Navigator: Dynamic Multi-kernel Scheduling to Improve GPU Performance

Jiho Kim
School of Electrical Engineering
KAIST
Daejeon, Republic of Korea
jihokim@kaist.ac.kr

John Kim
School of Electrical Engineering
KAIST
Daejeon, Republic of Korea
jjk12@kaist.ac.kr

Yongjun Park
Department of Computer Science
Hanyang University
Seoul, Republic of Korea
yongjunpark@hanyang.ac.kr

***Abstract*—Efficient GPU resource-sharing between multiple kernels has recently been a critical factor on overall performance. While previous works mainly focused on how to allocate resources to two kernels, there has been limited amount of work on determining *which* workloads to concurrently execute among multiple workloads. Therefore, we first demonstrate on a real GPU system how the selection of concurrent workloads can have significant impact on overall performance. We then propose GPU Navigator – a lookup-table-based dynamic multi-kernel scheduler that maximizes overall performance through online profiling. Our evaluation shows that GPU Navigator outperforms a greedy policy by 29.3% on average.**

***Index Terms*—GPGPU, Spatial Multitasking, Simultaneous Multitasking, multi-kernel**

I. Introduction

GPUs have recently become available commonly in cloud services, including Amazon Web services [8], Google Cloud Platforms [13], and Microsoft Azure [16]. These cloud services offer cost-effective server solutions by multi-tenancy, where multiple users share a physical server (e.g, Fig 1) by allocating multiple virtual machines (VMs) [21], [26]. While this policy increases total resource utilization of physical servers, total performance can be degraded drastically because of massive requests from multiple VMs. Therefore, a central challenge is to improve the GPU performance of concurrent multiple kernel execution using efficient GPU resource sharing [27].

To support GPU multitasking, Nvidia Hyper-Q [1] and Multi-Process Service [2] support multi-process scheduling on a shared GPU with a simple software level FIFO-style resource allocation policy. Several preemption techniques [19] have also been proposed by considering GPU workload characteristics. For further performance improvement, spatial multitasking [6], [7], [14], [18] and simultaneous multikernel (SMK) techniques are proposed [3], [28], [29] to divide the computation resources into multiple kernels concurrently. These prior works have mainly focused on determining how to improve performance or resource utilization, ***given*** two kernels.

However, as the cloud service market continues to grow and number of VM instances per host physical machine increase, another critical challenge is determining ***which*** pair of kernels to select from, given that there are multiple kernels that are available to execute. Total SMK performance is no longer dominated by each performance improvement of given co-running kernel pair. Instead, the overall SMK performance varies highly depending on which kernels are selected to run together among the massive number of waiting kernels, and selecting the best co-running kernel pairs becomes more critical to total SMK performance.

A main challenge is the difficulty in finding the best co-running kernel pairs from different combination of kernels. The SMK performance is generally better when kernels with different characteristics (e.g, compute-intensive vs. memory-intensive) are executed, but the criteria for selecting the best kernel pairs has various challenges. First, performance of co-running kernel pairs with different characteristics is not always better than pairs with similar characteristics. For example, it is well-known that mixing memory-intensive workloads (MEM) with compute-intensive workloads (COM) often results in the best overall performance [6]. However, we demonstrate in (Section III) that simply selecting MEM+COM workloads does not necessarily result in the best overall performance improvement on a real GPU system. In addition, another observation is that high performance variance exists even when a kernel is co-executed with similar characteristic kernels (e.g., MEM+MEM) Thus, an improved approach to find the best co-running kernel pairs beyond simple static categorization is needed to maximize GPU multitasking performance.

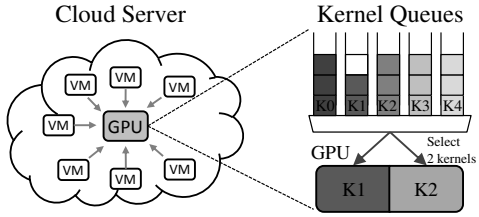

Fig. 1: A high-level block diagram of GPU usage in a cloud server with multiple users (or VMs) sharing a single GPU. Each GPU has multiple kernels ready to execute and the GPU is responsible for selecting which kernels to execute.

To address this SMK performance prediction difficulty problem, we propose a dynamic high-level kernel scheduler, referred to as *GPU Navigator*. The GPU Navigator consists of a profile-guided dynamic kernel scheduler to select an appropriate kernel to execute when SMK requires a new kernel. To obtain high profiling accuracy with minimum online profiling overhead, GPU Navigator utilizes a lookup-based kernel scheduler that caches profile results for both single and concurrent executions.

II. Background

A. Multitasking Support on GPUs

Most data-parallel applications written in CUDA/OpenCL generally consist of a host code for controlling GPU execution on a CPU and kernel codes on GPUs. A GPU kernel consists of multiple *threads* to maximize thread-level parallelism (TLP). Kernels are composed of multiple *Thread Blocks (TBs)* and each TB is further divided into multiple *warps* with small number of threads. Fig 2 illustrates a basic structure of a GPU and a management system to handle multiple kernel execution. To control multiple kernels on a GPU from each Stream Queue [1] [20], a kernel management unit such as GMU (Grid Management Unit) [1] selects several active kernels among multiple launched kernels and controls multikernel execution by changing the state of each waiting kernel into active, pending or suspend state. After the active kernel is selected by the kernel management unit (shown in Fig 2(a)), the thread block scheduler [30] assigns the TBs to SMs.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:58:21 UTC from IEEE Xplore. Restrictions apply.

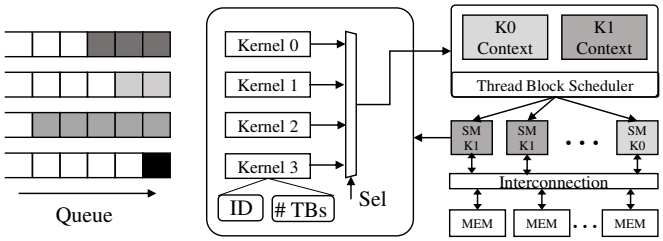

Fig. 2: A baseline GPU architecture with (a) a kernel management unit that determines which kernel to execute, and (b) a thread block scheduler that allocates kernels to different cores.

In spatial multitasking (Fig 2(b)), thread blocks from multiple kernels are assigned to different SMs. Though spatial multitasking shows decent performance gain, SM-level partitioning often suffers from intra-SM resource under-utilization problem such as register files, L1 caches, and execution units. To solve this problem, Simultaneous Multi-Kernel (SMK) which shares several intra-SM resources with multiple kernels has also been proposed [28]. SMK generally has the best performance when a MEM-kernel and a COM-kernel are executed together, and shows the lowest performance when two MEM-kernels are executed concurrently [3]. Recently, NVIDIA Volta [4] architecture supports advanced MPS (Multi Process Service), a system interface to support simultaneous multi-application execution on GPUs using spatial multitasking with a QoS (Quality-of-Service) feature.

B. Metrics & Methodology

Most previous techniques about GPU multitasking [19], [28] have used STP (System ThroughPut) to represent the overall progress of the system [10].

$$STP = \sum_{i=1}^{N} \frac{CPI_i^{(Single)}}{CPI_i^{(Multiple)}} = \sum_{i=1}^{N} \frac{IPC_i^{(Multiple)}}{IPC_i^{(Single)}} \quad (1)$$

Intuitively, STP is a sum of relative performance of each kernel based on IPC value change from single execution to concurrent execution. Unfortunately, STP is not perfectly matched to this work because it assumes that all the kernels are executed concurrently forever. However, we assume that multiple kernels are launched for a predetermined number of times required by the host application, and some kernels might run alone when no more kernels are waiting in stream queue. Therefore, we decide to use STP only as a profiling metric when multiple kernels run, and we use *total execution cycle* as a throughput metric which is the total cycles until all launched workloads are finished.

Name	Kernel	Type	Name	Kernel	Type
SD-1	mb_sad_calc	COM	HS-1	prescan	MEM
SD-2	larger_sad_calc_8	COM	HS-2	intermediates	MEM
SD-3	larger_sad_calc_16	COM	HS-3	main	MEM
CP	lattice6overlap	COM	HS-4	final	MEM
TP	gen_hists	COM	ST	hybrid_coarsen	MEM
LBM	streamcollide	MEM	SP	spmv_jds	MEM

TABLE I: Parboil benchmarks used in real system evaluation.

We select 7 Parboil benchmarks shown in Table I and we executed these workloads on an NVIDIA Volta V100 [17]. Because of the large number of simulations that need to be done to consider various different combinations, we limited our study to 7 workloads from Parboil, but selected representative benchmarks that included both compute-intensive and memory-intensive workloads. For simplicity, we assume 4 workloads are waiting and a maximum 2 concurrent execution is supported on the GPU.

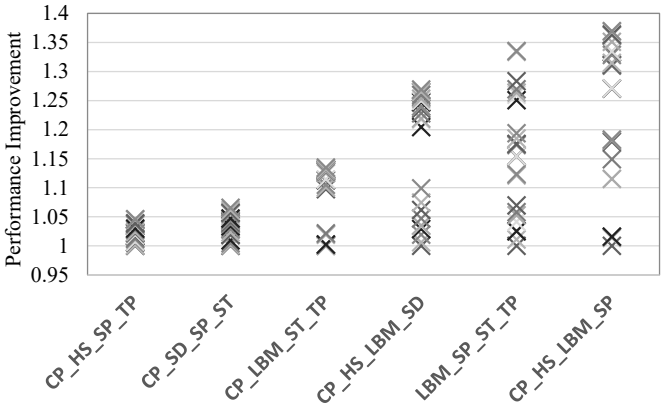

Fig. 3: Performance distribution with different execution order on an NVidia Volta GPU. For 4 workloads, 35 combinations are evaluated and 6 representative cases are shown. Each X represents performance evaluation from 1 out of 24 combination for a given 4 workloads. The results are normalized to the lowest performance combination for workload mix.

III. Motivation

In this section, we demonstrate the potential benefits of multi-kernel scheduling on a real system by evaluating alternative ordering of kernels on an NVidia Volta GPU using Multi Process Service (MPS). We also describe how simple type-based workloads classification technique (based on memory-intensive or compute-intensive workloads), which has been commonly used in previous GPU Multitasking works [6], [19], is not sufficiently effective.

A. Potential Impact of Kernel Scheduling on GPU

We demonstrate the potential impact of kernel scheduling on overall performance. Fig 3 shows the results from a real system where we manually modified the execution order of multiple workloads, to evaluate the potential benefit of kernel scheduling. In this experiment, we launched kernels of 4 workloads at the same time, but changed the order of the workloads to run different permutation of the 4 workloads. All kernels wait until 4 workloads finish memory copy from the host to the device. After all kernels are ready to start, multiple kernels within same workloads consequently executed on the GPU, and each workload is allocated 50% of GPU resources using Volta-MPS.

Fig 3 shows 6 representative results from 35 4-workloads combinations among the workloads. Each result of Fig 3 shows performance variance of all possible permutations (e.g., 24(=4!) with 4 workloads) and the results are shown in ascending order of performance improvement, which is normalized to minimum performance among the 24 scenarios. Fig 3 demonstrates the performance change with different multi-kernel scheduling for given multiple workloads. Based on this evaluation, Fig 3 shows up to 37.3% of the potential maximum performance improvement and 16.9% of average performance improvement over minimum performance execution order. This implies that selection of a well-matched kernel pair among multiple candidates can improve overall system throughput.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:58:21 UTC from IEEE Xplore. Restrictions apply.

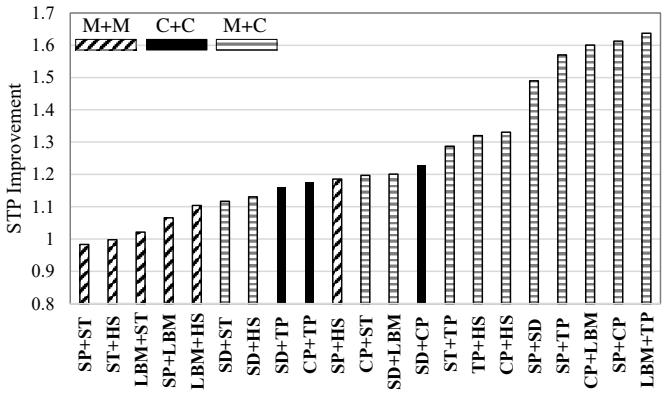

Fig. 4: STP improvement from multi-kernel execution using MPS on an NVidia Volta GPU. The workloads are classified as either memory-intensive (M) or compute-intensive (C), and combinations of workloads are classified as either M+C, C+C, or M+M.

B. Difficulty in Performance Prediction

Prior works on GPU spatial multitasking identified how concurrent execution of different type (i.e, Memory-Intensive (MEM) and Compute-Intensive (COM)) kernels results in better GPU resource utilization [6] [28] [29] [19] [3]. In general, it is assumed that MEM+COM combination provides better utilization of GPU resource (and thus higher performance) while MEM+MEM or COM+COM does not provide high performance because of resource conflict. However, we demonstrate such a simple type-based classification based on only a compute and memory intensity is not sufficient to predict performance since all MEM kernels (and all COM) kernels are not equal. In Fig 4, the STP improvement of all two concurrent execution combinations from 7 parboil benchmarks [24] is shown. The results are executed on the Volta GPU using Volta-MPS service with 50% of QoS.

Similar to the prior work [28], MEM+COM often results in higher STP improvement while MEM+MEM has the lower STP improvement since both workloads compete for the memory bandwidth. However, in some cases, COM+COM or MEM+MEM exceed the performance of MEM+COM cases (e.g., SD+CP and SP+HS show better performance compared to SD+ST).

In addition, for different MEM+COM combinations, the benefit from mixing MEM and COM can vary significantly. For example, ST+TP results in only 28.7% STP improvement while LBM+TP shows 63.7% STP improvement. The reason that the latter case (i.e., LBM+TP) shows better performance improvement is related to their memory intensity – LBM requires more thread level parallelism (TLP) to hide stalls from memory accesses than ST does. Thus, LBM can be referred to have higher memory intensity and the higher memory intensity brings less utilization of functional units which leads to potentially less contention to compute-intensive workloads (TP). As a result, simple classification based only the type (MEM or COM) cannot be the accurate metric to determine *which* workloads to schedule. In the next section, we describe GPU Navigator which finds the optimal kernel combination by using accurate dynamic profiling.

IV. GPU Navigator

In this section, we first describe GPU Navigator that we propose, a dynamic look-up based kernel scheduler that chooses the highest throughput expected kernel pair. We then discuss its effectiveness in contrast to a *greedy* algorithm that maximizes concurrent execution time.

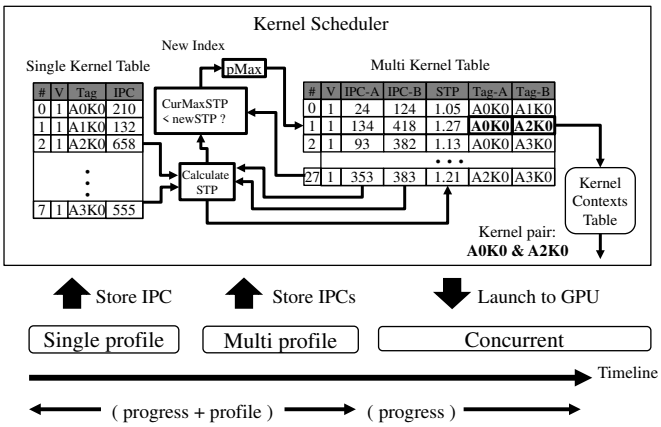

Fig. 5: GPU Navigator overview.

A. GPU Navigator Architecture

The GPU Navigator is implemented in the kernel management unit as a customized kernel scheduler. It is composed of two look-up tables for profiling information of singe kernel and multi-kernel executions, and a best kernel pair selection logic, as shown in Fig 5. The GPU Navigator keeps the dynamic profile information of the previously executed kernels and kernel pairs in two look-up tables: a single-kernel table and a multi-kernel table. A single-kernel table is a cache-like performance table to record IPC information of each kernel on single execution or when executed alone, and a multi-kernel table is a similar cache-like performance table to record IPC numbers of two kernels when executed together and resulting STPs for profiled kernel pairs on concurrent execution. The STPs of kernel pairs are calculated using both single execution IPCs of the kernels from the single-kernel table and IPCs of the kernels on concurrent execution, and are recorded to the multi-kernel table. Unique IDs given to input kernels are used as Tags for the tables. For the single kernel table, an entry contains a Tag from an input kernel, single execution IPC, and a valid bit. An entry of the multi kernel table consists of Tags and IPCs of two co-running kernels, an STP, and a valid bit. Based on the STP information in the multi-kernel table, the best kernel pair selection logic finds the best kernel pairs (with maximum STP) in multiple kernel queues to be executed.

Fig 5 shows a conceptual view of multi-kernel execution of the GPU Navigator. When several new kernels are waiting on multiple kernel queues, it first collects single-kernel IPCs of the kernels and multi-kernel IPCs of possible kernel pairs sequentially as shown in Fig 5 by utilizing hardware performance counters such as *executed instructions* and *active cycles* [5]. Sampling period per profiling is heuristically decided to be 400k cycles for both single and multi execution profiling phases in order to avoid frequent misprediction possibility due to wrong IPC measurement, as discussed in Section III. Along with the multi-kernel IPC measurement, STPs for the kernel pairs are also calculated using two single kernel IPCs and two multi kernel IPCs. GPU Navigator also has a pMax, a pointer to an entry with the highest STP among all the possible pairs from waiting kernels, and the current maximum STP from the entry pointed by pMax is compared to the newly calculated STP to update pMax if the calculated STP is higher. When all the profiling processes are finished, kernel pairs in the entry pointed by pMax are chosen and the information is transferred to the thread block scheduler in order to execute on the target GPU.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:58:21 UTC from IEEE Xplore. Restrictions apply.

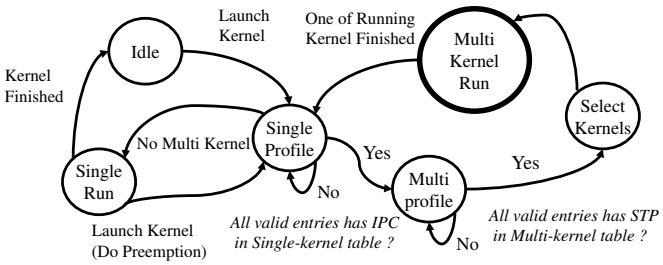

Fig. 6: A state transition diagram of the profiling mechanism.

Fig 6 illustrates an execution state transition diagram of the GPU Navigator. Initially, GPU first moves to the *Single Profile* phase from the *Idle* status when a new kernel is launched. The *Single Profile* phase continues during a sampling period and the navigator then changes a state of the kernel into the *Single Run* phase. If other kernels are ready on other kernel queues, the kernel execution is halted instantly, and the navigator performs single execution profiling of the new kernels by changing the state into the *Single Profile* phase again. For this phase change, a fast preemption technique on GPU multitasking by utilizing various techniques including *drain*, *context switching*, and *flush*, is used similar to the state-of-the-art techniques [19] [25].

The navigator repeats the *Single Profile* phase until all valid entries update their IPC values in the single kernel table, and preemption is performed iteratively after every sampling period in order to profile different kernels. Navigator then changes the state into the *Multi Profile* phase to collect all STPs of possible kernel pairs. Similar to the single profile phase, the navigator performs multi-profiling to prepare all valid entries for all the possible kernel pairs in the multi kernel table. Finally, navigator changes the state into *Select Kernels* phase to choose the best kernel pair to have the highest STP value among possible pairs in the multi-kernel table, and the navigator finds their kernel contexts to launch by looking up the Tag-A/Tag-B values from the entry with the highest STP.

Concurrent execution of a kernel pair continues until a kernel execution is finished. The navigator then changes the execution state into the *Single Profile* state and checks whether there are new waiting kernels, that have not been profiled during the multi-kernel execution, by checking valid entries in the single kernel table. If there is no newly launched kernel, the navigator promptly selects the best kernel pair among the existing kernel pairs. Otherwise, the navigator re-starts *Single/Multi Profile* phases for newly launched kernels and new possible kernel pair combinations.

B. Profiling Overhead

Fig 7 describes how the profiling overhead can be reduced using look-up tables by introducing execution examples of three profile scenarios. The navigator needs to perform profiling of large number of cases initially, but the number of profiling cases can be minimized if the same kernel is re-launched. In this example, we assume that the GPU navigator can handle up to 4 kernels from 4 different applications, and each application contains multiple kernels. Fig 7(a) shows the profiling process when 4 kernels are initially launched from 4 different applications (A, B, C, D). As there is no previously profiling information in the tables, single-profiling should be executed for all waiting kernels, multi-profiling should be executed for all possible pairs ($_4C_2$ in Fig 7), and the navigator finally decides to execute the best kernel pair of (A0K0, A2K0). Fig 7(b) shows the next execution scenario when a short kernel (A2K0) is finished and

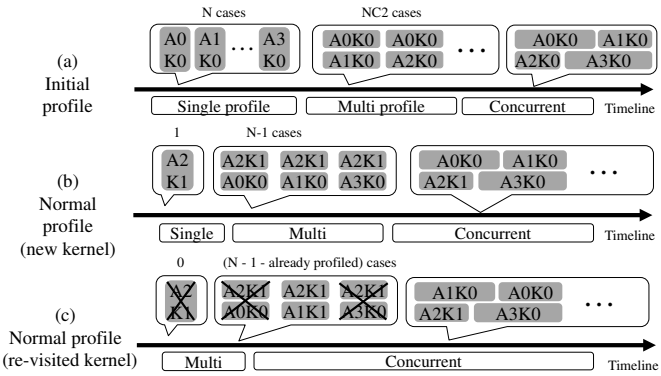

Fig. 7: Example profiling scenarios of (a) an initial profile, (b) a normal profile with a newly launched kernel and (c) a normal profile with a revisited kernel.

a new kernel (A2K1) is introduced to the queue. In this situation, both single and multi-profile processes are performed to identify whether the new kernel can change the best kernel pair. For this case, only one single profiling process and small number of multi-profiling processes (N-1 cases, where N is the number of kernel queues) are required because all other information is already ready in single/multi kernel tables. Fig 7(c) shows another scenario when a kernel is re-executed. In this example, A2K1 kernel is assumed to be re-executed, only A1K1 was not the waiting kernel when the A2K1 kernel was executed before, and previous profiling data are already recorded in the tables. Therefore, single-profiling process is not required, and multi-profiling processes except a (A2K1,A1K1) pair can be skipped as the information was updated before. Finally, A1K0-A2K1 pair is selected to run concurrently with the minimum profiling time.

C. Comparison with Greedy algorithm

As multiple kernels have different execution times, a GPU runs single kernel at the end of concurrent execution. Based on a simple high-level insight that concurrent execution mostly shows higher GPU utilization than single kernel execution, a simple *greedy* algorithm can be introduced to maximize concurrent execution time by launching the longest kernel first and executing other kernels that are well-matched to the longest kernel.

To find the longest kernel among all the waiting kernels, kernel execution time estimation is required. The estimation can be performed by executing a subset of total threads and predicting the total execution time by considering total number of threads. For this, we run 4% of total thread blocks (empirically determined) to know the average execution cycles of a thread block, and calculate the estimated total execution time by multiplying the number of thread blocks for all the waiting kernels. The estimated time is then saved in a look-up table, similar to the single-kernel table of the GPU navigator. Based on the information, a longest kernel is identified and a co-executing kernel can be chosen by performing multi-kernel profiling only for pairs of the longest kernel and other kernels. Therefore, it is expected to have light profiling overhead compared to the GPU Navigator because it does not require multi-kernel profiling of all possible pairs. However, the greedy policy still shows limited performance improvement because it cannot consider all the kernel combinations and the best co-executing kernel pair does not necessarily include the longest kernel. Another problem is that the single profiling overhead of the greedy policy is often higher since at least one thread block with long execution time must be executed for profiling.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:58:21 UTC from IEEE Xplore. Restrictions apply.

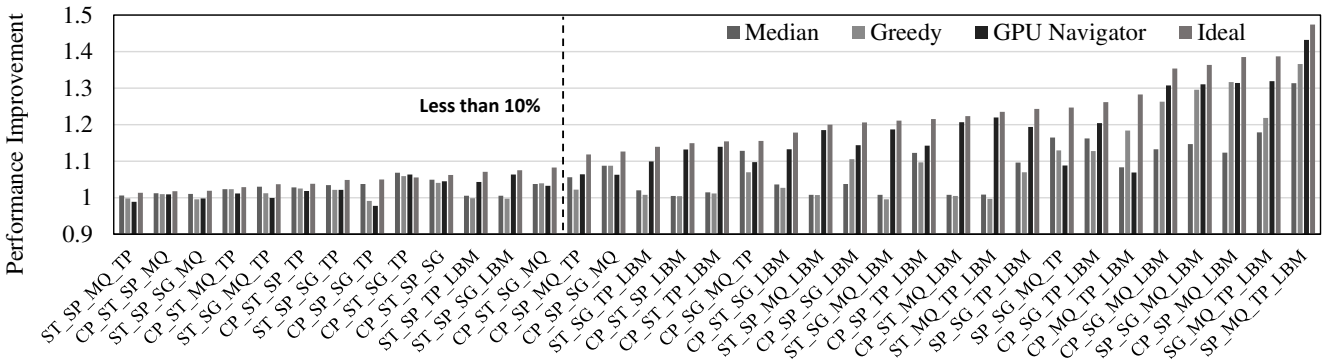

Fig. 8: The performance of all possible combinations using various scheduling policies including median, greedy, and GPU Navigator. Ideal indicates the maximum performance improvement with a best execution order.

Simulation	Parameters
System	8 Hardware Queue 2 Concurrency
Core(SM)	15 SMs, 700Mhz L1 Cache 16KB 2 warp Scheduler
Memory Subsystem	768KB L2 Cache 6 Memory partition 2 Sub partition 924Mhz Clock

TABLE II: GPGPU-Sim Configuration

Name	Kernel Name	Type	Iteration	Length (cycles)
MQ	computephimag_gpu computeq_gpu	COM COM	1 2	1.5K 9.2M
CP	lattice6overlap	COM	11	1.2M
TP	gen_hists	COM	1	6.2M
SG	mysgemmNT	COM	1	6.6M
ST	block2d_hybrid_coarsen_x	MEM	40	1.5M
SP	spmv_jds	MEM	50	0.9M
LBM	performstreamcollid	MEM	10	4M

TABLE III: Parboil benchmarks used for simulation evaluation of GPU Navigator.

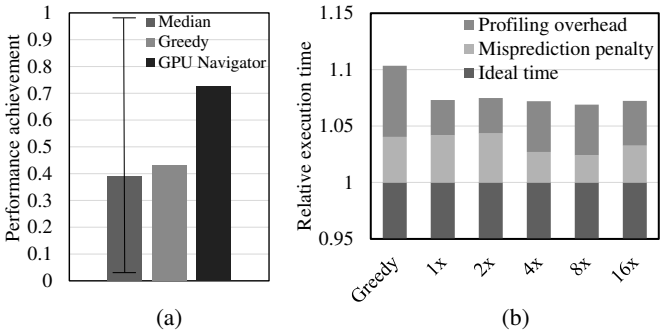

Fig. 9: (a) The average performance achievement, and (b) relative execution time distributions at different profiling periods.

V. Evaluation

A. Methodology

We used GPGPU-Sim v3.2.2 [9] with the Table II configuration. The baseline system is extended to run two concurrent execution of multi-programmed workloads with support of up to 8 hardware queues. The baseline system also utilizes smart preemption techniques [19] with SMK support. We selected 7 memory (MEM)/computation (COM)-intensive workloads in the parboil benchmark suit [24] (Table III), and their iterations are adjusted in order to have similar total execution times.

4 workloads are assumed to be launched concurrently, and the GPU Navigator is compared to median and greedy (Section IV-C) policies. Median policy chooses the execution order that shows the median performance of all the 24(=4!) possible kernel pairs from 4 input workloads. We also assumed that various meta-data such as each kernel's own unique identifier are generated from the compiler and embedded to the execution binary.

B. Performance Evaluation

Fig 8 shows the performance improvement of multiple combinations when choosing 4 workloads from total 7 workloads. The results show the relative performance of execution orders chosen from median, greedy, and GPU Navigator policies, and the best execution order. The performance results are normalized by the performance of the worst order, and the results are sorted in ascending order of best order performance. For the left side of Fig 8, which are the workload combinations with low maximum performance gains, greedy and GPU Navigator shows lower performance than the worst case because of profile overheads. However, for the right side of Fig 8, which are combinations with high performance gain, GPU Navigator usually shows the highest performance improvement.

Fig 9(a) shows the average performance achievement of median, greedy, and GPU Navigator. In this graph, the performance gain of each policy is normalized by the performance gain of the best order over the worst order. The median policy shows the 38.8% average performance achievement without any profile overheads. The performance achievement of median have variance. The performance of median policy (i.e., random) possibly results from a range of the worst execution order to best execution order. An interesting result is that the greedy policy shows only 43.2% performance achievement. This shows that to maximize the concurrent execution time does not always result in high performance gain. However, the GPU Navigator shows the 72.5% performance achievement in an average from all combinations. While it cannot show the same performance compared to the best execution order because it often finds suboptimal orders due to low profile accuracy (mis-prediction penalty) and has some profile overheads, it is proved to show fair performance for most combinations.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:58:21 UTC from IEEE Xplore. Restrictions apply.

C. Misprediction and Profile Overhead Analysis

In order to evaluate the misprediction penalty and profile overheads of the GPU Navigator, we measure the performance gains by changing the sampling period from 50k (1x) to 800k (16x). Fig 9(b) shows execution time breakdown of a greedy policy and GPU Navigator with different sampling periods per profiling. In this Figure, the total execution time is normalized to the best execution time and ideal time, mis-prediction penalty, and profiling overhead are shown. Mis-prediction penalty means the additional execution time to the execution time of the best execution order, as the execution time increases due to the wrong execution order selection (mis-prediction). Profiling overhead is the time for single and multi kernel profiling. The greedy policy shows low performance because it has high mis-prediction penalty due to wrong order selection, and it further requires substantial profiling overheads to find the longest kernel as discussed in Section IV-C. On the other hand, Navigator shows fair performance penalties from both misprediction and profiling.

As longer profiling time can reduce the misprediction penalty by providing more accurate STPs for testing kernel pairs, to find the best profiling period is the most critical challenge for the GPU Navigator system. As profiling overhead can be reduced by utilizing single/multi kernel lookup tables, to minimize the misprediction penalty is more important mission than profile overhead minimization. From this point of view, 400k can be considered as the best sampling period because it shows the minimum misprediction penalty, with the best performance among all the cases.

VI. Related Work

Symbiotic scheduling to co-schedule multiple threads with different characteristics on SMT architectures [11], [22] is similar to this work in a high-level concept. The priority-aware symbiotic scheduler [12], [23] co-schedules low priority processes with high priority ones to increase resource utilization. However, their works can not be directly applicable to throughput-oriented GPU architectures. A previous work [15] also studied the ordering effect of multiple kernels based on their static performance modeling on the GPU side. Their modeling is based on resource utilization, which is similar to our greedy algorithm. However, the study is limited to only one process that has multiple kernels. The kernelet [31] also proposed a kernel slicing technique to divide a kernel into multiple kernels with fixed size thread blocks and their co-scheduling algorithm using Markov-chain. However, the study can be applied to only the divided kernels rather than multi-programmed kernels and the performance model is hard to detect performance variance between multiple kernels.

VII. Conclusion

Although various multi-kernel handling approaches have been proposed for GPUs, prior work focused on appropriate resource assignment after target kernel selection. In this paper, we first demonstrate how *ordering* of kernels being scheduled on GPU's can have significant impact on overall performance and system throughput. To address this challenge, we propose a GPU Navigator, a dynamic multi-kernel scheduling system to calculate STPs of multiple combinations using two-stage sampling and select the most efficient co-running kernel pairs to improve overall throughput. Our proposed design shows better performance than median and greedy policies by 33.7% and 29.3% on average.

Acknowledgment

We would like to thank to Jason Jong Kyu Park for his valuable feedback. This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP)(No.2018R1D1A1B07050609, No.2015M3C4A7065647, No.2020R1A2B5B01001687), ICT R&D program of MSIP/IITP (No.2017-0-00142), and the R&D program of MOTIE/KEIT (No.10077609). Yongjun Park is the corresponding author.

References

[1] "NVIDIA's next generation cuda compute architecture: Kepler GK110," 2012.
[2] "Sharing a gpu between mpi processes: Multi-process service overview," 2014.
[3] "Dynamic resource management for efficient utilization of multitasking gpus," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 527–540.
[4] "NVIDIA's the world's most advanced data center gpu Volta V100," 2017.
[5] *CUDA Profiler User's guide*, 2018.
[6] J. T. Adriaens *et al.*, "The case for gpgpu spatial multitasking," in *IEEE International Symposium on High-Performance Computer Architecture*, Feb 2012.
[7] P. Aguilera, K. Morrow, and N. S. Kim, "Qos-aware dynamic resource allocation for spatial-multitasking gpus," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2014, pp. 726–731.
[8] Amazon, "Amazon web services," https://aws.amazon.com/ec2/.
[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. of the 2009 IEEE Symposium on Performance Analysis of Systems and Software*, Apr. 2009.
[10] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, 2008.
[11] S. Eyerman and Eeckhout, "Probabilistic job symbiosis modeling for smt processor scheduling," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736033
[12] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, "Symbiotic job scheduling on the ibm power8," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 669–680.
[13] Google, "Google clould platform," https://cloud.google.com.
[14] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving gpgpu resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 260–271.
[15] T. Li, V. K. Narayana, and T. El-Ghazawi, "Symbiotic scheduling of concurrent gpu kernels for performance and energy optimizations," in *Proceedings of the 11th ACM Conference on Computing Frontiers*, ser. CF '14. ACM, 2014.
[16] Microsoft, "Microsoft azure," https://azure.microsoft.com.
[17] NVIDIA, "NVIDIA DGX-1 Tesla V100."
[18] S. Pai *et al.*, "Improving gpgpu concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 407–418. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451160
[19] J. J. K. Park *et al.*, "Chimera: Collaborative preemption for multitasking on a shared gpu," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 593–606.
[20] J. J. K. Park, Y. Park, and S. Mahlke, "Dynamic resource management for efficient utilization of multitasking gpus," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 527–540, 2017.
[21] D. Sengupta, R. Belapure, and K. Schwan, "Multi-tenancy on gpgpu-based servers," in *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, ser. VTDC '13. New York, NY, USA: ACM, 2013.
[22] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX. New York, NY, USA: ACM, 2000, pp. 234–244.
[23] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic jobscheduling with priorities for a simultaneous multithreading processor," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '02. New York, NY, USA: ACM, 2002.
[24] J. A. Stratton *et al.*, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.
[25] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 193–204.
[26] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. Swift, "A placement vulnerability study in multi-tenant public clouds," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 913–928.
[27] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 18:1–18:17.
[28] Z. Wang *et al.*, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 358–369.
[29] Q. Xu *et al.*, "Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 230–242.
[30] M. K. Yoon *et al.*, "Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit," in *Proc. of the 43rd Annual International Symposium on Computer Architecture*, Jun. 2016.
[31] J. Zhong and B. He, "Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1522–1532, June 2014.

Authorized licensed use limited to: Korea Advanced Inst of Science & Tech - KAIST. Downloaded on March 04,2026 at 01:58:21 UTC from IEEE Xplore. Restrictions apply.