Dynamic Hardware Resource Management for Efficient Throughput Processing

by

Ankit Sethia

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2015

To my family

ACKNOWLEDGEMENTS

The first and foremost thanks goes to Professor Scott Mahlke, whose excellent guidance helped me immensely throughout my graduate school. He has been a constant source of energy and his guidance was critically important into shaping research ideas to completed projects. He made me realize that while success is to be celebrated, failures are a part of research and always motivated me to focus on the future and how in both the situations one can regain a sense of serenity by relying on Coke products.

I would like to thank my thesis committee Professor Trevor Mudge, Satish Narayanasamy and Zhengya Zhang for providing excellent feedback and suggestions that helped me refine my thesis. My experience in the industry influenced the way I evaluate the scope of research projects. I thank Stephen Hill of ARM and Debashis Bhattacharya from Huawei Technologies for helping me develop this outlook.

I have had the great opportunity to collaborate with a variety of insightful researchers in the Compilers Creating Custom Processors group; In my initial years, Dr. Ganesh Dasika was very helpful in setting my early foundations as a researcher. He was not only a highly inspirational collaborator, but he has been an influence on all spheres of my life in my graduate school and ensured that i survived the initial years, by picking me from dark parts of Ann Arbor late at night. Currently, I am really excited to work with John Kloosterman, who amazes me with his polite Canadian behaviour and impactful research ideas. Anoushe

Jamshidi's critical analysis of my work made sure that my research ideas were robust. His deep insights on landmark youtube videos led me to think out of the box several times.

I had some amazing friends at Michigan. I would like to thank Janghaeng Lee and Jason Park for teaching me Korean pleasantries which can never be used in public, Shruti Padmanabha for indirectly ensuring that I maintained decorum in the office, Yongjun Park for being the boss, Andrew Lukefahr for maintaining the fish tank. The importance of Hyoun Kyu Cho and Mehrzad Samadi in ensuring that I did not remain the undisputed slowest runner of the world cannot be under-stated. I would like to thank graduates of the CCCP group; Amin Ansari, Shuguang Feng, Shantanu Gupta, Amir Hormati, and Mojtaba Mehrara for showing that life after graduate school is beautiful enough to make progress.

I was lucky enough to have a group friends that provided me with a constant source of fun, laughter and tea to keep me fresh and motivated. I thank Anchal Agarwal, Mukesh Bachhav, Megha Bachhav, Animesh Banerjee, Gaurav Chadha, Ayan Das, Ujjwal Jain, Deepti Joshi, Vivek Joshi, Daya Shanker Khudia, Mohit Nahata, Gaurav Pandey, Ritesh Parikh and Abhayendra Singh for their support. Thanks also goes to Shaizeen Aga, Neha Agarwal and Aasheesh Kolli for discussing the mystic tales of graduate school.

Above all, the utmost of thanks go to my dear family for their steadfast support. My father Mansukh Lal Sethia persuaded me to have dreams other than watching TV and playing video games and he ensured that I followed them without fear. My mother Sashi Sethia exceled in the herculean task of keeping me focused on the important things in life. Finally and most importantly, I would like to express indebtness to my wife Divya Golchha, for her unconditional love and support and making sure that I reached the office everyday with sufficient determination in my heart and food in my stomach.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Dynamic Hardware Resource Management for Efficient Throughput Processing

by

Ankit Sethia

Chair: Scott Mahlke

High performance computing is evolving at a rapid pace, with throughput oriented processors such as graphics processing units (GPUs), substituting for traditional processors as the computational workhorse. Their adoption has seen a tremendous increase as they provide high peak performance and energy efficiency while maintaining a friendly programming interface. Furthermore, many existing desktop, laptop, tablet, and smartphone systems support accelerating non-graphics, data parallel workloads on their GPUs. However, the multitude of systems that use GPUs as an accelerator run different genres of data parallel applications, which have significantly contrasting runtime characteristics.

GPUs use thousands of identical threads to efficiently exploit the on-chip hardware resources. Therefore, if one thread uses a resource (compute, bandwidth, data cache) more heavily, there will be significant contention for that resource. This contention will eventually saturate the performance of the GPU due to contention for the bottleneck resource, leaving other resources underutilized at the same time. Traditional policies of managing the massive hardware resources work adequately, on well designed traditional scientific style applications. However, these static policies, which are oblivious to the application's resource requirement, are not efficient for the large spectrum of data parallel workloads with varying resource requirements. Therefore, several standard hardware policies such as using maximum concurrency, fixed operational frequency and round-robin style scheduling are not efficient for modern GPU applications.

This thesis defines dynamic hardware resource management mechanisms which improve the efficiency of the GPU by regulating the hardware resources at runtime. The first step in successfully achieving this goal is to make the hardware aware of the application's characteristics at runtime through novel counters and indicators. After this detection, dynamic hardware modulation provides opportunities for increased performance, improved energy consumption, or both, leading to efficient execution. The key mechanisms for modulating the hardware at runtime are dynamic frequency regulation, managing the amount of concurrency, managing the order of execution among different threads and increasing cache utilization. The resultant increased efficiency will lead to improved energy consumption of the systems that utilize GPUs while maintaining or improving their performance.

CHAPTER I

Introduction

Throughput oriented processors such as Graphics Processing Units (GPU) have been harnessed for their high raw computational power by several supercomputers in the top 500 list [2] and the green 500 list [1]. Many major domains of science and engineering have shown 10-300x speedup over general purpose processors. Furthermore, this speedup is achieved at similar power consumption levels as the general purpose processors. This has provided a significant boost to the overall energy efficiency of high performance computing systems for data parallel applications.

The reason for the success of GPU lies in the large scale exploitation of data level parallelism. Traditional CPUs process one element at a time and move to the next element after completely processing an element. In CPUs with SIMD extensions, upto 8 elements can be processed in parallel, before moving to the next set of elements. On the other hand, the scale of parallel processing is much higher in GPUs, to the tune of several hundreds of elements per cycle, with each element being processed on a different computational unit, such as floating point unit(FPU). They use high bandwidth GDDR5 memory to provide sufficient volume of data to the core, which can keep the compute units busy. GPUs have several thousands of threads to keep the several hundreds of compute units busy. The approach that GPUs use, is to divide the given work into several chunks of small, independent tasks(called as warps) and use fine grained multithreading between the warps to hide any stall in the pipeline. They can hide several hundreds of cycles of latency of an operation, if they are provided with enough work. For example, NVIDIA's GTX580 has 512 FPUs and uses over 20,000 threads to maintain high utilization of these FPUs via fine grained multithreading. As more elements are processed in parallel, with reduced stalls in the pipeline, the entire work is finished faster.

This parallel processing also enables higher energy efficiency for data parallel applications. GPUs ammortize the cost of fetch and decode stage of the pipeline, by operating only once for all threads in a warp. Furthermore, GPUs deploy energy efficient in-order cores and dedicate less area to traditional features such as branch prediction and data caches. While these features are performance critical for out-of-order CPUs, for throughput oriented architectures with fine grained multithreading such energy consuming hardware structures are not necessary. This is due to the fact that any stalls in the in-order pipeline of a GPU core can be overcome by swapping the stalling warp and executing another warp. All these factors lead to significant improvement in energy savings. As a point of comparison, the 4 core CPU by Intel(Haswell) with AVX2 instructions has peak throughput of 180 GFLOPS at 84W TDP as compared to 4.2 TFLOPS for NVIDIA's GTX780 at 250W TDP.

With the popularization of high level programming models such as CUDA and OpenCL, there has been a proliferation of general purpose computing using GPUs (GPGPU) on commodity systems to exploit this large computational power by non-scientific programmers as well. These programming models have facilitated the acceleration of wide range of data

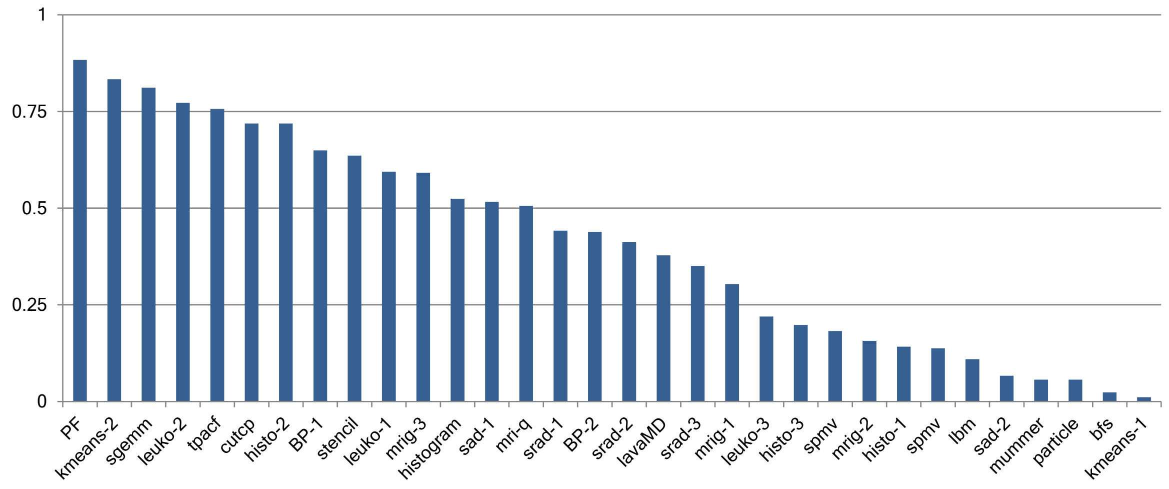


Figure 1.1: Utilization of GPU by kernels in parboil and rodinia benchmark suite.

parallel workloads such as image processing, computer vision, machine learning on GPUs on servers, desktop and mobile devices. Due to the prevalence of GPUs in commodity systems, a future with ubiquitous supercomputing is plausible.

The prevalence of GPUs to different genre of computing systems introduces a new set of challenges for the hardware designers. The runtime characteristics of these new domains do not strictly conform to that of the traditional scientific style applications. These new domains want to exploit not only the high computational power of the GPUs, they also want to exploit the high memory bandwidth and data cache present on the chip. Therefore the variation in requirements of modern GPU applications along with the large concurrency causes a significant problem. At full occupancy, more than twenty thousand, almost identical threads are executing on an SM. Therefore, if one thread has a high demand for using one of the resources of the GPU, then this imbalance in resource requirement is magnified many times causing significant contention. It is observed that the majority of the GPU applications are bottlenecked by the number of compute resources, available memory bandwidth or limited data cache [47]. As threads wait to access the bottleneck resource,

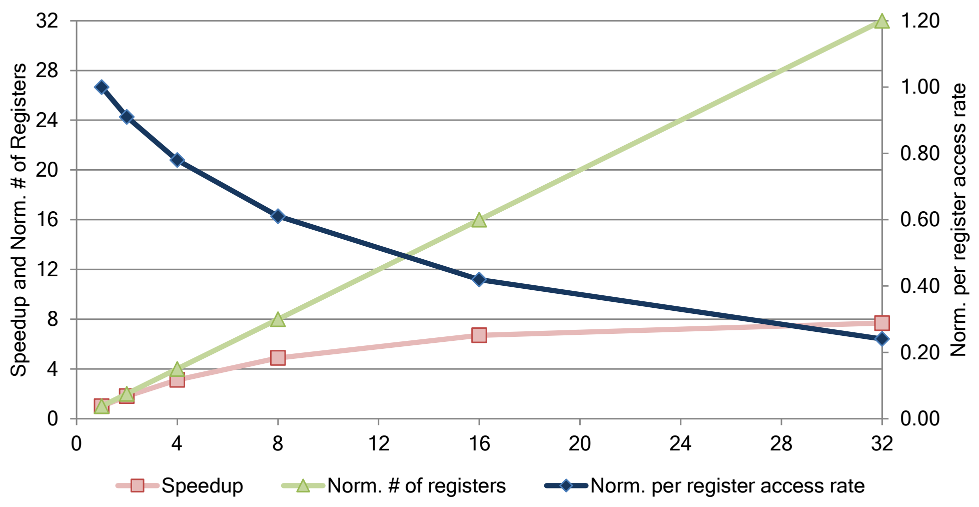


Figure 1.2: Increase in number of register file access rate and degradation in per register accesses. This shows that with SIMT more hardware is required, but the utilization of the hardware reduces significantly.

other resources end up being under-utilized, leading to imbalanced inefficient execution.

Figure 1.1 show the fraction of the GPU utilized by all the kernels in two of the most popular GPU benchmark suites. All these kernels provide sufficient amount of work that can be executed independently, ensuring that the GPU is not starved for work. Only 9 out of the 30 kernels achieve above 60% utilization, and from the remaining 21, 13 of achieve even less 30% utilization on the GPU. Low utilization of the GPU leads to lower energy efficiency as majority of the hardware is not being utilized and leaks static energy which is becoming a significant part of the total energy. This energy inefficiency, while unwanted in supercomputing systems, is unacceptable for untethered devices.

While GPUs are provisioned to have large concurrency on chip, the return on addition of new hardware to support large concurrency is not propotional. Figure 1.2 shows the variation in speedup, normalized number of registers as compared to one warp on the primary vertical axis and normalized per register access rate as compared to one warp on the secondary vertical axis. The increase in the number of registers is linear and propotional

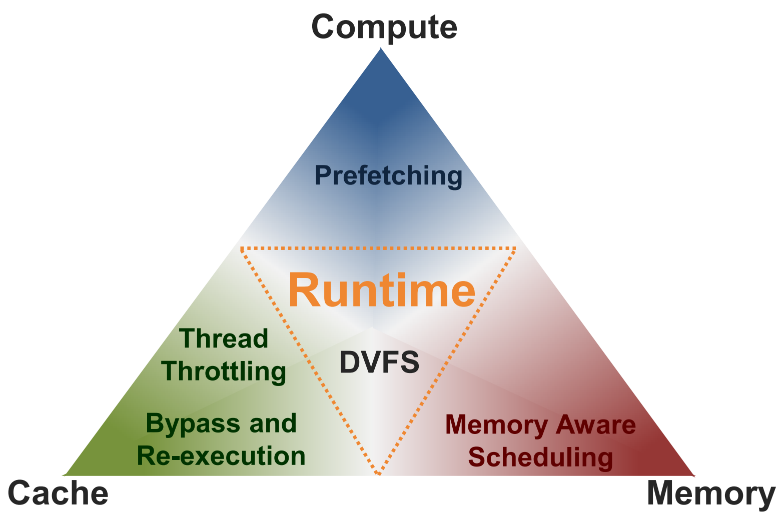


Figure 1.3: GPUs provide three major resources (compute, memory and cache) and at the center of this research is a runtime management system. This system observes the resource requirement of the GPU program and provides techniques for efficient execution depending on the required resource.

to the number of warps present on a core. However, the speedup achieved is not linear and saturates at 7.69 times for 32 warps. Therefore, the overall access rate per register for 32 warps is down to 0.24 times from the 1 warp case. This demonstrates that the design of GPUs with heavy multi- threading has a significant amount of underutilized hardware which can lead to a significant increase in power consumption due to leakage. Technological trends in semiconductor fabrication show that at present leakage power is around 30% of the total chip power for high performance designs [45, 26, 75] and will continue to grow in future technologies. Hence, reducing the power of underutilized hardware on GPU has been the focus of recent research activities [26, 27, 3].

To improve the utilization of GPU resources, this thesis uses a central runtime management system which deploys several efficiency improvement techniques. However, these techniques are effective when the GPU program exhibits certain characteristics (e.g. data locality). Therefore, the runtime system observes the requirement of the GPU program (kernel) and applies appropriate techniques to improve the efficiency of execution. The resources, the optimization techniques and their sphere of influence is illustrated in Figure 1.3. The runtime system observes the resource requirement of the kernel, by first looking at the requirements of the warps inside a core and then collecting the requirements of all the cores. Some decisions are taken at a core granularity (thread throttling) and some are taken at a global granularity (DVFS decisions).

When the runtime system detects that the kernel has data locality it uses a thread throttling mechanism to reduce the thrashing of the cache and it uses a unique re-execution mechanism to improve hit-under-miss opportunities. As the data cache size on the GPU core is small, reducing the number of concurrent threads, leads to increased cache per thread and hence reduces contention. The re-execution mechanism unblocks the stalled Load Store Unit in the core by using a re-execution to park the blocking request. This allows loads which have data in the cache to go ahead and make forward progress. These techniques significantly improve the data cache hit rate, leading to higher performance and efficiency. These mechanisms are explained and evaluated in Chapter III and IV.

The rate of increase of memory bandwidth has not been able to keep upto the increase in the number of FPUs in the GPU. Therefore memory intensive workloads will saturate the GPU bandwidth and the FPUs might starve for data to process. In several classes of algorithms, such as search, data mining, sparse matrix computation etc. the memory requirement can easily saturate the memory sub-system of the GPU. When the runtime detects that a memory intensive kernel is executing, it changes the traditional scheduling policy between warps in a core to a novel memory aware scheduling. While the traditional scheduling policies are oblivious to the resource requirement of the kernel, the new scheduling policy focuses on making all the memory resources available to one warp so that it can start its computation. This results in better overlap of computation and memory access. The details of this mechanism are in Chapter IV.

For compute intensive kernels there is not a significant amount of under-utilization of the compute resources. Therefore, to improve the efficiency of such kernels, we focus using prefetching to reduce the perceived memory access latency and hence reduce the parallelism required. This provides the runtime system with opportunities to shut down large parts of the GPU core that are used for supporting large levels of concurrency. This shutting down is done by power gating of register file, resulting in reduction of leakage power for the large structure. The details of this technique and the related performance and energy trade-offs are discussed in Chapter V.

A powerful technique to improve energy efficiency and performance of GPU kernels is Dynamic Voltage and Frequency Scaling (DVFS). This technique is another central contribution of this work. As the runtime detects the resource requirements of the kernel, the corresponding resource's voltage and frequency operating point could be increased to boost the capacity of that resource. As the performance of the bottleneck resource is boosted, the overall performance will be improved. Similarly, the voltage and frequency operating point of the under-utilized resource can be decreased to provide energy efficiency. The two modes of operation and the runtime decision process is explained in Chapter III.

The thesis explains the background in Chapter II to explain the GPU architecture, the runtime resource detection system, DVFS and thread throttling in Chapter III, the memory aware scheduling and cache re-execution mechanism in Chapter IV, the adaptive prefetching in Chapter V and is concluded in Chapter VI along with discussion for potential extensions to this work.

CHAPTER II

Background

This section provides background for the various aspects of GPU that this thesis focuses on and will be relevant for future chapters.

2.0.1 Modern GPU architecture

To develop a background for GPU architecture we choose NVIDIA's nomenclature in this paper. In a GPU architecture like NVIDIA's GTX580, the GPU has 16 Streaming Multiprocessors (SM) which operate independently. Figure 2.1 [26] shows the architecture of one SM. It has banks of register files for all the threads that it handles and has 32 ALUs. The 32 ALUs are divided into groups of 16 ALUs that execute the same instruction. It also has access to local memory, a texture cache and a special function unit (SFU) for transcendental functions, floating point divide, etc. In an SM, groups of 32 threads operate in a lock-step manner and execute the same instruction. These group of threads are known as warps. Each SM can handle several independent warps (upto 32 for GTX 280 and 48 for GTX 580). When a warp starts execution, it will stall for long latency instructions such as access to global memory, long latency floating point instructions, etc. Meanwhile, the warp

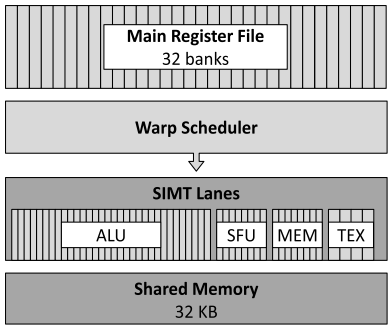


Figure 2.1: Architecture of a Streaming Multiprocessor(SM)

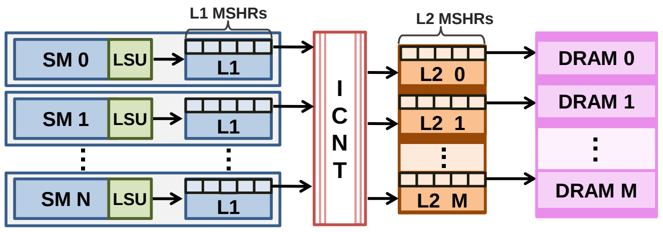


Figure 2.2: GPU Memory hierarchy block diagram which shows N SMs and M memory partitions.

scheduler selects another warp from the available ready warps. The warp scheduler will not consider the stalled warp for execution till the warp is ready. If the program running on the GPU is using the maximum number of threads, the scheduler has the option to choose from 32 warps. It is this massive amount of multi-threading that allows the ALUs in the SM to execute continuously, thereby resulting in very high throughput.

2.0.2 GPU memory hierarchy

Figure 2.2 shows the block level diagram of the system considered in this work. The details inside the SM which include the execution units, register file etc. have been abstracted. Each SM has its private L1 cache. The L1 cache has hardware support to handle a fixed number of outstanding requests by using Miss status handling register(MSHR) as shown in the figure. All the SMs are connected to an interconnect which is connected to the L2. Depending on the requested address, the request is sent to the corresponding L2 partition which is connected to a DRAM channel. This DRAM channel is responsible for the same set of addresses as the partition itself. Each L2 partition can also handle a fixed number of outstanding memory requests.

For memory intensive workloads, the number of memory requests being sent are high. This results in the MSHRs in the L2 getting filled up much faster than for compute intensive workloads. Therefore, L2 stops accepting requests and sends negative acknowledgements to new requests from the interconnect. With L2 rejecting any new requests, the interconnection between L2 and L1 is filled up. This results in the rejection of new requests from L1 to the interconnect. Any new request coming from the core is allocated a new MSHR at L1, but no new request can be sent to L2 due to the earlier congestion in the interconnect. This cascading effect of rejecting memory requests reaches the core once all the MSHRs in the L1 are filled up. At that point an SM can no longer issue a memory instruction from any of the warp. This results in serialization of all memory requests as a new memory request can only be sent if one request is completed by the memory sub-system which frees up one of the MSHRs.

If all the MSHRs in the L1 are filled up and the LSU requests an address from the L1 cache that is a miss in the cache, the cache sends a negative acknowledgement to the LSU. At this point, the LSU can no longer process new requests even from other warps, until an older requests is returned from the memory and frees one MSHR. Since, the LSU can no longer process a new request, no warp can access the L1 cache as the LSU is busy with the previous unallocatable request. Other warps that need the compute units of the SM can still continue execution. If all the memory requests of a warp return before other warps' compute is finished, then the new warp can hide the latency of warps whose requests are yet to return. But due to the serialized memory accesses, this required compute is significantly higher as compared to the compute required when the requests are pipelined.

2.0.3 GPU thread management

GPUs use the Single Instruction Multiple Thread (SIMT) model of execution which groups threads together into a warp and executes the same instruction for these threads using a common physical Program Counter (PC). Several warps form a Co-operative Thread Array (CTA) and threads within a CTA can communicate via the Shared Memory on a SM. The global work distribution engine (GWDE) in a GPU is the scheduler of all the CTAs and schedules them to SMs. The granularity of assignment of concurrent threads on an SM is the number of threads in a CTA. Initially, the GWDE fills up all the SMs with the maximum number of CTAs possible, which is dependent on the number of registers, shared memory and number of threads required by a CTA. After receiving the appropriate number of CTAs for execution, an SM continues execution until one of the CTAs finishes. At this point, it requests the GWDE to provide another CTA. This process continues until all the CTAs to be executed are transferred to one of the SMs.

CHAPTER III

Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution

In this chapter, i will reason for the need to match the resources on the GPU with the application's requirement. Once the requirement of the kernel are detected then, a dynamic runtime system can modulate the hardware parameters to improve the efficiency of the execution on the GPU. The efficiency improvements can happen by either boosting bottleneck resources or throttling under-utilized resources. The decision to do one or the either can be taken by the operating system or the application.

3.1 Introduction

Modern GPUs provide several TFLOPs of peak performance for a few hundred dollars by having hundreds of floating point units (FPUs) and keeping them busy with thousands of concurrent threads. For example, NVIDIA's GTX580 has 512 FPUs and uses over 20,000 threads to maintain high utilization of these FPUs via fine-grained multi-threading. GPUs are stocked with high memory bandwidth of up to 192 GBps and 64 kB of local storage per streaming multiprocessor (SM) to feed data to these FPUs.

At full occupancy, more than a thousand, almost identical threads are executing on an SM. Therefore, if one thread has a high demand for using one of the resources of the GPU, then this imbalance in resource requirement is magnified many times causing significant contention. We observe that the majority of the GPU applications are bottlenecked by the number of compute resources, available memory bandwidth or limited data cache [47]. As threads wait to access the bottleneck resource, other resources end up being under-utilized, leading to inefficient execution. While these hardware resources cannot be increased at runtime, there are three important parameters that can modulate the performance and energy consumption of these resources: number of concurrent threads, frequency of the SM and frequency of the memory system.

Running the maximum number of threads on an SM causes inefficient execution by saturating the compute resources in compute intensive workloads and memory bandwidth for memory intensive workloads. However, its impact on the data cache of an SM is even more critical. It significantly reduces the usefulness of the limited-capacity L1 data cache. For NVIDIA's Fermi architecture, each SM can have up to 48 kB of data cache resulting in each thread having fewer than 30 bytes of cache on average when running maximum threads. With such a small footprint allotted to each thread, the advantages of data locality are lost due to increased cache contention. Rogers et al. [70] and Kayiran et al. [40] have shown the detrimental effect of running large number of threads on cache hit rates in GPUs. However, their solutions use architecture dependent heuristics that are not as effective across architectures.

Another reason for inefficient execution is the rigidity of the core and memory system voltage and frequency operating points. For example, while the GDDR5 DRAM can provide a bandwidth of up to 192 GBps, the idle standby current is 30% higher as compared to when it provides 3.2 Gbps [34]. As the memory system is not utilized significantly for compute intensive kernels, there is an opportunity to save energy if the performance of memory system is lowered for such kernels without degrading the overall performance. Similar opportunity is present for memory intensive kernels, if the core frequency and voltage can be lowered as SMs are not the bottleneck. Lee et al. [49] analyzed the trade-off of saving energy by DVFS and core scaling for such cases, but do not have a robust runtime mechanism to deal with all scenarios.

If the proclivity of a workload can be known a priori, advanced programmers can set the desired number of concurrent threads, frequency of the core and the memory system. However, static decisions are often infeasible due to three reasons. First, the contention for a hardware resource may be heavily dependent on the input. For example, a small input set might not saturate the memory bandwidth, whereas a large input set might. Second, resource contention is dependent on the amount of given GPU hardware resources and an application optimized for a GPU may change its point of contention on another GPU. Third, due to the prevalence of general purpose computing on GPUs (GPGPU), more irregular parallel applications are being targeted for GPUs [13, 59]. This has resulted in GPGPU kernels having distinct phases where different resources are in demand.

As running the maximum number of threads at fixed core and memory system frequency is not always the best solution and they cannot be determined a priori and independently, an intelligent runtime system is required. This system should be able to tune three important architectural parameters: number of threads, core frequency and memory frequency in a coordinated fashion as these parameters are dependent.

To address the limitations of prior work and exploit the significant opportunities provided by modulating these three parameters, we propose Equalizer [78], a comprehensive dynamic system which coordinates these three architectural parameters. Based on the resource requirements of the kernel at runtime, it tunes these parameters to exploit any imbalance in resource requirements. As new GPU architectures support different kernels on each SM, Equalizer runs on individual SMs to make decisions tailored for each kernel. It monitors the state of threads with four hardware counters that measure the number of active warps, warps waiting for data from memory, warps ready to execute arithmetic pipeline and warps ready to issue to memory pipeline over an execution window. At the end of a window, Equalizer performs two actions to tune the hardware.

Firstly, it decides to increase, maintain, or decrease the number of concurrent threads on the SM. Secondly, it also takes a vote among different SMs to determine the overall resource requirement of the kernel based on the above counters. After determining the resource requirements of a kernel, Equalizer can work in either energy efficient or high performance modes. In the energy mode, it saves energy by throttling under-utilized resources. As only the under-utilized resources are throttled, its performance impact is minimal. In the performance mode, only the bottleneck resource is boosted to provide higher performance at modest energy cost. Equalizer achieves a net 15% energy savings while improving performance by 5% in the energy saving mode and achieves 22% performance improvement in the performance mode while consuming 6% additional energy across a spectrum of GPGPU kernels.

This work makes the following contribution:

- We explore the opportunity for energy savings and performance improvement that a dynamic adaptable system can achieve over a fixed GPU architecture for a variety of kernels that have sophisticated resource requirements.

- We provide an in-depth analysis of the time spent by the warps on an SM. Through this analysis, we introduce four novel hardware counters based on the execution state of the warps. These counters represent waiting warps, active warps, ready to execute memory warps, and ready to execute compute warps. They expose the collective resource utilization of the application.

- We propose a comprehensive, low overhead runtime tuning system for GPUs that dynamically adapts the number of threads, core frequency and memory frequency for a given kernel in unified way to either save energy by throttling unused resources or improve performance by reducing cache contention and boosting the bottleneck resource.

3.2 Opportunities for a Dynamic System

In this section we describe various opportunities that a runtime system can exploit. We study 27 kernels from the Rodinia and Parboil benchmark suites and classify them into four categories on a NVIDIA Fermi style (GTX 480) architecture: 1) compute intensive which have contention for the compute units, 2) memory intensive which stress the memory bandwidth, 3) cache sensitive which have contention for L1 data cache and 4) unsaturated which do not saturate any of the resources but can have inclination for one of the resources.

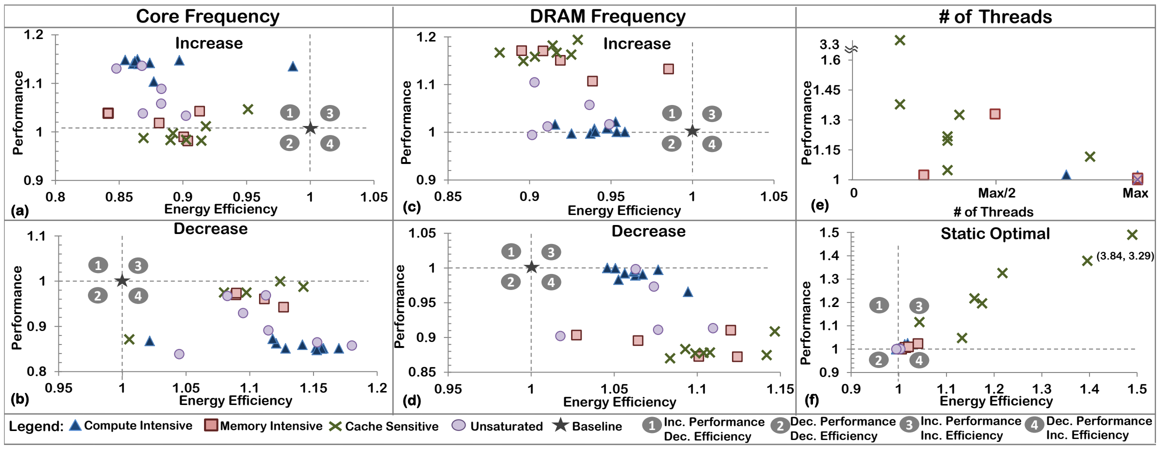


Figure 3.1: Impact of variation of SM frequency, DRAM frequency and number of concurrent threads on performance and energy efficiency. The black star mark shows the value for the baseline GPU and is always set to 1 for both performance and energy. The properties of the four quadrants relative to the star mark are explained in the legend at the bottom.

3.2.1 Effect of Execution Parameters

Figure 3.1 shows the impact of varying SM frequency, memory frequency and number of threads on the performance and energy efficiency of different kernels. Energy efficiency is defined as the ratio of energy consumed by the baseline Fermi architecture over the energy consumed by the modified system. Higher value of energy efficiency corresponds to lower energy consumption in the modified system. The kernels and methodology used for this experiment is described in Section 3.5. A black star mark on each sub-figure shows the position of the baseline. The four quadrants 1, 2, 3 and 4 in the sub-figures represent deviation from the black star. In quadrant 1 performance improves and efficiency decreases, while in quadrant 2 performance and efficiency decrease. In quadrant 3 performance and efficiency increase, while in quadrant 4 performance decreases and efficiency increases.

SM Frequency: Figure 3.1a shows the impact of increasing the SM frequency by 15%. The compute kernels show proportional improvement in performance and increase in energy by moving deep into quadrant ①. The result for memory and cache kernels are very different. Since these kernels are not constrained by the SM, faster computations by increasing the SM frequency does not reduce any stalls. Therefore, these kernels achieve insignificant speedup and stay close to the dotted line which represents baseline performance. Hence, increasing SM frequency is effective only for compute kernels and should be avoided for others.

On the other hand, when the SM frequency is reduced by 15%, the most affected kernels are compute kernels and they move significantly into quadrant ④losing performance while saving energy (Figure 3.1b. In such kernels, the SM's compute resources are the bottleneck and slowing the SM will slow these resources, reducing performance. While there is a significant reduction in the energy consumed, such large drops in performance are generally unacceptable. On the other hand, the loss in performance for memory and cache kernels is small, while the energy efficiency improves significantly, pushing the kernels into quadrant ④. The primary reason for this behavior is the large periods of inactivity of the compute resources.

Memory Frequency: While SM frequency affects energy and performance of compute kernels, memory frequency has similar effects on memory kernels. Cache kernels behave like memory kernels due to cache thrashing, which leads to higher bandwidth consumption. Figure 3.1c shows the impact of increasing the DRAM frequency by 15%. Memory and cache kernels move deep into quadrant ①due to the improved performance. The decrease in energy efficiency is lower than increasing SM frequency as the memory contributes less significantly to the total energy. Analogous to the impact of SM frequency on memory kernels, increasing DRAM frequency does not impact compute kernels as the memory is not fully utilized at the base frequency. These kernels achieve no speedup and increase the energy consumption by 5%.

Decreasing the memory frequency affects the memory and cache kernels as shown by Figure 3.1d. As memory bandwidth is the bottleneck for such kernels, this behavior is expected. However, reducing DRAM frequency has no performance impact on compute kernels while improving energy efficiency by 5%, indicating an opportunity to decrease the DRAM frequency and voltage for compute kernels.

Number of Thread Blocks: Increasing the DRAM frequency helps cache kernels get data back faster. However, controlling the number of threads to reduce L1 data cache thrashing will improve performance significantly with minimal energy increase. Therefore, we first analyze the optimal number of threads that need to run on an SM. Figure 3.1e shows the best performance achieved by the kernels by varying the number of concurrent threads on an SM. The compute and memory kernels achieve best performance with maximum threads and overlap at (Max Threads, 1) as saturating these resources does not hurt performance significantly and only leads to inefficient execution. The best performance for the cache kernels is achieved at lower concurrency levels where there is less contention for the cache. Therefore the big challenge for a runtime system is to find the most efficient number of threads to run. Note that if threads less than optimal are run, there might not be sufficient parallelism to hide memory access latency, which will result in lower performance.

The algorithm to decide the number of concurrent threads should ensure that the number of threads are not reduced significantly for compute and memory kernels as performance might suffer due to the lack of work. Figure 3.1f shows the improvement in energy effi-

Table 3.1: Actions on different parameters for various objectives

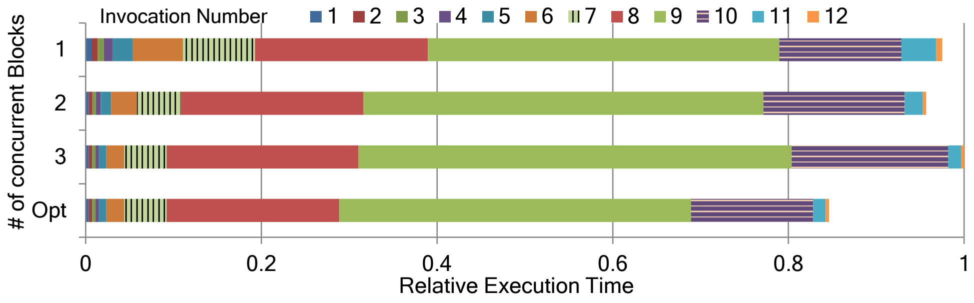
Kernel	**Objective**	**SM Frequency**	**DRAM Frequency**	**Number of threads**
Compute	Energy	Maintain	Decrease	Maximum
Intensive	Performance	Increase	Maintain	Maximum
Memory	Energy	Decrease	Decrease	Maximum
Intensive	Performance	Maintain	Increase	Maximum
Cache	Energy	Decrease	Maintain	Optimal
Sensitive	Performance	Maintain	Increase	Optimal

ciency, if the best performing number of concurrent threads are selected statically. There is significant improvement in performance and energy efficiency as kernels go high into quadrant ③. Therefore, choosing the best number of threads to run concurrently is suitable for saving energy as well as improving performance. For compute and memory kernels, running maximum threads leads to best performance and energy efficiency

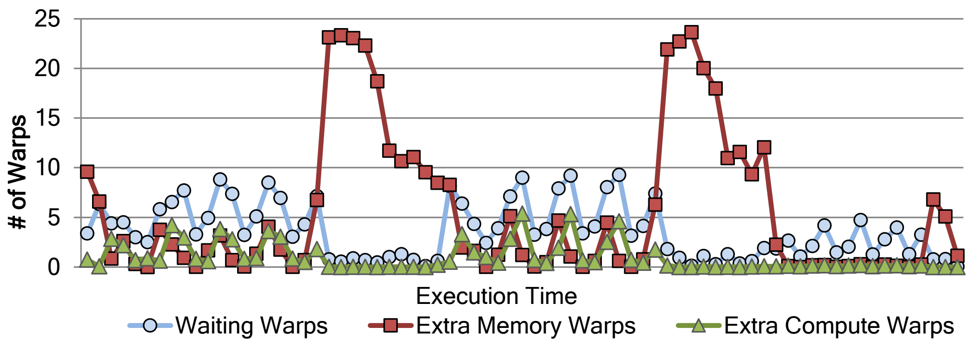
Actions for Dynamic System: The action of increasing, maintaining, or decreasing the three parameters depend on the objective of the user. If the objective is to save energy, the SM and memory frequency should be reduced for memory and compute kernels respectively. If the objective is to improve performance, the SM and memory frequency should be increased for compute and memory kernels respectively. Running the optimal number of threads blocks for cache sensitive cases is beneficial in both the objectives. These conditions and actions are summarized in Table 3.1.

3.2.2 Kernel Variations

Kernels not only show diverse static characteristics in the resources they consume, but their requirements also vary across and within invocations. Figure 3.2a shows the distribution of execution time across various invocations of the bfs-2 kernel for three statically fixed number of thread blocks. All values are normalized to the total time taken for the



(a) Distribution of total execution time across various invocations for different, statically fixed number of thread blocks for the bfs-2 kernel. No single configuration performs best for all invocations. Each color is a different invocation of the kernel.



(b) Resource requirement for entire execution of the mri-g-1 kernel

Figure 3.2: Variation of kernel requirements across and within kernel instances.

kernel with maximum concurrent thread blocks (3). The performance of having 3 thread blocks is better than having 1 block until invocation number 7 (vertical stripes). But from invocation number 8 to 10 (horizontal stripes), having 1 block is better. After invocation 10, having 3 thread blocks is better again. An optimal solution would never pick the same number of blocks across all invocations. A 16% improvement in performance is possible by simply picking the ideal number of thread blocks for every invocation as shown by the bottom bar.

An example of variation in resource requirements within a kernel invocation is shown in Figure 3.2b for the mri-g-1 benchmark. Over most of the execution time, there are more warps waiting for data to come back from memory than warps ready to issue to memory. However, for two intervals, there are significantly more warps ready to issue to memory,

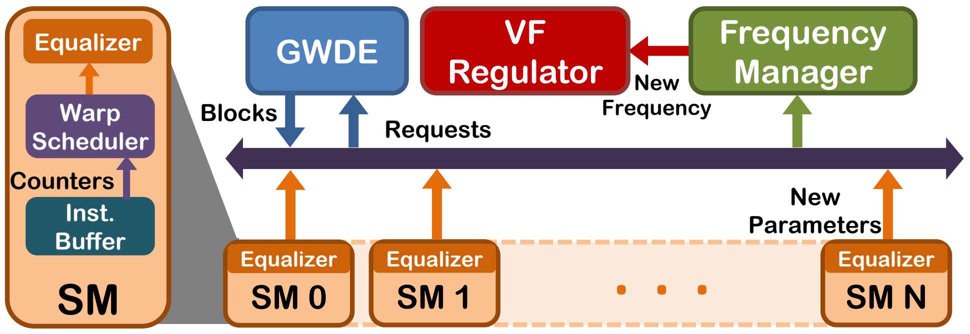


Figure 3.3: Equalizer Overview: Each SM requests new thread block and the new SM and memory frequency domain. The frequency manager takes the requests and changes the SM or memory frequency as requested by majority of the SMs. GWDE manages the request for new thread blocks.

putting pressure on the memory pipeline. During these intervals, a boost to the memory system will relieve the pressure and significantly improve the performance.

Overall, there are significant opportunities for a system that can control the number of threads, SM frequency and memory frequency at runtime. These opportunities are present not only across different kernels, but also across a kernel's invocations and within a kernel's invocation. In the following section, we describe how Equalizer exploits these opportunities to save energy or improve performance.

3.3 Equalizer

The goal of Equalizer is to adjust three parameters: number of thread blocks, SM frequency and memory frequency, to match the requirements of the executing kernels. To detect a kernel's requirement, Equalizer looks at the state of the already present active warps on an SM and gauges which resources are under contention. The state of active warps is determined by a collection of four values: 1) number of active warps, 2) number of warps waiting for a dependent memory instruction, 3) number of warps ready to issue to memory pipeline, and 4) number of warps ready to issue to arithmetic pipeline. Large values for the last two counters indicate that the corresponding pipelines are under pressure. At runtime, Equalizer periodically checks for contention of resources using the state of the warps. It makes a decision to increase, maintain or decrease the three parameters at the end of each execution window ($epoch$). If Equalizer decides to change any parameter, the new value differs from the previous value by one step. The details of the decision process are explained in Section 3.3.2.

Figure 3.3 shows the interaction of Equalizer with the other components of a GPU. It receives the four counters mentioned above, from the warp scheduler in an SM and makes a local decision. If the decision is to increase number of threads, the Global Work Distribution Distribution Engine (GWDE) which manages thread block distribution across SMs, issues a new thread block for execution to the SM. If Equalizer decides to reduce the number of concurrent threads, it uses the CTA Pausing technique used in [40]. In this technique, Equalizer blacklists a thread block and the warp scheduler does not issue instructions to the data-path from the warps of these thread blocks. After a thread block finishes on the SM, Equalizer maintains the reduced *numBlocks* by unpausing the paused thread block and does not request a new thread block from the GWDE, thereby maintaining the reduced value of *numBlocks* . Based on the objective of Equalizer, each SM submits a Voltage/Frequency (VF) preference to the Frequency Manager every $epoch$. The frequency manager shown in Figure 3.3 receives these requests and makes a global decision for the new VF level for the SM and memory based on a majority function.

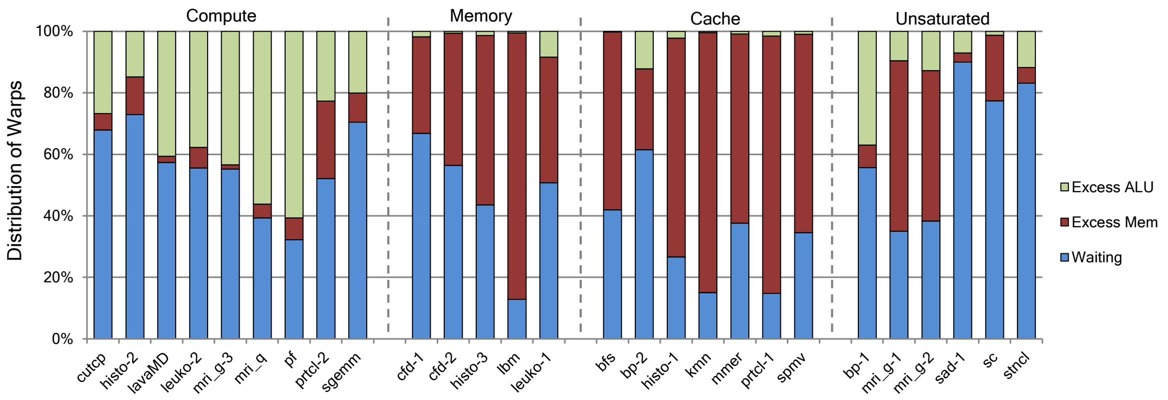


Figure 3.4: State of the warps for different categories of kernels. The names on top of the chart show the various categories.

3.3.1 State of Warps

When a kernel executes on an SM, warps of the kernel can be in different states. We classify the warps depending on their state of execution in a given cycle:

- $Waiting$- Warps waiting for an instruction to commit so that further dependent instructions can be issued to the pipeline are in this category. The majority of warps are waiting for a value to be returned from memory. The number of warps needed to hide memory latency is not only a function of the number of memory accesses made by the warps, but also of the amount of compute present per warp. **An SM should run more than $Waiting$ number of warps concurrently to effectively hide memory access latency.**

- $Issued$- Warps that issue an instruction to the execution pipeline are accounted here. It indicates the IPC of the SM and a high number of warps in this state indicate good performance.

- $Excess\ ALU$ (X_{alu})- Warps that are ready for execution of arithmetic operations, but cannot execute due to unavailability of resources are in this category. These are ready to execute warps and cannot issue because the scheduler can only issue a fixed number of instructions per cycle. X_{alu} **indicates the excess warps ready for arithmetic execution.**

- $Excess\ memory$ (X_{mem})- Warps that are ready to send an instruction to the Load/Store pipeline but are restricted are accounted here. These warps are restricted if the pipeline is stalled due to back pressure from memory or if the maximum number of instructions that can be issued to this pipeline have been issued. X_{mem} **warps represents the excess warps that will increase the pressure on the memory subsystem from the current SM.**

- $Others$- Warps waiting on a synchronization instruction or warps that do not have their instructions in the instruction buffer are called $Others$. As there is no instruction present for these warps, their requirements is unknown.

In principle, one warp in X_{alu} or X_{mem} state denotes contention for resources. However, Equalizer boosts or throttles resources in discrete steps and in either cases, there should not be lack of work due to the modulation of parameters. Hence, there should be some level of contention present before Equalizer performs its actions.

Figure 3.4 shows the distribution of three of the above states on an SM for the 27 kernels broken down by category, while running maximum concurrent threads. $Others$ category is not shown as their resource requirements cannot be observed. The following observations are made from the state of warps:

- Compute intensive kernels have a significantly larger number of warps in X_{alu} state as compared to other kernels.

- Memory intensive and cache sensitive kernels have a significantly larger number of warps that are in X_{mem} state as compared to the other categories.

- All unsaturated kernels still have inclination for either compute or memory resources as they have significant fraction of warps in X_{alu} or X_{mem} state.

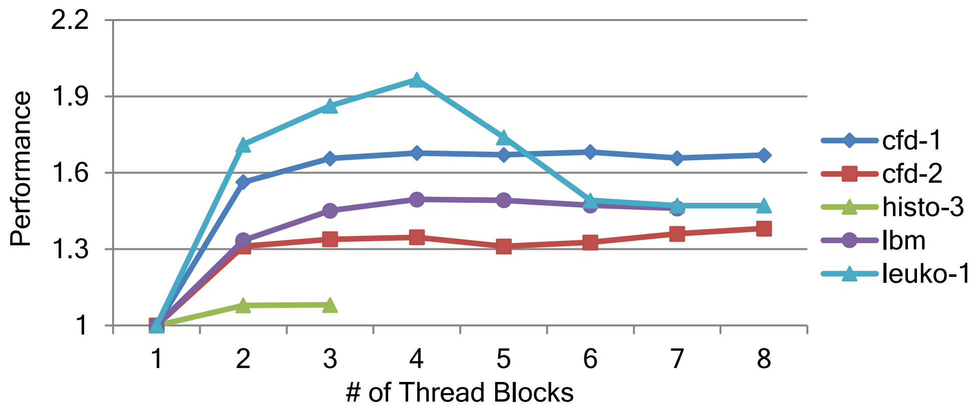


Figure 3.5: Performance of memory intensive kernels with number of concurrent thread blocks

Unifying Actions on Memory Intensive and Cache Sensitive Kernels: As the state of the warps for memory intensive and cache sensitive kernels are similar, we unify the process of tuning the resources for the two cases. Figure 3.5 shows the performance of memory intensive kernels with a varying number of thread blocks. All kernels saturate their performance well before reaching the maximum number of concurrent blocks. As long as the number of blocks for a memory intensive kernel is enough to keep the bandwidth saturated, we do not need to run the maximum number of blocks. In case the large number of warps in X_{mem} state were due to cache thrashing, this reduction in thread blocks will reduce cache contention.

In principle, if every cycle an SM sends a request that reaches DRAM, then as there are multiple SMs, the bandwidth will be saturated leading to back pressure at the SM. Therefore, the Load Store Unit(LSU) will get blocked and all warps waiting to access memory will stall. So even a single warp in X_{mem} state is indicative of memory back pressure. However, when this X_{mem} warp eventually sends its memory request, it might be hit in the L1 or L2 cache. Therefore the earlier X_{mem} state of the warp was not actually representing excess pressure on DRAM and so we conservatively assume that if there are two warps in X_{mem} state in steady state then the bandwidth is saturated. So Equalizer tries to run the minimum number of blocks that will keep the number of warps in X_{mem} greater than two and keep the memory system busy and reduce L1 cache contention with minimum number of thread blocks.

3.3.2 Equalizer Decision

To exploit the tendencies of a kernel as indicated by the state of warps, we propose a dynamic decision algorithm for Equalizer. Once the tendencies are confirmed, the algorithm performs actions based on the two objectives mentioned in Table 3.1. Due to an imbalance in the kernel's resource requirements, one of the resources is saturated earlier than others and hence, several warps end up in X_{alu} or X_{mem} state. Therefore, whenever Equalizer detects that X_{alu} and X_{mem} are beyond a threshold, it can declare the kernel to be compute or memory intensive. This threshold has to be conservatively high to ensure that the resources are not starved for work while changing the three parameters.

If the number of warps in the X_{alu} or X_{mem} are more than the number of warps in a thread block (W_{cta}), executing with one less thread block would not degrade the performance on average. These additional warps were stalled for resources and the remaining warps were sufficient to maintain high resource utilization. Therefore, W_{cta} is a conservative threshold that guarantees contention of resources if number of warps in X_{alu} or X_{mem} are above it.

Tendency detection: Algorithm 1 shows Equalizer decision process that implements actions mentioned in Table 3.1. If the number of X_{alu} or X_{mem} warps are greater than W_{cta} , the corresponding resource can be considered to have serious contention (lines 7

Algorithm 1 Decision algorithm of Equalizer

```
1:                 ▷ $nMem$, $nALU$ are the number of warps in $X_{alu}$ and $X_{mem}$ state
2:                 ▷ $nWaiting$ is the number of warps in $waiting$ state
3:                 ▷ $nActive$ is the number of active, accounted warps on an SM
4:                 ▷ $W_{cta}$ and $numBlocks$ are # warps in a block and # blocks
5:                 ▷ $MemAction$ and $CompAction$ are frequency changes
6: ------------------------------------------------------------
   ----------
7: if $nMem > W_{cta}$ then                          ▷ Definitely memory intensive
8:     $numBlocks$ = $numBlocks$ - 1
9:     $MemAction$ = true
10: else if $nALU$ ¿ $W_{cta}$ then                  ▷ Definitely compute intensive
11:     $CompAction$ = true
12: else if $nMem > 2$ then                          ▷ Likely memory intensive
13:     $MemAction$ = true
14: else if $nWaiting > nActive/2$ then              ▷ Close to ideal kernel
15:     $numBlocks$ = $numBlocks$ + 1
16:     if $nALU > nMem$ then                        ▷ Higher compute inclination
17:         $CompAction$ = true
18:     else                                         ▷ Higher memory inclination
19:         $MemAction$ = true
20:     end if
21: else if $nActive$ == 0 then
22:     $CompAction$ = true                          ▷ Improve load imbalance
23: end if
```

and 10). If none of the two states have large number of excess warps, Equalizer checks the number of X_{mem} warps to determine bandwidth saturation (line 12). As discussed in Section 3.3.1, having more than two X_{mem} warps indicates bandwidth saturation. If these three conditions (compute saturation, heavy memory contention and bandwidth saturation) are not met, there is a chance that the current combination of the three parameters are not saturating any resources and these kernels are considered to be unsaturated. Kernels in unsaturated category can have compute or memory inclinations depending on large X_{alu} or X_{mem} values (line 16-18). If a majority of the warps are not waiting for data in such cases (line 14), these kernels considered to be degenerate and no parameters are changed.

Equalizer actions: After determining the tendency of the kernel, Equalizer tunes the hardware parameters to achieve the desired energy/performance goals. For compute intensive kernels, Equalizer requests $CompAction$ from the frequency manager (line 11). Equalizer deals with the memory intensive workloads as explained in Section 3.3.1. Whenever Equalizer finds that the number of X_{mem} warps are greater than W_{cta} , it reduces the number of blocks by one (line 8) using the techniques in Section 3.4.2. Reducing the number of concurrent blocks does not hamper the memory kernels and it can help reduce cache contention. However, if the number of X_{mem} warps are greater than two, but less than W_{cta} , Equalizer does not decrease the number of blocks (line 12-13) because reducing number of thread blocks might under-subscribe the memory bandwidth as explained in Section 3.3.1. In both of these cases, Equalizer requests $MemAction$ from the frequency manager. For unsaturated and non-degenerate kernels, Equalizer requests $CompAction$ or $MemAction$ depending on their compute or memory inclination (line 16-18).

$CompAction$ and $MemAction$ in Algorithm 1 refer to the action that should be taken with respect to SM and memory frequency when compute kernels and memory kernels are detected, respectively. As per Table 3.1, for compute kernels, if the objective of Equalizer is to save energy, then it requests the frequency manager in Figure 3.3 to reduce memory VF. If the objective is to improve performance then Equalizer requests increase in the SM's VF. The opposite actions are taken for memory kernels as per Table 3.1.

Equalizer also provides a unique opportunity for imbalanced kernel invocations. If the kernel has a few long running thread blocks that do not distribute evenly across the SMs, then certain SMs might finish early and other SMs will finish late due to load imbalance. To neutralize this, Equalizer tries to finish the work early (line 21) or reduce memory energy. The assumption is that since majority of the SMs are idle and leaking energy, finishing the work early will compensate for the increase in energy due to reduction in leakage energy. In energy saving mode, having lower bandwidth will likely be sufficient for sustaining performance as the majority of the SMs are idle.

3.4 Equalizer Hardware Design

In the baseline Fermi GPU, there are two entries for every warp in the instruction buffer. During every cycle, the warp scheduler selects a warp for execution. In this process, the warp instruction's operands are checked in the scoreboard for readiness. If a warp instruction is ready for execution, it is sent to the issue stage. Equalizer monitors the state of the instructions in the instruction buffer to determine about the state of warps. Figure 3.6 shows the details of Equalizer on an SM. Every 128 cycles, Equalizer checks the head instructions of every warp in the buffer and collects the status of each warp. This process is continued throughout the *epoch* window at which point a decision is made.

3.4.1 Counter Implementation

Whenever a thread block is paused, the entries in the instruction buffer for the warps of that thread block are not considered for scheduling. In Figure 3.6, the dark grey entries in the instruction buffer are not considered for scheduling. The status of only unpaused warps are taken into account in the decision by Equalizer. If a warp is unpaused and it does not have a valid entry in the instruction buffer, it is considered as unaccounted. The four counters needed for Equalizer in Section 3.3 are implemented as follows:

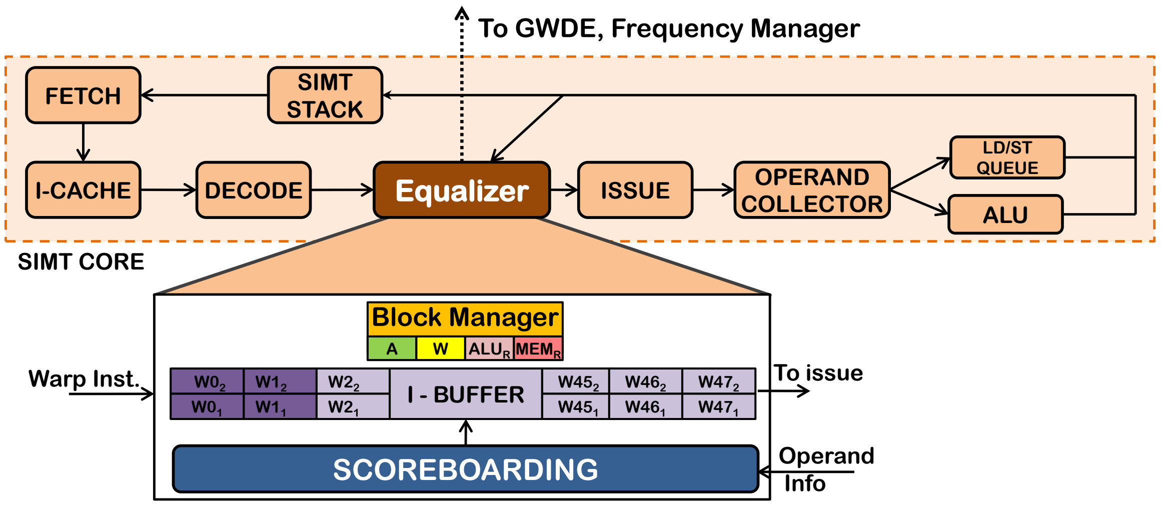


Figure 3.6: Equalizer Overview. Equalizer is attached with the instruction buffer and periodically calculates the four counter values. At the end of an *epoch* it sends a request to the Global Work Distribution Engine.

- Active warps: This counter is implemented by counting the number of warps that are not paused or unaccounted.

- Waiting warps: The head instructions of every warp that cannot be executed because the scoreboard has not made the operands available are considered for this counter.

- X_{alu} : All the head instructions waiting to issue to the arithmetic pipeline that have operands available from the scoreboard are accounted here.

- X_{mem} : All the warps instructions that are ready to access memory but cannot be issue because the LD/ST queue cannot accept an instruction in this cycle are in this category.

3.4.2 Thread Block Allocation Mechanism

After the collection of counter values, at the end of an *epoch* window, Equalizer uses the algorithm explained in Section 3.3.2 and decides to increase, maintain or decrease thread blocks (*numBlocks*). Equalizer does not change *numBlocks* on the basis of one window. If the decision of three consecutive *epoch* window results in different decision than the current *numBlocks* , then Equalizer changes *numBlocks* . This is done to remove spurious temporal changes in the state of the warps by the decision itself. The SM interacts with the GWDE to request more thread blocks whenever required as shown in Figure 3.3. If Equalizer decides to increase the number of thread blocks, it will make a request to GWDE for another thread block. In case, Equalizer decides to reduce the number of thread blocks, it sets the pause bit on the instruction buffer of all warps in that block corresponding warps' instruction buffer. After one of the active thread blocks finishes execution, Equalizer unpauses a thread block from the paused thread block. At this point, Equalizer does not make a new request to the GWDE. This automatically maintains the reduced $numBlocks$.

3.4.3 Frequency Management

At the end of each $epoch$, every SM calculates whether to increase, maintain or decrease its SM or memory system frequency based on the $CompAction$ and $MemAction$ values of Algorithm 1 in conjunction with Table 3.1. The frequency manager shown in Figure 3.3, receives these requests and makes a decision based on the majority vote amongst all the SM requests. If the request is to change SM VF level then the decision is sent to the on-chip or off-chip voltage regulator. If the request is to change the memory system VF level then, the operating points of the entire memory system which includes the interconnect between SMs and L2, L2, memory controller and the DRAM are changed. In this work, three discrete steps for each voltage/frequency domain are considered. The normal state refers to no change in frequency, low state and high state is reduction and increase in nominal frequency by 15% Any increase or decrease in the frequency is implemented in a gradual fashion between the three steps. For example, if the decision is to increase the

Table 3.2: Benchmark Description

Application	**Id**	**Type**	**Kernel Fraction**	**num Blocks**	$\mathbf{W_{cta}}$
backprop(bp)	1	Unsaturated	0.57	6	8
	2	Cache	0.43	6	8
bfs	1	Cache	0.95	3	16
cfd	1	Memory	0.85	3	16
	2	Memory	0.15	3	6
cutcp	1	Compute	1.00	8	6
histo	1	Cache	0.30	3	16
	2	Compute	0.53	3	24
	3	Memory	0.17	3	16
kmeans(kmn)	1	Cache	0.24	6	8
lavaMD	1	Compute	1.00	4	4
lbm	1	Memory	1.00	7	4
leukocyte(leuko)	1	Memory	0.64	6	6
	2	Compute	0.36	8	6
mri-g	1	Unsaturated	0.68	8	2
	2	Unsaturated	0.07	3	8
	3	Compute	0.13	6	8
mri-q	1	Compute	1.00	5	8
mummer(mmer)	1	Cache	1.00	6	8
particle(prtcl)	1	Cache	0.45	3	16
	2	Compute	0.55	3	16
pathfinder	1	Compute	1.00	6	8
sad	1	Unsaturated	0.85	8	2
sgemm	1	Compute	1.00	6	4
sc	1	Unsaturated	1.00	3	16
spmv	1	Compute	1.00	8	6
stencile(stncl)	1	Unsaturated	1.00	5	4

SM frequency in the current *epoch* and the current SM frequency is in the low state then it is change to normal in first step. If in the next *epoch* the same request is made then the frequency is increased from from normal to high.

3.5 Experimental Evaluation

3.5.1 Methodology

Table 3.2 describes the various applications, their kernels and characteristics. These kernels are from the Rodinia suite [14] and parboil [82] suite. We use the kmeans bench-

Table 3.3: Simulation Parameters

Architecture	Fermi (15 SMs, 32 PE/SM)
Max Thread Blocks:Warps	8:48
Data Cache	64 Sets, 4 Way, 128 B/Line
SM V/F Modulation	$\pm 15\%$, on-chip regulator
Memory V/F Modulation	$\pm 15\%$

mark used by Rogers et al. [70] that uses large input. We use GPGPU-Sim [8] v3.2.2, a cycle level GPU simulator and model the Fermi Architecture. The configurations for GTX480 are used. The important configuration parameters of the GPU are shown in Table 3.3.

3.5.1.1 Energy Modelling

We use GPUWattch [51] as our baseline power simulator. We enhance it to enable dynamic voltage and frequency scaling on the SM and the memory. GPUWattch creates a processor model and uses static DRAM coefficients for every DRAM active, precharge and access command. We create five different processor models (normal, SM high, SM low, memory high and memory low) for the five models that are simulated. At runtime, the simulation statistics are passed on to the appropriate processor. The current Kepler GPUs can boost the SM frequency by 15%[55] and we chose 15% as the change in GPU VF level. We assume linear change in voltage for any change in the frequency [63]. Furthermore, we integrate the SM and leakage calculation into GPUWattch for the different processors. We assume the baseline GPU to have a leakage of 41.9W as found by [51]. Leng et. al have shown that the voltage guardbands on GPUs are more than 20% [52] and therefore we reduce both voltage and frequency by 15%.

On the GPU, the SM works on a specific voltage frequency domain and the network

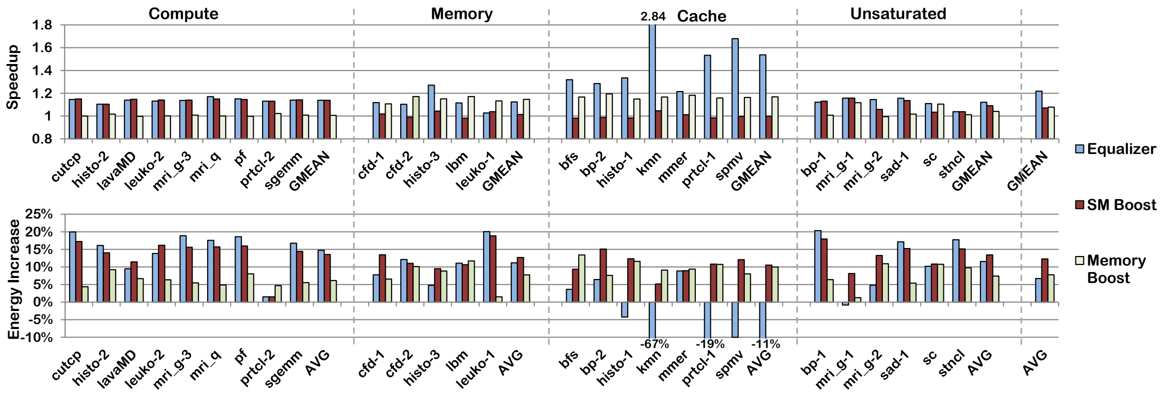


Figure 3.7: Performance and increase in Energy consumption for performance mode

on chip, L2 cache, memory controller and DRAM operate on separate domains. When we change the memory system VF level, we also change the network, L2 and the memory controller's operating point. For all these, GPUWattch uses the integrated McPAT model [79]. For DRAM modelling, we use the various operating points of the Hynix GDDR5 [34]. The major factor that causes the difference in power consumption of DRAM is the active standby power (when the DRAM is inactive due to lack of requests). We use the different values of Idd_{2n} along with the operating voltage, which is responsible for active standby power [12]. Commercial Kepler GPUs allow memory voltage/frequency to change by significant amount but we restrict the changes conservatively to 15%[55]. However, due to lack of public knowledge for this process of the GPUs, we do not compare with these methods.

For the results discussed in Section 3.5.2, we assume an on-chip voltage regulator module (VRM). This module can change the VF level of the SMs in 512 SM cycles. We do not assume a per SM VRM, as the cost may be prohibitive. This might lead to some inefficiency if multiple kernels with different resource requirements are running simultaneously. In such cases, per SM VRMs should be used.

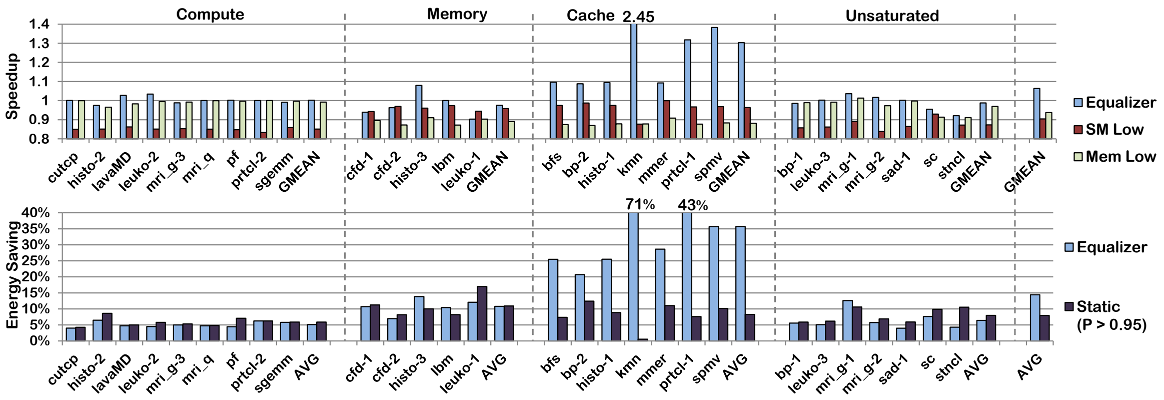


Figure 3.8: Performance and Energy savings in energy mode. Static bar on right for energy savings is either SM low or mem low from the top graph when the performance is above 0.95.

3.5.1.2 Equalizer Power Consumption

Equalizer has two stages in hardware. The first stage collects statistics and the second stage makes a decision. We show that the overall power consumption of Equalizer is insignificant as compared to SM power consumption. For the first part, Equalizer introduces 5 new counters. The active, $waiting$, X_{alu} and X_{mem} counters can have maximum values of 48 (total number of warps) per sample. After sensitivity study, we found that for a 4096 cycle $epoch$ window, the average behavior of the kernel starts to match the macro level behavior and is not spurious. As a sample is taken every 128 cycles for the $epoch$ window, there will be 32 samples in an $epoch$ and hence the maximum value of these counters can be 1536 (48 times 32). So, an 11 bit counter is sufficient for one counter. A cycle counter which will run for 4096 cycles and reset is also needed for Equalizer. Overall, 4 11-bit counters, and 1 12-bit counter are needed for the statistics collection process. We expect this area overhead to be insignificant as compared to the overall area of 1 SM, which includes 32 FPUs, 32768 registers, 4 SFUs, etc.

The Equalizer decision process is active only once in an $epoch$ window of 4096 cycles. The cycle counter explained above will signal the Equalizer decision process to begin, but as the process happens infrequently, we assume the decision calculation to consume insignificant amount of energy.

3.5.2 Results

We show the results of Equalizer in performance mode and energy mode for the four kernel categories in Figure 3.7 and 3.8 as compared to the baseline GPU. The top chart in both the figure shows the performance and the bottom chart shows the impact on energy. The bars show the data for Equalizer, changing SM and memory frequency statically by 15%. At runtime, equalizer modulates the number of threads and either changes SM frequency or memory frequency depending on the kernel's present requirements.

Performance Mode: The results for performance mode in Figure 3.7 show that Equalizer makes the right decision almost every time and matches the performance of the best of the two static operating points for compute and memory kernels. The performance improvement of Equalizer for compute kernels is 13.8% and for memory kernels a 12.4% speedup is achieved, showing proportional returns for the 15% increase in frequency. The corresponding increase in energy for Equalizer is shown in the bottom chart in Figure 3.7. For compute and memory kernels this increase is 15% and 11% respectively as increasing SM frequency causes more energy consumption. The increase in overall GPU energy is not quadratic when increasing SM frequency as a large fraction of the total energy is due to leakage energy which does not increase with SM VF level. Equalizer is unable to detect the right resource requirement for leuko-1 kernel as it heavily uses texture caches which can handle a lot more outstanding request than regular data caches. This results in large number of requests going to the memory subsystem and saturating it without the back pressure

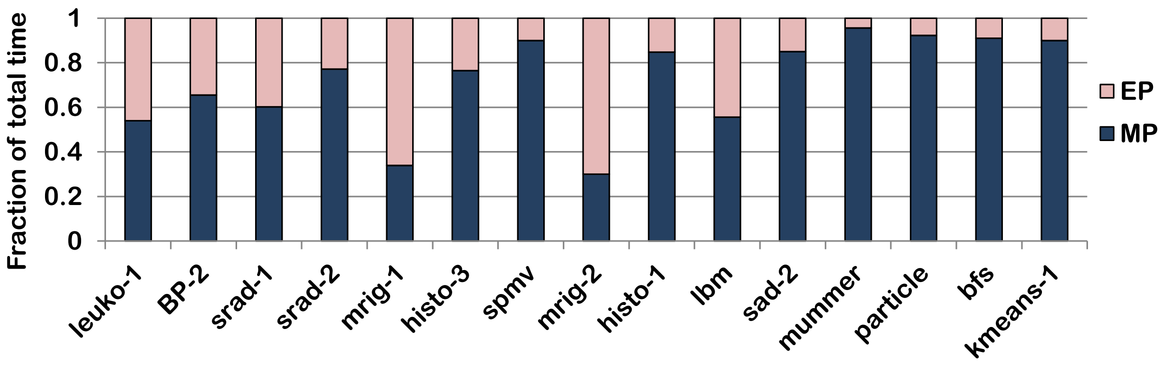


Figure 3.9: Distribution of time for the various SM and memory frequency. P and E are data for performance and energy mode respectively

being visible to LD/ST pipeline.

For cache sensitive kernels, the geometric mean of the speedup is 54%. Apart from the 17% improvement that these kernels receive by boosting memory frequency alone as shown by the memory boost geometric mean, the reduction in cache miss rate is the major reason for the speedup. The reduction in cache miss rate improves performance as the exposed memory latency is decreased and it also reduces the pressure on the bandwidth thereby allowing streaming data to return to the SM faster. The large speedup leads to improved performance and less leakage as the kernel finishes significantly faster and there is an overall reduction in energy by 11%. In fact, Kmeans achieves a speedup of 2.84x and up to 67% reduction in energy in the performance mode.

Among the unsaturated kernels, Equalizer can beat the best static operating point in mri_g-1, mri_g-2, sad-1 and sc kernels due to its adaptiveness. Equalizer captures the right tendency for bp-1 and stncl. prtcl-2 in the compute kernels has load imbalance and only one block runs for more than 95% of the time. So, by accelerating the SM frequency, we save significant leakage leading to less overall energy increase even after increasing frequency. Overall, Equalizer achieves a speedup of 22% over the baseline GPU with 6%

higher energy consumption. Always boosting SM frequency leads to 7% speedup with 12% energy increase and always boosting memory frequency leads to 6% speedup with 7% increase in energy. So, Equalizer provides better performance improvement at a lower increase in energy.

Energy Mode: The results for energy mode are shown in Figure 3.8. SM Low and Mem Low denote the operating points with 15% reduction in frequency for SM and memory respectively. As shown in the figure, Equalizer adapts to the type of the kernel and reduces memory frequency for compute kernels and SM frequency for memory kernels and as the bottlenecked resource is not throttled, the loss in performance for compute and memory kernels is 0.1% and 2.5% respectively. For cache sensitive kernels, reducing thread blocks improves the performance by 30% due to reduced L1 data cache contention. This improvement is lower in energy mode as compared to performance mode because instead of increasing the memory frequency, Equalizer decides to lower the SM frequency to save energy. Even for the unsaturated kernels, reducing frequency by Equalizer loses performance only in stncl. This is because it has very few warps in X_{mem} or X_{alu} state as shown in Figure 3.4 and hence decreasing frequency of any of the resources makes that resource under perform. Overall, the geometric mean of performance shows a 5% increase for Equalizer whereas lowering SM voltage/frequency and memory voltage/frequency by 15% leads to a performance loss of 9% and 7% respectively.

In the bottom chart of Figure 3.8, only two bars are shown. The first bar shows the energy savings for Equalizer and the second bar shows the same for either SM low or mem low from the above chart, depending on which parameter results in no more than 5% performance loss and is called static best. The savings of Equalizer for compute kernels is 5% as

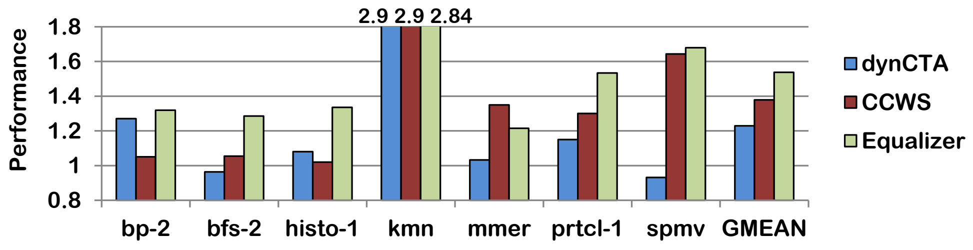
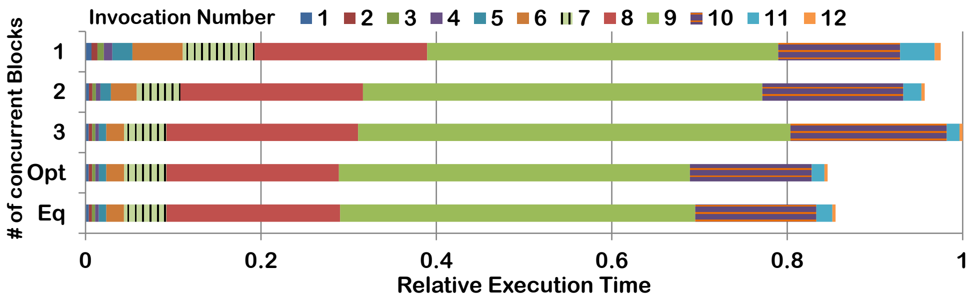


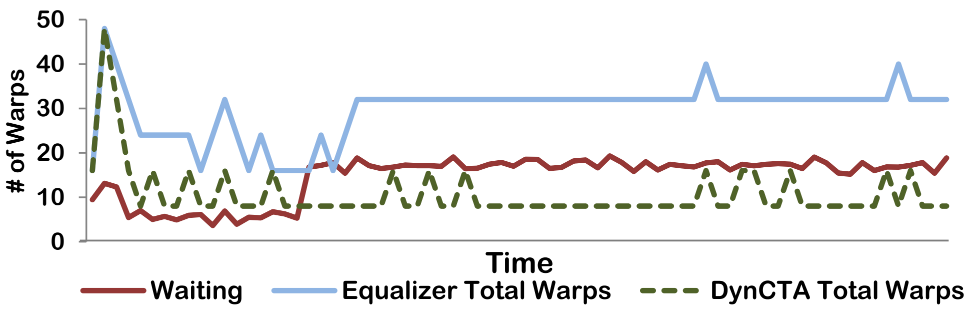
Figure 3.10: Comparison of Equalizer with DynCTA and CCWS

reducing memory frequency cannot provide very high savings. However, memory kernels save 11% energy by lowering SM frequency. For cache sensitive kernels the energy savings for Equalizer is 36%, which is larger than performance mode's energy savings. This is due to throttling SM frequency rather than boosting memory frequency. For unsaturated kernels the trend of their resource utilization is effectively exploited by Equalizer with 6.4% energy savings even though there was an overall performance loss of 1.3%. Overall Equalizer dynamically adjusts to the kernel's requirement and saves 15% energy without losing significant performance, while the static best voltage/frequency operating point saves 8% energy.

Equalizer Analysis: To analyze the reasons for the gains of Equalizer in energy and performance mode, we show the distribution of various SM and memory frequency operating points for all the kernels in Figure 3.9. Normal, high and low modes are explained in Section 3.4.3. The P and E below each kernel denotes the distribution for performance and energy mode for a kernel respectively. For compute kernels, the P mode mostly has SM in high frequency and in E mode the memory is in low frequency. Similarly for cache and memory kernels the memory is at high frequency in P mode and SM is in low frequency in E mode. Kernels like histo-3, mri_g-1, mrig_g-2, and sc in the unsaturated category exploit the different phases and spend time in boosting both the resources at different phases.



(a) Adaptiveness of Equalizer across invocations of bfs-2



(b) Adaptiveness of Equalizer within an invocation for spmv

Figure 3.11: Adaptiveness of Equalizer

We compare the performance of Equalizer with DynCTA [40], a heuristics based technique to control the number of thread blocks and CCWS [70], which controls the number of warps that can access data cache in Figure 3.10. We show results only for cache sensitive kernels as these two techniques are mostly effective only for these cases. DynCTA and CCWS get speedup up-to 22% and 38% respectively. The reason why Equalizer is better than DynCTA for cache sensitive kernels is explained later with Figure 3.11. While CCWS gets better speedup than Equalizer in mmer, Equalizer gets 16% better performance than CCWS. The performance of CCWS is sensitive to the size of the victim tags and the locality score cutoffs and is not as effective on the kernels that are not highly cache sensitive.

Equalizer adaptiveness: The impact of adaptability of Equalizer is demonstrated for inter-instance variation and intra instance variations in Figure 3.11. The stacked bars for 1, 2, 3 and the optimal number of blocks were shown in Figure 3.2a. Another bar for

Equalizer is added for comparison to that figure and shown in Figure 3.11a. To analyze only the impact of controlling the number of blocks we do not vary the frequencies in this experiment. The optimal solution for this kernel would change number of blocks after instance 7 and 10. Equalizer makes same choices but needs more time to decrease the number of blocks as it must wait for 3 consecutive decisions to be different from the current one to enforce a decision. Overall, the performance gain of Equalizer is similar to the Optimal solution.

Figure 3.11b shows the adaptiveness of Equalizer within an instance of the spmv kernel. We also compare the dynamism of Equalizer with DynCTA [40]. Initially, spmv has low compute and high cache contention and therefore less warps are in the waiting state. Both Equalizer and DynCTA reduce the number of threads to relieve the pressure on the data cache. After the initial phase, the number of warps waiting for memory increases as shown in the figure. While Equalizer adjusts to this nature and increases number of threads, DynCTA's heuristics do not detect this change and thus do not increase threads resulting in poor performance. Performance differences between the two for cache kernels is due to the better adaptiveness of Equalizer as it can accurately measure resource utilization as shown for spmv. However, for kernels with stable behavior, performance of Equalizer and DynCTA is closer (e.g., bp-2 and kmn).

3.6 Related Work

Managing number of threads for cache locality: Kayiran et al. propose dynCTA [40] which distinguishes stall cycles between idle stalls and waiting related stalls. Based on these stalls, they decide whether to increase or decrease thread blocks. While dynCTA uses these stall times as heuristics to determine bandwidth consumption, Equalizer uses micro-architectural quantities such as number of waiting warps and X_{mem} warps. Rogers et al. [70] propose CCWS to regulate the number of warps that can access a cache on the basis of a locality detecting hardware in the cache. While they introduce duplication of tags in the cache and changes in scheduling, Equalizer relies on relatively simpler hardware changes. We compare with the two techniques quantitatively in Section 3.5.2. Unlike [40, 70], we also target efficient execution for kernels that are not cache sensitive. Jog et al. proposed OWL Scheduling [39] which improves the scheduling between warps depending on several architectural requirements. While their work focuses on scheduling between different warps on an SM, Equalizer emphasizes on selecting three architectural parameters. None of these techniques boost memory frequency for cache sensitive kernels which can boost the performance significantly. Lee et al. [47] propose a thread level parallelism aware cache management policy for integrated CPU-GPU architectures.

DVFS on GPU: Lee and Satisha et al. [49] have shown the impact of DVFS on GPU SMs. They provide a thorough analysis of the various trade-offs for core, interconnect and number of SMs. However, they only show the benefits of implementing DVFS on GPU without describing the detailed process of the runtime system. They do not focus on memory side DVFS at all. Leng et al. [51] further improves on this by decreasing SM voltage/frequency by observing large idle cycles on the SM similar to what has been done on CPUs [36, 10, 44]. Equalizer provides a more comprehensive and fine-grained runtime system that is specific to GPU and looks at all warps in tandem to determine the suitable core and memory voltage and frequency setting. Commercial NVIDIA GPUs provide control for statically and dynamically boosting the core and memory frequency using the Boost and Boost 2.0 [65, 64] technology. However, these are based on the total power budget remaining and the temperature of the chip. Equalizer looks at key micro-architectural counters to evaluate the exact requirements of the kernel and automatically scale hardware resources.

Resource Management on CPU: There has been numerous work on managing DVFS on CPU systems [29, 36, 22] and is deployed by several commercial processors. They depend on measurement and feedback of the current workload by measuring certain hardware characteristics such as performance, cache hits etc. to decide on the future voltage/frequency operating point. There also has been significant work in managing the number of threads to run on multi-processor systems [11, 50, 83]. While some adaptations of these techniques might be effective on GPUs, Equalizer provides a system which is lightweight, does not depend on cross core communication and manages number of threads, core-side and memory side voltage/frequency in a unified fashion.

DVFS on Memory: DVFS of DRAM has been studied by Deng et al. [18, 19]. They show the significance of low power states in DRAM for reducing background, memory controller and active power by reducing the operational frequency of the memory subsystem. While they focus on servers, their inferences are valid for GPUs as well.

Energy Efficiency on GPU: Abdel-Majeed et al. [3] add a trimodal MTCMOS switch to allow the SRAM register file to be in on, off and drowsy and select the states at runtime depending on register utilization. Abdel-Majeed et al. also show energy savings on GPU by selectively power gating unused lanes in an SM [4]. Gilani et al. [28] show the benefits of using scalar units for computations that are duplicated across threads. Gebhart et al. [26] propose a register file cache to access a smaller structure rather than the big register file.

All these techniques target different aspects of GPU under-utilization than what is targeted by Equalizer.

3.7 Summary

The high degree of multi-threading on GPUs leads to contention for resources like compute units, memory bandwidth and data cache. Due to this contention, the performance of GPU kernels will often saturate. Furthermore, this contention for resources varies across kernels, across invocations of the same kernel, and within an invocation of the kernel. In this work, we present a dynamic runtime system that observes the requirements of the kernel and tunes number of thread blocks, SM and memory frequency in a coordinated fashion such that hardware matches kernel's requirements and leads to efficient execution. By observing the state of the warps through four new hardware performance counters, Equalizer dynamically manages these three parameters. This matching of resources on the GPU to the kernel at runtime leads to an energy savings of 15% or 22% performance improvement.

CHAPTER IV

Mascar: Speeding up GPU Warps by Reducing Memory Pitstops

In this chapter we show that the majority of the applications that get low utilization on the GPU are memory and cache intensive. So I focus on providing architectural solutions to improve the performance of these kinds of GPU programs. While GPUs provide large numbers of compute resources, the resources needed for memory intensive workloads are more scarce. Therefore, managing access to these limited memory resources is a challenge for GPUs. I describe a novel Memory Aware Scheduling and Cache Access Re-execution (Mascar) system on GPUs tailored for better performance for memory intensive workloads. This scheme detects memory saturation and prioritizes memory requests among warps to enable better overlapping of compute and memory accesses. Furthermore, it enables limited re-execution of memory instructions to eliminate structural hazards in the memory subsystem and take advantage of cache locality in cases where requests cannot be sent to the memory due to memory saturation.

4.1 Introduction

With the popularization of high level programming models such as CUDA [62] and OpenCL [43], there has been a proliferation of general purpose computing on GPUs to exploit their computational power. While OpenCL and CUDA enable easy portability to GPUs, significant programmer effort is necessary to optimize the programs and achieve near peak performance of the GPU [72, 74]. This effort is increased when data parallel workloads that are memory intensive are targeted for acceleration on GPUs due to their parallel processing paradigm and are still far from achieving the potential performance offered by a GPU.

Modern GPUs utilize a high degree of multi threading in order to overlap memory accesses with computation. Using the Single Instruction Multiple Thread (SIMT) model, GPUs group many threads that perform the same operations on different data into warps, and a warp scheduler swaps warps that are waiting for memory accesses for those that are ready for computation. One of the greatest challenges that prevents achieving peak performance on these architectures is the lack of sufficient computations. When a memory intensive workload needs to gather more data from DRAM for its computations, the resources for managing outstanding memory requests (at the L1, L2, and memory subsystem) become saturated. Due to this saturation, a new memory request can only be sent when older memory requests complete. At this point, subsequent accesses to the memory can no longer be pipelined, resulting in serialized accesses. In such a scenario, the computation portion of any of the parallel warps cannot begin, as all the memory requests needed to initiate the computation have been serialized. Furthermore, the amount of computation to hide the latency of the unpipelined memory requests is much larger. For such memory constrained workloads, a GPU's compute units are forced to wait for the serialized memory requests to be filled, and the workloads cannot achieve high throughput.

Another of memory intensive applications saturating memory subsystem resources is the reduction in data reuse opportunities for the L1 cache. Due to the saturated memory pipeline, including the Load Store Unit (LSU), warps whose data are present in L1 cannot issue memory instructions. This delay in utilizing the already present data may result in the eviction of the reusable data by other warp's data being brought in to the cache. This leads to increased cache thrashing, forcing more DRAM requests to be issued, thus worsening the serialization of memory accesses. While ideal GPU workloads tend to have very regular, streaming memory access patterns, recent research has examined GPU applications that benefit from cache locality [70, 47] and have more non-streaming accesses. If this data locality is not exploited, cache thrashing will occur, causing performance degradation.

In this work, we establish that the warp scheduler present in a GPU's Streaming Multiprocessor (SM) plays a pivotal role in achieving high performance for memory intensive workloads, specifically by prioritizing memory requests from one warp over those of others. While recent work by Jog et al. [39] has shown that scheduling to improve cache and memory locality leads to better performance, we stipulate that the role of scheduling is not limited to workloads which have such locality. We show that scheduling is also critical in improving the performance of many memory intensive workloads that do not exhibit data locality.

We propose Memory Aware Scheduling and Cache Access Re-execution (Mascar) [77] to better overlap computation and memory accesses for memory intensive workloads. The intuition behind Mascar is that when the memory subsystem is saturated, all the memory requests of one warp should be prioritized rather than sending a fraction of the required requests from all warps. As the memory subsystem saturates, memory requests are no longer pipelined and sending more requests from different warps will delay any single warp from beginning computation. Hence, prioritizing all requests from one warp allows this warp's data to be available for computation sooner, and this computation can now overlap with the memory accesses of another warp.

While Mascar's new scheduling scheme enables better overlapping of memory accesses with computation, memory subsystem saturation can also prevent reuse of data in the L1 cache. To ameliorate this, we propose to move requests stalled in LSU due to memory back pressure to a re-execution queue where they will be considered for issuing to the cache at a later time. With such a mechanism, the LSU is free to process another warp whose requested addresses may hit in the cache. Re-execution can both improve cache hit rates by exploiting such hits under misses, allowing the warp to now execute computation, as well as reduce back pressure by preventing this warp from accessing DRAM twice for the same data.

This paper makes the following contributions:

- We analyze the interplay between workload requirements, performance, and scheduling policies. Our results show that the choice of scheduling policy is critical for memory intensive workloads, but has lesser impact on the performance of compute intensive workloads.

- We propose a novel scheduling scheme that allows better overlapping of computation and memory accesses in memory intensive workloads. This scheme limits warps that can

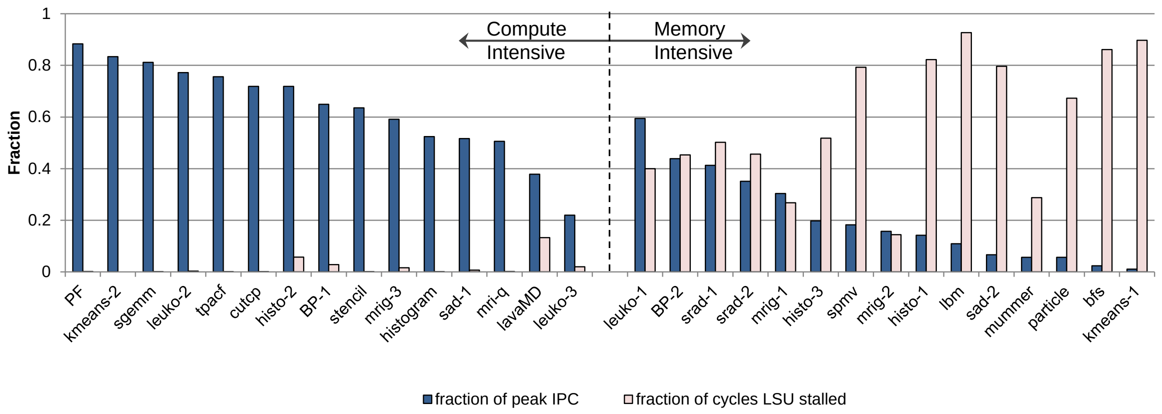


Figure 4.1: Fraction of peak IPC achieved and the fraction of cycles for which the LSU stalled due to memory subsystem saturation.

simultaneously access memory with low hardware overhead.

- We also propose a memory instruction re-execution scheme. It is coupled with the LSU to allow other warps to take advantage of any locality in the data cache, when the LSU is stalled due to memory saturation.
- An evaluation of Mascar on a model of the NVIDIA Fermi architecture achieves 34% performance improvement over baseline GPU for workloads sensitive to scheduling while reducing average energy consumption by 12%.

4.2 Motivation

To identify data parallel application kernels that suffer due to the memory subsystem back pressure issues described in Section II, we classify kernels from the Rodinia [14] and Parboil [82] benchmark suites as compute or memory intensive. For each of these kernels, Figure 4.1 shows the fraction of the theoretical peak IPC achieved (left bar) and the fraction of cycles for which the SM's LSU is forced to stall (right bar) due to memory

[1]The methodology for this work is detailed in Section 4.4.1.

subsystem saturation[1]. The kernels are considered to be compute intensive if the number of instructions executed on an SM per L1 cache miss is greater than 30, otherwise it is considered to be memory intensive.

Of the 30 kernels in the two benchmark suites, 15 are in the compute intensive category whereas 15 are considered memory intensive. The kernels are arranged in decreasing order of their fraction of peak IPC achieved within their category. While 13 out of 15 kernels in the compute intensive category achieve more than 50% of the peak IPC for compute intensive kernels, only one out of the 15 kernels in the memory intensive category achieves 50% of the peak IPC. In fact, eight kernels achieve less than 20% of the peak performance in this category, whereas no kernel in the compute intensive category suffers from such low performance.

Figure 4.1 illustrates the strong correlation between the performance achieved and the memory demands of the GPU workloads. When examining the impact of memory subsystem back pressure on these applications, we observed that compute intensive kernels rarely stall the LSU. This results in the timely processing of memory requests required to initiate their computation. On the other hand, the memory intensive kernels show a significant rise in the number of LSU stall cycles due to growing back pressure in the memory subsystem. This stalling hampers the GPUs ability to overlap memory accesses with computation. The percent of LSU stall cycles seen in Figure 4.1 for the memory intensive workloads is indicative that that these workloads struggle to achieve peak throughput primarily due to saturation of the memory subsystem. The stalled requests in the LSU can only be issued when a prior request is returned from memory, forcing serialization of future memory requests. Due to this serialization, data required for the computation portion of these workloads takes

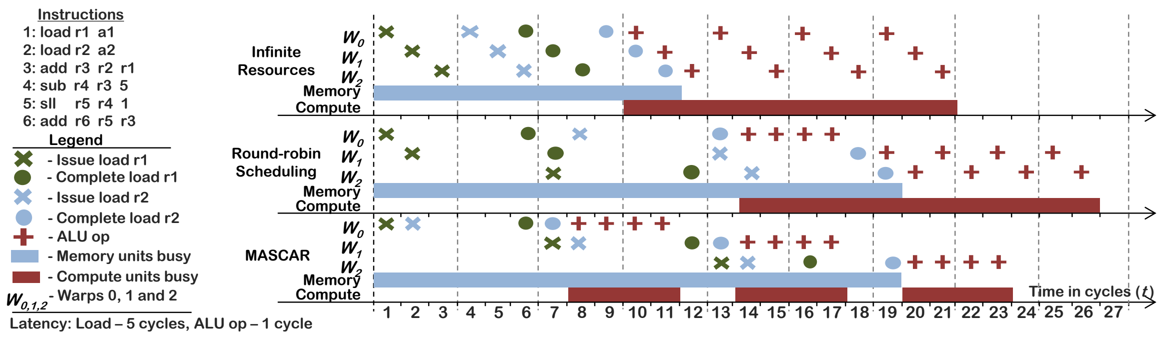


Figure 4.2: Execution timeline for three systems. The top illustrates an infinite bandwidth and MSHR system. The middle shows the baseline system with round-robin scheduling. The timeline on the bottom depicts Mascar's scheduling.

longer to arrive at the SM, which in turn reduces the ability to overlap memory access and computation.

4.2.1 Impact of Scheduling on Saturation

Warp scheduling can have a significant impact on how many memory accesses can be overlapped with computation. Figure 4.2 shows execution timelines for an example workload with three warps run on three architectures. For the sake of simplicity, in this example we assume that each arithmetic instruction takes one cycle, load instruction takes five cycles and that there are three warps that can execute in parallel. In this miniature system, only one warp can issue an arithmetic or memory operation per cycle. The example workload is shown on the left in the figure. In this workload, the first compute operation occurs on line 3 and it cannot begin until both of the loads finish execution. The top timeline illustrates theoretical GPU hardware that has infinite bandwidth and MSHRs, and uses a round-robin warp scheduler. The second timeline shows the execution for a system with support for two outstanding memory requests and also uses round-robin scheduling. The final timeline demonstrates how Mascar's scheduling works with hardware that supports two outstanding requests.

Infinite resources: When the system has infinite resources with round-robin scheduling, the load for r1 is launched for each of the three warps over three consecutive cycles. At $t = 4$, the load unit is ready for the next load instruction, so the load for r2 gets issued for all three warps in a similar fashion. After five cycles, the load for r1 for W_0 returns, and because there were enough MSHRs and the memory requests were fully pipelined, the loads for r1 for W_1 and W_2 complete in the next two cycles. At $t = 9$, the load for r2 returns and the computation can finally begin for W_0 in the next cycle. This computation is completed by $t = 13$. As only one warp's instruction can be executed per cycle, the computation for all three warps takes 12 cycles for the workload's four add instructions due to round-robin scheduling. This theoretical system finishes execution in 21 cycles for this synthetic workload.

Round-robin with two outstanding requests: The performance of this system is hindered due to it only supporting a limited number of outstanding memory requests. The first two cycles behave similar to the infinite resource case, but in the third cycle, the memory request cannot reserve an MSHR and thus has to wait to be sent to memory until one of the first two requests returns. As the first request comes back at $t = 6$, W_2's load for r1 must be issued at $t = 7$. At $t = 8$, W_0's load for r2 is issued, delaying computation until this load completes at $t = 14$. This computation can hide the memory access of W_1 and W_2's loads of r2. These loads of r2 return one after another at $t = 18$ and $t = 19$. The computation for both of the warps takes 10 cycles to complete as only one warp can execute per cycle in round-robin fashion. The overlap of memory accesses with computation is shown by the bands for memory and compute, respectively. It can be seen that the compute operations are ineffective in hiding any memory accesses until $t = 13$ as none of the data required by any warp is available. The execution of this workload finishes at $t = 26$, five cycles later than the theoretical system with infinite MSHRs.

Mascar with two outstanding requests: Whenever Mascar detects memory subsystem saturation, rather than selecting instructions from warps in a round-robin fashion, it prioritizes memory requests of a single warp. In the illustrated example, the workload has been running for some time such that saturation occurs at $t = 1$. Mascar detects this situation and prioritizes memory requests made by W_0 to be issued at $t = 1$ and $t = 2$. No other memory requests can be sent as the system can only handle two outstanding requests at a time. When the data returns for the first load at $t = 6$, W_1 is given priority to issue its memory requests for load of r1 in the next cycle. At $t = 7$, W_0's memory request for the load for r2 returns and the warp is now ready for execution. W_0's computation can begin simultaneously with W_1's next memory request at $t = 8$. This computation from cycles 8 to 11 completely overlaps with memory accesses, which was not possible when using the round-robin scheduler in either of the previously mentioned architectures. Similarly, the computation for W_1 begins at $t = 14$ and overlaps with memory accesses until it completes at $t = 17$. The program finishes execution at $t = 23$, and due to the increased overlap of computation with memory access, the workload running on Mascar finishes earlier than the traditional scheduler.

4.2.2 Increasing Memory Resources to Resolve Saturation

The most intuitive way to resolve memory resource saturation is to provision the memory subsystem with more resources to handle such demands. To model this, we ran memory intensive kernels on a simulator modeling a NVIDIA Fermi GTX 480 GPU modified to have *a*) a large number of L1 and L2 MSHRs and large queues throughout the subsystem (all sized at 10240); *b*) fully associative L1 and L2 caches; *c*) increasing the memory frequency by 20%; *d*) doubling the L1; and *e*) a combination of *(a)*, *(b)*, and *(c)*. Figure 4.3 shows the performance improvements achieved over an unmodified GTX 480 when using these extra provisions. While a combination of all these improvements yields decent speedups for some workloads and modest results for others (Geo-mean of 39% speedup), such a system is extraordinarily hard to realize. This system will have high complexity, energy consumption and is dependent on future technology. Adding MSHRs is very costly due to the complexity of the necessary content-addressable memories [66]. Searching through a fully associative cache's tag array requires significant power just to improve hit rates. Memory bandwidth is not scaling as quickly as GPU computational throughput, and increasing the frequency will also exacerbate issues of GPU energy consumption and thermal power dissipation [41]. Doubling the cache of 15 SMs can lead to a significant area overhead. As provisioning GPUs with additional or faster memory resources is prohibitive, we elect to focus on better overlapping memory accesses with computation through a new warp scheduling scheme and minimal hardware additions.

4.2.3 Impact of Memory Saturation on Data Reuse

The stalling of the LSU during memory saturation also has an impact on the ability of the warps to reuse data present in the cache. As the LSU is stalled due to the back pressure from memory subsystem, warps whose data might be present in the data cache cannot access it. Figure 4.4 shows the percentage of cycles when the core could not issue any

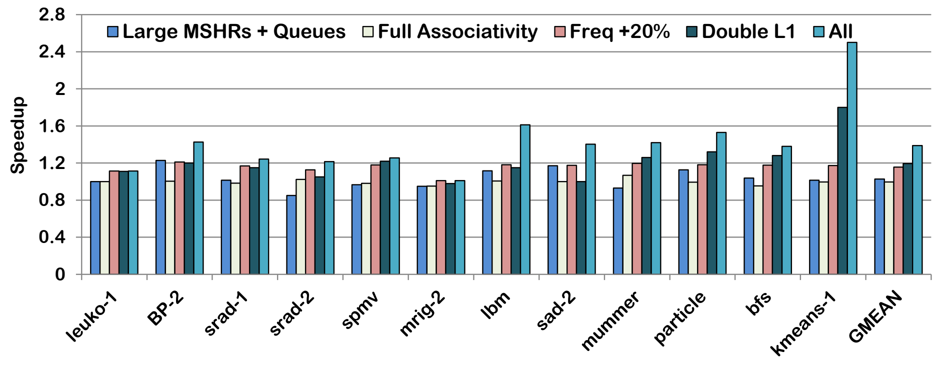


Figure 4.3: Performance impact on memory intensive kernels when increasing the number of MSHRs and memory subsystem queues, fully associative L1 and L2 caches, increasing the DRAM frequency by 20%, doubling the L1 and a combination of all four.

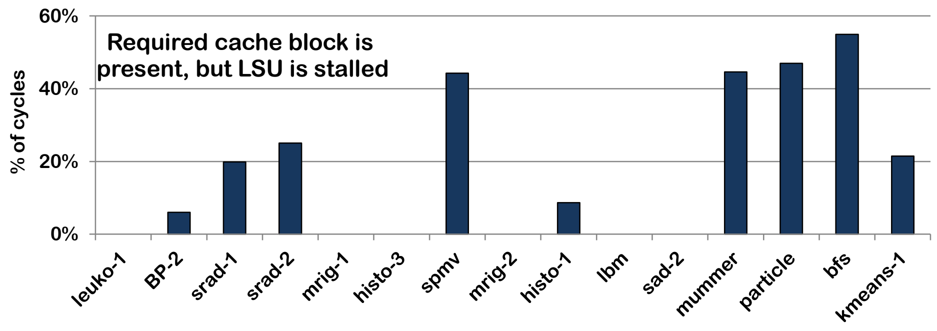


Figure 4.4: Fraction of cycles for which there is at least one warp which has one cache block present in the cache that cannot be accessed due to a stalled LSU.

instruction for execution, while the data required by at least one of the warps was present in the cache. It shows that seven of the fifteen kernels have at least one warp's data present in the cache for 20% of the time during saturation. These warps do not need any memory resources as their data is already present in the data cache if a mechanism exists to allow these warps to exploit such hits-under-misses, there is a significant opportunity to improve performance for these kernels through cache reuse.

4.3 Mascar

To ameliorate the dual problems of slowed workload progress and the loss of data cache locality during periods of memory subsystem back pressure, Mascar introduces a new, bimodal warp scheduling scheme along with a cache access re-execution system. This section details these contributions, and Section 4.4 evaluates their efficacy.

To reduce the effects of back pressure on memory intensive workloads by improving the overlap of compute and memory operations, Mascar proposes two modes of scheduling between warps. The first mode, called the *Equal Priority (EP)* mode, is used when the memory subsystem is not saturated. In this case, Mascar follows the SM's traditional warp scheduling scheme where all warps are given equal priority to access the memory resources. However, if the memory subsystem is experiencing heavy back pressure, the scheduler will switch to a *Memory Access Priority (MP)* mode where one warp is given priority to issue all of its memory requests before another warp can issue any of its requests.

The goal of giving one warp the exclusive ability to issue its requests when scheduling in MP mode is to schedule warps to better overlap computation and memory accesses. By doing so, Mascar is able to reduce the impact of performance bottlenecks caused by saturation of resources in the memory subsystem. As discussed in Section 4.2.1, round-robin scheduling permits all warps to issue *some* of their memory requests, but none can continue with their execution until *all* of their requests are filled. This also holds true for state of the art schedulers, e.g. GTO. To prevent this from occurring, Mascar's MP mode gives one warp priority to issue all of its requests while other warps must wait. This makes all of the prioritized warp's data available for computation sooner than in conventional scheduling approaches. As this warp can now compute on its data, another warp can be given priority to access memory, thus increasing the likelihood that computational resources are used even as many other warps wait on their data.

The detection of memory subsystem saturation is accomplished by a signal from the SM's L1 cache, called the memory saturation flag. The details of the logic used to determine the flag's value are explained later in this section. Once this back pressure signal is asserted, the SM switches scheduling from EP to MP mode. If the back pressure is relieved over time, the saturation flag will be cleared and Mascar switches the scheduler back to EP mode, allowing for rapid and simple toggling between Mascar's two modes of scheduling.

To exploit data locality that might be present during periods of memory saturation, Mascar couples a re-execution queue with the LSU. By providing a means for warps to access the L1 data cache while other memory resources are saturated, this queue allows an SM to exploit hit-under-miss opportunities as warps with accesses that might hit in the L1 can run ahead of other stalled accesses. If the access misses in the L1 the system's memory resources have saturated such that the request cannot reserve an MSHR, the request is pushed onto the re-execution queue and its access will be retried at a later time. This reduces the delay a warp incurs between accessing data present in the L1 and when the warp can start computation. Section 4.3.3 provides implementation details and an example of Mascar's cache access re-execution in action.

4.3.1 Scheduling in Memory Access Priority (MP) Mode

In the baseline SM, after an instruction is decoded it is put into an instruction buffer. Once the instruction is at the head of this buffer and its operands are ready, the scheduler

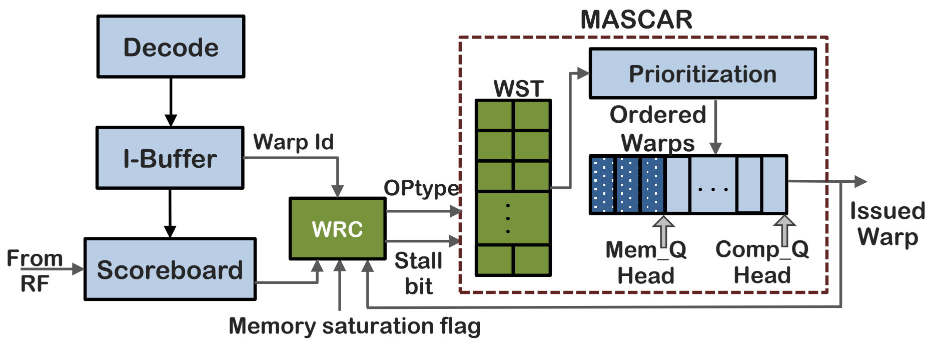


Figure 4.5: Block diagram of the Mascar scheduler. The WRC and WST are new hardware structures that interact with the front-end of the SM's pipeline.

will add the instruction's warp to a queue of ready warps according to its scheduling policy. Mascar alters this step by gathering these warps into separate memory-ready and compute-ready warp queues as shown in Figure 4.5. This allows Mascar to give priority to one memory-ready warp to issue to the LSU and generate its memory requests while stalling all other warps waiting on this resource.

Identifying the memory warp to issue: To track which memory-ready warp should be issued to the LSU, Mascar uses a Warp Status Table (WST) that stores two bits of information per warp. The first bit indicates whether a warp's next instruction will access memory, and the second tells the scheduler to stall issuing of the warp's instruction.

The state of a warp's bits in the WST are determined by the Warp Readiness Checker (WRC). To set a WST entry's memory operation bit, the WRC simply examines the instruction buffer to determine whether or not each warp's next instruction will access memory and sets the bit accordingly. To set a stall bit in the WST, the WRC must first determine which warp is given exclusive ownership and access to the LSU. This warp is called the *owner* warp, and the details of managing ownership are described in *owner warp management*. If a warp's next instruction needs to access memory but it is not the owner warp, the

WRC sets its stall bit in the WST. A warp's stall bit will also be set if the scoreboard indicates that an operand needed by that warp's memory or compute instruction is not ready. If a new warp is granted ownership, the stall bits are updated according to the aforementioned process. If during execution the memory back pressure is relieved and the scheduler switches from MP back to EP mode, all stall bits are cleared.

Owner warp management: The owner warp continues to issue all of its memory requests through the LSU as memory resources become available. It does so until it reaches an instruction which is dependent on one of the issued loads. At this point it relinquishes its ownership. In order to identify when an operation is dependent on a long latency load, each scoreboard entry is augmented with one extra bit of metadata to indicate that its output register is the result of such an instruction. The WRC, shown in Figure 4.5, requests this dependence information from the scoreboard for each instruction belonging to the owner warp, and the scoreboard finds the disjunction of all this new metadata for this instruction's operands. When the WRC is informed that the owner warp's instructions are now waiting on its own loads, the WRC relieves this warp of ownership and resets all other warps' stall bits in the WST. Now that all warps are free to issue to memory, one will go ahead and access memory. If the memory saturation flag remains asserted and the scheduler remains in MP mode, this warp will become the new owner.

Warp prioritization: Mascar prioritizes warps into two groups. The first group is for warps that are ready to issue to the arithmetic pipeline and are called compute-ready warps. Conversely, the second group of warps are called memory-ready warps, and are warps which are ready to be issued to the memory pipeline. These groups are illustrated by the unshaded and shaded regions, respectively, of the ordered warps queue shown in

Figure 4.5. When scheduling in MP mode, compute-ready warps are given priority over memory-ready warps to allow a maximum overlap of computation with memory accesses during periods of heavy back pressure in the memory subsystem. Within these groups, the oldest warp will be scheduled for issue to their respective pipelines.

Once Mascar switches from EP to MP mode, warps that do not have ownership status will no longer be able to issue memory instructions to the LSU. However, earlier instructions from such warps might already be present in the memory-ready warps queue. If Mascar does not allow these warps to issue, the owner warp's memory instructions will not be able to reach the head of the queue, preventing forward progress. To address this potential bottleneck, Mascar allows these non-owner, memory-ready warps to issue to the L1 data cache. If a non-owner's request hits in the L1, its data returns, and the instruction can commit. Otherwise, the L1 will not allow this non-owner's request to travel to the L2, and instead returns a negative acknowledgement. Mascar informs the L1 which warp has ownership, allowing the L1 to differentiate between requests from owner and non-owner warps. Negative acknowledgements may still cause the LSU to stall when non-owner warps get stuck waiting for data to return, but Mascar overcomes this limitation with cache access re-execution, described in Section 4.3.3.

Multiple schedulers: Mascar's scheduling in MP mode allows one warp to issue its memory accesses at a time, but modern NVIDIA GPU architectures like Fermi and Kepler have multiple warps schedulers per SM and are capable of issuing multiple warps' instructions per cycle. To ensure that each scheduler does not issue memory accesses from different warps when MP mode is in use, the WRC shares the owner warp's information with all schedulers present in an SM. Now, the scheduler that is handling an owner warp's instructions will have priority to issue its memory instructions to the LSU during periods of memory subsystem saturation, while any scheduler is free to issue any warp's computation instructions to their respective functional units.

Memory subsystem saturation detection: The memory saturation flag informs Mascar's scheduler of memory back pressure. This flag is controlled by logic in the SM's L1 cache.

The L1 cache has a fixed number of MSHRs as well as entries in the miss queue to send outstanding request across the interconnect. If either structure is totally occupied, no new request can be accepted by the cache that needs to send an outstanding request. Therefore, whenever these structures are almost full, the L1 cache signals to the LSU that the memory subsystem is saturating. The LSU forwards this flag to Mascar's scheduler so that it toggles to MP mode. The cache does not wait for these structures to completely fill as once this occurs, the owner warp will not be able to issue any requests. Instrumenting the L1 cache with this saturation detection is ideal as the L1 is located within the SM. Doing the same at the L2 requires information to travel between the L2 partitions and the SM, likely through the interconnect, which will incur more design complexity and delay Mascar's scheduler from shifting to MP mode in a timely manner.

Our benchmarks show that global memory accesses are the dominant cause of back pressure. However, for certain workloads, texture or constant memory accesses are major contributors to saturation. Mascar also observes saturation from these caches. In all kernels we observed, only one of the three memory spaces causes saturation at a given time.

4.3.2 Scheduling in Equal Priority (EP) Mode

In contrast to MP mode, in EP mode, the workload is balanced in such a way that that the memory subsystem is not saturated. Therefore, EP mode sends out as many memory requests as possible to utilize the under-utilized memory subsystem by prioritizing warps from memory queue over warps from compute queue. Within each queue, the warps are ordered in greedy-then-oldest policy. Memory warps are prioritized over compute warps because in EP mode the memory pipeline is not saturated and there should be enough computation to hide the latency of all the pipelined memory requests.

4.3.3 Cache Access Re-execution

When the memory subsystem becomes saturated with requests, the L1 data cache stops accepting new requests from the SM's LSU. At this point, the LSU is stalled and cannot process any new requests. When a memory request returns from memory and an MSHR is freed, the LSU can issue a new request to the L1. During this time, another warp whose data may be available in this cache cannot progress with its computation as the LSU is stalled. If this warp was able to access the L1 and retrieve its data, it could have helped hide the latency of other memory accesses with its own computation.

An example of this situation is illustrated in Figure 4.6(a) for the theoretical device depicted in Figure 4.2 with round-robin scheduling and a load instruction latency of five cycles if the data is not present in the cache. While W_0 is stalled in the LSU as no MSHRs are available in the L1, W_1 and W_2 are ready to access memory but cannot be issued. During this stall, W_1's data is actually available in the cache, but at $t = 5$ this data gets

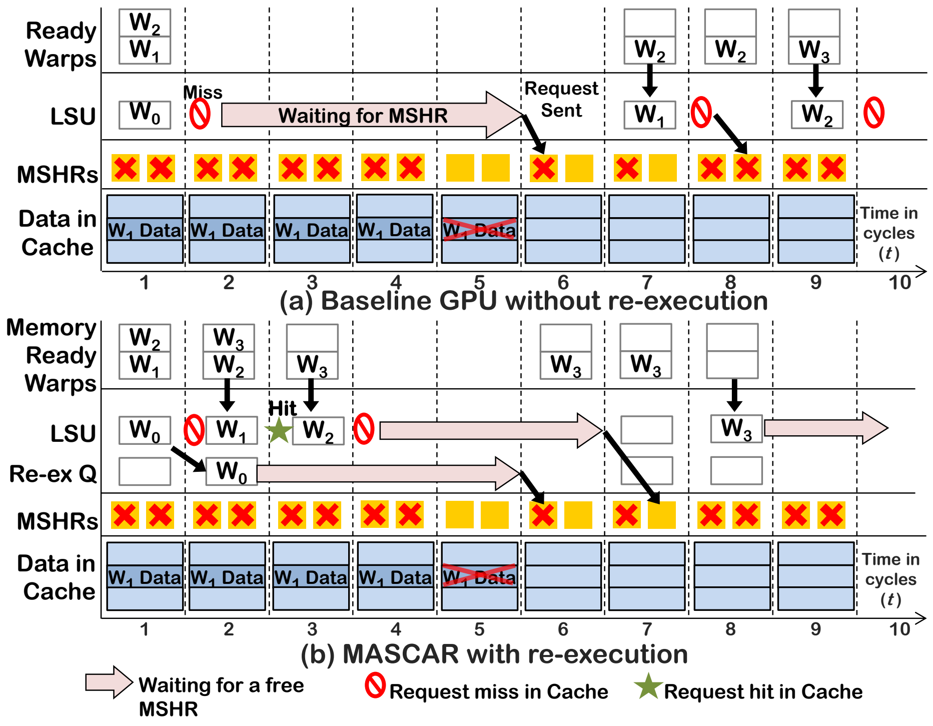


Figure 4.6: (a) The baseline system is ready to issue W_0's request but is stalled in the LSU as the L1's MSHRs are full, preventing W_1's request from hitting in the L1. (b) Mascar's cache access re-execution allows W_1's request to hit in the L1 while W_0 waits for an MSHR.

evicted when a previous request completes. After W_0's request gets sent to global memory, W_1 misses in the L1 and must reload its data from the L2/global memory. If there was some mechanism in place to enable a hit under miss while the MSHRs were occupied, W_1 could have gone ahead with its memory request to the cache, accessed the required data and started computation, all without needing to reload its data.

When the LSU stalls, warps with memory requests that might hit in the L1 cache are served their data much later. If this delay is too long, there is a chance that what may have been a hit in the L1 will become a miss as another request might return from the L2/DRAM and evict this request's data. This effect is exacerbated due to the limited size of the data cache, for which 1536 threads share up to 48KB of L1 data cache in modern architectures. Also, a system with more MSHRs might provides warps with more opportunities to access the cache, however, having more MSHRs can also exacerbate cache thrashing. Without increasing the size of an already small L1 cache, adding support for more outstanding requests may force some of these requests to evict data that would have soon been reused.

To overcome these issues and take advantage of hits under misses, we propose to add a cache access re-execution queue alongside the LSU as shown in Figure 4.7. The queue enables hit under miss without sending new requests to the L2/DRAM for non-owner warps. Whenever a request stalls in the LSU, the generated address and associated meta-data is removed from the head of the LSU's pipeline and is pushed onto the re-execution queue, freeing the LSU to process another request. If the newly processed request misses in the L1 cache, it is also added to this queue. Otherwise, if the next request hits in the cache, that warp can commit its memory access instruction and continue execution.

Requests queued for re-execution are processed if one of two conditions are met. First, if the LSU is not stalled and has no new requests to process, it can pop a request from the re-execution queue and send it to the L1. Second, if the re-execution queue is full, the LSU is forced to stall as it cannot push more blocking requests onto this queue. If this occurs, the LSU only issues memory requests from its re-execution queue. New memory instructions can only be issued to the LSU once entries in the re-execution queue are freed and the LSU is relieved of stalls. Address calculation need not be repeated for queued accesses, as this was already done when the request was first processed by the LSU.

Mascar only allows one memory instruction per warp to be pushed to the re-execution queue at a time. This is to ensure that if a store is followed by a load instruction to the same address, they are serviced in sequential order and thereby maintaining the weak memory consistency semantics of GPUs. As an instruction from a warp may generate several How-

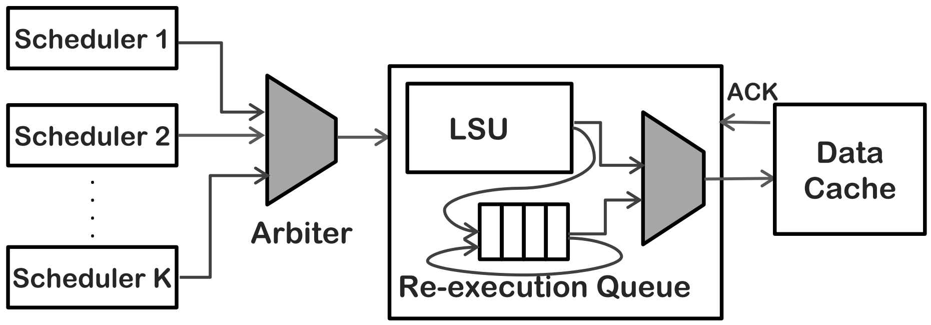


Figure 4.7: Coupling of the re-execution queue with an LSU. When a request from the LSU is stalled it is pushed to the re-execution queue. The cache sends an ACK if it accepts a request, or a NACK otherwise.

ever, our results show that a 32 entry re-execution queue satisfies the memory demands of our benchmarks' kernels. This design incurs less overhead than adding 32 MSHRs per SM as the complexity of a larger associative MSHR table is avoided.

The impact of coupling a re-execution queue with the LSU is illustrated in Figure 4.6(b). To demonstrate a simple example, this queue only has one entry. At $t = 0$, W_0 gets issued to the LSU, and because the MSHRs are full it is moved to the re-execution queue at $t = 2$. Now, W_1 is issued to the LSU, before a prior request evicts its data from the cache as had occurred in Figure 4.6(a). By moving W_0's request to the re-execution queue, W_1 can now go ahead and access the L1, where it experiences a hit under W_0's miss. Having obtained its data, W_1's memory instruction can commit, allowing W_2 to issue to the LSU. As the re-execution queue is full and W_2 misses in the L1 at $t = 4$, the LSU is forced to stall. However, as W_1 finished its load, the SM is able to perform useful computation on that data while W_0 and W_2 await an MSHR. Furthermore, as W_1's memory instruction was serviced and hit earlier, better utilization of the MSHRs is possible, allowing W_3 to be issued to the LSU earlier than in the baseline system. By exploiting hit under miss opportunities when using a re-execution queue, warps are able to bypass other warps that normally would block access to the L1, permitting more reuse of data in the cache.

4.3.4 MP Mode and Re-execution

The impact of re-execution on a memory subsystem experiencing back pressure is important. We have to ensure that re-execution does not interfere with MP mode scheduling as re-execution can send requests to the L2/global memory for any warp, not just the owner. Mascar resolves this at the L1 by preventing misses from non-owner warps from accessing the next level cache. As described in Section 4.3.1, Mascar provides the L1 with knowledge of which warp holds ownership when MP mode is active. If a non-owner warp's request is sent from the queue to the L1 and misses, the L1 returns a negative acknowledgement and this request is moved from the queue's head to its tail. Otherwise, if this request belonged to an owner warp and enough resources were available to send the request across the interconnect to the L2/DRAM, it would send this request to get its data. If an owner warp's request missed in the L1 and could not be sent to the L2/DRAM, the request would also be moved to the re-execution queue's tail. This recycling of requests, as shown in Figure 4.7 ensures that the owner warp can make forward progress when its requests are in the middle of the queue. If a warp relinquishes ownership and the scheduler's memory-ready warps queue is empty, the warp of the request at the head of the re-execution queue is given ownership and can now send requests to the L2/DRAM.

Data Reuse Improvement: Mascar improves L1 data cache hit rates in two ways. First, the owner warp alone can bring new data to the L1 and thereby the data brought by different warps is reduced significantly. Rogers et al. [70] have shown that for cache sensitive kernels, intra-warp locality is more important than inter-warp locality. Therefore,

Table 4.1: Simulation Parameters

Architecture	Fermi (GTX 480, 15 SMs, 32 PEs/SM)
Schedulers	Loose round-robin (LRR), Greedy-then-oldest (GTO) OWL [39], CCWS [70], Mascar
L1 cache/SM	32kB, 64 Sets, 4 way, 128 B/Line, 64 MSHRs [60]
L2 cache	768kB, 8 way, 6 partitions 64 MSHRs/partition, 200 core cycles latency
DRAM	32 requests/partition, 440 core cycles latency

Table 4.2: New Hardware Units per SM

Unit	Entry Size	# Entries	Total
Warp Readiness Checker	6 bits	n/a	1 byte
Warp Status Table	2 bits	48	12 bytes
Re-execution Queue	301 bits	32	1216 bytes

it is important for warps to consume the data they request before it gets evicted from the cache. By prioritizing one warp's requests over others, Mascar allows one warp to bring its data to the L1 and perform computation upon it before it is evicted by another warp's data. Second, re-execution also enables reuse for warps whose data has been brought to the cache by enabling hit-under-miss, leading to an overall improvement in hit rate.

4.4 Experimental Evaluation

4.4.1 Methodology

We use GPGPU-Sim [8] v3.2.2 to model the baseline NVIDIA Fermi architecture (GTX 480) and the Mascar extensions. We use the default simulator parameters for the GTX 480 architecture and the relevant parameters are shown in Table 4.1. All of our benchmarks, shown in Table 4.3, come from the Rodinia [14, 15] and Parboil [82] benchmark suites. The last column in Table 4.3 shows the number of instructions executed by the SM per miss in the L1 cache. Benchmarks that exhibit a ratio of instructions executed per L1 miss of greater than 30 are considered compute intensive and are marked C in the type column,

Table 4.3: Benchmark Descriptions

Application	Type	Comp/Mem	Application	Type	Comp/Mem
backprop (BP-1)	C	107	mrig-1	M	13.4
backprop (BP-2)	M	19	mrig-2	M	22.4
bfs	M	2.4	mrig-3	C	147
cutcp	C	949	mri-q	C	3479
histo-1	M	15.3	mummer	M	4.9
histo-2	C	35.9	particle	M	3.5
histo-3	M	14.3	pathfinder (PF)	C	55.83
histogram	C	91.2	sgemm	C	75
kmeans-1	M	0.27	spmv	M	4.2
kmeans-2	C	88	sad-1	C	2563
lavaMD	C	512	sad-2	M	10.2
lbm	M	10.4	srad-1	M	25
leuko-1	M	28	srad-2	M	22
leuko-2	C	173	stencil	C	38
leuko-3	C	5289	tpacf	C	10816

and others are marked M for memory intensive. GPUWattch [51] is used to estimate the power consumed by these applications for both the baseline and Mascar.

We compare Mascar scheduling with Loose round-robin (LRR, our baseline), Greedy-then-oldest (GTO), OWL [39] and Cache Conscious Wavefront Scheduling (CCWS) [70]. CCWS is modeled using [69] with simulator parameters modified to match our baseline GTX 480 architecture.

4.4.2 Hardware overhead

Three hardware structures are added to the baseline SM to implement Mascar's modifications. Table 4.2 shows the per SM overheads of these structures. To support scheduling requirements for MP mode, the Warp Status Table (WST) stores two status bits for each warp. As the Fermi architecture supports a maximum of 48 warps per SM, the WST requires 12 bytes of storage. The Warp Readiness Checker (WRC) stores the current owner warp's ID in a six bit field, and uses simple, single-bit boolean logic to determine the stall

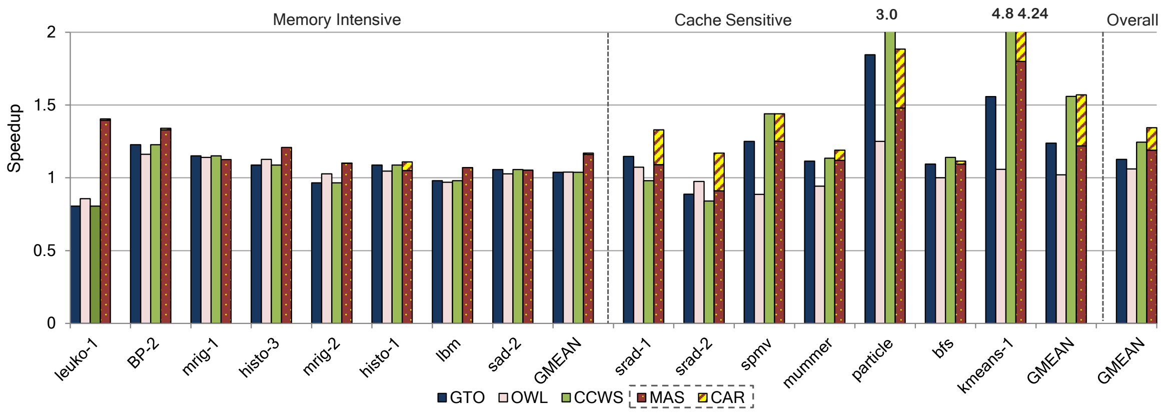


Figure 4.8: Comparison of performance for different policies on memory intensive kernels, relative to round-robin scheduling. The two components for the last bar in each kernel represents the contribution of scheduling and cache re-execution, respectively.

bit.

To support Mascar's cache access re-execution, each re-execution queue entry stores 301 bits of information. This includes the request's base address (64 bits), each thread's byte offset into this segment (224 bits — 7 bit 128B segment offset $\times$ 32 threads), and bits to identify the warp (6 bits) this request belongs to and its destination register (7 bits). A sensitivity study in Section 4.4.3 found that 32 entries were sufficient to expose ample hit under miss opportunities, making the queue 1216 bytes in size. Comparing the queue to each SM's 64KB L1 data cache/shared memory using CACTI 5.3 [85], we find that the queue's size is just 2.2% of that of the cache and that a queue access uses 3% of the energy of an L1 access which is a small component of the overall energy consumption.

4.4.3 Results

Performance Improvement: Figure 4.8 shows the speedup achieved for memory intensive kernels when using the warp scheduling schemes mentioned in Section 3.5 with respect to a round-robin scheduler. The chart is splits memory intensive kernels into those that are cache sensitive and those that are not. GTO focuses on issuing instructions from the oldest warp, permitting this one warp to make more requests and exploit more intra-warp data reuse. This greedy prioritization allows GTO alone to achieve a geometric mean speedup of 4% for memory intensive and 24% for cache sensitive kernels. However, GTO swaps warps whenever the executing warp stalls for the results of long latency operations including floating point operations, barrier synchronization and lack of instruction in instruction buffer. This allows memory accesses to be issued by more than one warp in parallel during memory saturation, resulting in the phenomenon explained in Section 4.2.1. The overall speedup achieved by GTO over the baseline is 13%.

OWL [39] tries to reduce cache contention by prioritizing sub-groups of warps to access the cache in an attempt to give high-priority sub-groups a greater chance to reuse their data. OWL is effective for several workloads that are sensitive to this prioritization, such as BP-2, mrig-1, histo-3 and particle. Overall, however, OWL is not as effective as reported for the older GTX 280 architecture due to improvements in our Fermi baseline. Prior work has shown that preserving inter-warp data locality is more beneficial when improving the hit rate of the L1 data cache [70]. The scoreboarding used in modern architectures allows a warp to reach an instruction reusing cached data much faster, enabling higher reuse of data in the cache. Overall, OWL scheduling shows 4% performance improvement over the baseline. We do not implement OWL's memory-side prefetching as it is orthogonal to our work and is applicable for any scheduling scheme.

For cache insensitive workloads, CCWS is equivalent to GTO, achieving a speedup of 4%. CCWS shows a significant speedup of 55% over the baseline for cache sensitive kernels as it is designed specifically to improve performance of such kernels with reuse, leading to an overall speedup of 24%.

The speedup achieved by Mascar is due to two components: memory aware scheduling and cache access re-execution (stacked on top of each other in Figure 4.8 for the last bar). For memory intensive kernels, Mascar performs better than or almost equal to all other scheduling schemes except for mrig-1 and mrig-2. These kernels have brief phases of high memory intensity, and the memory back pressure in these phases is relieved before these benchmarks can benefit from Mascar. While CCWS provides 4% speedup for memory intensive kernels, Mascar provides a significantly higher speedup of 17%. This is because CCWS only focuses on cache sensitive kernels, whereas Mascar improves the overlap of compute and memory for all kinds of memory intensive kernels.

The main reason for the high speedup of Mascar over GTO is that GTO provides priority to the oldest warp, whereas Mascar provides exclusivity to one warp. This inherently leads to higher utilization in cases where there is data locality. For example, leuko-1's innermost iteration reads two float data from the texture cache. The first data is constant across all warps in a thread block for that iteration, and the second is brought in by one warp and can be used by one other warp in the same block. However, with GTO, all other warps will bring their own data for the second load. This results in the eviction of the first warp's data before the reuse by the other warp can occur. With Mascar, all data from one warp is brought in and only then can another warp bring in its data. In the meantime, the data brought in by the first warp can be reused by the second warp. Kmeans-1 is a very cache sensitive kernel. Each warp brings in a significant number of cache lines per load due to uncoalesced accesses. Several of these loads are also reused by another warp. Therefore, thread throttling schemes such as CCWS do very well on it. Similarly, Mascar slows down

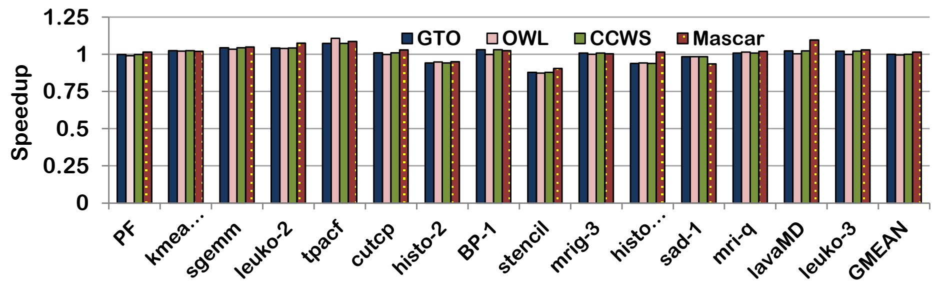


Figure 4.9: Performance of compute intensive kernels for four different scheduling policies. Overall, the choice of scheduler has little impact on these workloads.

the impact of the heavy multi-threading by allowing only one warp to issue requests at a time and its results are close to CCWS for kmeans-1. With GTO, more cache thrashing takes place due to inter-warp cache thrashing between 48 warps.

Cache access re-execution alone provides performance benefits of 35% and its combination with Mascar scheduling provides an overall speedup of 56% for cache sensitive kernels. CCWS is designed specifically for cache sensitive kernels and provides a speedup of 55%. While Mascar's speedup for cache sensitive workloads is comparable to CCWS, the area overhead of Mascar is only one-eighth of CCWS' additional hardware. The design complexity of CCWS involves a victim tag array and a point-based system for warps, whereas the design of Mascar involves a queue and tables such as WST and WRC which do not use an associative lookup. Overall, Mascar achieves a geometric mean speedup of 34% as compared to 13% speedup for GTO and 24% for CCWS at one-eighth the hardware overhead.

The choice of scheduling policy has little impact on the performance of compute intensive workloads, as seen in Figure 4.9. OWL, GTO and CCWS are within 1% of the baseline's performance, while Mascar achieves a 1.5% geometric mean speedup. Occasionally, short phases in compute intensive workloads suffer from memory saturation, and

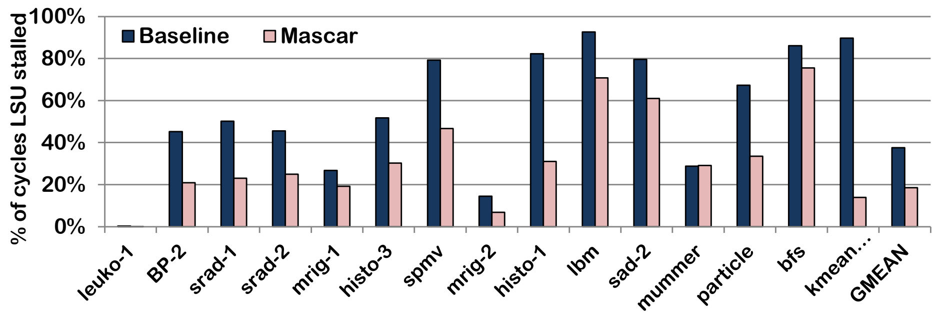


Figure 4.10: Reduction of LSU stalls when using Mascar.

Mascar can accelerate these sections. Leuko-2, lavaMD, and histogram are examples of such workloads. Leuko-2's initial phase exhibits significant pressure on the memory system due to texture accesses, which Mascar's MP mode alleviates. LavaMD and histogram behave similarly but with saturation due to global memory accesses. Mascar does cause performance degradation compared to the baseline scheduling for histo-2, stencil and sad-1, but these slowdowns are not significant.

Microarchitectural impacts of Mascar: Mascar's scheduling and re-execution have a significant impact on the fraction of cycles for which the LSU stalls, previously discussed in Section 4.2. Figure 4.10 shows that Mascar is capable of reducing these stalls on average by almost half, from 40% down to 20%. By alleviating these stalls, Mascar more efficiently brings data to the SM and overlaps accesses with computation. Spmv, particle and kmeans-1 experience great reductions in stalls, which correlates with their significant speedups in Figure 4.8. Because compute intensive workloads stall the LSU far less frequently than memory intensive applications, we do not include these figures.

Depending on the severity of memory intensity, Mascar's scheduler will be in either EP or MP modes for different durations of time as shown in Figure 4.11. There is a direct correlation between the number of cycles a workload is in MP mode to the number of

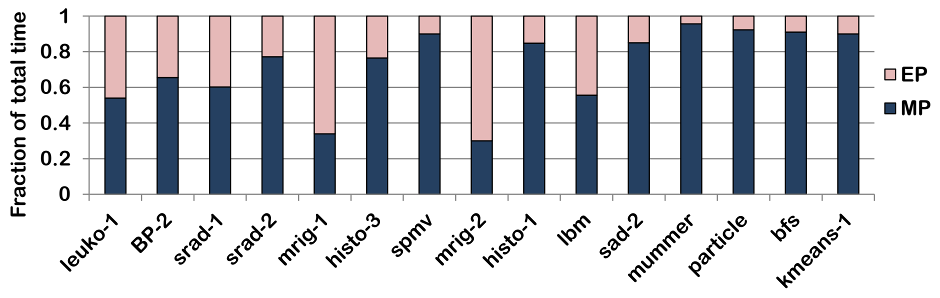


Figure 4.11: Distribution of execution time in EP and MP modes across workloads.

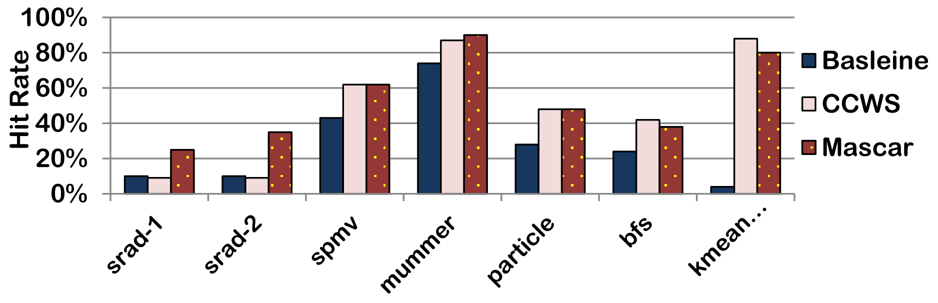


Figure 4.12: Improvement in L1 data cache hit rates.

cycles the LSU is stalled for the baseline scheduler used in Figure 4.1. Leuko-1, srad-1, mrig-1, mrig-2 and lbm are workloads that spend some time in both modes and exhibit phased behavior with intermixed periods of compute and memory intensity. As previously described, mrig-1 and mrig-2 spend most of their time in EP mode and their MP mode phases are so short that the benefits of Mascar are muted for these benchmarks. Severely memory intensive workloads, including kmeans-1, bfs, particle, and mummer, operate in MP mode for 85%–90% of their execution.

The impact of the re-execution queue is illustrated by the improvements in L1 hit rates shown in Figure 4.12. We only show results for the seven cache sensitive workloads and compare them with CCWS. While CCWS achieved better hit rates than Mascar for three kernels as it reduces the number of warps that can access the data cache to preserve locality, Mascar's hit rate improvements are better than CCWS for three other kernels. Because the

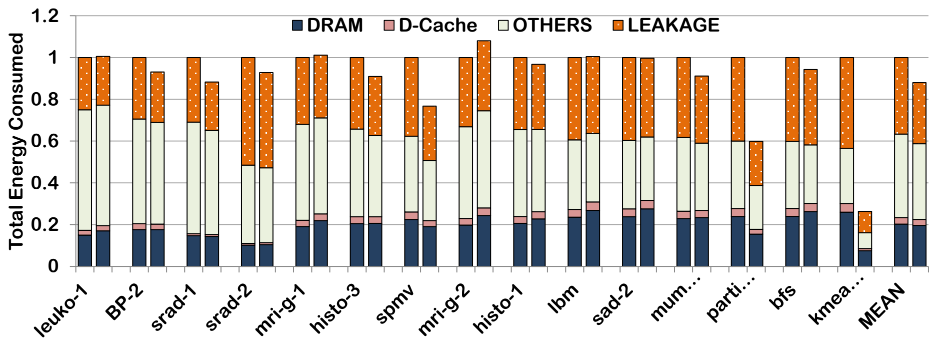


Figure 4.13: Energy consumption breakdown for baseline (left bar) and Mascar (right bar, relative to baseline).

number of requests going to DRAM are significantly reduced, the exposed memory latency is reduced such that it better overlaps with computation. The resulting exposed latency can be effectively hidden by both CCWS and Mascar, resulting in less than 1% difference in performance between the two. However, CCWS's victim tag array incurs more hardware overhead and design complexity than Mascar, which has a that can induce a higher energy overhead while not improving the performance of cache insensitive kernels.

We performed a sensitivity study of the impact of the size of the re-execution queue on performance of the cache sensitive kernels. For all kernels except kmeans-1, a 16 entry re-execution queue is sufficient to expose greater reuse. Kmeans-1 has a significant number of uncoalesced accesses, and requires more entries as each uncoalesced access breaks down into multiple requests. Overall, the performance of the kernels saturates with 32 entries, and hence was chosen for our design.

Improvement in Energy Efficiency: Mascar's speedups from scheduling and re-execution leads to energy savings. This can be seen in Figure 4.13, where each workload's left and right bar represent the energy consumed by the baseline versus Mascar architectures. On average, Mascar reduces energy consumption by 12% compared to the baseline system.

To further analyze Mascar's energy efficiency, we break these energy figures down into four components: *DRAM*, *L1 data cache*, *leakage* and *others*, which includes the interconnect, SM pipelines, and L2 cache. All components are normalized with respect to the total energy consumed on the baseline.

DRAM energy consumption is mostly unchanged except for kmeans-1, particle, and spmv show noticeable reductions. By reducing the warps that can issue memory requests during MP mode scheduling, less thrashing and higher L1 hit rates reduce DRAM traffic. A few kernels (srad-2, mrig-2, and lbm) experience increased DRAM energy consumption. Jog et al. [39] discuss how consecutive thread blocks access the same DRAM row. In the case of these kernels, Mascar sometimes reduces row locality by allowing all of one warp's requests to go to DRAM, forcing other warps to reopen previously used rows. The data cache's energy consumption is slightly reduced as Mascar exploits hit-under-miss opportunities, reducing failed access attempts. This indicates that the energy impact of re-executing cache accesses is negligible. Other components exhibit a 3.5% decrease in energy, primarily due to reduced interconnect traffic in MP mode. The greatest contribution to energy savings is due to the savings in leakage energy, which improved by 7% on average. As workloads made quicker progress during phases of memory saturation, they finished earlier, thus reducing the leakage energy consumed.

4.5 Related Work

Warp scheduling: Scheduling on GPUs has received wide attention from many works. Jog et al. [39] propose a scheduler which aims to reduce cache contention and exploit bank level parallelism to improve performance. With the addition of scoreboarding, newer GPUs have improved abilities to overlap computation with memory accesses and alleviate cache contention. Scoreboarding allows warps to reach loads faster, hence taking advantage of available data locality. Narasiman et al. [58] proposed a two-level warp scheduler (TLS) which divides warps into fetch groups to overlap computation with memory access by staggering warp execution so that some warps perform computation while others execute memory operations, modern GPU architectures, however, allow enough warps per SM to naturally generate an equivalent or better overlapping. Our experiments show that TLS scheduling shows 1.5% speedup over the baseline.

Improving cache locality in GPUs: Rogers et al. [70] propose a cache conscious scheduling mechanism which detects cache locality with new hardware and moderates the warps that access the cache to reduce thrashing. DAWS [71] improves upon this mechanism by using a profile-based and online detection mechanism. Jia et al. [37] use a memory request prioritization buffer (MRPB) to reorder cache requests so that the access order preserves more data locality. Furthermore, in MRPB, certain requests bypass the cache and are transferred and received by the core directly. Cache and Concurrency Allocation (CCA) [88] also limits the number of warps that can allocate cache lines to improve data locality while other warps bypass the cache, ensuring sufficient bandwidth utilization.

Mascar targets general memory intensive workloads that may not have data locality, which is not addressed by CCWS, DAWS, MRPB or CCA. While these schemes try to avoid stalling the LSU during back pressure, Mascar enables the LSU to continue issuing requests to the L1 even when it is stalled through its re-execution queue. It also improves data reuse by only allowing one warp to bring its data, preventing other warps from thrashing the cache. Mascar incurs less hardware overhead than the above techniques. We compare Mascar with OWL and CCWS in Section 4.4.3.

Improving resource utilization on GPUs: There have been several works on improving GPU resource utilization. Gebhart et al. [27] propose a unified scratchpad, register file, and data cache to better distribute the on-chip resources depending on the application. However, they do not provide resources to increase the number of outstanding memory requests. Jog et al. [38] propose a scheduling scheme that enables better data prefetching and bank level parallelism. Rhu et al. [68] create a locality aware memory hierarchy for GPUs. Lakshminarayana and Kim et al. [46] evaluate scheduling techniques for DRAM optimization and propose a scheduler which is fair to all warps when no hardware-managed cache is present. Our work focuses on systems with a saturated memory system in the caches and DRAM. Ausavarungnirun et al. [7] propose a memory controller design that batches requests to the same row to improve row locality, hence improving DRAM performance.

Adriaens et al. [5] propose spatial multi-tasking on GPUs to share resources for domain specific applications. Gebhart et al. [26] propose a two-level scheduling technique to improve register file energy efficiency. Kayiran et al. [40] reduce memory subsystem saturation by throttling the number of CTAs that are active on an SM. Fung et al. [24] and Meng et al. [57] optimize the organization of warps when threads diverge at a branch. Bauer et al. [9] propose a software-managed approach to overlap computation and memory accesses. In this scheme, some warps bring data into shared memory for consumption by computation warps. Hormati et al. [32] use a high-level abstraction to launch helper threads that bring data from global to shared memory and better overlap computation.

Re-execution: The concept of re-executing instructions has been studied for CPUs by

CFP and iCFP [81, 30] which insert instructions into a slice buffer whenever the CPU pipeline is blocked by a load instruction and re-execute from a checkpoint when a miss returns. Sarangi et al. [73] minimize this checkpointing overhead by speculatively executing forward slices. While these schemes benefit single threaded workloads, having slice buffers and checkpointing micro-architectural state for every warp can be prohibitively expensive. Hyeran et. al propose WarpedDMR [33] to re-execute instructions on the GPU during under-utilization of the hardware, providing redundancy for error protection. They focus on re-execution of instructions that run on the SM and do not re-execute accesses to the data cache, which is done by Mascar to improve performance.

4.6 Summary

Due to the mismatch between the applications' requirements and resources provided by GPUs, the memory subsystem is saturated with outstanding requests which impairs performance. We show that with a novel scheduling scheme that prioritizes the memory requests of one warp rather than issuing several requests of all the warps, better overlap of memory and computation can be achieved for memory intensive workloads. With a re-execution queue, which enables access to data in the cache while the cache is blocked from accessing main memory, there is significant improvement in cache hit rate. Mascar achieves 34% performance improvement over the baseline scheduler. Due to the low overhead of Mascar, this translates to an energy savings of 12%. For compute intensive workloads, Mascar is 1.5% faster than the baseline scheduler.

CHAPTER V

APOGEE: Adaptive Prefetching on GPU for Energy Efficiency

In this work, I investigate the use of prefetching as a means to increase the efficiency of GPUs without degradation in performance for compute intensive kernels which require maximum concurrency for maximum performance. Classically, CPU prefetching results in higher performance but worse energy efficiency due to unnecessary data being brought on chip. Our approach, called APOGEE, uses an adaptive mechanism to dynamically detect and adapt to the memory access patterns found in both graphics and scientific applications that are run on modern GPUs to achieve prefetching efficiencies of over 90%. Rather than examining threads in isolation, APOGEE uses adjacent threads to more efficiently identify address patterns and dynamically adapt the timeliness of prefetching. The net effect of APOGEE is that fewer thread contexts are necessary to hide memory latency and thus sustain performance. This reduction in thread contexts and related hardware translates to simplification of hardware and leads to a reduction in power.

5.1 Introduction

The demand for rendering increasingly real world scenes is leading to a surge in computational capability of GPUs. The combination of programmability and high computational power have made GPUs the processor of choice even for throughput oriented scientific applications. High throughput is achieved in GPUs by having hundreds of processing units for floating point computations. To keep all these processing units busy, GPUs use high degrees of multi-threading to hide latency of global memory accesses and long latency floating point instructions. For example, one of NVIDIA's GPUs, the GTX 580, has 512 processing units and can use over 20,000 threads to maintain high utilization of the compute resources via fine-grained multi-threading [61]. The basic approach that GPUs use is to divide the given work into several chunks of small, independent tasks and use fine-grained multi-threading to hide any stall in the pipeline. They can hide several hundreds of cycles of latency if they are provided with enough work.

The support for such levels of multi-threading and fast context switching requires deployment of considerable hardware resources such as large register files. For example, the GTX-580 has 2 MB of on-chip register file. In addition to the register files, more resources are required to orchestrate the fine-grained multi-threading and maintain thread state including divergence stacks and warp schedulers. An unwanted cost of this design style is high power consumption. While providing teraflops of compute horsepower, modern graphics cards can consume 100W or more of power. As modern systems are becoming power limited [41], such a trajectory of power consumption is a difficult challenge to meet on desktop devices.

Furthermore, the increasing trend of using scientific style compute with OpenCL and visually rich applications and games on mobile GPUs will continue to push the performance needs of GPUs in mobile domain as well. Straight-forward down scaling of desktop GPUs for mobile systems often results in insufficient performance as the power budget for mobile systems is a small fraction of desktop systems. Thus, improvements in the inherent energy efficiency are required so that performance can be scaled faster than energy consumption in both the domains.

To attack the energy efficiency problem of GPUs, we propose APOGEE: Adaptive Prefetching On GPU for Energy Efficiency [76], that leverages prefetching to overlap computation with off-chip memory accesses. Successful prefetching reduces the exposed memory latency and thereby reduces the degree of multi-threading hardware support necessary to sustain utilization of the datapath and thus maintain performance. As the degree of multi-threading support is reduced, power efficiency is increased as datapath elements can be correspondingly reduced in size (e.g. register file, divergence stacks, etc.).

Prefetching is not a novel concept, it has been around for years and is used in commercial CPUs. Conventional wisdom suggests that prefetching is more power hungry due to unnecessary off-chip accesses and cache pollution. We postulate that this wisdom is untrue for GPUs. On the contrary, prefetching improves the energy efficiency of GPUs for two reasons. First, the reductions in memory latency provided by prefetching can directly lead to reductions in hardware size and complexity that is not true for CPUs. Second, prefetching efficiency for the GPU can be increased well beyond those seen on the CPU because the address patterns for graphics and scientific applications that are commonly run on GPUs are regular.

In order to reduce the degree of necessary multi-threading in an energy efficient way, Apogee has to address three challenges. Firstly, it should be able to predict the address patterns present in GPU applications - both graphics and scientific. Prefetching for scientific applications for CPUs is a well studied area, where stride prefetching is known to be effective [16, 20, 23]. In the GPU domain, such a prefetching technique for scientific applications will not be efficient for thousands of threads as explained in Section 5.2.2. On the other hand, for graphics applications, strided patterns are not the only common access patterns [48]. Secondly, it should be able to prefetch in a timely fashion, such that the data arrives neither too early nor too late to the core. For energy-efficiency, timeliness of prefetching is an important factor, as it reduces the need for storing correct but untimely prefetched data. Thirdly, the prefetcher should not over burden the bandwidth to memory, as aggressive prefetching can cause actual read requests to be delayed.

Apogee exploits the fact that threads should not be considered in isolation for identifying data access patterns for prefetching. Instead, adjacent threads have similar data access patterns and this synergy can be used to quickly and accurately determine the access patterns for all the executing threads. Apogee exploits this characteristic by using address streams of a few threads to predict the addresses of all the threads leading to a low hardware overhead design as well as a reduced number of prefetch requests to accomplish the necessary prefetching. The key insight that Apogee uses to maintain the timeliness of prefetching is by dynamically controlling the distance of prefetching on a per warp basis. The same technique can also be used to control the aggressiveness of prefetching.

In order to solve the challenges mentioned above, this paper makes the following contributions:

- We propose a low overhead prefetcher which *adapts* to the address patterns found in both graphics and scientific applications. By utilizing an intra-warp collaboration approach, the prefetcher learns faster and by sharing this information amongst warps, the area/power overhead of the prefetcher is lower than traditional prefetchers.

- We show that Apogee can *dynamically adjusts the distance* of prefetching depending on the kernel for a SIMT architecture. This provides timely and controlled prefetching for variable memory latency.

- We show that the required degree of multi-threading can be significantly reduced and thus energy efficiency is increased without any loss in performance for both Graphics and GPGPU applications.

5.2 Techniques to hide large memory latency

This section discusses the shortcomings of the two major prevalent techniques used for hiding/reducing memory access latency: i) High-degree of multithreading as in SIMT execution model and ii) Traditional prefetching.

5.2.1 High-degree multithreading

GPU Performance and Power. This section illustrates the performance and power effect of the high degree of multi-threading, which is required for keeping the GPU processing units busy. Figure 5.1 illustrates the change in speedup and power consumed with increasing the number of warps from two to 32. The methodology for obtaining these results is explained in Section 5.4. The baseline case is a system with 1 warp implying

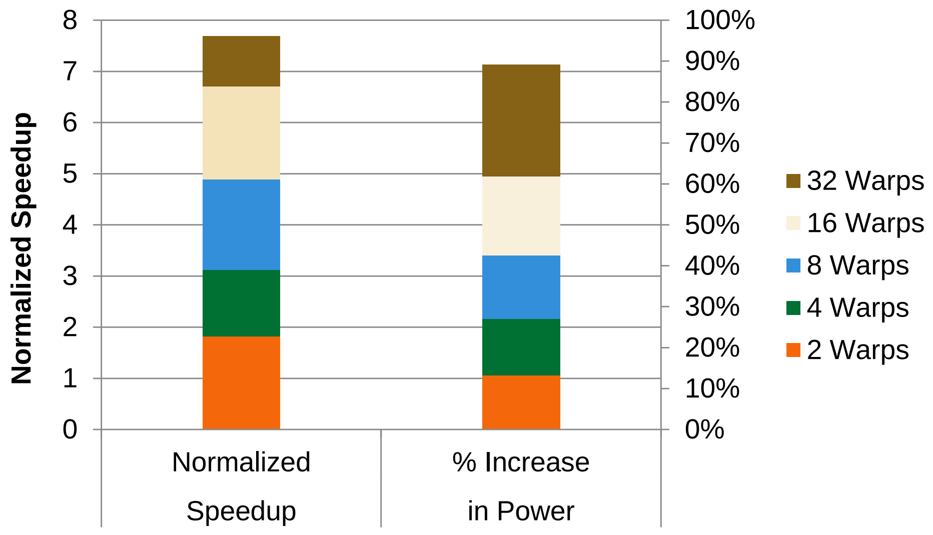


Figure 5.1: Increase in speedup and increase in power as number of warps are increased

no multi-threading support. As number of maximum concurrent warps is increased, the speedup increases because computation by concurrent warps hides the latency of access to memory reducing stall cycles. With 32 warps around 85% of the total cycles are reduced as compared to the baseline. When the system has two and four warps, the decrease in number of cycles is substantial as all the compute cycles hide some kind of stall. Adding further warps reduces a smaller fraction of cycles as many of the stalls are already hidden by the fewer warps present earlier. The right bar in Figure 5.1 shows the increase in power as the number of warps are increased. Even though 16 and 32 warps provide around one-third speedup, the corresponding increase in power is half of the total increase in power. This is because of the fact that large number of resources are added to support the high degree of multi-threading and all these resources might not be completely utilized as explained below.

GPU Underutilization. Figure 5.2 shows the variation in normalized access rate of the register file on the primary vertical axis and normalized per register accesses of the register

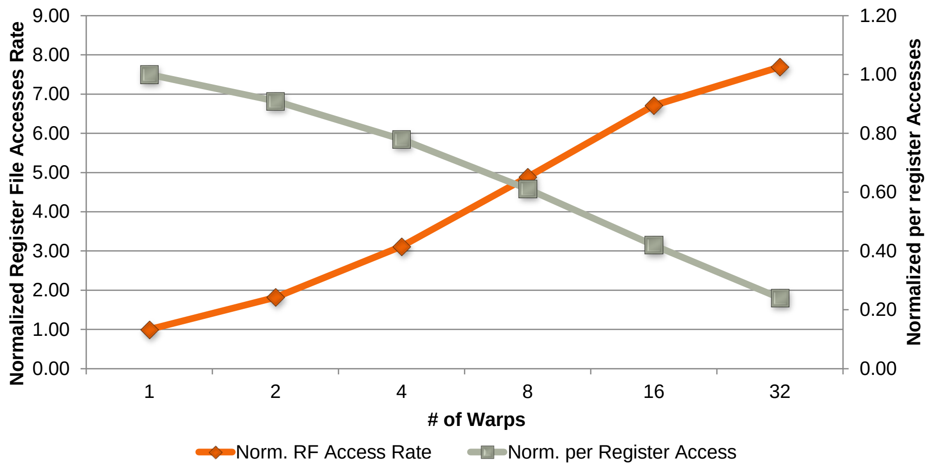


Figure 5.2: Increase in number of register file access rate and degradation in per register accesses. This shows that with SIMT more hardware is required, but the utilization of the hardware reduces significantly.

file in the secondary vertical axis of an SM as the number of maximum active warps per SM are increased. All values are normalized to the case when the SM has one warp. Due to the reduction in stalls as more warps are added, the access rate of the register file also increases as the number of accesses is same but the total time is reduced. However, as more warps are added to the SM, the size of the register file is also increased correspondingly. The size of the register file with 32 maximum warps is 32 times the baseline case of 1 warp. While the access rate of the register file increases to 7.69 times for 32 warps, the overall number of access per register access is down to 0.24 times of the 1 warp case. With the decreased number of accesses per register, several thousand registers are underutilized and leaking power.

Similary, whenever a warp is added to the system, register files, branch divergence book-keeping support, warp schedulers, scoreboarding, etc. need extra capacity. While the strategy of high multi-threading keeps the compute units like FPU busy, the additional hardware mentioned above has lower utilization and hence increased power consumption. This

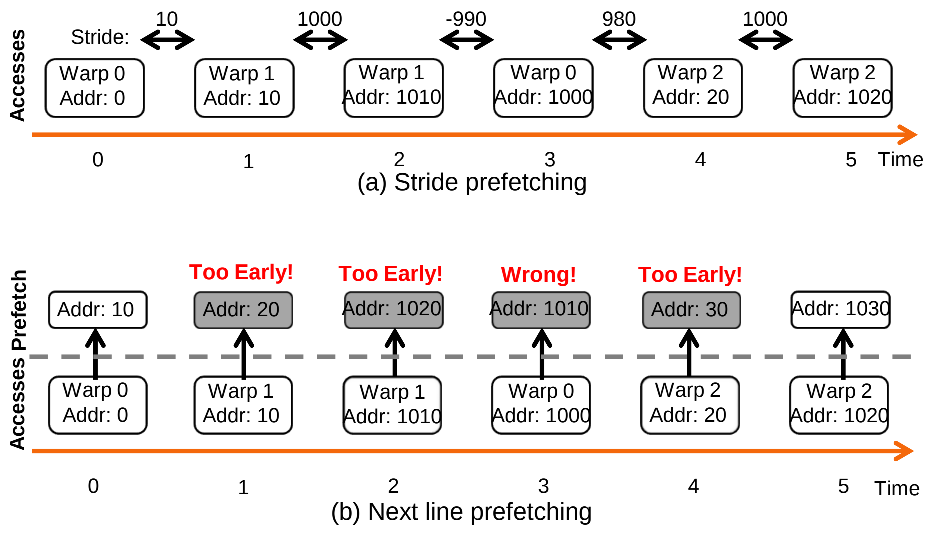


Figure 5.3: Examples to demonstrate the ineffectiveness of common CPU prefetching techniques on GPU (a) stride prefetching and (b) next-line prefetching. These techniques do not work well due to the unordered execution of warps.

demonstrates that the design of GPUs with heavy multi-threading has a significant amount of underutilized hardware which can lead to a significant increase in power consumption due to leakage.

5.2.2 Traditional CPU Prefetching

Prefetching has been studied extensively for multi-processors. However, although common prefetching techniques work well for CPUs with limited numbers of threads, for prefetching memory accesses of thousands of threads, those techniques cannot predict the memory access pattern correctly. The rest of this section shows why common CPU prefetching techniques such as stride prefetching and next-line prefetching do not work efficiently for GPUs.

Stride Prefetching. One of the most common CPU prefetching technique is stride prefetching which focuses on array-like structures, computes the stride between accesses, and uses that to predict the next accesses. However, this technique is not efficient on GPUs due to interleaved memory accesses from different warps. Figure 5.3(a) shows an example of memory addresses of different warps. The difference between addresses of two consecutive memory requests is shown in the figure. Although there is a constant stride between two accesses within a warp, a stride prefetcher only sees a random pattern and as a result, it cannot compute the stride correctly. Therefore, stride prefetching technique is not a good choice for GPU architecture due to interleaved memory accesses.

Next-line Prefetching. Another common CPU prefetching technique is next-line prefetching which fetches the next sequential cache line whenever it is not resident in the cache. Again, this technique is not efficient for GPU execution model. Figure 5.3(b) shows how next-line prefetcher finds the address that it wants to prefetch. In this example, cache line size is equal to 10B. There are two problems with this scheme. First, next-line prefetcher may fetch an address after the actual memory access. For example, after the fourth memory access , prefetcher brings the address 1010 to the cache. However, this address is already accessed by the third access so prefetching does not help and consumes bandwidth. Secondly, this prefetcher might prefetch an address far ahead of actual memory request. This might results in eviction of correctly prefetched data from the cache before it was used. For example, the second memory access prefetches the address 20 but this address will be accessed after several accesses. Therefore, the probability of eviction of useful data is high for this method.

Overall, the two prevalent CPU prefetching technique cannot be implemented as such on a GPU.

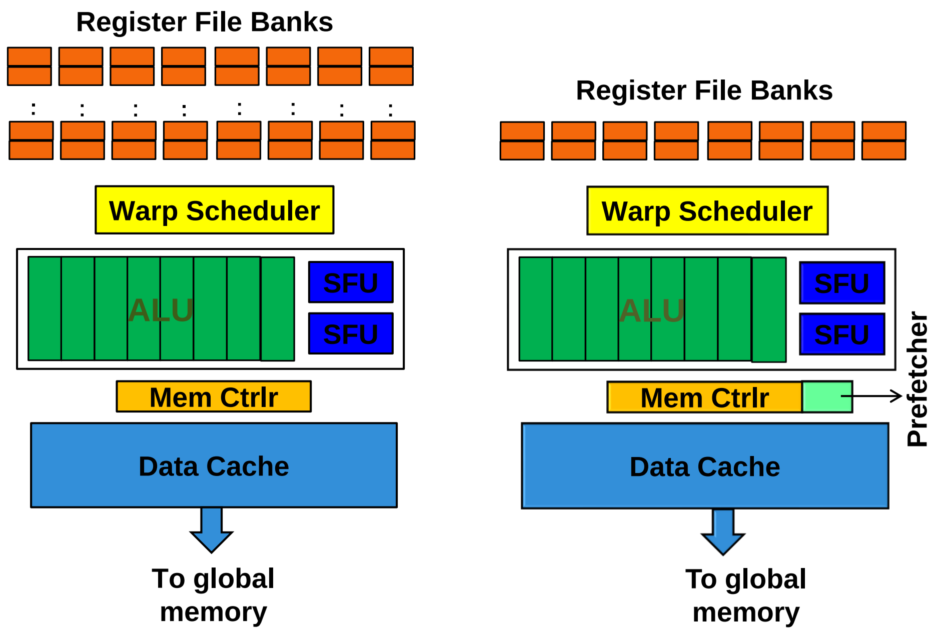


Figure 5.4: (a) Traditional SM (b) Apogee SM with prefetching in data cache and reduced register bank size.

5.3 Apogee

Section 5.2 described how high degree multithreading has a problem with resource utilization and traditional CPU prefetching techniques show poor performance on GPU. To hide memory access latency in a energy efficient way, Apogee adopts new prefetching techniques that are tailored specifically for GPU architecture in order to improve the prefetching efficiency. As a result, Apogee maintains the performance by using fewer hardware resources. By using these prefetching techniques, underutilized hardware required for multi-threading as discussed in Section 5.2.1 can be scaled back which results in improved energy efficiency.

Figure 5.4 compares the design of a traditional SM with an Apogee SM. Apogee interacts with the memory system and looks at the memory access pattern between the threads of a warp. If it can detect a consistent pattern in the addresses accessed by adjacent threads, it stores various characteristics of that load. These characteristics are the Program Counter (PC), the address accessed by one thread, its offset with the adjacent thread and how consistent the offset is across adjacent threads. Once Apogee is confident of the access pattern, it prefetches data into the data cache as shown in Figure 5.4(b). The requests are sent from the data cache to the global memory.

In GPUs, Apogee exploits two important features:

1. The SIMD nature of the pipeline forces the same load instruction to be executed by all threads in the warp. This helps in the characterization of the access pattern of the load, as every thread in the warp provides a sample point for the detection of the pattern.

2. Adjacent threads in a warp have adjacent thread indices, so the difference between the memory addresses accessed by these threads can provide a regular offset.

For Apogee to utilize prefetching for reduction of hardware it should be able to predict the access pattern found in Graphics and GPGPU applications. The major memory access pattern found in these applications are generally a linear function of the thread index of the thread accessing memory. This occurs because distribution of work in a kernel is done primarily using thread index. Since, adjacent threads differ in thread index by 1, the offset between the addresses accessed by adjacent threads is fixed. This constant difference is valid for all adjacent threads in a SIMT load instruction. We call such an access pattern as *Fixed Offset Address(FOA)*. The prefetcher for FOA is explained in Section 5.3.1.

Apart from predicting the correct address accessed by a warp in the next access, a major factor in prefetching performance is the timeliness of the prefetch requests. If the prefetch request is made too late, then the next access of that load would be a miss in the cache even though a prefetch request for the same data has been made. On the other hand if the prefetch request is made too early, the data will be in cache while it is not needed. Since, the data brought by prefetching is not guaranteed to be used by the core, it might evict useful data. Apogee maintains the state of prefetching and adjusts the distance of prefetching dynamically as explained in Section 5.3.1.1.

Since Apogee reduces hardware support for high degree multithreading, the latency hiding capacity of the system through SIMT execution is reduced. Therefore, acccess patterns that are not frequent, but miss in cache, can also have significant impact on performance, unless, the prefetcher is able to prefetch those access patterns. For graphics applications, moderate number of accesses are made to the same address by all threads. We call these memory accesses *Thread Invariant Access(TIA)*. As the addresses in TIA are same, predicting these addresses is trivial. In prefetching for TIA accesses, Apogee focuses on timeliness of prefetching rather than prediction accuracy. Apogee uses a novel prefetching method which finds an earlier load at which to prefetch the thread invariant address. Details of TIA prefetching is explained in Section 5.3.2.

By timely prefetching of data into the cache for both the access patterns, Apogee reduces the need for additional structure to hold the prefetched data. These structures are required in traditional and current GPGPU solutions [48], to prevent cache pollution. However, these hardware structures are not efficient for a design which targets low power. Also, a static software/hardware prefetching solution which uses a constant distance of prefetching for timeliness, will not be a good solution for GPUs. The problem of timeliness of prefetching in GPUs is exacerbated by the fact that the look-ahead distance is not only a

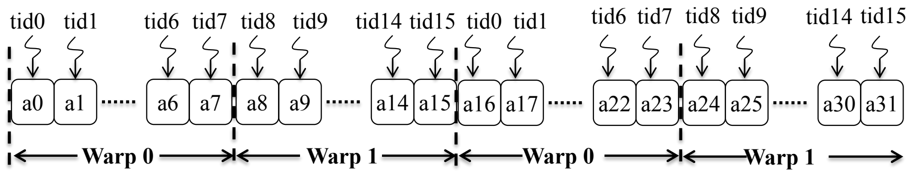


Figure 5.5: Example of processing an array of elements in SIMT model with 2 warps

function of the kernel size, but also of the number of warps and scheduling. Apogee dynamically adapts to individual memory accesses and adjusts how far ahead data should be prefetched based on that instruction only, so it can prefetch in a timely fashion.

5.3.1 FOA Prefetching

Predictability. FOA accesses are generally predictable as they are based on thread index. Apogee exploits this fact to detect such streaming accesses. Figure 5.5 shows an example of accesses to various regions of an array by two warps in the SIMT model. It also shows the accesses made by each thread in a warp. In this example there are eight threads per warp, with a total of 16 threads. Each thread has a unique thread index. Using this thread index, a thread can access some element of the array. Programs in SIMT execution are modelled such that adjacent threads access adjacent memory locations. So after processing the first 16 elements, tid 0, moves to processing the 17th element and other threads follow the same pattern. In this example, during its lifetime, a thread with tid n processses $n + i * (total\_threads)$ elements, where i is a factor which represents the current section of data being processed and varies from zero to $N/(total\_threads)$ where N is the total number of elements. Therefore, the addresses accessed by a thread are predictable for FOA accesses.

In this example, each thread accesses data elements as a linear function of the thread in-

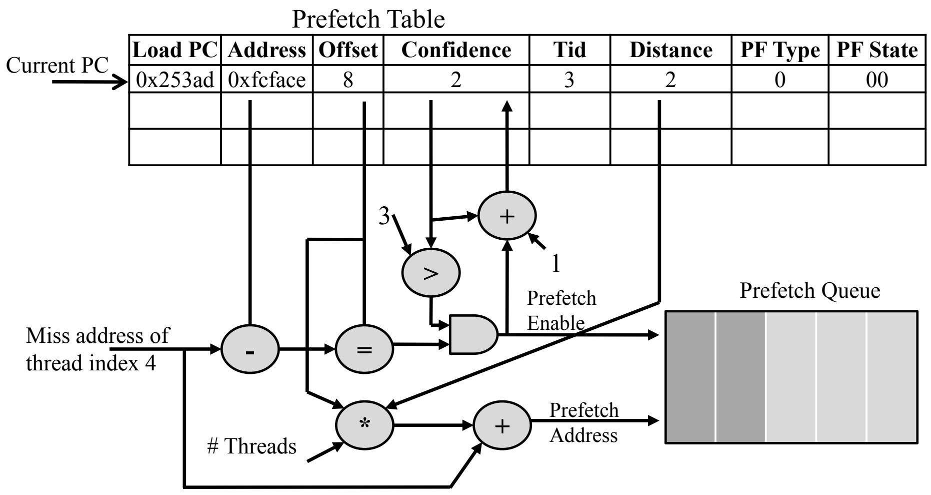


Figure 5.6: Fixed Offset Address prefetcher design

dex. However, the access can be a complex function of the thread index as well. Generally, the function is rarely complicated because if the addresses are not contiguous, uncoalesced memory access can lead to a heavy performance penalty. Therefore, for performance reasons, most programs access addresses that are usually simple functions of the thread index.

For the FOA pattern, the offset between the memory accesses of adjacent threads in a warp is confirmed by using all the threads in the warp. After that, it can predict the addresses accessed in the future iterations of a warp by using the number of active threads in the warp, the number of active warps, and the validated offset. Doing this for all the warps will result in the ability to prefetch all the data required for the next iteration of all of the threads.

Prefetcher Implementation. Apogee uses FOA prefetching where access patterns between threads in a warp for a load are used to decide the address that will be accessed by the next access by these threads. The structure of the FOA prefetcher is shown in Figure 5.6. It consists of a Prefetch Table and other logic to check the offset between addresses of different threads in a warp. When the warp makes a load request, the access requests of all the threads in the warp are visible to the prefetcher. The load address of the first active thread is stored in the address column and the thread index is also updated. Then, with the address and thread index of the second active thread, Apogee computes the difference in addresses and difference in thread indices between the two threads. The offset is then calculated to be the ratio of the two quantities. We also update the address column and thread index column with the address of the second thread and its thread index respectively.

For the next active thread, Apogee computes the offset and compare it with the previous offset. If they are equal, Apogee increases the confidence column of the entry. Apogee does this for the remaining threads in the warp. At every step a ratio is taken between the difference in address and difference in thread index to calculate the offset. This is done because a thread might be inactive due to branch divergence and no load request will be made from that thread. In such a case, the difference of the next active thread, with the address stored in the table will be two times the offset. So a ratio is taken between the quantities instead. If the confidence of the load is fewer than two less than the number of active threads in the warp, then we conclude that the offset found is indeed the characteristic of the load.

For a system with one warp, and N threads per warp, once the offset has been fixed, we can prefetch the data of the next access. By multiplying the offset with i (where i varies from 0 to $N-1$) and adding to the last stored address in the prefetch table for that load. This address is then put in the prefetch queue which sends the address to be prefetched to the global memory. Prefetch requests for all addresses that are in the same cache line are merged.

With more than one warp prefetching, we must ensure that the Apogee is not confused

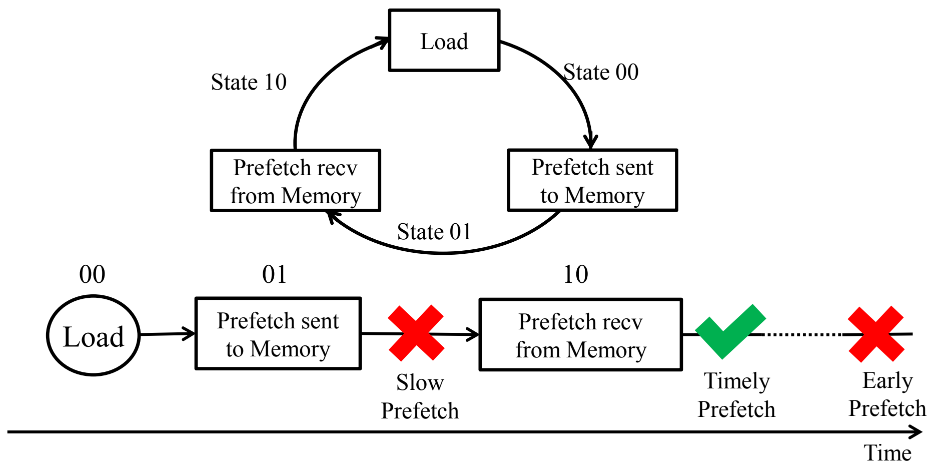


Figure 5.7: (a) The three stages in the lifetime of a prefetch entry. (b) Timeline of a prefetch and related load requests. If next load happens between 01 and 10, the prefetching is slow. If it happens just after 10 it is timely, else it is too late. Once the next load happens the status is reset to 00

by the addresses accessed by the loads of different warps. To this effect, to calculate the data to be prefetched for a warp accessing the cache, we further multiply the offset by an additional factor of $(n - N + 1)$ to get the new offset. n is the number of threads in the SM. This will ensure that only that data is prefetched which will be used by the warp in the next access. The prefetcher is designed to ensure that requests to the same cache-line are coalesced.

During any access if the offset of addresses between threads doesn't match the offset stored in the table, Apogee resets the confidence of that entry to 0. New offset calculations are done again for the next access. If some loads do not have a fixed offset, their confidence is always 0. When we need to evict an entry from the prefetch table, the entry with the lowest confidence is removed.

5.3.1.1 Dynamic distance adjustment

To adjust the distance of prefetching, every entry in the prefetch table, has a 2-bit entry per warp to store 3 states: 00, 01 and 10. The lifetime of a prefetch entry is shown in Figure 5.7. Once a load of that entry happens, its state is set to 00. After the computation of the future address, if a prefetch request is sent, the state is changed to 01. Transition from state 01 to 10 occurs when the data comes back from memory to the cache. Whenever a new load of the same entry occurs, the state is reset to 00.

When an entry is in the 01 state, it indicates that the prefetch request has been sent to the memory, but the data has not come back. If the next load for that entry happens when the entry is in the 01 state, this means the prefetched data has not returned from memory and prefetching is slow. Thus, the distance of prefetching for that entry is incremented and stored in the table. Future load instructions of that PC will use the updated distance of prefetching. As the distance of prefetching was incremented, the load for that prefetch will happen later. This will reduce the slowness of prefetching. The same mechanism will continue to increase the distance till the distance is sufficiently ahead, such that when the next load instruction accesses the cache, the data is already prefetched.

Sometimes it may happen that the distance of prefetching is higher than required. In such cases, prefetched data will have to stay in the limited cache for a longer period of time. It may get evicted even before its use occurs. These cases of early prefetching can be caught by Apogee in the following way: When a prefetch request comes back from memory to the cache, the status of that entry is updated to 10. Whenever an entry is 10 and the next load instruction of that entry is a miss in the cache, we can assume that the

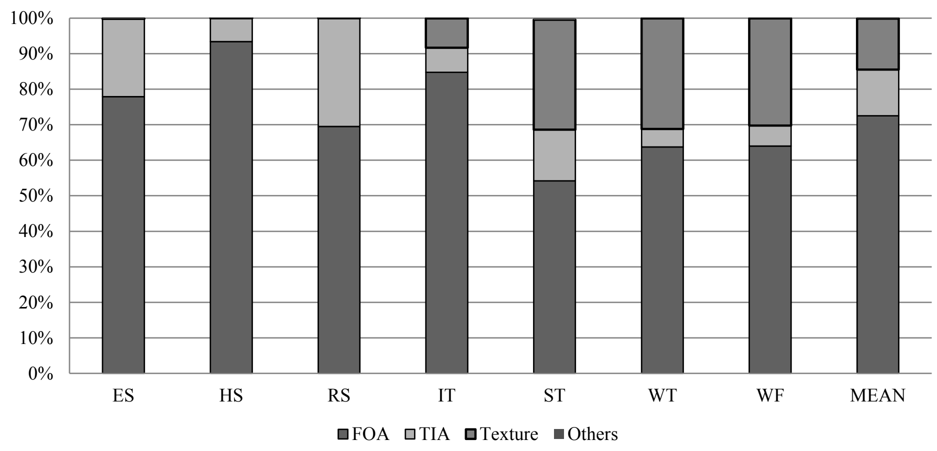


Figure 5.8: Breakdown of access patterns of loads that missed in cache.

data prefetched was either wrong or too early. Apogee recomputes the address sent for the prefetching from the entry in the table and if the addresses match, then Apogee is certain that the data was missed because it was prefetched too early. In such cases the distance of prefetching is decremented and stored back in the table. For future iterations, the new reduced distance is used for prefetching which will decrease the earliness of prefetching.

5.3.2 Thread Invariant Prefetching

Figure 5.8 shows the three major memory access patterns found in graphics applications. FOA accesses have alread been discussed in the earlier section. The remaining two access patterns are described below:

1. **Thread Invariant Addresses(TIA)**: Apart from streaming data coming in to the various stages of the graphics pipeline, a significant number of accesses are to values that are used to set up the graphics pipeline before the innermost loop in a warp starts. These variables specify the various kinds of rendering, transformations, lighting, and related information. While they are at the same address for all the threads, their

values change between invocations of the kernel.

2. **Texture Accesses**: Texturing operations are always handled by hardware texturing units as these units are designed for performance of specific texturing algorithms. Performing texture operations on generic shader cores is fundamentally different from what is done by the hardware texturing units and results in unreal memory access patterns. Since texturing operations will be handled by these hardware units, the accesses made for texturing in Figure 5.8 are not considered for this work.

TIA timeliness. Apogee dynamically detects and adapts to FOA and TIA access patterns at runtime. TIA accesses can be detected when the offset detected for an FOA is found to be zero. The data accessed by TIA accesses is 16% of the total data as shown in Figure 5.8 and to reduce multi-threading, misses to these addresses can become a performance bottleneck as per Amdahl's law. Therefore, Apogee is extended to dynamically adapt to these kinds of accesses as well. We modify the prefetcher shown in Section 5.3.1 for TIA. For an entry in the prefetch table in Figure 5.6, if the offset is found to be zero for all the accesses in that warp, we assume that the address will be constant. The thread index of that entry is set to all ones to show that the access pattern of this PC is scalar.

The main focus of TIA prefetching is not on predicting addresses, which are same anyways, but to issue the prefetch request in a timely fashion to bring evicted values back into the cache. Figure 5.9 shows an example to illustrate how the TIA prefetcher works. For a load accessing a thread invariant address (PC0) across all threads in a warp, the prefetcher tries to find a load instruction which is executed earlier than PC0 as shown in Figure 5.9(a). This earlier load is called a prefetching load instruction and behaves like

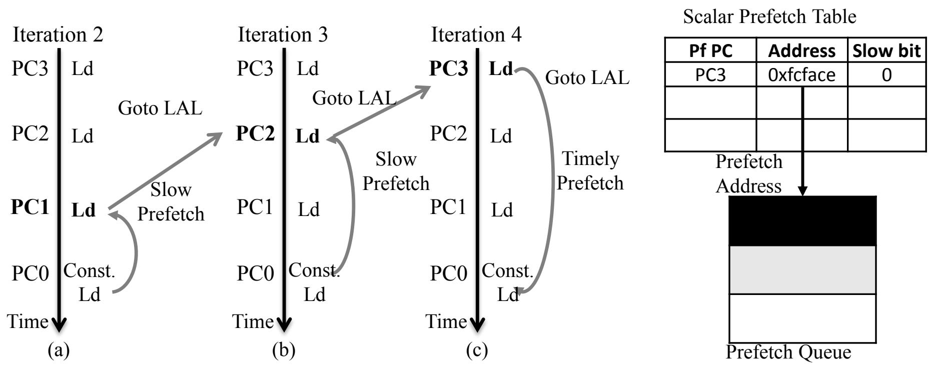


Figure 5.9: PC0 always access same address. Apogee keeps shifting the load from which the prefetch requests should be issued till it can prefetch early enough and stores that information in the Prefetch Table.

a triggering mechanism. Whenever the prefetching load instruction accesses the cache, it sends a prefetch request for the constant load for which it has been designated as a prefetching load instruction.

TIA Implementation. To find the prefetching load instruction, the prefetcher keeps the PC of the load which accessed the cache most recently. This load is known as Last Accessed Load (LAL). For every load instruction, LAL is the most recently executed load before the current load instruction. When we see a miss to a load (PC0) and if the offset entry for that PC in the prefetch table is zero, we add an entry to the table shown in Figure 5.9. The LAL (PC1) is set to be the Load PC in the entry. The address in that entry is set to be the address(AD0) which was needed by PC0. In this way the prefetch request for PC0 is made before PC0 is executed as PC1 is the LAL for PC0. This table is part of the Prefetch Table and the fields of the entry are modified on the basis of the access type. Next time when PC1, which was the LAL for PC0, does a load request, an entry for it is checked in the prefetching table and the address AD0 is found. This address(AD0) is added to the prefetch queue and sent to global memory for prefetching. PC1 may not be sufficiently ahead of PC0 such that by the time the prefetching completes, PC0 might have already sent out a request to global memory as shown in Figure 5.9(b). In such a case the entry in the prefetch table is marked as slow by using the slow bit. When the next iteration is executed and the PC of the warp is at PC1, since the entry was marked as slow, we update the Load PC of that entry to the current LAL. This LAL will be the Load that was executed before PC1. In this way we prefetch 2 load instructions ahead of PC0. This is like traversing in a linked list. We continue this till a load has been reached which is sufficiently ahead such that prefetching at that load's execution results in a hit for the TIA load as shown in Figure 5.9(c).

5.3.3 Other Stalls

Apart from hiding the latency to memory the high degree of multithreading in GPUs can hide many kinds of stalls, such as long latency floating point operations. If we completely remove multi-threading, then the stalls due to these operations cannot be hidden. Hiding the access to global memory is the majority of the work for the multithreading. With prefetching we can remove this work and the remaining stalls that the multithreading has to hide mostly include truly dependent floating point instruction. We observe that with scoreboarding, 4 warps are sufficient to hide the latency of other stalls such as datapath stalls.

Table 5.1: Experimental Parameters

Graphics	
HandShaded (`HS`)	5.8M vertices, Maya 6.5
Wolftextured (`WT`)	16.2M vertices texturing
WolfWireFrame (`WF`)	19.6M vertices wireframe
InsectTextured (`IT`)	5.5M vertices
SquidTextured (`ST`)	2.4M vertices with texturing
EngineShaded (`ES`)	1.2M vertices, state changes
RoomSolid (`RS`)	Lightscape Virtualization System
GPGPU	
FFT (`FFT`)	Fast Fourier Transform algorithm
Filter (`FT`)	bilinear filter detects edges
Hotspot (`HP`)	Evaluate temperature across grid
Mergesort(`MS`)	Parallel merge sort algorithm
Shortestpath (`SP`)	Dynamic programming algorithm

Table 5.2: Benchmark Description. (M = million)

5.4 Experimental Evaluation

5.4.1 Simulation Setup:

The benchmarks used for Graphics and GPGPU applications are mentioned in Table 5.2. We use viewsets from the SPEC-Viewperf version 9 benchmark suite which is a SPEC suite to evaluate graphics performance. To evaluate the OpenGL applications, we use Mesa3D version 7.11 as our driver, which is OpenGL version 2.1 compliant. We parallelized the main stages of the graphics pipeline using the Fractal APIs [56]. The MV5 reconfigurable simulator [56] understands these modifications in the source code and creates as many threads as mentioned in the Mesa source code whenever a kernel is launched. These threads are grouped into warps and are run in a SIMT fashion. We use the round-robin policy for scheduling of warps [8]. We modified the simulator to maintain a scoreboard and stall for true and output dependencies between instructions.

We compare Apogee with a previous prefetching technique for GPGPU applications, Many Thread Aware(MTA) [48]. In MTA prefetching, once the prefetcher is trained, it prefetches data of all the warps into a prefetch cache. Section 5.6 explains MTA in further detail. We consider five benchmarks for the analysis of GPGPU applications.

5.4.2 Performance Evaluation:

The simulation parameters for the architecture are shown in Table 5.3a. The simulator models an in-order SIMT processor, scoreboarding, timing accurate cache and models all resources such as read MSHRs, write MSHRs and uses LRU replacement policy.

Table 5.3: Experimental Parameters

(a) Simulation Parameters

Frequency	1 GHz
SIMD Width	8
Warp Size	32 threads
Number of Warps	1, 2, 4, 8, 16, 32
I-Cache	16kB, 4 way, 32B per line
D-Cache	64kB, 8 way, 32B per line
D-Cache Access Latency	4 cycles
Memory latency	400 cycle
Integer Instruction	1 cycle
FP Instruction	24 cycles [86]
Scoreboarding	True
Prefetch Table	64 Entry, 76 bits
Prefetch Issue Latency	10 cycles
Memory Bandwidth	12 GBps/SM

(b) Prefetcher Entry Area

Prefetcher Entry	
PC	10 bits
Address	28 bits
Offset	8 bits
Confidence	8 bits
Thread Idx	6 bits
Distance	6 bits
PF Type	2 bits
PF State	8 bits

5.4.3 Power Evaluation:

As the finer details of a GPU are not open knowledge, we calculate the power consumed by the GPU by calculating the dynamic and static power separately. We use the analytical power model of Hong et al. [31] to calculate the dynamic power consumed by the various portions of the GPU. For our work, we focus only on the power consumed by one SM

Technology	**TSMC 65 nm**		
Component	**Source for Area**	**GPU**	**Apogee**
Fetch/Decode	McPAT, RISC in-order model	Default	Default
FPU	Published Numbers [25]	8	8
SFU	4 FPUs [54]	2	2
Register File	Artisan RF Compiler	16k, 32-bit	2k, 32-bit
Shared Memory	CACTI	64kB	64kB
Prefetcher	Artisan SRAM Compiler	None	64, 73-bit entries

Table 5.4: Details of Area Calculation, and component wise description for the major components of baseline GPU and Apogee

with and without our solution. We do not consider the power consumed by the global memory system. The results show that on average only 2.2% extraneous memory requests are made to the memory because of the prefetcher and, so, the difference due to the power consumption of these extra requests will not be significant.

For computation of leakage power one third of the SM total power is considered to be leakage power, based on prior studies [45, 26, 75]. We classify the leakage power of an SM into core and cache power. Leakage power of the cache is measured using CACTI [67]. The remaining leakage power is distributed amongst the various modules of the SM ”core” on the basis of area. Areas of different modules are measured from EDA design tools and published figures. Table 5.4 shows component-wise source for the SM area used in the static power calculation. The most relevant leakage power number for this work, that of the register file, were obtained through the ARM Artisan Register File Compiler tool. All the measurements are at 65 nm using TSMC technology.

The prefetcher is considered to be an SRAM structure integrated with the cache controller. The dynamic and leakage power of this structure is calculated from the Artisan SRAM Compiler. The data from the global memory is brought directly into the cache and

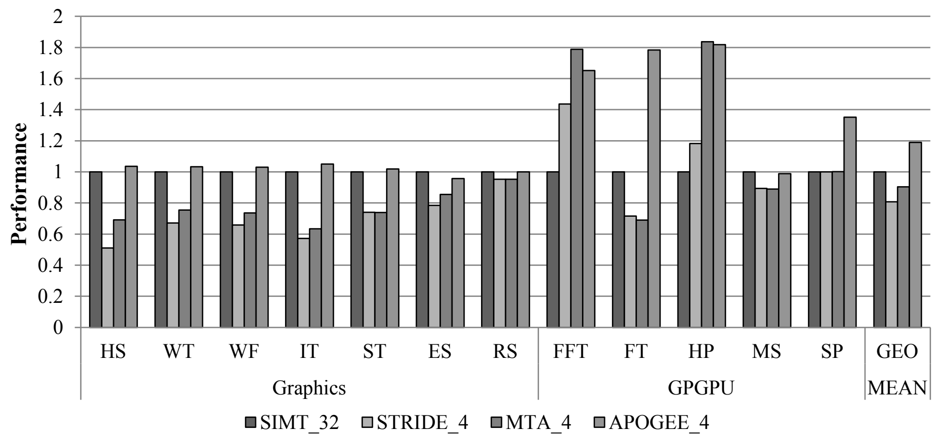


Figure 5.10: Comparison of overall speedup of SIMT with 32 warps, stride prefetcher per warp with 4 warps, MTA with 4 warps and Apogee.

no stream buffers are used. Table 5.3 shows the breakup of the prefetcher area. For all the prefetching types there is only one table. The interpretation of the entries of the table vary depending on the Prefetch Type bit. FOA prefetching needs the most amount of information. TIA prefetching needs less bits. Hence, Table 5.3 shows the number of bits required by one entry of FOA prefetching.

5.5 Results

5.5.1 Performance and Power

Performance of Prefetching with Low Multithreading. Figure 5.10 shows the speedup achieved by different prefetching techniques with a maximum of 4 concurrent warps when compared to a baseline configuration of SIMT with 32 warps (shown as SIMT_32, and as always having a value of '1'). The other configurations are stride prefetching per warp, MTA (MTA_4) and Apogee_4, all with four warps. In graphics benchmarks, Apogee_4 is similar in performance to SIMT with 32 warps. The stride prefetching and MTA techniques are

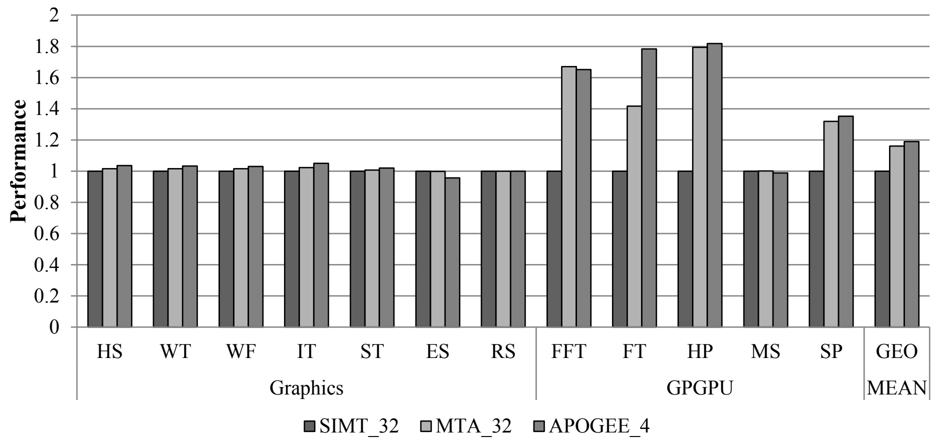


Figure 5.11: Comparison of overall speedup of SIMT with 32 warps, MTA with 32 warps and Apogee.

designed for use in GPGPU applications and, therefore, do not provide performance benefits for graphics applications. Apogee_4, however, performs better than the baseline and the two compared techniques as it has the ability to prefetch for TIA accesses in a timely fashion. For graphics applications, due to the addition of the TIA prefetcher, Apogee_4 is at par with, or better than, massively-multithreaded SIMT for six out of the seven benchmarks, providing a 1.5% speedup over the baseline 32-warp design.

For GPGPU benchmarks, stride prefetching per warp and MTA perform considerably better. However, Apogee_4 performs significantly better than these techniques in FT, MS and SP. For FFT and HP benchmarks, it is within a few percent of other techniques. The performance of Apogee_4 varies from a 4% loss to 82% better performance than the baseline. Overall Stride prefetching and MTA with 4 warps have 19% and 10% performance degradation as compared to baseline, respectively, whereas, Apogee_4 with 4 warps performs 19% better than SIMT with 32 warps across all benchmarks.

Comparison with MTA_32. The performance of MTA with 32 warps(MTA_32) and Apogee_4 with four warps as shown in Figure 5.11. The baseline configuration is SIMT

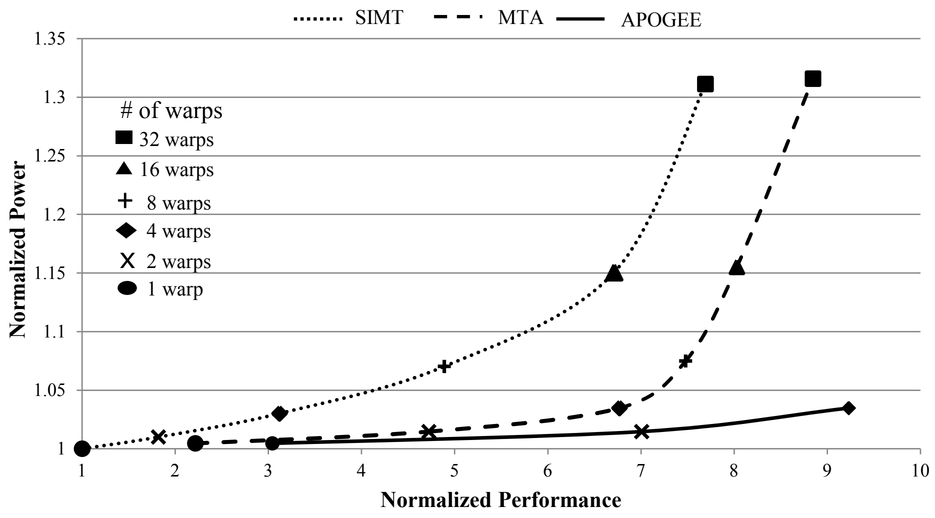


Figure 5.12: Comparison of Performance/Watt of high-degree of multithreading and multithreading with prefetching

with 32 warps. On average Apogee_4 is 3% faster than MTA_32 across all the benchmarks. MTA_32 can hide the latency of accesses that it cannot prefetch for, with the 32 warps that it has at its disposal. But with the TIA prefetcher, Apogee_4 can prefetch the data for such accesses and does not need the additional warps. The performance of MTA varies from 1% degradataion in performance to 79% improvement over baseline. On average MTA_32 provides 16% speedup over SIMT with 32 warps. Overall, MTA_4 warps provides only 90% of the performance of SIMT_32 and MTA_32 provides a 16% performance improvement over baseline. To achieve the 26% performance difference between MTA_4 and MTA_32, MTA requires an eight fold increase in the number of concurrent warps from 4 to 32.

Efficiency. Figure 5.12 shows the power consumption on the y-axis and performance on the x-axis for SIMT, MTA and Apogee_4 as the number of warps are varied. The baseline for power and performance is a configuration with one warp and no prefetching. As more warps are added to SIMT, the performance increases as long latency stalls are hidden by multi-threading. Initially with 2, 4, and 8 warps, some stalls are hidden by the additional multi-threading and the performance gain is almost linear. Due to the addition of multi-threading hardware, the power also increases. However, the change in power for the corresponding change in performance is much higher for SIMT when it goes from 16 warps to 32 warps. For the maximum performance in SIMT, the power overhead is 31.5% over baseline.

MTA provides higher performance compared to SIMT for 2, 4 and 8 warps. At 8 warps it provides 52% speedup over SIMT at similar levels of power consumption. But to get the maximum performance of MTA which is 18% more than the performance of 8 warps with MTA, it needs 32 warps which consumes a 28% higher power as compared to 8 warps. The addition of 24 warps provide diminishing returns over the first 8 warps.

Apogee_4 provides performance benefits with 4 warps, even over 32-warp SIMT and MTA implementations, while having the power consumption of 4-warp SIMT and MTA implementations. Due to the high performance of the dynamic prefetching techniques, the performance-per-watt of Apogee_4 in the best-performance case is 51% higher than that of SIMT and 33% higher than that of MTA. The performance-per-watt efficiency is much higher for Apogee_4 as fewer warps and a relatively small prefetch table provide 19% speedup and having 8 times less multithreading hardware provides significant power benefits. Every warp that is removed from the system results in simplifying the register file, branch divergence management, scheduler, etc. In this work, moving from 32 warps to 4 warps results in the removal of 14K 32-bit registers. As shown in Figure 5.2, these hardware structures are underutilized and leak a significant amount of power. The overall improvement in power-efficiency by prefetching, therefore, is due to both performance increases and reduced power consumption.

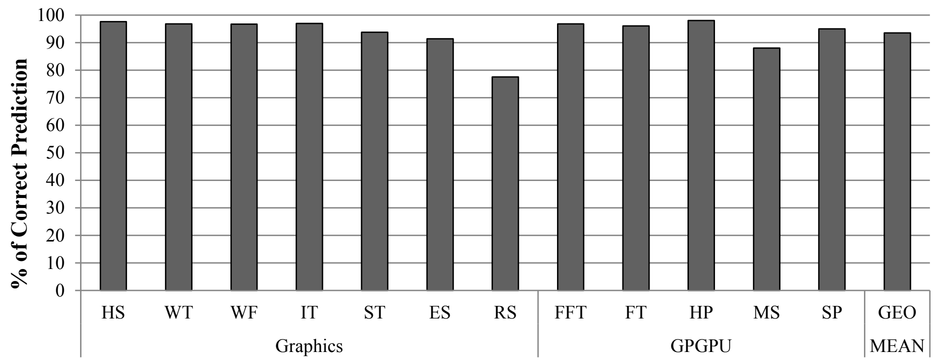


Figure 5.13: Accuracy of the overall prefetching technique.

5.5.2 Analysis of Performance:

This section analyzes the reasons for the difference in performance of Apogee as compared to MTA and SIMT. We do not analyze the same for stride prefetching on a per warp basis due its poor performance as shown in Figure 5.10.

Accuracy. The two major criterion for successful prefetching are correctly predicting addresses and their timely prefetching. Figure 5.13 shows the accuracy of the overall prefetching technique for all the benchmarks. On average, 93.5% of the addresses that were predicted by the prefetcher were correct and accessed by the core in the future. The high correctness of the prefetcher is due to the regular access patterns of the programs. Sometimes, due to certain conditional execution of instructions leading to a change in memory access patterns, the prefetcher may reset itself for a particular entry. This leads to low correctness in RS and MS benchmark. The overall access time to global memory for all loads for Apogee is reduced to 43 cycles from 400 cycles and hence does not require high degree of multithreading.

Miss Rate. We show the effectiveness of prefetching by looking at the variation in the data cache miss rate. A reduction in the data cache miss rate indicates that there are

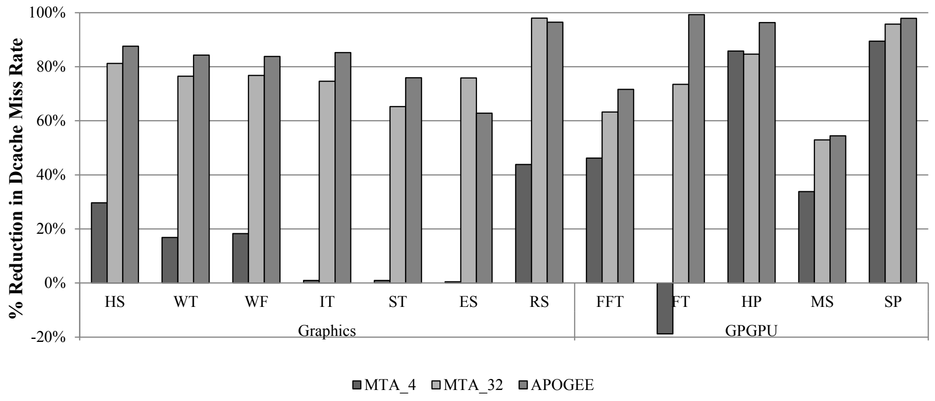


Figure 5.14: Variation in Dcache Miss Rate.

fewer requests that have to be brought in from L2 and memory. While prefetching can help in reducing the dcache miss rate, if the prefetching predicts wrong addresses or brings in data that evicts useful data, the overall miss rate will increase. So, it is a good metric to determine the effectiveness of prefetching. Figure 5.14 shows the reduction in dcache miss rate when compared to 32 warp SIMT with no prefetching. MTA_4 is able to reduce the dcache miss rate only for 4 out of the 7 graphics benchmarks. It does better for GPGPU benchmarks except for FT where it actually increases the miss rate. MTA_32 and Apogee perform considerably better than SIMT with 32 warps. They are able to reduce the baseline dcache miss rate by close to 80% in 8 benchmarks. Due to the effectiveness in prefetching, access to the main memory needs to be hidden only for the remaining 20% of the times. A low reduction in miss-rate actually transfers to low performance for ES and MS for both MTA and Apogee as shown in Figure 5.11.

Reduction in Stalls. The reduction in stall cycles of the pipeline of the core is shown in Figure 5.15. The y-axis shows the reduction in stall time normalized to SIMT with 32 warps. Due to the reduction in the dcache miss rate in MTA_32 and Apogee, there is a significant reduction in stalls. For MTA_4, since the reduction in dcache rate was

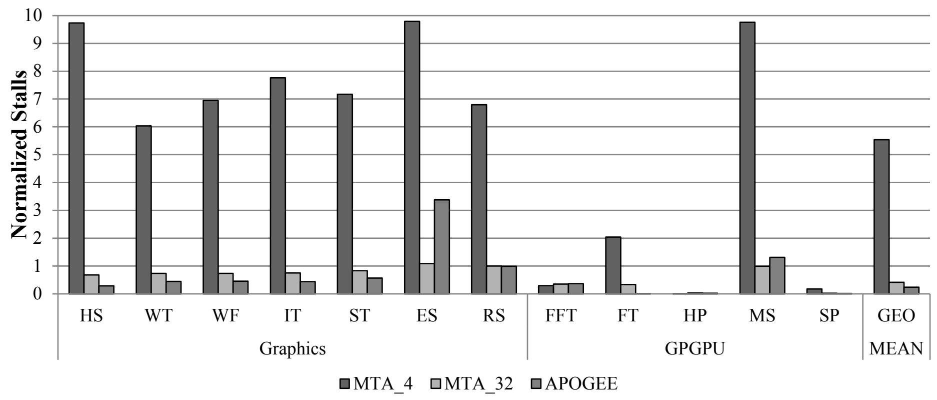


Figure 5.15: Variation in stall cycles of the SMs, normalized to SIMT with 32 warps.

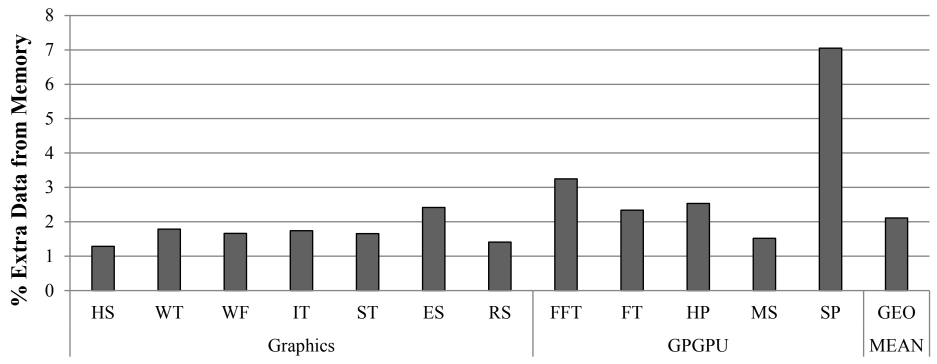


Figure 5.16: Extra data transferred from memory to the cache including prefetching requests.

not considerable, the number of cycles for which the shader pipeline stalls increases significantly due to the lack of multi-threading. MTA_32's reduction in cycles is achieved through prefetching as well as high degree of multi-threading. In Apogee there is only a fraction of multi-threading present and, yet, due to the high degree of correct prefetching, as shown in Figure 5.13, and higher reduction in dcache miss rate, as shown in Figure 5.14, it is able to eliminate more stalls than the MTA_32 technique.

Bandwidth. Data that is wrongly or untimely prefetched will not be used by the core and unnecessarily strains the bandwidth of the system. Figure 5.16 shows that on average around 2.2% extra requests are sent to the global memory by Apogee. These extra requests include prefetch and regular read requests. Graphics benchmarks have low bandwidth wastage due to prefetching as compared to the GPGPU benchmarks. Even though the accuracy of the prefetcher is around 92%, the extraneous data overhead is 2.2%. There are times when the data accessed by different threads overlap and, hence, prefetcher may predict the address that is already in the cache In such cases, no request is sent to the memory, reducing the prefetcher overhead. Overall, due to the high accuracy and timeliness of the prefetcher, it has a very small impact on an application's bandwidth consumption.

5.6 Related Work

Prefetching on GPUs has been studied in the past. Lee et al. [48] proposed MTA, Many Thread Aware prefetching for GPGPU applications. This work compares MTA with APOGEE in detail in Section 5.5. Their solution primarily addresses accuracy of prefetching with a complex throttle for wasteful prefetching which involves finding data that was prefetched and got evicted before being utilized. Such counters can be complex to implement. Apogee on the otherhand, focuses on timely prefetching by adjusting the distance of prefetching using two bits in a table. Furthermore, translating their work would addresses only one kind of access pattern for graphics application. Arnau et al. [6] shows the inefficiency of using MTA for Graphics applications. In this work we show the effectiveness of our prefetching for two major access pattern to overcome the deficiency of GPGPU prefetching techniques for graphics applications. In MTA prefetching the prediction of the next warp's data is based on the currently trained prefetcher. Once the training is complete, the data for all the warps is prefetched and stored in the prefetch cache. This approach is not very cordial for energy efficiency as show in Figure 5.12.

Arnau et al. [6] show the ineffectiveness of standard prefetching techniques for graphics applications and use a decoupled access/execute mechanism for graphics processing. But our work addresses access patterns that are specific to Graphics and work for GPGPU applications. Along with dynamically adapting to these patterns, our prefetcher also maintains the timeliness of prefetching. So, our technique is more effective than the prefetching techniques compared in [6]. They change the graphics core architecture substantially, whereas our work has less invasive changes to the core itself. We change the cache-controller and provide efficiency through prefetching. While their technique is effective for complex access pattern, their system is analyzed with 100 cycle memory access latency, we show that our technique is effective for 400 cycle latency for a graphics memory access. Prefetching for texture caches was studied by Igehy et al. [35]. They take advantage of distinct memory access pattern of mip-mapping. In Apogee only operations done at the shader core are considered and texturing is considered to be handled by separate hardware structures.

Energy efficiency on GPUs has been addressed from various directions. Gebhart et al. [26] provide a register file cache to access a smaller structure rather than the big register files. They have a two-level scheduler to select from a small set of active threads to reduce the amount of register caching required. Lin et al. [53] propose software prefetching for GPU Programs to optimize power. Their work adds prefetch instructions and vary voltage to address high power consumption. Dasika et al. [17] create a fused FPU design to reduce the number of register file accesses in a SIMT-like architecture. Yu et al. [42] increase the efficiency of the register files for GPU with a SRAM-DRAM memory design. Zhao et al. [87] propose an energy efficient mechanism to reduce the memory power consumption for GPUs via a reconfigurable in-package wide interface graphics memory on a silicon interposer. Tarjan et al. [84] proposed ”diverge on miss” and Meng et al. [56] proposed Dynamic Warp Subdivision where performance improves due to the advantage of prefetching effects by early execution of some threads in a warp. All these approaches are orthogonal to the approach used by Apogee for energy efficiency.

Adaptive prefetching has been studied for CPU architectures. Srinath et al. [80] use a feedback mechanism to adjust the prefetch accuracy, timeliness and cache pollution. Ebrahimi et al. [21] coordinate multiple prefetcher for a multicore architecture to address prefetcher caused inter-core interference. Apogee focuses on adapting to different access patterns for throughput oriented applications in GPUs.

5.7 Summary

GPUs use multithreading to hide memory access latency. The associated cost of thread contexts and register state leads to an increase in power consumption and a reduction in performance-per-watt efficiency. In this work, we demonstrate that dynamically adaptive prefetching technique is a more power-efficient mechanism to reduce memory access latency in GPUs with comparable performance. Even though conventional wisdom suggests that prefetching is more energy hungry, when used in the context of GPUs to reduce the high-degree of multithreading and related hardware, it can be used to reduce the overall power consumption.Through analysis of the memory access patterns of GPU applications, we found that fixed-offset address access and thread invariant access can prefetch for majority of the meory accesses. We demonstrate that a dynamically adaptive prefetcher can correctly predict these addresses in 93.5% of the cases. Furthermore, adaptive prefetching with dynamic distance adjustment and multithreading of 4 warps per SM, reduce the requests that go to memory by 80%. Overall, apogee with 4 warps can provide a 19% perfomance improvement over a GPU which has 32 warps per SM leading to a 52% increase in performance per watt, with 8 times reduction in multithreading support.

CHAPTER VI

Conclusion and Future Directions

The adoption of GPUs as an accelerator for data parallel programs has seen a tremendous increase as they provide high peak performance and energy efficiency. Modern GPUs are exploited not only for their large computational power but also for the large memory bandwidth and on-chip cache. GPUs use thousands of identical threads to exploit these hardware resources. In this thesis, I establish that if a GPU kernel is imbalanced then, there will be significant contention for one of the resources. This contention will eventually saturate the performance of the GPU. Traditional policies that are oblivious to a kernel's resource requirement need to be overhauled for higher efficiency.

The central theme of this thesis is to know the resource requirements at runtime and enable techniques to improve the execution efficiency for the specific case. These techniques include Dynamic Voltage Frequency Scaling, Thread throttling, better ordering of execution of warps and prefetching. I evaluate the different techniques, their effectiveness and the situation when they are most effective.

This theme of modulating resources to match application's requirement is equally applicable to other systems as well. Upcoming heterogeneous systems from AMD and Intel which fuse CPU and GPU together can extend the philosophy of this thesis. This would include implementing system wide counters to measure resource consumption in such systems at runtime and boost bottleneck resources or throttle under-utilized resources.

Currently, the DVFS mechanism used in the GPUs are emerging. The hardware and circuit that enable DVFS limit it to one decision for the entire chip. The state of the art of DVFS on CPUs have evolved to a more finer granularity which allows per cor DVFS. If this capability were to be available in future GPUs, this will make Equalizer even more effective leading to even higher efficiency. However, challenges will emerge when the different SMs require different amount of memory resources which is still a single resource. Therefore, designing future smart DVFS systems will be more rewarding with greater challenges.

Another major avenue where the learnings of this thesis can be extended is while running multiple kernels on the GPU. These kernels can have varying requirements for all kinds of resources. However, the runtime system will have to distinguish the different requirement of the different kernels and improve the overall systems. Future GPUs may have several new features at their disposal; i) per-core DVFS, ii) runtime cache bypassing and iii) kernel pre-emption. Using these tools to improve the efficiency of the GPU can be more effective with the mechanisms of this thesis.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] The Green 500.

http://green500.org. 1

[2] Top 500 Supercomputer sites.

http://top500.org. 1

[3] M. Abdel-Majeed and M. Annavaram. Warped register file: A power efficient register file for gpgpus. In *Proc. of the 19th International Symposium on High-Performance Computer Architecture*, pages 412–423, 2013. 5, 44

[4] M. Abdel-Majeed, D. Wong, and M. Annavaram. Warped gates: Gating aware scheduling and power gating for gpgpus. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 111–122, 2013. 44

[5] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012. 79

[6] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis. Boosting mobile gpu performance with a decoupled access/execute fragment processor. In *Proc. of the 39th Annual International Symposium on Computer Architecture*, pages 84–93, 2012. 112, 113

[7] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 416–427, 2012. 79

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, pages 163–174, Apr. 2009. 34, 68, 102

[9] M. Bauer, H. Cook, and B. Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011. 79

[10] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture*, pages 171–182, Jan. 2001. 43

[11] G. Chadha, S. Mahlke, and S. Narayanasamy. When less is more (limo):controlled parallelism forimproved efficiency. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, pages 141–150, 2012. 44

[12] K. Chandrasekar, B. Akesson, and K. Goossens. Improved power modeling of ddr sdrams. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 99–108, Aug 2011. 35

[13] S. Che, B. Beckmann, S. Reinhardt, and K. Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 185–195, 2013. 14

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, , J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the IEEE Symposium on Workload Characterization*, pages 44–54, 2009. 33, 50, 68

[15] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. pages 1–11, 2010. 68

[16] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995. 84

[17] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke. Pepsc: A power-efficient processor for scientific computing. In *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 101–110, 2011. 113

[18] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. Multiscale: Memory system dvfs with multiple memory controllers. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 297–302, 2012. 44

[19] Q. Deng, L. Ramos, R. Bianchini, D. Meisner, and T. Wenisch. Active low-power modes for main memory with memscale. *Micro, IEEE*, pages 60–69, 2012. 44

[20] P. Diaz and M. Cintra. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 81–92, 2009. 84

[21] E. Ebrahimi, O. Mutlu, J. L. Chang, and Y. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 316–326, 2009. 114

[22] S. Eyerman and L. Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Transactions on Architecture and Code Optimization*, 8(1):1–24, Feb. 2011. 44

[23] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proc. of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, 1992. 84

[24] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 407–420, 2007. 79

[25] S. Galal and M. Horowitz. Energy-efficient floating-point unit design. *IEEE Transactions on Computers*, 60(7):913 –922, july 2011. 104

[26] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron. Energy-efficient mechanisms for managing thread context in throughput processors. In *Proc. of the 38th Annual International Symposium on Computer Architecture*, pages 235–246, 2011. 5, 8, 44, 79, 104, 113

[27] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally. Unifying primary cache, scratch, and register file memories in a throughput processor. Proc. of the 45th Annual International Symposium on Microarchitecture, pages 96–106, 2012. 5, 79

[28] S. Gilani, N. S. Kim, and M. Schulte. Power-efficient computing for compute-intensive gpgpu applications. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 330–341, 2013. 44

[29] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proc. of the 2005 International Symposium on Low Power Electronics and Design*, pages 38–43, 2007. 44

[30] A. Hilton, S. Nagarakatte, and A. Roth. iCFP: Tolerating all-level cache misses in in-order processors. *IEEE Micro*, 30(1):12–19, 2010. 80

[31] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between CPU and GPU. In *Proc. of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 217–226, 2010. 103

[32] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 381–392, 2011. 79

[33] J. Hyeran and M. Annavaram. Warped-dmr: Light-weight error detection for gpgpu.

In *Proc. of the 45th Annual International Symposium on Microarchitecture*, pages 37–47, 2012. 80

[34] Hynix. 1gb (32mx32) gddr5 sgram h5gq1h24afr, 2009. http://www.hynix.com/datasheet/pdf/graphics/ H5GQ1H24AFR%28Rev1.0%29.pdf. 14, 35

[35] H. Igehy, M. Eldridge, and K. Proudfoot. Prefetching in a texture cache architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 133–ff, 1998. 113

[36] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 359–370, Dec. 2006. 43, 44

[37] W. Jia, K. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 272–283, 2014. 78

[38] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Orchestrated scheduling and prefetching for gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 332–343, 2013. 79

[39] A. Jog, O. Kayiran, C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das. Owl: cooperative thread array aware scheduling techniques for im-

proving gpgpu performance. 21th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 395–406, 2013. 43, 48, 68, 69, 71, 77

[40] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, pages 157–166, 2013. 13, 23, 41, 42, 43, 79

[41] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011. 55, 82

[42] W. kei S.Yu, R. Huang, S. Q. Xu, S.-E. Wang, E. Kan, and G. E. Kan. Sram-dram hybrid memory with applications to efficient register files in fine-grained multi-threading. In *Proc. of the 38th Annual International Symposium on Computer Architecture*, pages 247–258, 2011. 113

[43] KHRONOS Group. OpenCL - the open standard for parallel programming of heterogeneous systems, 2010. 47

[44] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, pages 123–134, 2008. 43

[45] P. Kogge, , et al. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. Technical Report TR-2008-13, University of Notre Dame, 2008. 5, 104

[46] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin. Dram scheduling policy for gpgpu architectures based on a potential function. *Computer Architecture Letters*, 11(2):33–36, 2012. 79

[47] J. Lee and H. Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. Proc. of the 18th International Symposium on High-Performance Computer Architecture, pages 1–12, 2012. 3, 13, 43, 48

[48] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc. Many-thread aware prefetching mechanisms for GPGPU applications. In *Proc. of the 43rd Annual International Symposium on Microarchitecture*, pages 213–224, 2010. 84, 92, 103, 112

[49] J. Lee, V. Sathisha, M. Schulte, K. Compton, and N. S. Kim. Improving throughput of power-constrained GPUs using dynamic voltage/frequency and core scaling. In *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–120, 2011. 14, 43

[50] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 270–279, 2010. 44

[51] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: enabling energy optimizations in gpgpus. In *Proc. of the 40th Annual International Symposium on Computer Architecture*, pages 487–498, 2013. 34, 43, 69

[52] J. Leng, Y. Zu, M. Rhu, M. Gupta, and V. J. Reddi. Gpuvolt: Characterizing and mitigating voltage noise in gpus. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, 2014. 34

[53] Y. Lin, T. Tang, and G. Wang. Power optimization for GPU programs based on software prefetching. In *Trust, Security and Privacy in Computing and Communications*, pages 1339–1346, 2011. 113

[54] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, march 2008. 104

[55] X. Mei, L. S. Yung, K. Zhao, and X. Chu. A measurement study of gpu dvfs on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, pages 10:1–10:5, 2013. 34, 35

[56] J. Meng. The mv5 simulator: An event-driven, cycle-accurate simulator for heterogeneous manycore architectures, 2010. https://sites.google.com/site/mv5sim/. 102, 114

[57] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 235–246, 2010. 79

[58] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 308–317, 2011. 78

[59] R. Nasre, M. Burtscher, and K. Pingali. Data-driven versus topology-driven irregular computations on gpus. In *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*, pages 463–474, 2013. 14

[60] C. Nugteren, G. J. van den Braak, H. Corporaal, and H. Bal. A detailed gpu cache model based on reuse distance theory. In *Proc. of the 20th International Symposium on High-Performance Computer Architecture*, pages 37–48, 2014. 68

[61] NVIDIA. Fermi: Nvidias next generation cuda compute architecture, 2009. http://www.nvidia.com/content/PDF/ fermi_white_papers/NVIDIA_Fermi_-Compute_Architect - ure_Whitepaper.pdf. 82

[62] NVIDIA. *CUDA C Programming Guide*, May 2011. 47

[63] NVIDIA. Gtx 680 overview, 2012. http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review/4. 34

[64] NVIDIA. Gpu boost 2.0, 2014. http://www.geforce.com/hardware/technology/gpu-boost-2. 44

[65] NVIDIA. Nvidia gpu boost for tesla, 2014. http://www.nvidia.com/content/PDF/kepler/nvidia-gpu-boost-tesla-k40-06767-001-v02.pdf. 44

[66] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated CPU-GPU systems. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 457–467, 2013. 55

[67] G. Reinman and N. P. Jouppi. Cacti 2.0: An integrated cache timing and power model. Technical Report WRL-2000-7, Hewlett-Packard Laboratories, Feb. 2000. 104

[68] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A locality-aware memory hierarchy for energy-efficient gpu architectures. In *Proc. of the 47th Annual International Symposium on Microarchitecture*, pages 86–98, 2013. 79

[69] T. G. Rogers. CCWS Simulator. https://www.ece.ubc.ca/ tgrogers/ccws.html. 69

[70] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. Proc. of the 45th Annual International Symposium on Microarchitecture, pages 72–83, 2012. 13, 34, 41, 43, 48, 67, 68, 69, 71, 78

[71] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proc. of the 46th Annual International Symposium on Microarchitecture*, pages 99–110, 2013. 78

[72] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, 2008. 47

[73] S. R. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. Reslice: selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, 2005. 80

[74] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pages 440–451, 2012. 47

[75] G. Sery., S. Borkar, and V. De. Life is cmos: why chase the life after? In *Proc. of the 39th Design Automation Conference*, pages 78 – 83, 2002. 5, 104

[76] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. Apogee: Adaptive prefetching on gpus for energy efficiency. In *Proc. of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 73–82, Sept 2013. 83

[77] A. Sethia, D. Jamshidi, and S. Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *Proc. of the 21th International Symposium on High-Performance Computer Architecture*, pages 174–185, 2015. 48

[78] A. Sethia and S. Mahlke. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *Proc. of the 47th Annual International Symposium on Microarchitecture*, pages 647–658, 2014. 15

[79] L. Sheng, H. A. Jung, R. Strong, J.B.Brockman, D. Tullsen, and N. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 469–480, 2009. 35

[80] S. Srinath et al. Feedback directed prefetching: Improving the performance and

bandwidth-efficiency of hardware prefetchers. In *Proc. of the 13th International Symposium on High-Performance Computer Architecture*, pages 63–74, Feb. 2007. 114

[81] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Ghandi, and M. Upton. Continual flow pipelines: achieving resource-efficient latency tolerance. *IEEE Micro*, 24(6):62–73, 2004. 80

[82] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. Technical report, University of Illinois at Urbana-Champaign, 2012. 33, 50, 68

[83] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, 2009. 44

[84] D. Tarjan, J. Meng, and K. Skadron. Increasing memory miss tolerance for simd cores. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 22:1–22:11, 2009. 114

[85] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, Hewlett-Packard Laboratories, Apr. 2008. 70

[86] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Proc. of the 2010 IEEE*

Symposium on Performance Analysis of Systems and Software, pages 235–246, 2010. 103

[87] J. Zhao, G. Sun, G. H. Loh, and Y. Xie. Energy-efficient gpu design with reconfigurable in-package graphics memory. pages 403–408, 2012. 113

[88] Z. Zheng, Z. Wang, and M. Lipasti. Adaptive cache and concurrency allocation on gpgpus. *Computer Architecture Letters*, PP(99), 2014. 78