Improving GPU Multitasking Efficiency Using Dynamic Resource Sharing

Jiho Kim, Jehee Cha, Jason Jong Kyu Park, Dongsuk Jeon, and Yongjun Park

Abstract—As GPUs have become essential components for embedded computing systems, a shared GPU with multiple CPU cores needs to efficiently support concurrent execution of multiple different applications. Spatial multitasking, which assigns a different amount of streaming multiprocessors (SMs) to multiple applications, is one of the most common solutions for this. However, this is not a panacea for maximizing total resource utilization. It is because an SM consists of many different sub-resources such as caches, execution units and scheduling units, and the requirements of the sub-resources per kernel are not well matched to their fixed sizes inside an SM. To solve the resource requirement mismatch problem, this paper proposes a *GPU Weaver*, a dynamic sub-resource management system of multitasking GPUs. GPU Weaver can maximize sub-resource utilization through a shared resource controller (SRC) that is added between neighboring SMs. The SRC dynamically identifies idle sub-resources of an SM and allows them to be used by the neighboring SM when possible. Experiments show that the combination of multiple sub-resource borrowing techniques enhances the total throughput by up to 26 and 9.5 percent on average over the baseline spatial multitasking GPU.

Index Terms—Computer architecture, GPUs, multi-programmed, resource sharing, spatial multitasking

1 Introduction

TODAY 's mobile and automotive systems require both high performance and energy efficiency to support various applications such as wireless signal processing, multimedia decoding and augmented reality. Multicore CPUs are the most popular to power the systems. However, heterogeneous systems, which add multiple accelerators to the traditional systems, have become more promising solutions for further improvement. Graphics processing units (GPUs) are the most successful accelerators as they show high throughput with sustainable power budget and are well supported by single instruction multiple thread (SIMT) programming models such as CUDA [9] and OpenCL [6]. Recent mobile application processors (APs) contain multiple CPU cores and a shared GPU, and hence efficient GPU resource sharing support is essential to enhance the performance of concurrent multikernel executions that are requested from different CPUs. Furthermore, efficient multitasking is expected to be more critical since the trend of recent mobile AP scaling is to increase the "*size*" of a shared GPU with a greater "*number*" of CPU cores.

To meet the multitasking requirement, many approaches have been introduced by both industry and academia. Nvidia's Hyper-Q [12], Multi-Process Service [13] and AMD's Multi-Process Sharing [11] enable multiple CPU threads/processes to schedule their tasks on a single GPU, but they support multitasking on software only, or use a simple FIFO-style resource allocation policy. Several recent developments by the academia have improved the efficiency of preemptive multitasking by introducing GPU-specific preemption techniques [16]. Spatial multitasking [1], which assigns a different number of Streaming Multiprocessors (SMs) to different applications, is one of the most common architectural solutions. Although spatial multitasking has proven quite effective, SM-level partitioning often incurs low resource utilization due to the mismatch between the requirement and the actual amount of sub-resources within an SM. For example, compute-intensive kernels assigned to several SMs may require more execution units but few memory accesses. A simultaneous multikernel (SMK) [20] technique can be a promising solution for the sub-resource low utilization problem as it supports multikernel execution within an SM. However, efficient orchestration of multikernel execution within an SM is quite complex [17] due to interferences of multikernel.

To overcome these inefficiencies, this study proposes the *GPU Weaver*, which is a light-weight approach to maximize resource utilization over spatial multitasking while avoiding interference and complexity of multikernel execution. In addition to Cache sharing [7], we propose a complete multitasking framework that contains three sub-resource sharing techniques: L1D caches, execution units, and scheduling structures. The purpose of this work is to solve the low sub-resource utilization problem for mapping multiple applications onto a shared GPU using spatial multitasking. We first design a configurable SM architecture that can fine-tune the amount of sub-resources to meet the requirement of running applications, referred to as the *Cooperative Streaming Multiprocessor (Co-SM)*. A Co-SM basically consists of a typical SM pair with a shared resource controller to perform sub-resource allocation and assignment to each SM. When two SMs inside a Co-SM execute different applications, the shared resource controller dynamically assigns an appropriate amount of each sub-resource per SM, and then recycles the under-utilized sub-resources by assigning them to be used by the other SM whenever possible.

2 Background and Motivation

Basic GPU Multitasking. GPUs consist of multiple SMs (Cores), memories (Caches) and interconnection networks. The most common method for multitasking is spatial multitasking [1] which divides GPU as an SM granularity for mapping multiple kernels. This is the simplest multitasking method because multiple SMs on a GPU do not share kernel states, and therefore, there is no need to consider any interference between multiple kernels in each SM. Though it shows considerable performance gain, sub-resource under-utilization within each SM exists. Because the number of concurrent threads on an SM is limited by one of several factors such as storages (RF, SharedMem) and scheduling units (PC, Stack). Non-critical resources are likely to be wasted when only single kernel runs on a single SM.

Execution Unit Resource Variance. Though SMs have their own sizes of streaming processors (SPs) and special function units (SFUs) [9], the execution unit utilization can vary highly according to the kernel characteristics. For example, execution units are idle for most of time in memory intensive kernels, but can be the performance bottleneck in compute-intensive kernels. Fig. 1 shows the relative execution cycles per different input buffer status to total execution cycles of three execution units: SP, SFU, and MEM. *Empty* means the target execution unit has no input values (idle), and *proper* means the unit has input values for only one cycle and therefore the inputs can be resolved at the cycle. While these two conditions do not incur pipeline stalls, *starve* may incur pipeline stalls as it means many input values are waiting to be processed at

- *J. Kim and J. Cha are with the Hongik University, Seoul 04066, South Korea. E-mail: {jihokimhi, carjehee}@gmail.com.*
- *J.J.K. Park is with the University of Michigan, Ann Arbor, MI 48109. E-mail: jasonjk@umich.edu.*
- *D. Jeon is with the Seoul National University, Seoul 151-742, South Korea. E-mail: djeon1@snu.ac.kr.*
- *Y. Park is with the Hanyang University, Seoul 04763, South Korea. E-mail: yongjunpark@hanyang.ac.kr.*

Manuscript received 9 July 2018; revised 16 Nov. 2018; accepted 17 Dec. 2018. Date of publication 20 Dec. 2018; date of current version 14 Jan. 2019.
(Corresponding author: Yongjun Park.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2018.2889042

1556-6056 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:57:35 UTC from IEEE Xplore. Restrictions apply.

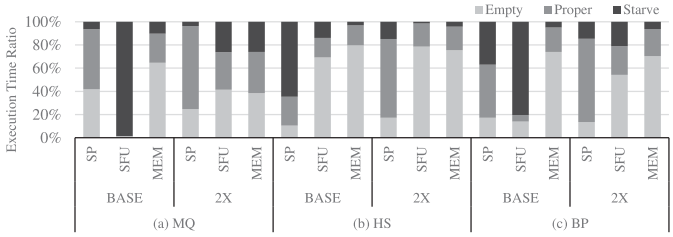


Fig. 1. Execution unit status categorization: the components of the bar indicate ratio of execution time in Empty, Proper, and Starve status.

the target execution unit and only the subset can be handled at the cycle. Therefore, the portion of execution cycles for *starve* status should be minimized. As shown in Fig. 1, there are a substantial number of cycles for *starve* status with the baseline (*base*) execution unit configuration (32 SPs, 4 SFUs). Among three compute-intensive benchmarks, MQ and BP performance is limited by SFUs, and HS is limited by SPs. An interesting question here is whether the ratio of *starve* status can be greatly reduced if more execution resources are assigned. In order to estimate the effectiveness, we also performed the same experiment with the 2x execution unit resources over the baseline configuration (*2x*). As shown, assigning more resources can be a successful performance enhancing approach as the ratio of cycles for *starve* status are highly reduced. Therefore, it will be highly effective if more execution unit resources can be assigned when the input buffer is full.

Scheduling Unit Resource Variance. In order to fully exploit Thread Level Parallelism (TLP), maximum allowable thread blocks should be assigned to each SM. The assigned number of thread blocks for each SM are determined by multiple factors [21] such as the number of issuable thread blocks, threads and the size of in-SM memory structures (register file and shared memory). Thread block and thread number limitation are due to the control logic complexity, and the in-SM memory structure limitation is due to the total register and shared memory requirement of all assigned threads. Therefore, threads with a small size of in-SM memory can be maximally assigned to SMs if the maximum issuable thread block and thread number increases. However, a simple increase of the thread control logic is not the best solution due to its high logic complexity. Fig. 2 depicts the relative utilization of four possible limiting factors, including size of register file, shared memory, number of threads and thread blocks. In this figure, we also categorize the applications according to their actual limiting factors. Applications in a *task* category are applications that do not contain enough threads to fully utilize at least one limiting factor. Applications limited by threads and thread blocks on the left side of Fig. 2 do not fully utilize the in-SM memory, and the other applications on the right side do not fully utilize the scheduling structures. This means that more threads can be assigned to SMs if left side applications can use more scheduling units, and they can be borrowed from the SMs that execute the applications on the right side without more complex scheduling structure implementation.

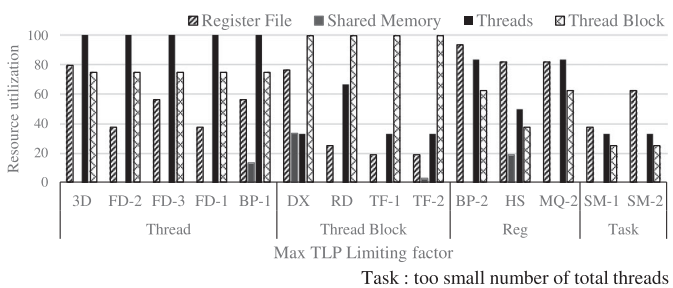


Fig. 2. Utilization of in-SM memory (RF and shared memory) and scheduling structures (thread and TB) when a resource is fully occupied.

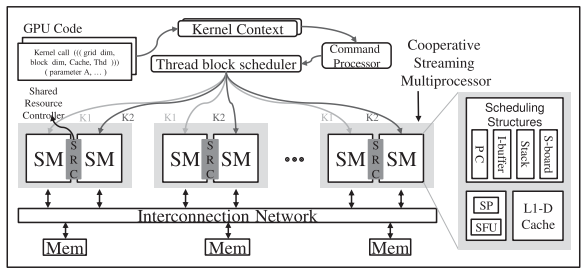


Fig. 3. GPU-weaver overview.

Previous Related Works and Insight. To exploit the sub-resources within an SM more efficiently, SMK [20], which schedules multiple kernels on the same SM has been proposed. However, sub-resource interference between multiple kernels are hard to eliminate when same type of kernels are running concurrently or unpredictable resource usage patterns are shown [17]. To avoid this interference, we decided to allocate multiple kernels to different SM subsets, and to borrow under-utilized sub-resource from the other SM subset only when possible. To achieve this, GPU Weaver performs conditional resource sharing according to the kernel, similar to the Conjoined-core [8] on the CPU side. Therefore, higher energy efficiency can be achieved by minimizing performance loss from interference with minimum HW overhead, based on spatial partitioning.

3 GPU Weaver Architecture

The GPU Weaver is composed of multiple Co-SMs. Fig. 3 shows a high level overview of the GPU Weaver with multiple Co-SMs. A pair of neighboring SMs forms a Co-SM. Each warp within a SM shares sub-resources of the other SM within the same Co-SM including the scheduling structures (PC, I-buffer, Stack, and Scoreboard) using Scheduling-Sharing, execution units (SP, SFU) using Ex-Sharing, and a L1 D-cache using Cache-Sharing [7], only when the neighboring sub-resources are temporally or spatially available. A Co-SM structure is not a simple double-sized SM, because it supports efficient sub-resource sharing mechanisms when executing multiple kernels. Each Co-SM has a Shared Resource Controller (SRC) to allow warp-instructions of a SM to use sub-resources of the other SM efficiently. In order to achieve high sub-resource utilization on Co-SMs, allocating different kernels to each SM within a Co-SM is essential. Therefore, the kernel scheduler of the GPU Weaver tries to launch different kernels on each SM per Co-SM as shown in Fig. 3.

3.1 Execution Unit Sharing

The purpose of execution unit sharing is to alleviate the SP and SFU shortage problem when many warp instructions are ready to execute. Fig. 4a depicts the micro-architectural details of a Co-SM including an SRC for execution sharing. In Fig. 4a, additional pipeline stages are shown: 1) from dispatch units of an SM to execution units of the other SM (Pre-sharing stage) and 2) from execution units of an SM to the writeback stage of the other SM (Post-sharing stage). Because additional interconnects across SMs highly increase wire delay, the additional stages are required to avoid critical path problems. The performance side effects of a deeper pipeline are negligible due to the fast context switching mechanisms of GPUs [14], as shown in Fig. 4a. Further, the complexity of data-forwarding does not increase, because GPUs utilize simple in-order execution with a scoreboarding system [2]. A previous system proved the hardware-level feasibility of execution unit sharing [3], so the area overhead is acceptable and will gradually become smaller as technology improves.

Issued warp-instructions first stay in Collector Units of the Operand Collector [10] until all operand values are loaded from the register file. Then, dispatch units send ready warp-instructions

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:57:35 UTC from IEEE Xplore. Restrictions apply.

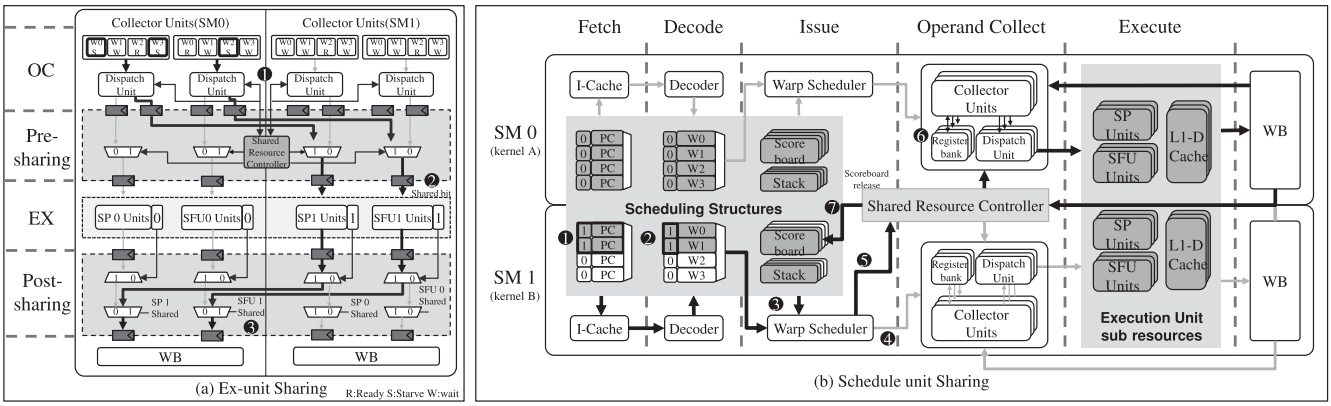


Fig. 4. (a) Execution sharing flow is highlighted and only the unidirection case is represented. (b) The cooperative streaming multiprocessor: 1) shared resources and additional features are highlighted and 2) execution flow of scheduling unit sharing is also provided.

to appropriate execution units for calculation. At this point, if the number of ready warp-instructions is more than the available execution units, the instructions will be executed later in conventional GPUs. To resolve this problem, first, the SRC checks the Collector Unit status from dispatch units and the number of issuable execution units from both SMs within a Co-SM (Fig. 4a ❶).

If an SM has many ready warp-instructions and the other SM has idle SP or SFU units, the SRC finds the number of migratable warp-instructions to avoid write-back stalls by checking the write-back bus of the SM. It then allows dispatch units of the SM to migrate the instructions to the other SM's SP or SFU units through the additional interconnects and the instructions are marked as "Shared" (Fig. 4a ❷). When the execution of the migrated warp-instructions is done, the results are returned to the write back stage of the source SM using saved "Shared" bit information (Fig. 4a ❸).

3.2 Scheduling Unit Sharing

In a GPU Weaver system, scheduling unit sharing resolves the TLP limitation by leveraging existing idle scheduling units from the other SM without context saving and additional storage overhead. Algorithm 1 shows how thread blocks are issued to use the scheduling unit sharing technique. When a SM has a kernel to execute, thread blocks are first allocated to the original SM when enough sub-resources are available in the SM (line 2-3). If the kernel has more remaining thread blocks and all the sub-resources other than scheduling units can be secured, the thread block scheduler decides to issue more thread blocks to the other SM whenever possible (line 4-5).

Algorithm 1. Threadblock Scheduling

Input: *KernelList (with initial context information)*
Output: *KernelList (with updated context information)*
1: *Kernel = SM.getKernel()*;
2: **if** (Kernel.hasTBtoLaunch()) *and* (!SM.fullOrigin())) **then**
3: *IssueBlock_OriginSM()*;
4: **else if** (Kernel.hasTBtoLaunch()) *and* (!NeighborSM.fullShared())) **then**
5: *IssueBlock_NeighborSM()*;
6: **end if**

A detailed execution flow of scheduling unit-shared warp instructions of kernel A on the neighboring SM is shown in Fig. 4b. In this scenario, we assume that kernel A is scheduling-limited, and kernel B is not limited by the scheduling structures. ❶ Some TBs of kernel A (Shared TBs) are first allocated to SM 1 with Shared bits as the sharing indicator by the thread block scheduler. ❷ The warps from Shared TBs fetch their instructions from the I-cache of SM 1 and the instructions are assigned to their own I-buffer based on original-WarpIds that are given by SM 1. ❸ The warp scheduler in SM 1 issues warp instructions for both kernel A and kernel B using their own stack, barrier control logic, and score board structure with shared bit indication. ❹ Kernel B's warp instructions are issued to its own Collector Units to access the SM 1 register file. ❺ However, warp instructions of Shared TBs should be issued to SM 0 for their register file access. When warp instructions of Shared TBs access the SM 0 register file, their WarpIDs must be transformed similar to *VWI (Virtual Warp Id)* in [21] to avoid conflicts between WarpID of two SMs. In order to avoid same register location access by WarpID, we transform WarpID by adding *MaxWarpId - NeighborKernel's MaxWarpID* of SM (e.g., MaxWarpId is 48 in our configuration). Thereby, A share-warp which has assigned Id *24* in Neighbor-SM (bottom SM in Fig. 4b) with *16* max NeighborKernel's warps change their warp id to 56 during issue to original-SM. ❻ Using transformed WarpIDs, Shared TB instructions can access their own registers in SM 0. After accessed own operands from original SM's register file and executed, the result of shared warp-instruction should be write-back to original register file, ❼ however, release of scoreboards should be done in neighbor SM where initially assigned for next instruction in progress.

3.3 Runtime Overhead

A GPU Weaver has two kinds of runtime performance overheads. First, a newly launched kernel to a target SM needs to wait for some period of time when it cannot secure enough scheduling structures because some scheduling structures are used for other kernel execution on the other SM using Scheduling-Sharing. Second overhead comes from interference of each techniques. The scheduling sharing enables more threads to run on execution units within a SM, thereby decreasing the chance of neighbor kernel's execution sharing. The cache sharing also reduces the idle time of execution units by enhancing the performance of the kernel execution with higher cache hit ratio. However, the overheads are small as shown in Section 5.

TABLE 1
Specification List of Benchmarks

Benchmark	Kernel Type	Limiting Sub-Unit	Cache Type
Hotspot(HS) [4]	Com	SP	Insensitive
Backprop(BP) [4]	Com	SP/SFU	Sen/Insensitive
Dxtc (DX) [15]	Sched	Sched	Insensitive
StringMatch (SM) [5]	Mem	LD/ST	Sensitive
Reduction (RD) [15]	Sched	Sched	Insensitive
Mri-q(MQ) [19]	Com	SFU	Insensitive
ThreadFence Reduction [15]	Sched	Sched	Insensitive
3DConvolution (3D) [18]	Mem	LD/ST	Sensitive
Fdtd-2d (FD) [18]	Mem	LD/ST	Sensitive

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:57:35 UTC from IEEE Xplore. Restrictions apply.

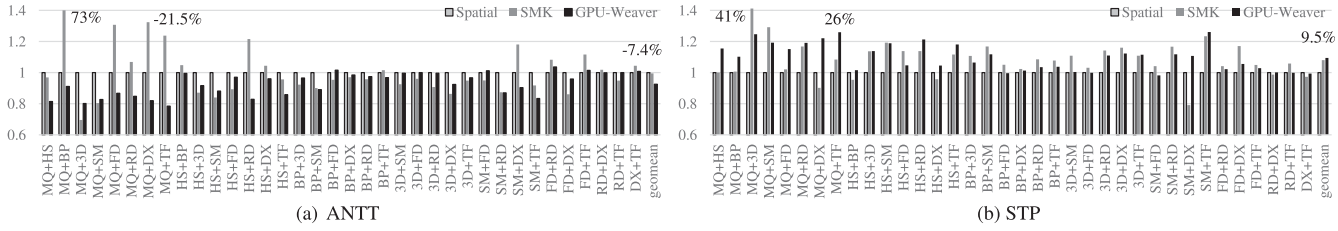


Fig. 5. (a) Total ANTT reduction and (b) Total STP improvement.

4 Experimental Results

We used a GTX480-like modified GPGPU-Sim v3.2.2 [2] to support concurrent execution of multi-programmed kernels [16]. Smart Even in spatial multitasking [1] with Drain preemption [16] is used as the baseline structure, and SMK [17] is also used for our evaluation. In addition to the baseline, we modified SM structures to implement Co-SMs consisting of two SMs and an SRC. For scheduling unit sharing, sizes of an I-cache and a decoder are doubled to minimize the interference between the kernels, and the additional hardware cost is small. We also added two additional pipeline stages to avoid increased critical path delay problems due to expensive wirings for execution sharing. We employ a widely used re-run based simulation [16] for fair performance comparison. We considered various kernels among Rodinia [4], Parboil [19], Polybench [18], Mars [5], and Cuda SDK [15]. Table 1 shows the benchmark list along with several important parameters. We evaluated all 36 pairs using nine workloads classified by various kernel characteristics (compute-intensive, cache-sensitive, and scheduling unit limited). We used average normalized turnaround time (ANTT) and system throughput (STP) as performance metrics [16]. In general, ANTT shows the user-perceived response time and STP represents the overall progress of the system.

Figs. 5a and 5b show ANTT and STP results of the GPU Weaver, respectively, and the performance results are compared to baseline SM level spatial multitasking and SMK. Fig. 5a shows that GPU Weaver successfully reduces ANTT of most workload combinations by up to 21.5 percent. Fig. 5b also shows that up to 26 percent STP performance improvement is seen in the GPU Weaver over the simple spatial multitasking. On average, GPU Weaver shows better performance gain of 7.4 percent ANTT reduction and 9.5 percent STP improvement than SMK (−0.02 percent ANTT and 7.9 percent STP improvement). Each technique individually contributes 3.8 percent (Ex-sharing), 2 percent (Cache-sharing) and 2.8 percent (Sched-sharing) for ANTT reduction and 3.9 percent (Ex-sharing), 4 percent (Cache-sharing) and 2 percent (Sched-sharing) for STP improvement. The results show that GPU weaver can share sub-resources efficiently similar to SMK and also minimize sub-resource interference problem successfully. For example, SMK shows low ANTT reduction for workload pairs having MQ because MQ requires high utilization of SFUs and the other kernels cannot fully utilize the SFU units. However, GPU weaver guarantees the fair performance of other kernels (BP/FD/RD/DX/TF) because the execution sharing for MQ execution is allowed only when they do not use their own SFUs.

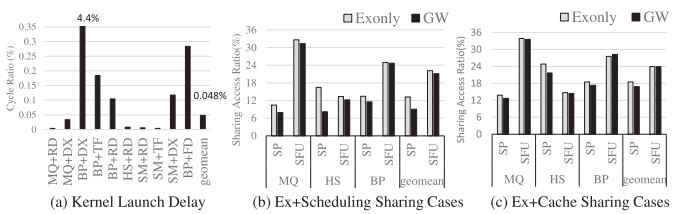


Fig. 6. 1) Delay cycle ratio of (a) kernel launch, 2) execution unit sharing access ratio of compute-intensive (MQ,HS,BP) in (b) compute + scheduling workload cases (c) compute + cache workload cases.

5 Runtime and Hardware Overhead Estimation

As mentioned in Section 3.3, GPU Weaver has two runtime overheads: *kernel launch delay* and *interference between sharing techniques*. In this evaluation, interferences between execution and the other sharing techniques are only considered because execution unit sharing performance is expected to mainly depend on the performance of the other kernel. Fig. 6a shows the ratio of kernel launch delay due to scheduling sharing to the total execution cycles. In the case of (BP+DX), BP kernel needs to wait for 4.4 percent of total execution cycles because of the long TB execution time of DX with scheduling sharing. However, the total overhead of kernel launch delay looks small because the overheads are less than 0.3 percent for all the other cases with 0.048 percent on average. Figs. 6b and 6c show the ratio of shared execution unit accesses to the entire execution unit accesses when 1) only execution unit sharing is applied (Exonly) and 2) all the sharing techniques are applied (GW) to workload pairs having compute-intensive workloads. For all cases in Fig. 6b, GPU-Weaver shows slightly less execution sharing accesses than Exonly case. Similarly, Fig. 6c also shows that the performance impact of execution sharing is minimal from cache sharing as the decrease of shared execution unit accesses is negligible.

Additional hardware cost per SM is estimated as follows: *a) Execution unit sharing:* 6 2-input muxes and pipeline registers for pre-sharing stage and 3 2-input muxes and pipeline registers for post-sharing stage (buffers are added to meet timing) *b) Cache sharing:* 64 2-input muxes (all cache lines for 2 sharing ways) and a 1-bit register (sensitivity), *c) Scheduling unit sharing:* a 77-bit I-fetch buffer, an additional instruction decoder, 96 1-bit registers (shared bits), and 3 6-bit registers (WarpId transformation). In addition to the above-mentioned costs, additional control logic costs are also required. In order to estimate the total hardware overhead of GPU Weaver, we synthesized the hardware logic using 40 nm low power profile standard cell library at 1 GHz target frequency. Based on the synthesis and P&R result, the additional hardware logic occupies 5.27 mm^2, resulting in only 0.997 percent of the total area of the baseline GPU (529 mm^2) [7].

6 Conclusion

In this paper, we proposed a *GPU Weaver*, which is a dynamic sub-resource management system on spatial multitasking GPUs. It enhances total performance by orchestrating three sharing techniques of execution units, scheduling units and L1 caches on top of baseline GPU spatial multitasking. Execution unit sharing is performed by executing ready warp instructions using execution units of a neighbor SM when possible. It also maximize thread level parallelism by allocating more thread blocks using scheduling structures of a neighbor SM whenever possible. Evaluations show that GPU Weaver can improve ANTT and STP by up to 21.5 and 26 percent, respectively.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:57:35 UTC from IEEE Xplore. Restrictions apply.

ACKNOWLEDGMENTS

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NO.NRF-2015R1C1A1A01053844, NO.NRF-2016R1C1B2016072), ICT R&D program of MSIP/IITP (No.2017-0-00142), and the R&D program of MOTIE/KEIT (No.10077609).

REFERENCES

[1] J. T. Adriaens, et al., "The case for GPGPU spatial multitasking," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, Feb. 2012, pp. 1–12.
[2] A. Bakhoda, et al., "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2009, pp. 163–174.
[3] M. Butler, L. Barnes, D. D. Sarma, and B. Gelinas, "Bulldozer: An approach to multithreaded compute performance," *IEEE Micro*, vol. 31, no. 2, pp. 6–15, Mar. 2011.
[4] S. Che, et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Symp. Workload Characterization*, 2009, pp. 44–54.
[5] B. He, et al., "Mars: A MapReduce framework on graphics processors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 260–269.
[6] KHRONOS Group, "OpenCL-the open standard for parallel programming of heterogeneous systems," 2010. [Online]. Available: http://www.khronos.org
[7] J. Kim, et al., "Efficient GPU multitasking with latency minimization and cache boosting," *IEICE Electron. Express*, vol. 14, 2017, Art. no. 20161158.
[8] R. Kumar, N. P. Jouppi, and D. M. Tullsen, "Conjoined-core chip multiprocessing," in *Proc. 37th Int. Symp. Microarchitecture*, Dec. 2004, pp. 195–206.
[9] E. Lindholm, et al., "NVIDIA tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.
[10] S. Liu, et al., "Operand collector architecture," US Patent 7,834,881, Nov. 16, 2010.
[11] M. Mantor and B. Sander, "Amd's radeon next generation GPU," in *Proc. Hotchips*, Aug. 2017. [Online]. Available: https://www.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.21-Monday-Pub/HC29.21.10-GPU-Gaming-Pub/HC29.21.120-Radeon-Vega10-Mantor-AMD-f1.pdf
[12] NVIDIA's Kepler GK110, 2012. [Online]. Available: http://www.nvidia.com/content/PDF/NVIDIA-kepler-GK110-architecture-whitepaper.pdf
[13] NVIDIA, "Sharing a GPU between MPI processes: Multi-process service (MPS) overview, 2014. [Online] Available: http://docs.nvidia.com/deploy/mps/index.html
[14] NVIDIA's the world's most advanced data center GPU Volta V100, 2017. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
[15] NVIDIA, "GPU computing sdk," 2013. [Online]. Available: https://developer.nvidia.com/gpu-computing-sdk
[16] J. J. K. Park, et al., "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2015, pp. 593–606.
[17] J. J. K. Park, et al., "Dynamic resource management for efficient utilization of multitasking GPUs," in *Proc. 22nd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 527–540.
[18] L.-N. Pouchet, "PolyBench: The polyhedral benchmark suite," 2012. [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench
[19] J. A. Stratton, et al., "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center Reliable High-Perform. Comput.*, 2012. [Online] Available: http://impact.crhc.illinois.edu/parboil/parboil.aspx
[20] Z. Wang, et al., "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 358–369.
[21] M. K. Yoon, et al., "Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit," in *Proc. 43rd Annu. Int. Symp. Comput. Archit.*, Jun. 2016, pp. 609–621.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:57:35 UTC from IEEE Xplore. Restrictions apply.