Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling

Minseok Lee, Seokwoo Song,
Joosik Moon, John Kim
KAIST
Daejeon, Korea
{lms135, sukwoo24, jsmoon78, jjk12}@kaist.ac.kr

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

Abstract

High performance in GPGPU workloads is obtained by maximizing parallelism and fully utilizing the available resources. The thousands of threads are assigned to each core in units of CTA (Cooperative Thread Arrays) or thread blocks – with each thread block consisting of multiple warps or wavefronts. The scheduling of the threads can have significant impact on overall performance. In this work, explore alternative thread block or CTA scheduling; in particular, we exploit the interaction between the thread block scheduler and the warp scheduler to improve performance. We explore two aspects of thread block scheduling – 1) LCS (lazy CTA scheduling) which restricts the maximum number of thread blocks allocated to each core, and 2) BCS (block CTA scheduling) where consecutive thread blocks are assigned to the same core. For LCS, we leverage a greedy warp scheduler to help determine the optimal number of thread blocks by only measuring the number of instructions issued while for BCS, we propose an alternative warp scheduler that is aware of the "block" of CTAs allocated to a core. With LCS and the observation that maximum number of CTAs does not necessary maximize performance, we also propose mixed *concurrent kernel execution that enables multiple kernels to be allocated to the same core to maximize resource utilization and improve overall performance.*

1. Introduction

GPGPUs are becoming widely used for different workloads because of its significant computing capability [21]. These architectures allow for thousands of threads to be executed in parallel to exploit large amount of computation capability. With programming models such as CUDA [24, 10] or OpenCL [19], GPGPUs are often programmed through a hierarchy of threads. A collection of threads are grouped to form a warp or a wavefront and the warps are combined to create a CTA (cooperative thread array) or a thread block. [1] All threads within a CTA are executed on the same core and the threads in a warp are often executed together. As a result, there are two level of schedulers within a GPGPU – a warp (or a wavefront) scheduler to determine which warp is executed and a thread block or CTA scheduler to assign CTAs to cores. To increase performance, there has been some recent work on different warp schedulers [20, 29, 15]. They improve resource utilization and/or improve hiding the impact of long memory latency operations. However, to the best our knowledge, very few work have investigated the impact of CTA or thread block scheduling on overall performance. In this work, we explore the impact of thread block scheduling on overall performance as well as the interaction between the thread block and warp scheduling.

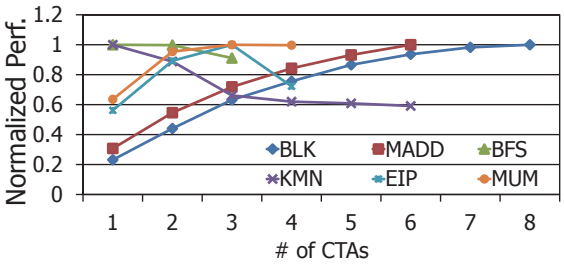

Figure 1: Performance of different workloads as the maximum number of CTAs allocated to each core is varied on the NVIDIA Tesla M2050 hardware.

To evaluate the impact of alternative CTA scheduling in real hardware, we vary the number of CTAs on a NVIDIA Tesla M2050 GPU and plot the performance for different workloads in Figure 1. By default, the current CTA scheduler in hardware assigns the *maximum* number of CTAs to each core. The maximum number of CTAs depends on the resources used by each thread and the upper limit is determined the architecture (e.g., 8 CTAs in the Tesla architecture that we evaluate). In order to approximate varying the number of CTAs on current hardware, we increase the usage of shared memory by modifying the source code. The workloads shown in Figure 1 do not use shared memory thus we add a simple write command to shared memory. For some of the workloads (e.g., MADD, BLK), the results are intuitive as performance mostly increases as the number of CTAs assigned to each core increases. However, for some workloads (e.g., EIP, KMN, BFS), the performance actually *degrades* as the number of CTAs assigned to a core continues to increase. For such workloads, assigning the maximum number of CTAs does not necessarily result in maximum performance as additional CTAs degrade performance by likely creating

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

978-1-4799-3097-5/14/$31.00 ©2014 IEEE

resource contention.

Prior work [29, 16] have also made similar observations that maximizing the number of threads executed concurrently does not necessarily maximize performance. Cache Conscious Wavefront Scheduling (CCWS) [29] proposes a warp scheduler that tracks L1 cache accesses to throttle the number of warps scheduled. Dynamic CTA scheduling (DYNCTA) [16] attempts to allocate the optimal number of CTAs to each core based on the application characteristics. However, these approaches require detailed monitoring of the workload behaviors for the entire kernel execution and based on some empirical thresholds, the number of warps (or CTAs) scheduled is adjusted. In addition, the value of the thresholds has a significant impact on overall performance and the same set of thresholds is not likely to be optimal across all workloads.

In this work, we leverage the observation that the execution of threads on accelerators such as GPGPUs is impacted by both the warp scheduler and the thread block scheduler. As a result, we propose a holistic approach of considering both schedulers to improve the efficiency in GPGPU architecture. For workloads where the maximum number of CTAs does not maximize performance, we leverage a greedy warp scheduler [29] to propose a *lazy* CTA scheduling (LCS) where the maximum number of CTAs allocated to each core is reduced to avoid resource contention and performance degradation. In addition, to exploit inter-CTA locality, we propose *block* CTA scheduling (in conjunction with an appropriate block-aware warp scheduling) to improve performance and efficiency. Our approach of alternative thread blocking also provides additional opportunity to improve efficiency (and performance) when the maximum number of threads are not assigned to each core. We propose *mixed* concurrent kernel execution (mCKE) where multiple kernels are scheduled on the same core to improve resource utilization and improve overall performance.

In particular, the contributions of this work include the following.

- We characterize different workload behavior as the number of thread blocks (or CTAs) is varied and analyze the impact on overall performance.
- We exploit the interaction between the warp scheduler and the thread block scheduler to improve the overall efficiency of the GPGPU system. By leveraging a greedy warp scheduler [29] and characteristics of a thread block, we propose Lazy CTA Scheduling (LCS) to reduce the number of thread blocks allocated to each cores and avoid performance degradation.
- To exploit the inter-CTA cache locality, we propose block CTA scheduling (BCS) where consecutive CTAs are assigned to the same cores. To fully exploit such benefits, we propose a CTA-aware greedy warp scheduler that is aware of consecutive CTA allocation to maximize performance.
- With non-maximal number of thread blocks scheduled for each core, we propose *mixed* concurrent kernel execution (mCKE) to improve resource utilization.

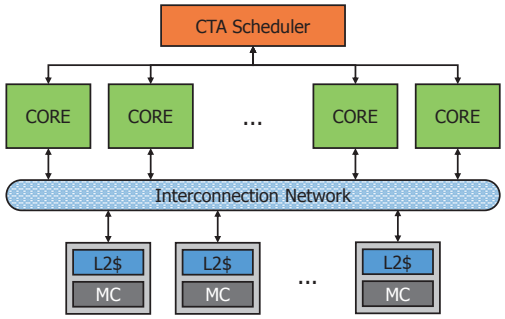

Figure 2: High-level block diagram of a GPGPU.

The rest of the paper is organized as follows. In Section 2, we describe our baseline GPGPU architecture, evaluation methodology, and thread block scheduling in modern GPUs. We then characterize the different workloads and the impact of varying the number of CTAs on overall performance in Section 3. Alternative CTA scheduling is presented in Section 4, which includes the lazy CTA scheduling (LCS) and block CTA scheduling (BCS). The simulation results of the alternative CTA scheduling are presented in Section 5. Section 6 presents related work and we conclude in Section 7.

2. Background

In this section, we first describe the experimental methodology used in this work and then, provide background information on thread block (or CTA) scheduling in modern GPGPUs.

2.1. Methodology

The GPGPU that we model consists of 28 cores (or streaming multiprocessors(SMs)) connected to 8 memory controllers, as shown in Figure 2. Each core has its own private L1 data cache, texture cache, and shared memory while each memory partition has a slice of the L2 cache and a memory controller that is shared among the SMs. The cores and the memory partitions are interconnected through an on-chip network. We use a detailed GPGPU simulator (GPGPU-sim v3) [4] in our evaluation and our configuration parameters are described in Table 1. The simulator was modified to implement the different warp and CTA scheduling that we evaluate in this work. We considered a wide range of GPGPU CUDA workloads, including applications from Rodinia [6], Parboil [30], NVIDIA SDK [22], and workloads from GPGPU-sim [4], as summarized in Table 2. To estimate power, we use the GPUWattch [18] which is integrated with GPGPU-sim and assume 45nm technology.

2.2. GPU CTA Scheduling

On NVIDIA GPUs [23, 25], a GigaThread Engine is the hardware engine on both Fermi and Kepler GPUs which is responsible for CTA scheduling – i.e., distributing the CTAs

Parameters	Value
Compute Units	28
Warp Size	32
Resources / Core	max 1536 Threads, 32768 Registers
Core / ICNT / Memory Clock	1400 MHz / 1400 MHz / 924 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, 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 1: Baseline Configuration

to the SM (stream multiprocessors) [2]. However, there is very little public information available on the details of CTA scheduling. In this work, we assume a baseline round-robin (RR) CTA scheduling [1] where the CTAs are assigned to each SM in a round-robin manner and assign the maximum number of CTAs to each core. The maximum number of CTAs assigned to each core depends on the resource usage of the workload, including the amount of registers, shared memory, etc. Once a particular CTA finishes, the CTA scheduler assigns another CTA to that particular SM, until all CTAs have been assigned to the cores.

We analyzed the behavior of the CTA scheduler through instrumentation. In the source code of the workloads, we used the PTX register `%smid` to determine which SM each CTA was assigned to. An example output of a CTA assignment to the different SMs is shown in Figure 3 for the VADD workload on the Tesla M2050 hardware, which had 14 SMs. We evaluated other workloads and synthetic workloads and they also showed similar trend. Although the CTAs are not exactly assigned in a round-robin manner, it is approximately round-robin as the the CTA assignment rotates between the different nodes. The SMs are organized hierarchically in the Tesla GPU (e.g., two SMs share a single TPC) and the scheduler might take the hierarchy into account when scheduling and not assign the CTAs using an exact round-robin scheduling. In the rest of this work, round-robin CTA scheduling is used as the baseline and present alternative CTA or thread block scheduling.

3. Workload Characteristics

We first repeat the results shown earlier in Figure 1 with a simulator and vary the number of CTAs assigned to each core as shown in Figure 4. With a simulator, there is no need to change the source code to modify the maximum number of CTAs assigned to each core. In the architecture that we simulated, the maximum number of CTAs that can be assigned to each core is 8, similar to the Fermi architecture, but depending on the workload, the maximum number of CTAs can be smaller than 8. The results in Figure 4 are categorized into four categories, based on their performance

[2]In this work, we use the term "core" and SM (stream multiprocessor) interchangeably.

Allocated CTA ID	Different SMs													
	0	1	2	3	4	5	6	7						
	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	22	23	24	25	26	27	28	29	30	31	32	33	34	35
	39	40	41	42	43	44	45	46	36	37	38	47	48	49
									50	51	52	53	54	55
	56	57	58	59	60	61	62	63	64	65	66	67	68	69
	70	71	72	73	74	75	76	77	78	79	80	81	82	83

Figure 3: CTA assignment across the different SMs on Tesla M2050 for VADD workload. The maximum number of CTAs per SM is 6 for VADD.

Name	Abbr.	# of kernels	# of CTAs	CTA Dim.	Type
Separable Convolution Filter[22]	CONV	34	18432	16×8	I
Matrix Multiplication[22]	MMUL	1	864	16×16	I
Back Propagation[6]	BP	2	65536	16×16	II
Breadth First Search[6]	BFS	24	1954	512	II
Coulombic Potential[30]	CP	1	1024	16×8	II
Ray Tracing[4]	RAY	1	8192	16×8	II
Structured Grid[6]	SRAD	1	16384	16×16	II
Matrix Addition[5]	MADD	1	1024	16×16	II
Transpose[22]	TRAN	8	4096	16×16	II
Vector Addition[22]	VADD	1	8182	256	II
3D Finite-Difference Time-Domain[22]	FDTD	1	576	16×16	II
Kmeans[6]	KMN	1	1936	256	III
MUMmerGPU[4]	MUM	1	196	256	III
EstimatePInlineP[22]	EIP	2	391	256	III
3D finite difference[22]	3DFD	1	900	16×16	IV
Black-Schole[22] option pricing	BLK	1	8192	128	IV
Seven point stencil[30]	STN	1	1024	32×4	IV
Lattrice-Boltzmann Method[30]	LBM	1	13000	100	IV
3D Laplace Solver[4]	LPS	1	2048	16×8	IV

Table 2: GPGPU Workload Description

behaviour as the number of CTAs assigned to each core is increased.

- *Type I : Increased Performance* – As the number of CTAs assigned to each core increases, the overall performance continues to increase. As more CTAs are available to each core, the workloads can explore the additional parallelism available by more efficiently utilizing the resources and/or effectively hiding the high memory latency.
- *Type II : Increased Performance and Saturate* – Similar to Type I, the performance initially increases but after some number of CTAs are assigned to each core, the performance saturates and there is no benefit of further assigning additional CTAs.
- *Type III : Decreased Performance* – Assigning minimal number of CTAs to each core results in the best performance as additional CTAs reduce performance.
- *Type IV : Increase then Decrease*: Similar to Type I and II, there is an increase in performance initially but after an optimal point, the performance decreases – similar to Type III.

In order to understand why the performance differs for the different workloads, we analyze the core activity into **AC-**

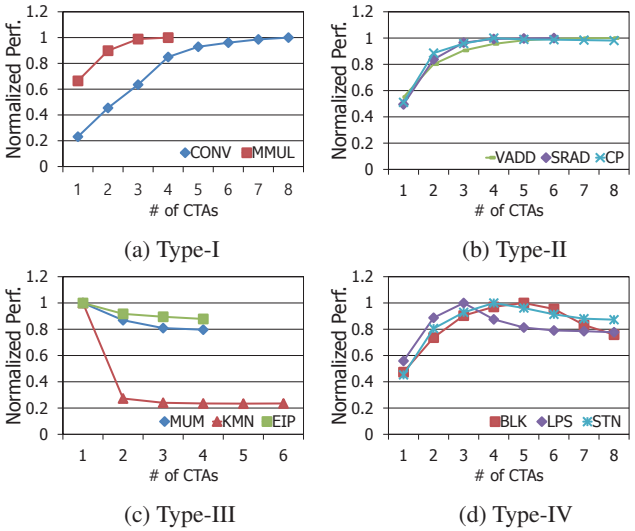

Figure 4: Performance as the number of CTAs assigned to each core increases for different workloads.

TIVE or **INACTIVE**. **ACTIVE** refers to core cycles when a warp has been issued and **INACTIVE** refers to when no warps are issued. **INACTIVE** can be partitioned into the following three categories:

- **IDLE:** There are no available warps that can be issued in the cycle (e.g., all warps do not have a valid instruction). This can occur when there are not sufficient warps (and CTAs) assigned to the core.
- **MEM_STALL:** Most of the warps in the core are stalled waiting for data reply from memory while other warps have no valid instruction to issue.
- **CORE_STALL:** The core pipeline is stalled and no warp can be issued. While some of the warps in the core might be stalled waiting for data from memory, other warps are stalled because of core/pipeline resource contention (e.g., lack of MSHR entries).

Figure 5 shows the core activity breakdown for several representative workloads from the different workload categories. As the number of CTAs assigned to each core increases, both the IDLE cycles and the MEM_STALL cycles decrease significantly for Type-I workloads (Figure 5(a)). The increase in the number of threads from larger number of CTAs helps to hide the memory latency while increasing the core utilization — and results in continuous improvement in performance. In comparison, for Type-II workloads such as SRAD (Figure 5(b)), both MEM_STALL and IDLE cycles decrease initially as the number of CTAs increase. However, as the number of CTAs continue to increase, the MEM_STALL cycles continue to decrease, similar to Type-I but the fraction of CORE_STALL cycles begin to increase. The additional threads do not necessarily help in improving performance and performance saturates.

In comparison, Figure 5(c,d) show examples where increasing the number of CTAs actually *decrease* the overall performance. For these workloads, there are significant

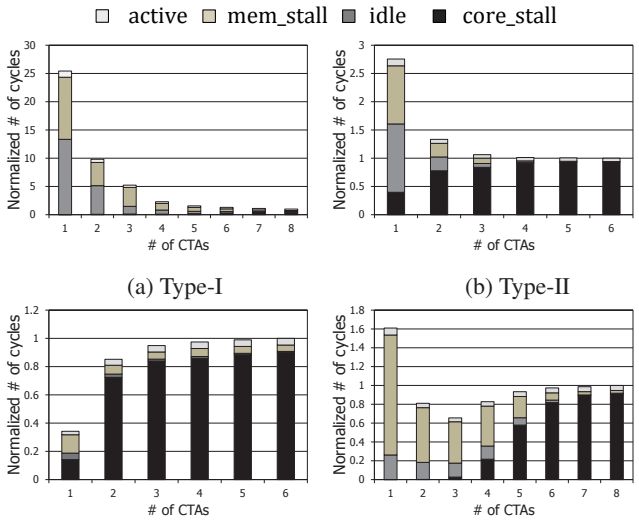

Figure 5: Core activity analysis for different representative workloads. (a) Type I - CONV (b)Type II - SRAD (c) Type III - KMN (d) Type IV - LPS

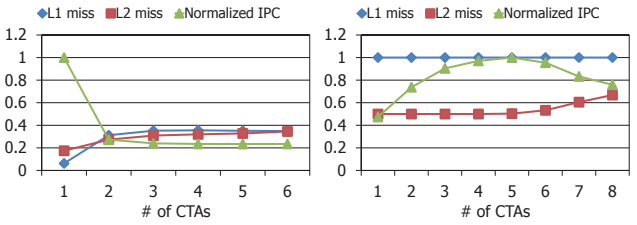

Figure 6: Performance and cache miss rate when varying CTAs in (a) KMN (Type-III) and (b) BLK (Type-IV).

CORE_STALL cycles as the additional threads result in more contention for some of the resources. The reasons for higher CORE_STALL depends on the workload itself. For some L1 cache sensitive workloads (e.g., EPI, KMN and LBM), the additional contention in the L1 data cache from the larger number of CTAs degrades overall performance (Figure 6(a)). For other workloads (e.g., LPS, BLK), performance initially increases but then, performance starts to degrade because of increased L2 cache miss rate (Figure 6(b)). For MUM, the texture cache contention increases with more CTAs, which results in higher average memory access latency and degrades overall performance.

In the following section, we propose alternative thread block scheduling that has negligible impact on Type-I workloads while improving the performance or efficiency for the other type of workloads. In particular, for Type-III and Type-IV workloads, we propose thread block scheduling to reduce the number of thread blocks allocated to each core to maximize performance (Section 4). In addition, without the maximum number of threads allocated to each core, we explore opportunities to improve efficiency through power-gating and mixed concurrent kernel execution (Section 4.4).

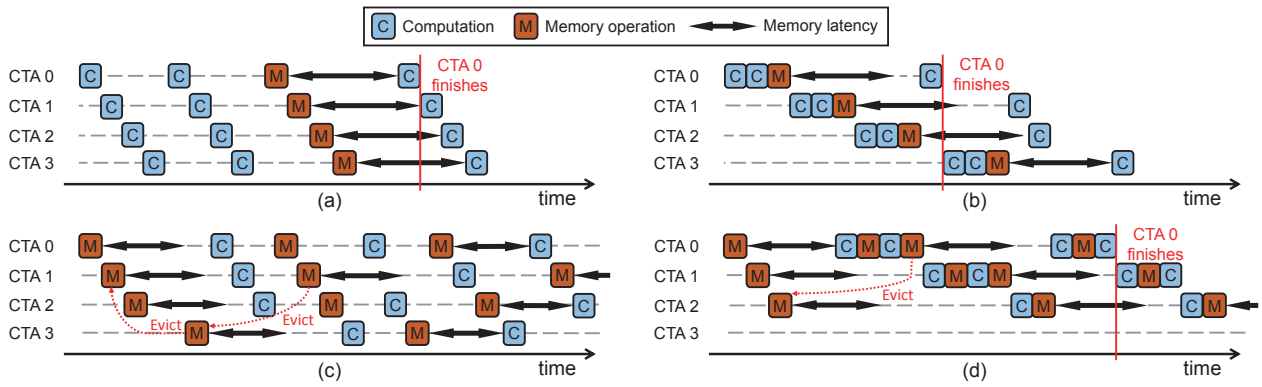

Figure 7: Example of different thread block with (a,c) round-robin warp scheduler and (b,d) greedy warp scheduler.

4. Alternative Thread Block Scheduling

In this section, we describe two alternative thread block scheduling for GPGPUs to improve the efficiency. In particular, we focus on a holistic approach by investigating the interaction between the warp scheduler and thread block (or CTA) scheduler. By leveraging a greedy warp scheduler, we propose a Lazy CTA scheduling (LCS) that reduces the maximum number of CTAs that can be assigned to each core to improve performance and energy efficiency. We then present Block CTA scheduling (BCS) where sequential CTA blocks are assigned to the same core to improve inter-CTA cache locality and an appropriate warp scheduler that exploits such locality.

4.1. Lazy CTA Scheduling (LCS)

The goal of Lazy CTA Scheduling (LCS) is to reduce the number of thread blocks allocated to each core dynamically and maximize performance (and minimize resource contention) without requiring any static analysis of the code. Prior work (DYNCTA [16]) also reduced the number of thread blocks assigned to the cores but requires dynamic analysis of the workload characteristics, including the number of idle cycles, number of cycles warps are waiting for memory data, etc. In addition, empirically determined thresholds are necessary to determine whether to reduce or increase the number of thread blocks allocated. In comparison, LCS only requires a single measurement during the execution of the first thread block and based on the data collected, the number of thread blocks allocated to the core is adjusted.

The LCS is based on the following two observations.

- Since work distribution across the different cores are done at the granularity of thread blocks, we use a single thread block (or CTA) to monitor the characteristics of a particular kernel in a workload.
- To help identify the number of sufficient threads, we leverage the interaction between the warp scheduler and thread block scheduler – in particular, exploit greedy warp scheduler to help guide the thread block scheduler.

We leverage a greedy-warp scheduler (greedy-then-oldest (GTO) [29]) to properly adjust the number of thread blocks per core instead of a round-robin warp scheduler. An example that illustrates the difference between a greedy warp scheduler and round-robin warp scheduler is shown in Figure 7. We assume workload where 4 CTAs or thread blocks is the maximum number of thread blocks allocated to each core. With a round-robin warp scheduler, each thread block will likely have issued a similar number of instructions when the first thread block finishes, as shown in Figure 7(a). However, in comparison, a greedy scheduler (e.g., GTO) prioritizes a single warp until it stalls, and then selects the oldest warp. As a results, GTO ends up prioritizing a single thread block until all warps in the given thread block are stalled – and then, selects a warp from the oldest thread block. When the first thread block finishes (Figure 7(b)), the number of instructions executed for each thread block will differ with a greedy warp scheduler – for example, while CTA0 completed and CTA1 and CTA2 issued most of its instructions (3 out of the 4 instructions), CTA3 was not able to issue any instruction. As a result, although 4 CTAs is the maximum number of CTAs, three CTAs are sufficient for this workload as the fourth CTA does not provide any further benefit – suggesting that the maximum number of threads blocks can be reduced. This can illustrate the behavior of Type-II workload where performance saturates with additional thread blocks.

Similarly, Figure 7(c)) shows an example of Type-III or Type-IV workloads where performance can degrade. In this example, we assume a memory access, followed by a long arrow, represents a memory access that is a cache miss and accesses the main memory while the other memory access that is immediately followed by a computation resulted in a cache hit. For simplicity in the example, we will assume each core has only 3 MSHR entry. Since all CTAs initially generates a memory access, the fourth CTA cannot issue its memory instruction. In this example, we also assume that CTA1 and CTA3 memory access creates an access to the same cache entry and results in a conflict miss, as shown with the dotted line with Figure 7(c)). This is another example of where the maximum number of thread blocks degrade overall performance and reducing the number of thread blocks can improve overall performance. By leveraging a greedy sched-

uler as shown in Figure 7(d)), the number of instructions issued within each thread block differs. In this example, two thread blocks are sufficient to keep the core busy, based on the number of instructions issued. In addition, the reduced number of thread blocks avoid the cache eviction that occur between CTA1 and CTA3 and avoid performance degradation. Thus, based on leveraging a greedy warp scheduler, we propose to throttle the number of the thread blocks for workloads that fall under the category Type II, III, and IV.

The LCS scheduling consists of three phases, monitor, reduce, and lazy execution and is described below. We use the terminology *lazy* execution since if necessary, the maximum number of thread blocks are not allocated to each core.

Phase 1: Monitor

Based on the workload characteristics, T_max thread blocks are initially allocated to each core. During the monitor phase, the number of instructions issued ($inst$) for each thread block x is measured. The monitor phase continues until the first thread block finishes execution.

Phase 2: Throttle

As soon as the first thread block finishes its execution, the new number of optimal thread block is calculated based on the following equation:

$$T_{new} = \left\lfloor \sum_{x=0}^{T_{max}} (inst_x) / \max(inst_x) \right\rfloor \quad (1)$$

The total number of instruction issued across all the thread block in the core is divided by the number of instructions issued from the first thread block that completed, which also corresponds to the maximum value of $inst_x$ among the different thread blocks assigned to the core. This approximation provides the number of optimal thread blocks that should be allocated, based on the core utilization. In the example Figure 7(b)), $T_{new} = \lfloor 10/4 \rfloor = 3$ and thus, the new maximum number of threads blocks is reduce from 4 to 3.

Phase 3: Lazy Execution

After Phase 2 completes (i.e., all cores have T_{new} active thread blocks assigned to each core), the kernel runs to completion with only T_{new} thread blocks allocated to each core.

The algorithm is repeated for each kernel within each workload since the behavior of each kernel can differ. Figure 8 shows an example of Phase 1 result for LPS workload. The x-axis is time while the y-axis represents the different thread blocks (or CTAs) and the plots shows the number of instructions issues for each thread block. It is clear that some of the CTAs continue to issue instruction while some CTAs (e.g, 6,7) rarely issue instructions. For this workload, the value of T_{max} is 8 initially but after phase 1, the new maximum number of threads (T_{new}) allocated to each core is reduced to 3, using the estimation shown in Equation 1. LPS is an example of Type-IV workload and thus, reducing the number of thread blocks per core can improve performance. However,

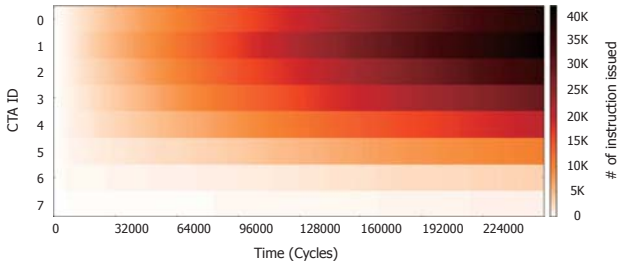

Figure 8: The number of issued instructions per CTA for LPS. LPS has eight CTAs initially but the effective number of CTAs is reduced to three.

for workloads from Type-I where reducing the maximum number of thread blocks is not necessarily beneficial, the number of instructions issued across all of the threads are approximately similar (i.e., $T_{new} = T_{max}$) and thus, the maximum number of thread blocks is not reduced and there is minimal impact on overall performance.

Hardware Complexity: The algorithm described above can be done for each core in the GPGPU system. However, because of the similar behavior of the thread blocks, analysis showed that this was not necessary and the measurement is only needed in a single core. To support LCS scheduling, performance counters are needed to measure the number of instructions issued from each thread block. Modern GPUs provide performance counters to measure this metric [27, 26] and thus, the only additional logic necessary to calculate the number of optimal number thread blocks is the logic to calculate Equation 1.

The performance of LCS is also determined by the length of Phase 1 and Phase 2. As we show in Section 5, if the number of thread blocks is relatively large, then Phase 1 and 2 represent relatively small fraction of total execution time and the overhead is negligible. However, if there are only small number of thread blocks, Phase 1 and Phase 2 can represent some overhead since the optimal number of thread blocks is not determined until the first thread block completes.

4.2. Block CTA Scheduling (BCS)

Many workloads (kernels) in GPGPU workloads are organized as a 2D array of CTAs, as shown in Figure 9(a). The CTA or the thread block size ($X \times Y$) is a parameter that is determined by the programmer but a commonly used CTA size is a 2D 16×16 (256 threads), as suggested by the CUDA programming manual [24]. Because of how data is laid out, *inter-CTA locality* can exist among sequential CTAs in the workloads. For example, assume a kernel with 16×16 CTA dimensions and data that is accessed by each thread is a single word (4Bytes). Each row of data from a CTA will occupy $16 \times 4 = 64$ Bytes and since the cache line size of L1 cache is 128Bytes in current GPUs, spatial locality exists between neighboring CTAs. However, with a round-robin thread block scheduling, the inter-CTA spatial locality is lost since sequential CTAs are not assigned to same core,

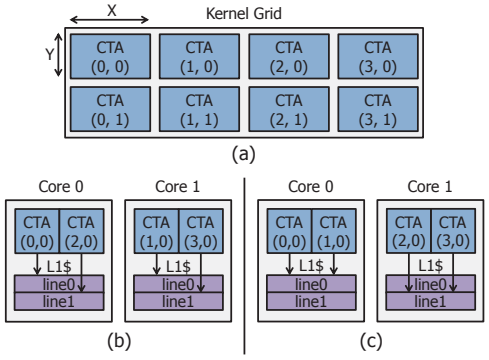

Figure 9: (a) An example of kernel with Two-Dimensional CTAs, (b) conventional round-robin thread block scheduling and (c) proposed block CTA scheduling (BCS).

as shown in Figure 9(b).

To exploit inter-CTA locality, we propose Block CTA scheduling (BCS) that assigns a *block* of sequential thread blocks or CTAs to the same core. In this work, we focus on inter-CTA locality that exploits L1 cache spatial locality – thus, we focus on a block of size 2 CTAs. BCS is not applicable to workloads with one-dimensional CTAs as there is little inter-CTA L1 locality. Figure 9(c) shows an example of BCS as pair of sequential CTAs are assigned to same core and their locality can be exploited — i.e. Core 0 is assigned CTA(0,0) and CTA(1,0) while Core 1 is assigned CTA(2,0) and CTA(3,0). This allocation can exploit spatial locality across the same cache line within the local L1.

One challenge in BCS is how to assign new thread blocks when prior thread blocks finish execution since the block of CTAs do not necessarily finish execution at the same time. As a result, to assign sequential thread blocks to the same core, we used *delayed* scheduling or assignment of thread blocks – i.e., a new thread block is not allocated to a core until pair of sequential thread blocks finish execution.

To effectively exploit the inter-CTA locality with BCS, the warp scheduler also needs to be aware of inter-CTA spatial locality and schedule the warps accordingly. Thus, we propose *sequential CTA-aware* (SCA) warp scheduling that combines both round robin and greedy warp scheduling. The warps are scheduled in a round-robin manner between two warps of neighboring thread blocks or within a block. However, the warp scheduler remains *greedy* as these set of warps are prioritized, similar to GTO warp scheduler, until one of the two warps stall. Then, the next group of warps within the same block is scheduled. In our evaluation in Section 5, unless otherwise noted, the result of BCS thread block scheduling also implies that the warp scheduler used is SCA.

4.3. Combined Thread Block Scheduling

Figure 10 illustrates how both LCS and BCS can be combined. Initially, the workload can be categorized as 1D or 2D workload. For 1D workloads, the LCS described earlier in

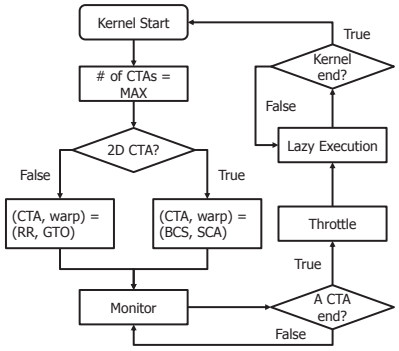

Figure 10: The Flow Chart of Combined Architecture for thread block scheduling, combining LCS and BCS.

Section 4.1 is applied. However, for 2D workloads, LCS and BCS is combined as BCS described in Section 4.2 is used to not only improve inter-CTA locality but is also leveraged to determine the optimal number of thread blocks within LCS. After the initial monitor phase, the new value of T_{new} is used as the maximum number of thread blocks and GTO is used for warp scheduler for 1D workload while SCA is used as the warp scheduler for 2D workloads. As we show later in Sectio 5, the combined architecture (LCS+BCS) has no impact on 1D workloads but improves performance on 2D workloads. In particular, by leveraging BCS in the monitor phase, we show how it improves scalability by increasing the number of optimal thread blocks allocated to each core (compared with running LCS alone) and improves overall performance.

4.4. Increasing Efficiency of GPGPUs : mixed Concurrent Kernel Execution (mCKE)

Allocating less than the maximum number of thread blocks to each core presents opportunities to improve the efficiency of the GPGPUs as there are un-utilized resources. In particular, modern GPUs have very large register file and shared memory to support the larger number of threads. For example, NVIDIA GPUs can have 128KB register file and 48KB shared memory per core in Fermi architecture [23] and for the Kepler [25], the register file capacity has been doubled to support more threads. Similar to prior work [2], the unused resource (e.g., register file, shared memory) can be power-gated to improve energy-efficiency with the reduced number of thread blocks allocated to each core with LCS

In addition, the underutilized resources within a core provide opportunity for concurrent execution of different kernels on the *same* core, which we refer to as *mixed* concurrent kernel execution (mCKE). Modern GPU architectures support concurrent kernel execution (CKE) where independent kernels can be launched and executed at the same time [24, 23, 25]. The main goal of CKE is to efficiently utilize the GPU by overlapping kernel execution. However, the baseline CKE assumes that the different kernels are executed on different cores. Since the resources available within

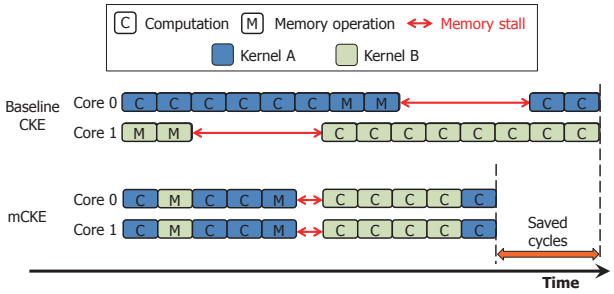

Figure 11: Block Diagram of Mixed Concurrent Kernel Execution (mCKE).

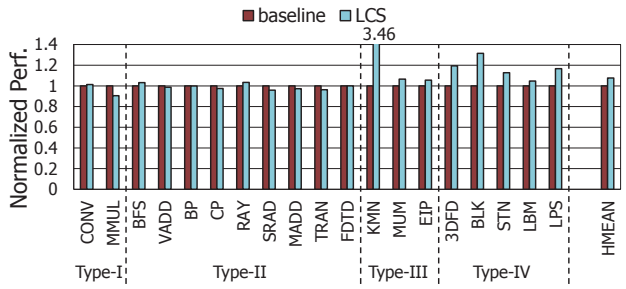

Figure 12: Performance results of Lazy CTA Scheduling (LCS).

a single core are not fully utilized with LCS, we propose to assign CTAs or thread blocks from different kernels on to the same core, which we refer to as *mixed concurrent kernel execution* (mCKE)

The proposed mCKE can not only increase resource utilization but improve overall performance when workloads with different characteristics are combined on a single core. This is also true when the mixed kernels have different characteristic for resource usage. Figure 11 shows an example of how performance can improve with mCKE, compared with baseline CKE. We assume two kernels (kernel A and B) are scheduled across two cores. In the baseline CKE, each core can be stalled at different point in time while waiting for the response from the memory and result in the core being idle for significant amount of time. However, by interleaving the kernels on the same core with mCKE, the memory latency can be hidden (or overlapped) with other kernel execution and effectively improve overall performance. This is similar to the benefits of two-level warp scheduling [20] where the memory accesses from the warps within a thread block are not necessarily schedule together but partitioned into different fetch groups.

5. Evaluation

In this section, we use the simulation methodology described earlier in Section 2.1 and evaluate the proposed alternative thread block scheduling described earlier in Section 4.

5.1. Lazy CTA Scheduling Results

The results of LCS are shown in Figure 12 with the results normalized to baseline that has greedy-then-oldest (GTO) warp scheduler and round-robin CTA scheduler. On average, there is approximately 7% improvement in performance but for Type-III and Type-IV workloads, there is approximately 23% increase in performance. For Type-I and II workloads, the purpose of LCS was to maintain the performance provided by the baseline scheduler while for Type-III and IV workloads, the goal was to reduce the number of thread blocks and improve performance. As a result, LCS resulted in very little improvement or slight degradation in performance for Type-I and II workloads. For Type III and IV workloads, LCS improves overall performance by reducing the maximum number of thread blocks that can be assigned to a core and improve L1 and/or L2 cache utilization.

Figure 13 shows the number of CTAs allocated to a core with LCS. The results are compared against the baseline which allocates the maximum number of CTAs for each core (determined by the usage of shared resources such as the register file or shared memory). In addition, we also compared against the optimal number of thread blocks (OPT), which is the number of thread blocks when performance saturates or reaches its peak – based on the simulations shown earlier in Figure 4. In general, LCS is able to approach the optimal number of thread blocks and in general, reduce the number of thread blocks, compared with the baseline.

However, for some of the workloads (such as MUM) from Type-III workloads, even though LCS was able to determine the near optimal number of thread blocks, the performance improvement was very minimal – only a few percent increase in performance for MUM (Figure 12). As described earlier in Section 4.1, the minimal performance improvement is from the overhead of the monitor phase in LCS and the total number of thread blocks. For MUM which has a total of 196 CTAs, approximately 57% of the CTAs are initially assigned to all of the cores and thus, Phase 1 (monitor phase) resulted in a significant fraction of total execution time and overall benefit from LCS was relatively small. For some workloads, such as MMUL, there was some performance degradation as the number of optimal thread blocks determined by LCS was smaller than OPT value — OPT was 3 CTAs while LCS determined it to be two. With the relatively small number of CTAs allocated to each core, the allocation of only two CTA (compared with 3 CTAs), meant the number of available threads were reduced by approximately $1/3$ and resulted in the performance loss. To minimize the performance degradation for Type-I workloads, one modification to the LCS algorithm in Equation 1 would be to use the $\lceil T_{new} \rceil$ instead of $\lfloor T_{new} \rfloor$. This would result in a trade-off in performance improvement of Type-I workload, while resulting in some performance improvement loss for Type-III and Type-IV workloads.

The energy improvements are shown in Figure 14 for the entire GPGPU, including the core, on-chip memory and network, and memory controller. We assume ideal power-gating such that unused resources (such as shared memory and register file) are power-gated without any overhead for the baseline

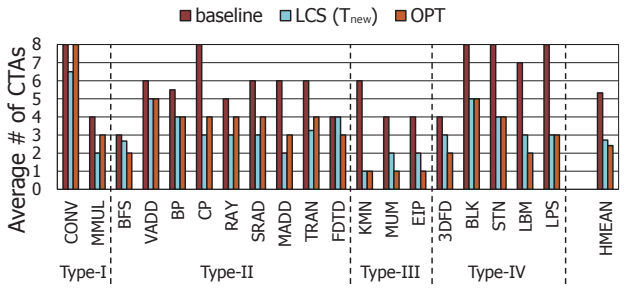

Figure 13: The optimal number of CTAs (OPT), compared with T_{new} determined by LCS.

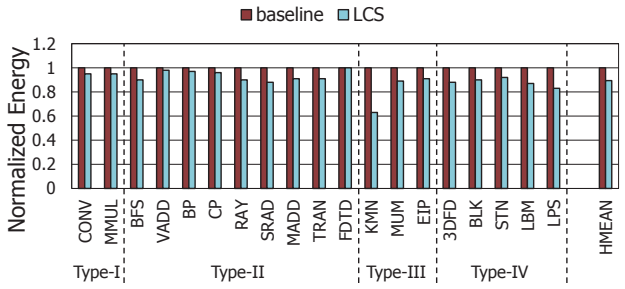

Figure 14: Energy saving results of LCS

and if there are unutilized resources, we assume it is power-gated. For LCS, when less than the maximum number of threads blocks are allocated, additional resources are power-gated. Across all type of workloads, LCS results in energy improvement, up to 37% for Type-III workloads and on average, 11% improvement.

5.2. Block CTA Scheduling Results

Results from Block CTA Scheduling (BCS) is shown in Figure 16 for only 2D workloads. We compare the results of baseline with BCS using GTO warp scheduler as well as SCA warp scheduler. On average, BCS+GTO results in only 3% improvement in performance as only the block assignment of thread blocks to each core does not necessarily result in performance improvement with a GTO warp scheduler. In comparison, BCS+SCA is able to improve performance by 15% on average and up to 70% for some of the workloads as the SCA warp scheduler is able to fully exploit the inter-CTA cache locality. For workloads such as STN which has a thread block size of 32 in the x-dimension (larger than 16), it does not necessarily exploit the same inter-CTA L1 locality as the other workloads but there is still approximately 7% performance improvement. Since this is a stencil-type workload, data is still shared between adjacent thread blocks and BCS helps to improve overall performance.

Figure 17 shows the L1 data miss rate improvement with BCS. Results show that the combination of BCS+SCA significantly reduces the L1 miss rate while BCS+GTO does not provide the same benefit – on average, BCS+SCA reduce L1 miss rate by 24% while BCS+GTO only reduces L1 miss rate by 8%. However, for some workloads such as MMUL, L1

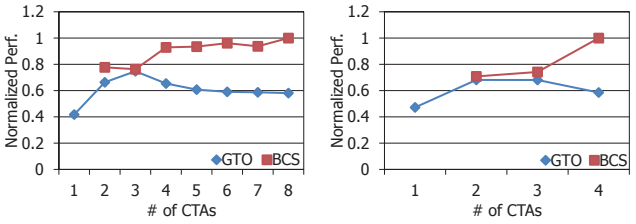

Figure 15: Performance when varying the number of thread block allocated to each core for (a) LPS and (b) 3DFD with BCS.

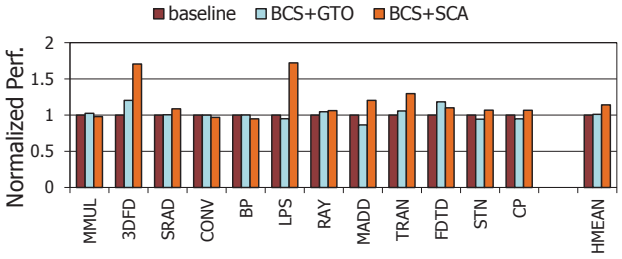

Figure 16: Performance of results with Block CTA scheduling (BCS) and Sequential CTA-aware (SCA) warp scheduling.

miss rate is decreased by more than 30% but there is minimal impact on overall performance. As described earlier in Section 4.2, one possible performance overhead of BCS is the *delayed* thread block assignment in order to assign consecutive CTAs to the same core. For most of the workloads, this had minimal impact since this additional delay is hidden as long as the core is active with other thread blocks. However, for MMUL with only 4 thread blocks per core, the delayed thread block scheduling results in only 50% occupancy. Thus, although the miss rate is reduced with BCS, the delayed thread block assignment negates the benefit from reduce miss rate. For workload such as BP which had very little inter-CTA locality and consisted of more write than reads, BCS+SCA results is slight performance degradation.

With BCS, additional benefit is the increased scalability of the workload as additional thread blocks helps to improve overall performance. Figure 15 shows the performance scaling when increasing the number of thread blocks for each core for two particular workloads (LPS and 3DFD), comparing the baseline with GTO and BCS+SCA. Both of these workloads were classified as Type-IV workloads earlier with baseline GTO where performance started to decrease after some number of thread blocks were assigned. However, after using BCS, the behavior of the workloads approaches Type-I or II as the improved thread block scheduling, in combination with appropriate warp scheduling, improve the efficiency of the resources.

5.3. Mixed Concurrent Kernel Execution Results

The results of mCKE is shown in Figure 18 and compared with baseline CKE. Since workloads that we evaluated do

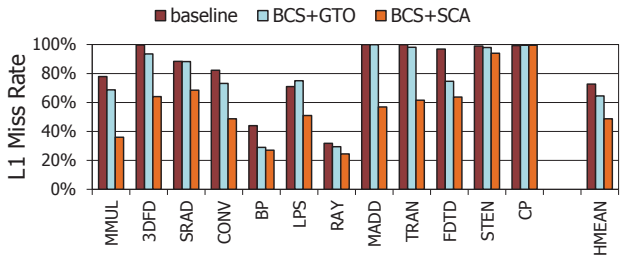

Figure 17: L1 cache miss rate comparisons.

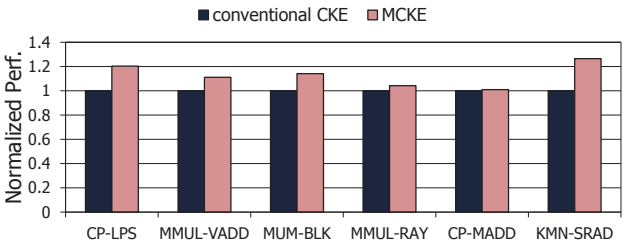

Figure 18: Performance Results for Mixed Concurrent Kernel Execution.

not leverage concurrent kernel execution, we merge different workloads such that we can execute CKE, similar to prior work [28]. The different workload mixes are summarized in Table 3 and the number of CTAs allocated to each core is based on the optimal number of thread blocks from LCS. For the baseline CKE, we also assume the LCS was used such that the optimal number of thread blocks were allocated. For some of the workload mix, mCKE has minimal impact on overall performance compared with baseline CKE. However, for other mix of workloads such as KMN-SRAD, mCKE results in up to 27% improvement in performance. For KMN where the optimal number of thread blocks was only one thread block with LCS, it provides opportunity for performance improvement with additional kernel executing on the same core. In addition, for a given workload, the benefit of mCKE depends on which workload it is mixed with – for example, CP-MADD mix has minimal impact from mCKE while CP-LPS results in 20% improvement with mCKE. CP-MADD results in another form of *load-imbalance* as the execution time of the two workloads differs significantly and mCKE has minimal impact. In comparison, LPS is a memory intensive workload while CP is compute intensive – thus, mix of these two kernels improves resource utilization to result in performance gain.

5.4. Comparison to Alternative Scheduling

We compare the performance of the proposed thread block scheduling with two previously proposed warp scheduler (TLV and CCWS) and a CTA scheduler (DYNCTA). The results are normalized to GTO warp scheduler [29].

Two-Level Round Robin scheduler (TLV) [20]: The warp scheduler subdivides warps to fetch groups and select from one fetch groups until all warps in the fetch groups are stalled. It schedules Round Robin in a fetch group.

Mixed workloads	Type	# of CTAs per kernel
CP-LPS	II - IV	3, 5
MMUL-VADD	I - II	2, 3
MUM-BLK	III - IV	2, 4
MMUL-RAY	I - II	2, 2
CP-MADD	II - II	3, 4
KMN-SRAD	III - II	1, 5

Table 3: Workload Description for mCKE evaluation.

Cache Conscious warp scheduler (CCWS) [29]: The warp scheduler dynamically controls the number of warps which are allowed to be scheduled to improves L1 hit rates for cache-sensitive applications. Within CCWS, GTO scheduling is used within the selected warp boundary.

Dynamic CTA Scheduling (DYNCTA) [16]: The CTA scheduler initially allocates $T_{max}/2$ thread blocks to each core and the number of CTAs is incremented or decremented by 1 continuously for within each sampling period. DYNCTA uses *CTA pausing* to deprioritize the warps in the most recently assigned CTA on the SM. We use DYNCTA with GTO warp scheduler for fair comparison.

Figure 19 shows the results comparison and we show both LCS and combined LCS+BCS. On average, LCS+BCS exceeds the performance of other scheduler, by 16% compared with the baseline GTO scheduler, by 13% over CCWS warp scheduler and by 14% over DYNCTA scheduler. CCWS performs well on cache sensitive workloads (such as KMN) but LCS+BCS exceeds the performance of CCWS on these workloads. CCWS uses a victim tag array in the L1 cache and warp scheduling *reacts* to the hit rate of the victim tag array to reduce the number of active threads on a core. In comparison, the LCS+BCS the number of thread blocks to one after the completion of a single thread block and maintains it throughout the execution of the kernel. In our configuration, the optimal number of warps for KMN was closer to 8 warps, hich was the size of the thread block. If the optimal number of warp was significantly less than 8 warps (or the size of a single thread block), LCS+BCS cannot further reduce the number of warps scheduled and a warp scheduler such as CCWS could further improve performance. However, for EIP, CCWS outperform LCS+BCS by 13%. Similar to MUM, EIP is also a workload with relatively small number of thread blocks and thus, Phase 1 and Phase 2 of LCS represents a significant amount of entire execution time. As a result, LCS+BCS is not able to adjust the number of thread blocks quickly enough while CCWS, working at the warp granularity, is able to adjust the warps scheduled more quickly. Instead of using GTO as the baseline warp scheduler, CCWS can also be used as the warp scheduler within LCS to improve performance on workloads such as EIP, but at the cost of higher complexity warp scheduler. However, in general, LCS+BCS outperform CCWS since CCWS only targets workloads which have intra-warp locality [29] in L1 cache while LCS+BCS can be applied to wide range of workloads.

LCS (and LCS+BCS) provides performance improvement

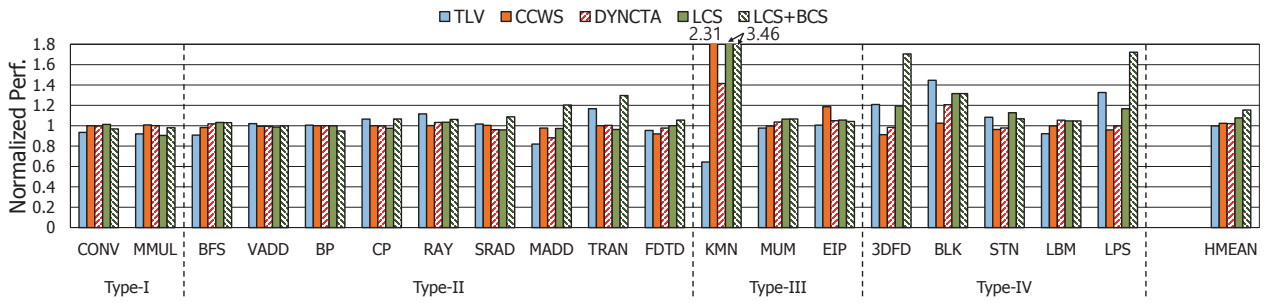

Figure 19: Performance comparison against CCWS, TLV, and DYNCTA. The results are normalized to GTO.

of TLV but for a few of the workloads (e.g., BLK), TLV provides the best performance improvement. Since TLV partitions the warps within a group into different fetch groups and schedules within the fetch group before moving on to the next fetch group, it can be more effective in hiding the memory latency and also improves the inter-CTA L2 locality – compared with LCS, the TLV reduces L2 miss rate by approximately 10%. Similar to CCWS, if TLV is used as the warp scheduler for such workloads within LCS, LCS can likely provide further benefits as well.

Performance of LCS+BCS also exceeds the performance DYNCTA by 14%, on average. We used the same set of thresholds that were used in [16]. Since DYNCTA depends on empirically thresholds to determine whether to increase or decrease the number of thread blocks assigned to each core, a single of set of threshold is not likely to be optimal across all the workloads. In addition, in our evaluation, we noted that DYNCTA can unnecessarily fluctuate in the number of thread blocks allocated to each and impact performance. Note that prior work [16] showed performance benefits while our results do not show significant benefits of DYNCTA. One key difference is that the baseline we assumed is a greedy, GTO warp scheduler. Load-imbalance [4] was one aspect that DYNCTA tried to address but with a greedy scheduler, most of the load-imbalance problem can be removed and thus, the benefit of DYNCTA was reduced.

In addition, on average, LCS provided 8% improvement in performance while LCS+BCS provided 15% improvement. For some of the workloads (such as LPS and 3DFD). the use of LCS+BCS improved the scalability as the optimal number of thread blocks increased and improved overall performance (as shown earlier in Figure 15).

6. Related Work

To improve the performance in GPGPU architectures, different schedulers have been proposed. The two-level warp scheduling [20], cache-conscious warp scheduling (CCWS) [29], and dynamic CTA scheduling (dynCTA) [16] were discussed earlier in Section 5.4 and compared with our proposed scheduling mechanism. Gebhart et al. [11] also proposed two-level warp scheduling for energy efficiency in GPUs where warps are separated into active set and pending set. The Cooperative Thread Array Aware Scheduling [15] differs from prior warp schedulers as it is CTA-*aware* but it is still based on the warp scheduler. CTA pausing [16] was presented to deprioritize CTAs in order to reduce the number of CTA scheduled on a core and is similar to the Throttle phase in the proposed LCS algorithm. Laskshminarayana et al. [17] explore many warp scheduling in GPUs. They observe that the performance of workloads with a balanced instructions per warp increase the fairness of their warp scheduling and dram scheduling policy. Fung et al. [9] proposed Dynamic Warp Formation(DWF) to reduce underutilization of resources from branch divergence. Thread block compaction [8] improved upon DWF by exploit control flow locality among threads. There are some similarities of this work with prior work in different aspects while some of the previously techniques are orthogonal and can be combined with the alternative thread block scheduling proposed in this work. However, this work differs as it explores the interaction between the warp and thread block scheduling to increase overall efficiency.

In evaluating scheduling and prefetching within GPGPU, Jog et al. [14] used a CTA allocation strategy where consecutive CTAs were assigned to the same core in their baseline architecture. This share similarity with BCS but there is no analysis on the impact of such CTA scheduling or the benefits. In addition, it is not clear exactly how additional CTAs are allocated after CTAs complete. As discussed earlier, prior work [4, 16] have also made similar observation as this work that maximal number of CTAs per core is not not always the optimal policy – i.e., increasing the number of CTAs does not necessarily improve performance. However, no solution was provided in [4] and a detailed comparison with [16] was discussed earlier in Section 5.4.

Within general purpose CPUs, prior work has also shown that more threads are not necessarily better in CPUs. Guz et al. [13] described the "performance valley" where too many threads can degrade performance because of resource contention. Suleman et al. [31] showed similar results for multithreaded workloads and described a dynamic method to find the optimal number of threads. Cheng et al. [7] proposed a thread throttling scheme to reduce memory access latency in a multithreaded system. However, the number of threads

in a conventional processor is significantly smaller than the GPGPU architecture that we consider and these techniques are not necessarily applicable to GPGPU architectures with thousands of threads.

Adriaens et al. [3] proposed spatial multi-tasking where multiple applications share the GPU resources by partitioning cores among the different applications. However, in our work, the core resources are partitioned among the different kernels. Phi et al. [28] propose a kernel converting technique, Elastic Kernel, which allows fine-grained control of GPU resources. The number of logical and physical thread blocks mapped can be changed to maximize utilization of GPU resource by transformation of the kernel. Gregg et al. [12] introduce kernel scheduling framework and, kernel merge, which increase the concurrency of kernel execution. Kernel merge provides a tuning ability of control executing thread blocks with different scheduling algorithms. It remains to be seen how mixed concurrent kernel execution can leverage these techniques to further improve overall performance.

7. Conclusion

In this work, we explored alternative thread block or CTA scheduling in GPGPU to improve performance. We first analyzed how varying the number of thread blocks allocated to each core impacts performance. Since the maximum number of thread blocks does not necessarily maximize performance, we propose LCS (lazy CTA scheduling) that leverages a greedy warp scheduler to determine the optimal number of thread blocks per core. In addition, we show how BCS (block CTA scheduling), where consecutive thread blocks are assigned to the same cores, can exploit inter-CTA locality to improve overall performance. To efficiently leverage BCS, we propose an alternative warp scheduler that is aware of the consecutive thread blocks allocated to the same core and exploit the inter-CTA locality. In addition, since the maximum number of thread blocks does not necessarily improve performance, we exploit this opportunity by proposing *mixed* concurrent kernel execution to improve performance and resource utilization by executing multiple kernels on the same core.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Tom Conte, for their comments. 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 Ministry of Science, ICT & Future Planning (NRF-2011-0015039).

References

[1] K. M. Abdalla et al. Scheduling and Execution of Compute Tasks, US Patent US20130185725, 2013.

[2] M. Abdel-Majeed et al. Warped Register File: A Power Efficient Register File for GPGPUs. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 344–355, Tel-Aviv, Israel, 2013.

[3] J. Adriaens et al. The Case for GPGPU Spatial Multitasking. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, New Orleans, LA, USA, 2012.

[4] A. Bakhoda et al. Analyzing CUDA Workloads using a Detailed GPU Simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, Boston, Massachusetts, USA, 2009.

[5] I. A. Buck. Programming CUDA. In *Supercomputing 2007 Tutorial Notes*, 2007.

[6] S. Che et al. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *International Symposium on Workload Characterization (IISWC)*, pages 44–54, Austin, TX, USA, 2009.

[7] H.-Y. Cheng et al. Memory Latency Reduction via Thread Throttling. In *International Symposium on Microarchitecture (MICRO)*, pages 53–64, Atlanta, Georgia, USA, 2010.

[8] W. W. L. Fung et al. Thread block compaction for efficient simt control flow.

[9] W. W. L. Fung et al. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *International Symposium on Microarchitecture (MICRO)*, pages 407–420, Chicago, Illinois, USA, 2007.

[10] M. Garland et al. Parallel Computing Experiences with CUDA. *Micro, IEEE*, 28(4), 2008.

[11] M. Gebhart et al. Energy-Efficient Mechanisms for Managing Thread Context in Throughput Processors. In *International Symposium on Computer architecture (ISCA)*, pages 235–246, San Jose, California, USA, 2011.

[12] C. Gregg et al. Fine-Grained Resource Sharing for Concurrent GPGPU Kernels. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar)*, pages 10–10, Berkeley, CA, USA, 2012.

[13] Z. Guz et al. Many-Core vs. Many-Thread Machines: Stay Away From the Valley. *IEEE Computer Architecture Letters*, 8(1):25–28, 2009.

[14] A. Jog et al. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 332–343, Tel-Aviv, Israel, 2013.

[15] A. Jog et al. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 395–406, Houston, TX, USA, 2013.

[16] O. Kayiran et al. Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs. In *International Conference on Parallel Architecture and Compiliation Techniques(PACT)*, pages 157–166, Edinburgh, Scotland, UK, 2013.

[17] N. B. Lakshminarayana et al. Effect of Instruction Fetch and Memory Scheduling on GPU Performance. Workshop on Language,Compiler, and Architecture Support for GPGPU, 2010.

[18] J. Leng et al. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *International Symposium on Computer Architecture (ISCA)*, pages 487–498, Tel-Aviv, Israel, 2013.

[19] A. Munshi. The OpenCL Specification, 2011.

[20] V. Narasiman et al. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In *International Symposium on Microarchitecture (MICRO)*, pages 308–317, Porto Alegre, Brazil, 2011.

[21] J. Nickolls et al. The GPU Computing Era. *Micro, IEEE*, March-April.

[22] NVIDIA. CUDA C/C++ SDK Code Samples, 2011.

[23] NVIDIA. Fermi: NVIDIA's Next Generation CUDA Compute Architecture, 2011.

[24] NVIDIA. CUDA C Programming Guide, 2012.

[25] NVIDIA. Kepler: The Fastest, Most Efficient HPC Architecture Ever Built, 2012.

[26] NVIDIA. NVIDIA PerfKit: NVIDIA Performance Toolkit, 2013.

[27] NVIDIA. Profiler User's Guide, 2013.

[28] S. Pai et al. Improving GPGPU Concurrency with Elastic Kernels. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 407–418, Houston, TX, USA, 2013.

[29] T. Rogers et al. Cache-Conscious Wavefront Scheduling. In *International Symposium on Microarchitecture (MICRO)*, pages 78–85, Vancouver, Canada, 2012.

[30] J. Stratton et al. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012.

[31] M. A. Suleman et al. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–286, Seattle, WA, USA, 2008.