Efficient GPU multitasking with latency minimization and cache boosting

Jiho Kim, Minsung Chu, and Yongjun Park[a)]

School of Electronic and Electrical Engineering, Hongik University, Seoul, Korea

a) *yongjun.park@hongik.ac.kr*

Abstract: GPU spatial multitasking has been proven to be quite effective at executing different applications concurrently using SM partitioning. However, while it maximizes total throughput, latency-critical applications often cannot meet their deadlines due to the increased execution time. Furthermore, SM partitioning cannot allocate the appropriate L1 cache size per kernel. To solve these problems, this paper proposes a new application-aware resource allocation framework called *GPU Fine-Tuner*, for assigning appropriate resources to GPU kernels. To minimize the execution time of latency-constrained applications, it assigns them more SMs when performance is not affected. It also increases the cache size of SMs for cache-sensitive kernels using resource borrowing from neighbors for cache-insensitive kernels. Experimental results show that the Fine-Tuner outperforms GPU spatial multitasking with up to 15% less average latency without performance degradation.

Keywords: GPGPU, multitasking, energy, resource sharing, workload balancing

Classification: Integrated circuits

References

[1] J. T. Adriaens, *et al.*: "The case for GPGPU spatial multitasking," HPCA (2012) 1 (DOI: 10.1109/HPCA.2012.6168946).

[2] P. Aguilera, *et al.*: "QoS-aware dynamic resource allocation for spatial multitasking GPUs," ASP-DAC (2014) 726 (DOI: 10.1109/ASPDAC.2014.6742976).

[3] D. H. Albonesi: "Selective cache ways: On demand cache resource allocation," MICRO (1999) 248.

[4] A. Bakhoda, *et al.*: "Analyzing CUDA workloads using a detailed GPU simulator," ISPASS (2009) 163 (DOI: 10.1109/ISPASS.2009.4919648).

[5] S. Che, *et al.*: "Rodinia: A benchmark suite for heterogeneous computing," IISWC (2009) 163 (DOI: 10.1109/IISWC.2009.5306797).

[6] P. N. Glaskowsky: "NVIDIA's Fermi: the first complete GPU computing architecture," White paper (2009).

[7] C. Gregg, *et al.*: "Fine-grained resource sharing for concurrent GPGPU kernels," USENIX (2012) 10.

[8] B. He, *et al.*: "Mars: A MapReduce framework on graphics processors," PACT (2008) 260 (DOI: 10.1145/1454115.1454152).

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

[9] Y. Liang, *et al.*: "Efficient GPU spatial-temporal multitasking," IEEE Trans. Parallel Distrib. Syst. **26** (2015) 748 (DOI: 10.1109/TPDS.2014.2313342).

[10] J. Nickolls, *et al.*: "NVIDIA CUDA software and GPU parallel computing architecture," Microprocessor Forum (2007).

[11] S. Pai, *et al.*: "Improving GPGPU concurrency with elastic kernels," ASPLOS (2013) 407 (DOI: 10.1145/2451116.2451160).

[12] J. J. K. Park, *et al.*: "Chimera: Collaborative preemption for multitasking on a shared GPU," ASPLOS (2015) 593 (DOI: 10.1145/2694344.2694346).

[13] Polybench: "The Polyhedral benchmark suit" (2011) http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU.

[14] I. Tanasic, *et al.*: "Enabling preemptive multiprogramming on GPUs," ISCA (2014) 193 (DOI: 10.1109/ISCA.2014.6853208).

[15] Z. Wang, *et al.*: "Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," HPCA (2016) 358 (DOI: 10.1109/HPCA.2016.7446078).

[16] Q. Xu, *et al.*: "Warped-slicer: Efficient Intra-SM slicing through dynamic resource partitioning for GPU multiprogramming," ISCA (2016) 230 (DOI: 10.1109/ISCA.2016.29).

[17] G. Pekhimenko, *et al.*: "A case for toggle-aware compression for GPU systems," HPCA (2016) 188 (DOI: 10.1109/HPCA.2016.7446064).

[18] N. Agarwal, *et al.*: "Selective GPU caches to eliminate CPU-GPU HW cache coherence," HPCA (2016) 494 (DOI: 10.1109/HPCA.2016.7446089).

[19] J. Kloosterman, *et al.*: "WarpPool: Sharing requests with inter-warp coalescing for throughput processors," MICRO (2015) 433 (DOI: 10.1145/2830772.2830830).

1 Introduction

Computing platforms for mobile and high-performance computing devices must support high-performance capabilities while retaining low energy consumption. Graphics processing units (GPUs) are attractive solutions for this because they provide high throughput by accelerating massively data-parallel applications efficiently. Therefore, GPUs have become the essential parts of most computing systems in the form of heterogeneous architectures. As most heterogeneous systems generally consist of multiple CPUs and a shared GPU, the shared GPU must provide a smart multitasking mechanism in order to handle multiple data-parallel kernels simultaneously from the CPUs.

In order to efficiently utilize GPU resources between multiple applications, spatial multitasking [1, 2] is one of the most popular solutions, which partitions resources for each application at streaming multiprocessor (SM) granularity. As many applications do not require full GPU resources, spatial multitasking can improve total system throughput with concurrent execution of multiple applications compared to temporal multitasking [12, 14].

However, spatial multitasking may be ineffective for latency-sensitive applications as the execution latency of each application highly increases compared to normal (single-application) execution, since it cannot fully occupy all the available resources [1]. With increased execution latency, latency-sensitive applications often violate deadlines, even when they are launched immediately. The second problem

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

of spatial multitasking is the mismatch between the L1 cache resource requirement of each kernel and the fixed L1 cache size of SMs. For example, compute-intensive kernels generally do not fully utilize the L1 cache but memory-intensive kernel performance highly depends on L1 cache size.

To resolve these inefficiencies, this paper proposes an application-aware resource allocation framework referred to as the *GPU Fine-Tuner*. Based on spatial multitasking, it first assigns more SMs to higher priority kernels only when the total system throughput is not changed with different SM partitioning. With the smart SM resource adjustment, the execution latency of higher priority kernels, which are latency-sensitive kernels in most cases, can be minimized so as not to violate their strict deadlines. GPU Fine-Tuner also tries to allocate more L1 cache resources to cache-sensitive kernels by borrowing from neighboring SMs running cache-insensitive kernels (cache boost). The cache resource allocation process is performed when a new kernel is launched with minimum reallocation overhead because the original data backup process in the borrowing partition is not required due to the GPU programming model [10]. Based on these novel techniques, the GPU Fine-Tuner achieves up to 15% lower average latency without the total throughput loss.

2 Background and motivation

In this paper, we first identified compute-/memory-intensive kernels from well-known GPGPU benchmark suites using a well-known GPU simulator (GPGPU-Sim), and the methodology for this task is detailed in Section 4 with Fig. 8 and Table I.

GPU spatial multitasking [1] is effective in improving total system throughput by resource partitioning for mltiple applications. However, as each application cannot utilize all the resources, the execution latency of each kernel greatly increases compared to its single execution. Fig. 1 shows the impact of execution latency of different applications when each kernel is executed with spatial multitasking compared to single execution. In this figure, all the compute-intensive kernels have more than 60% increased latency, and most of the memory-intensive kernels also show high performance degradation. This is because compute-intensive applications normally require full SMs, and the SM requirement of memory-intensive applications may be smaller than the full GPU but still suffer. This result means that spatial multitasking may cause latency violation when some applications have critical deadlines, even though the total system throughput is dramatically improved and the starvation problem is solved.

Fixed L1 cache size of each SM may also be a limiting factor in maximizing the total utilization of computation resources in SM-level spatial multitasking due to the different cache-sensitivities of applications. Fig. 2 shows the cache sensitivity variance over multiple data parallel kernels. In this figure, we measured the performance of kernels with different sizes of L1 cache from 8 kB to 32 kB by changing the number of ways and normalized them according to the performance of the 16 kB size. Fig. 2(a) shows the kernels that do not require full cache size (Cache-Insensitive) for maximum performance. Most of these kernels are compute-intensive or task-bounded, which means the total number of threads is too small.

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

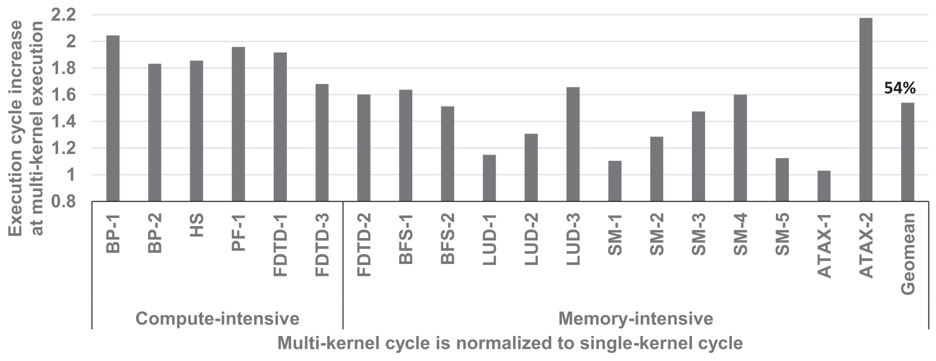


Fig. 1. Single kernel latency increase of multi-kernel execution compared to single-kernel execution on GPUs (simulation-based)

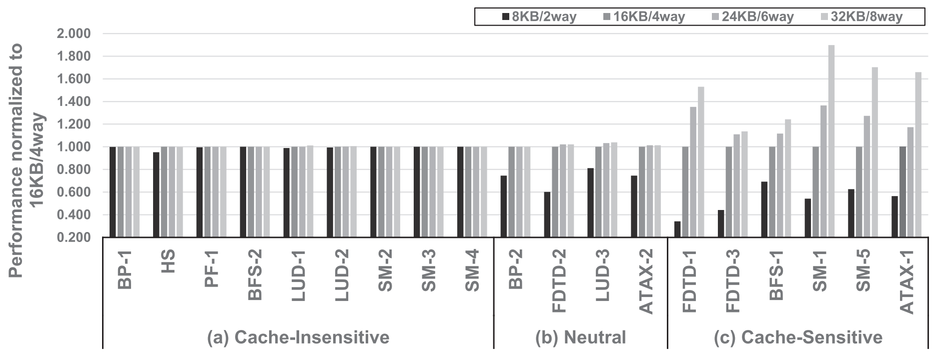


Fig. 2. Different kernel types based on cache sensitivity (simulation-based)

Fig. 2(b) depicts the kernels with saturated performance with respect to cache size. Performance of the kernels is highly improved with the baseline cache size and then shows only little improvement over this size. Fig. 2(c) shows the kernels with high performance scalability with respect to the L1 cache size (Cache-Sensitive). Based on the cache sensitivity variance, we found an opportunity to further improve the performance of cache-sensitive kernels by using the unused cache partition for cache-insensitive kernels.

3 GPU Fine-Tuner

The GPU Fine-Tuner presented here is an advanced kernel scheduler for adjusting two parameters: the number of SMs and the L1 cache size per kernel. The Fine-Tuner decides these parameters based on the priority comparison between two co-running kernels. They are updated whenever new kernel is launched, and the Scheduling-Tuner and the Cache Tuner then determine SM and cache allocation policy.

Fig. 3(a) illustrates the overall architecture of the GPU Fine-Tuner. The Fine-Tuner consists of three parts: Master/slave priority selector, Scheduling Tuner, and Cache Tuner. The Master/slave priority selector identifies the types of kernels and determines the priorities of kernels at the application and kernel level. In this priority selector, high priority is called Master, and low priority is called Slave. The scheduling tuner chooses the SM partitioning policy based on the kernel type combination. If two co-running kernels are both compute-intensive, priority-based

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

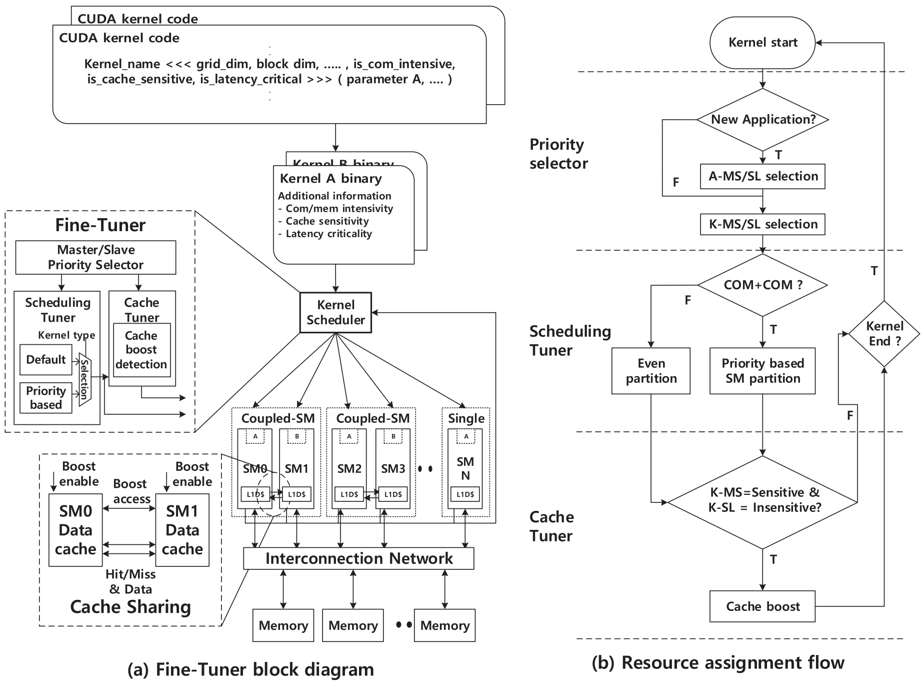


Fig. 3. GPU Fine-Tuner

SM partitioning policy is selected for latency minimization, or even partitioning is used. In priority-based partitioning, more SMs are allocated to the application-level high priority kernel. The cache tuner performs the L1 cache size allocation for co-running kernels based on the cache sensitivity combination. As cache insensitive kernels do not need to use all the L1 cache of the SMs, some partitions can be used for other kernels. Based on this insight, if a current kernel (kernel-master) is cache sensitive and a newly launched kernel (kernel-slave) is cache insensitive, the current kernel decides to use some partition of the L1 cache of SMs for the new kernel. (cache boost) As the cache borrowing is performed between neighbor SMs, every two neighbor SMs comprise the coupled-SM, which share L1 caches with each other as shown in Fig. 3(a).

Fig. 3(b) describes the resource assignment flow of the GPU Fine-Tuner. First, if a kernel is newly launched, both the kernel and the currently running kernel have to update their priorities on the priority selector. (Section 3.1) The scheduling tuner then tries to choose the priority-based SM allocation policy that gives more SMs to the application-level master kernel when both kernels are compute intensive. (Section 3.2) Lastly, the cache tuner tries to assign more L1 cache resources to a cache-sensitive kernel by resource borrowing from the neighboring SM. (Section 3.3)

3.1 Master/slave priority selector

Two tuners basically control their resource allocation based on their kernel priorities and their characteristics. As discussed above, we call high priority the master and the other slave. The master/slave priority selector contains the kernel type and the priority information and provides them to the tuners. For kernel type information, it saves the compute-intensity and cache-sensitivity of each kernel.

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

The information can be determined by the programmer's knowledge about the target application or using profiling tools from GPU vendors such as *nvprof* from NVIDIA. For example, compute/memory intensity could be determined by the ratio of Load/Store instructions to total instructions and cache sensitivity could also be determined by the hit/miss ratio of caches from profiling. In our framework, the information is manually added through simple code modification of the original CUDA program, as shown in Fig. 3(a). The modified-compiler then embeds the additional information to the execution binary, and GPUs can save the data to the additional special registers when each kernel is newly launched. In addition to this, the programmer can set kernels of an application as latency critical in a similar way when the programmer decides the application is latency constrained, such as in health monitoring system applications that need to check the user's health status promptly with a hard deadline.

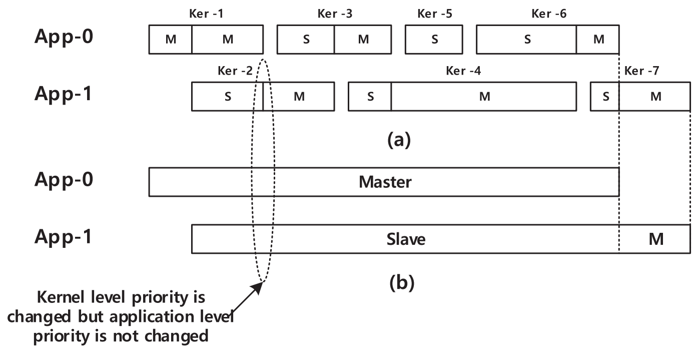


Fig. 4. Master/slave priority selection example

Fig. 4(a) depicts the kernel level master/slave (K-MS/SL) selection process. When several kernels are co-executing on a GPU, the earliest invoked kernel is designated the Master and the remainders are Slaves. Thus, the Master kernel can be changed when the previous Master kernel is finished. Similar to K-MS/SL selection, the application level master/slave (A-MS/SL) relationship is also determined (Fig. 4(b)). Different from K-MS/SL, the A-MS/SL relationship is the priority difference between whole applications having multiple kernels. Therefore, the relationship depends on which application is more latency constrained based on user-annotated information. For example, if both applications are not latency constrained, the earlier started application will be the A-MS. This relationship is only changed when previous Master application's last kernel is finished. The Fine-Tuner updates the A-MS/SL and K-MS/SL information for the Scheduling and Cache tuner when either one of the executing kernels is finished or a new kernel is launched as shown in Fig. 3(b) and Fig. 4.

3.2 Scheduling tuner

The Scheduling tuner decides to give more SMs (up to 75%) to A-MS kernels only when the total system throughput is not affected, since both kernels are compute intensive. As compute intensive kernel performance is nearly proportional to the number of SMs, the total system throughput will be nearly constant per different SM allocated combination and the execution time of the A-MS application can be minimized by obtaining more SMs. In general, when co-running two applications where compute intensive kernels are dominant, the A-MS application execution time will be highly reduced.

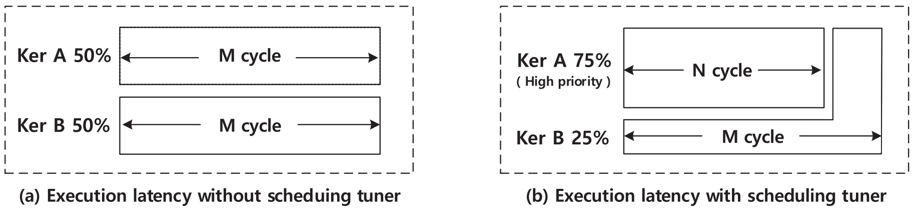

Ker A 50%
M cycle
Ker B 50%
M cycle
(a) Execution latency without scheduling tuner
Ker A 75%
(High priority)
N cycle
Ker B 25%
M cycle
(b) Execution latency with scheduling tuner

Fig. 5. System throughput and execution latency analysis

Fig. 5 describes the total system throughput and kernel execution latency comparison for two different cases when running two compute intensive kernels with and without a scheduling tuner on SM multitasking. Fig. 5(a) shows that the total execution time and the average execution latency of two kernels are the same as M cycles on evenly partitioned execution. However, when Kernel A is latency constrained, the scheduling tuner allocates more SM resources to Kernel A, as shown in Fig. 5(b). In this case, the average execution latency is reduced due to the smaller latency of Kernel A execution (N cycles) without total execution time loss. This analysis shows that the scheduling tuner can vary SM allocation freely without performance degradation, and therefore, the latency of critical applications can be minimized by an allocation ratio adjustment when co-running kernels are compute intensive.

3.3 Cache Tuner

The Cache Tuner adjusts the L1 cache resource assignment of co-running kernels when possible (cache boost). The Cache Tuner first checks the possibility of cache borrowing. Cache borrowing is only allowed when the K-MS (earlier) kernel is cache sensitive and the K-SL (newly launched) kernel is cache insensitive. Borrowing for the opposite combination is not allowed because performance

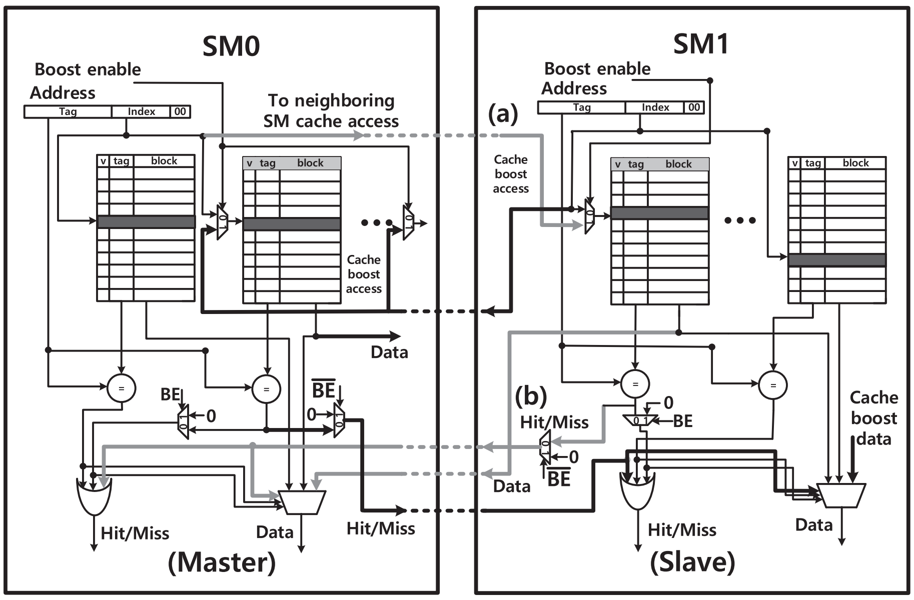


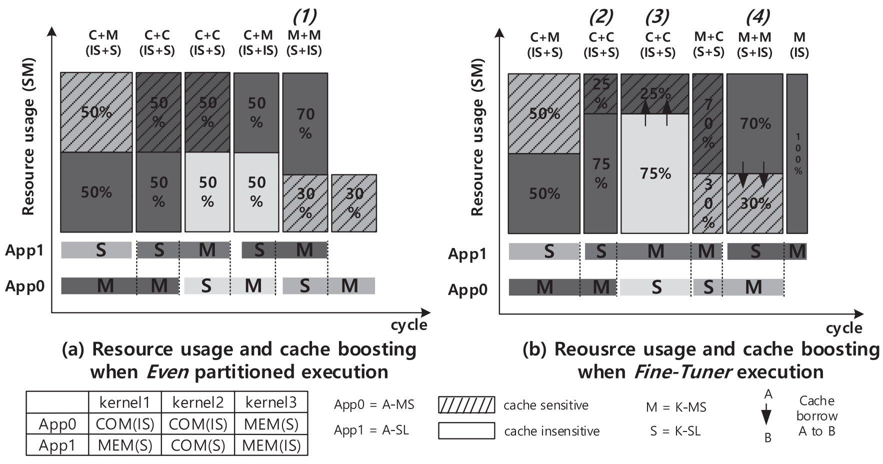
Fig. 6. Cache boost architecture

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

degradation may occur when cached data for the master kernel is lost by cache repartitioning.

Implementation of cache boost is similar to well-known cache way partitioning [3]. At the high-level, a subset of the ways in the L1 cache of SMs for the K-SL kernel is logically disabled for its own execution, and the subset is used for K-MS kernel execution as the logically extended ways of the L1 cache of SMs for the K-MS kernel. To realize this, only small cache access selection logics and data read selection logics are added between every two neighboring cores (coupled-SM) as shown in Fig. 6. Fig. 6(a) shows that the cache access of SM0 can be extended to a subset of the ways of the SM1 L1 cache by boost enable signal, and Fig. 6(b) shows that the cached data from the neighboring SM (SM1) can be used for the K-MS kernel. Different from general dynamic cache repartitioning techniques, a data consistency problem does not exist because the GPU L1 cache's write policy is basically write-evict for Hit and no-allocation for Miss [10].

3.4 Example execution scenario using the GPU Fine-Tuner



	kernel1	kernel2	kernel3
App0	COM(IS)	COM(IS)	MEM(S)
App1	MEM(S)	COM(S)	MEM(IS)

Fig. 7. Example execution using the GPU Fine-Tuner

An example execution scenario is shown in Fig. 7. In this example, we assume that two applications (App0, App1) having three kernels each are executed in parallel, and App0 is A-MS as it is latency-constrained. Based on this assumption, Fig. 7(a) shows the normal execution using even partitioning, where each half of the SMs is allocated to two applications without considering the hard deadline of App0.

Compared to this, Fig. 7(b) shows the execution with the GPU Fine-Tuner. In this scenario, the execution latency of App0 can be minimized by assigning App0 more SMs when both co-running kernels are compute intensive ((2): scheduling tuner) and assigning App0 more L1 cache ways when the App0 kernel is cache sensitive and the App1 kernel is cache insensitive ((4): cache tuner). Also, the increased latency of App1 can be reduced using cache borrowing as it is not based on application level priority. ((3): cache tuner)

4 Experiments

In this work, we used GPGPU-Sim v3.2.2 [4, 17, 18] to evaluate the GPU Fine-Tuner. The Fermi architecture model [6, 18, 19] was used for our evaluation.

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

Table I(a) shows the configuration parameters of our model. We use *Extended Even Partition* as a baseline scheduling policy for a fair comparison. *Extended* means non-used SMs are allocated to another kernel if one of the co-running kernels is task-bounded, which means that the number of thread blocks is too small to fully occupy the SMs.

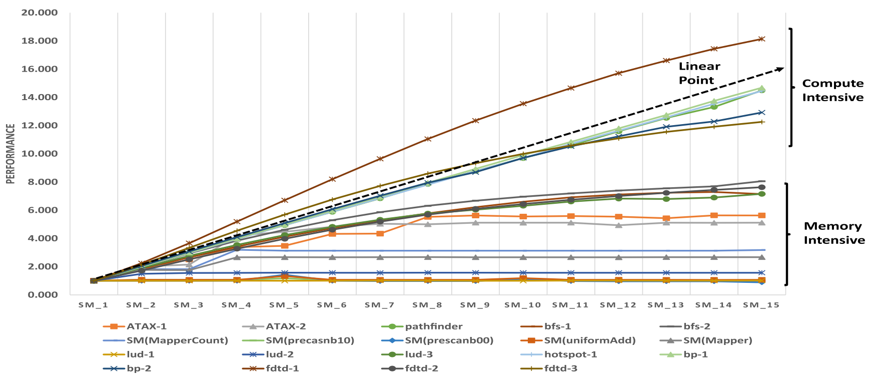


Fig. 8. Kernel type classification

In this work, we classify various kernels chosen from well-known GPGPU benchmarks such as Rodinia [5], polybench [13] and Mars [8] into compute (COM) and memory (MEM) intensive types based on profile information. For this, we measured the performance of each kernel with different numbers of SMs from 1 to 15, and normalized them with the performance of one SM as shown in Fig. 8. Kernels with highly scalable performance are compute intensive, and others are memory intensive. All the benchmarks for evaluation are shown in Table I(b) with each kernel type and cache sensitivity. We also categorize the application type based on the type of the dominant kernel.

Table I. GPGPU-Sim configuration and benchmark list

SM	15
Warp Size	32
Resource per SM	Max 1536 thread Max 8 cta 32768 Registers 48K shared Memory
Warp scheduler	GTO
L1 data cache size	16KB / 4way

(a) GPGPU-Sim configuration

Benchmarks	source	kernel	Block dim	Cache type	Kernel type	Application type
Breadth First Search (BFS)	Rodinia	KernelP4Node	(1954.1.1)	Sensitive	Memory	Memory
		Kernel2	(1954.1.1)	Insensitive	Memory	
Backprop (BP)	Rodinia	Bpnn_layerforward	(1.4096.1)	Insensitive	Compute	Compute
		Bpnn_adjust_weights	(1.4096.1)	Sensitive	Compute	
ATAX	Polybench	atax_kernel1	(16.1.1)	Sensitive	Memory	Memory
		atax_kernel2	(16.1.1)	Sensitive	Memory	
FDTD	Polybench	fdtd_step1_kernel	(32.128.1)	Sensitive	Compute	Compute
		fdtd_step2_kernel	(32.128.1)	Sensitive	Memory	
		fdtd_step3_kernel	(32.128.1)	Sensitive	Compute	
LUD	Rodinia	lud_diagonal	(1.1.1)	Insensitive	Memory	Memory
		lud_perimeter	(15.1.1)	Insensitive	Memory	
		lud_internal	(15.15.1)	Sensitive	Memory	
Pathfinder (PF)	Rodinia	dynproc_kernel	(463.1.1)	Insensitive	Compute	Compute
Hotspot (HS)	Rodinia	calculate_temp	(43.43.1)	Insensitive	Compute	Compute
StringMatch (SM)	Mars	MapperCount	(4.1.1)	Sensitive	Memory	Memory
		prescanlLb1ELb0	(2.1.1)	Insensitive	Memory	
		prescanlLb0ELb0	(1.1.1)	Insensitive	Memory	
		uniformAdd	(2.1.1)	Insensitive	Memory	
		Mapper	(4.1.1)	Sensitive	Memory	

(b) Benchmark list

4.1 Results

We simulated all pairs of the listed benchmarks. In order to avoid long-running benchmarks having a significant portion of running time on their own, different execution lengths of two benchmarks were modified to be similar within 3% of the total cycle by re-running each benchmark several times.

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

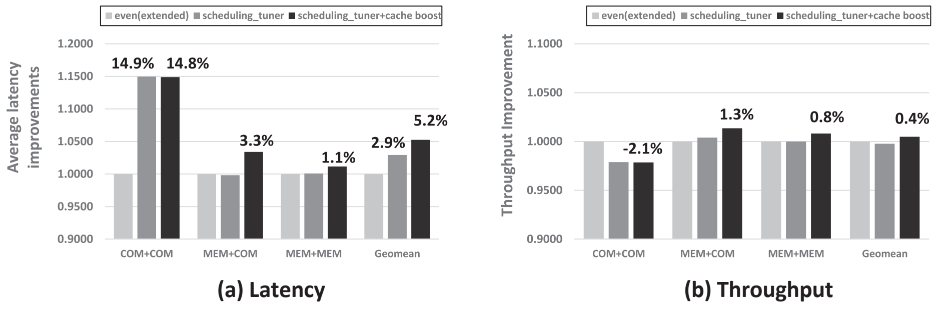


Fig. 9. Latency and system throughput

Fig. 9(a) shows the total average latency improvement of the GPU Fine-Tuner. On average, the Fine Tuner improves the average latency for COM+COM, MEM+COM, and MEM+MEM by 14.9%, 3.3%, 1.1%, respectively. The overall average latency improvement of the Fine Tuner is 5.2%. The scheduling tuner mainly contributes to the COM+COM case and the cache tuner mainly contributes to the MEM+COM case. In the MEM+MEM case, a slight performance improvement can be obtained with cache borrowing because some memory intensive kernels are cache insensitive. Fig. 9(b) also shows the system throughput of our experiments. The result shows a 0.4% average improvement of system throughput with a slight loss from priority-based SM scheduling and a slight gain from cache borrowing. Therefore, we can conclude that the GPU Fine-Tuner has little effect on system throughput when latency is optimized.

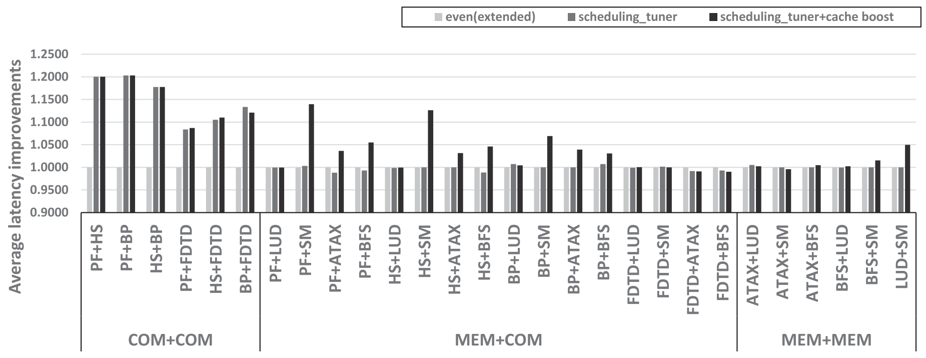


Fig. 10. Individual latency improvement of applications

Fig. 10 presents detailed improvements of average latency in all the application combinations. All the combinations in COM+COM improve up to 20% and more than half of the combinations in MEM+COM improve up to 13%. For MEM+MEM combinations, latency is improved for two combinations with SM because cache borrowing is applied because a dominant kernel in SM is cache sensitive.

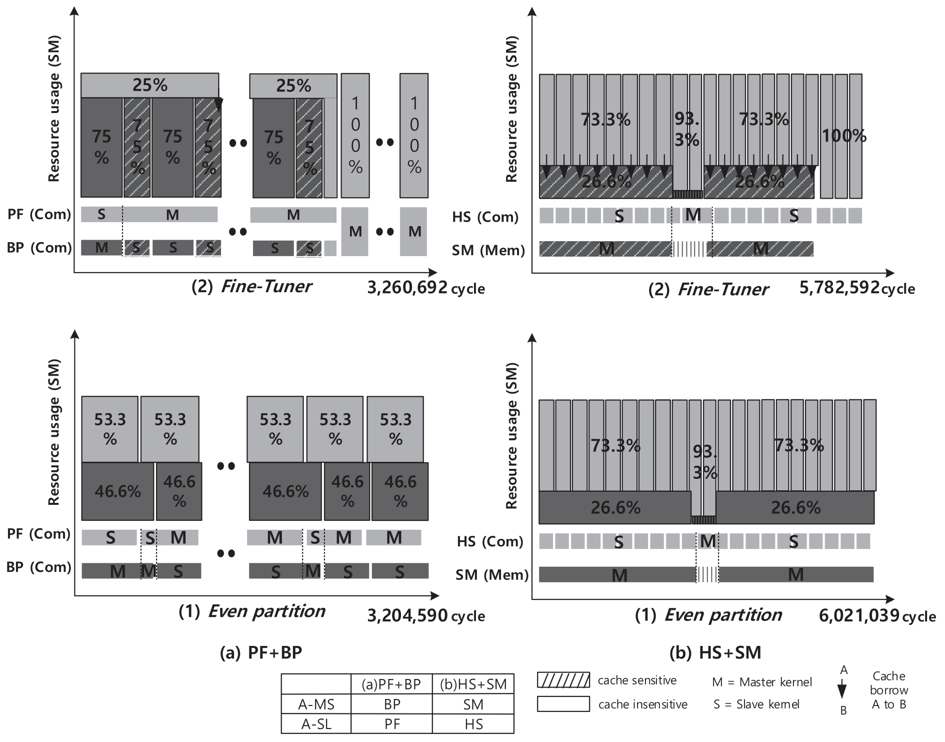
4.2 Hardware overhead

The hardware overhead of the GPU Fine-Tuner is twofold: special registers and cache borrowing support logic. First, several special registers are required to store the annotated data from the user (cache sensitivity, compute intensity, and latency criticality) and two types of master/slave information, but the overhead is negli-

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

gible. Second, a substantial amount of additional logic is required to support cache borrowing. To access another SM's cache, as shown in Fig. 6, a 2-input mux for tag match per cache line, two muxes to determine Hit/Miss information and output data direction per cache way, and some additional logic are required per SM. In order to estimate the hardware overhead of the logic, RTL Verilog implementation for the routing logic is used and synthesized with the Synopsys tools using 65 nm technology. We then scaled the result to 40 nm technology to match the GTX 480 baseline system [6, 19]. As a result, the GPU fine-tuner specific logic is only 0.014 mm^2, which is only 0.003% of the total GPU area (529 mm^2) [19].

4.3 Case study: PF+BP and HS+SM



	(a)PF+BP	(b)HS+SM
A-MS	BP	SM
A-SL	PF	HS

Fig. 11. Execution cases: PF+BP and HS+SM

This section shows the detailed difference between *Even partitioning* and *Fine-Tuner* execution from two executions. In both cases, total number of executed instructions are the same for *Even* and *Fine-Tuner* execution.

Fig. 11(a) shows PF+BP execution. For even partition, total SMs are divided by two and they are allocated to the applications (Fig. 11(a)(1)). However, the Fine-Tuner allocates BP more SMs through priority-based partitioning because BP is the A-MS application. Therefore BP execution is finished earlier than the evenly partitioned case while retaining similar throughput (Fig. 11(a)(2)).

Fig. 11(b) shows HS+SM execution, which has memory-intensive and compute-intensive kernels. For even partition, HS is executed using more SMs because SM is task-bounded (small total number of threads) (Fig. 11(b)(1)). Compared to this, the total cycle with the Fine-Tuner can be minimized (4% gain) by applying cache borrowing (Fig. 11(b)(2)).

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017

5 Related works

Several software [7, 9, 11] and hardware [1, 12, 2] approaches for GPU multi-tasking has been introduced before. These works generally focus on SM level partitioning, and did not consider resources inside SMs. Preemption mechanism [12] and dynamic scheduling [2] are orthogonal to our research, and therefore can be applied to our system. Recently, simultaneous multi-kernel, (SMK) which executes multiple kernels within a same SM, is proposed to improve the utilization of resources inside SMs [15, 16]. Though these hardware algorithms can optimize thread block allocation for high resource utilization inside SMs, the control logic is too complex. Our research is different from the technique in that we don't require complex control logic to use idle resources.

6 Conclusion

Graphics processing units (GPUs) present a powerful platform by providing high computation throughput for massively parallel applications. As modern GPUs are used as an shared resource on recent heterogeneous platforms, efficient execution of multiple applications on GPUs has become one of the central challenges. GPU spatial multitasking is one of the promising solutions for this but still suffer from increased latency and resource under-utilization inside SMs. In this paper, we propose the *GPU Fine-Tuner* to perform smart resource allocation process. First, it minimizes the execution time of latency-critical applications by allocating more SMs. Second, it improves the performance of cache sensitive kernels by getting more L1 cache ways from neighboring SMs when possible. By applying these novel techniques, the GPU Fine-Tuner achieves up to 15% lower average latency without the total throughput loss.

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 and No. NRF-2015K2A1A2070541) and the Korea Institute for Advancement of Technology (KIAT) grant funded by the Korean government (Motie: Ministry of Trade, Industry & Energy, HRD Program for Software-SoC convergence) (No. N0001883).

© IEICE 2017
DOI: 10.1587/elex.14.20161158
Received November 23, 2016
Accepted December 20, 2016
Publicized January 16, 2017
Copyedited April 10, 2017