Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution

Ankit Sethia and Scott Mahlke
Advanced Computer Architecture Laboratory
University of Michigan, Ann Arbor, MI
{*asethia, mahlke*}@*umich.edu*

***Abstract*—GPUs use thousands of threads to provide high performance and efficiency. In general, if one thread of a kernel uses one of the resources (compute, bandwidth, data cache) more heavily, there will be significant contention for that resource due to the large number of identical concurrent threads. This contention will eventually saturate the performance of the kernel due to contention for the bottleneck resource, while at the same time leaving other resources underutilized. To overcome this problem, a runtime system that can tune the hardware to match the characteristics of a kernel can effectively mitigate the imbalance between resource requirements of kernels and the hardware resources present on the GPU. We propose Equalizer, a low overhead hardware runtime system, that dynamically monitors the resource requirements of a kernel and manages the amount of on-chip concurrency, core frequency and memory frequency to adapt the hardware to best match the needs of the running kernel. Equalizer provides efficiency in two modes. Firstly, it can save energy without significant performance degradation by throttling under-utilized resources. Secondly, it can boost bottleneck resources to reduce contention and provide higher performance without significant energy increase. Across a spectrum of 27 kernels, Equalizer achieves 15% savings in energy mode and 22% speedup in performance mode.**

***Keywords*-GPGPUs; Runtime System; Resource Utilization; Dynamic Voltage and Frequency Scaling;**

I. 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 6 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 [17]. 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. [26] and Kayiran et al. [15] 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 6 Gbps, the idle standby current is 30% higher as compared to when it provides 3.2 Gbps [12]. 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. [18] 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 [6, 22]. 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, 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.

II. 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.

A. *Effect of Execution Parameters*

Figure 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 V. A black star mark on each sub-figure shows the position of the baseline. The four quadrants ①, ②, ③and ④in the sub-figures represent deviation from the black star. In quadrant ①performance improves and efficiency decreases, while in quadrant ②performance and efficiency decrease. In quadrant ③performance and efficiency increase, while in quadrant ④performance decreases and efficiency increases.

SM Frequency: Figure 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 1b. In such kernels, the SM's

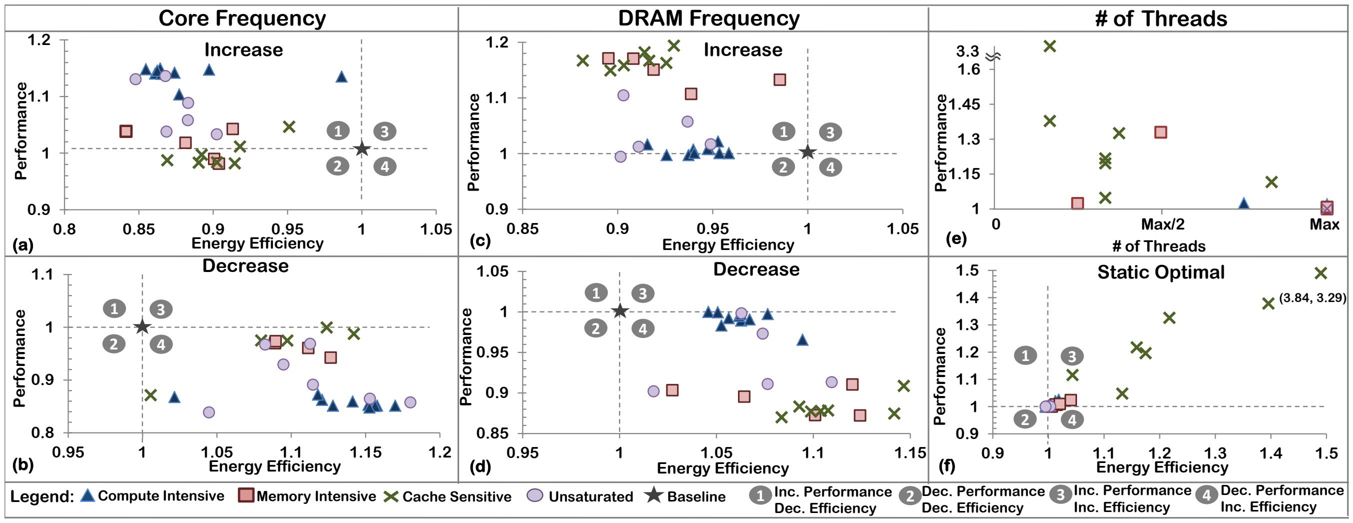


Figure 1: Impact of variation of SM frequency, DRAM frequency and number of concurrent threads on performance and energy efficiency. The black star mark shows the value for the baseline GPU and is always set to 1 for both performance and energy. The properties of the four quadrants relative to the star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

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 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 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.

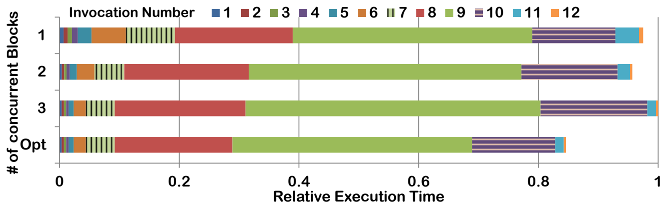
Number of Thread Blocks: Increasing the DRAM frequency helps cache kernels get data back faster. However, controlling the number of threads to reduce L1 data cache thrashing will improve performance significantly with minimal energy increase. Therefore, we first analyze the optimal number of threads that need to run on an SM. Figure 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 1f shows the improvement in energy efficiency, 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

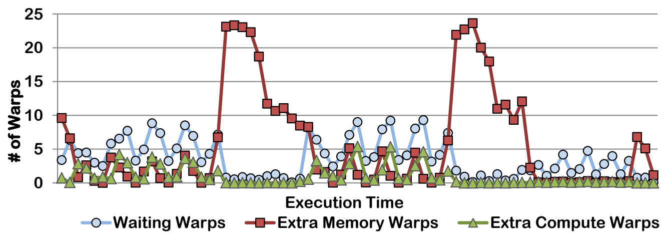
Actions for Dynamic System: The action of increasing, maintaining, or decreasing the three parameters depend on the objective of the user. If the objective is to save energy, the SM and memory frequency should be reduced for memory and compute kernels respectively. If the objective is to improve performance, the SM and memory frequency should

Table I: Actions on different parameters for various objectives

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



(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 2: Variation of kernel requirements across and within kernel instances.

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 I.

B. Kernel Variations

Kernels not only show diverse static characteristics in the resources they consume, but their requirements also vary across and within invocations. Figure 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 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 2b for the mri-g-1 benchmark. Over most of the execution time, there are

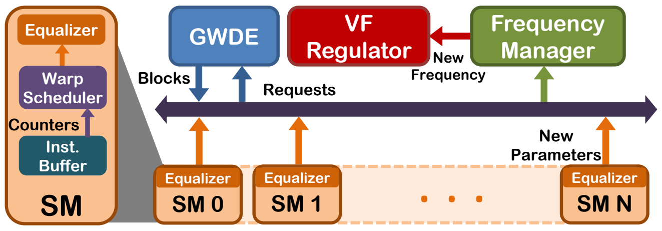


Figure 3: Equalizer Overview: Each SM requests new thread block and the new SM and memory frequency domain. The frequency manager takes the requests and changes the SM or memory frequency as requested by majority of the SMs. GWDE manages the request for new thread blocks.

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, 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.

III. 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 III-B.

Figure 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

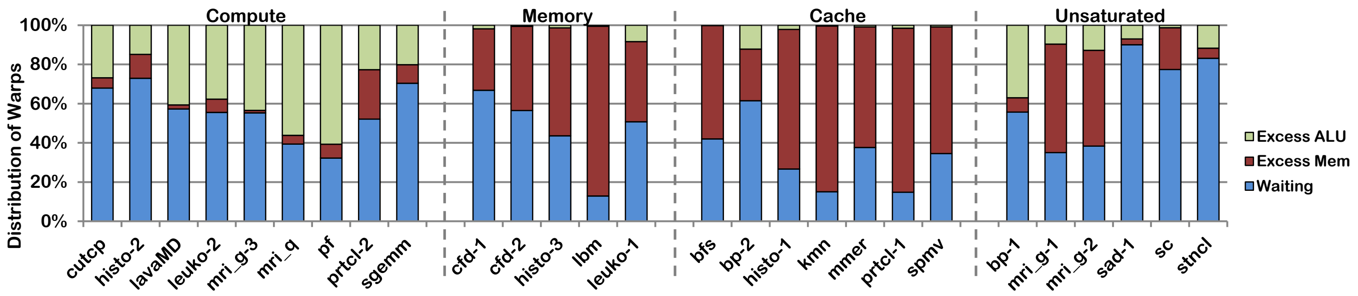


Figure 4: State of the warps for different categories of kernels. The names on top of the chart show the various categories.

threads, it uses the CTA Pausing technique used in [15] (Section IV-B). 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 receives these requests and makes a global decision for the new VF level for the SM and memory based on a majority function.

A. 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.

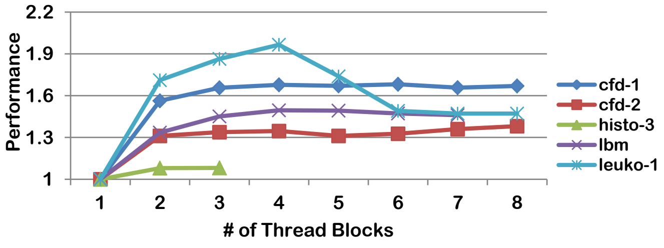


Figure 5: Performance of memory intensive kernels with number of concurrent thread blocks

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 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.

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 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

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
```

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.

B. 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 I. 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 I. 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 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 III-A, 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 III-A. 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 IV-B. 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 III-A. 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 I, for compute kernels, if the objective of Equalizer is to save energy, then it requests the frequency manager in Figure 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 I.

Equalizer also provides a unique opportunity for imbalanced kernel invocations. If the kernel has a few long

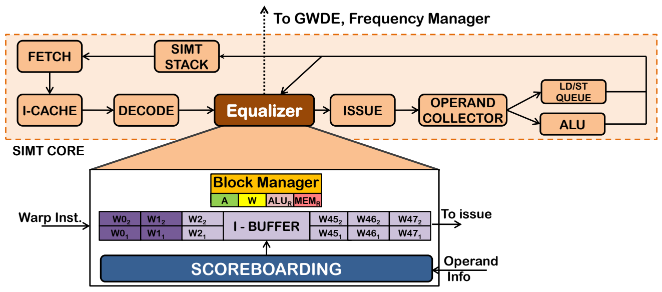


Figure 6: Equalizer Overview. Equalizer is attached with the instruction buffer and periodically calculates the four counter values. At the end of an *epoch* it sends a request to the Global Work Distribution Engine.

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.

IV. 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 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.

A. 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 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 III are implemented as follows:

- 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.

B. Thread Block Allocation Mechanism

After the collection of counter values, at the end of an *epoch* window, Equalizer uses the algorithm explained in Section III-B 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. 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 $numBlocks$.

C. 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 I. The frequency manager shown in Figure 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 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.

Table II: 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

Table III: 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	±15%, on-chip regulator
Memory V/F Modulation	±15%

V. Experimental Evaluation

A. Methodology

Table II describes the various applications, their kernels and characteristics. These kernels are from the Rodinia suite [7] and parboil [28] suite. We use the kmeans benchmark used by Rogers et al. [26] that uses large input. We use GPGPU-Sim [3] 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 III.

1) Energy Modelling: We use GPUWattch [19] 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%[21] and we chose 15% as the change in GPU VF level. We assume linear change in voltage for any change in the frequency [24]. 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 [19]. Leng et. al have shown that the voltage guardbands on GPUs are more than 20% [20] 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 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 [27]. For DRAM modelling, we use the various operating points of the Hynix GDDR5 [12]. 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 [5]. Commercial Kepler GPUs allow memory voltage/frequency to change by significant amount but we restrict the changes conservatively to 15%[21]. 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 V-B, 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.

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.

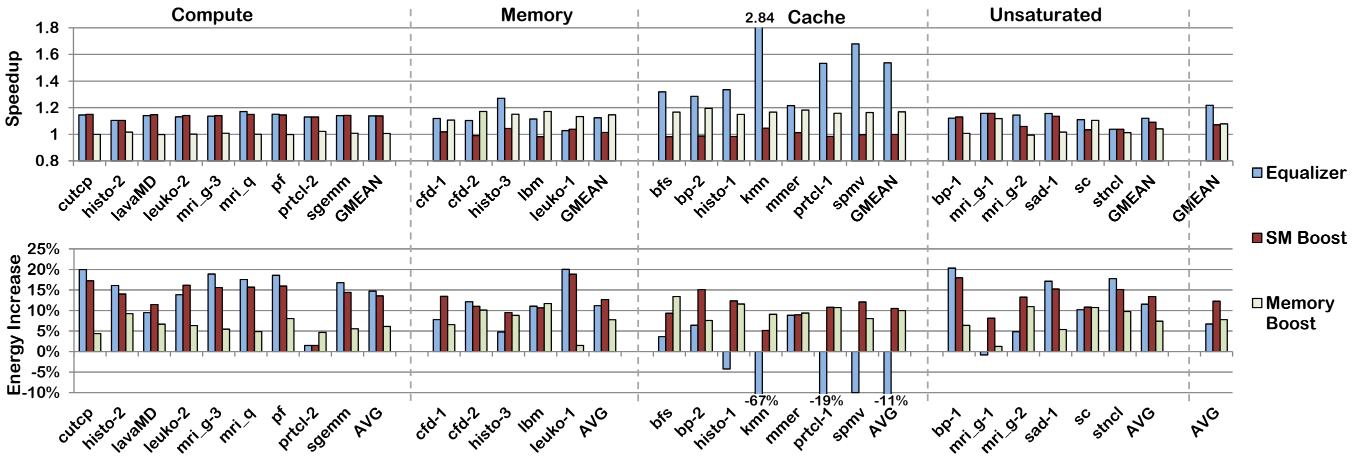


Figure 7: Performance and increase in Energy consumption for performance mode

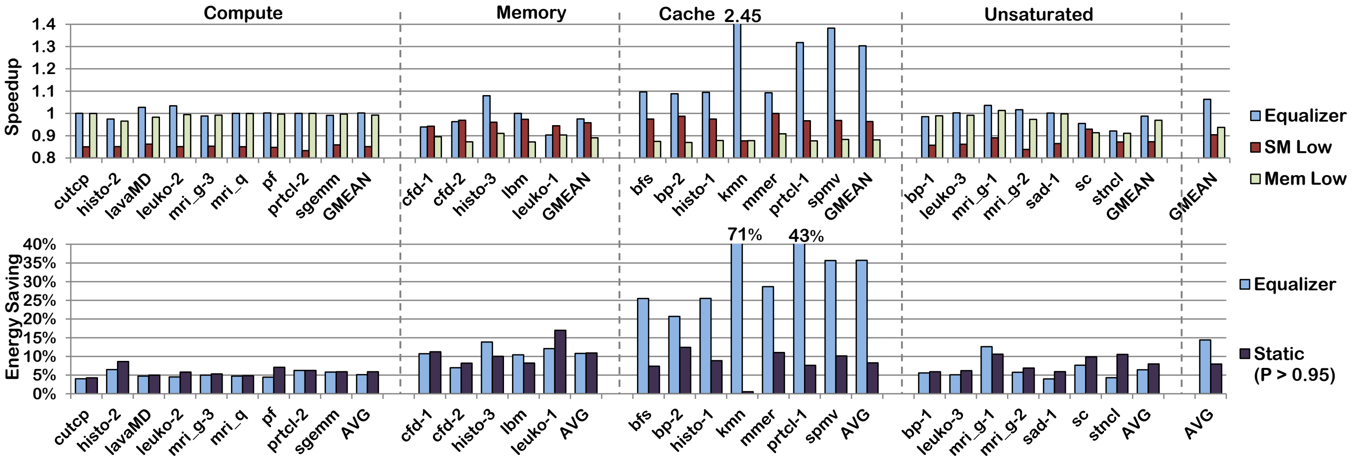


Figure 8: Performance and Energy savings in energy mode. Static bar on right for energy savings is either SM low or mem low from the top graph when the performance is above 0.95.

B. Results

We show the results of Equalizer in performance mode and energy mode for the four kernel categories in Figure 7 and 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 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 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 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

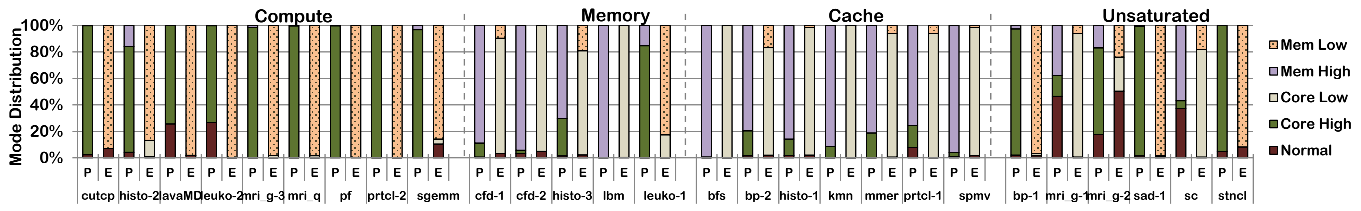


Figure 9: Distribution of time for the various SM and memory frequency. P and E are data for performance and energy mode respectively

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 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 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 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 reducing memory frequency cannot provide very high savings. However, memory kernels save 11% energy

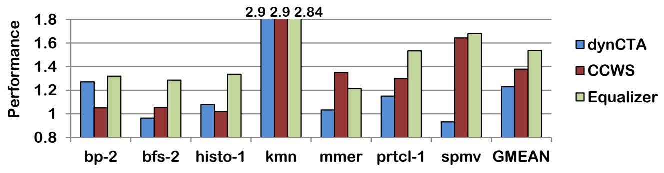
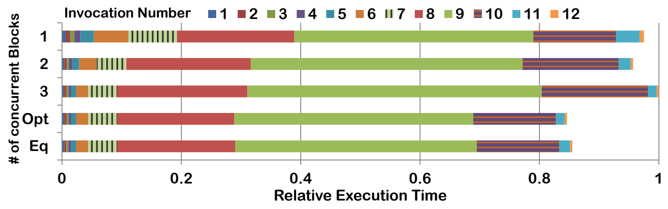


Figure 10: Comparison of Equalizer with DynCTA and CCWS

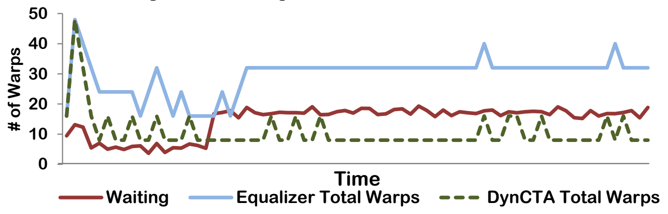
by lowering SM frequency. For cache sensitive kernels the energy savings for Equalizer is 36%, which is larger than performance mode's energy savings. This is due to throttling SM frequency rather than boosting memory frequency. For unsaturated kernels the trend of their resource utilization is effectively 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 9. Normal, high and low modes are explained in Section IV-C. 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.

We compare the performance of Equalizer with DynCTA [15], a heuristics based technique to control the number of thread blocks and CCWS [26], which controls the number of warps that can access data cache in Figure 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 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.



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



(b) Adaptiveness of Equalizer within an invocation for spmv

Figure 11: Adaptiveness of Equalizer

Equalizer adaptiveness: The impact of adaptability of Equalizer is demonstrated for inter-instance variation and intra instance variations in Figure 11. The stacked bars for 1, 2, 3 and the optimal number of blocks were shown in Figure 2a. Another bar for Equalizer is added for comparison to that figure and shown in Figure 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 11b shows the adaptiveness of Equalizer within an instance of the spmv kernel. We also compare the dynamism of Equalizer with DynCTA [15]. 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).

VI. Related Work

Managing number of threads for cache locality: Kayiran et al. propose dynCTA [15] 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. [26] 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 V-B. Unlike [15, 26], we also target efficient execution for kernels that are not cache sensitive. Jog et al. proposed OWL Scheduling [14] 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. [17] propose a thread level parallelism aware cache management policy for integrated CPU-GPU architectures.

DVFS on GPU: Lee and Satisha et al. [18] 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. [19] 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 [13, 4, 16]. 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 [25, 23] 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.

DVFS on Memory: DVFS of DRAM has been studied by Deng et al. [8, 9]. 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. [1] 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 [2]. Gilani et al. [11] show the benefits of using scalar units for computations that are duplicated across threads. Gebhart et al. [10] 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.

VII. Conclusion

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.

Acknowledgements

This work has been was supported by ARM Ltd. and National Science Foundation grant SHF-1217917. The authors would like to thank members of the CCCP research group, and the anonymous reviewers for their insightful suggestions and feedback.

References

[1] Mohammad Abdel-Majeed and Murali Annavaram. "Warped Register File: A Power Efficient Register File for GPGPUs". In: *Proc. of the 19th International Symposium on High-Performance Computer Architecture*. 2013, pp. 412–423.

[2] Mohammad Abdel-Majeed, Daniel Wong, et al. "Warped Gates: Gating Aware Scheduling and Power Gating for GPGPUs". In: *Proc. of the 46th Annual International Symposium on Microarchitecture*. 2013, pp. 111–122.

[3] Ali Bakhoda, George L. Yuan, et al. "Analyzing CUDA Workloads Using a Detailed GPU Simulator". In: *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*. Apr. 2009, pp. 163–174.

[4] David Brooks and Magaret Martonosi. "Dynamic Thermal Management For High-Performance Microprocessors". In: *Proc. of the 7th International Symposium on High-Performance Computer Architecture*. Jan. 2001, pp. 171–182.

[5] K. Chandrasekar, B. Akesson, et al. "Improved Power Modeling of DDR SDRAMs". In: *Digital System Design (DSD), 2011 14th Euromicro Conference on*. Aug. 2011, pp. 99–108.

[6] Shuai Che, B.M. Beckmann, et al. "Pannotia: Understanding irregular GPGPU graph applications". In: *2013 IEEE International Symposium on Workload Characterization*. 2013, pp. 185–195.

[7] Shuai Che, M. Boyer, et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization*. 2009, pp. 44–54.

[8] Qingyuan Deng, David Meisner, et al. "MultiScale: Memory System DVFS with Multiple Memory Controllers". In: *Proc. of the 2012 International Symposium on Low Power Electronics and Design*. 2012, pp. 297–302.

[9] Qingyuan Deng, L. Ramos, et al. "Active Low-Power Modes for Main Memory with MemScale". In: *Micro, IEEE* (2012), pp. 60–69.

[10] Mark Gebhart, Daniel R. Johnson, et al. "Energy-efficient mechanisms for managing thread context in throughput processors". In: *Proc. of the 38th Annual International Symposium on Computer Architecture*. 2011, pp. 235–246.

[11] S.Z. Gilani, Nam Sung Kim, et al. "Power-efficient computing for compute-intensive GPGPU applications". In: *Proc. of the 19th International Symposium on High-Performance Computer Architecture*. 2013, pp. 330–341.

[12] Hynix. *1Gb (32Mx32) GDDR5 SGRAM H5GQ1H24AFR*. http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR%28Rev1.0%29.pdf. 2009.

[13] Canturk Isci, Gilberto Contreras, et al. "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*. Dec. 2006, pp. 359–370.

[14] Adwait Jog, Onur Kayiran, et al. "OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance". In: 21th International Conference on Architectural Support for Programming Languages and Operating Systems. 2013, pp. 395–406.

[15] Onur Kayiran, Adwait Jog, et al. "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs". In: *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*. 2013, pp. 157–166.

[16] Wonyoung Kim, Meeta S. Gupta, et al. "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*. 2008, pp. 123–134.

[17] Jaekyu Lee and Hyesoon Kim. "TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture". In: Proc. of the 18th International Symposium on High-Performance Computer Architecture. 2012, pp. 1–12.

[18] Jungseob Lee, V. Sathisha, et al. "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*. 2011, pp. 111–120.

[19] Jingwen Leng, Tayler Hetherington, et al. "GPUWattch: enabling energy optimizations in GPGPUs". In: *Proc. of the 40th Annual International Symposium on Computer Architecture*. 2013, pp. 487–498.

[20] Jingwen Leng, Yazhou Zu, et al. "GPUVolt: Characterizing and Mitigating Voltage Noise in GPUs". In: *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*. 2014.

[21] Xinxin Mei, Ling Sing Yung, et al. "A Measurement Study of GPU DVFS on Energy Conservation". In: *Proceedings of the Workshop on Power-Aware Computing and Systems*. 2013, 10:1–10:5.

[22] R. Nasre, M. Burtscher, et al. "Data-Driven Versus Topology-driven Irregular Computations on GPUs". In: *IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS)*. 2013, pp. 463–474.

[23] NVIDIA. *GPU Boost 2.0*. http://www.geforce.com/hardware/technology/gpu-boost-2. 2014.

[24] NVIDIA. *GTX 680 Overview*. http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review/4. 2012.

[25] NVIDIA. *NVIDIA GPU BOOST FOR TESLA*. http://www.nvidia.com/content/PDF/kepler/nvidia-gpu-boost-tesla-k40-06767-001-v02.pdf. 2014.

[26] Timothy G. Rogers, Mike O'Connor, et al. "Cache-Conscious Wavefront Scheduling". In: Proc. of the 45th Annual International Symposium on Microarchitecture. 2012, pp. 72–83.

[27] Li Sheng, Ho Ahn Jung, et al. "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures". In: *Proc. of the 42nd Annual International Symposium on Microarchitecture*. 2009, pp. 469–480.

[28] John A. Stratton, Christopher Rodrigrues, et al. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Tech. rep. IMPACT-12-01. University of Illinois at Urbana-Champaign, Mar. 2012.