ABSTRACT

Emerging GPU architectures offer a cost-effective computing platform by providing thousands of energy-efficient compute cores and high bandwidth memory that facilitate the execution of highly parallel applications. In this paper, we show that different applications, and in fact different kernels from the same application might exhibit significantly varying utilizations of compute and memory resources. In order to improve the energy efficiency of the GPU system, we propose a run-time characterization strategy that classifies kernels as compute- or memory-intensive based on their resource utilizations. Using this knowledge, our proposed mechanism employs core shut-down technique for memory-intensive kernels in order to manage energy in a more efficient way. This strategy uses performance and memory bandwidth utilization information to determine the ideal hardware configuration at run-time. The proposed technique saves on average 21% of total chip energy for memory-intensive applications, which is within 8% of the optimal saving that can be obtained from an oracle scheme.

1 INTRODUCTION

GPUs are being increasingly employed to accelerate different types of computing platforms ranging from embedded devices to supercomputers. As a result, today’s GPUs are not running only graphics applications, but also, applications from database domain and high-performance computing domain, among others. This makes GPU a general-purpose computing platform or shortly GPGPU. To cope with contrasting demands of these different types of applications, GPU architects keep increasing on-chip resources such as cores, caches, software-managed memories and memory controllers (MCs); and projections include even more powerful resource-rich GPU systems in future. An important issue at this juncture is whether current applications effectively utilize available on-chip resources in GPUs and, if not, what are the reasons behind it? and more importantly, which classes of design optimizations can be done in GPGPUs to improve their performance/energy per cost? Recent studies (Hong and Kim 2010, Lee et al. 2011, Leng et al. 2013, Rogers et al. 2012) have focused on this resource utilization problem and proposed techniques that handle underutilized hardware components in GPUs. Here, we look at resource utilization and the promising energy optimization based on run-time characteristics of the kernels.
Our experimental study shows that both compute cores and memory bandwidth are underutilized in many GPU applications. In highly memory-intensive applications, cores get stalled frequently because all the threads are waiting for the memory requests. On the other hand, in compute-intensive applications, memory experiences long idle periods. Moreover, as different kernels execute different parts of the same application, they may exhibit significant variations regarding utilizations of compute cores and memory bandwidth, making a universal solution that works across different applications highly unlikely. Motivated by this observation, this paper proposes an energy-saving strategy that exploits resource underutilization in GPUs. More precisely, in this work we focus on core underutilization in GPUs. With the knowledge that increasing the degree of parallelism does not necessarily improve the performance (due to congestion in the interconnection network, contention in the last level cache, and memory bandwidth saturation (Hong and Kim 2010, Guz et al. 2009)), we propose a strategy to regulate the number of active cores when they are over provisioned for a particular memory-intensive kernel. Our proposed mechanism manages the degree of parallelism through a feedback-driven CTA scheduling and adopts a core shut-down strategy to manage energy consumption. Although a warp-throttling technique (Rogers et al. 2012) could be employed to resolve the existing problem on memory side by reducing the number of concurrently running threads, our feedback-driven CTA scheduler is compatible with core shut-down techniques. Note that, a core shut down mechanism also reduces the static leakage power which contributes to a considerable portion of the total chip power consumption (Leng et al. 2013). We also demonstrate that our mechanism outperforms a typical core-side DVFS technique.

The rest of the paper is organized as follows. Section 2 provides background on our target GPU architecture. In Section 3, we describe the resource underutilization problem in GPUs. In Section 4, we present our runtime technique for regulating number of active cores through feedback-driven CTA scheduling. Section 5 presents an experimental evaluation of the proposed strategy. Section 6 reviews related works and Section 7 concludes the paper.

2 BACKGROUND

In this section, we provide a brief background on the GPU architecture targeted by our work.

**GPU Architecture:** Our target GPU consists of multiple streaming multiprocessors (SMs) (Terms “core” and “SM” are used interchangeably in this paper.), each containing 32 CUDA cores (Fermi 2009). Each CUDA core can execute a thread, in a “Single-Instruction, Multiple-Threads” fashion. This architecture is supported by a large register file that hides the latency of memory accesses. The memory requests generated by multiple concurrently executing threads in an SM are coalesced into fewer cache lines and sent to L1 data cache, shared by all CUDA cores in the SM. Memory requests are injected into the network, which connects the SMs to 6 memory partitions through a crossbar. Each memory partition includes a slice of shared L2 cache, and a memory controller. Figure 1 shows this baseline architecture. We further assume that the system supports per-SM power-gating (Hong and Kim 2010).

**GPGPU Applications:** A typical GPGPU application consists of one or multiple kernels each of which is launched once or multiple times during the entire execution of the application. These kernels implement specific modules of an application. Each kernel consists of a set of parallel threads. As shown in Figure 1, these threads are divided into groups of threads, called Cooperative Thread Arrays (CTAs). The underlying architecture further divides each CTA into groups of threads, called warps, that is transparent to the programmer. The execution on GPU starts with performing memory allocation in GPU memory. Then, CPU copies the required data into the allocated memory, and a kernel is launched on GPU. After a kernel is launched, the CTA scheduler schedules available CTAs associated with the kernel on all the available cores. The maximum number of CTAs per core is limited by core resources (i.e., number of threads, size of shared memory, register file size, and etc). In a finer granularity, the CTA assignment policy is followed by per-core
warp scheduling. Warps associated with CTAs are scheduled on the assigned cores and get equal priority. Once a core finishes executing of a CTA, the CTA scheduler assigns another CTA to that core to execute. In such scheduling mechanism, there is no priority among CTAs and the process continues until all the CTAs are executed. When all the threads finish their computation, the results are copied to the CPU memory and the GPU memory is freed. At this point the CPU can launch the next kernel on the GPU. In Section 4, we explain how our sampling and reconfiguration mechanism cooperates with the CTA scheduler at run-time.

Figure 1: Target GPGPU architecture and the details of the computation hierarchy in GPGPU applications.

3 MOTIVATION

3.1 Investigating Resource Underutilization

The main philosophy of GPGPU architectures is to provide a large number of computing cores supported by a high bandwidth memory, in order to have a high throughput system. Such a resource-rich design will be cost-effective only if the main resources such as computing cores and memory are effectively utilized by hosted application. Thus, it is vital to understand the impact of different applications on the utilization of GPU resources. Typically, memory-intensive applications utilize the memory bandwidth properly, but might not need all the cores to achieve the optimal performance. In this section, we discuss different type of applications/kernels in terms of core utilization and its relationship to the available memory bandwidth.

- **Core Utilization at the Application Level:** To illustrate the effect of the number of cores on the system performance, let us examine Figure 2 (our experimental setup will be given in Section 5.1). This figure shows the application performance, as we vary the number of available cores. Among these applications, PATH is the only compute-intensive application, and its performance increases linearly as we increase the number of cores. The other two applications are memory-intensive, and we observe that their performance does not improve beyond a certain point. In fact, we even observe some performance loss in BFS as we allocate it more than about 20 cores. Since each thread has a certain memory bandwidth demand, as we increase the number of cores we also increase the number of memory transactions from cores to memory per unit of time. Beyond a certain number of cores, this increase in memory requests could cause the memory bandwidth to saturate. Therefore, beyond this point, using additional cores will not improve performance; instead, it could lead to longer memory accesses latencies. Furthermore, it could also cause too many contentions in the last level cache (as it is the case for BFS application) which consequently degrades the performance.

- **Core Utilization at the Kernel Level:** In order to have a more detailed analysis on the impact of number of active cores on the system performance, we investigate our applications at a finer granularity, i.e., at the kernel level. Each GPGPU application consists of one/multiple kernel(s), each of which is launched once/multiple-times during the execution of that application. Based on our observations, not only different applications but also different kernels that belong to the same application might exhibit a large variance in their resource demands, leading to diverse resourceutilizations across different phases of the application.
Figure 2: Effect of increasing the number of cores on the performance of different applications. The results are normalized to the highest IPC observed in each application over 32 different core allocations. (Each application consists of one main kernel that is called multiple times during the course of execution.)

Figure 3: Effect of increasing the number of cores on the performance of four different kernels from MST application. Kernel-4 exhibits compute-intensive behavior while other kernels have memory-intensive characteristics each of which with different saturation point.

4 GPU RESOURCE MANAGEMENT

In this section, we describe our strategy to find the ideal SM count for each individual kernel. First, we discuss our dynamic CTA-based kernel characterization mechanism. Then, we explain how our proposed scheme uses the collected statistics to converge to the optimal number of SMs through feedback-driven CTA scheduling.

4.1 Run-Time Characterization

In order to investigate the effect of available compute and memory resources on the overall system performance, we monitor the memory bandwidth utilization (MBU) as well as instruction per cycle (IPC) metrics. Once a kernel is launched by the CPU, we allocate all the available SMs to that kernel. The next step is to perform sampling to study the characteristics of the running kernel. In our experimental study, we observed that assuming a fixed window size for sampling phase (in terms of number of cycles) is not accurate because the execution time of different kernels are widely variable for a fixed window size. Therefore, our goal is to have a dynamic window size for the sampling phase to accurately capture the behavior of each individual kernel at run-time. To this end, we analyze kernel-based and CTA-based sampling techniques.

- Kernel-Based Sampling: Within a majority of GPGPU applications, two basic properties exist. First, most of the kernels of an application are launched multiple times during the execution of the application. Second, different invocations of the same kernel exhibit very similar behavior. These two common properties motivate us to exploit the first execution of a kernel as the sampling phase. Note that, such kernel-based approach is not applicable for two different situations: First, if the kernel is launched only once during the execution of the program. Second, if the kernel does not exhibit consistent behavior over different invocations.
Figure 4: Normalized IPC of different kernels during the execution of the application. Kmean, PVC1, and PVR1 do not exhibit consistent behavior over different invocations. LIB and MUM consist of one main kernel that are executed only once.

Figure 5: Transitioning between the compute-intensive (CI) and memory-intensive (MI) states in PVC. Kernel-1 exhibits memory-intensive behavior. Kernel-2 however, experiences transitions between compute-intensive and memory-intensive phases.

Figure 4 shows the normalized IPC of different kernels over different invocations. As can be seen, most of the kernels exhibit consistent behavior over different invocations. However, LIB and MUM consist of only one kernel which are executed only once. Kmean, PVC1, and PVR1 on other hand, although executed multiple times, exhibit inconsistent behavior over different executions. Figure 5 demonstrates this issue for PVC more clearly. Kernel-1 represents a memory-intensive kernel with consistent behavior. Kernel-2 however, transits between compute and memory-intensive phases. Therefore, we cannot use one execution to optimize future invocations of that kernel. Besides these two drawbacks, a kernel-based approach does not capture the behavioral transitions during each kernel execution.

• CTA-Based Sampling: As described in Section 2 (see Figure 2), each kernel is split into smaller blocks called CTAs that execute similar portions of the code. SMs start executing a kernel with the maximum possible number of CTAs, and whenever a CTA finishes, the CTA scheduler launches another CTA (if any) to the available SM. This procedure continues until all the CTAs are finished. Because these CTAs run similar code, each of them can represent the behavior of the kernel. This property motivates us to employ a CTA as a fine yet accurate granularity in our sampling process. Based on our experimental observations, in order to have an accurate sampling, we consider the first group of scheduled CTAs (i.e., #SMs*#CTAs-Per-SM) as the sampling window. However, the monitoring hardware continuously analyzes the behavior of running kernel, and if it recognizes noticeable changes in the behavior of running kernel, it will accordingly update the collected statistics. Unlike kernel-based approach, a CTA-based scheme can be exploited for (i) kernels that are launched only once, (ii) kernels with inconsistent behavior over different invocations, and (iii) also recognizes the behavioral changes within each kernel execution. In this work, we adopt such CTA-based sampling for our characterization purposes.

4.2 Memory-Intensive Kernels

Theoretically, as long as the available memory bandwidth is not saturated (i.e., MBU < 100%), we expect to see performance improvement as we increase the number of SMs. However, in our experimental studies we observed that the performance of memory-intensive kernels get saturated when MBU is much less than 100%. For instance, in SP, using all 32 SMs causes severe contention in last level cache while MBU is only about 60%. If we keep reducing the number of active SMs down to 11, we still observe the same IPC for SP. In other words, in memory-intensive kernels, memory bandwidth is not the only bottleneck and the running kernel might instead suffer from contention in the last level cache and/or congestion in the interconnection network while memory bandwidth is not saturated.
Based on our experimental evaluations, we consider 50% memory bandwidth utilization as the primary threshold to classify a kernel as memory-intensive. In other words, kernels with MBU above that threshold could be potentially using too many SMs. Once a kernels is recognized as memory-intensive, our feedback-driven CTA scheduler gradually reduces the number of running CTAs to find the ideal number of CTAs/SMs for that kernel. Note that, the existence of such threshold is not essential. Meaning that, we can perform our optimization scheme on all kernels. However, this approach causes useless executions of our optimization scheme which could hurt the performance of compute-intensive kernels during the search process. Therefore, this threshold only eliminates wasteful optimization processes for non-memory-intensive kernels.

4.3 Regulating Number of Active Cores

The idea behind our proposed technique is to monitor IPC and MBU statistics to determine the ideal number of active SMs accordingly. The goal is to assign minimum possible number of SMs to the running kernel without losing performance. To determine the ideal number of active SMs, we employ a feedback driven CTA scheduling approach which changes the number of running CTAs over multiple samplings. To do so, we first run the kernel when it has been allocated all the SMs. If that kernel is recognized as memory-intensive, our scheme allocates that kernel fewer number of SMs, and recollects the statistics to evaluate the impact of reconfiguration on IPC and MBU. In order to converge to the ideal point faster, in this work we used a binary search for regulating number of active SMs. The binary search takes $\log(\text{Number-of-SMs})$ steps to find the ideal answer. Our base architecture has 32 SMs; consequently, our scheme converges to the ideal number of SMs after 5 steps. In the proposed procedure, we compare the IPC of the new configuration with the IPC of the very first sampling phase that had all the SMs activated. When we compare two IPCs from two different configurations, we consider a small margin for IPC variations, meaning that as long as $|IPC_i - IPC_1| \leq \alpha$, we technically have the same performance. When we cross the saturation point (discussed in Figures 2 and 3) and step in the linear part of the performance curve, the new observed IPC will be considerably less than $IPC_1$ and our scheme accurately detects the saturation point. Figures 6 represents an optimization process for two kernels from PVC. As can be seen, search process stabilizes after 5 $(\log(\text{#SMs}))$ sampling/reconfiguration phases. In this example, PVC1 and PVC2 are assigned 13 and 24 SMs, respectively. Figure 7 on the other hand, shows the number of assigned SMs to each kernel over different invocations. In Figures 4 and 5 we observed that PVC2 exhibits similar characteristics over different executions. Consequently, our scheme assigns almost the same number of SMs to PVC2 over different executions. PVC1 however, as discussed in Figures 4 and 5, experiences widely different compute- and memory-intensive phases over time; therefore, the number of dedicated SMs to that changes from 13 to 28 SMs over different invocations.
Note that, off-line characterization and optimization techniques cannot recognize such variations over different executions. Therefore, techniques like (Hong and Kim 2010) fail in determining the optimal number of SMs especially for cases like PVC1 with unpredictable run-time behaviour.

**Pausing Technique.** CTAs, once assigned to a core, cannot be preempted, or assigned to another core (NVIDIA Corporation 2010). Therefore, during successive sampling and reconfiguration periods, we pause the SMs instead of shutting them down. Such pausing approach during the sampling phase will not cause any migration/context-switch overhead among SMs. All this process is managed by our feedback driven CTA scheduler. After determining the ideal number of active SMs (after 5 sampling and reconfiguration steps), we use a SM power-gating mechanism to turn the rest of the SMs off once they finish executing previously assigned CTAs.

**Predict When to Reactivate the SMs.** Once a kernel execution is finished, the CPU launches a new kernel on the GPU. At this point, the hardware needs to allocate all the available SMs to the new kernel. However, some of the SMs might be shut down for the previous kernel and the delay of powering them on could negatively affect performance of the newly launched kernel. To avoid this, we predict the time when a kernel will be finished. This can be implemented by hardware since it can measure the average execution time of a CTA at run-time (i.e., in terms of number of cycles). Therefore, based the number of left CTAs to schedule, and the average execution length of a CTA, we can determine when to reactivate the power-gated SMs in order to overlap the SM reactivation delay with the remaining execution time of the running kernel.

**Hardware Overhead.** Our proposed scheme monitors IPC and MBU to regulate number of active SMs in a feedback-driven fashion. To collect this information at run time, we assume that each SM has 2 counters (overall, 32*2*4 Bytes) to track the number of executed instructions as well as the number of cycles. Besides, each memory channel needs 2 counters (overall, 6*2*4 Bytes) to track the number of memory transactions and number of memory cycles. Considering all the performance counters, the proposed mechanism has an overall capacity overhead of 304 bytes.

## 5 EXPERIMENTAL RESULTS

### 5.1 Methodology

**Platform:** In order to evaluate our proposal, we used GPGPU-Sim v3.2.2 (Bakhoda et al. 2009), a publicly-available cycle-accurate GPGPU simulator. The details of the simulated configuration are listed in Table 1a. This configuration is similar to GTX480 configuration. In our experiments, we changed the number of active SMs between 1 and 32, and used 32 in our baseline. Each SM is supported by a separate 16KB L1D and L1I caches. SMs are connected to 6 memory channels. Each memory channel is coupled with a portion of L2 cache with a size of 256KB. Misses in L2 cache are sent to the memory (Rixner et al. 2000).

**Benchmarks:** Table 1b lists the applications we used in our evaluations. We consider a wide range of memory-intensive applications from various benchmark suites: CUDA SDK (NVIDIA 2011), Parboil (Stratton et al. 2012), Mars (He et al. 2008), Shoc (Danalis et al. 2010), and LonestarGPU (Burtscher et al. 2012). We classify the kernels as compute-intensive (COMP), and memory-intensive (MEM) in Table 1b. As can be seen, each of the studied applications consists of at least one memory-intensive kernel.

**Performance Metrics:** In this work, we focus on energy efficiency, thus we report three metrics. First, we report application performance in terms of normalized IPC with respect to the baseline configuration described in Table 1a. Second, we report the power consumption of the system using GPUWattch (Leng et al. 2013). In particular, we focus on dynamic power, leakage power, and DRAM power. Third, based on performance and power results, we calculate the energy consumption of the system. The results presented below includes all the runtime overheads brought by our approach.
Jadidi, Arjomand, Kandemir, and Das

(a) Baseline configuration.

<table>
<thead>
<tr>
<th>SM Config.</th>
<th>Resources / Core</th>
<th>Caches / Core</th>
<th>Warp Scheduler</th>
<th>Features</th>
<th>Interconnect</th>
<th>Memory Model</th>
<th>GDDR5 Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 Shader Cores, 1400MHz, SIMT Width=32</td>
<td>1536 Threads (48 warps, 32 threads/warp), 48KB Shared Memory, 32684 Registers</td>
<td>16KB 4-way L1D, 12KB 24-way Texture, 8KB 2-way Constant Cache, 2KB 4-way L1I</td>
<td>Greedy-then-oldest (Rogers et al. 2012)</td>
<td>Memory Coalescing, Inter-warp Merging, Immediate Post Dominator (Fung et al. 2007)</td>
<td>Crossbar, 1400MHz, 32B Channel Width</td>
<td>6 GDDR5 MCs, 2GHz, 1V, FR-FCFS, 8 DRAM-banks/MC</td>
<td></td>
</tr>
</tbody>
</table>

(b) List of GPU benchmarks: In the last column, MEM and COMP refers to memory- and compute-intensive behavior of the kernels.

<table>
<thead>
<tr>
<th>Suite</th>
<th>Application</th>
<th>Abbr.</th>
<th>Kernel Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lonestar</td>
<td>Single-Source Shortest Paths</td>
<td>SSSP</td>
<td>MEM</td>
</tr>
<tr>
<td>Lonestar</td>
<td>Breath-First Search</td>
<td>BFS</td>
<td>MEM</td>
</tr>
<tr>
<td>Lonestar</td>
<td>Survey Propagation</td>
<td>SP</td>
<td>MEM</td>
</tr>
<tr>
<td>Lonestar</td>
<td>Minimum Spanning Tree</td>
<td>MST</td>
<td>MEM-COMP</td>
</tr>
<tr>
<td>Parboil</td>
<td>Saturating Histogram</td>
<td>HIST</td>
<td>MEM-COMP</td>
</tr>
<tr>
<td>Shoc</td>
<td>2D Stencil Computation</td>
<td>stencil</td>
<td>MEM</td>
</tr>
<tr>
<td>SDK</td>
<td>MUMerGPU</td>
<td>MUM</td>
<td>MEM</td>
</tr>
<tr>
<td>SDK</td>
<td>LIBOR Monte Carlo</td>
<td>LIB</td>
<td>MEM</td>
</tr>
<tr>
<td>Mars</td>
<td>Kmeans Clustering</td>
<td>Kmean</td>
<td>MEM-COMP</td>
</tr>
<tr>
<td>Mars</td>
<td>Page View Count</td>
<td>PVC</td>
<td>MEM-COMP</td>
</tr>
<tr>
<td>Mars</td>
<td>Page View Rank</td>
<td>PVR</td>
<td>MEM-COMP</td>
</tr>
</tbody>
</table>

Table 1: Experimental setup in our evaluations.

5.2 Evaluation

In this section we analyze the impact of our proposed mechanism on the chip energy consumption, and overall system performance.

Static and Dynamic Power Consumption: Figure 8 reports the breakdown of total energy consumption in terms of static and dynamic power ratios. For each application, we have reported the energy saving gained by our proposed technique as well as the energy saving of the optimal configuration (optimal configuration is the configuration with lowest energy consumption while performance loss is kept less than 2% compared to the baseline. In order to find the optimal point, we ran the system under all 32 possible SM assignments.)

As can be seen, for most of the applications the static power contribution in overall improvement is dominant which is achieved by power-gating some of the SMs. In this category of application, regulating the number of SMs does not affect the system performance. However, since we use less number of active SMs, the power-gating can effectively reduce leakage power consumption. The energy saving gained by lowering the static power consumption is linearly dependent on the number of power gated SMs.

For some of the kernels/applications (i.e., BFS, MUM, SP, and SSSP), we also observe improvements in the dynamic power consumption. As we keep decreasing the number of active SMs, for kernels running in the saturated region, we could potentially decrease the dynamic power consumption by shortening the execution time. For instance, reducing the number of active SMs from 32 to 20 in BFS, reduces the contentions in last level cache such that the miss-rate reduces from 77% to 45% which consequently improves IPC by 25%. We observed similar impacts in MUM, SP, and SSSP. Overall, our proposed mechanism achieves up to 35% and on average about 21% energy saving, which is within 8% of the optimal saving.

Performance: Figure 9 reports the average number of SMs that our proposed scheme stabilizes at for each application. Our proposed scheme reduces the number of active SMs to as low as 10 SMs and with an average of 28 SMs. Figure 10 demonstrates the impact of our proposed technique on system performance. As can be observed, four of those applications (i.e., BFS, MUM, SP, and SSSP) experience severe performance loss in baseline configuration because of the contention in last level cache and/or congestion in the inter-connection network. For instance, by allocating the ideal number of SMs to SSP, and BFS, their last level cache hit-rate improves by 45%, and 30%, respectively. Our technique improves the performance of those four applications by up to 25%, with an average of 12%. Our proposed mechanism reduces he performance of the remaining applications 2% on average.
Figure 8: Energy saving gained by using optimal number of SMs. BL, FD, and OP represent BaseLine, our proposed Feedback-Driven, and OPtimal system configurations, receptively.

Figure 9: Average number of active SMs.

Figure 10: Normalized IPC values for different applications with respect to the baseline configuration.

Application of DVFS Techniques on Memory-Intensive Kernels: Figure 12 compares the impact of our proposed mechanism with a common DVFS scheme. We assumed that the GPGPU has 7 power-states as reported in Figure 11. A wide range of DVFS techniques are available to regulate the voltage/frequency of different resources in a computing platform. In our feedback-driven CTA scheduling approach, we adopt a SM power-gating mechanism to shut down some of the SMs. However, one can adopt of a DVFS technique to regulate voltage/frequency of the SMs in order to resolve the memory saturation problem (Leng et al. 2013). Here we discuss two major differences between power-gating and DVFS techniques: First, power-gating is more effective in reducing the leakage power compared to DVFS techniques, as reported in Figure 12. Second, although reducing the frequency of the SMs could mitigate the memory bandwidth saturation problem, it does not resolve the cache contention problem because cache contention is not a function of time but a function of the sequence of cache accesses. In other words, cache access pattern is a function of number of running threads which is modulated by our feedback-driven CTA scheduler but it is not affected by a core-side DVFS scheme. As can be seen in Figure 12, for BFS, MUM, SP, and SSSP that suffer from cache contention problem, our proposed technique considerably outperforms DVFS. Overall, our proposed technique reduces the energy consumption 21% on average while DVFS improvement is about 9% on average.
Voltage | Frequency
--- | ---
1.000 V | 2.00 GHz
0.925 V | 1.75 GHz
0.850 V | 1.50 GHz
0.775 V | 1.25 GHz
0.700 V | 1.00 GHz
0.625 V | 0.75 GHz
0.550 V | 0.50 GHz

Figure 11: List of adopted V/F states to dynamically manage SM power consumption during the memory-intensive phases.

Figure 12: Energy saving gained by different techniques. BL, and FD represent BaseLine, our proposed Feedback-Driven configurations, receptively.

6 RELATED WORK

Theoretically speaking, assigning more SMs to a highly multi-threaded application improves its performance as long as the memory bandwidth does not saturate. Huang et al. (2009) evaluated the effect of number of active SMs on energy consumption and discussed that having all the SMs activated is the most energy efficient configuration. The lack of that study is that they did not consider any memory-intensive application. In order to have a more accurate analysis, we need to consider the possible congestion in the interconnection network and the contention in last level cache caused by enormous number of memory requests (issued by huge number of concurrently running threads). In this line, Guz et al. (2009) showed that increasing the parallelism improves the performance as long as the memory access latency is not affected considerably. An orthogonal category of works (Jadidi et al. 2011, Arjomand et al. 2011, Arjomand et al. 2016) exploit large last level caches and/or accelerate the memory accesses to reduce average data access latency. Li and Martínez (2005) analytically estimated the optimal number of processors to achieve the best EDP in CMPs. In GPU domain, Hong and Kim (2010) proposed an analytical model which predicts the optimal number of SMs based on offline characterizations. Our proposed mechanism however, exploits run-time characteristics for regulating number of active cores. As shown in Section 3, many kernels go into saturation state while MBU is much less than 100%. Therefore, although Hong and Kim (2010) statically provide us estimations for the number of active cores, it does not consider potential contention in the last level cache and/or congestion in the interconnection network. On the other hand, DVFS techniques (Leng et al. 2013) can be exploited to regulate the voltage/frequency of the SMs during memory-intensive phases. Although a core-side DVFS technique can improve the energy consumption of memory-intensive kernels, as discussed in Section 5, our proposed mechanism outperforms DVFS techniques in terms of reducing static leakage power as well as resolving the last level cache contention issue.

7 CONCLUSIONS

In this paper, we proposed a feedback-driven mechanism that dynamically adjusts the number of active SMs based on the kernel demand. The proposed mechanism uses a CTA-based sampling and reconfiguration scheme to dynamically analyse the kernel and determine the ideal number of active SMs. This technique reduces the chip energy consumption up to 35% and about 21% on average over the studied memory-intensive applications, which is within 8% of the optimal saving that can be obtained from an oracle scheme.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants 1302557, 1213052, 1439021, 1302225, 1629129, 1526750, and 1629915 and a grant from Intel.
REFERENCES


Burttscher, M., R. Nasre, and K. Pingali. 2012. “A quantitative study of irregular programs on GPUs”. In IISWC.


Fermi, N. 2009. “Nvidia’s next generation cuda compute architecture”.


NVIDIA 2011. “CUDA C/C++ SDK Code Samples”.


AUTHOR BIOGRAPHIES

AMIN JADIDI received the B.Sc. degree from University of Tehran of Tehran, Iran, in 2009, and the M.Sc. degree from the Sharif University of Technology, Tehran, Iran, in 2011, all in electrical and computer engineering. He currently is a PhD candidate in Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA. His current research interests include multicore and manycore architectures, GPGPU architectures, and memory systems. His email address is axj945@cse.psu.edu.

MOHAMMAD ARJOMAND (S’09) received the B.Sc. degree from the Shahid Bahonar University of Kerman, Kerman, Iran, in 2006, and the M.Sc. and Ph.D. degrees from the Sharif University of Technology, Tehran, Iran, in 2008 and 2014, respectively, all in computer engineering. He currently holds a post-doctoral position with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA. His current research interests include multicore and manycore architectures, memory systems, storage systems, and power-aware architectures. Mr. Arjomand is a Student Member of the Association for Computing Machinery. His email address is mxa51@cse.psu.edu.

MAHMUT TAYLAN KANDEMIR is a professor in the Computer Science and Engineering Department at the Pennsylvania State University. He is a member of the Microsystems Design Lab. Dr. Kandemir’s research interests are in optimizing compilers, runtime systems, embedded systems, I/O and high performance storage, and power-aware computing. He is the author of more than 80 journal publications and over 300 conference/workshop papers in these areas. He is a recipient of NSF Career Award and the Penn State Engineering Society Outstanding Research Award. He currently serves as the Graduate Coordinator of the Computer Science and Engineering Department at Penn State. His email address is kandemir@cse.psu.edu.

CHITA R. DAS is a distinguished professor in the Department of Computer Science and Engineering at Pennsylvania State University. He currently serves as the Dean of the Computer Science Department at Penn State. His main areas of interest include parallel and distributed computer architectures, multi-core architectures, mobile computing, performance evaluation, and fault-tolerant computing. He has published more than 200 papers in the above areas, has served on many program committees, and editorial boards. He has a PhD in computer science from the University of Louisiana, Lafayette. He’s a fellow of IEEE. His email address is das@cse.psu.edu.