Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs

Onur Kayıran, Adwait Jog, Mahmut T. Kandemir and Chita R. Das
Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802
{onur, adwait, kandemir, das}@cse.psu.edu

Abstract—**General-purpose graphics processing units (GPGPUs) are at their best in accelerating computation by exploiting abundant thread-level parallelism (TLP) offered by many classes of HPC applications. To facilitate such high TLP, emerging programming models like CUDA and OpenCL allow programmers to create work abstractions in terms of smaller work units, called cooperative thread arrays (CTAs). CTAs are groups of threads and can be executed in any order, thereby providing ample opportunities for TLP. The state-of-the-art GPGPU schedulers allocate maximum possible CTAs per-core (limited by available on-chip resources) to enhance performance by exploiting TLP.**

However, we demonstrate in this paper that executing the maximum possible number of CTAs on a core is not always the optimal choice from the performance perspective. High number of concurrently executing threads might cause more memory requests to be issued, and create contention in the caches, network and memory, leading to long stalls at the cores. To reduce resource contention, we propose a dynamic CTA scheduling mechanism, called DYNCTA, which modulates the TLP by allocating optimal number of CTAs, based on application characteristics. To minimize resource contention, DYNCTA allocates fewer CTAs for applications suffering from high contention in the memory subsystem, compared to applications demonstrating high throughput. Simulation results on a 30-core GPGPU platform with 31 applications show that the proposed CTA scheduler provides 28% average improvement in performance compared to the existing CTA scheduler.

Keywords—GPGPUs; thread-level parallelism; scheduling

I. INTRODUCTION

Interest in GPGPUs has recently garnered momentum because they offer an excellent computing paradigm for many classes of applications, specifically HPC applications with very high thread-level parallelism (TLP) [8], [13], [22], [25], [26]. From the programmer's perspective, evolution of CUDA [29] and OpenCL [27] frameworks has made programming GPGPUs simpler. In the CUDA programming model, applications are divided into work units called *CUDA blocks* (also called as *cooperative thread arrays* – CTAs). A CTA is a group of threads that can cooperate with each other by synchronizing their execution. Essentially, a CTA encapsulates all synchronization primitives associated with its threads. GPGPU architecture provides synchronization guarantees within a CTA, and no dependencies across CTAs, helping in relaxing CTA execution order. This leads to an increase in parallelism and more effective usage of cores. Current GPGPU schedulers attempt to allocate the maximum number of CTAs per-core (streaming multiprocessor [31]), based on the available on-chip resources (register file size, shared memory size, and the total number of SIMT units), to enhance performance by hiding the latency of a thread while executing another one [29].

However, we demonstrate in this paper that exploiting the maximum possible TLP may not necessarily be the best choice for improving GPGPU performance, since this leads to high amounts of inactive time at the cores. The primary reason behind high core inactivity is high memory access latencies primarily attributed to limited available memory bandwidth. Ideally, one would expect that exploiting the maximum available TLP will hide long memory latencies, as the increased number of concurrently running CTAs, in turn, threads, will keep the cores busy, while some threads wait for their requests to come back from memory. On the other hand, more threads also cause the number of memory requests to escalate, aggravating cache contention as well as network and memory access latencies. To quantify this trade-off, Figure 1 shows performance results of executing the optimal number of CTAs per-core (in turn, optimal TLP) and the minimum number of CTAs, which is 1. The optimal number of CTAs per-core is obtained by exhaustive analysis, where each application is simulated multiple times, with all the possible per-core CTA limits. The results are normalized with respect to the default CUDA approach where the maximum number of CTAs execute on the cores. These results suggest that varying TLP at the granularity of CTAs has a significant impact on the performance of GPGPU applications. The average IPC improvement across all 31 applications with optimal TLP over maximum TLP is 39% (25% geometric mean). This number is very significant in applications such as `IIX` (4.9×) and `PVC` (3.5×).

The optimal TLP might be different for each application, and determining it would be impractical as it would require executing the application for all possible levels of thread/CTA parallelism. Motivated by this, we propose a method to find the optimal TLP at the granularity of CTAs dynamically during run-time. We believe that our work is the first to propose an optimal CTA scheduling algorithm to improve performance of GPGPUs. In this context, we propose a dynamic CTA scheduling algorithm (DYNCTA) that modulates per-core TLP. This is achieved by monitoring two metrics, which reflect the memory intensiveness [1] of an application during execution, and changing TLP dynamically at the granularity of CTAs depending on the monitored metrics. DYNCTA favors (1) higher TLP for compute-intensive applications to provide latency tolerance, (2) lower TLP for memory-intensive applications to

[1]Our application suite is primarily divided into two categories: memory- and compute-intensive applications. Details on the classification criteria is described in Section III-A.

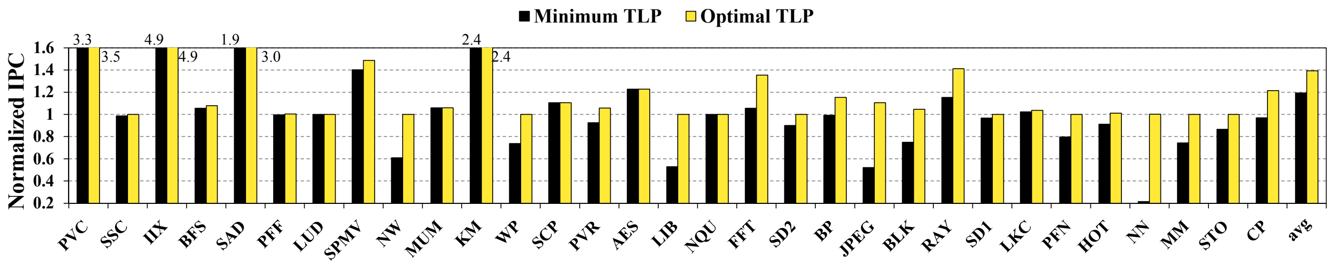


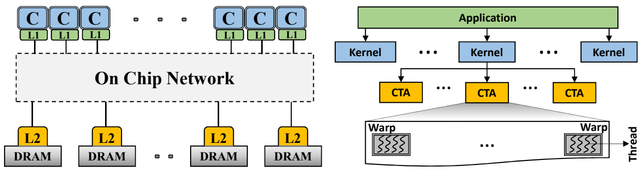
Fig. 1: Impact of varying TLP on IPC. The first, and the second bars show the normalized speedup with respect to the CUDA default (maximum TLP), when TLP is minimum (1 CTA on each core), and optimum (the optimal number of CTAs on each core), respectively.

reduce cache, network and memory contention. Evaluation on a 30-core GPGPU platform with 31 diverse applications indicate that DYNCTA increases application performance (IPC) by 28% (up to 3.6×), on average, and gets close to the potential improvements of 39% with optimal TLP.

II. Background

Baseline GPGPU architecture: A GPGPU consists of many simple in-order shader cores, with each core typically having "single-instruction, multiple-threads" (SIMT) lanes of 8 to 32. Our target GPGPU architecture, shown in Figure 2a, consists of 30 shader cores each with 8 SIMT lanes, and 8 memory controllers (MCs), similar to [9], [10], [33]. Each core has a private L1 data cache, a read-only texture and a constant cache, along with a low-latency shared memory. 10 clusters each of which contain 3 cores are connected via a crossbar interconnect to 8 MCs [9], [10], [33]. Each MC is associated with a slice of shared L2 cache bank. An L2 cache bank with an MC is defined as one "memory partition". The baseline architecture models memory coalescing in detail, where nearby memory accesses are coalesced into a single cache line, reducing the total number of memory requests. The baseline platform configuration used in this work is given in Table I.

GPGPU application design: Figure 2b shows the hierarchy of a GPGPU application consisting of threads, CTAs, and kernels. A group of threads constitute a "CTA" or "thread block". A CTA is essentially a batch of threads that can coordinate among each other by synchronizing their execution streams using barrier instructions. Since all the synchronization primitives are encapsulated in the CTA, execution of CTAs can be performed in any order. This helps in maximizing the available parallelism and any core is free to schedule any CTA. Further, each kernel is associated with many CTAs, and one or multiple kernels form a GPGPU application. A "warp" or a "wavefront" is the granularity at which threads are scheduled to the pipeline, and is a group of 32 threads. Note that a warp is an architectural structure rather than a programming model concept.



(a) Baseline Architecture *(b)* Application Hierarchy

Fig. 2: Hardware and Software Architecture of GPGPUs

TABLE I: Baseline Configuration

Shader Core Config. [33]	30 Shader Cores, 1300MHz, 5-Stage Pipeline (Fetch, Decode, Memory, Execute, WriteBack), SIMT Width = 8
Resources / Core	Max.1024 Threads, 32KB Shared Memory, 32684 Registers
Caches / Core	32KB 8-way L1 Data Cache, 8KB 4-way Texture, 8KB 4-way Constant Cache, 64B Line Size
L2 Unified Cache [36]	256 KB/Memory Partition, 64B Line Size, 16-way associative
Scheduling	Round Robin Warp Scheduling (Among Ready Warps), Load Balanced CTA Scheduling
Features	Memory Coalescing, 64 MSHRs/core, Immediate Post Dominator (Branch Divergence)
Interconnect [33]	1 Crossbar/Direction (SIMT Core Concentration = 3), 650MHz, Dimension-Order Routing, 16B Channel Width, 4VCs, Buffers/VC = 4, Routing Delay = 2, Channel Latency = 2, Input Speedup = 2
DRAM Model [37]	FR-FCFS (128 Request Queue Size/MC), 4B Bus width, 4 DRAM-banks/MC, 2KB page size, 4 Burst Size, 8 MCs
GDDR3 Timing [3], [9], [33]	800MHz, t_{CL} =10, t_{RP} =10, t_{RC} =35, t_{RAS} =25, t_{RCD} =12, t_{RRD} =8, t_{CDLR} =6, t_{WR} =11

Kernel, CTA, warp and thread scheduling: In GPGPUs, scheduling is typically a three-step process. First, a kernel of a GPGPU application is launched on the GPU. In our work, we assume that only one kernel is active at a time. After launching the kernel, the global block (CTA) scheduler (GigaThread in [31]) assigns CTAs of the launched kernel to all the available cores (We assume there are C cores in the system.) [3], [4]. The CTA assignment is done in a load-balanced round-robin fashion [4]. For example, CTA 1 is launched on core 1, CTA 2 is launched on core 2, and so on. If there are enough CTAs, each core is assigned with at least one CTA. Then, if a core is capable of executing multiple CTAs, a second round of assignment starts, if there are enough available CTAs. This process continues until all CTAs have been assigned or all the cores have been assigned with their maximum limit of CTAs. Assuming there are enough CTAs to schedule, the number of concurrently executing CTAs in the system is equal to $N \times C$. The maximum CTAs (N) per-core is limited by core resources (the total number of SIMT units, shared memory and register file size [4], [18], [21]), and cannot exceed the limit of 8 [29]. Given a baseline architecture, N may vary for a kernel depending on how much resource is needed by the CTAs of a particular kernel. For example, if a CTA of kernel X needs minimum 8KB of shared memory and the baseline architecture has 32KB available, only 4 CTAs of kernel X can be launched simultaneously on the same core. After the CTA assignment, the third step is the scheduling of warps associated with the launched CTA(s) on a core. The warps are scheduled in a round-robin fashion to the SIMT lanes. Every 4 cycles,

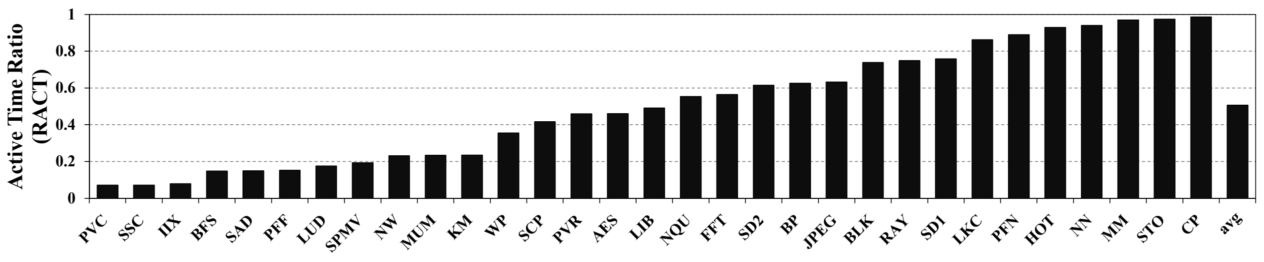


Fig. 3: The average Active Time Ratio (RACT) of applications. RACT is defined as the average of ratio of cycles during which a core can fetch new warps, to the total application execution cycles. The rightmost bar shows the arithmetic mean across all applications.

a ready warp (ready to fetch instruction(s)) is fed into these lanes for execution. If the progress of a warp is blocked on a long latency operation (e.g. waiting for data to come back from memory), the entire warp is scheduled out of the pipeline and put in the pending queue of warps. When the corresponding data arrives, the warp proceeds to the write-back stage of the pipeline and marked as ready to fetch new instructions.

Application suite: In this paper, we consider a wide range of CUDA applications targeted to evaluate the *general purpose* capabilities of GPUs. Our application suite includes CUDA NVIDIA SDK [30] applications, and Rodinia [6] benchmarks, which are mainly targeted for heterogeneous CPU-GPU-accelerator platforms. We also study Parboil [34] benchmarks, which mainly stress throughput-computing focused architectures. To evaluate the impact of our schemes on large scale and irregular applications, we also consider emerging MapReduce [14] and a few third party GPGPU applications. In total, we study 31 applications described in Table II. We evaluate our techniques on GPGPU-Sim v2.1.2b [4], a publicly-available cycle-accurate GPGPU simulator. We model the configuration described in Table I in GPGPU-Sim. Each application is run till completion or 1 billion instructions, whichever comes first.

III. Detailed Analysis of TLP

A. Is More Thread-Level Parallelism Better?

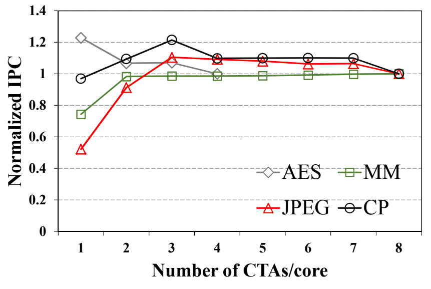
Although maximizing the number of concurrently executing CTAs on a core is an intuitive way to hide long memory latencies, Figure 1 shows that the applications exhibit sub-par performance with maximum TLP. For analyzing the primary cause of performance bottlenecks, we study the core active/inactive times for the benchmarks shown in Table II. We define the core active time ($Nact$), and the core inactive time ($Ninact$) as the number of cycles during which a core is able to fetch new warps, and is unable to fetch new warps, respectively. The average active time ratio ($RACT$) is defined as the average of $Nact/(Nact + Ninact)$ across all cores.

Figure 3 shows the applications in their increasing order of $RACTs$. We observe that, on average, cores are inactive for 49% of the total execution cycles, and this goes up to around 90% for memory-intensive applications (`PVC`, `SSC`, `IIX`). It is important to stress that these high percentages are observed even though the maximum possible number of CTAs are concurrently executing on all cores. These results portray a not-so-good picture of GPGPUs, which are known for demonstrating high TLP and delivering high throughput [11], [18], [19]. Inactivity at a core might happen mainly because of three reasons. First, all the warps could be waiting for the data from main memory and hence, their progress is road-blocked.

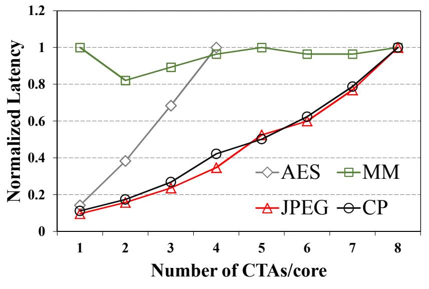
TABLE II: List of benchmarks: *Type-C*: Compute-intensive applications, *Type-M*: Memory-intensive applications, *Type-X*: Applications not exhibiting enough parallelism.

#	Suite	**Applications**	Abbr.	Type
1	MapReduce	Page View Count	PVC	Type-M
2	MapReduce	Similarity Score	SSC	Type-M
3	MapReduce	Inverted Index	IIX	Type-M
4	SDK	Breadth First Search	BFS	Type-M
5	Parboil	Sum of Abs. Differences	SAD	Type-M
6	Rodinia	Particle Filter (Float)	PFF	Type-M
7	Rodinia	LU Decomposition	LUD	Type-X
8	Parboil	Sparse-Matrix-Mul.	SPMV	Type-M
9	Rodinia	Needleman-Wunsch	NW	Type-X
10	SDK	MUMerGPU	MUM	Type-M
11	Rodinia	Kmeans	KM	Type-M
12	SDK	Weather Prediction	WP	Type-M
13	SDK	Scalar Product	SCP	Type-M
14	MapReduce	Page View Rank	PVR	Type-M
15	SDK	AES Cryptography	AES	Type-M
16	SDK	LIBOR Monte Carlo	LIB	Type-M
17	SDK	N-Queens Solver	NQU	Type-X
18	Parboil	FFT Algorithm	FFT	Type-M
19	Rodinia	SRAD2	SD2	Type-M
20	SDK	Backpropogation	BP	Type-M
21	SDK	JPEG Decoding	JPEG	Type-M
22	SDK	Blackscholes	BLK	Type-C
23	SDK	Ray Tracing	RAY	Type-C
24	Rodinia	SRAD1	SD1	Type-C
25	Rodinia	Leukocyte	LKC	Type-C
26	Rodinia	Particle Filter (Native)	PFN	Type-C
27	Rodinia	Hotspot	HOT	Type-C
28	SDK	Neural Networks	NN	Type-C
29	Parboil	Matrix Multiplication	MM	Type-C
30	SDK	StoreGPU	STO	Type-C
31	SDK	Coulombic Potential	CP	Type-C

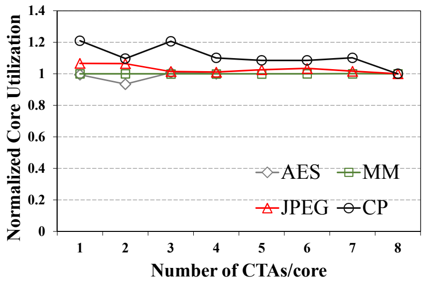
We call this as “memory waiting time”. Second, pipeline may be stalled because of excessive write-back (WB) contention at the WB stage of the pipeline, which we call as “stall time”. This may happen when the data associated with multiple warps arrive in a short period of time and proceed to the WB stage. This leads to the stalling of the pipeline for multiple cycles, preventing new warps from being fetched. Third contributor is the “idle time”, which is the number of cycles during which the core cannot fetch any warps as all the warps in the core have finished their execution. Synchronization stalls, which are modeled in our studies, are also considered a part of idle time. Through empirical analysis, we observe that the first two components are the major contributors for the core inactivity, whereas the third component constitutes more than 10% of the total execution cycles for only three applications. As GPGPU memory bandwidth is limited and will continue to be a bottleneck with increasing number of cores, the warps will tend to wait longer periods of time for the requested data to come back from DRAM and cause the pipeline to stall. These inactive periods will continue to grow as more highly memory-intensive applications are ported to GPGPUs.



(a) Effect on IPC



(b) Effect on lat_{rt}



(c) Effect on $util$

Fig. 4: The effect of increase in number of CTA on various metrics. Higher TLP increases lat_{rt}. Higher TLP causes $util$ to go down on some applications. lat_{rt} and $util$ have significant impact on IPC.

To analyze the reason for inactivity, we start by classifying the applications into three categories. First, applications with significant idle time ratio ($> 20\%$) are categorized as Type-X applications. Type-X applications do not exhibit high level of parallelism, utilize only a few cores, thus are not very suitable for parallel computing. Among the remaining applications, the ones with high $RACT$ ($> 66\%$) are categorized as compute-intensive applications (Type-C), and the ones with low $RACT$ ($\leq 66\%$) are classified as memory-intensive (Type-M) applications. We observe that, in many memory-intensive applications, increasing the number of CTAs has detrimental impact on performance. It is important to note that executing more CTAs concurrently leads to more warps to timeshare the same set of computing resources. Also, the number of memory requests sent simultaneously escalates in proportion, thereby increasing memory access latencies [24]. Moreover, this increase in TLP stresses the limited DRAM bandwidth even more.

Let us now consider three GPGPU applications with varied properties (`AES`, `MM`, `JPEG`) and observe their behavior as the number of CTAs/core (parallelism) is increased (Figure 4). Discussion of the fourth application (`CP`) in this figure is in Section III-C. We mainly focus on IPC, round-trip fetch latency (lat_{rt}), and core utilization ($util$) of the GPGPU system. lat_{rt} is defined as the number of core clock cycles between which a memory request leaves the core and comes back to the core. $util$ is defined as the average of $util_i$ for all i, where $util_i$ is the ratio of cycles when at least one CTA is available on core i, to the total execution cycles. Note that, in a GPGPU system with C cores, the maximum number of CTAs that can concurrently execute on the GPGPU is $N \times C$. In this experiment, we vary the limit of the number of CTAs launched on a core, n, from 1 to N. In turn, we increase parallelism from $C \times 1$ to $C \times N$, in steps of 1 CTA per core (C CTAs per GPGPU system). The results for these three metrics are *normalized* with respect to their values when N CTAs are launched on the core.

Figures 4a, 4b, and 4c show the effect of varying TLP (at the granularity of CTAs) on IPC, latency, and core utilization, respectively. For `AES`, the results are normalized to the case where $N = 4$, as maximum 4 CTAs can execute concurrently according to the resource restrictions. We observe that in `AES`, increasing the number of concurrent CTAs from $n = 1$ to $n = N$ ($N = 4$) has detrimental impact on lat_{rt} (increases by $9\times$). Since `AES` is classified as a Type-M application (54% core inactive time), as the number of concurrent CTAs increases, the number of memory requests escalates, causing more contention in memory sub-system. We notice that $n = 1$

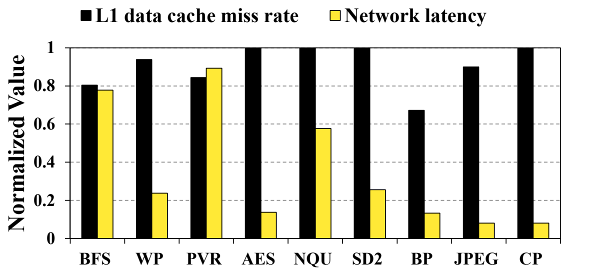
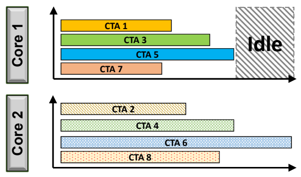


Fig. 5: The effect of TLP on L1 data cache miss rate and network latency. The first and the second bars show the normalized L1 data cache miss rate and the normalized network latency when TLP is minimum, respectively, with respect to the default CUDA approach.

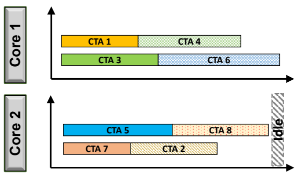
(minimum CTA count per core, minimum TLP) leads to the highest performance (lowest lat_{rt} and the highest $util$) for this application. Thus, for `AES`, we define optimal CTA count (opt) as 1. On the other hand, for `MM`, which is a Type-C application, as we increase the number of CTAs from $n = 1$ to $n = 8$, IPC improves at each step, but less steeply after $n = 2$. The total IPC improvement is around 34%. Since this application is compute-intensive, varying TLP does not have a significant impact on lat_{rt}. Note that the utilization problem is not significant in this application and CTA load is balanced across all cores. In `MM`, we have $opt = 8$. These two applications demonstrate the impact of exploiting the maximum and minimum TLP available. In comparison, `JPEG` exhibits variances in IPC, latency and utilization as TLP is increased. In this application, when parallelism is low ($n = 1$ and $n = 2$), the ability to hide the memory latency is limited, resulting in lower performance compared to when TLP is higher ($n = 3$). However, as the number of CTAs increases beyond $n = 3$, the increase in TLP leads to an increase in the memory access latency, leading to a loss in performance. Hence, we define $n = 3$ as the optimal CTA count for `JPEG`.

B. Primary Sources of Contention

As discussed in Section III-A, we observe that memory fetch latency can be affected by the level of parallelism. We identify that the increase in memory latency is primarily attributed to three reasons. First, the cache miss rates increase with increasing TLP. High TLP causes the working data set to be larger, which causes more cache blocks that are spatially and temporally far from each other to be brought to cache [16]. This increases the cache contention and leads to higher miss rates. Our simulation results show that, the average L1 data cache miss rate of 31 applications is 62% when TLP is



(a) A case where $n = 4$



(b) A case where $n = 2$

Fig. 6: The effect of TLP on core utilization. In (a), all 8 CTAs are distributed to 2 cores initially. Core 1 finishes execution before Core 2 and we observe significant idle time for Core 1. In (b), only 4 CTAs are distributed to 2 cores initially. Once a core finishes executing one CTA, it is assigned with another one. Since only 2 CTAs share the same core resources instead of 4, execution times of CTAs become shorter compared to (a). In (b), we see better core utilization and shorter execution time due to better CTA load balance. Note that this example is for illustrative purposes only.

minimum (cores execute only 1 CTA at a time), and 72% when TLP is maximum (default CUDA approach). Second, increasing TLP causes network to be congested, and network latencies of request and reply packets increase. The number of warps that request data increase with higher TLP, causing more network injections. Average network latency increases by $2.1\times$ when TLP is changed from its minimum value to the CUDA default. Third, higher number of memory requests leads to longer queuing delays at MCs. Average DRAM queuing latency goes up by 7% when TLP is maximized. The comparison of minimum and maximum TLP in terms of L1 cache and network performance is given in Figure 5.

C. Implications of Tuning TLP on Core Utilization

Next, we observe that varying the number of CTAs has also an impact on the core utilization. This is mainly contributed by the variation of execution times of different CTAs, causing an imbalance on the execution times of cores [32]. One of the potential reasons for this is the fact that having more CTAs on cores might increase the time during which some cores are idle towards the end of execution (once they finish executing all the CTAs they have), while others are still executing some CTAs. Although it is not always the case, increasing n tends to worsen the utilization problem. Thus, this problem is mostly evident on applications with high N. Figure 6 depicts an example illustrating this problem. We assume that there are 2 cores executing 8 CTAs. The y-axis shows the number of concurrently executing CTAs, and the x-axis shows the execution time. This example shows the impact of TLP on core utilization, and how it is related to execution time.

To understand the effect of core utilization on performance, we pick `CP`, the most compute-intensive application in our suite (99% RACT), and observe the impact of varying n. Since `CP` is fetching a warp without spending much inactive time, one would expect to see an increase in performance with increasing n. The effect of varying TLP on IPC, lat_{rt} and $util$ is plotted in Figure 4. As we increase n, we see that lat_{rt} also increases linearly. However, since lat_{rt} is low (32 cycles when $n = 1$), and the benchmark is highly compute-intensive, increasing lat_{rt} does not have a negative impact on performance. Except when $n = 1$ where memory latency tolerance is limited due to low TLP, the trend of IPC always follows the trend of $util$. As expected, $util$ goes down as n approaches N, and IPC drops due to unbalanced CTA load distribution. $util$, hence IPC, reach their peaks when $n = 3$.

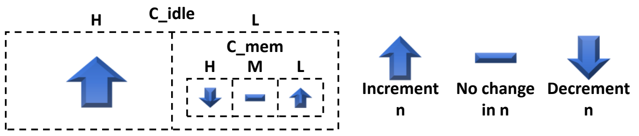


Fig. 7: The overview of DYNCTA algorithm. If C_idle is high, n is incremented. Otherwise, if C_mem is low, n is incremented; if it is high, n is decremented. Otherwise n is not changed.

To summarize, we observe that, out of 31 applications evaluated, 15 of them provided more than 5%, and 12 of them yielded more than 10% (up to $4.9\times$ for `IIX`) better performance with optimal TLP, compared to the baseline. Thus, increasing TLP beyond a certain point has a detrimental effect on the memory system, and thus on IPC.

IV. Proposed CTA Modulation

In this section, we describe our approach to determine the optimal CTA number on a core dynamically, and discuss its implications on performance. We apply our approach to each core separately, and consequently, each core can work with a different CTA count at a given time.

Problem with finding opt**:** One way of finding opt for an application is to run it exhaustively for all possible n values, and determine the one that gives the shortest execution time. This could be a viable option only if we are running a few predetermined applications all the time. Instead, we would like our approach to be applicable under all circumstances. Thus, we propose a scheme which changes the number of CTAs on a core *dynamically* during the course of execution.

Idea for dynamic CTA modulation: When an application exhibits a memory-intensive behavior during a phase, we limit TLP by reducing the number of CTAs on the cores in order to reduce the cache, network and memory contention. On the other hand, when an application is compute-intensive, we would like to exploit TLP by increasing the number of CTAs in order to improve latency tolerance. Below, we describe our approach to determine whether an application is memory or compute-intensive during a phase.

Monitored metrics: In order to modulate n during run-time, we monitor the following metrics: (1) *C_idle*, and (2) *C_mem*. *C_idle* is the number of core cycles during which the pipeline is not stalled, but there are no threads to be issued on the core, and thus this core is idle. A very high *C_idle* value indicates that the core does not have enough threads to keep the pipeline busy, thus it might be beneficial to increase TLP. *C_mem* is the number of core cycles during which all the warps are waiting for their data to come back. This metric indicates how much pressure is being exerted on the memory. A very high *C_mem* indicates that the cores are waiting for very long latency memory operations, thus it is better to limit TLP. Statistics for these two metrics are collected separately on each core, and a CTA modulation decision is made locally.

How the algorithm works: Instead of assigning N CTAs to each core, the CTA scheduler starts with assigning $\lfloor N/2 \rfloor$ CTAs to each core ($n = \lfloor N/2 \rfloor$), and distributes the CTAs to cores in a round-robin fashion. Therefore, even if there are less than $C \times \lfloor N/2 \rfloor$ CTAs in the application kernel, the difference between the number of CTAs on any two cores cannot be greater than 1, where C denotes the total number of cores in the system. After this initialization step, at each core, *C_mem* and *C_idle* are checked periodically to make a decision, and then reset to 0. For *C_idle*, we use a threshold, *t_idle*, that categorizes the value as *low*, or *high*. For *C_mem*, we use a low threshold (*t_mem_l*), and a high threshold (*t_mem_h*) to categorize the value as low, medium, or high.

Figure 7 shows how the number of CTAs is modulated. The key idea behind the CTA modulation algorithm is to keep the cores utilized, but not with too much work, so that the performance does not suffer due to contention in the memory sub-system. First, *C_idle* is checked to monitor if the core is utilized. If it is high ($> t\_idle$), a new CTA is assigned to the core. The reason for this is to make an otherwise idle core busy. Second, if *C_idle* is low ($< t\_idle$), we check *C_mem* to optimize the workload on the core. If *C_mem* is low, the warps do not wait for a long time for their requests to come back from the memory. Thus, we can increase the level of parallelism and assign one more CTA to the core. If *C_mem* is high, then the warps are waiting for a long time for their requests to come back from the memory. This implies that lat_{rt} has grown too large to hide the memory latency, thus we decrement n by 1. Otherwise, n remains the same. Note that the decisions are made locally at each core. Once a decision is made, *C_mem* and *C_idle* are reset to 0 to capture the behavior of the next window. Note that $1 \leq n \leq N$ must always hold, as long as there is a CTA that is available to be issued to the core. We ensure that $n \geq 1$ so that the cores are executing threads, instead of staying idle. If there are no CTAs available to be assigned to the core, then $n \leq N$ must hold. These two conditions are always checked when making a decision whether to increase or decrease n. If the resulting n after the decision violates these conditions, then n is not changed.

CTA pausing: CUDA blocks (CTAs), once assigned to a core, *cannot* be preempted, or assigned to another core [29]. This presents a problem when the algorithm decrements n. In order to address this problem, we propose a technique called CTA pausing. This technique deprioritizes the warps belonging to the most recently assigned CTA on the core, if n needs to be decremented by 1. In this case, we say that the CTA is paused. If n needs to be decremented further, the warps belonging to the second most recently assigned CTA on the core are also deprioritized. However, employing CTA pausing has implications on incrementing n. If n is incremented during a time in which a paused CTA is present on the core, a new CTA is not issued to the core. Instead, the most recently paused CTA resumes its execution. The pseudo-code of DYNCTA with CTA pausing is given in Algorithm 1.

To explain the behavior of our algorithm with CTA pausing, let us consider an example where $N = 4$ and $n = 3$ at a given instant. Let us further assume that CTA1 is the oldest CTA issued to the core, and CTA3 is the most recently assigned CTA on the core. If the algorithm decrements n by 1, the warps that belong to CTA3 are deprioritized. A warp of CTA3 can be fetched only if there is not a ready warp that belongs to CTA1 or CTA2. Let us now assume that n is decremented further. This time, CTA2 is deprioritized. If there are ready warps that belong to CTA1, they will have the priority to be fetched. If n is incremented by 1, CTA2 again gets the priority to have its warps fetched. If n is incremented further, all warps have the same priority. To summarize, a CTA is paused when n is decremented. It can resume execution only when another CTA finishes its execution, or n is incremented.

CTA pausing might not be effective when an application phase is highly memory-intensive. If memory fetch latencies are high, paused CTAs would eventually be executed before higher-priority CTAs fetch their data and resume execution.

Comparison against *opt***:** We observe that some applications can have high and low RACT values at different time intervals. For such applications, *opt* might be different for intervals showing different behaviors, and our algorithm potentially can outperform the case where $n = opt$. As discussed in Section III, the level of parallelism mainly affects lat_{rt} and $util$. The problem related to $util$ manifests itself towards the end of kernel execution, as illustrated in Figure 6. For this reason, our algorithm aims to solve the problem caused by lat_{rt}. Since the utilization problem is usually more pronounced when n is large (Section III-C), and our algorithm limits TLP by limiting n, we indirectly mitigate the effects of the utilization problem as we limit n for some applications. For some applications, *opt* is dependent on $util$, as explained in Section III-C. Thus, our algorithm may not be able to match the performance of the $n = opt$ case. For example, in `CP`, which is the most compute-intensive benchmark, we have $opt = 3$ and $N = 8$. The reason why *opt* is equal to 3 is explained in Section III-C. Since `CP` is very compute-intensive, our algorithm eventually sets n to 8, thus fails to converge to *opt*. We believe that addressing utilization in order to more closely approach optimal TLP requires predicting CTA execution times on cores. This execution time prediction requires an accurate performance model, which is not considered in this work. Although there are cases where the algorithm fails to

Algorithm 1 DYNCTA: Dynamic Cooperative Thread Array Scheduling

```
                                  ▷ $N$ is the maximum # of CTAs on a core
                                  ▷ $n$ is the CTA limit (running CTAs) on a core
  ▷ $nCTA$ is the total number of CTAs (paused and running CTAs) on a core
           ▷ Issue_CTAs_To_Core($n$):Default CTA scheduler with CTA limit=$n$

procedure INITIALIZE
    $n \leftarrow \lfloor N/2 \rfloor$
    ISSUE_CTAS_TO_CORE($n$)

procedure DYNCTA
    for all cores do
        INITIALIZE
    for each sampling period do
        for all cores do
            if $C\_idle \geq t\_idle$ then
                if $nCTA > n$ then
                    Unpause least recently assigned CTA
                else if $n < N$ then
                    $n \leftarrow n + 1$
            else if $(C\_mem < t\_mem\_l)$ then
                if $nCTA > n$ then
                    Unpause least recently assigned CTA
                else if $n < N$ then
                    $n \leftarrow n + 1$
            else if $(C\_mem \geq t\_mem\_h)$ then
                if $n > 1$ then
                    $n \leftarrow n - 1$
            for $i = 0 \rightarrow (nCTA - n)$ do
                Pause most recently assigned CTA
```

TABLE III: Variables and thresholds

Variable	Description
$Nact$	Active time, where cores can fetch new warps
$Ninact$	Inactive time, where cores cannot fetch new warps
$RACT$	Active time ratio, $Nact/(Nact + Ninact)$
C_idle	The number of core cycles during which the pipeline is not stalled, but there are no threads to execute
C_mem	The number of core cycles during which all the warps are waiting for their data to come back
t_idle	Threshold that determines whether C_idle is low or high
t_mem_l & t_mem_h	Thresholds that determine if C_mem is low, medium or high

converge to *opt* (mostly for Type-C benchmarks which suffer from low core utilization), the algorithm usually converges to a value that is close to *opt*.

Parameters: Our algorithm depends on parameters such as *t_idle*, *t_mem_l*, and *t_mem_h*. We experimentally determined the values of these parameters to be 16, 128, and 384, respectively. These values are micro-architecture dependent, and need to be recalculated for different configurations. Another parameter is the sampling period (2048 cycles). Sensitivity analysis on these parameters are reported in Section V-B. All variables and thresholds are described in Table III.

Initial value of n: As described above, all cores are initialized with $\lfloor N/2 \rfloor$ CTAs provided that there are enough CTAs. We also tested our algorithm with initial n values of 1 and N. Starting with $n = 1$ gave very similar results to starting with $\lfloor N/2 \rfloor$, and n converged to approximately the same value. However, starting with N did not yield as good results as starting with 1 or $\lfloor N/2 \rfloor$. This is a limitation of our algorithm, since all possible CTAs are already distributed to the cores initially. However, according to our observations, starting with a small initial n does not make a significant difference as the algorithm converges to the same value eventually.

Synchronization: We model atomic instructions and take into account latency overheads due to inter-CTA synchronization. We make sure that we do not deprioritize CTAs indefinitely to prevent livelocks and starvation. Deprioritizing CTAs indefinitely would improve the performance of the system further, however such design requires a lock-aware mechanism. Also, we have not observed performance losses due to CTA-pausing.

Hardware overhead: Each core needs two 11-bit counters to store *C_mem*, and *C_idle*. The increment signals for each counter come from the pipeline. The contents of these counters are compared against three pre-defined threshold values, and the outcome of our CTA modulation decision is communicated to the global CTA scheduler. Since the cores and the scheduler already communicate every clock cycle (e.g., as soon as a core finishes the execution of a CTA, global scheduler issues another CTA), there is no extra communication overhead required. We implemented DYNCTA in RTL using Verilog HDL, and synthesized the design using Synopsys Design Compiler [35] on 65 nm TSMC libraries. We found that DYNCTA occupies 0.0014 mm^2 on each core.

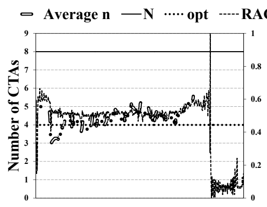
V. Experimental Results

We evaluate DYNCTA with 31 applications, using the configuration described in Table I. We use the default CUDA approach for determining the number of concurrent CTAs.

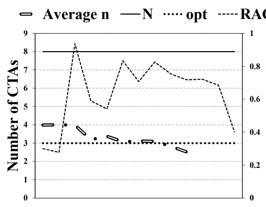
A. Performance Results with DYNCTA

We start by showing the dynamism of our algorithm in allocating CTAs to the cores. For each application, we study the maximum number of CTAs that can be allocated to the cores, the optimal number of CTAs determined by exhaustive analysis, and how our algorithm modulates the number of CTAs. Figure 8 shows how the number of CTAs changes during the execution for four applications. In `SAD` ($N = 8$), we start with $n = 4$. Initially, due to high RACT, the average of n across all cores goes up to 5, but then fluctuates around 4. For this application, we have $opt = 4$, and average of n across cores is around 4.3. Note that we do not show average n beyond the point where all the CTAs are issued to the cores, since there is nothing to modulate. In `JPEG` ($N = 8$), initial value of n is 4. Beyond the point where RACT almost reaches 1, n slowly converges to 3, which is equal to *opt* for this application. Since this application is fairly compute-intensive in the middle of its execution, no CTAs are paused. Thus, DYNCTA effectively optimizes TLP of this application. In `RAY`, we have $N = 6$, and start with $n = 3$. The application is initially not compute intensive, but most of the inactivity happens due to write-back contention, not memory waiting time. Thus, n is not decremented. After $RACT$ goes beyond 0.8, n follows the trend of $RACT$, and eventually reaches N. Even though this is a Type-C application, *opt* is not equal to N due to the core utilization problem (see in Section III-C). Due to this, although DYNCTA correctly captures the application behavior, n fails to converge to *opt*. In `FFT`, we have $N = 8$ and therefore start with $n = 4$. Since this application's behavior changes very frequently, average n also fluctuates. Overall, we observe that average n stays very close to *opt* and DYNCTA is successful in capturing the application behavior. A graph showing N, average n and *opt* for all applications is given in Figure 9. We see that DYNCTA is close to *opt* for most Type-M applications. Type-C applications suffering from core utilization problem such as `CP` and `RAY` fail to reach the optimal point. Type-X applications such as `NW` and `LUD` do not have enough threads to be modulated, so the algorithm fails to reach the optimal point and they do not benefit from DYNCTA. Overall, average N is 5.38 and average *opt* is 2.93. With DYNCTA, we get very close to *opt*, obtaining an average of 2.69 CTAs across 31 applications.

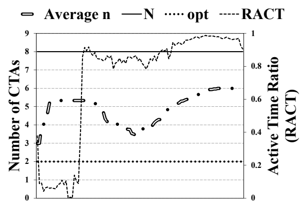
Figure 10 shows the performance improvements obtained by DYNCTA. Across 31 applications, we observe an average speedup of 28% (18% geometric mean (GMN)). This result is close to the improvements we can get if we use $n =$ *opt* for each application (39% mean, 25% GMN). Most Type-M applications such as `IIX` (2.9×), `PVC` (3.6×), `SAD` (2.9×), and `KM` (1.9×) benefit significantly from DYNCTA. Some Type-M applications such as `WP` and `LIB` do not have any room for improvement since the number of CTAs available on the cores is not high enough for modulation. For example, `LIB` has 64 CTAs. Although we have $N = 8$, the cores cannot get more than 3 CTAs according to load balanced CTA assignment in a 30-core system. `PFF` has $N = 2$, and does not have much room for improvement since we can only modulate between either 1 or 2 CTAs per core. We do not get improvements from Type-X applications (`NW`, `LUD`, and `NQU`), since they do not exhibit enough parallelism and have very few threads. Also, Type-C applications do not benefit from DYNCTA, except for `RAY` and `CP`. They gain improvements



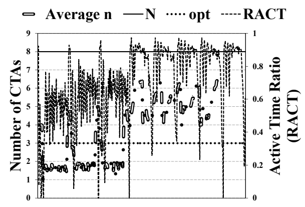
(a) SAD



(b) JPEG



(c) RAY



(d) FFT

Fig. 8: CTA modulation over time. Except RAY, which is Type-C and suffering from low $util$, DYNCTA is able to modulate TLP accurately.

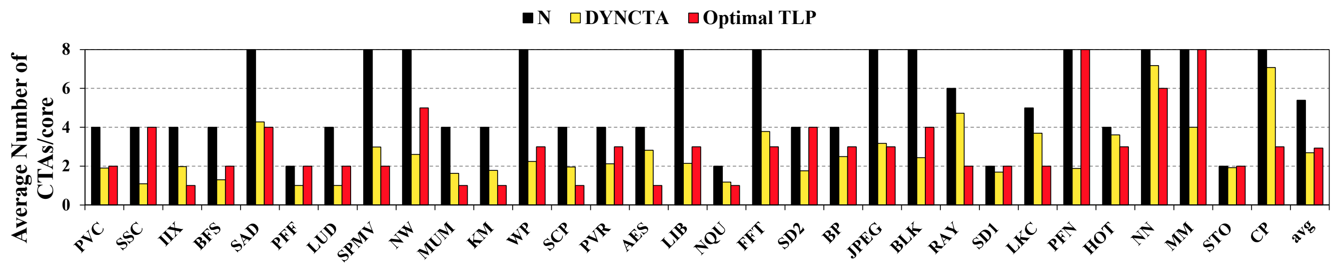


Fig. 9: Average number of CTAs assigned to the cores, with the default CUDA approach, DYNCTA, and the optimal TLP.

due to mitigating the effects of low core utilization, although the improvements for RAY are far from reaching optimal TLP results. There are 4 applications that lose more than 2% performance with DYNCTA. The performance reductions in NW, which is Type-X, and BLK are around 2%. DYNCTA degrades the performance of PFF by 2.5%, and SCP by 3%.

Although more evident in Type-C applications, almost all applications suffer from $util$ towards the end of kernel execution. DYNCTA does not aim to optimize TLP according to $util$, which is an important factor in determining the optimal TLP. Thus, DYNCTA fails to outperform optimal TLP in most cases. Among 31 applications, 10 applications perform at least as good as optimal TLP. DYNCTA outperforms optimal TLP by more than 2% in BFS, SD1, STO, and PVC. We observe a $4\times$ reduction in lat_{rt} for STO. DYNCTA reduces the memory waiting time of BFS by $2.2\times$. SD1 benefits from reductions in both memory waiting time and lat_{rt}.

We also compared DYNCTA against the two-level scheduler (TL) proposed by Narasiman et al. [28]. Our experiments show that DYNCTA outperforms TL by 12.5%, on average. TL yields significantly higher performance than DYNCTA for RAY and BLK. TL sends the memory requests in groups, instead of sending them at once. This approach allows cores to receive data in groups as well, reducing the write-back contention. In BLK, this proves effective, and even though the memory waiting time is less in DYNCTA, TL manages write-back better, and shows better performance, even outperforming optimal TLP. RAY performs better with TL because the load distribution across cores becomes more balanced compared to DYNCTA, due to the similar reasons explained earlier in this section. Note that TL is a warp scheduling policy, and DYNCTA is a CTA scheduling policy, and they are independent of each other. In fact, these two schemes can be used in tandem to boost GPGPU performance further.

Since DYNCTA mainly targets Type-M applications by reducing DRAM contention, we expect to observe lower lat_{rt}. Although we have shown the effects of $util$ on IPC for some applications, DYNCTA does not directly aim to improve $util$. Figure 11 plots the impact of DYNCTA on lat_{rt} and $util$. DYNCTA provides a better load balancing in NQU, CP, and RAY, and increases $util$ by 28%, 19%, and 12%, respectively. For the rest of the applications, $util$ does not change significantly. We observe that lat_{rt} drops for most applications, which is in line with their IPC improvements. Most Type-C applications also have lower lat_{rt}, but since they are compute-intensive, change in lat_{rt} does not translate to change in IPC. The average reduction in lat_{rt} is 33% across all applications. DYNCTA effectively reduces lat_{rt} while keeping the cores busy. Since GPUs are throughput processors and most GPGPU applications are not latency sensitive, the improvement in lat_{rt} might not necessarily translate into performance benefits, whereas RACT would be a better metric reflecting the effectiveness of DYNCTA. We observe that DYNCTA improves RACT by 14% on average. Note that RACT is the average of per-core active cycle ratios thus, does not take CTA imbalance effects into account. By partially balancing CTA load, we gain extra performance on top of better RACT.

Since limiting the number of CTAs reduces the working set of the application in the memory, cache contention reduces and hit rates improve. On average, L1 miss rates reduce from 71% to 64%. These results show that the increasing hit rates contributes to the performance of DYNCTA.

B. *Sensitivity Analysis*

We conducted a sensitivity analysis to demonstrate the performance of DYNCTA on larger systems. Since crossbar is not scalable for large systems, we conducted sensitivity analysis using a 2-D mesh interconnect. As we increase the number of cores (C) in the system, we expect more DRAM contention, which would increase the benefits of our schemes. On the other hand, increasing C would limit the working region of DYNCTA, since there will be fewer CTAs assigned to the cores, limiting the benefits of our schemes. We used two different configurations: (1) a 56 core system with 8 MCs (8×8

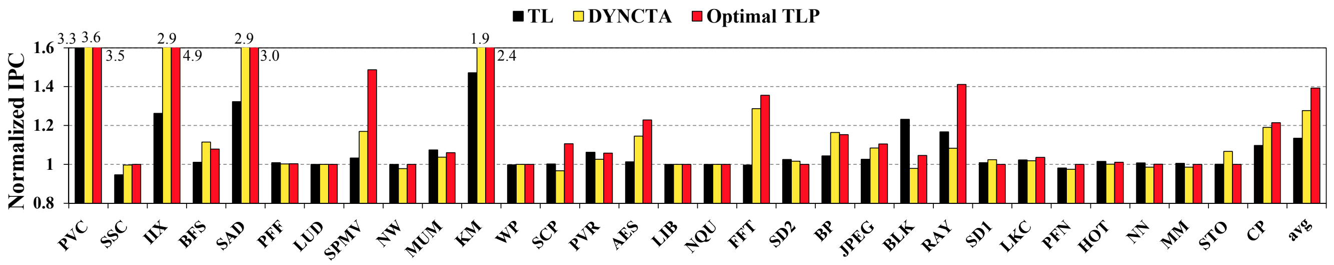


Fig. 10: IPC improvements by DYNCTA. The first, second and third bars show the normalized IPC of two-level scheduler [28], DYNCTA, and optimal TLP ($n = opt$), respectively, with respect to the default CUDA approach.

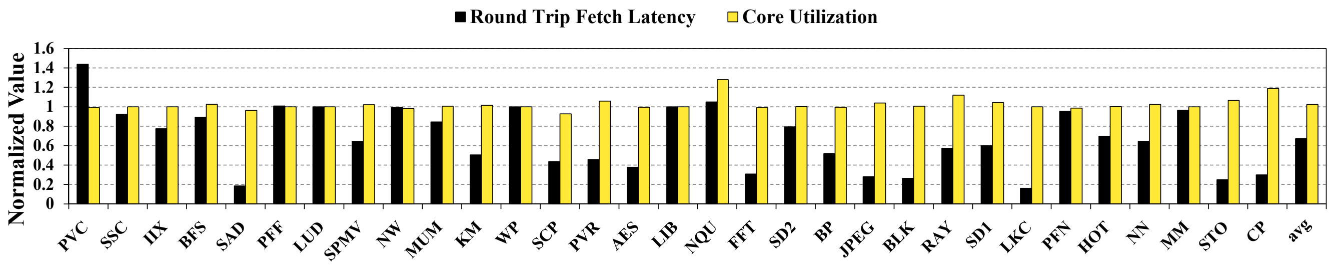


Fig. 11: Effects of DYNCTA on round-trip fetch latency (lat_{rt}), and core utilization ($util$), w.r.t. CUDA default.

mesh), and (2) a 110 core system with 11 MCs (11×11 mesh). We observe that, DYNCTA is effective in both configurations, providing average performance improvements around 20%.

As DYNCTA mostly helps Type-M applications, the number of memory requests sent from a core is an important factor that can affect its benefits. To evaluate this, we varied the size of MSHRs per core and observed that changing MSHR/core from 64 to 32 and 16 degrades the average performance by 0.3% and 0.6%, respectively. We also examined the impact of memory frequency on DYNCTA by evaluating the system with 1107 MHz [3] and 1333 MHz GDDR3, and observed up to only 1% reduction in our benefits. Slight performance loss is expected, since DYNCTA aims to improve memory bandwidth. However, memory bandwidth is the main bottleneck in GPUs, and projected to be even more so in the future [20]. Thus, we believe that our schemes will be applicable to future GPUs.

We also conducted a sensitivity analysis on the parameters. Changing the sampling period from 2048 to 4096 cycles degraded our benefits by 0.1%. We also varied the thresholds (*t_idle*, *t_mem_l*, and *t_mem_h*) between 50% and 150% of their default values (given in Section IV), and observed losses between 0.7% and 1.6%. Thus, DYNCTA can work with almost equal efficiency with a broad range of threshold values.

VI. Related Work

To the best of our knowledge, this is the first paper that proposes a simple and a low-overhead architectural solution to dynamically optimize TLP for holistically improving the efficiency of the entire memory-subsystem of GPGPUs. In this section, we briefly describe the closely related works.

Optimizing thread-level parallelism: Rogers et al. [33] propose a cache conscious scheduling scheme, which dynamically varies TLP to reduce cache contention. Our work takes a holistic view, with the intention of improving the efficiency of the whole memory sub-system. As shown in the paper, the proposed dynamic CTA allocation strategy does not only reduce contention in caches, but also in DRAM and interconnect. Bakhoda et al. [4] also show the benefits of lowering the number of concurrently CTAs below the default CTA limit. Our work takes a further step ahead, and dynamically calculates the optimal number of CTAs that should be executed concurrently. Hong et al. [15] develop an analytical model to predict the execution time of GPUs. This model considers available TLP and memory intensiveness of an application. However, in our paper, we propose a dynamic technique that monitors the changing behavior of an application and calculates the optimal TLP for GPGPUs in terms of number of CTAs. Assigning work to the cores at the granularity of CTAs also allows to take advantage of the data locality present among warps belonging to the same CTA [17]. In the context of CMPs, Chadha et al. [5] also propose a dynamic TLP management scheme. An analytical model showing the effects of the number of concurrent threads on shared caches was proposed by Chen et al. [7].

Scheduling techniques in GPUs: Various warp scheduling techniques have been proposed to reduce cache contention and improve DRAM bandwidth. The two-level warp scheduler proposed by Narasiman et al. [28] splits the concurrently executing warps into groups to improve memory latency tolerance. We have already provided quantitative comparisons of our proposal with the two-level scheduler. Gebhart and Johnson et al. [12] propose a two-level warp scheduling technique that aims to reduce energy consumption in GPUs. Jog et al. [19] propose OWL, a series of CTA-aware warp scheduling techniques to reduce cache contention and improve DRAM performance for bandwidth-limited GPGPU applications. Jog et al. [18] propose a prefetch-aware warp scheduling policy, which effectively coordinates with a prefetcher for further improving the memory latency tolerance in GPGPUs. Several memory [2], [23] and application [1] scheduling techniques have also been proposed. All these schedulers assume a *given* amount of TLP. Our approach is complementary to all the above works, as our dynamic strategy first calculates optimal

TLP and then later the above mentioned schedulers can be implemented for better performance.

VII. Conclusions

Enhancing the performance of applications through abundant TLP is one of the primary differences of GPGPUs compared to CPUs. Current GPGPU schedulers attempt to allocate the maximum number of CTAs per core to maximize TLP and enhance performance by hiding the memory latency of a thread by executing another. However, in this paper, we show that executing the maximum number of CTAs per core is not always the best solution to boost performance. The main reason behind this is the long round-trip fetch latencies primarily attributed to high number of memory requests generated by executing more threads concurrently. Instead of effectively hiding long memory latencies, executing high number of threads concurrently might degrade system performance due to increasing cache, network and memory contention.

The main contribution of this paper is a dynamic CTA scheduling algorithm for GPGPUs, which attempts to allocate optimal number of CTAs per-core based on application demands, in order to reduce contention in the memory subsystem. The proposed DYNCTA scheme uses two periodically monitored metrics, *C_idle* and *C_mem*, to allocate fewer CTAs to applications suffering from high resource contention, and more CTAs to applications that provide high throughput.

Experimental evaluations show that DYNCTA enhances application performance on average by 28% (up to 3.6×) compared to the default CTA allocation strategy, and is close to the optimal static allocation, which is shown to provide 39% performance improvement. We conclude that optimizing TLP via CTA scheduling can be an effective way of improving GPGPU performance by reducing resource contention.

Acknowledgments

We thank the anonymous reviewers, Nachiappan Chidambaram Nachiappan, Emre Kültürsay, Bikash Sharma, and Mahshid Sedghy for their feedback on earlier drafts of this paper. This research is supported in part by NSF grants #1213052, #1152479, #1147388, #1139023, #1017882, #0963839, #0811687, #0953246, and grants from Intel and Microsoft.

References

[1] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The Case for GPGPU Spatial Multitasking," in *HPCA*, 2012.

[2] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Prformance and Scalability in Heterogeneous Systems," in *ISCA*, 2012.

[3] A. Bakhoda, J. Kim, and T. Aamodt, "Throughput-Effective On-Chip Networks for Manycore Accelerators," in *MICRO*, 2010.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[5] G. Chadha, S. Mahlke, and S. Narayanasamy, "When Less is More (LIMO):Controlled Parallelism for Improved Efficiency," in *CASES*, 2012.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IISWC*, 2009.

[7] X. Chen and T. Aamodt, "Modeling Cache Contention and Throughput of Multiprogrammed Manycore Processors," *Computers, IEEE Transactions on*, vol. 61, no. 7, pp. 913 –927, July 2012.

[8] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *SC*, 2004.

[9] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *MICRO*, 2011.

[10] W. Fung and T. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *HPCA*, 2011.

[11] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *MICRO*, 2007.

[12] M. Gebhart, D. R. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *ISCA*, 2011.

[13] T. Halfhill, "AMD's Fusion Finally Arrives," *Microprocessor Report*, 2010.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce Framework on Graphics Processors," in *PACT*, 2008.

[15] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *ISCA*, 2009.

[16] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez, " Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems ," in *HPCA*, 2012.

[17] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and Improving the Use of Demand-fetched Caches in GPUs," in *ICS*, 2012.

[18] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated Scheduling and Prefetching for GPGPUs," in *ISCA*, 2013.

[19] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *ASPLOS*, 2013.

[20] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the Future of Parallel Computing," *Micro, IEEE*, vol. 31, no. 5, pp. 7 –17, Sept.-Oct. 2011.

[21] D. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[22] K. Krewell, "NVIDIA Lowers the Heat on Kepler," *Microprocessor Report*, 2012.

[23] N. B. Lakshminarayana, J. Lee, H. Kim, and J. Shin, "DRAM Scheduling Policy for GPGPU Architectures Based on a Potential Function," *Computer Architecture Letters*, 2012.

[24] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," in *MICRO*, 2010.

[25] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," in *PPoPP*, 2009.

[26] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *Micro, IEEE*, vol. 28, no. 2, 2008.

[27] A. Munshi, "The OpenCL Specification," June 2011.

[28] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *MICRO*, 2011.

[29] Nvidia, "CUDA C Programming Guide," Oct. 2010.

[30] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples

[31] Nvidia, "Fermi: NVIDIA's Next Generation CUDA Compute Architecture," Nov. 2011.

[32] M. Rhu and M. Erez, "CAPRI: Prediction of Compaction-Adequacy for Handling Control-Divergence in GPGPU Architectures," in *ISCA*, 2012.

[33] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *MICRO*, 2012.

[34] J. A. Stratton *et al.*, "Scheduler-Based Prefetching for Multilevel Memories," University of Illinois, at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[35] *Design Compiler*, Synopsys Inc. [Online]. Available: http://www.synopsys.com

[36] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU Microarchitecture Through Microbenchmarking," in *ISPASS*, 2010.

[37] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures," in *MICRO*, 2009.