Energy-Efficient Scheduling for Memory-Intensive GPGPU Workloads

Seokwoo Song, Minseok Lee, John Kim
KAIST
Daejeon, Korea
{sukwoo24, lms135, jjk12}@kaist.ac.kr

Woong Seo, Yeongon Cho, Soojung Ryu
Samsung Electronics
Giheung, Korea
{brad.seo, yeongon.cho, soojung.ryu}@samsung.com

***Abstract*—High performance for a GPGPU workload is obtained by maximizing parallelism and fully utilizing the available resources. However, this is not necessarily energy efficient, especially for memory-intensive GPGPU workloads. In this work, we propose *Throttle* CTA (cooperative-thread array) Scheduling (TCS) where we leverage two type of throttling – throttling the number of actives cores and throttling of warp execution in the cores – to improve energy-efficiency for memory-intensive GPGPU workloads. The algorithm requires the global CTA or thread block scheduler to reduce the number of cores with assigned thread blocks while leveraging the local warp scheduler to throttle memory requests for some of the cores to further reduce power consumption. The proposed TCS scheduling does not require off-line analysis but can be done dynamically during execution. Instead of relying on conventional metrics such as miss-per-kilo-instruction (MPKI), we leverage the memory access latency metric to determine the memory intensity of the workloads. Our evaluations show that TCS reduces energy by up to 48% (38% on average) across different memory-intensive workload while having very little impact on performance for compute-intensive workloads.**

I. Introduction

Because of the significant computing capability, accelerators such as GPGPU are becoming widely used for different workloads [15]. These architectures allow for thousands of threads to be executed in parallel through programming models such as CUDA or OpenCL. The programming models results in a hierarchy of threads – a group of threads forming a warp or a wavefront, and a collection of warps forming a thread block, referred to as CTA (cooperative thread array) or a working group. As a result, there are two levels of scheduling within a GPGPU – a thread block or CTA scheduler to assign the CTAs to each core and a warp scheduler to determine which warp is executed within a core. To increase performance of these parallel architectures, alternative warp schedulers [19], [14], [17], [7] have been proposed to improve resource utilization and performance. Recently, an alternative CTA scheduler has been proposed to reduce the number of CTAs per core to maximize performance by reducing the memory system contention [8], [11]. In this work, we propose an combined CTA-warp scheduling that throttles both CTA and warp execution to improve energy-efficiency of memory-intensive workloads. We observe that the increasing the number of cores does not continuously improve the performance of memory-intensive workloads and thus, dynamically determine the optimal number of cores and power-gate the remaining cores.

This work was supported in part by the IT R&D program of MSIP/KEIT (10041313, UX-oriented Mobile SW Platform) and in part by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the MSIP (NRF-2011-0015039).

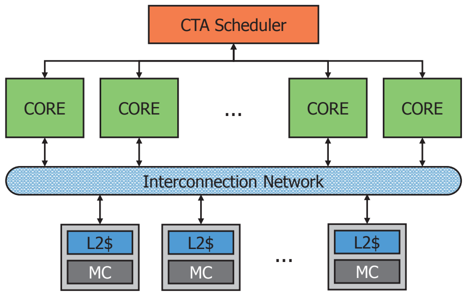

Fig. 1: Baseline GPGPU Architecture Bloack Diagram.

The memory system can be the bottleneck in multi-threaded workloads since multiple threads contend for finite memory bandwidth. Prior work has investigated thread throttling to improve performance [4], [19] in multithreaded architectures. In the GPGPU architectures that we consider, the large number of threads increases this problem for memory-intensive workloads. Thus, the challenge is in properly determining the optimal number of cores to power-gate such that there is minimal impact on performance while reducing the overall energy. It is also critical that the cores are not power-gated for compute-intensive workloads which have a significant amount of data parallelism. In this work, we propose throttle CTA scheduling (TCS) – an online CTA [1] scheduling mechanism which combines dynamically detecting memory intensive workloads with throttling to determine the optimal number of cores. The TCS is partitioned into two phases – MONITOR and REDUCE phase and this process is repeated for each kernel within each workload.

II. Motivation

In this section, we first describe the simulation methodology used in our evaluation and show the impact of static power consumed in GPGPUs. We then show how the performance changes as the number of cores increases and its impact on the energy consumption of GPGPUs.

A. Methodology

The GPGPU that we model consist of 28 cores (or streaming multiprocessors (SMs)) connected to 8 memory controllers. A high-level block diagram of our baseline architecture is

[1]In this work, we use the term CTA and thread block interchangeably.

TABLE I: Baseline Configuration

# Compute Units	28
Warp Size	32
Resources / Core	MAX. 1536 Threads, 32768 Registers 32 MSHRs
Core / ICNT / Memory Clock	1400 MHz / 1400 MHz / 1848 MHz
Shared Memory	48KB
Constant Cache	8KB
Texture Cache	32KB, 16-way, 64B line
L1 Data Cache	32KB, 8-way, LRU, 128B line
L2 Cache	128KB/Memory Channel, 8-way, LRU, 256B line
Interconnect	Crossbar, 32B channel width
DRAM Model	FR-FCFS memory scheduling, 8MCs, 16 DRAM banks/MC
GDDR5 Timing	t_{CL}=12, t_{RP}=12, t_{RC}=40 t_{RAS}=28, t_{RCD}=12, t_{RRD}=6

TABLE II: GPGPU Workload Description.

Name	Abbr.	# of kernels	Grid Dim.	Type
MUMmerGPU[2]	MUM	1	(196,1,1)	Mem
Lattrice-Boltzmann Method[18]	LBM	1	(100,130,1)	Mem
Kmeans[3]	KMN	1	(1936,1,1)	Mem
Breadth First Search[3]	BFS	24	(1954,1,1)	Mem
Page View Count[5]	PVC	29	(1,1,1) (128,1,1) (256,1,1) (1953,1,1) (3907,1,1)	Mem
Separable Convolution Filter[16]	CONV	34	(24,768,1)	Com
Black-Schole option pricing[16]	BLK	1	(480,1,1)	Com
Back Propagation[3]	BP	2	(1,4096,1)	Com
Binomial Options[16]	BINO	1	(512,1,1)	Com
DXT Compression[16]	DXTC	2	(16384,1,1)	Com

shown in Figure 1 and is similar to prior work on GPGPUs [17], [1]. Each core has its own private L1 data cache, texture cache, and shared memory while each memory controller has a slice of the L2 cache that is shared. The cores and the memory controllers are interconnected through a crossbar network. We use a detailed GPGPU simulator (GPGPU-sim v3) [2] in our evaluation and the different parameters are described in Table I. The simulator was modified to implement the different scheduling that we evaluate in this work. We considered a wide range of GPGPU CUDA workloads, including applications from Rodinia [3], Parboil [18], MapReduce [5] NVIDIA SDK [16], and workloads from GPGPU-sim [2] – and include both memory-intensive and computer-intensive workloads (Table I). All of the workloads are run until completion, except for workloads from [5] which are simulated for 2 billion instructions. Power is estimated using GPUWattch [12] – a tool which has been shown to accurately match actual hardware – and we assume 45nm technology. We assume a warp size of 32 threads and each core or SM (stream multiprocessor) can have a maximum of 1536 threads, similar to the NVIDIA GTX480 architecture. Within the GPGPU architecture, there are two schedulers – a warp scheduler within each core that determines which warp is issued while the CTA scheduler (referred to as GigaThread scheduler using NVIDIA terminology) determines which CTA is assigned to which cores. In this work, we leverage both the CTA-warp scheduler to implement *throttle*-based scheduling to power-gate cores and improve energy-efficiency.

B. Static Power in GPUs

As technology continues to shrink, it is well-known that static power consumed in modern CPUs cannot be ignored [9].

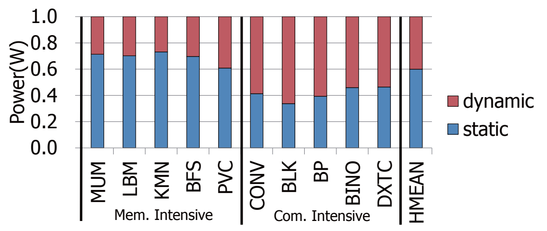

Fig. 2: Average static power consumption ratio of GPU for the different workloads.

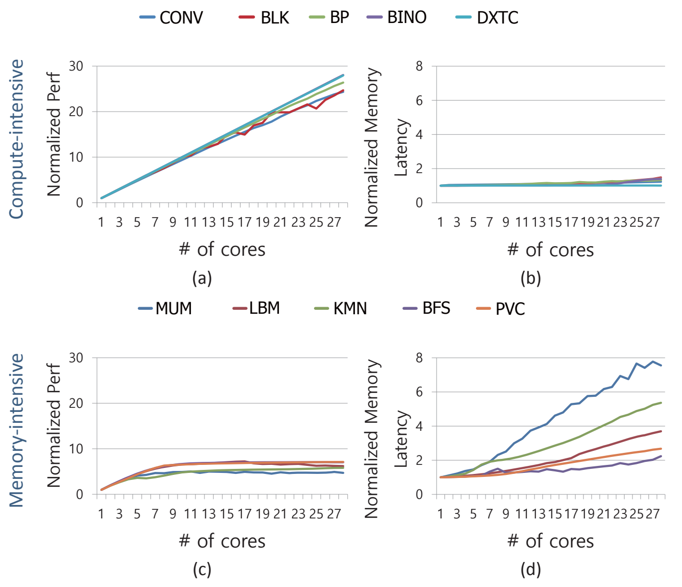

Fig. 3: (a,c) Normalized performance and (b,d) normalized memory latency values as the number of cores is varied. (a,b) are the results from compute-intensive workloads while (c,d) are from memory-intensive workloads. Performance is measured in terms of IPC and thus, higher is better.

Because of the relative large die size of GPUs, static power also represents a significant portion of the total power in GPUs. In Figure 2, the breakdown of static and dynamic power across the different workloads is shown using the configuration described earlier in Table I. The power estimate includes all on-chip components, including the cores, on-chip network, shared memory, and all of the cache structures. Results show that static power can represent over 60% of the total power for some workloads and on average, approximately 40% of the total power. As a result, by power-gating cores that do not significantly contribute to overall performance, there is opportunity to save overall power in GPUs and improve energy-efficiency. Modern CPUs often have support to power-gate cores individually. To the best of our knowledge, current modern GPUs do not have the capability to power-gate individual cores but since power is also a concern in GPUs, we expect GPUs to support this in the future. Recent work [13] has shown that per-core power gating overhead results in a leakage power of roughly a milliwatt and thus, overhead to support per-core power gating is relatively small.

C. Impact of the Number of Cores

Figure 3 shows the performance of different workloads as the number of cores increases. For some of the workloads (*compute-intensive* workloads), performance continues to increase linearly as the number of cores increases (Figure 3(a)).

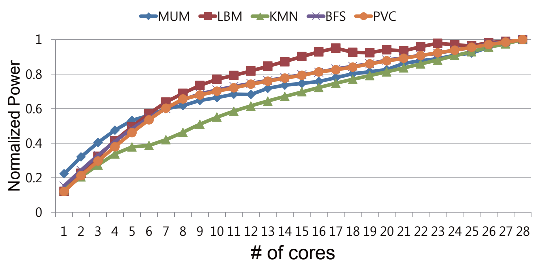

Fig. 4: Power consumption for memory intensive workloads.

This is expected for many data-parallel workloads which improve in performance as additional computing resources (e.g., cores) are added. However, for other workloads (Figure 3(c)), the performance initially increases as the number of cores increases, but at some point, the performance saturates and there is little or no improvement in performance. These workloads are often *memory-intensive* workloads with the memory-system being the bottleneck. As the number of cores increases, the memory system performance (e.g., memory latency) can actually degrade and many of the cores are effectively stalled, waiting for data to return from memory.

The average memory access latency is shown in Figure 3(d) for the memory-intensive workloads – with the latency normalized to the latency value when the number of cores is one. As the number of cores increase, the average memory latency can increase significantly – by up to 7.5× for some workloads and on average, approximately a 4× increase in latency when all of the cores are fully utilized. In comparison, for compute-intensive workloads, the memory latency can also increase but the increase is significantly smaller (only up to 1.5×) while for some workloads, there is no increase in memory latency. In this work, we focus on identifying memory-intensive workloads and identify the number of cores to utilize such that the remaining cores can be power-gated – while ensuring that we do not unnecessarily power-gate cores for compute-intensive workloads.

Figure 4 plots the total power consumed for memory-intensive workloads. Even though the performance saturates around 5-10 cores for these workloads, the power consumption continues to increase since the additional cores results in increased static power. As a result, fully utilizing all of the cores does not necessarily result in the most energy-efficient system. In this work, we optimize for the EDP (energy-delay-product) metric and the *optimal* number of cores is defined as the number of cores with minimal EDP. The optimal number of cores is determined statically by varying the number of cores to determine the number of cores with the lowest EDP. The proposed TCS attempts to determine this optimal number of cores dynamically and we show that TCS can nearly match it.

III. Throttle-CTA Scheduling (TCS)

A. Overview

The goal of TCS is to identify the optimal number of cores to achieve energy-efficiency – i.e., find the number of cores to power-gate for memory-intensive workloads. An example of TCS is shown in Figure 5 for a 6-core system. Initially, the CTA scheduler assigns CTAs to all of the cores (Figure 5(a)). However, if a core is determined to be memory intensive, then the number of active core is reduced by 1. In Figure 5(b),

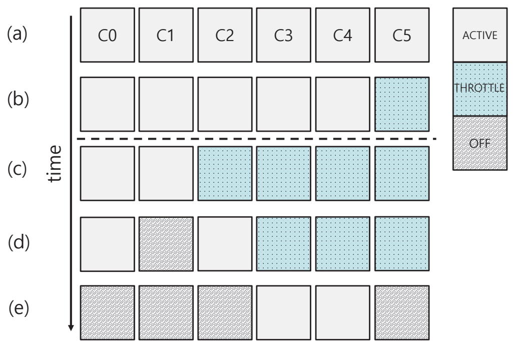

Fig. 5: Throttle CTA scheduling(TCS) example for a 6-core(CO-C5) system.

Algorithm 1 MONITOR Algorithm within TCS Scheduling

```
stall_cycle : number of cycles stalled from the memory unit (per core)
thr: threshold to determine whether memory system is congested (per core)
mon_cycle: monitoring period in cycle (per core) 8192 cycles
AMAT: average memory latency (per core)
$C_{mem}$: the number of active cores that are memory-bound
$C_{active}$: the number of active cores
$C_{throttle}$: the number of cores throttled

procedure MONITOR
    for each (mon_cycle) do
        for (i=1; i $<=$ $C_{active}$; i++) do
            if stall_cycle(i) $>$ mon_cycle/2 and AMAT(i) $>$ thr(i) then
                $C_{mem}$++
                stall_cycle(i) = 0
            end if
        end for
        if $C_{mem}$ $>=$ $C_{active}$ / 2 then
            select a CORE$_j$ that is in ACTIVE state.
            CORE$_j$ switch from ACTIVE to THROTTLE state.
            $C_{throttle}$++
            $C_{active}$−−
            $C_{mem}$ = 0
        end if
    end for
end procedure
```

one of the cores (C5) is switched to THROTTLE state and the number of active cores is reduced by 1. This process continues until the optimal number of active cores is reached – in this example, we assume the optimal number of cores is two (Figure 5(c)). Once the optimal number of cores is reached, the ACTIVE cores run until completion – i.e., *all* CTAs assigned to the core are completed. In Figure 5(d), we assumed that C1 is the first core to finish. Instead of assigning additional CTAs to this core, this core is power-gated and one of the cores in the THROTTLE state (C2) is switched to the ACTIVE state. This process is repeated until the optimal number of active cores is reached (Figure 5(e)) and the remaining cores are power-gated. The details of TCS are described in the following section.

B. Algorithm Description

The high-level description of the proposed Throttle CTA scheduling (TCS) is described in Algorithm 1. Each core can be in one of three states : ACTIVE, THROTTLE, or OFF states.

- **ACTIVE**: The core has active CTAs and is currently executing the CTAs [2]
- **THROTTLE**: The core has active CTAs assigned to the cores but the CTA scheduler determines to throttle the memory requests.
- **OFF**: The core is power-gated and no CTAs are assigned to that particular core for the given kernel.

In addition, the TCS is partitioned into two phases – MONITOR phase and the REDUCE phase. In the MONITOR phase, the cores are monitored to determine the optimal number of cores to utilize in order to minimize energy. In the REDUCE phase, the cores are power-gated appropriately to reduce the number of active cores.

The goal of the MONITOR phase is to classify memory-intensive workloads and gradually decrement the number of cores to reduce memory contention in the system. Within each core, we monitor the number of *consecutive* cycles when the core is stalled from the memory unit ($stall\_cycle$) and compare it with a threshold value to determine if the workload is memory intensive. This stall often occurs because memory instructions cannot be issued or the memory requests cannot be injected into the network – i.e., this frequently occurs when the Miss Status Holding Register (MSHR) is fully occupied. In addition to this condition, we compare the average memory latency (AMAT) with a threshold value (thr) to determine if the workload is memory-intensive. Most prior work have commonly leveraged MPKI (miss-per-kilo-instructions) to determine memory-intensive workloads. However, for the GPGPU workloads that we evaluate, MPKI does not provide an accurate representation of the memory intensity because of the different memory access characteristics (e.g., a warp can result in up to 32 "misses", a higher row locality memory access pattern, etc.). As a result, we use the AMAT parameter to determine memory-intensive workloads. The thr parameter is empirically determined and is based on the approximate contention-free memory-access latency that includes the on-chip network latency.

Once a core is identified to be memory-intensive, that particular core is switched to THROTTLE state and no additional CTAs or thread blocks are assigned to that particular core in the MONITOR phase. The MONITOR phase continues until the first CTA completes, or 10,000 cycles, which ever occurs later. We used the unit of a CTA since the workload is partitioned into units of CTA within a kernel and one CTA is sufficient to understand the characteristics of a given kernel. However, we empirically added a threshold of 10000 cycles since for some of the workloads, the CTAs finished very early and thus, a single CTA was not sufficient to properly estimate the memory intensity of the workload.

Once the MONITOR phase is completed, the number of active cores is identical to the optimal number of cores determined by the algorithm while the remaining cores are in a throttled state. In the REDUCE phase, the object is to power gate $C_{throttle}$ number of cores such that only $C_{optimal}$ found in the MONITOR phase are used for the remaining execution of the kernel. The REDUCE phase is described in Algorithm 2

[2]The core can be stalled for various reasons but we classify the core as ACTIVE for the purpose of TCS.

Algorithm 2 REDUCE Algorithm within TCS

```
C$_{throttle}$: number of cores in THROTTLE state after MONITOR phase

procedure REDUCE
    while C$_{throttle} > 0$ do
        if CORE$_i$ finished then
            power gate CORE$_i$

            select a CORE$_j$ that is in C$_{throttle}$ state.
            CORE$_j$ switch from THROTTLE to ACTIVE state.
            C$_{throttle}$−−
        end if
    end while
end procedure
```

– as each active core finishes, that particular core is power-gated while another core is switched from the THROTTLE state to the ACTIVE state. This process repeats until there are no cores in the THROTTLE phase. At the end of a kernel, all of cores are switches back to ACTIVE states and the MONITOR-REDUCE phase is repeated. As we show in Section IV, some workloads have both memory-intensive and compute-intensive kernels and thus, this is necessary.

Although not shown in either Algorithm 1 or Algorithm 2, TCS requires the support of the warp scheduler to ensure that cores in the THROTTLE state have their memory request throttled. Once a core enters the THROTTLE state, the warp scheduler continues to issue warps for execution until a warp needs to execute a memory instruction. These warps are stalled (e.g., not scheduled) to throttle requests into the memory system; once all warps are stalled, the core is effectively in the THROTTLE state. As we shown in Section IV, this throttling help to improve the energy efficiency of TCS.

C. Half TCS (hTCS)

The current TCS assumes that all of the cores are allocated with CTAs in the beginning of execution and as necessary, reduce the number of active cores. However, this does introduce overhead in finding the optimal number of cores to reach energy-efficiency. As a result, we also evaluate an *half* TCS (hTCS) where instead of allocating CTAs to all of the cores, only half of the cores are initially allocated CTAs initially. For memory intensive workloads, if necessary, the number of cores will be further power-gated or for other workloads that require more than N/2 cores (or compute-intensive workloads), more cores will be activated as necessary. This also introduces trade-off between energy-efficiency and performance – for memory-intensive workloads, hTCS provides an opportunity for more energy savings while for compute-intensive workloads, it can result in performance degradation.

IV. Evaluation

Using the experimental setup described earlier in Section II-A, we evaluated the impact of Throttle CTA Scheduling (TCS) on performance (IPC) and energy. The results are normalized to the baseline when all of the cores fully utilized in Figure 6. For memory-intensive workloads, the average performance degradation is approximately 6% while the average energy reduction is approximately 38% for TCS. For some of the workloads (such as LBM and BFS), it is interesting to note that performance actually improved with TCS – 6% for LBM and, 5% for BFS. With the relatively small L2 cache

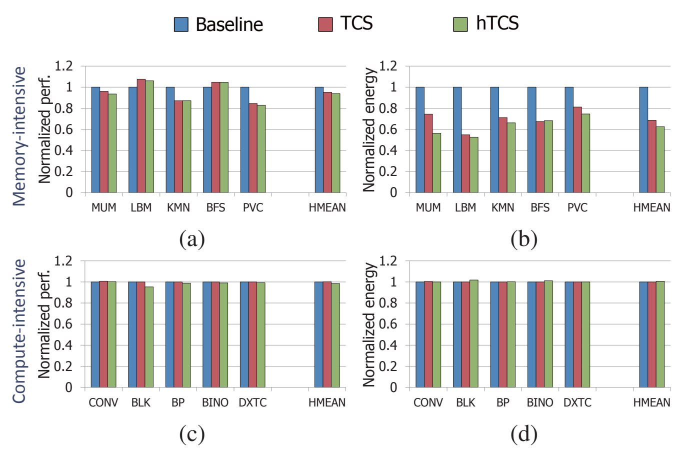

Fig. 6: Evaluation of TCS and hTCS for (a,c) performance and (b,d) energy for (a,b) memory-intensive workloads and (c,d) compute-intensive workloads.

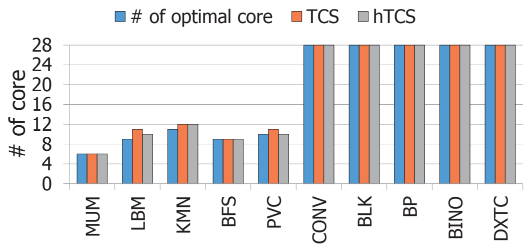

Fig. 7: Comparison of the number of optimal cores to minimize EDP.

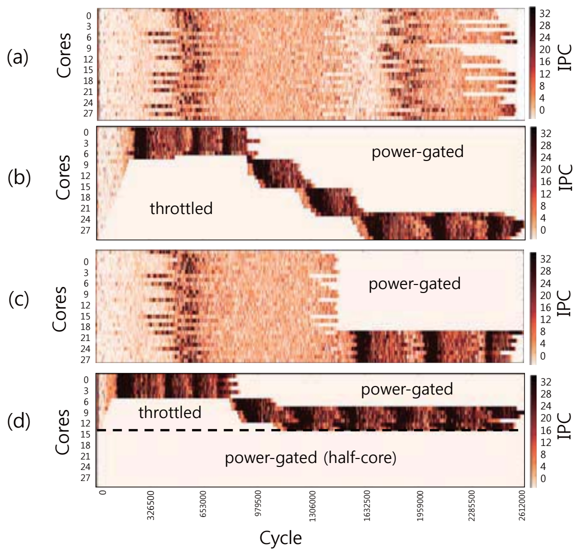

Fig. 8: Performance (IPC) with (a) baseline and maximum number of cores, (b) TCS with throttling, (c) TCS without throttling and (d) hTCS with throttling.

size in modern GPUs, the smaller number of cores increase the cache utilization as the L2 hit rate increased by 35% (7%) for LBM (BFS) and improved overall performance. For the compute-intensive workloads, there is almost no degradation of performance with TCS.

By initially allocating only half of the cores with CTAs, hTCS can further reduce energy by up to 18% (6% on average) by power-gating the cores earlier – while having minimal impact on overall performance. For compute-intensive workloads, hTCS reduces overall performance by 1.5% on average (and up to 5% for BLK) since half the cores are inactive in the MONITOR phase. The impact was more significant for BLK because of the relatively short execution time. For most of the workloads, the TCS algorithm was almost able to find an optimal number of cores. as shown in Figure 7 where TCS is compared with the optimal number of cores through static analysis, as described earlier in Section II. For the compute-intensive workloads, TCS did not reduce the number of cores while for memory-intensive workloads, TCS was within 6% the optimal number of cores from static analysis. Our on-line analysis provided significantly higher accuracy than the model propose by Hong and Kim [6], which is discussed further in Section V.

A detailed analysis of TCS and core throttling is shown in Figure 8 with Aerialvision [1] – comparing the baseline architecture with the maximum number of cores utilized (Figure 8(a)) to TCS where some cores are power-gated (Figure 8(b,c,d)). The y-axis represents the different cores and the x-axis is the time (cycles) and the figure plots the performance (IPC) of each core. The darker shading shows cores with higher utilization (e.g., higher IPC) while lighter shading corresponds to lower IPC. Figure 8(a) shows the baseline configuration and the various regions of the plot with low IPC correspond to cores which are mostly stalled and waiting for data from the memory. In comparison, Figure 8(b) shows TCS and how the activity of the cores change. Once the optimal number of cores is determined to be 6, the first 6 cores are executing. The remaining cores are throttled from injecting any additional requests into the network and effectively stalls the cores – hence the plots shows the remaining 22 cores to have zero IPC. When some of the cores finish executing, additional CTAs are not assigned to these cores but instead, they are power-gated and then, the throttled cores are unthrottled as they begin injecting requests into the network. [3] This process continues until only 6 remaining cores are active.

For comparison, Figure 8(c) shows TCS without throttling. Without throttling the cores, there is very little impact on overall performance, as the performance degradation compared to TCS is less than 2%. However, not using throttling reduces the energy/power benefits of TCS since the time when the cores can be power-gated is delayed. With all cores continuing to inject packets into the network (and memory), all of the cores are effectively delayed. As a result, instead of having some cores finish earlier, all of the cores that would be power-gated are delayed, reducing the energy benefit. Thus, without throttling, there is still some energy savings over the baseline but compared with TCS with throttling, the amount of energy is increased by up to 15%. Figure 8(d) shows *half* TCS (hTCS) with throttling. TCS only checks whether MAX/2 core is sufficient or not during the first monitoring cycle. If a memory-bound situation is detected, TCS starts to decrease the number of cores from MAX/2 core.

The results in Figure 8 are shown for a workload with just one kernel. However, workloads often have multiple kernels and Figure 9 shows the impact of TCS across a workload with multiple kernels. The result is shown for the BFS workload consisting of 24 kernels which includes both compute-intensive and memory-intensive kernels. Most of the compute-intensive workloads are very short and thus, no power-gating is applied.

[3]Because of the scale of the plots, it appears as if the next set of 6 cores are only executed when the first 6 cores finish executing. However, the cores are executed one at a time.

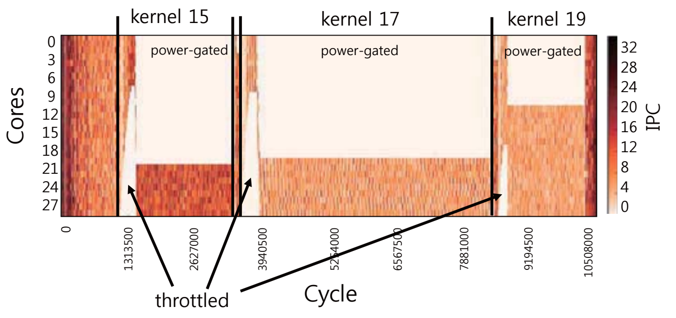

Fig. 9: Impact of TCS across the different kernels within a benchmark. The results are shown for BFS with TCS and power-gating.

However, for memory-intensive kernels 15, 17, and 19 which are highlighted in the figure, TCS reduces the number of active cores for each kernel differently – i.e., 8,9, and 17 active cores for kernels 15,17, and 19, respectively. Thus, by monitoring the behavior of each kernel separately, TCS is able to provide a different number of optimal cores for different kernels.

V. Related Work

Several prior work have also noticed that increasing the number of threads does not necessarily increase performance. However, these prior work have focused on conventional multi-threaded architectures and are not necessarily appropriate for massively multithreaded architectures, such as the GPGPU accelerator architectures. Guz et al. [4] described the *performance valley* and showed that although increasing threads improves performance, too many threads can degrade performance because of resource contention as explained earlier. Similar to [4], we also note that increasing the number of concurrent threads does not necessarily improve performance and exploit it to improve energy efficiency. Suleman et al. [19] proposed a dynamic method to find the optimal number of threads for multi-threaded workload on CMPs through feedback system during run-time at the beginning of loop iterations. In this work, we do not focus on loop iterations but focus on the behavior of an entire CTA to determine the optimal number of cores. Prior work [8], [11] also showed that more threads (or thread blocks) per core can actually degrade performance in a GPGPU and proposed alternative thread block scheduling to reduce the number of thread blocks per core. Our work is orthogonal to their work since we reduce the number of cores utilized. It remains to be seen if both approaches can be combined to further improve energy efficiency. Lee et al. [10] describes how to scale the voltage and frequency of GPUs to improve throughput in a power-constrained GPU. Their work focuses on optimizing performance/throughput for a given power constrained while we attempt to reduce power and energy for memory-intensive workloads. Hong and Kim [6] proposed an analytical power model to find optimal of cores for highest performance per watt – similar to our work. However, their work is limited to statically attempting to find the optimal cores and cannot predict runtime characteristics. For example, for MUM workload, the optimal number of cores was 6 cores with TCS while the energy efficiency number of cores with [6] was 28 since their model did not incorporate the impact of texture cache which is utilized in the MUM workload.

VI. Conclusions

In this work, we showed that always maximally utilizing all the cores is not necessarily energy-efficient for memory intensive workloads in the GPGPU architecture as the memory system becomes a bottleneck. To overcome this limitation, we proposed Throttle CTA Scheduling (TCS) that reduces the number of active cores (and throttles the individual cores) to improve energy-efficiency by power-gating the remaining cores. By leveraging the ratio of stall cycles from the memory unit (compared with the active cycles) – as well as the change in average memory latency, we gradually determine the number of cores to power-gate. In combination with CTA scheduling, we leverage the warp scheduler to throttle core requests into the memory system while adjusting for the number of cores. Our evaluations show that TCS can reduce energy by 48% (38% on average) across different memory-intensive workload while having a minimal impact on performance for compute-intensive workloads. In this work, we focused on achieving energy efficiency when executing only one kernel. However, current GPU hardware can support multiple kernels simultaneously and it remains to be seen how TCS can be adapted for multiple kernels. For example, the current TCS approach is proposed to determine the energy-efficient number of cores but instead of power-gating some of the cores, the remaining cores could be utilized by another kernel.

References

[1] A. Ariel *et al.*, “Visualizing complex dynamics in many-core accelerator architectures,” in *ISPASS*, 2010.

[2] A. Bakhoda *et al.*, “Analyzing cuda workloads using a detailed gpu simulator,” in *ISPASS*, 2009.

[3] S. Che *et al.*, “Rodinia: A benchmark suite for heterogeneous computing,” in *IISWC*, 2009.

[4] Z. Guz *et al.*, “Many-core vs. many-thread machines: Stay away from the valley,” *IEEE Comput. Archit. Lett.*, vol. 8, no. 1, 2009.

[5] B. He *et al.*, “Mars: a mapreduce framework on graphics processors,” in *PACT*, 2008.

[6] S. Hong *et al.*, “An integrated gpu power and performance model,” in *ISCA*, 2010.

[7] A. Jog *et al.*, “Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance,” in *ASPLOS*, 2013.

[8] O. Kayiran *et al.*, “Neither more nor less: Optimizing thread-level parallelism for gpgpus,” in *PACT*, 2013.

[9] N. Kim *et al.*, “Leakage current: Moore’s law meets static power,” *Computer*, vol. 36, no. 12, 2003.

[10] J. Lee *et al.*, “Improving throughput of power-constrained gpus using dynamic voltage/frequency and core scaling,” in *PACT*, 2011.

[11] M. Lee *et al.*, “Improving gpgpu resource utilization through alternative thread block scheduling,” in *HPCA*, 2014.

[12] J. Leng *et al.*, “Gpuwattch: enabling energy optimizations in gpgpus,” in *ISCA*, 2013.

[13] J. Leverich *et al.*, “Power management of datacenter workloads using per-core power gating,” *IEEE Comput. Archit. Lett.*, vol. 8, no. 2, pp. 48–51, Jul. 2009.

[14] V. Narasiman *et al.*, “Improving gpu performance via large warps and two-level warp scheduling,” in *MICRO*, 2011.

[15] J. Nickolls *et al.*, “The gpu computing era,” *MICRO, IEEE*, 2010.

[16] NVIDIA, “Cuda c/c++ sdk code samples,” 2011.

[17] T. Rogers *et al.*, “Cache-conscious wavefront scheduling,” in *MICRO*, 2012.

[18] J. Stratton *et al.*, “Parboil: A revised benchmark suite for scientific and commercial throughput computing,” In IMPACT Technical Report. IMPACT-12-01, 2012.

[19] M. A. Suleman *et al.*, “Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps,” in *ASPLOS*, 2008.