KIPS Tr. Comp. and Comm. Sys.
Vol.6, No.5 pp.219~230 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2017.6.5.219

A Novel Cooperative Warp and Thread Block Scheduling Technique for Improving the GPGPU Resource Utilization

Do Cong Thuan† · Yong Choi†† · Jong Myon Kim††† · Cheol Hong Kim††††

ABSTRACT

General-Purpose Graphics Processing Units (GPGPUs) build massively parallel architecture and apply multithreading technology to explore parallelism. By using programming models like CUDA, and OpenCL, GPGPUs are becoming the best in exploiting plentiful thread-level parallelism caused by parallel applications. Unfortunately, modern GPGPU cannot efficiently utilize its available hardware resources for numerous general-purpose applications. One of the primary reasons is the inefficiency of existing warp/thread block schedulers in hiding long latency instructions, resulting in lost opportunity to improve the performance. This paper studies the effects of hardware thread scheduling policy on GPGPU performance. We propose a novel warp scheduling policy that can alleviate the drawbacks of the traditional round-robin policy. The proposed warp scheduler first classifies the warps of a thread block into two groups, warps with long latency and warps with short latency and then schedules the warps with long latency before the warps with short latency. Furthermore, to support the proposed warp scheduler, we also propose a supplemental technique that can dynamically reduce the number of streaming multiprocessors to which will be assigned thread blocks when encountering a high contention degree at the memory and interconnection network. Based on our experiments on a 15-streaming multiprocessor GPGPU platform, the proposed warp scheduling policy provides an average IPC improvement of 7.5% over the baseline round-robin warp scheduling policy. This paper also shows that the GPGPU performance can be improved by approximately 8.9% on average when the two proposed techniques are combined.

Keywords : GPGPU, Parallelism, Performance, Warp Scheduling, Resource Utilization

GPGPU 자원 활용 개선을 위한 블록 지연시간 기반 워프 스케줄링 기법

Do Cong Thuan† · 최 용†† · 김 종 면††† · 김 철 홍††††

요 약

멀티스레딩 기법이 적용된 GPGPU는 내부 병렬 자원들을 기반으로 데이터를 고속으로 처리하고 메모리 접근시간을 감소시킬 수 있다. CUDA, OpenCL 등과 같은 프로그래밍 모델을 활용하면 스레드 레벨 처리를 통해 응용프로그램의 고속 병렬 수행이 가능하다. 하지만, GPGPU는 범용 목적의 응용프로그램을 수행함에 있어 내부 하드웨어 자원들을 효과적으로 사용하지 못한다는 단점을 보이고 있다. 이는 GPGPU에서 사용하는 기존의 워프/스레드 블록 스케줄러가 메모리 접근시간이 긴 명령어를 처리하는데 있어서 비효율적이기 때문이다. 이와 같은 문제점을 해결하기 위해 본 논문에서는 GPGPU 자원 활용률을 개선하기 위한 새로운 워프 스케줄링 기법을 제안하고자 한다. 제안하는 워프 스케줄링 기법은 스레드 블록의 워프들 중 긴 메모리 접근시간을 가진 워프와 짧은 메모리 접근시간을 가진 워프들을 구분한 후, 긴 메모리 접근시간을 가진 워프를 우선 할당하고, 짧은 메모리 접근시간을 가진 워프를 나중에 할당하여 처리한다. 또한, 메모리와 내부 연결망에서 높은 경합이 발생했을 때 동적으로 스트리밍 멀티프로세서의 수를 감소시켜 워프 스케줄러를 효과적으로 사용할 수 있는 기법도 제안한다. 실험결과에 따르면, 15개의 스트리밍 멀티프로세서를 가진 GPGPU 플랫폼에서 제안된 워프 스케줄링 기법은 기존의 라운드로빈 워프 스케줄링 기법과 비교하여 평균 7.5%의 성능(IPC)이 향상됨을 확인할 수 있다. 또한, 제안된 두 개의 기법을 동시에 적용하였을 경우에는 평균 8.9%의 성능(IPC) 향상을 보인다.

키워드 : GPGPU, 병렬성, 성능, 워프 스케줄링, 자원 활용

※ 이 논문은 2016년도 전남대학교 연구년교수연구비 지원에 의하여 연구되었음.
† 비 회 원 : 전남대학교 전자컴퓨터공학부 박사과정
†† 준 회 원 : 전남대학교 전자컴퓨터공학부 박사과정
††† 정 회 원 : 울산대학교 전기공학부 교수
†††† 종신회원 : 전남대학교 전자컴퓨터공학부 교수
Manuscript Received : November 1, 2016
Accepted : January 24, 2017
* Corresponding Author : Cheol Hong Kim(chkim22@chonnam.ac.kr)

1. Introduction

Modern Graphics Processing Units (GPUs) have the ability in quickly performing context-switching and massive multithreading. Moreover, graphics hardware is fast and quickly accelerating due to different fabrication technology from the Central Processing Units (CPUs) [21]. Consequently, utilizing GPUs for handling general purpose computation has gained more and more attention. This leads GPU to become one of the most attractive computing platforms for executing general purpose applications. For such parallel applications, it was proven that GPUs achieve much higher performance than CPUs due to the exploitation of a large number of streaming multiprocessors (SMs) and gigabytes of high memory bandwidth [1].

With new parallel programing models such as CUDA [2, 3], OpenCL [4], GPGPU applications are created with work abstractions in terms of smaller work units, called thread blocks (or Cooperative Thread Arrays - CTAs). A thread block is a collection of threads grouped to form a warp or wavefront. GPGPU architecture is based on the Single Instruction Multiple Thread (SIMT) execution model. This architecture allows the Single Instruction Multiple Data (SIMD) compute units (also known as shader core) of a GPGPU to execute threads in a warp together, thereby amortizing the instruction fetch and decode overhead. In addition, GPGPU architecture provides synchronization guarantee within a thread block and assumes that no dependencies exist across thread blocks, helping in relaxing execution order of thread blocks. This leads to a remarkable increase in parallelism and more effective usage of streaming multiprocessors.

As GPGPU architectures are designed with a focus on high throughput computing, they use non-speculative in-order processor pipelines as a tradeoff for a large number of computational units. Consequently, reducing the overhead of long latency operations is one of the most crucial issues to maximize GPU's hardware resources. GPGPUs, therefore, rely on fast context-switching and a large number of concurrent warps for hiding latency (whenever the execution of a warp is stalled, it can be swapped out and another warp can be swapped in for immediate execution to increase resource utilization without any penalty). To ensure the execution pipeline is kept active in the presence of long latency operations, a hardware warp scheduler must decide in each cycle which of the multiple active warps will be executed next. However, although high thread-level parallelism (TLP) can be theoretically achieved, GPGPU streaming multiprocessors suffer from periods of inactive times that results in underutilization of hardware resources [5, 6].

Warp schedulers play a key role in increasing GPU's hardware resource utilization. The round-robin warp scheduling policy, which is commonly-used, assigns the same priority to all warps. Prior research work [6, 7, 22], however, showed that the traditional round-robin scheduling fails to hide long memory fetch latencies, that are primarily caused by limited off-chip DRAM bandwidth, contributing substantially to the underutilization of GPGPU streaming multiprocessors. Yet, such round-robin fashion also wastes warp/thread parallelism for hiding short latency instructions. On the other hand, when all warps execute a global memory load with long latency, round-robin scheduling cannot hide such long latency loads and causes stall in each streaming multiprocessor.

In this paper, we first propose a new warp scheduling policy that can alleviate the drawbacks of the baseline scheduling policy. We want to introduce a different approach in the effectively using warp parallelism to improve the overall performance of GPGPUs. The goal of the proposed scheduling is to overlap long latency instructions by efficiently utilizing the "warp/thread parallelism". The proposed warp scheduling algorithm enables the hardware scheduler to identify the warps in a thread block which cause a large number of stall cycles so that such warps can be assigned higher priority than the remaining warps in the thread block. By doing this, such warps can overlap latencies each other and the thread parallelism can be saved to use later compute region when is not sufficient to overlap subsequent long latencies. Furthermore, in order to support the proposed warp scheduler, we also propose a mechanism that can dynamically reduce the number of streaming multiprocessors in a GPGPU to which will be assigned thread blocks when encountering a high contention degree of memory bandwidth and interconnection network.

We make the following contributions: (1) we comprehensively rethought of warp scheduling in terms of effective warp parallelism usage, considered interaction between warp and thread block scheduling, (2) finally, designed and evaluated a new hardware warp/ thread block scheduler.

The rest of this paper is organized as follows. Section 2 discusses related work, Section 3 briefly describes GPGPU architecture, Section 4 presents our proposed scheduling policy, Section 5 describes methodology, Section 6 presents our results, and Section 7 concludes the paper.

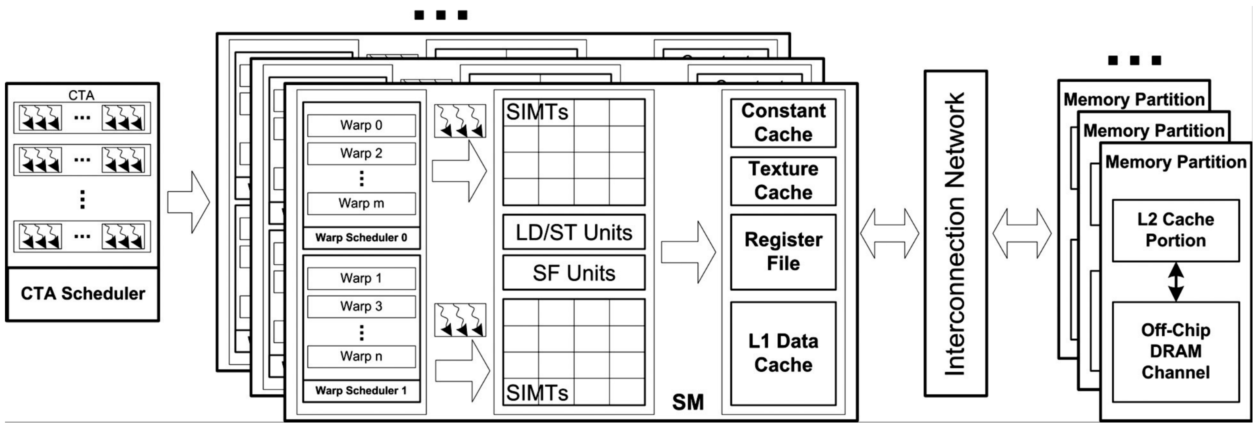


Fig. 1. Baseline GPGPU Architecture

2. Related Work

Prior works have introduced various techniques to improve GPGPU performance or increase computational hardware resources. For example, Fung et al. [15] focused on the impact of branch divergence on GPGPU performance for general-purpose applications. They proposed a combination of threads from different warps to address the hardware underutilization caused by branch divergence. Lee [16] proposed a prefetching mechanism for the presence of many threads in-flight. They also proposed the adaptive throttling to solve the performance degradation even with 100% prefetching accuracy. Some recent research works on GPGPUs [9-12] mostly aimed on optimizing parallel applications to take advantage of the special architecture of GPGPUs.

With the final goal of improving GPGPU performance, warp scheduling has received much attention. A number of scheduling policies have been proposed to increase GPGPU hardware resource utilization. Narasiman et al. [6] proposed a two-level warp scheduling mechanism, where warps are separated into active set and pending set, that can increase the utilization of streaming multiprocessors by creating larger warps and employing a two-level warp scheduling scheme. Meanwhile, Gebhart et al. [13] also proposed a two-level warp scheduling technique, but the primary purpose of their approach is energy reduction. Cache-conscious warp scheduling proposed by Rogers et al. [14] is a family of static and dynamic greedy scheduling approaches to improve intra-warp data locality. Their work improves L1 hit rates for cache-sensitive applications. Jog et al. [30] presented several warp scheduling schemes that can be aware of a subset of thread blocks to reduce the impact of long memory latencies. Lee et al. [31] also tried to alleviate the warp criticality problem by a criticality-aware warp acceleration technique (CAWA).

At the level of thread blocks, round-robin is the most common policy that has been applied to modern GPGPU architectures. Previous works [5, 8] pointed out the drawback of the existing thread block schedulers that maximizing the number of thread blocks assigned to a streaming multiprocessor is not always effective - i.e. increasing the number of thread blocks does not necessarily improve performance. Some other works also suggest that throttling the number of thread blocks in a system can benefit performance. Bakhola et al. [7] demonstrated that limiting the number of thread blocks assigned to a streaming multiprocessor can reduce the contention for the memory system.

Guz et al. [18] presented an analytical model that quantifies the “performance valley” where too many threads can degrade performance because of the resource contention. Cheng et al. [19] also introduced a thread throttling scheme to reduce memory latency in multi-thread CPU systems. They proposed an analytical model and memory task limit throttling mechanism to limit thread interference in the memory stage. Their model relies on a streaming programming language that decomposes applications into separate tasks for computation and memory and then scheduling tasks at this granularity.

A number of recent works have explored cache bypassing, where the memory requests can selectively bypass the cache, for GPUs. Jia et al. [32] proposed a hardware structure called memory request prioritization buffer (MRPB), which employs request reordering and cache bypassing, to avoid a system bottleneck in GPU caches. Meanwhile, Xie et al. [33] introduced a coordinated static and dynamic cache bypassing to boost performance. At compile-time, the global loads with strong preferences are identified for caching or bypassing through profiling. The rest of global loads are bypassed for a fraction of threads.

In evaluation of scheduling and prefetching within GPGPUs, Jog et al. [17] used a thread block allocation strategy where consecutive thread blocks were assigned to the same streaming multiprocessor in their baseline architecture. Recent work from Kayiran et al. [5] dynamically estimated the amount of thread-level parallelism that would improve GPGU performance by reducing the cache and DRAM contention.

3. Background

In this section, we provide a brief background on GPGPU architecture and typically on scheduling strategies. Further details on these can be found in [14, 15, 23, 24].

Fig. 1 illustrates the architecture of a contemporary GPGPU, which is similar in nature to NVIDIA's Fermi design. The GPGPU consists of many streaming multiprocessors (also called shader cores), with each typically having "single-instruction, multiple-threads" lanes of 8 to 32. The target GPGPU architecture used in this work consists of 15 streaming multiprocessors each with an SIMT width of 32 that can collectively issue up to 32 instructions per cycle (IPC), one instruction from each of 32 threads. These streaming multiprocessors are connected by an interconnection network. The baseline GPGPU has multiple levels of memory hierarchies. The global memory (also called device memory) located off-chip with long latency can be accessed by all of the thread blocks in a grid. Each memory controller is associated with a slice of shared L2 cache bank. An L2 cache bank with a memory controller is defined as a memory partition. In the baseline model, the programmers partition the per-SM on-chip storage into programmer-controlled shared memory and hardware-managed data L1 cache. The private L1 data caches of streaming multiprocessors have short latency and are accessible from all of the threads within a thread block. Furthermore, each streaming multiprocessor is equipped a read-only texture, constant cache and low-latency shared memory. For supporting parallel computing, GPGPUs have vast on-chip register file resources for each streaming multiprocessor in order to accommodate a large number of set of threads. Similar to CPUs, GPGPUs also provide several special function units (SFUs) and Load/Store units (L/SUs).

On NVIDIA GPGPUs, the programmers use CUDA, a scalable parallel programming model and software platform for GPGPU and other parallel processors, to parallelize the applications in hierarchies consisting of threads, warps, thread blocks (or CTAs) and kernel. At the highest level, a kernel can be evoked from the host CPU to create a single grid to run on the GPU. Parallel kernel invocations are allowed on the same or multiple GPUs [25]. In cases where the application contains multiple CUDA streams, multiple kernels will be launched simultaneously. Each kernel grid is divided into multiple thread blocks that can be specified as a three-dimensional (3D) array. Each thread block is organized into groups of multiple threads (called warps, each has 32 threads) that are also specified in a 3D format. Thread blocks can be executed on streaming multiprocessors in any order because all of the synchronization primitives are encapsulated in the thread blocks. This leads to an increase of available parallelism since there are no restrictions on thread blocks and any streaming multiprocessor is free to schedule any thread blocks. Modern GPGPUs can provide better performance by effectively utilizing the computational resources since they can exploit FGMT which allows the assignment of one or multiple thread blocks to one streaming multiprocessor [20].

Scheduling in a GPGPU is performed as a three-step process. In the first step, a kernel of a GPGPU application is launched on the GPGPU. After launching the kernel, the global CTA scheduler (e.g. GigaThread[26]) will assign thread blocks (or CTAs) of the launched kernel to all available streaming multiprocessors. We assume that the thread block assignment is done in a round-robin fashion. After this assignment, if a streaming multiprocessor is capable of executing multiple thread blocks and there are enough thread blocks, a second round of assignment starts. This process (second step) continues until all thread blocks have been assigned their maximum limit of thread blocks (limited only by the hardware resources) [7]. After the thread block assignment, warps associated with the launched thread block(s) will be scheduled to SIMT lanes of the corresponding streaming multiprocessor (third step). The warps are also scheduled in a round-robin fashion (baseline). A ready (or active) warp, which is ready to fetch instruction(s), is fed into these lanes for execution every 4 cycles [5]. If there is no ready instruction available, the scheduler will try again in the next cycle and a stall cycle is encountered [27]. The baseline GPGPU provides two parallel warp schedulers and instruction dispatch units - one for odd warps and the other for even warps - in each streaming multiprocessor [25].

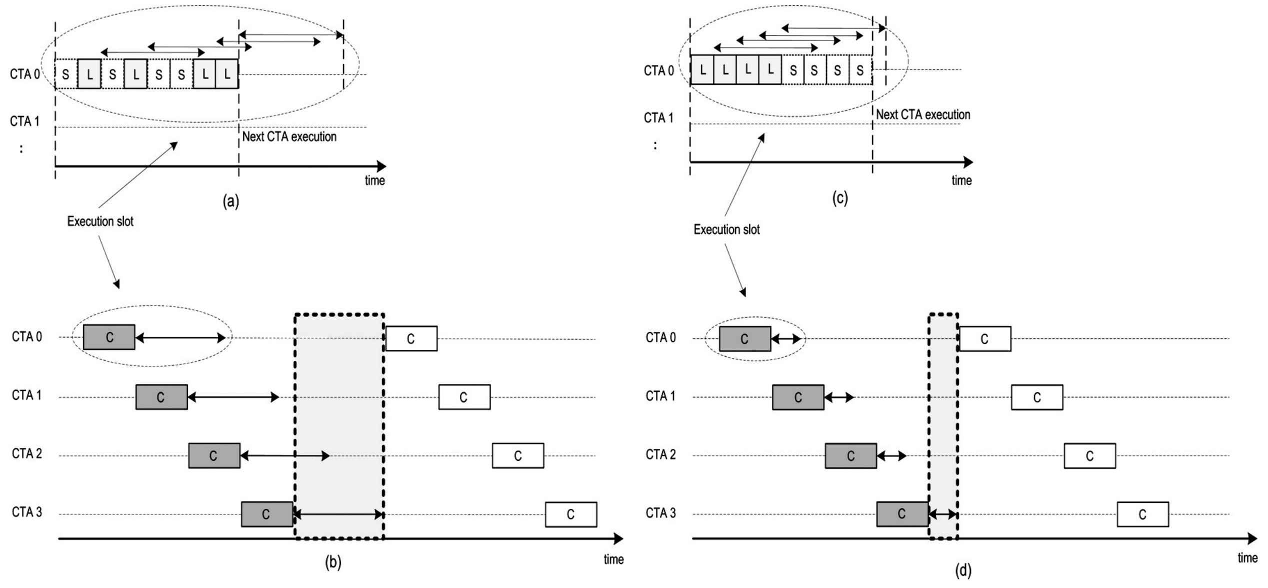


Fig. 2. Example of Different Warp/Thread Block with (a, b) Round-Robin Scheduling and (c, d) Proposed Scheduling

4. Proposed Techniques

In this section, we analyze both the advantages and disadvantages of the round-robin fashion in scheduling warps. We subsequently describe our motivation for improving warp scheduling policy and detail our proposed approach for a new warp scheduling policy.

4.1 Round-robin scheduling and motivation

Warp is the smallest unit for executing instructions and GPGPUs hide latency of operations with its SIMT pipeline structure by quick warp switching. The principle of warp scheduling is relying on the priority of warps to select a warp among all ready warps to issue in the next cycle. The round-robin (RR) scheduling policy, as the name suggests, schedules warps in an iterative manner. All of the warps in the warp pool are treated equally by setting the priority of the most recently issued warp to the lowest. This means that the scheduling order of warps in a CTA (or thread block) is decided by the warp identification (or warp ID).

Theoretically, there are two variations in round-robin scheduling. The fine-granularity multithreading schedules a ready instruction from the active warps in every cycle in a round-robin fashion. Once scheduled, the next instruction in the same warp will not be scheduled until all of the current ready instructions in the other warps have been scheduled. The coarse-granularity multithreading (also known as greedy scheduling) continuously schedules ready instructions from the same warp until stall (mainly due to data dependence) is encountered. It then switches to scheduling ready instructions from the next warp also in a round-robin fashion.

Although the round-robin scheduling is common and applied in many modern GPGPUs, prior research works analyzed the drawbacks of the round-robin fashion. In this section, we will further demonstrate the challenges of using round-robin warp scheduling schemes in order to hide a wide range of instruction and memory latencies. The round-robin scheduler is not effective for hiding long latency operations since it is highly probable that many warps executes the same load instructions within a short time window. This is because warps execute the same kernel and warps ordered in the round-robin fashion execute the same load instructions within a short time interval, unless there is large divergence in warps. Yet, the round-robin scheduling may cause thrashing in the Level 1 Data Cache. For GTRR (greedy-then-round-robin), as it applies round-robin manner to greedily schedule a new warp, the data locality is hardly kept. The GTO (greedy-then-oldest) scheduler tends to maintain data locality, which relieves the memory contention at the data cache and thereby improves performance.

4.2 Proposed warp scheduler

Based on our mentioned analysis above, there is a high percentage stall in warp scheduling. This underutilizes the hardware of streaming multiprocessors, therefore, directly degrades kernel performance. In this section we will present our motivation in solving the problem of the round-robin scheduling with a different approach and then describe our warp scheduling algorithm.

The motivation for a new warp scheduling compared to the round-robin scheduling is shown in Fig. 2. The small squares denote the warps and the length of the squares represents the execution time slot. The arrows indicate latency and its length defines the degree of latency. Part (a) shows how the round-robin scheduling works at the warp level. We assume that the scheduling order of warps in a thread block is random and that the warps at the positions 2, 4, 7 and 8 (from the left) cause longer latency than the remaining warps. For simplicity, the length of latencies is set equal. When a warp is stalled, the next warp will substitute immediately to increase the utilization of hardware resources. At the CTA level, the scheduling process performs similarly to that at the warp level. As a CTA ends its execution slot, another CTA will be assigned to take the hardware (Part (b)). From the example, we can see the impact of scheduling at the warp level on scheduling at the CTA level where the CTA 0 finishes its execution time slot with long latency. Of course, to keep the pipeline busy, the access of a streaming multiprocessor resource can be immediately shifted to the next CTA by performing fast context-switching. However, a problem might occur when there is not enough parallelism to overlap the previous long latency. This means that the streaming multiprocessor will be inactive because there might be no warps that are not stalling, resulting in a significant decrease in capability of hiding long latencies (see Part (b)). In other words, all compute instructions are exhausted in each compute region and causing long stalls for each. Recall that a stall cycle is encountered when the scheduler fails to find a ready instruction.

Although this example just illustrates round-robin fashion (fine-granularity multithreading), the same issue happens to coarse-granularity multithreading (or greedy fashion) but at a lower level. In general, for these scheduling polices, instruction and memory latencies can be hidden only if there is sufficient amount of concurrent threads (i.e. thread-level parallelism) and therefore, the problem may be solved. However, the parallelism level depends on characteristics of each application. One solution for the mentioned problem is to use parallelism efficiently.

Part (c) and (d) in Fig. 2 explain our motivation in solving the problem of the round-robin scheduling policy. It demonstrates the ability of overlapping long latencies using the proposed scheduling policy. Our primary goal is to save available warp/thread parallelism for hiding multiple long latencies of compute regions. In GPGPU applications, each warp often executes a small number of global loads that are separated by a few simple compute instructions and/or shared memory stores before arriving in a real compute region with a large number of short-latency instructions. When these global loads are close together, shifting priority upon a stall on every long latency load leaves several small compute regions which are insufficient to overlap with the subsequent global loads. Therefore, we want to make a different approach in saving parallelism where warps with long latency are gathered closely together. To implement that, the proposed warp scheduling policy then tries to make those warps overlap each other and scheduled them first (see in Part (c)). Hence, the thread parallelism can be saved and will be used later when the compute region is not sufficient to overlap the subsequent long latency.

Part (b, d) show the effect of the scheduling strategies at the warp level on the CTA level scheduling in cases of round-robin and the proposed scheduling. In the example, we assume a workload where 4 is the maximum number of CTAs allocating to each streaming multiprocessor. As depicted in Fig. 2 (Part c), when the warps with long latency (L-warps) are scheduled before warps with short latency (S-warps), it is likely that L-warps will overlap their latencies each other and the remaining parallelism will be used when L-warps are not sufficient. We suppose that this is an effective way to hide the long latency because it not only saves parallelism for necessary cases, but also does not cause streaming multiprocessors idle. Such scheduling strategy at the warp level impacts on scheduling at the CTA level. Indeed, CTAs finish its execution time slot with a shorter latency compared to round-robin scheduling policy (Part d). Consequently, the degree of hardware resources utilization increases, resulting in higher overall performance of GPGPU.

In summary, the proposed warp scheduling policy has a new way of scheduling warps to SIMT lanes of a streaming multiprocessor but it positively impacts on CTA scheduling. To classify warps in a CTA, a mechanism to track the number of stall cycles caused by each warp is necessary. This mechanism can be implemented simply by using a counter and some hardware storage to save the value of the

Algorithm 1

Step 1: Monitoring the number of stall cycles that is caused when executing a particular warp. These stall cycles (caused by *control hazards*, *RAW hazards*, and *pipeline stalled*) delay a warp's instruction from getting executed. The number of stall cycles is accumulated.

Step 2: After collecting information about the stall cycles of each warp in a particular thread block, all warps can be classified into 2 groups: L-warps (cause large amount of stall cycles, meaning long latency) and S-warp (cause small amount of stall cycles, meaning short latency)

Step 3: The warp scheduler will try to schedule L-warps before S-warps meaning that L-warps have higher priority than S-warps in the output scheduling list (which decides which warp is issued first). To classify all warps in a CTA into L-warps and S-warps, a threshold is required. However, our idea tries to save warp parallelism by utilizing L-warps to hide long latencies each other before the remaining warps in a CTA. Therefore, in implementation, we can choose a simple way like sorting warps in descending order of amount of stall cycles.

counter. The information about the number of stall cycles will decide the priority of each warp in a thread block. Algorithm 1 provides further information about the proposed warp scheduling policy.

Although the proposed warp scheduling policy can improve the overall performance of GPU, we observe that it may cause an extra amount of interconnection network or memory bandwidths. Originally, GPGPUs provide limited network and memory bandwidth and depending on the number of streaming multiprocessors and the characteristics of applications, there can be a bottleneck in GPGPU performance [5]. This issue is becoming increasingly critical. As we mentioned above, when L-warps, which cause a large number of stall cycles, are scheduled closely together, the number of memory accesses or interaction among streaming multiprocessors will increase and, therefore, the memory and interconnection bandwidth also increase. In the worst case, this causes interconnection and memory stalls, indirectly degrading GPGPU performance, thus reducing the effectiveness of our warp scheduler.

Current thread block schedulers (e.g. those used in Fermi and Kepler GPGPUs), which are responsible for thread block scheduling - i.e., distributing the thread blocks to the streaming multiprocessor, attempt to allocate maximum

Algorithm 2

```
Initialize:
1.  n_issued_SM = max_n_SM
2.  last_n_issued_SM = n_issued_SM
3.  last_n_gpu_cycle = 0
4.  last_n_gpu_cycle = 0
5.  last_n_gpu_stall_icntSM = 0
6.  last_contention_degree = 0
Monitor:
7.      n_gpu_stall_dramfull
8.      n_stall_icnt2SM
Calculate:
9.      delta_n_gpu_stall_dramfull = n_gpu_stall_dramfull
          - last_n_gpu_stall_dramfull
10.     delta_ n_stall_icnt2SM = n_stall_icnt2SM
      - last_n_stall_icnt2SM
11.   delta_gpu_cycle = n_gpu_cycle - last_n_gpu_cycle
Calculate:
12.   contention_degree=( delta_n_gpu_stall_dramfull+
      delta_ n_stall_icnt2SM) /delta_gpu_cycle
13. if (contention_degree > last_contention_degree)
      && (n_issued_SM > 2)
14. then  n_issued_SM  = n_issued_SM - 1
15. else if (contention_degree < last_contention_degree)
      && (n_issued_SM > max_n_SM -1)
16. then n_issued_SM  =  n_issued_SM -
17. else
18. then n_issued_ SM = last_n_issued_SM

Update:
19. last_n_gpu_cycle = n_gpu_cyle
20. last_n_gpu_stall_dramfull = n_gpu_stall_dramfull
21. last_n_gpu_stall_icnt2SM = n_gpu_stall_icnt2SM
22. last_contention_degree = contention_degree
23. last_n_issued_SM = n_issued_SM
```

number of thread blocks per-streaming multiprocessor. The maximum number of thread blocks assigned to each streaming multiprocessor depends on the resource usage of the workload, including the amount of registers, shared memory, etc. [7, 28]. Once a particular thread block finishes, the thread block scheduler assigns another thread block to that particular streaming multiprocessor, until all of the thread blocks have been assigned to the streaming multiprocessors.

Prior research work [5, 8], however, has shown that executing the maximum possible number of thread blocks on a streaming multiprocessor is not always the optimal choice from the performance perspective due to inefficient utilization of streaming multiprocessor resources. Indeed, when the thread block scheduler always assigns the maximum thread blocks to a streaming multiprocessor, it might cause a higher number of memory and interconnection network stalls. This issue directly causes pipeline stall, resulting in inactive status of the streaming multiprocessor, and thus, performance degradation.

We propose a supplemental technique to alleviate the issue that may cause by the proposed warp scheduling policy (see Algorithm 2). More specifically, we use a simple mechanism that monitors the number of memory stalls and interconnection network stalls during a period of GPGPU cycles. Every time the process of assigning new thread blocks to streaming multiprocessors occurs, the mechanism will check the contention degree (of the memory and interconnection network) by making comparison with the last contention degree. When the contention degree is greater than previous value, the number of streaming multiprocessors that will be assigned new thread blocks in the next cycle is decremented. Otherwise, the number will be incremented. In case we cannot increase or decrease the number of streaming multiprocessors that will be assigned thread blocks for the next round of round-robin scheduling, it will be kept unchanged. By doing that, the number of thread blocks assigned to streaming multiprocessors will be adjusted dynamically depending on the degree of DRAM-full and network stalls. As a result, the proposed technique can reduce the memory and network bandwidth when necessary.

Hardware Overhead: To support the proposed warp scheduler, each hardware warp is allocated a private counter to tracks the stall cycles. As the maximum number of warps per streaming multiprocessor 32 (baseline architecture), the number of counters is 32 per streaming multiprocessor. The proposed warp scheduler does not require further storage since it only needs the accumulated counter value. Although there is latency to sort the warps according to stall cycle values, this process does not cause the overall latency since it is substituted for the baseline process that needs to make the next cycle prioritized warp list. This list is used by the warp scheduler to know the warp scheduling order.

5. Experiments Methodology

We modified GPGPU-sim version 3.2 [7] to implement the proposed warp scheduling policy. GPGPU-sim is a cycle-level performance simulator that models a general-purpose GPU architecture supporting NVIDIA CUDA [2] and its PTX ISA. We ran GPGPU-sim with the default configuration representing the NVIDIA Fermi GTX480 architecture. The details of the baseline platform configuration used in this work are presented in Table 1 and Table 2. To evaluate the performance of our proposed techniques, we use benchmarks from CUDA SDK toolkit [29] and ISPASS [7]. Table 3 shows the list of selected benchmarks and some characteristics of the benchmarks.

Table 1. Baseline System Configuration

Architecture	NVIDIA Fermi GTX480
# of shader cores	15
Warp size and SIMD width	32
# of threads/SM	1024
# of registers/SM	32768
Shared memory/SM	48KB
L1 Data Cache	16KB per SM (32-sets/4-ways)
L1 Inst Cache	2KB per SM (4-sets/4-ways)
L2 Cache	768KB (64-sets/16-ways/6-banks)
Min. L2 access latency	120 cycles
Min. DRAM access latency	220 cycles
# of memory controllers	12
# of memory chips/controllers	2
Memory channel bandwidth	4KB
DRAM request queue size	32
GDDR3	tCL = 10, tRP = 10, tRC = 35, tRAS = 25, tRCD = 12, tRRD = 8

Table 2. Interconnection Network Configuration

Parameter	Value
Topology	Fly
Routing mechanism	Destination tag
Routing delay	0
Virtual channels	1
Virtual channel delay	0
Virtual channel buffers	8
Virtual channel allocator	iSLIP
Input speedup	2
Output speedup	1
Internal speedup	1
Flit size	32 bytes

Table 3. Simulated Benchmarks

#	Suite	Application	Total CTA/kernel	Concurrent CTAs/SM
1	NVIDIA CUDA SDK	VectorAddition	196	6
2	NVIDIA CUDA SDK	SimpleMultiGPU	32	6
3	NVIDIA CUDA SDK	Monte-Carlo	193	7
4	NVIDIA CUDA SDK	MergeSort	8156	7
5	ISPASS	StoreGPU	384	3
6	ISPASS	Mummer-GPU	196	4

6. Experimental Results and Discussion

In this section, we show the impact of our scheduling schemes on GPGPU performance. We start by showing the performance obtained by the proposed warp scheduling policy. Fig. 3 presents the IPC of our evaluated schedulers, normalized to Round-Robin scheduling policy. Across the applications used in this work, we observed that the proposed warp scheduler achieved performance improvement over the round-robin policy by an average of 7.5%. Moreover, the proposed warp scheduler did not cause any performance degradation in the selected applications (up to 21% improvement in the case of MUMmerGPU).

We also compared the proposed warp scheduler to a greedy scheduler [14]. In fact, there are many variations of greedy scheduling schemes, for example greedy-then round-robin (GRR) and oldest-first (GTO). In summary, a greedy scheme runs a single warp until it stalls and then picks another ready warp to issue. Warp age is determined by the time the warp is assigned to the shader core (streaming multiprocessor). For warps that are assigned to a streaming multiprocessor at the same time (i.e., they are in the same thread block), warps with the smallest scalar warp IDs are prioritized. Overall, previous works proved that a greedy warp scheduling policy performs better than round-robin warp scheduling policy. In this work, we also compared the proposed warp scheduling policy to the GTO because it provides the best results among other variations. As shown in Fig. 3, the proposed policy outperformed the GTO in all cases except MergeSort although the improvement is smaller compared to the round-robin policy (about 2% on average). The reason behind the little performance improvement of MergeSort is due to the negative impact of the proposed technique that we will explain later.

As mentioned in Section 4, in the worst case, the proposed warp scheduler may cause a higher amount of memory and interconnection network bandwidth due to consecutively scheduling warps that causes many stall cycles. This leads to a fact that always assigning the maximum number of thread blocks to all streaming multiprocessors is not always an effective way to utilize hardware resources. To alleviate this issue, we proposed a supplemental technique for the proposed warp scheduling policy (see Algorithm 2) that can dynamically reduce the number of thread blocks that are assigned to streaming multiprocessors when encountering a high

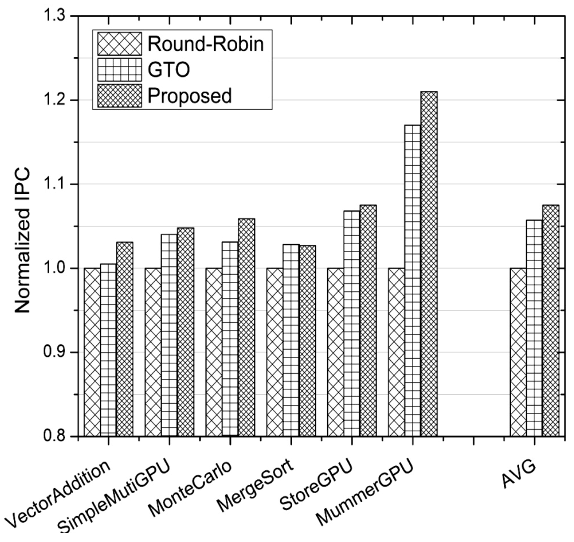


Fig. 3. IPC of Proposed Warp Scheduling Normalized to RR

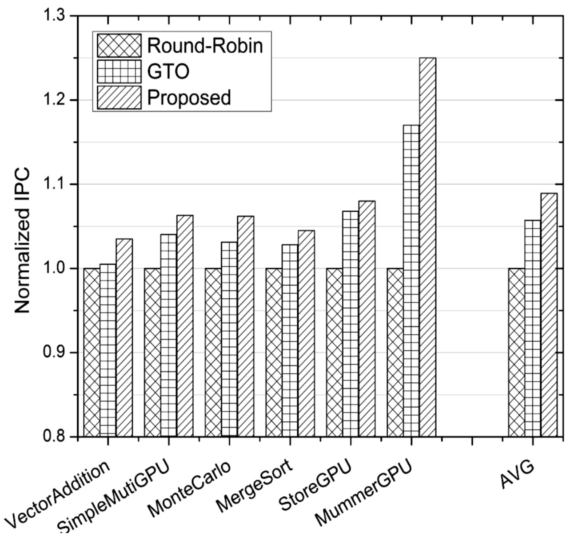


Fig. 4. Normalized IPC with Different Scheduling Schemes

degree of contention (in terms of memory and inter-connection network bandwidth). Fig. 4 shows the IPC performance of various scheduling policies normalized to the round-robin scheduling policy. Overall, with the technique in Algorithm 2, our warp scheduler performs better than the original proposal and can achieve a mean improvement over the round- robin scheduler by 8.9% and over the GTO by 3.2%.

Based on the idea in Algorithm 2, we expect that it will be effective for applications that have a high degree of contention. After examining MUMmerGPU, SimpleMultiGPU and StoreGPU, which benefits more from reduction of the degree of contention compared to the remaining applications, we see that these cases a large ration of dram-full and interconnection-network stalls to the GPU cycles is much large. Therefore, it can make room for improvement after the application of Algorithm 2. In contrast, the corresponding ratio for the remaining cases (VectorAddition, MonteCarlo and StoreGPU) is quite small, thus, resulting in a very slight improvement compared to the original proposed algorithm. In fact, although we cannot receive that kind of information until execution finishes when executing random applications in real systems, it proves the correctness of the technique in Algorithm 2.

7. Conclusion

This paper proposes a new warp scheduling policy to enhance GPGPU performance by overcoming the resource underutilization problem of the traditional round-robin scheduling policy. The idea of the proposal is based on classification of warps within a thread block into two groups, warps that cause long latency and warps that cause short latency and then assigning the warps with long latency higher priority than the remaining warps. This seeks to save the thread parallelism in case there is not enough parallelism to hide long latency operation. This paper also proposes a supplemental technique that alleviates the problem of memory and interconnection-network contention that may be caused by the originally proposed warp scheduler. Our experimental evaluations, derived from a 15-streaming multiprocessor GPGPU platform, demonstrate that the proposed warp scheduler outperformed the commonly- used round-robin warp scheduler, leading to an IPC performance improvement of 7.5%, and up to 8.9% when applying the supplemental technique.

References

[1] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and Wen-Mei W. Hwu, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, pp.73-82, 2008.

[2] NVIDIA. "CUDA C Programming Guide," 2012.

[3] M. Garland et al., "Parallel Computing Experiences with CUDA," *MICRO, IEEE*, Vol.28, No.4, 2008.

[4] A. Munshi, "The OpenCL Specification," Version 1.2, Khronos OpenCL Working Group, 2011.

[5] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-Level Parallelism for GPGPUs," in *CSE Penn State Tech Report*, TR-CES-2212-006, 2012.

[6] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture,* ACM, pp.308-317, 2011.

[7] A. Bakhola, G. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU simulator," in *Proceedings of the 2009 International Symposium on Analysis of Systems and Software (ISPASS-2009)*, pp. 163-174, Apr. 2009.

[8] M. Lee et.al, "Improving GPGPU Resource Utilization through Alternative Thread Block Scheduling," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp.260-271, 2014.

[9] V. V. P. Harish and P. J. Narayanan, "Large graph algorithms for massively multithreaded architectures," in Technical report, IIIT, 2009

[10] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU Computing," in *Proceedings of the IEEE*, Vol.96, No.5, pp.879-899.

[11] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization Principles and Aapplication Performance Evaluation of a Multithreaded GPU Using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* ACM, pp.73-82, 2008.

[12] V. Volkov and J. W. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp.1-11, 2008.

[13] M. Gebhart, R. D. Johnson, D. Tarjan, S. W. Keckler, W. J. Dally, E. Lindoholm, and K. Skadron, "Energy-efficient Mechanisms for Managing Thread Context in Throughput Processors," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pp.235-246, 2011.

[14] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO)*, pp.72-83, 2013.

[15] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," in *Proceedings of the 40th Annual IEEE/ ACM International Symposium on Microarchitecture,* IEEE Computer Society, pp.407-420, 2007.

[16] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO),* IEEE Computer Society, pp.213-224, 2010.

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

[18] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U. Weiser, "Many-Core vs. Many-Thread Machines: Stay Away From the Valley," *Computer Architecture Letters*, Vol.8, No.1, pp.25-28, 2009.

[19] H.-Y. Cheng, C.-H. Lin, J. Li, and C.-L. Yang, "Memory Latency Reduction via Thread Throttling," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp.53-64, 2010.

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

[21] J. D. Owens et al., "A Survey of Genera-Purpose Computation on Graphics Hardware," in Eurographics 2005, State of the Art Reports, pp.21-51, Aug., 2005.

[22] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp.356-367, 2011.

[23] K. Krewell, "AMD's Fusion Finally Arrives," Microprocessor Report, 2011.

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

[25] NVIDIA, Whitepaper: NVIDIA's Next Generation CUDA Compute and Graphics Architecture: Fermi.

[26] NVIDIA, "NVDA Tegra Multiprocessor Architecture," Feb. 2010.

[27] J. Chen et al., "Guided Region-Based GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency," in *Proceedings of the IEEE 27th International Symposium on Parallel and Distributed Processing*, pp.441-451, 2013.

[28] D. Kirk, "NVIDIA CUDA Software and GPU Parallel Computing Architecture," in *ISMM,* pp.103-104, 2007.

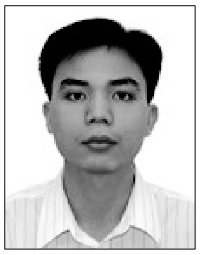
[29] NVIDA, CUDA SDK [Internet], http://developer.nvidia.com/gpu-computing-sdk.

[30] A. Jog et al., "OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp.395-406, 2013.

[31] S.-Y. Lee, A. Arunkumar, and C.-J. Wu, "CAWA: Coordinated Warp Scheduling and Cache Prioritization for Critical Warp Acceleration of GPGPU Workloads," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp.515-527, 2015.

[32] W. Jia, K. Shaw, and M. Martonosi, "MRPB: Memory Request Prioritization for Massively Parallel Processors," in *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp.272-283, 2014.

[33] X. Xie et al., "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp.76-88, 2015.



Do Cong Thuan

e-mail : congthuan.hut@gmail.com

He received the Engineer's degree from Hanoi University of Science and Technology, Hanoi, Vietnam in 2012, Master's degree from Chonnam National University, Gwangju, Korea in 2014. Currently, he is pursuing his Ph.D. at Chonnam National University. His research interests include computer architecture, parallel processing, microprocessors, embedded systems and GPGPU.

Yong Choi

e-mail : potchy0927@gmail.com

He received the Engineer's degree from Chonnam National University, Gwangju, Korea in 2015, Master's degree from Chonnam National University, Gwangju, Korea in 2017. Currently, he is pursuing his Ph.D. at Chonnam National University. His research interests include computer architecture, parallel processing, microprocessors, embedded systems and GPGPU.

Jong Myon Kim

e-mail : jongmyon.kim@gmail.com

He received the B.S. degree in electrical engineering from the Myong-Ji University, Yong-In, Korea, in 1995, the MS degree in electrical and computer engineering from University of Florida, Gainesville, in 2000, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, in 2005. He was a senior research staff in the Chip Solution Center of Samsung Advanced Institute of Technology from 2005 to 2007. Since 2007, he has been with the School of Electrical Engineering at the University of Ulsan, Ulsan, Korea, where he is currently a Professor. His research interests include embedded systems, application-specific processors, and parallel processing.

Cheol Hong Kim

e-mail : chkim22@chonnam.ac.kr

He received the B.S. degree in Computer Engineering from Seoul National University, Seoul, Korea in 1998 and M.S. degree in 2000. He received the Ph.D. in Electrical and Computer Engineering from Seoul National University in 2006. He worked as a senior engineer for SoC Laboratory in Samsung Electronics, Korea from Dec. 2005 to Jan. 2007. Now he is working as an Associate Professor at School of Electronics and Computer Engineering, Chonnam National University, Korea. His research interests include embedded systems, mobile systems, computer architecture, SoC design, low power systems, and multiprocessors.