J Inf Process Syst, Vol.17, No.6, pp.1157~1169, December 2021
https://doi.org/10.3745/JIPS.01.0084

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

KAWS: Coordinate Kernel-Aware Warp Scheduling and Warp Sharing Mechanism for Advanced GPUs

Viet Tan Vo* and Cheol Hong Kim**

Abstract

Modern graphics processor unit (GPU) architectures offer significant hardware resource enhancements for parallel computing. However, without software optimization, GPUs continuously exhibit hardware resource underutilization. In this paper, we indicate the need to alter different warp scheduler schemes during different kernel execution periods to improve resource utilization. Existing warp schedulers cannot be aware of the kernel progress to provide an effective scheduling policy. In addition, we identified the potential for improving resource utilization for multiple-warp-scheduler GPUs by sharing stalling warps with selected warp schedulers. To address the efficiency issue of the present GPU, we coordinated the kernel-aware warp scheduler and warp sharing mechanism (KAWS). The proposed warp scheduler acknowledges the execution progress of the running kernel to adapt to a more effective scheduling policy when the kernel progress attains a point of resource underutilization. Meanwhile, the warp-sharing mechanism distributes stalling warps to different warp schedulers wherein the execution pipeline unit is ready. Our design achieves performance that is on an average higher than that of the traditional warp scheduler by 7.97% and employs marginal additional hardware overhead.

Keywords

GPU, Multiple Warp Schedulers, Resource Underutilization, Warp Scheduling

1. Introduction

Over the past decade, graphics processor units (GPUs) have become an attractive platform as these provide a remarkable computing paradigm for graphical applications. Incorporated with substantial amounts of parallel logical units and fabricated using an advanced semiconductor process, GPUs are displaying increasing potential for general-purpose GPU (GPGPU) applications. The key features of a GPU are its high degree of throughput and exceptional computation capability. From a programming perspective, the CUDA and OpenCL platforms are increasing its popularity among programmers and researchers. This has resulted in several attempts to improve the performance and energy consumption of GPUs. The computational capacity of GPUs originates mostly from its capability for instant context-switching to hide long latency instructions and a substantial level of multi-threading. In general, a GPU achieves this using a warp scheduler. At each cycle, the warp scheduler iterates through a pool of ready warps (a group of 32 threads) to issue warp instructions for the next execution pipeline.

Manuscript received May 17, 2021; first revision July 9, 2021; accepted July 17, 2021.

Corresponding Author: Cheol Hong Kim (cheolhong@ssu.ac.kr)
* ICT Convergence System Engineering, Chonnam National University, Gwangju, Korea (vviettansa@gmail.com)
** School of Computer Science and Engineering, Soongsil University, Seoul, Korea (cheolhong@ssu.ac.kr)

The traditional warp scheduler policy (loose round-robin [LRR]) assigns equal priority to each warp in the scheduler's scheduling list. Thereby, all the warps achieve equivalent progress and attain long latency instructions almost simultaneously. This dramatically reduces the latency-hiding capability. LRR effectively exploits an inter-warp locality. However, it cannot utilize intra-warp locality since a nature of constant changing warp to issue at each cycle. Another popular warp scheduler is greedy then oldest (GTO) [1]. It is a standard for comparison. GTO prioritizes warps in a better manner. Intra-warp locality is preserved because the GTO scheduler continues to issue the same warp until it stalls, before moving to the oldest warp to be issued. The warp scheduler plays a crucial role in GPU performance. Several studies have proposed various warp scheduler schemes to address different aspects of GPU architectures. However, this is not the case for advanced architectures. Substantial upgrades over several generations (including advanced memory hierarchy, high bandwidth memory, numerous execution units, and multiple warp schedulers) have solved diverse limitations of the GPU. This has also reduced the impact of the warp scheduler on the overall performance. Our experiments (the simulation setup is described in Section 4) show that GTO outperforms LRR by 4.7% on an average for nine applications. Therefore, to further improve the performance of advanced GPU architectures, we need to analyze the operation of the warp scheduler following a novel approach and provide a supplementary technique.

Most earlier warp schedulers consistently applied the scheduling algorithm for the entire kernel execution. This is notwithstanding that the execution process involves inconsistent execution patterns over the kernel execution process. During one period of kernel execution, scheduling policy A may be more effective than scheduling policy B. In other periods, the converse may be true. In this study, we reveal the ineffectiveness of the application of the same warp scheduling policy over the entire kernel execution process. We solve this problem by proposing a simple kernel-aware warp scheduler that switches to a more effective scheduling policy when the kernel execution attains a certain milestone. Another aspect that has not been considered is the operation of multiple warp schedulers. Previous studies considered warp schedulers in streaming multiprocessors (SMs) to be independent of each other. However, each warp scheduler shares the execution workload whereby, scheduling activities may influence each other. Based on this observation, we propose a warp-sharing mechanism that enables each warp scheduler to acknowledge others warp schedulers' activities. Consequently, it provides a more effective scheduling operation among multiple warp schedulers in an SM.

This paper is organized as follows: Section 2 discusses the background organization of the baseline GPU and reviews the related work. In Section 3, we describe our proposed concept in detail. In Section 4, we evaluate our implementation, analyze its advantages, and compare it with the common warp schedulers. Section 5 concludes the paper.

2. Background

2.1 GPU Architecture

An advanced GPU architecture leverages multiple SMs to address parallel computing problems. From a memory perspective, each SM is equipped with an advanced memory hierarchy consisting of register files (RF), L1 cache, shared memory, L2 cache, and off-chip GDDR DRAM. The L1 cache is private for each SM and is responsible for caching temporary register spills of complex programs. In addition, according to the SM configuration, shared memory is programmer visible. This enables intra-CTA communication to increase the reusability of on-chip data. The requests are forwarded to the L2 cache via interconnection if these miss the L1 cache. To handle miss requests at the L2 cache, the memory controller schedules the memory requests to the DRAM. At each level, missed requests are tracked by miss-status holding registers (MSHRs). Fig. 1 shows the detailed microarchitecture of modern GPUs.

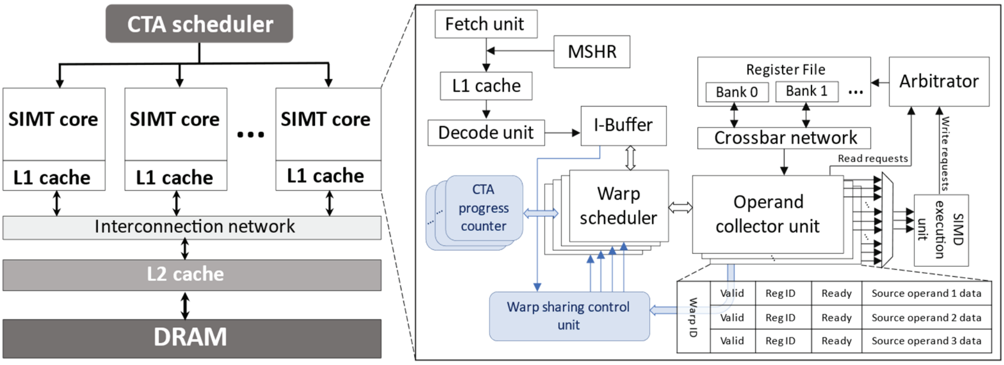

Fig. 1. Baseline GPU architecture. The blue components represent additional hardware in this study.

Inside an SM, a scalar front-end fetches, decodes, and stores instructions in the instruction buffer (I-buffer) indexed by warp ID. Warp instructions are scheduled and issued to the execution pipeline by the warp schedulers. In advanced GPU architectures, there are typically two or four warp schedulers in an SM. Each warp scheduler can issue a maximum of two instructions from the I-buffer per cycle to the available GPU cores. An operand collector unit (OCU) is assigned to a warp instruction when it is issued. This is performed to load a source operand value for it. Source operand requests are queued by an arbitrator and then sent to the RF. To reduce interconnect complexity and area cost, the OCU is designed as a single-ported buffer so that it receives only one operand per cycle [2]. Each instruction can have a maximum of three source operands. It may require more than three cycles if bank conflict occurs in the RF. During this time, the OCU is not available to receive new warp instruction. When all the required operands are collected, the instruction is ready to be issued to the SIMD (single instruction, multiple data) execution unit—streaming processor (SP), special function unit (SFU), and memory (MEM). In the Pascal architecture, four SP, four SFU, and four MEM pipeline widths correspond to four warp schedulers. The SP unit executes ALU, INT, and SP operations. The SFU unit executes double-precision (DP), sine, cosine, log, and other operations. The MEM unit is responsible for load/store operations. Each unit has an independent issue port from the OCU. Therefore, the warp scheduler cannot issue new warp instructions corresponding to an execution unit if the OCU of that unit is stalled.

In the GPU programming model, the GPGPU application consists of one or several kernels. Each kernel groups the threads into cooperative thread arrays (CTAs). The threads within a CTA execute in a group of 32 threads called warp. All the threads in a warp execute instructions in a lock-step manner (single-instruction multiple thread). At the SM level, the CTA scheduler assigns CTAs to each SM in each clock cycle until it attains the SM's resource saturation [3]. The maximum number of CTAs that can concurrently run in an SM is determined by the number of threads per CTA, the configured architecture limit number, the per-thread register, and the shared memory usage. When all the threads in the CTA are completed, the CTA is marked as complete and new CTA is assigned in the next cycle.

2.2 Related Works

Many prior studies have leveraged warp schedulers to improve latency-hiding capability, reduce branch divergence, or increase cache performance. In this section, we discuss certain well-known scheduling schemes and compare these with our proposed scheduler. CCWS [1] monitors cache contention by using a locality detector to throttle the number of active warps if the detector observes frequent cache eviction. Although this scheduler can reduce cache contention, it targets only cache-sensitive applications. Chen et al. [4] proposed adaptive cache management to overcome the limitations of the pure cache bypassing technique. The concept is built on top of warp throttling to protect hot cache lines and reduce cache contention. However, the management scheme still prefers cache-sensitive benchmarks while employing complex hardware overhead. CAWA [5] designed a criticality predictor to forecast critical warps in CTA. They proposed a criticality-aware warp scheduler that prioritizes critical warps and a cache reuse predictor to enhance warp execution speed. The underlying concept is that excessive work is required to design the instruction-based and stall-based predictor. This is in addition to the effort required to partition the data cache dedicated to critical warp execution. iPAWS [6] utilized existing warp schedulers such as GTO and LRR by switching to a suitable warp scheduler based on the instruction pattern. However, the performance gain is limited by the maximum performance of GTO and LRR. The implementation also requires execution time to trade for making warp-scheduler decisions. If this execution time accounts for a large portion of the total execution time, it can substantially reduce the performance. SAWS [7] addressed the synchronization issue of multiple warp schedulers. Liu et al. [7] designed schedulers that coordinate with each other to reduce the barrier waiting time and thereby, prevent warps from being stalled at a barrier for excessively long periods. The limitation of this study is that it is effective only for rich-barrier synchronization applications.

Only a few studies have been investigated to improve the resource utilization of GPUs by using warp schedulers. Our motivation in this work is to use the kernel-aware warp scheduling to fine-tune the resource allocation process by targeting the absence of multi warp scheduling policies during kernel execution. The proposed warp scheduler is differentiated from previous warp schedulers by employing two warp scheduling policies to effectively adapt to kernel execution progress. On the other hand, warp sharing mechanism is deviated from the lack of cooperation between multi warp schedulers within an SM. The combination of these two main ideas can improve the resource utilization of GPUs, resulting in significant performance gains.

3. Coordinate Kernel-Aware Warp Scheduling and Warp Sharing Mechanism

Resource underutilization occurs when the number of running CTAs in the SM is less than the maximum number of CTAs that the SM can handle. Thus, the key to the prevention of resource underutilization is to maximize the utilization of the SM by allocating the maximum number of CTAs to it. Fig. 2(a) illustrates how present GPUs waste resources during different periods of kernel execution. For convenient demonstration, we have used a GPU with two SMs. Assume that a kernel contains 32 CTAs and that each SM can manage a maximum of four concurrent CTAs. First, SM_0 and SM_1 receive four CTAs from the CTA scheduler (one in each cycle in a round-robin manner). Whenever a CTA completes execution, a new CTA is assigned to the empty CTA slot in the next cycle. At time t0, CTA26 completes its execution in SM_1. However, no new CTA remains in the kernel to fill the CTA slot. This leaves three CTAs to run freely in SM_1. From that instant, SM_1 starts to exhibit more severe resource underutilization. The number of running CTAs in SM_1 after the completion of CTA24 and CTA29 remain two and one, respectively. The non-availability of a sufficient number of CTAs to utilize all the available resources in an SM results in substantial wastage. This underutilization continues until the final CTA (CTA31) completes its execution. Subsequently, a new kernel is launched to fill the empty CTA slot in the SM. In SM_0, resource underutilization begins to occur from time t_1.

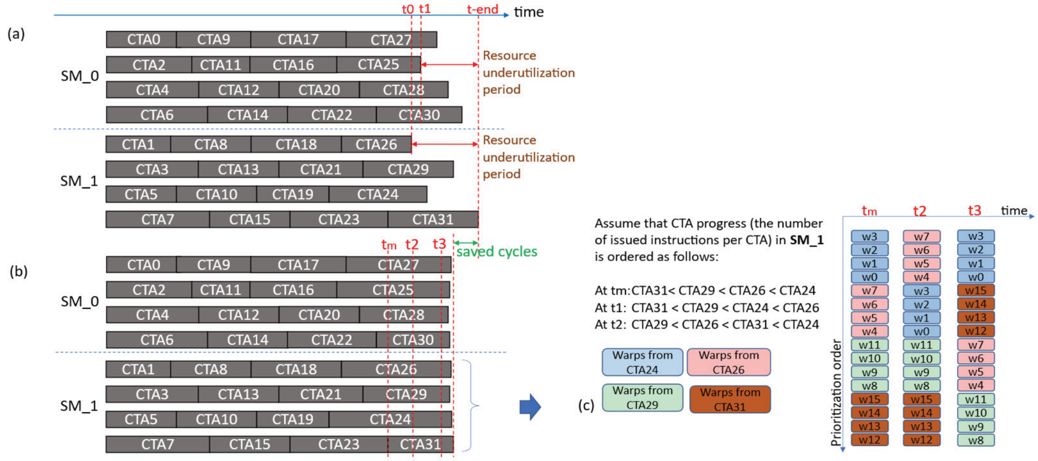

Fig. 2. Resource underutilization at the completion of kernel execution: (a) conventional kernel-unaware warp scheduler, (b) proposed kernel-aware warp scheduler, and (c) warp-level illustration for kernel-aware warp scheduler.

From the motivation example, we understand that resource underutilization occurs in SM_1 when any CTA in SM_1 (in this example: CTA26 at t0) completes its execution after SM_1 receives the final CTA (in this example, CTA31) from the CTA scheduler. In general, we consider the time when the final CTA is issued in SM_x as a milestone of resource underutilization. This is because within a short period from that time, SM_x would complete one running CTA without additional CTA being filled and thereby exhibit resource underutilization. However, it is difficult to identify the CTA that is issued last to SM_x. This is because each SM can receive a different number of CTAs depending on the speed with which the CTAs are executed. Therefore, we determine the time when the last CTA in the kernel is issued is the milestone of resource underutilization. After this milestone, a more efficient scheduling policy is required to reduce resource underutilization. We propose a kernel-aware warp scheduler. This scheduler prioritizes warps based on their ages during normal kernel execution. As soon as it detects the milestone when the final CTA in the kernel is issued, it switches to a progress-based prioritization policy to speed up the execution for more recently issued CTAs. Warps that belong to more recently issued CTAs are prioritized based on the CTA progress. We define the CTA progress as the number of instructions issued from one CTA. Older CTAs (or CTAs issued earlier) tend to issue more instructions and are likely to be completed early. Therefore, we deprioritize these to speed up younger CTAs (or CTAs issued more recently). Thereby, CTAs issued more recently (particularly the last issued CTA in a kernel) can be completed earlier whereas the execution time of previously issued CTAs can be prolonged. As a result, all the CTAs are adjusted for these to be completed almost simultaneously. This implies that the SM is filled with running CTAs most of the time. In Fig. 2(b), immediately as CTA31 (finally issued CTA in the kernel) is assigned (at time = t_m), a progress-based scheduling policy is applied. Initially, this policy provides the highest priority to warps that belong to CTA31 in SM_1 (illustrated in Fig 2(c)). For SM_0, warps from CTA30 are prioritized first. Then, the scheduler dynamically prioritizes warps based on the progress of CTA24, CTA26, CTA29 and CTA31 in SM_1; CTA25, CTA27, CTA28, and CTA30 in SM_0. Fig. 2(c) shows the CTA progress order for SM_1 at t2 and t3 to present how the proposed scheduler works at warp-level. At t2, warps in CTA31 are assigned the highest priority because the progress of CTA31 is slower than the other CTAs. Similarly, at t3, the warps from CTA29 are prioritized. Note that the warps from the same CTA have the same CTA progress, so the scheduler can prioritize them according to their age (oldest warp or smallest warp ID first). Consequently, in SM_1, the execution times of CTA31 and CTA29 can be reduced whereas those of CTA26 and CTA24 are prolonged. In SM_0, the progress-based policy speeds up CTA30 while lengthening the execution times of CTA25, CTA27, and CTA28. Finally, all the CTAs in the SM are completed almost simultaneously, which prevents resource underutilization and accounts for several saved cycles. Note that underutilization depends significantly on the execution time of the last issued CTA in the kernel.

Our proposed warp scheduler can significantly improve the performance when there are few CTAs in the kernel. If the kernel consists of many CTAs, the fraction of saved cycles out of the total execution time is marginal. Thereby, the performance improvement is marginal. As mentioned in the Introduction section, the impact of the warp scheduler, particularly the scheduler that consistently employs a single scheduling policy for the general performance, is not apparent in modern GPU architectures such as Pascal. We propose a supplemental concept to improve resource utilization from a different perspective. The motivation is that a warp scheduler cannot issue new warp instructions if the OCU, which is supposed to load operands corresponding to that specific warp instruction, is unavailable. Owing to the multiple-warp-scheduler configuration, OCUs from different warp schedulers are likely to be available. Our concept is to utilize these to maintain the pipeline and prevent pipeline stall. This concept is called “warp sharing mechanism.” Fig. 3 illustrates how the mechanism works. Assume that eight warps are scheduled in four warp schedulers. After the decode stage, we can determine (1) the type of instructions of each warp by scanning the I-buffer and (2) which execution units are supposed to execute these. S0 denotes Warp scheduler 0, and w0-mem indicates that Warp 0 is carrying an instruction that would be executed in the MEM unit. Similarly, *sp* and *sfu* denote the instructions to be executed by the SP and SFU units, respectively. Presently, S0 cannot issue w0 and w4 because the OCU for the MEM unit corresponding to S0 is unavailable. An identical scenario occurs with w6 at S2, and w3 and w7 at S3. Meanwhile, the OCU for the MEM unit corresponding to S1 is free. However, S1 does not issue warp instruction because the instructions w1 and w5 are supposed to issue the OCU for the SP and SFU units, respectively. We propose a new hardware component called “warp-sharing control unit.” This unit can be aware of all the OCU statuses and simultaneously collect instruction information from I-buffer to make warp-sharing decisions. In this example, the warp-sharing control unit enables the memory instruction of w0 to be issued by S1. Therefore, the OCU for the MEM unit of S1 is utilized. Although w4, w6, w3, and w7 also carry memory instruction, w0 is prioritized over these because it is the oldest warp. By applying the same rule, the warp-sharing control unit permits w5 and w1 to be issued by S2 and S3, respectively. This is because OCUs for the SFU and SP units in these warp schedulers are available. The primary goal of improving resource utilization is achieved.

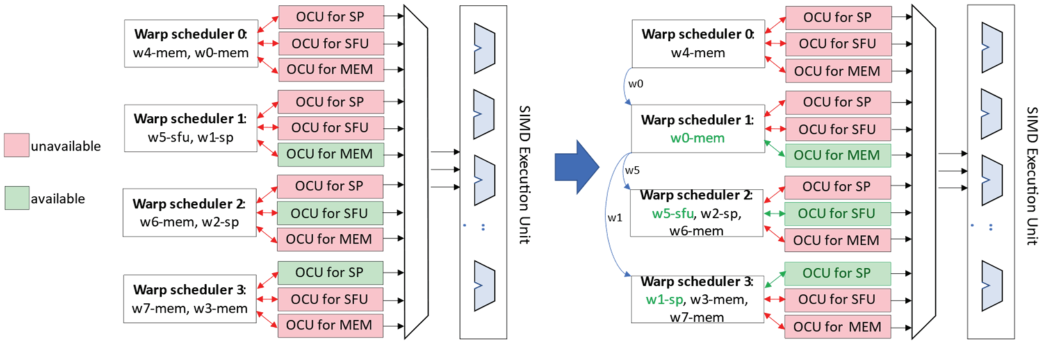
unavailable
available
Warp scheduler 0: w4-mem, w0-mem
Warp scheduler 1: w5-sfu, w1-sp
Warp scheduler 2: w6-mem, w2-sp
Warp scheduler 3: w7-mem, w3-mem
OCU for SP
OCU for SFU
OCU for MEM
SIMD Execution Unit
Warp scheduler 0: w4-mem
w0
Warp scheduler 1: w0-mem
w5
w1
Warp scheduler 2: w5-sfu, w2-sp, w6-mem
Warp scheduler 3: w1-sp, w3-mem, w7-mem
SIMD Execution Unit

Fig. 3. Warp-sharing mechanism.

In terms of hardware overhead for the kernel-aware warp scheduler, we used 32 instruction counters to maintain the entire CTA progress. We used a 4 B register per CTA to record the number of warp instructions issued during CTA execution. This consumes 4×32 bytes of additional storage, where 32 is the maximum number of CTAs per SM in the Pascal architecture. Therefore, 128 B per SM is adequate for storage overhead. As shown in Fig. 4(a), we use an additional comparator to be aware of the kernel execution. When the number of issued CTAs becomes equal to the total number of CTAs in the kernel (i.e., immediately after the CTA scheduler issues the final CTA in the kernel), the comparator sends a notification signal to the warp scheduler to switch to a progress-based scheduling policy. A more effective warp scheduler is selected to improve resource utilization.

Fig. 4(b) illustrates the hardware implementation of the warp-sharing mechanism. The warp scheduler and OCU for each execution unit are stacked in four layers. This illustrates that there are four warp schedulers and four corresponding OCUs for each execution unit inside the SM. The "warp sharing control unit" scans the opcode of each warp instruction (indexed by warp ID) from the I-buffer. Each warp instruction type is classified using an instruction classifier to identify the execution unit that should be employed for individual warp instruction. The output is the warp ID information corresponding to each execution unit, which indicates the warp that can be shared among warp schedulers. The shared warp IDs are transferred only to suitable warp schedulers via a transmission gate (TX1, TX2, and TX3) if the availability requirements are satisfied. The conditions to switch on the TX are combined by an AND gate fed by "in" input status from OCUs. The AND gate functions as a requirement examiner. It sends the enable signal to the TX only when the OCU conditions are satisfied. In general, the warp ID corresponding to the SP execution unit is passed through TX1 to Warp scheduler *x* if the OCU for the SP execution unit that connects to this warp scheduler is available. In addition, other OCUs for SP execution units that connect to other warp schedulers are unavailable. Thus, the OCU and SP execution units corresponding to Warp scheduler *x* are utilized. Similarly, TX2 and TX3 are dedicated to the SPU and MEM warp instruction transmissions, respectively. Our warp sharing control unit is designed to read the I-buffer and OCU statuses in each clock cycle between the decode stage and issue stage. The operation of this unit relies mainly on simple logic gates. Therefore, it is fast and reasonably low-cost owing to the inexpensive logic devices.

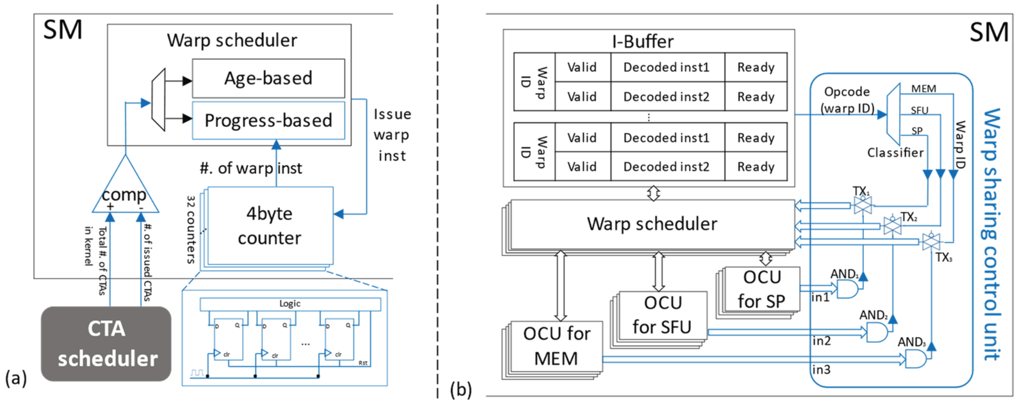

Fig. 4. Details of hardware implementation. Additional hardware (in blue) and how these communicate with existing components in the SM. (a) Employed comparator and counter for kernel-aware warp scheduler. (b) Warp sharing control unit to manage the sharing mechanism.

4. Evaluation

We implemented kernel-aware and warp-sharing concepts on the cycle-level simulator GPGPU-Sim v4.0 [8,9]. The simulator carves the GPU architecture in a C-like environment and provides a significantly high correlation with the actual GPU. In the fourth version, the simulator supports various highly advanced GPU architectures. We select the Pascal configuration (Titan X; it is a cutting-edge GPU architecture) as a baseline for comparison because it is highly effective, stable, and highly popular (as indicated by its market share). We consider that our implementation works effectively with other GPU generations because these continue to rely on Pascal, improve machine learning features, and/or reduce device size and energy consumption by using more advanced semiconductor processes. Table 1 lists the important parameters used in our configuration.

We evaluated the performance based on nine benchmarks—Streamingcluster (SC), Pathfinder (PF), B+tree (BT), Hotspot (HS), Mri-q (MRI), Quasirandomgenerator (QRG), 3Dconvolution (3DCV), three MatrixMultiplication (3MM), and ResNet (RN)—selected from several popular GPU benchmark suites: Rodinia [10,11], Parboil [12], Polybench [13], CUDA SDK, and Tango [14]. These benchmarks cover various aspects of actual GPU computations, including traditional cache-sensitive, memory-intensive, compute-intensive, and machine-learning computation on a GPU. Most of the benchmarks were simulated to completion or until the variation is negligible in the case of time-consuming benchmarks.

Table 1. Pascal configuration (Titan X)

Parameter	Value
SIMT	28 cores
Clock	1417 (core) : 1417 (interconnection) : 1417 (L2) : 2500 (Dram)
Max. # of CTA per SM	32
L1 data cache	24 kB, 48-way
L1 instruction cache	4 kB, 48-way
L2 cache	3 MB
# of memory controllers	12
# of warps scheduler per SM	4

4.1 Performance

Fig. 5 shows a comparison of the IPC performance of the KAWS with that of the default built-in warp scheduler. All the results have been normalized to the baseline configuration using the LRR policy. On an average, KAWS achieves a performance that is 7.97% and 3.26% higher than those of LRR and GTO, respectively. Note that although GTO is one of the most effective warp schedulers, its performance is only 4.7% higher than that of LRR on an average in advanced GPU architectures such as Pascal. Overall, the performance of the KAWS is higher than that of LRR for eight out of the nine benchmarks. In particular, KAWS shows a significant improvement in RN (19.91%), HS (19.64%), and 3DCV (13.88%) because KAWS displays a significantly high performance in terms of latency hiding. These benchmarks commonly execute uniform instruction patterns. LRR is likely to achieve equal warp progress among warps. The result is that warps attain long latency instructions almost simultaneously, which implies that LRR provides low latency hiding capability. MRI computes a matrix that represents the scanner configuration for calibration [15]. It has a large number of iterations and a strong inter-warp locality (which explains why LRR is the best scheduler for MRI), whereby the performance of GTO and KAWS degrades. However, KAWS performs more effectively than GTO in MRI. The SC is a memory-intensive benchmark. A large number of stall cycles caused by cache misses halts GTO and KAWS to result in an apparent performance gain over the LRR for this benchmark.

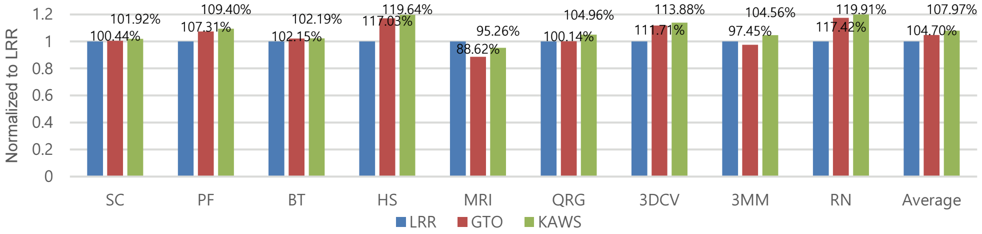

Fig. 5. Performance comparison.

As described earlier, KAWS prioritizes warps according to their ages during normal kernel execution. This is similar to GTO without greedy warp. Therefore, KAWS inherits the capability to hide latency by distributing unequal progress among warps. In general, KAWS exhibits higher performance than GTO for nearly all the evaluated benchmarks because it employs warp sharing and kernel-aware implementation. KAWS can achieve a performance gain of up to 7.5% and 7.3% over GTO in MRI and 3MM, respectively. These benchmarks substantially exploit matrix computations, which require continuous computation and memory instructions. Meanwhile, the warp-sharing mechanism is an effective means for increasing the usage of computing and memory execution units within an SM. BT is the only benchmark wherein KAWS achieves almost zero improvement over GTO. This is because BT is a rich-synchronization application. It comprises several parallel regions between barrier instructions, whereas KAWS is not designed to identify barrier instructions. In this case, sharing warp has no impact because warps are generally stalled at the barrier regardless of the scheduler that issues these.

4.2 Resource Utilization

The proposed kernel-aware warp scheduling operates to reduce the execution time of the last issued

CTA in the kernel while prolonging that of previously issued CTAs. This strategy results in an almost simultaneous completion of all the CTAs and thereby, a reduction in resource underutilization. Hence, the key metric to evaluate the effect of warp scheduling on SM utilization is the execution time of the last issued CTA in the kernel. Moreover, this time is also correlated to the execution time of the entire kernel. Table 2 shows a comparison of the average execution time of the last CTA in the kernel between KAWS and the GTO policy. We compare our design only with GTO because resource underutilization generally occurs in GTO. In LRR, prioritization is intrinsically distributed equally to all the warps, whereby CTAs are completed almost simultaneously. KAWS and GTO are not comparable with the LRR. On an average, KAWS reduces the last CTA execution time by 5.09% compared with that of GTO. In particular, there is a significant degradation of execution time in MRI and QRG. It corresponds to performance gains of 7.5% and 4.81%, respectively, over GTO. As mentioned in Section 3, kernel-aware warp scheduling contributes significantly to the overall performance when the number of CTAs in the kernel is not excessive. This is the reason for the significant improvement in MRI and QRG, which contain 128 CTAs per kernel. Although KAWS can considerably reduce the execution time of the last issued CTA in HS, it does not significantly improve the overall performance because the HS kernel consists of many CTAs. Meanwhile, KAWS increases the last CTA execution time by 3.35% in RN. However, its effect on the performance is compensated for by the effectiveness of the warp-sharing mechanism to prevent performance degradation.

Table 2. Average execution time (cycle) of last issued CTA in kernel

Benchmark	Number of kernels	Number of CTAs per kernel	Last CTA execution time		Normalized to GTO (%)
			GTO	KAWS	
SC	140	128	119511.6	117218.33	98.08
PF	5	463	11914.4	11631.6	97.63
BT	2	6000 or 10000	5300	5291	99.83
HS	1	1849	3372	2878	85.35
MRI	4	128	200378.3	186167.67	92.91
QRG	42	128	57937.19	48287.57	83.34
3DCV	254	256	2994.29	2859.68	95.50
3MM	3	1024	199260.7	195657.33	98.19
RN	38	64 or 256	300766.9	310851.34	103.35
Average	-	-	-	-	94.91

Fig. 6 presents a breakdown of the portion of SM inactive cycles, which are the cycles wherein no warps are issued. Inactive cycles can be categorized into the following three stalls. All the results are normalized to that of LRR:

- *Idle*, in which all available warps are issued to the pipeline, and none of these are ready to execute the next instruction. The following are possible reasons: warps are waiting at the barrier, empty I-buffer, and control hazard.
- *Scoreboard stall*, in which all available warps wait for data from memory. The scoreboard prevents WAW and RAW dependency hazards by tracking which registers would be written to but has not yet been written to because it is waiting for its results back to the register file.
- *Pipeline stall*, when all the execution pipelines are full regardless of having valid instructions with available operands. It occurs because of the limited number of existing execution units.

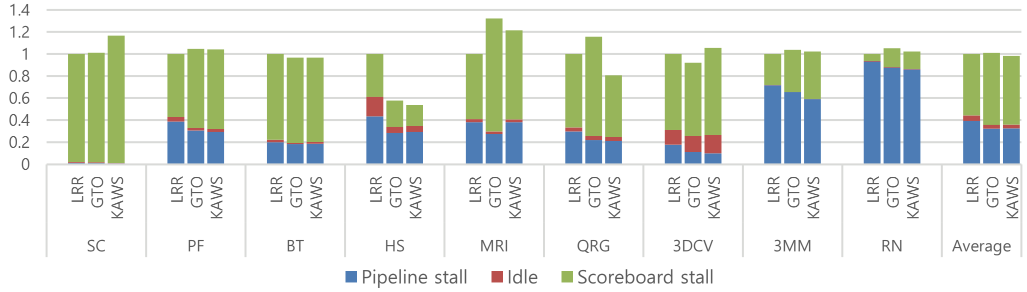

Fig. 6. Details of stalls in LRR, GTO, and KAWS scheduler.

The warp sharing mechanism enables warps that fall into the pipeline stall to be issued in different warp schedulers by available operand collector and execution units. That is, the available resources are utilized. Thus, pipeline stall reduction over most of the benchmarks is predictable. The larger the reduction in pipeline stalls and total stalls, higher is the increase in resource usage. Unlike pipeline stall, our concept introduces more scoreboard stalls compared with LRR. This is because it increases the communication traffic among multiple warp schedulers, which can cause conflict in the sharing of OCU. In general, this tradeoff is unavoidable and is more advantageous than a reduction in performance. KAWS presents a substantial decrease in total stall cycles in HS and QRG. This results in a good performance gain for these benchmarks. As shown in Fig. 6, KAWS produces more stall cycles in MRI, thereby degrading the IPC performance by 4.74% compared with that of LRR. Nonetheless, the performance of KAWS is higher than that of GTO by 7.5% owing to the coordination with kernel-aware warp scheduling. On an average, our implementation reduces pipeline stall and total stall by 17.35% and 1.82%, respectively, compared with those of LRR.

5. Conclusion

In this study, we analyzed the hardware underutilization owing to a deficiency of available CTAs when kernel execution approaches completion. Our kernel-aware warp scheduler switches to a progress-based scheduling policy immediately as the kernel releases its final CTA. The scheduler prioritizes warps based on the CTA progress in which warps are involved. The objective is to reduce the execution time of subsequently issued CTAs while prolonging the execution time of previously issued CTAs, so that the CTAs complete their execution almost simultaneously. Therefore, a new kernel can be launched earlier to prevent resource underutilization.

Moreover, we coordinated the kernel-aware warp scheduling and warp sharing mechanism (KAWS) to further improve hardware utilization. The controller detects the warp schedulers whose OCUs are free to share stalling warps from other warp schedulers. Thereby, pipeline stall is averted, and the available execution units are utilized. Although there is a drawback of escalation in scoreboard stall, it is compensated for by the reduction in pipeline stall and total stall under various types of workloads. Our experiments demonstrated that on an average, KAWS outperformed LRR and GTO by 7.97% and 3.26%, respectively. We plan to investigate the methods to adapt the proposed KAWS mechanism to more advanced warp schedulers for upcoming GPUs in the future.

Acknowledgement

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2021R1A2C1009031).

References

[1] T. G. Rogers, M. O'Connor, and T. M. Aamodt, “Cache-conscious wavefront scheduling,” in *Proceedings of 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Vancouver, Canada, 2012, pp. 72-83.

[2] H. Asghari Esfeden, F. Khorasani, H. Jeon, D. Wong, and N. Abu-Ghazaleh, “CORF: coalescing operand register file for GPUs,” in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, RI, 2019, pp. 701-714.

[3] C. T. Do, J. M. Kim, and C. H. Kim, “Application characteristics-aware sporadic cache bypassing for high performance GPGPUs,” *Journal of Parallel and Distributed Computing*, vol. 122, pp. 238-250, 2018.

[4] X. Chen, L. W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W. M. Hwu, “Adaptive cache management for energy-efficient GPU computing,” in *Proceedings of 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, UK, 2014, pp. 343-355.

[5] S. Y. Lee, A. Arunkumar, and C. J. Wu, “CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads,” *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 515-527, 2015.

[6] M. Lee, G. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu, “iPAWS: instruction-issue pattern-based adaptive warp scheduling for GPGPUs,” in *Proceedings of 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain, 2016, pp. 370-381.

[7] J. Liu, J. Yang, and R. Melhem, “SAWS: synchronization aware GPGPU warp scheduling for multiple independent warp schedulers,” in *Proceedings of 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, HI, 2015, pp. 383-394.

[8] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *Proceedings of 2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Boston, MA, 2009, pp. 163-174.

[9] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-Sim: an extensible simulation framework for validated GPU modeling,” in *Proceedings of 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, Valencia, Spain, 2020, pp. 473-486.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, “Rodinia: a benchmark suite for heterogeneous computing,” in *Proceedings of 2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, 2009, pp. 44-54.

[11] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, “A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads,” in *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*, Atlanta, GA, 2010, pp. 1-11.

[12] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, “Parboil: a revised benchmark suite for scientific and commercial throughput computing,” Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Champaign, IL, Technical Report No. IMPACT-12-01, 2012.

[13] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proceedings of 2012 Innovative Parallel Computing (InPar)*, San Jose, CA, 2012, pp. 1-10.

[14] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and J. Jeon, "Tango: a deep neural network benchmark suite for various accelerators," in *Proceedings of 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Madison, WI, 2019, pp. 137-138.

[15] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: a methodology for evaluating the error resilience of GPGPU applications," in *Proceedings of 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Monterey, CA, 2014, pp. 221-230.

Viet Tan Vo **https://orcid.org/0000-0002-7465-6243**

He received the B.S. degree in Electronics and Telecommunication Engineering from Ho Chi Minh City University of Technology, Ho Chi Minh, Vietnam in 2018. He is pursuing his M.S. degree in ICT Convergence System Engineering at Chonnam National University. His research interests include computer architecture, parallel processing, microprocessors, and GPGPUs.

Cheol Hong Kim **https://orcid.org/0000-0003-1837-6631**

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. He also worked as a Professor at Chonnam National University, Korea from 2007 to 2020. Now he is working as a Professor at School of Computer Science and Engineering, Soongsil University, Korea. His research interests include computer systems, embedded systems, mobile systems, computer architecture, low power systems, and intelligent computer systems.