltage, dramatically, reduce leakage power. Whenever a cache access request is generated by an SM, the voltage of the cache line is checked by the cache controller. The voltage level of the cache line is checked in parallel with the read and comparison of the cache tag; therefore, it does not incur any extra delay. If the cache line is in normal mode (high voltage), it can be accessed without any sacrifice in performance as there is no delay. However, in the case of a cache block being in drowsy mode, its voltage has to be raised to the normal voltage before the data on the cache line can be accessed. It is imperative to raise the voltage before accessing the cache block to prevent discharging bit-line of the cache line as it may read out wrong data if the voltage is low. However, it needs a certain amount of time to raise the voltage of the cache line. Therefore, we need to wait for

the supply voltage to switch to normal mode before the data in the cache line can be accessed. This extra time to raise the voltage of the cache line can be referred to as wake-up latency. Since we have to wait a certain amount of time before the voltage of the cache line is raised to normal and the cache line can be accessed, it raises a concern that the execution time of applications may increase due to this delay. We can effectively hide this wake-up latency by raising the voltage of the cache line to be accessed in near future. However, to do this, we need to devise a scheme to know which cache block is going to be accessed in future. We use a two-level scheduler [17] to select a warp for execution. The scheduler selects a ready warp from the active list and sends it for execution. This process is repeated every cycle. To hide wake-up latency of drowsy cells, the scheduler should send the source operands of a load/store instruction to the memory unit before the instruction is issued for execution. To accomplish this, the scheduler can issue a warp and look into active list, in parallel, for the warp that is going to be issued in the next cycle. In this way, the overhead of drowsy cells with 1-cycle delay can be effectively hidden. Similarly, the scheduler can look into active list and send information of the warp to the memory unit n cycles ahead and may wake-up drowsy cells to hide n cycles of wake-up delay. In this way, the two level scheduler is able to hide the latency associated with drowsy cells.

In our work, we also explore the scenario of using a scheduler that is not able to hide the latency associated with the wake-up delay of drowsy cells. For a scheduler that is not able to hide the wake-up latency of drowsy cells, we propose coarse grained drowsy mode scheme in which we partition cache blocks into regions of contiguous locations. We switch cache blocks between drowsy and ON states, in granularity of cache regions (partitions of cache blocks) instead of individual cache blocks. The last accessed region is kept ON, while all the other regions are put into drowsy mode. Figure 3-6 shows granularity of 1, 2 and 4 cache blocks in a cache region for a cache having 8 rows. Cache regions have been highlighted by using same colour for the cache blocks in a region. This means that contiguous cache blocks having the same colour constitute a cache region. If one of the cache blocks in the region is accessed, all the cache blocks in the region are switched on. This approach of partitioning cache blocks into cache regions improves performance of workloads that exhibit spatial and

temporal locality in cache accesses, since no wake-up delay is incurred for accesses within a cache region. In chapter 5, we evaluate the performance impact of drowsy cells having different wake-up latencies assuming that it is not feasible to hide the latency of drowsy cells. We also analyze the sensitivity of performance to region size to optimize for maximum power saving while keeping performance impact marginal.



a). granularity = 1 b). granularity = 2 c). granularity = 4

Figure 3-6: Different granularities for a cache with 8 rows.

3.3.2 Reducing Dynamic Power Using Active Mask

To reduce leakage power, we propose using drowsy cells for cache blocks when they are idle. This technique saves the static power associated with cache lines. To reduce dynamic power we employ another technique that we discuss in this section. In Fermi family [9], cache blocks are 128-byte wide. Whenever an SM executes a load/store instruction, the whole 128-byte cache block is accessed. Using the drowsy cells, all 128 bytes are switched to ON state for a cache block whenever it is accessed. Accessing such a large number of SRAM cells incurs significant dynamic power because of activating word-lines, bit-lines and sense amplifiers.

The GPGPUs utilize a SIMT model and all threads (within a warp) execute in a lock-step manner. Since all threads execute the same instruction, control instructions in a program result in divergence of threads. The branch instructions are executed in two phases. In the

first phase, the threads in the taken path are active and the rest of the threads in a warp are idle. In the second phase, threads in the not-taken path are active and the rest are idle. However, in the baseline scheme, a warp with partial utilization still activates the complete cache line. This implies that Word-Line (WL), Bit-Line (BL), Bit-Line-Bar (BLB) and sense amplifiers are pre-charged for the whole cache line even though only a subset of the cache line is used for warp execution. This presents an opportunity to reduce dynamic power by accessing only the portion of cache blocks that correspond to active threads. In our evaluation of a variety of benchmarks, as shown in Table 3-1, we see that the percentage of active threads varies across the benchmarks. This is due to branch divergence. The technique of accessing portion of cache lines corresponding to active threads can help to reduce dynamic energy for benchmarks where not all threads are active at the same time. We can use the active mask in GPUs, which identifies active threads within a warp, to detect the number of active threads. The active mask is a vector of 32 bits with each bit corresponding to a single thread within a warp. When a branch instruction diverges, the bits corresponding to active threads are set and the rest are cleared. This vector, thus, provides the basis for detection of inactive threads and can be used to disable portions of cache blocks associated with inactive threads.

We use the Divided Word Line (DWL) [23] technique to implement active *mask-aware* access to caches. The structure of a Divided Word Line is illustrated in figure 3-7. In DWL, the Word Line (WL) is segmented into several Small WLs (SWLs). Each SWL enables or disables access to the portion of cache block attached to it. We segment the Word Line such that each SWL covers 4-byte of the cache block. The output of a row decoder is connected to SWLs. Each SWL has to be enabled or disabled based on which portion of the cache block needs to be accessed. GPU baseline architecture provides the necessary control signals, making the integration of DWL into caches rather simpler. Every warp's active mask contains the bit-vector that can be used to specify portions of the cache block that need to be enabled or disabled based on the thread being active or inactive. This mask can be used in conjunction with the row decoder to enable the corresponding cache portion within the cache line. The output of row decoder is fed into an "AND" gate which has its other input coming

from the active mask. The actual implementation of this technique is illustrated in figure 3-7. Each SWL is activated by an AND gate having two inputs, one coming from the row decoder (the horizontal line) and the other coming from the active mask (the vertical line). DWL reduces dynamic power since whenever a cache block is accessed, the bytes corresponding to the inactive cells within the cache block are disabled.

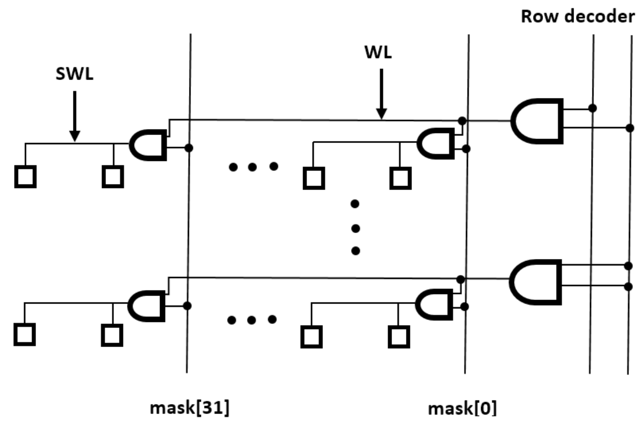


Figure 3-7: Structure of DWL.

Chapter 4
Methodology and Results

In this chapter, we explain our experimental framework and evaluate power savings achieved by using our proposed optimization techniques. We used GPGPU-Sim (version 3.2.1) [3] to evaluate our power-aware optimization techniques. GPGPU-Sim is an open-source, detailed cycle-accurate simulator for GPGPUs. We configure the simulator to closely match NVIDIA's Fermi GTX480 as recommended in the GPGPU-Sim manual (Table 4-1). We use a collection of benchmarks from CUDA Computing SDK [18], Rodinia Benchmark suite [20] and Parboil Benchmark suite [21]. The benchmarks are listed in Table 3-1. We ran the benchmarks until completion or for 1 billion instructions, whichever comes first.

Table 4-1: GPGPU-Sim Configuration

Number of SMs	16
Warps/Shader	48
Threads per Warp	32
PEs/SM	32
Registers per core	32768
$L1 data cache (size/assoc/line)	16KB/4-way/128B
$L2 (size/assoc/line)	768KB/16-way/128B
$Constant (size/assoc/line)	8KB/2-way/64B
$Texture (size/assoc/line)	12KB/24-way/128B
Shared Memory (size/assoc/line)	48KB/4-way/128B
$Instruction (size/assoc/line)	2KB/4-way/128B

4.1 Experimental Results

In this section, we report the power savings achieved in L1 caches (data, texture and constant), shared memory and L2 cache. We also evaluate the performance impact in case of using a scheduler that is not able to hide the wake-up latency of drowsy cells.

As discussed in Section 3.3.1, a two level scheduler can activate a cache block ahead of time and avoid any penalty due to wake-up latency. However, if the GPGPU uses a scheduler other than a two level scheduler, it might not be possible to hide the wake-up latency. We also investigate such a scenario and assume that we are unable to hide wake-up latency

causing delay in execution of warps. To take into account the effect of wake-up latency, we ran the benchmarks with one and two extra cycles overhead. These extra cycles are sufficient to switch cache cells from drowsy mode to normal mode [8]. It should also be noted that these latencies are in addition to the latency of the baseline cache. For the scheduler that is not able to hide the wake-up latency, we propose coarse grained drowsy scheme. We partition contiguous cache blocks into regions and switch between ON and drowsy in granularity of cache regions. After a cache block is accessed in a cache region, it is kept ON till the time another cache block, in another region, is requested for access. This improves the performance for workloads exhibiting spatial and temporal locality in cache access patterns.

We ran the benchmarks with granularity of 1, 2, 4, 8, 16 and 32 cache blocks for region size and analyzed performance and power saving. The optimal performance and power saving was achieved for granularity of 16 cache blocks in a region. Figure 4-1 shows the performance of drowsy caches with granularity of 16 cache blocks in a cache region relative to the baseline scheme. The first and second bars show one and two-cycle wake-up delay, respectively. Bars less than one show slow-down. A GPGPU has many warps and if a warp is stalled due to cache delay, the GPGPU can issue and execute another warp. Hence, the performance changes slightly with wake-up delay.

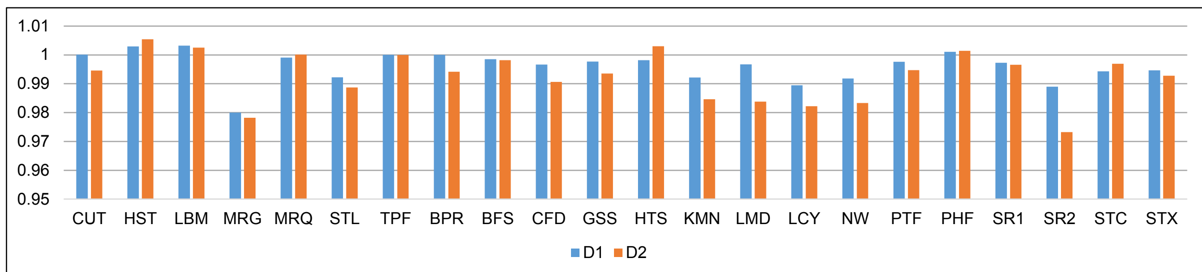
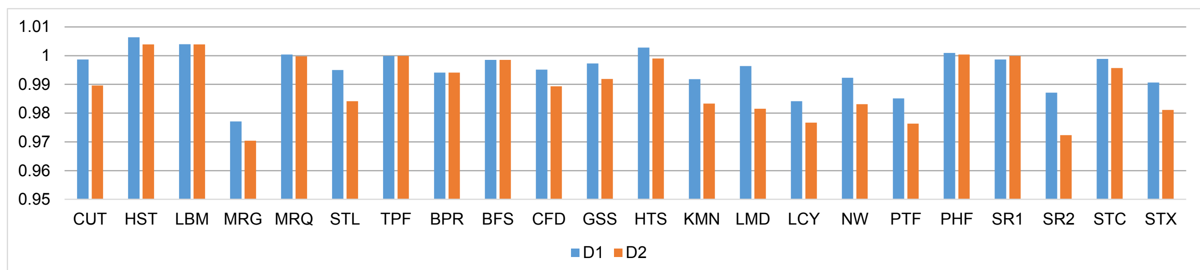


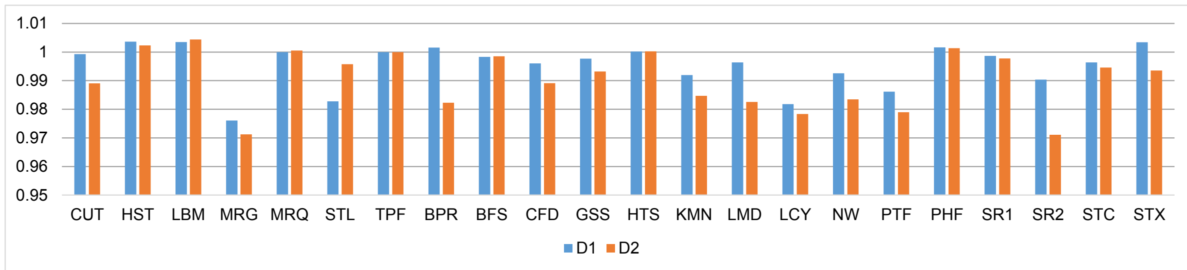
Figure 4-1: Performance impact with region size of 16 for one and two cycles wake-up delay.

We observed that for granularity of 1 cache block in a region, the performance of only two benchmarks was significantly affected. This indicates that most of the benchmarks are tolerant to using drowsy mode for region size of 1 except only two of these benchmarks. We analyzed these two benchmarks and found that the performance impact, for region size of 1, was due to instruction cache wake-up delay. Increasing the granularity of region size to 16 improved the performance for the two benchmarks since all cache blocks are turned ON in

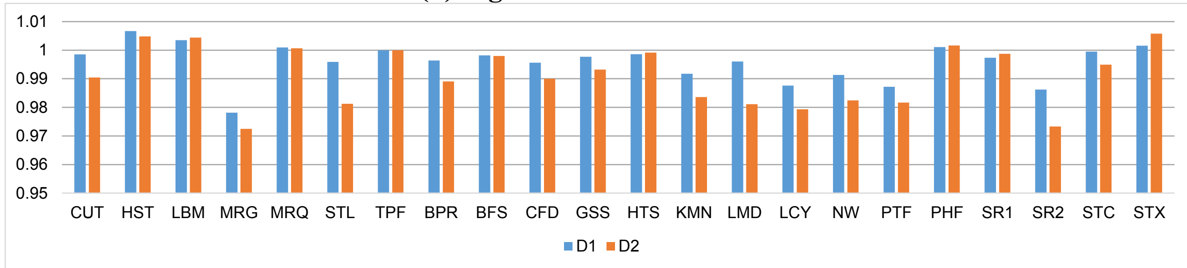
instruction cache at this granularity. According to our analysis, in chapter 3, the power share for instruction cache is only 5% of the total leakage power of all caches. Therefore, not using instruction cache does not affect the power saving significantly, but, it does help to ensure that performance is not affected adversely for any type of workload. We want our optimization techniques to be general for all benchmarks and do not want to sacrifice performance for even the rare cases. Therefore, we do not use instruction cache in our results for power and performance. We performed sensitivity analysis on the region size for all other caches, without enabling instruction cache, to determine the optimum region size for our coarse grained drowsy scheme. We performed sensitivity analysis for granularity of 1, 2, 4, 8, 16 and 32 cache blocks for region size. Figure 4-2 shows performance of drowsy cache relative to the baseline scheme for coarse granularity of 1, 2, 4, 8, 16 and 32 cache blocks in a cache region, without enabling instruction cache. We observed optimal performance and power saving for granularity of 1 cache block in a region. There is a slow-down of less than 0.5%, on average, for drowsy cell wake-up delay for coarse granularity of 1 cache block. Our sensitivity analysis shows that increasing the region size results in an increase in overall performance. For region size of 32, the slow-down is not more than 0.26%, on average. Since using a larger region size erodes power saving, and the performance gain is not significant to justify using a larger granularity, we use coarse granularity of 1 cache block for the region size. In some benchmarks the performance is improved, over baseline, by using our coarse grained drowsy scheme. We analyzed these benchmarks and found that the sequence of executed warps changes in our scheme which reduces cache miss rates and improves performance of these benchmarks.



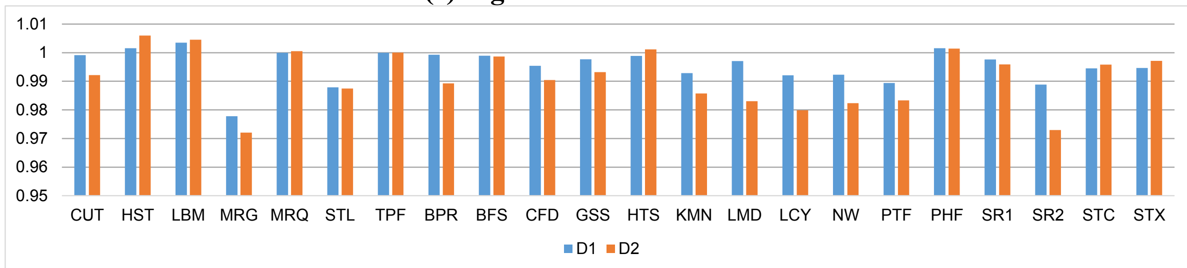
(a) region size=1 cache block



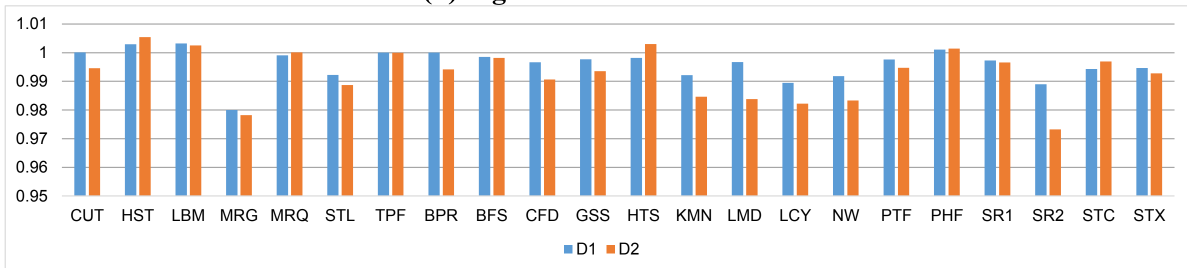
(b) region size=2 cache block



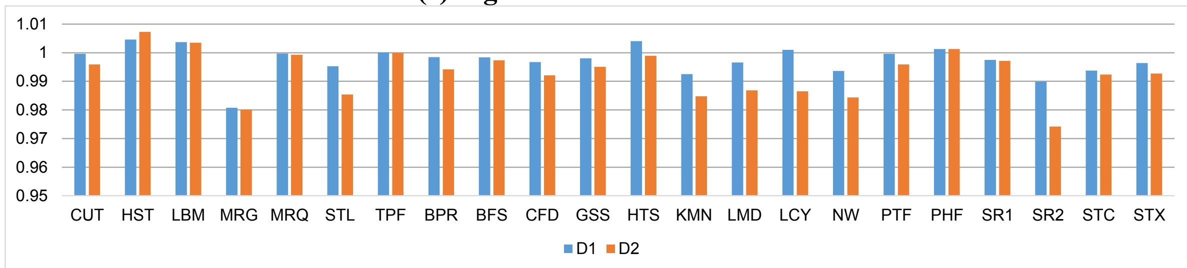
(c) region size=4 cache block



(d) region size=8 cache block



(e) region size=16 cache block



(f) region size=32 cache block

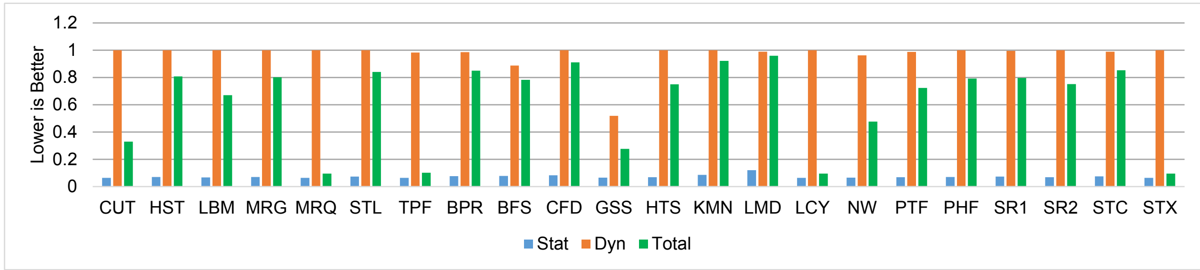
Figure 4-2: Performance impact when region size changes for one and two cycles wake-up delay.

For evaluating static power in caches, we used HSPICE to model our caches based on 6T SRAM cells. We use the technology files from Predictive Technology Models (PTM) [12] with feature size of 32-nm and nominal voltage of 0.9V. Using our model, we were able to quantify the leakage current in caches at different levels of voltage. Table 4-2 lists static power for nominal voltage and reduced voltages for different types of caches. We found that the state of SRAM cells can be maintained even if V_{dd} is reduced to 0.2V. In ideal scenario, a drowsy cell can work at 0.2V, but for real life applications it is imperative to add safety margin to take into account noise and also mismatch between transistors. From Table 4-2, we can see that even when V_{dd} is reduced to 0.4V, the static power is less than 6% of static power when cache cells operate at full V_{dd}. Having a higher than 0.2V V_{dd} also helps to reduce the time it takes to raise the voltage level of cache cells from drowsy to nominal voltage. We evaluate and report our results in this section assuming drowsy cells operate at 0.4V.

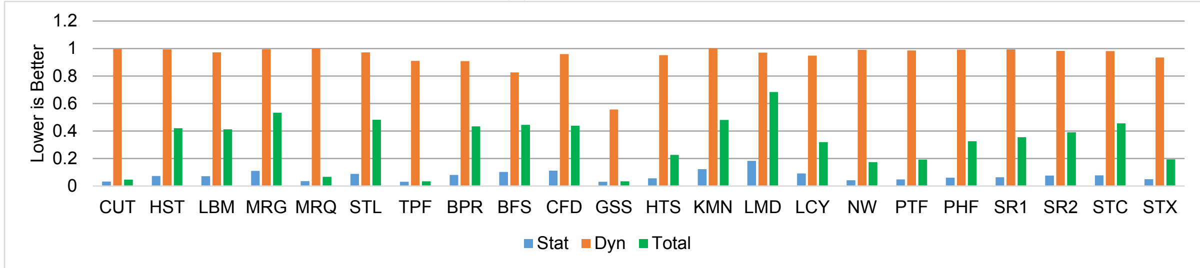
Table 4-2: Static Power of Caches

Cache		**Vdd**			
		0.2	**0.3**	**0.4**	**0.9**
L1 data cache	Static Power	0.01988	0.03484	0.05956	1.05984
	% Saving	98.1	96.7	94.4	NA
L2	Static Power	0.15952	0.42464	0.9552	30.128
	% Saving	99.5	98.6	96.8	NA
Shared memory	Static Power	0.01084	0.029	0.06516	2.236
	% Saving	99.5	98.7	97.1	NA
Texture Cache	Static Power	0.0096	0.02448	0.05448	3.264
	% Saving	99.7	99.3	98.3	NA
Constant Cache	Static Power	0.0019	0.00496	0.01128	0.412
	% Saving	99.5	98.8	97.3	NA
Instruction Cache	Static Power	0.0016	0.00408	0.00908	0.544
	% Saving	99.7	99.3	98.3	NA

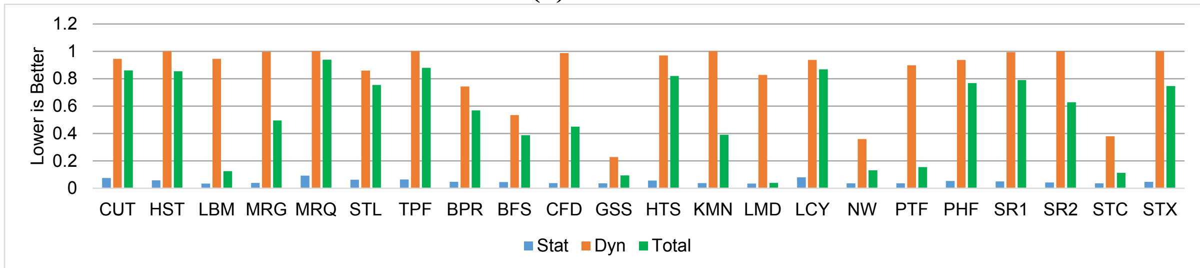
Figure 4-3 shows the power saving achieved for different types of caches for coarse granularity of 1 cache block for the region size. We report static, dynamic and total power saving for the caches. We take into account the power overhead of DWL for reporting power saving.



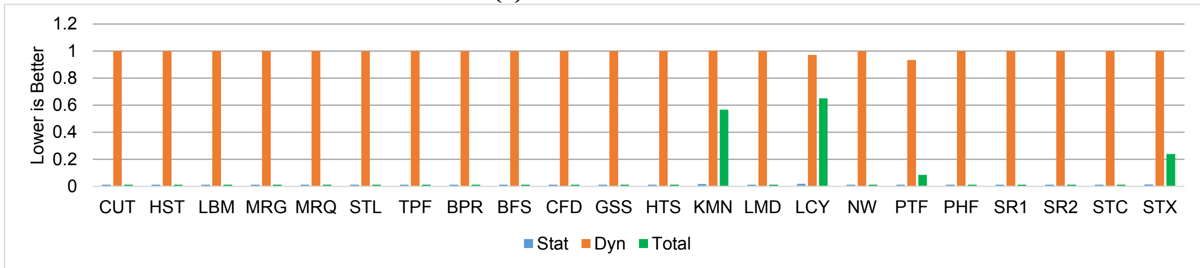
(a) L1 data cache



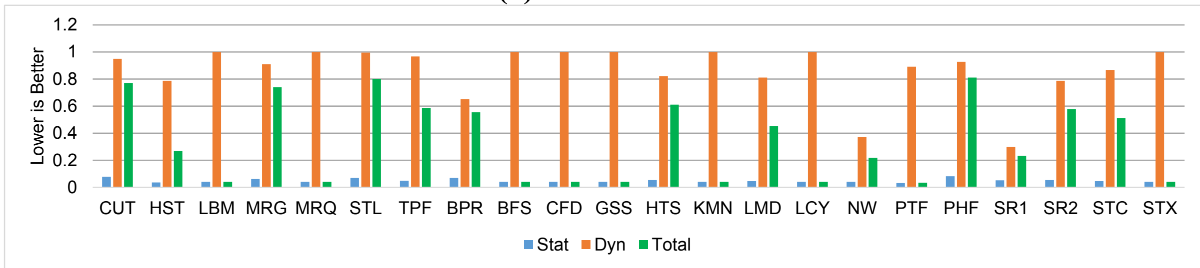
(b) L2 cache



(c) Constant cache



(d) Texture cache



(e) Shared memory

Figure 4-3: Static, dynamic and total power saving.

For each benchmark in figure 4-3, the first bar represents the static power in caches with drowsy mode relative to the static power of the baseline scheme. The second column in figure 4-3 shows dynamic power in caches with SWL activated by active mask relative to dynamic power of the baseline scheme. Bars less than one show power reduction. We used the model in CACTI v6.0 [15] to extract resistance and capacitance of SWLs. Similar to static power, we used HSPICE with PTM [12] and feature size of 32-nm to estimate dynamic power. The dynamic power depends on warp utilization of benchmarks. The warp utilization of benchmarks is listed in Table 3-1. Benchmarks with low warp utilization have more potential for dynamic power saving, whereas benchmarks with moderate warp utilization have limited dynamic power saving. The third column in figure 4-3 shows the normalized total power. The combined system employs both techniques proposed for reducing static and dynamic power. Benchmarks that have low warp utilization, exhibit highest power saving because they take advantage of both the leakage and dynamic power saving techniques. On average, static and dynamic power is reduced by more than 92% and 3%, respectively, in L1 data cache. The average power savings for L2 caches is more than 92% and 5% for static and dynamic power, respectively. In constant caches, static power is reduced by more than 94% and dynamic power is reduced by more than 15%. Texture caches report a power saving of more than 98% for static power and more than 2% for dynamic power. For shared memory, we observe a power saving of more than 94% and 13% for static and dynamic power, respectively.

We also evaluated power saving achieved for our techniques relative to the total GPGPU power. We assume that 1/3 of the power is consumed by leakage power [42]. Optimizing power of L1 data and L2 caches reduces total power of GPGPU by 0.93% and 1.53%, respectively. For constant and texture caches, we achieve a total power saving of 0.43% and 0.19%, respectively. Results for shared memory indicate that total power of GPGPU is reduced by 1.18%. Power saving results combined for all the caches indicate an overall GPGPU power saving of 4.26%.

The size of cache memories is increasing with every new generation of GPGPUs. This increases the power share of caches in the overall power budget of the GPGPU. We have

evaluated our techniques for Fermi architecture in this work. However, our proposed techniques can be used to achieve even greater power savings for the newer generation of GPGPUs owing to the larger size and, consequently, higher power of the caches.

Chapter 5
Summary and Future Work

General Purpose Graphics Processing Units (GPGPUs) have evolved from fixed function graphics pipeline to massively parallel general purpose processors capable of running thousands of threads on hundreds of cores. The massively parallel architecture of GPGPUs makes them a suitable choice for running workloads that exhibit high level of parallelism. However, for running parallel workloads from diverse computing domains, GPGPUs need to be able to provide huge amounts of data from memory sub-systems. This places a great demand on GPGPU manufacturers to include low-latency memory spaces, such as caches and shared memory, to service huge amounts of data to PEs without suffering access delays. The size of the low-latency memory spaces is continually increasing to keep up with the requirements of massively parallel workloads. However, increasing the size of these memory spaces also adds to power dissipation which has recently become a major design constraint in the design of microprocessor systems. The designers of these systems are now forced to look for energy efficient designs to be able to deliver continued growth in performance while still confining to acceptable energy dissipation constraints.

5.1 Contributions

Our work analyzes architecture and data access patterns, peculiar to GPGPUs, to propose techniques to reduce power consumption in caches in GPGPUs. We propose two power-aware optimization techniques that target static and dynamic power of L1 caches, shared memory and L2 cache in GPGPUs. Due to unique scheduling policies and execution model of GPGPUs, cache blocks have large inter-access distance and provide unique opportunities to reduce power. Our first optimization technique, coarse grained drowsy cells, puts cache blocks at a coarse granularity of cache regions into drowsy state while keeping only the last referenced region in ON state. GPGPUs have large number of warps to schedule for execution and putting cache blocks in drowsy state reaps huge power savings. The performance impact of using this coarse grained drowsy state management technique is

negligible since GPGPUs have a large pool of warps to choose from, for execution. The partitioning of cache blocks into regions exploits spatial and temporal locality for cache accesses and reduces the performance impact caused by wake-up latency. The cache lines in GPGPUs are wide to provide data to all the 32 threads in a warp in one single access. However, not all the threads in a warp may be active and unnecessary accesses to unused portions of cache blocks waste significant amount of power. Our second technique exploits active masks and eliminates activation of unused portions of cache blocks. On average, these two optimization techniques combined are able to reduce static and dynamic power by up to 98% and 15%, respectively.

5.2 Future Work

To further improve our current work in future, we can provide a small filter cache for L1 caches so that number of accesses to L1 caches are reduced. This small filter cache will also help to reduce access energy associated with L1 caches. Our coarse grained drowsy scheme attempts to exploit spatial and temporal locality for cache accesses within a cache region. Locality can be increased by increasing the region size. However, increasing the region size erodes effective power saving. Even if the cache region size is increased, the increase in locality is only exploitable within a cache region in our proposed scheme. A small filter cache can help to exploit spatial and temporal locality in a more effective manner. The granularity of the cache region can be kept small to ensure optimal power savings and the recently accessed cache blocks can be kept in filter cache to exploit locality. Another benefit of adding a filter cache to our scheme is that spatial and temporal locality can be exploited for cache blocks that lie outside a cache region. Since cache region is composed of contiguous cache blocks, it limits the spatial and temporal locality to within that region. The filter cache can help overcome this limitation by keeping recently used cache blocks from any address within the cache. Our results for sensitivity show that the performance improves with increasing the cache region size. Therefore, it is expected that the inclusion of filter cache will help to improve the performance of our scheme further without eroding much of the power saving.

References

[1] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In the Proceedings of ISCA, pp. 240-251, 2001.

[2] M. Gebhart et al., Unifying primary cache, scratch, and register file memories in a throughput processor. In the Proceedings of MICRO-45, pp. 96-106, 2012.

[3] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In the Proceedings of ISPASS, pp. 163-174, April 2009.

[4] A. Bakhoda, J. Kim, and T. Aamodt. 2010. Throughput-effective On-chip Networks for Manycore Accelerators. In Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 421-432, 2010.

[5] W. Fung, I. Sham, G. Yuan, and T. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 407-420, 2007.

[6] Matthias Boettcher et al. 2013. MALEC: A Multiple Access Low Energy Cache. In Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 368-373 (2013).

[7] A. Sankaranarayanan, E. K. Ardestani, J. L. Briz, and J. Renau. 2013. An Energy Efficient GPGPU Memory Hierarchy with Tiny Incoherent Caches, In ISLPED, pp. 9-14 (2013).

[8] K. Flautner et al. 2002. Drowsy caches: simple techniques for reducing leakage power. In Proceedings of ISCA, pp. 148 –157 (2002).

[9] NVIDIA Corp. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi (2009). NVIDIA.

[10] CUDA 2013. CUDA Programming Guide Version 5.0 (2013).

[11] NVIDIA Corp. 2012. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 (2012).

[12] PTM. 2012. Arizona state university predictive technology model, http://ptm.asu.edu.

[13] E. Demers. 2011. Evolution of AMD graphics, (2011). AMD Fusion Developer Summit.

[14] A. Agrawal, P. Jain, A. Ansari, J. Torrellas. 2013. Refrint: Intelligent refresh to minimize power in on-chip multiprocessor cache hierarchies. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 400-411, 2013.

[15] N. Muralimanohar et al. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In the Proceedings of MICRO (2007), pages 3-14.

[16] M. Abdel-Majeed and M. Annavaram. 2013. Warped Register File: A Power Efficient Register File for GPGPUs. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 412-423, 2013.

[17] M. Gebhart et al. 2011. Energy-efficient mechanisms for managing thread context in throughput processors. In Proceedings of the ISCA, pp. 235–246 (2011).

[18] NVIDIA. 2013. CUDA C/C++ SDK code samples (2013).

[19] Ehsan Atoofian. 2014. Reducing Static and Dynamic Power of L1 Data Caches in GPGPUs, in the Proceedings of HPPAC, Phoenix AZ, pp. 798-804, 2014.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In IISWC (2009), pages 44-54.

[21] J. A. Stratton et al. 2012. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing (2012).

[22] H. Zhou et al. 2001. Adaptive mode-control: A static-power-efficient cache design. Proceedings of international conference on Parallel Architectures and Compilation Techniques, pp. 61-70, 2001.

[23] M. Yoshimoto et al. 1983. A divided word-line structure in the static ram and its application to a 64k full cmos ram. IEEE Journal of Solid-State Circuits, 18(5):479 –485 (October 1983).

[24] NVIDIA Corp. 2014. NVIDIA GeForce GTX 750 Ti v1.1. NVIDIA.

[25] M. Powell et al. 2000. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In Proceedings of the 2000 international symposium on Low power electronics and design, pp. 90-95, 2000.

[26] Ehsan Atoofian et al. 2014. Power-Aware L1 and L2 Caches for GPGPUs. In Proceedings of 20th International Conference, Porto, Portugal (August 25-29, 2014), pp. 354-365.

[27] M. Abdel-Majeed et al. 2013. Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 111-122, 2013.

[28] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng. 2011. Power gating strategies on gpus. ACM Transactions on Architecture and Code Optimization (TACO), vol. 8, issue 3, article 13, October 2011.

[29] W. Yu, R. Huang, S. Xu, S.-E. Wang, E. Kan, and G. Suh. 2011. Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading. In 38th Annual International Symposium on Computer Architecture, pp. 247-258 (2011).

[30] J. Zhao and Y. Xie. 2012. Optimizing Bandwidth and Power of Graphics Memory with Hybrid Memory Technologies and Adaptive Data Migration. In Proceedings of the International Conference on Computer-Aided Design, pp. 81-87, 2012.

[31] David B. Kirk and Wen-mei W. Hwu. “Programming Massively Parallel Processors – A Hands-on Approach”. Morgan Kaufmann. ISBN: 978-0-12-381472-2, 2010.

[32] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pp. 133-ff, 1998.

[33] Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro and Murali Annavaram. 2015. Warped-Compression: Enabling Power Efficient GPUs through Register Compression, in Proceedings of the 42nd Annual International Symposium on Computer Architecture, pp 502-514, (2015).

[34] M. Kondo and H. Nakamura, "A small, Fast and Low-power Register File by Bit-partitioning," in Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 40-49, 2005.

[35] X. Guan and Y. Fei, "Register File Partitioning and Recompilation for Register File Power Reduction," ACM Trans. Des. Autom. Electron. Syst., vol. 15, no. 3, pp. 24:1-24:30, Jun. 2010.

[36] Sparsh Mittal, “A Survey of Techniques for Managing and Leveraging Caches in GPUs,” in Journal of Circuits, Systems, and Computers vol. 23, issue 8, pp.1430002, Sep 2014.

[37] W. Jia, K. A. Shaw, and M. Martonosi, “Characterizing and improving the use of demand-fetched caches in GPUs," in 26th ACM international conference on Super-computing, pp. 15-24, 2012.

[38] M. Rhu, M. Sullivan, J. Leng, and M. Erez, “A Locality-Aware Memory Hierarchy for Energy-Efficient GPU Architectures," in International Symposium on Microarchitecture (MICRO), pp. 86-98, 2013.

[39] Syed Zohaib Gilani , Nam Sung Kim , Michael J. Schulte, Power-efficient computing for compute-intensive GPGPU applications, Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), p.330-341, February 23-27, 2013.

[40] Q. Xu and M. Annavaram, "PATS: Pattern Aware Scheduling and Power Gating for GPGPUs," in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 225-236, 2014.

[41] J. Leng, S. Gilani, A. El-Shafiey, T. Hetherington, N. Sung Kim, T. M. Aamodt, and V. J. Reddi, “GPUWattch: Enabling Energy Optimization in GPGPUs”, in Proceedings of the International Symposium on Computer Architecture, pp. 487-498, 2013.

[42] 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, pp. 235-246, June 2011.