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

Technical Report CSE-12-006

September, 2012

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

Abstract

General-purpose Graphic 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), consisting of a group of threads. The CTAs can be executed in any order, thereby providing ample of 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 high TLP.

However, we demonstrate in this paper that executing the maximum possible CTAs on a core is not always the optimal choice from the performance perspective due to inefficient utilization of core resources. Therefore, we propose a dynamic CTA scheduling mechanism, called DYNCTA, which modulates the core-level TLP by allocating optimal number of CTAs, based on application characteristics. DYNCTA allocates more CTAs for compute-intensive applications compared to memory-intensive applications to minimize resource contention. Simulation results on a 30-core GPGPU platform with 31 applications demonstrate that the proposed CTA scheduling provides 28% (up to $3.6\times$*) improvement in performance compared to the existing CTA scheduler, on average. We further enhance DYNCTA by turning off some cores during run-time to limit TLP and power consumption. This proposed scheme, DYNCORE, is shown to provide 21% speedup, while reducing power consumption by 17% and saving energy by 52%, compared to existing CTA schedulers.*

1. 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) [3, 7, 20, 23]. From the programmer's perspective, evolution of CUDA [26] and OpenCL [24] 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 and barrier primitives associated with its group of threads. GPGPU architecture provides synchronization guarantees within a CTA and assumes no dependencies exist 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 maximum number of CTAs per-core, based on the available on-chip resources, to enhance performance.

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 core inactive time. The primary reason behind high core inactivity is very high round-trip fetch latencies of memory requests (core to memory and back) mainly attributed to limited available memory bandwidth. Ideally, one would expect that exploiting the maximum available TLP will hide long memory latencies, as 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 main memory. On the other hand, more threads also cause the number of memory requests to escalate, aggravating the memory fetch latencies. To understand this trade-off, consider Figure 1, which shows performance results of executing optimal number of CTAs per-core (in turn, optimal TLP) and minimum number of CTAs, which is 1. The optimal number of CTAs per-core is statically obtained by exhaustive analysis. The results are normalized with respect to the default GPGPU execution approach, where maximum number of CTAs (restricted by core resources: register file size, shared memory size, maximum number of threads) are exe-

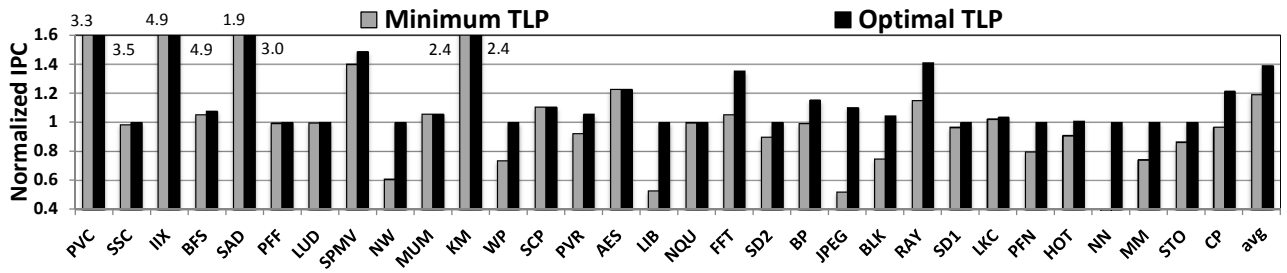


Figure 1: Impact of varying TLP on IPC. First bar shows the speedup when TLP is minimum (only 1 CTA is assigned on each core). Second bar shows the speedup when TLP is optimal (a core executes an optimum number of CTAs such that the performance is maximized). All results are normalized with respect to the CUDA default, where TLP is maximum [26].

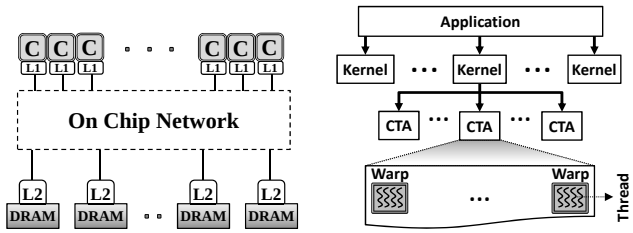
cuting on the cores. These results suggest that varying TLP at the granularity of CTAs has a significant impact on the overall performance of GPGPU applications. The average IPC improvement over all 31 applications with optimal TLP over maximum TLP is 39% (25% geometric mean (GMN)), and goes up to 4.9× in IIX and 3.5× in PVC. Thus, the goal of this paper is to propose a CTA scheduling scheme that enhances performance by executing the optimal number of CTAs on the cores. This number might be different for each application, and determining it would not be practical as it would require executing the application exhaustively 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 and power of GPGPUs. In this context, we propose two schemes:

• First, 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 of an application during execution and changing TLP dynamically at the granularity of CTAs depending on the monitored metrics. DYNCTA favors higher TLP when the application is compute-intensive and lower TLP when memory-intensive. Evaluation on a 30-core GPGPU platform with 31 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.

• Second, building upon DYNCTA, we propose a core gating mechanism (DYNCORE) that limits TLP further for some applications by turning off some cores during run-time. Similar to DYNCTA, DYNCORE limits TLP for memory-intensive applications. On average, this scheme increases performance by 21%, reduces power consumption by 17%, and provides 52% energy savings compared to the case when maximum TLP is exploited.

2. Background

In this section, we provide a brief background on the GPGPU architecture, applications considered, and typical scheduling strategies. Further details on these can be found in [1, 10, 11, 15, 21, 22, 31, 32].



(a) Baseline Architecture (b) Application Hierarchy

Figure 2: GPGPU Background

Baseline GPGPU Architecture:- A GPGPU consists of many simple in-order cores (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 an SIMT width of 8, and 8 memory controllers (MCs). This configuration is similar to the ones studied in recent works [8, 9, 30]. Each core has a private L1 data cache, read-only texture and 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 [8, 9, 30]. Further, 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". Detailed baseline platform configuration used in this work is described in Table 1.

Table 1: Baseline Configuration

Shader Core Config. [30]	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	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 [30]	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	FR-FCFS (128 Request Queue Size/MC), 4B Bus width, 4 DRAM-banks/MC, 2KB page size, 4 Burst Size, 8 MCs
GDDR3 Timing [2, 30, 8]	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

GPGPU Application Design:- Figure 2b shows the hierarchy of a GPGPU application consisting of threads, warps, CTAs, and kernel. Threads associated with a GPGPU appli-

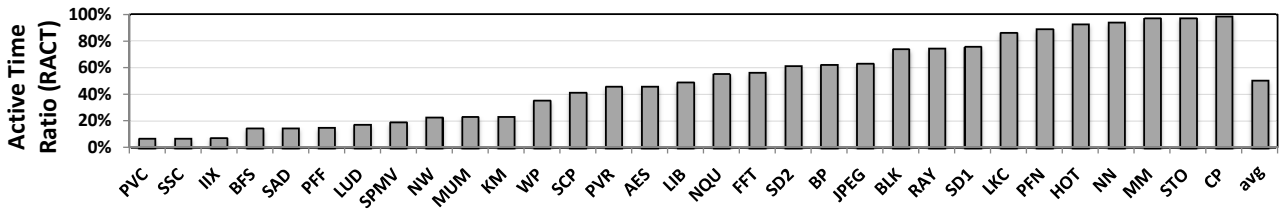


Figure 3: The average Active Time Ratio (RACT) of applications. RACT is defined as the average of ratio of cycles during which a core can fetch new warps, to the total application execution cycles.

cation are grouped into "warps" or "wavefronts" (group of 32 threads). Group of warps constitute a "CTA" or "thread block". A CTA is essentially a batch of threads that can coordinate amongst 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 as there are no ordering restrictions on CTAs and any core is free to schedule any CTA. Further, each kernel is associated with many CTAs as shown in Figure 2b, which in turn form a GPGPU application. Kernels implement specific modules of a GPGPU application.

Kernel, CTA, Warp and Thread Scheduling:- In GPGPUs, scheduling is a three-step process. First, a kernel associated with 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 [27]) assigns CTAs of the launched kernel to all the available cores (there are C cores in the system) [2, 3]. The CTA assignment is done in a round-robin fashion. 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 at least one CTA. After this assignment, if a core is capable of executing multiple CTAs, a second round of assignment starts (provided enough CTAs for the launched kernel are available). This process continues until all CTAs have been assigned or all the cores have been assigned 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 (number of threads, shared memory and register file size [3, 19]), and cannot exceed the limit of 8 [26]. 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. Also, if a CTA of kernel X needs more resources than that of kernel Y, then the N value for kernel X will be smaller compared to that of kernel Y. 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 of the core. Every 4 cycles, 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 (waiting for data to come back from main memory), the entire warp (32 threads) 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.

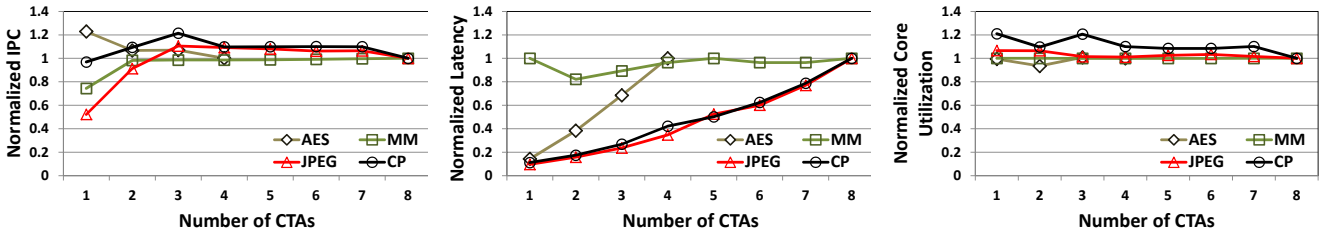
Application Suite:- In this paper, we consider a wide range of CUDA applications targeted to evaluate the *general purpose* capabilities of GPUs. Our application suite includes CUDA NVIDIA SDK [28] applications, and Rodinia [5] benchmarks, which are mainly targeted for heterogeneous CPU-GPU-accelerator platforms. We also study Parboil [17] 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 [12] and a few third party GPGPU applications. In total, we study 31 applications tabulated in Table 2. We evaluate our techniques on GPGPU-Sim [3], a publicly-available cycle-accurate GPGPU simulator. We model the configuration described in Table 1 in GPGPU-Sim. Each application is run till completion or 1 billion instructions, whichever comes first.

3. Detailed Analysis of TLP

3.1. 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 the applications do not reach their potential. For analyzing the primary cause of performance bottlenecks, we study the core active/inactive times for the benchmarks shown in Table 2. We define the core inactive time (*Ninact*) as the number of cycles when a core is not able to fetch any new instructions (warps) and the core active time (*Nact*) as the number of cycles when a core is able to fetch new warps. The average active time ratio (*RACT*), is defined as the average of $Nact/(Nact+Ninact)$ across all cores.

Figure 3 shows the applications in increasing order of their *RACTs*. We observe that, on average, cores are inactive for almost 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 maximum possible CTAs are concurrently executing on all cores. These results paint an ugly picture of GPGPUs, which are known for demonstrating high TLP and delivering high throughput. Inactivity at a core might happen mainly because of three reasons. First, all the



(a) Effect on IPC (b) Effect on lat_{rt} (c) Effect on $util$

Figure 4: 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.

Table 2: List of benchmarks: *Type-C*: Compute-intensive applications, *Type-M*: Memory-intensive applications, *Type-X*: Applications not exhibiting enough parallelism.

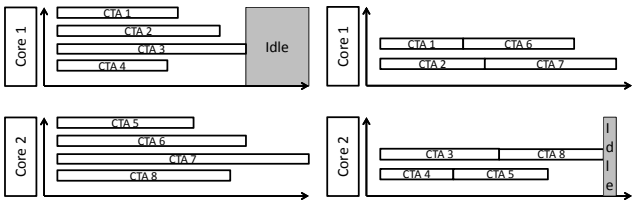
#	Suite	**Applications**	Abbr.	Type
1	MapReduce	Page View Count	PVC	Type-M
2	MapReduce	Similarity Score	SSC	Type-M
3	MapReduce	Inverted Index	IIX	Type-M
4	SDK	Breadth First Search	BFS	Type-M
5	Parboil	Sum of Abs. Differences	SAD	Type-M
6	Rodinia	Particle Filter (Float)	PFF	Type-M
7	Rodinia	LU Decomposition	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

warps could be waiting for the data from main memory and hence, their progress is road-blocked. We call this as "memory waiting time". Second, pipeline may be stalled because of excessive write-back (WB) contention at 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 WB stage. This leads to stalling of pipeline for multiple cycles, preventing new warps from being fetched. Third 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. 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 high memory applications are ported to GPGPUs.

To analyze the reason for inactivity, we start by categorizing the applications into three categories. Applications with high *RACT* (> 66%) are categorized as compute-intensive applications (Type-C). Among the remaining applications, the ones with small idle time ratio (< 20%) are classified as memory-intensive (Type-M) applications, and the remaining as Type-X applications. Type-X applications do not have enough threads to execute to keep all cores busy, and do not exhibit high level of parallelism. 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 will escalate in proportion, thereby increasing the memory latency that needs to be hidden. Moreover, this increase in TLP will saturate 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 3.2. We mainly focus on IPC, round-trip fetch latency, and core utilization of the GPGPU system. Round-trip fetch latency (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. Core utilization ($util$), 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 cycles for core i. 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 at a core, n, which is from 1 (minimum) to N (maximum). 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 all the three metrics considered are *normalized* with respect to their values when N CTAs are launched on the core (maximum parallelism).

Figures 4a, 4b, and 4c show the effect of varying TLP (at the granularity of CTAs) on IPC, latency, and core utiliza-



(a) A case where $n = 4$ (b) A case where $n = 2$

Figure 5: 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 a 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 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 balancing on the distribution of CTAs.

tion, 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×). 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 channels. Surprisingly, we notice that $n = 1$ (minimum CTA count per core, minimum TLP) leads to the highest performance (lowest lat_{rt} and the highest $util$) for this application. Thus, for this application, 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 with a very low slope 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 interesting variances in IPC, latency and utilization as TLP is increased. In this application, when parallelism is not enough ($n = 1$ and $n = 2$), the ability to hide the memory latency is limited, resulting in lower performance compared to when TLP ($n = 3$) is higher. However, as the number of CTAs increases beyond $n = 3$, the corresponding increase in TLP leads to increase in the memory access latency, leading to loss in performance. Hence, we define $n = 3$ as optimal CTA count for JPEG.

3.2. Implications of Tuning TLP on Core Utilization

Next, we observe that varying the number of CTAs also has an impact on the core utilization. This is mainly contributed by the differing execution times across CTAs, causing an imbalance on the execution times of cores. 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 5 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 the application is fetching/decoding a warp without losing much time stalling or waiting for data from the memory, one would expect to see an increasing performance as we increase 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 hiding capability is limited due to low TLP, the IPC bar always follows the $util$ bar. As expected, we see that $util$ goes down as n approaches N, and IPC drops due to unbalanced CTA load distribution. $util$, hence IPC, reaches its peak when $n = 3$.

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× 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.

4. Proposed CTA and Core Power Modulation

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

4.1. DYNCTA: Dynamic CTA Scheduling

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 period of time, we limit TLP by reducing the number of CTAs on the cores.

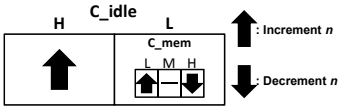


Figure 6: The overview of DYNCTA algorithm. If C_idle is high, n is incremented. Otherwise, C_mem is checked. If it is low, n is incremented. If it is high, n is decremented.

On the other hand, when an application is compute-intensive, we would like to exploit maximum TLP by increasing the number of CTAs on the cores. Below, we describe our approach to determine whether an application is memory or compute-intensive during a period of time.

What metrics to monitor for dynamic modulation: 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 this core is idle. A very high *C_mem* value indicates that the core does not have enough threads to execute to keep the pipeline busy, thus it is better to increase TLP. *C_mem* is the number of core cycles during which all the warps are waiting for their data to come back, regardless of the pipeline state being stalled or not. This metric indicates how much pressure the execution is exerting on the memory channels. If this number is very high, that means 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 a core, the global block scheduler starts with assigning $\lfloor N/2 \rfloor$ CTAs to each core ($n = \lfloor N/2 \rfloor$). CTAs are distributed to cores one by one, as in round-robin CTA scheduler. 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 (at every 2048 clock cycles, a tunable parameter) 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*) which categorizes the value as low, medium, or high.

Figure 6 shows how the number of CTAs is modulated. First, *C_idle* is checked. 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. If *C_idle* is low ($< t\_idle$), then we check *C_mem* to make a decision. 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. 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 on the core, then $n \leq N$ must hold. These two conditions are always checked when making a decision on incrementing or decrementing n. If the resulting n after the decision violates these conditions, then n is not changed.

CTA pausing: According to [26], CUDA blocks (CTAs), once assigned to a core, *cannot* be preempted, or assigned to another core. This presents a problem when our algorithm wants to reduce 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 further decremented by 1, 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 paused CTA, which is the least recently assigned CTA on the core, resumes 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. This means that, as long as there is a ready warp that belongs to CTA1 or CTA2, it will always be prioritized to be fetched over a warp of CTA3. A warp of CTA3 can be fetched only if there are no ready warps that belong to CTA1 or CTA2. Let us now assume that n is decremented further. This time, CTA2 is deprioritized. As long as there are ready warps that belong to CTA1, they will have the priority to be fetched. Note that since $n = 1$, we cannot decrement n any further. If n is incremented by 1, CTA2 again gets the priority to have its warps fetched. If n is incremented further, warps of CTA1, CTA2 and CTA3 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.

Comparison vs. *opt*: We observe that some applications can have high RACT in a given time interval and low RACT in another interval. For such applications, *opt* might be different for intervals showing different behaviors. Thus, our algorithm potentially can outperform the case where $n = opt$ for some applications. As discussed in Section 3, the level of parallelism mainly affects lat_{rt} and $util$. The problem related to $util$ manifests itself towards the end of kernel execu-

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_Cores(n):Default CTA scheduler with CTA limit=n
procedure INITIALIZE
    n ← ⌊N/2⌋
    ISSUE_CTAS_TO_CORES(n)

procedure DYNCTA
    INITIALIZE
    for all cores do
        if C_idle ≥ t_idle then
            if nCTA > n then
                Unpause least recently assigned CTA
            else if n < N then
                n ← 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 ← n+1
        else if (C_mem ≥ t_mem_h) then
            if n > 1 then
                n ← n−1
        for i = 0 → (nCTA − n) do
            Pause most recently assigned CTA
```

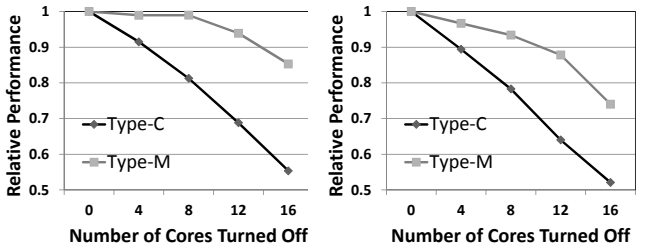
tion, as illustrated in Figure 5. For this reason, our algorithm aims to solve the problem caused by lat_{rt}. Since the utilization problem is usually more pronounced when n is large, and our algorithm limits TLP by limiting n, we indirectly mitigate the effects of the utilization problem as we limit n for some applications. For some Type-C applications like CP, opt is dependent on $util$, as explained in Section 3.2. Thus, our algorithm may not be able to match the performance of the $n = opt$ case. For example, 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 3.2. Since CP is very compute-intensive, our algorithm eventually sets n to 8. Thus, the algorithm fails to converge to opt. Although there are cases where the algorithm fails to converge to opt (mostly in 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 determined the values of these parameters experimentally, which are 16, 128, and 384, respectively. These values are micro-architecture dependent, and need to be recalculated by experiments for different configurations. These thresholds are based on heuristics, and they are unlikely to differ significantly across different micro-architectures. Another parameter is the sampling period of 2048 cycles. Sensitivity analysis of these parameters are reported in Section 5.3. All variables and thresholds we used in this work are given in Table 3.

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.

Table 3: 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
G_act	The sum of Nact of each core that is not turned off
t_act	Threshold that determines whether *G_act* is low or high



(a) Effect of turning off cores on baseline

(b) Effect of turning off cores on DYNCTA

Figure 7: Effects of turning off cores. The results are normalized with respect to using all the cores in a 30-core system.

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.

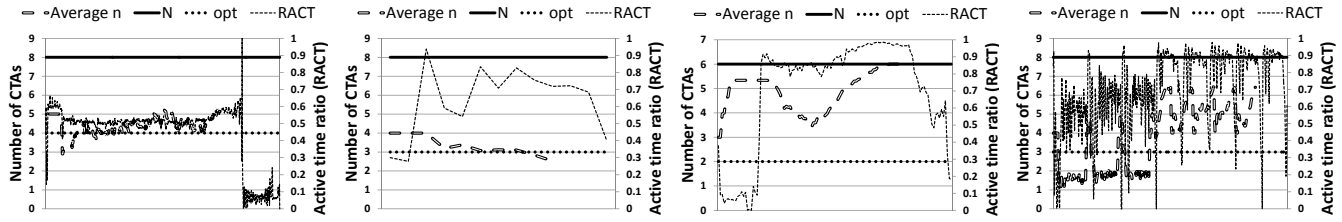
Synchronization: We model atomic instructions and take into account latency overheads due to inter-CTA synchronization. We make sure that we do not deprioritize CTAs indefinitely to prevent livelocks and starvation. Also, our simulation results show that CTA-pausing provides extra performance benefits, which means that the paused CTAs do not degrade performance because of synchronization.

4.2. DYNCORE: Power-Performance Optimizations

In this section, we describe how we extend DYNCTA to achieve power savings as well.

Power problem: As the number of cores in GPUs increases, power consumption also escalates [14]. In Section 3, we have shown that some applications suffer from low core utilization, and some applications spend a lot of time stalling due to the network and memory congestion. Inefficient usage of resources can aggravate the power problem further.

Effects of number of cores: A recent work [14] has investigated the effect of turning off a number of cores, on system power and performance. It is suggested that power consumption can be significantly lowered without losing much performance by turning off cores in memory-intensive applications. In order to observe the effect of the number of cores in the system, we have run experiments with various number of cores. Figure 7a shows the normalized IPCs when 0, 4, 8, 12 and 16 cores are turned off in our baseline configuration.



(a) SAD (b) JPEG (c) RAY (d) FFT

Figure 8: CTA modulation over time. Except RAY**, which is a Type-C application suffering from low core utilization, DYNCTA is able to modulate TLP accurately.**

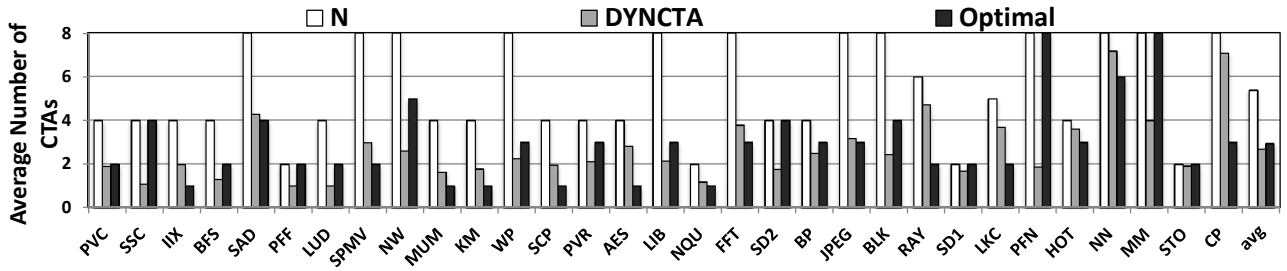


Figure 9: Average number of CTAs assigned to the cores, with the default CUDA approach, DYNCTA, and the optimal TLP.

Figure 7b shows the same graph when DYNCTA is applied. In both graphs, we observe that Type-C applications' performance loss is almost directly proportional to the number of cores that are turned off. However, Type-M applications manage to tolerate the performance loss due to using fewer cores. This stems from the fact that Type-M applications suffer from long inactive times due to congestion in the network and the memory, and deploying fewer cores would mitigate the negative effects of high parallelism. However, since Type-C applications have a lot of core active time, reducing the number of cores would hamper their processing rate. In a hypothetical case, where the application is not suffering from low core utilization described in Section 3 (assuming that the load distribution is perfectly balanced, and all the cores finish their execution exactly at the same time) and $RACT = 100\%$, the performance loss of that application would be directly proportional to the number of cores turned off.

Power-performance benefits: In Section 4.1, we have described our approach that limits TLP to obtain better performance over the default CUDA approach. The aim of this section is to obtain both power and performance improvements over baseline. In order to achieve this, we propose DYNCORE, which schedules CTAs using DYNCTA, and limits TLP further by turning off cores.

How power optimization is done: Since Type-M applications can tolerate execution with fewer cores, we turn off cores if the application behaves like a Type-M application (with $< 60\%$ RACT) during a period of time. This is achieved with a strategy similar to that employed in DYNCTA. During run-time, every 2048 cycles, we monitor *G_act*, which is the sum of *Nact* of each core that is not turned off. Note that *G_act* is collected globally, using information from all the cores in the system, unlike *C_mem*, and *C_idle* which are per-core metrics. If *G_act* is greater than *t_act*, the application behaves like a Type-C application. *t_act* is a threshold equal to 40960, which stands for 66% RACT in a 30-core system with a sampling period of 2048. Thus, we continue executing the application using all the cores in the system. On the other hand, if $G\_act < t\_act$, the application behaves like a Type-M application and we turn off 8 of the cores in the system (we analyze why we turn off 8 cores in Section 5.3). Since we cannot preempt a running CTA on a core, we wait for all the CTAs to finish execution on the cores that are going to be turned off and we do not assign any new CTAs to these cores. If *G_act* becomes greater than *t_act* during the time in which we are waiting for all CTAs to finish, those cores can continue scheduling more CTAs, and are not turned off. The number of CTAs allocated per core is determined by DYNCTA.

4.3. Hardware Overheads

Overheads associated with DYNCTA:- Each core will need 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 predefined threshold values, and the outcome of our CTA modulation decision is communicated to the global block (CTA) scheduler, associated with GPUs [27]. Since cores and the scheduler already communicate (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.

Overheads associated with DYNCORE:- DYNCORE compares *t_act* against *G_act*. This global parameter is calculated by gathering information from all the cores. Since the outcome of the comparison is used by the global scheduler to

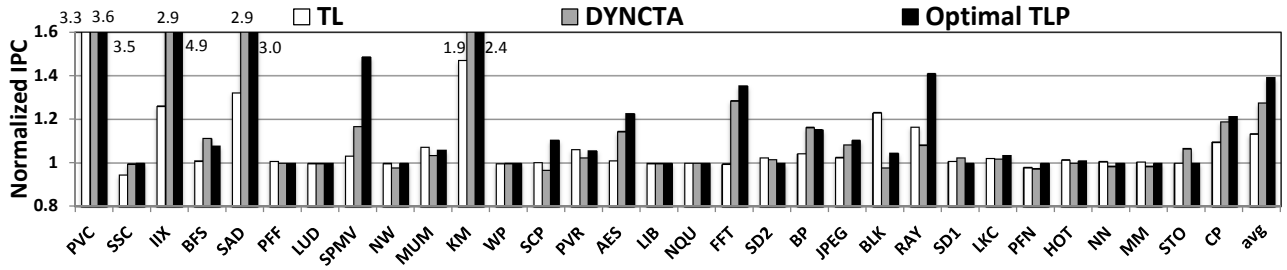


Figure 10: IPC improvements by DYNCTA. Results are normalized with respect to the default CUDA approach. The first bar shows the normalized IPC of two-level scheduler [25]. The second bar shows the normalized IPC of DYNCTA. The third bar shows the normalized IPC of the optimal TLP case where $n = opt$.

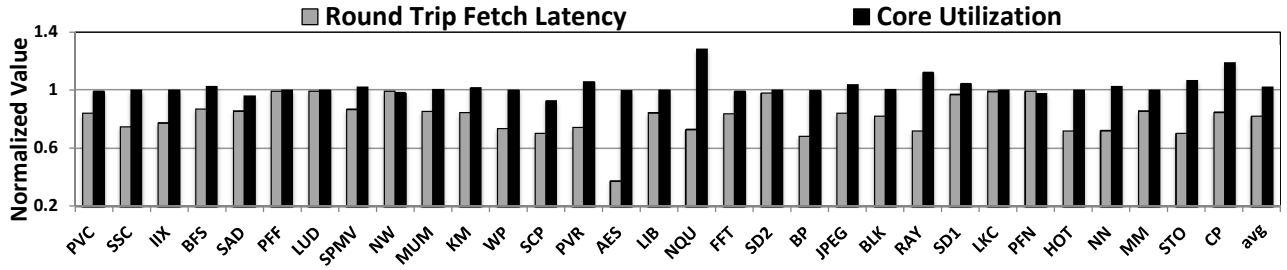


Figure 11: Effects of DYNCTA on round-trip fetch latency (lat_{rt}), and core utilization ($util$), w.r.t. CUDA default.

make a decision, the hardware for gathering all local information can be accommodated by the global scheduler. This will entail additional adders and a comparator.

We designed the required hardware using Verilog HDL and implemented it with TSMC 65 nm libraries using Synopsys Design Compiler. For a 30-core system, DYNCTA and DYNCORE require 0.041 mm^2 area (less than 0.01% of GeForce GTX 285 area), and consumes 6.9 mW extra dynamic power.

5. Experimental Results

We evaluate DYNCTA and DYNCORE with 31 applications, using the cycle accurate GPGPU-Sim [3], with the configuration described in Table 1. Our baseline configuration uses the default CUDA approach for determining the number of CTAs assigned to the cores.

5.1. 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 `SAD`, `JPEG`, `RAY`, and `FFT`. 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 correctly captures the optimal behavior of this application. In `RAY`, we have $N = 6$, thus we start with $n = 3$. In the beginning, the application is not compute intensive. However, during this period, 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. Since this is a Type-C application, opt is not equal to N due to the core utilization problem (see in Section 3.2). Although DYNCTA correctly captures the behavior of this application, it fails to converge to opt because of the core utilization problem. 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

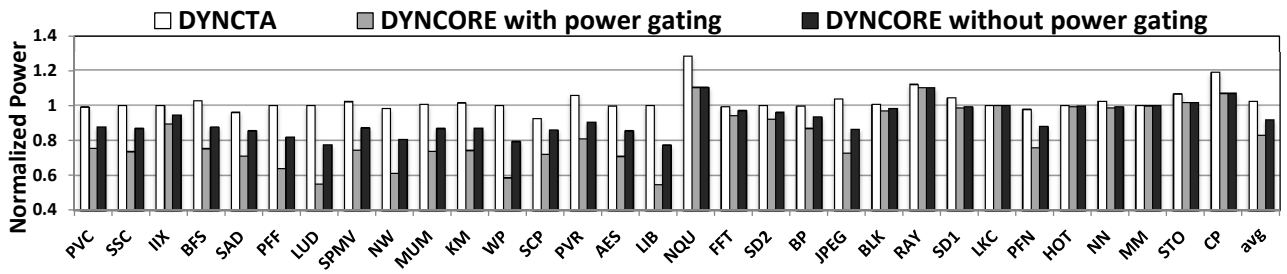


Figure 12: Power consumption with DYNCTA and DYNCORE. Results are normalized with respect to the default CUDA approach.

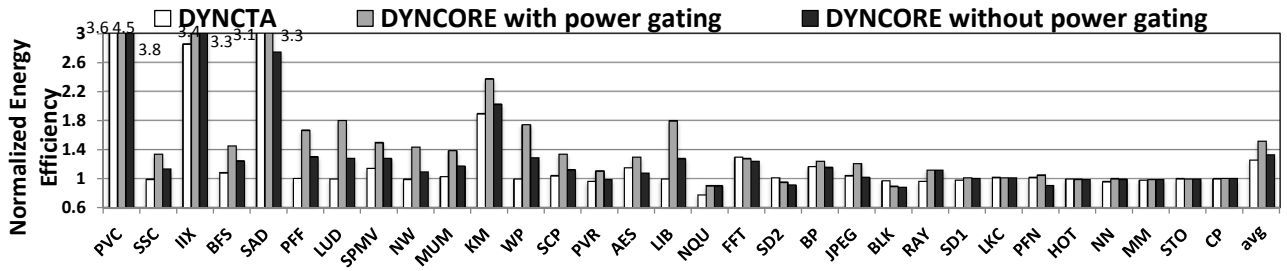


Figure 13: Energy efficiency with DYNCTA and DYNCORE. Results are normalized with respect to the default CUDA approach.

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 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× reduction in lat_{rt} for STO. DYNCTA reduces the memory waiting time of BFS by 2.2×. 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. [25]. Our experiments show that DYNCTA outperforms TL by 12.5%, on average. TL yields significantly higher performance than DYNCTA in 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 lat_{rt} to be lower compared to that of the default CUDA approach. 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*, compared to the baseline. 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 consistent with the IPC improvements that those applications are getting. 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 18% across all applications. DYNCTA effectively reduces the memory fetch latency while keeping the cores busy, which in turn, translates to perfor-

mance improvements. Since limiting the number of CTAs reduces the working set of the application in the memory, data locality also improves, resulting in better cache hit rates. Due to space constraints, we report a brief overview of the cache performance of DYNCTA. On average, L1 miss rates reduce from 71% to 64%. These results show that increasing hit rates is a contributor of the performance of DYNCTA.

5.2. Power-Performance Results with DYNCORE

In this section, we evaluate the impact of DYNCORE on performance, power consumption and energy efficiency of the applications. To calculate power consumption, we have assumed that half of the total GPGPU power is "idle power", as suggested by [14]. Idle power constitutes a significant portion of the total power consumption when per-core-power-gating (PCPG) techniques are not employed. NVIDIA Tegra architecture [29] is an example that employs PCPG for CPUs. Even though current GPUs do not use PCPG, future architectures are expected to employ this mechanism as the power problem becomes more severe. Based on the model by [14], we estimate the *relative* power consumption of DYNCORE with respect to the baseline. The number of each type of instructions (e.g. ALU, FP, REG) does not differ across our schemes (e.g. same FP instructions are executed), thus not affecting the relative power consumption. The main difference comes from the number of active cores, and the utilization of these active cores. When the cores are not utilized, they consume idle power. Turned-off cores consume zero power if PCPG is deployed, and idle power if not. Idle power consumption is estimated from a simple model derived from the experimental results given in [14]. In this work, we only focus on the core power, not memory and network power.

Figure 12 shows the power consumption of GPGPU applications. DYNCTA consumes 2% more power compared to the baseline. With DYNCORE, the average power savings are 17% and 8%, with and without PCPG, respectively. As DYNCORE mainly targets Type-M applications, the power savings in Type-M applications are greater than Type-C applications. Figure 13 shows the energy efficiency of GPGPU applications. We define energy efficiency as "the number of instructions committed per Watt". DYNCTA achieves an average energy efficiency improvement of 26% compared to the baseline. This improvement is contributed by higher performance, and the same power envelope. With DYNCORE, where we turn off 8 cores, the average energy efficiency improvements are 52% and 33%, with and without PCPG, respectively. DYNCORE achieves an average speedup of 21% (12% GMN) (see Figure 14a) compared to the baseline. Thus, the energy efficiency improvement of DYNCORE is attributed to both performance and power benefits.

5.3. Sensitivity Analysis

Figure 14a shows the effect of the number of cores turned off. As explained in Section 4.2, DYNCORE turns off 8 cores depending on the value of *G_act*. As we turn off more cores, performance drops, but power savings and energy efficiency (assuming PCPG is applied) increase. Since we want DYNCORE to achieve both power and performance improvements, we choose to turn off 8 cores in DYNCORE.

Figure 14b shows how DYNCORE performs in larger systems. Since crossbar is not scalable for larger 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. In a 56 core system with 8 MCs (8×8 mesh), on average, we obtain 9% speedup, and 10% power savings. Even though we have increased C while keeping the number of MCs constant, the improvements are moderate. In a 110 core system with 11 MCs (11×11 mesh), we obtain 27% speedup, and 32% power savings, on average. Here, DRAM contention plays a bigger role, and the benefits of our schemes are more significant. Note that DYNCORE turns off 16 cores in a 56-core system, and 32 cores in a 110-core system.

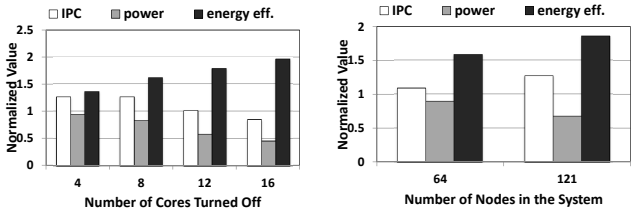
As DYNCTA mostly helps Type-M applications, the number of memory requests sent from a core is an important factor that can affect its benefits. To evaluate this, we varied the size of MSHR per core and observed that changing MSHR/core from 64 to 32 degrades the performance by 0.3%. Reducing it to 16 causes 0.6% performance loss, on average.

Next, we examined the impact of memory frequency. We evaluated our schemes on systems consisting of 1107 MHz [2] 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 [18]. Thus, we believe that our schemes will be applicable to upcoming GPUs.

We also conducted a sensitivity analysis on the parameters in DYNCTA algorithm. Changing the metric sampling period from 2048 cycles to 4096 cycles degraded our benefits by 0.1%. We have also varied the thresholds (*t_idle*, *t_mem_l*, and *t_mem_h* between 50% and 150% of their default values (given in Section 4.1), and observed a loss between 0.7% and 1.6%. Thus, DYNCTA can work with almost equal efficiency with a broad range of threshold values.

6. Related Work

While previous works [3, 13] observed the problem caused by exploiting all available TLP, the only mechanism that has placed sanctions on TLP is proposed by Rogers et al. [30]. It applies a warp scheduling method to lower the effect of TLP on cache contention. However, in our work, we attack the same problem to reduce long memory latencies and write-back contention. We believe that this work is one of the first



(a) Sensitivity analysis on the number of cores turned off

(b) Sensitivity analysis on system size (56 and 110-core systems)

Figure 14: Sensitivity analysis of DYNCORE. In (a), the results are normalized with respect to the baseline with no cores turned off. In (b), the results are normalized with respect to the baseline with 64 and 121 nodes, using a 2-D mesh.

works that proposes an architectural solution to dynamically optimize TLP in GPUs. Bakhoda et al. [3] shows that 7 out of 15 applications show improvements over baseline when the number of CTAs is lowered. We have evaluated the applications with all possible CTA limits, whereas [3] shows results for only 25%, 50%, and 100% of the default CUDA limit. Chen et al. [6] proposed an analytical model showing the effects of the number of concurrent threads on shared caches for multicores; whereas we mainly consider the effects of TLP on the memory system. Chadha et al. [4] proposed a dynamic TLP management scheme for CMPs. Hong et al. [13] developed an analytical model to predict the execution time of GPUs. This prediction considers available TLP and memory intensiveness of an application. However, in our paper, we proposed dynamic techniques which incorporate the changing behavior of an application and calculates the optimal TLP for the GPGPU system. Moreover, we modeled network and memory congestion in detail and controlled the parallelism at CTA-level. Tuning TLP in GPGPUs via CTA allocation is a much effective way mainly because of two reasons. First, the work assignment on cores is done at the CTA-level and hence, one can potentially control how much work to be assigned to cores during the scheduling process. Second, according to Jia et al. [16] the threads that belong to the same CTA exhibit good data locality and therefore, limiting TLP at the granularity of CTAs also brings the benefit of maintaining, and even improving data locality and thus, increasing cache performance. Hong et al. [14] presented a GPGPU power model to determine the optimal number of cores required to optimize the power and performance of the system. In our work, we strive to amalgamate the performance benefits with the power savings obtained by turning off cores at run time.

7. Conclusions

Enhancing the performance of applications through abundant TLP is the primary advantage of GPGPUs compared to CPUs. Therefore, current GPGPU scheduling attempts to allocate maximum number of CTAs per core to maximize TLP. However, we show that executing maximum number of CTAs per core is not always the best solution to boost performance. This is primarily due to high core inactive times caused by longer round-trip fetch latencies of memory requests. Thus, instead of hiding the longer memory latencies through TLP, more threads could degrade system performance. The main contribution of this paper is to mitigate the negative effects of high TLP by proposing a dynamic CTA scheduling algorithm for GPGPUs that attempts to allocate optimal number of CTAs per core based on application demands. The proposed DYNCTA scheme uses two metrics (*C_idle* and *C_mem*) to decide the optimal allocation of CTAs. It is shown that DYNCTA enhances application performance on average by 28% (up to 3.6×) compared to the default CTA allocation, and is close to the best possible static allocation, which is shown to provide 39% performance improvement. In addition, DYNCORE, a dynamic core gating mechanism, is proposed. DYNCORE works in conjunction with DYNCTA to shut down shader cores for power-performance optimization in GPGPU platforms. Detailed evaluation shows that, on average, DYNCORE can provide 21% IPC improvement, while reducing power and energy consumption by 17% and 52%, respectively, compared to the default CUDA approach.

References

[1] J. Adriaens, K. Compton, N. S. Kim, and M. Schulte, "The case for gpgpu spatial multitasking," in *HPCA*, 2012.

[2] A. Bakhoda, J. Kim, and T. Aamodt, "Throughput-effective on-chip networks for manycore accelerators," in *MICRO*, 2010.

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

[4] G. Chadha, S. Mahlke, and S. Narayanasamy, "When less is more (limo):controlled parallelism for improved efficiency," in *CASES*, 2012.

[5] 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.

[6] 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.

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

[8] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for gpu architectures," in *MICRO*, 2011.

[9] W. Fung and T. Aamodt, "Thread block compaction for efficient simt control flow," in *HPCA*, 2011.

[10] W. Fung, I. Sham, G. Yuan, and T. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *MICRO*, 2007.

[11] 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.

[12] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *PACT*, 2008.

[13] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *ISCA*, 2009.

[14] S. Hong and H. Kim, "An integrated GPU power and performance model," in *ISCA*, 2010.

[15] 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.

[16] W. Jia, K. A. Shaw, and M. Martonosi, "Characterizing and improving the use of demand-fetched caches in GPUs," in *ICS*, 2012.

[17] John A. Stratton et al., "Parboil: A revised benchmark suite for scientific and commercial throughput computing," 2012.

[18] 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.

[19] D. Kirk, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, 2007.

[20] K. Krewell, "Amd's fusion finally arrives," *Microprocessor Report*, 2011.
[21] K. Krewell, "Nvidia lowers the heat on kepler," *Microprocessor Report*, 2012.
[22] J. Lee, N. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for gpgpu applications," in *MICRO*, 2010.
[23] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," in *PPoPP*, 2009.
[24] A. Munshi, "The OpenCL Specification," June 2011.
[25] 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.
[26] NVIDIA, "CUDA C Programming Guide," Oct. 2010.
[27] NVIDIA, "NVIDIA Tegra Multi-processor Architecture," Feb. 2010. [Online]. Available: http://www.nvidia.com/docs/IO/90715/Tegra_Multiprocessor_Architecture_white_paper_Final_v1.1.pdf
[28] NVIDIA, "CUDA C/C++ SDK code samples," 2011. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples
[29] NVIDIA, "Fermi: NVIDIA's next generation CUDA compute architecture," Nov. 2011. [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html
[30] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *MICRO*, 2013.
[31] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *ISPASS*, 2010.
[32] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity effective memory access scheduling for many-core accelerator architectures," in *MICRO*, 2009.