smCompactor: A Workload-aware Fine-grained Resource Management Framework for GPGPUs*

Qichen Chen
Seoul National University
Seoul, South Korea
charliecqc@dcslab.snu.ac.kr

Hyerin Chung
Seoul National University
Seoul, South Korea
hrchung@dcslab.snu.ac.kr

Yongseok Son
Chung-Ang University
Seoul, South Korea
sysganda@cau.ac.kr

Yoonhee Kim
Sookmyung Women's University
Seoul, South Korea
yulan@sookmyung.ac.kr

Heon Young Yeom
Seoul National University
Seoul, South Korea
yeom@snu.ac.kr

ABSTRACT

Recently, graphic processing unit (GPU) multitasking has become important in many platforms since an efficient GPU multitasking mechanism can enable more GPU-enabled tasks running on limited physical GPUs. However, current GPU multitasking technologies, such as NVIDIA Multi-Process Service (MPS) and Hyper-Q may not fully utilize GPU resources since they do not consider the efficient use of intra-GPU resources. In this paper, we present smCompactor, which is a fine-grained GPU multitasking framework to fully exploit intra-GPU resources for different workloads. smCompactor dispatches any particular thread blocks (TBs) of different GPU kernels to appropriate stream multiprocessors (SMs) based on our profiled results of workloads. With smCompactor, GPU resource utilization can be improved as we can run more workloads on a single GPU while their performance is maintained. The evaluation results show that smCompactor improves resource utilization in terms of the number of active SMs by up to 33% and it reduces the kernel execution time by up to 26% compared with NVIDIA MPS.

CCS CONCEPTS

• **Computer systems organization → Operating systems**; **GPU system**;

KEYWORDS

HPC, OS, GPU multitasking, GPU resource management, Parallel computing

ACM Reference Format:
Qichen Chen, Hyerin Chung, Yongseok Son, Yoonhee Kim, and Heon Young Yeom. 2021. smCompactor: A Workload-aware Fine-grained Resource Management Framework for GPGPUs. In *Proceedings of ACM SAC Conference (SAC'21)*. ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/xx.xxx/xxx_x

*Produces the permission block, and copyright information

1 INTRODUCTION

Graphics processing unit (GPU) has been widely deployed in many platforms since massive research fields such as deep learning and high-performance computing require its exascale computational power [25]. Currently, GPU is available in supercomputers such as Summit [21] and cloud services such as Google [5], Amazon [3] due to their high performance on computational intensive workloads and energy efficiency. To fully utilize the resource of GPU in these platforms, efficiently sharing GPU resources is a challenging work [7].

GPU multitasking improves the GPU resource utilization where multiple GPU kernels with different resource consumption patterns can run on the GPU simultaneously [14]. NVIDIA provides Hyper-Q[10] and MPS [14] technologies to enable GPU multitasking on NVIDIA GPUs. The Hyper-Q technology makes multiple CUDA kernels from the same CUDA context [12] issued within different CUDA stream [13] being scheduled simultaneously. The MPS technology expands the single context multitasking into multiple contexts. In our research, we found that GPU workloads can achieve its best performance even use parts of the GPU intra-resources if the dispatching of thread blocks is well organized. In detail, depending on their features such as memory and computation consumption, some workloads need fewer SMs with more thread blocks dispatched on each SM while others need more SMs with fewer thread blocks on each SM to obtain their best performance. However, both the Hyper-Q and MPS cannot exploit GPU resources efficiently, since they use the left-over policy to dispatch thread block to each SM, which is unaware of the relation between resource usage pattern and the performance of each kernel. In addition, kernels run with MPS are not preemptable, which means that small kernels have to wait for large ones to finish, downgrading the quality of service (QoS). Finally, MPS does not support dynamic parallelism [1], which is widely used due to its additional parallelism that can be exposed to the GPU scheduler and load balancer.

To handle these issues, previous studies [18, 23, 24] proposed both hardware and software-based strategies to improve the performance when multiple kernels run in parallel with or without MPS. Slate [2] introduced a workload-aware kernel-based scheduling system to improve performance when multiple workloads share the GPU. It efficiently reduces the interference caused by co-locating and isolating kernels into separate SMs. Our study is inline with these studies [2, 18, 23, 24] in terms of investigating the performance of GPU. In contrast, we focus on improving the resource utilization of intra-GPU.

In this paper, we aim to improve the utilization of GPU resources and reduce the wall time of kernel execution in the multitasking environment. To do this, we first analyze the performance of GPU workload and observe an optimal kernel execution time can be achieved when part of the GPU resources are used if the dispatching of thread blocks is well organized. Second, we devise smCompactor as a GPU multitasking framework that can improve intra-GPU utilization. It dispatches thread blocks (TBs) of different GPU kernels to appropriate steam multiprocessors (SMs) based on our analyzed results. This smCompactor can fully exploit the intra-GPU resources, including the shared memory, registers, and stream processors (SPs) by supporting more workloads that can share the GPU at the same time. We implement smCompactor and evaluate it on an NVIDIA Titan Xp based system. The experimental results show that smCompactor can improve resource utilization in terms of activated SM by up to 33% and reduce the execution time by up to 26% compared with MPS.

2 BACKGROUND AND MOTIVATION

2.1 Resource Sharing Model

In NVIDIA GPU, there are two resource sharing models which are the time-sharing model and the spatial-sharing model. Originally, multiple concurrent launched kernels with different CUDA streams are scheduled using the time-sharing model. In this model, time budgets are assigned to each GPU kernel. Each kernel can utilize GPU resources only when both its time budget and the required resources are available. After its time budget expires, other kernels are executed through context switching.

On the other hand, the spatial-sharing model is used by exploiting NVIDIA Hyper-Q and MPS technology. In this model, kernels launched on the GPU are scheduled depending on only their resource usage. In this way, multiple independent kernels can be simultaneously executed on different SMs, as long as their resource requirements are satisfied.

2.2 Multi-Process Service with Hyper-Q

The NVIDIA MPS [14] is a binary-compatible client-server runtime implementation of the CUDA API. It is designed to enable cooperative multi-process CUDA applications in a concurrent manner using Hyper-Q. Starting from the Fermi architecture [9], Hyper-Q enables multiple CPU threads or processes to launch tasks on a single GPU simultaneously. It allows CUDA kernels to be processed concurrently on the same GPU, which can improve the performance and utilization of GPU resources. While the concurrent scheduling of Hyper-Q is limited to a single CUDA context, MPS with Hyper-Q can collect the contexts from different applications and map them into a single one. MPS works in a client-server architecture, the control daemon of MPS acts as the server, which coordinates connections between the clients and the server. MPS client runtime is built into the CUDA driver library and can be used transparently by any CUDA applications. The client passes its kernel and its CUDA context to the server, and the server merges them into one context.

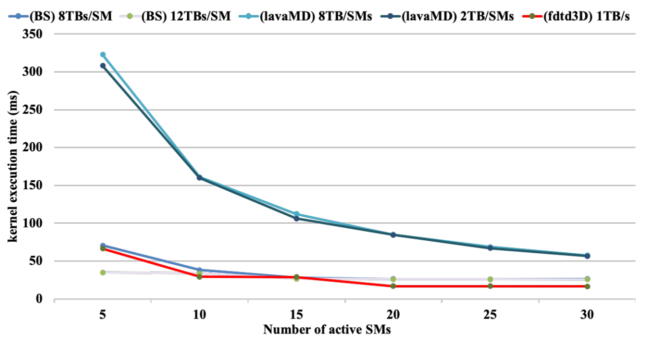
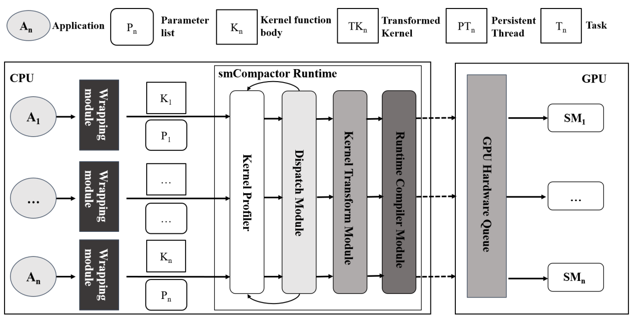


Figure 1: Kernel execution time varies with launched thread block per stream multiprocessor (SM) and active SM changes. (BlackSholes (BS), lavaMD (LM), fdtd3D(FT)

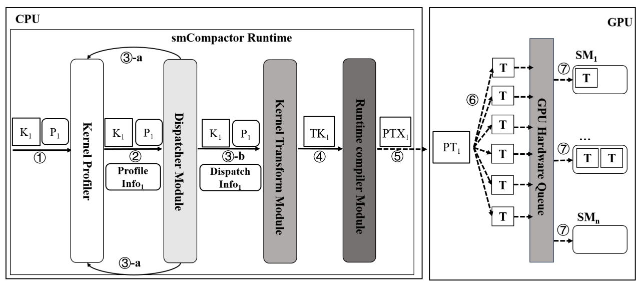
However, multitasking with MPS cannot fully exploit the intra-GPU resources, because it uses left-over policy to schedule thread blocks of the different kernels, ignoring the relation between resource usage patterns of kernels and their performance. The MPS may also block the later-launched kernel until the previously launched one finishes its execution, decreasing the QoS and resource utilization if the former kernel is not resource-intensive with long execution time. We verified this property by staggering the launch time of two workloads (FDTD3d, lavaMD) with a huge difference in their kernel execution time. The results show that in the worst case, the execution time of later launched small workloads can reach to 460 times of its normal execution time.

2.3 Motivation

Because the GPU was originally used for processing pixel rendering, multitasking on GPU is not as efficient as on CPU since it is initially designed for monopoly use for a single workload. Besides, monopolizing the whole GPU causes a waste of resources since the best performance of some workloads can be achieved when parts of the resources are used if their thread blocks are efficiently dispatched. Figure 1 shows the relation between the execution time and the number of activated SMs with a various number of thread blocks launched on each SM for different workloads. As shown in the figure, two different features appeared among these workloads. Workload like BlackScholes (BS) needs fewer SMs (15) but with many thread blocks(12) on each SM to obtain its best performance. On the other hand, workload as lavaMD (LM) needs more SMs (30) while with fewer thread blocks per SM (2) to achieve its best performance. Both of the thread block dispatching policy mentioned above consumes part of the GPU resources, for example, 12 thread blocks of BlackScholes requires 35328 registers in each SM. NVIDIA Titan Xp [11] has 65536 registers in each SM and the remaining 30208 registers can be used to launch four thread blocks of lavaMD additionally on the same SM which are described in Table 1 in the evaluation section. However, the default scheduling policy of MPS uses the "left-over" policy, which will dispatch the thread block to another SM only when the previous one is fully occupied. This dispatch policy can not exploit the best utilization of GPU intra-resource when multiple tasks running concurrently. The newest NVIDIA Volta MPS [14] provides an SM-level resource provisioning, which can allocate the thread blocks of the kernel to specific SMs. It is useful for workload such as BlackScholes, which needs



(a) Overview of the smCompactor architecture



(b) Control flow of the smCompactor

Figure 2: System architecture and control flow of smCompactor

fewer SMs. However, workloads need more SMs with fewer thread blocks on it such as lavaMD may still not execute efficiently with Volta MPS.

Inspired by these observations, we proposed smCompactor, where thread blocks of each kernel can be intentionally launched to specific SMs to obtain a near-optimal performance according to the profiling result while using as few resources as possible, leaving more resources available for executing other workloads.

3 DESIGN AND IMPLEMENTATION

There are several goals in our design of the smCompactor. First, smCompactor aims to support GPU multitasking without using NVIDIA MPS to complement the shortcomings of the MPS. Second, smCompactor aims to dispatch task (at thread block level) efficiently to exploit the GPU resources efficiently and reduce the wall time of kernel execution; Third, smCompactor aims to provide a GPU kernel multitasking environment that is transparent to the users. To achieve our design goal, we perform several works as follows. (1) we profile the kernel information including the resource consumption, predicting the necessary number of SMs and thread blocks per SM to achieve the best performance. (2) we automatically extract the kernel function and its parameters from the source code, (3) we convert them into a manually controllable version transparently, (4) we merge CUDA contexts into a single context and finally managing the thread block dispatching according to that information.

3.1 System Architecture

Our approach is based on the client-server structure. As shown in Figure 2(a), each application is individually bound to the CUDA API wrapping module, which acts as a client. The CUDA API wrapper module intercepts the original CUDA kernel functions and their parameter lists and passes them to the smCompactor runtime daemon that runs on the host side. As shown in the figure, it contains the kernel profiler, dispatch module, kernel transform module, and runtime compiler module.

Figure 2(b) shows the control flow of the smCompactor runtime. When the application calls CUDA API to allocate device memory and launch a kernel, the wrapping module intercepts the kernel function body and parameter lists and forwards them to the smCompactor runtime (①). The kernel profiler profiles the kernel and forwards the profiling result to the dispatch module (②). The dispatch module adds dispatching related information to the kernel function body and forwards them to the kernel transform module (③-b); however, if the dispatch information of a specific kernel changes, it sends modified dispatch information back to the kernel profiler and triggers a new profile (③-a). The kernel transform module converts the original kernel function into a transformed version with dispatching information, which can be adopted in the persistent thread model [6]. Then, it forwards the transformed kernel to the runtime compiler module (④). The runtime compiler module compiles the transformed kernel source into a ptx file, and launches the modified kernel from that file (⑤). After the transformed kernel is launched on the GPU side, a persistent thread is generated to dispatch the thread blocks as tasks (⑥) to the specific SMs (⑦) according to the implanted dispatching information. We will discuss each module in the smCompactor runtime in detail in the following subsections.

3.2 CUDA API Wrapping Module

Overall, CUDA applications use the CUDA runtime and driver APIs to communicate and control the NVIDIA GPU. In our proposed design, smCompactor runtime is responsible for transforming the original kernel functions received from each client into the modified version that can be scheduled; thus, the kernel functions and their parameters should be separated in advance before being transferred to the runtime.

However, as the CUDA API is not open source, we cannot modify the CUDA function itself; instead, we develop the CUDA API wrapping module to capture the APIs mentioned above and implement our design to derive the kernel function from the source code transparently. Particularly, we intercepted the *cudaLaunch* call to get the kernel function body as a string by parsing the *entry* parameter used in the *cudaLaunch* function. Finally, we transfer the kernel function source and their parameter list as well as their sizes to the smCompactor runtime.

3.3 Kernel Profiler

The kernel profiler is responsible for profiling the thread block dispatch features of each kernel. As introduced in [4], the Empirical Criteras (EC) can be used to distinguish a workload into computational intensive or memory intensive (large EC indicates computational intensive). According to the EC of each workload, we categorize our workload and analyze the relation between their performance and thread block dispatch pattern. Figure 3 and Figure 4 demonstrate their performance changes depending on the dispatching of the thread blocks on various number of SMs.

Figure 5 and Figure 6 show the change of global load throughput and achieved occupancy of each workload with different thread

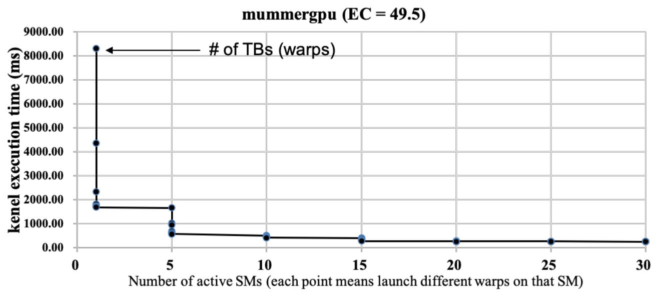


Figure 3: Performance of a various number of thread blocks (1,2,4,8) dispatched on SMs for mummergpu.

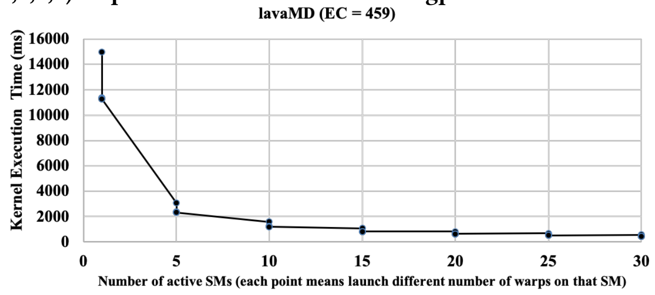
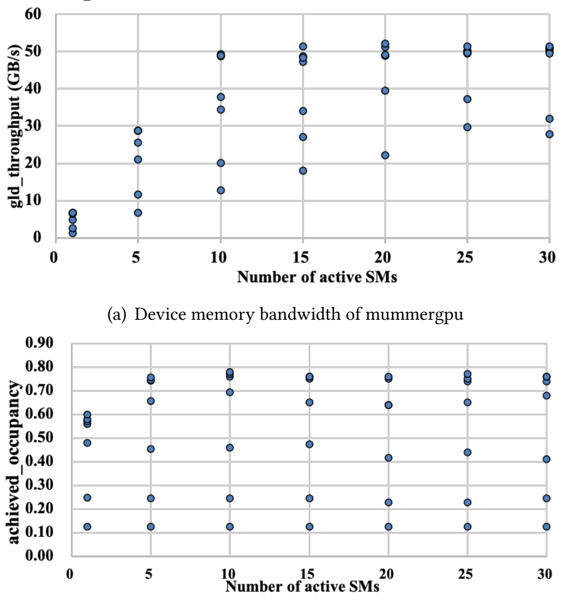


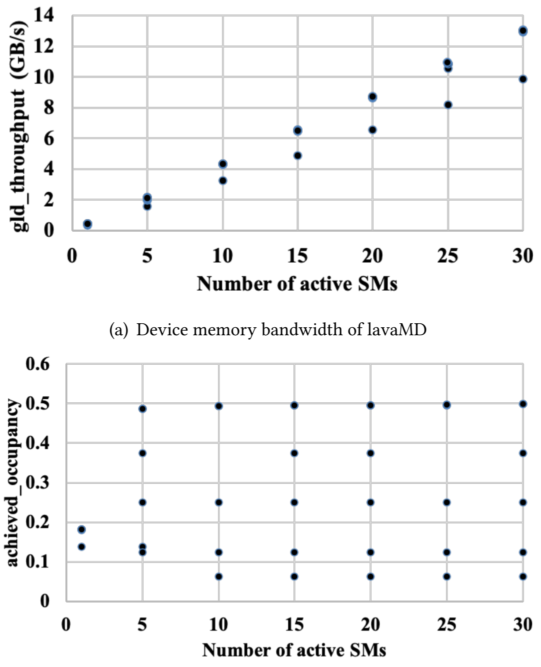
Figure 4: Performance of a various number of thread blocks (1,2,4,8) dispatched on SMs for lavaMD.



(a) Device memory bandwidth of mummergpu

(b) Parallelism of mummergpu

Figure 5: Device memory bandwidth and parallelism of mummergpu on various thread block dispatching



(a) Device memory bandwidth of lavaMD

(b) Parallelism of lavaMD

Figure 6: Device memory bandwidth and parallelism of lavaMD on various thread block dispatching

block dispatch strategy. Comparing with Figure 3 and Figure 4, we can find that the memory bandwidth has more impact than parallelism on the performance. We also find a saturated gld/gst_throughput can indicate the best performance. As a result, the kernel profiler profiles global store throughput (*gst_throughput*), and global load throughput (*gld_throughput*) for different number of the thread blocks when all SMs are activated and only one SM is activated. We also observe that the gld/gst_throughput is directly proportional to the number of SM and thread blocks per SM before they are saturated. The kernel profiler collects this information for the first time when each kernel is launched.

The profiler obtains these features by utilizing NVIDIA CUDA Compiler (NVCC) [15] options and NVProf [16]. After profiling, the profiled results are forwarded to the dispatch module and based on this collected information, we calculate how many resources remain in each SM and decide how many more thread blocks can be issued to each SM.

3.4 Dispatch Module

The dispatch module injects the dispatching information into the original kernel according to the obtained profiling information from the kernel profiler. The dispatching information contains the thread block and SM mapping data. It will be a guideline to dispatch thread blocks of different kernels to each SM to maximize resource utilization while achieving optimal performance. Particularly, it is created by considering all related features such as profiling result, current resource usage of each SM, and the resource consumption of target kernels. As mentioned above, the best kernel performance can be achieved when (*gst_throughput* and *gst_throughput*) are saturated. Both of these two features are functions of the number of activated SMs and the number of warps running on each SM. In detail, the *gld_throughput* and *gst_throughput* firstly increase as the numbers of warps and activated SMs increase. However, at some point, the throughput is saturated even if the numbers of warps and activated SMs increase. We noticed that before the throughput is saturated, the increment of these two features is directly proportional to the increment of warps and SMs. Thus, the dispatch module can use the profiled information of *gst* and *gld_throughput* in one SM and the

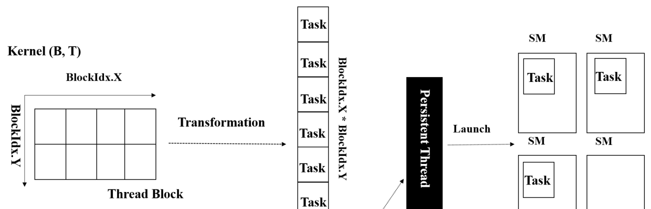


Figure 7: Kernel transformation.

saturated *gst* and *gld_throughput* to infer the increment coefficient and calculate the necessary number of thread blocks and SMs.

3.5 Kernel Transform Module

Generally, GPU thread blocks are scheduled and dispatched to random SMs depending on the hardware scheduler. Thus, to improve the resource utilization according to each kernel's feature, manually controlling the dispatch of the GPU thread block is difficult to implement. Thus, [6] proposed the concept of the persistent thread model, which is based on the dynamic parallelism mechanism provided by NVIDIA. In this programming model, thread blocks are treated as tasks; meanwhile, a persistent thread is running permanently, picking up tasks from the task queue, and launching the tasks asynchronously.

Our proposed smCompactor adopts this persistent thread mechanism by modifying the original kernel into a revised version. The modification details are as follows: First, the kernel transform module replaces the built-in CUDA variables *blockIdx* and *gridIdx* with the persistent thread model related code segments to implement its functionality. As shown in Figure 7, a general CUDA kernel with a multi-dimension grid will be first converted into a one-dimension grid where the size of the grid is equal to the product of the size of each dimension in the original grid. Each thread block in the converted dimension is treated as a task. Meanwhile, a persistent thread, which is also a kernel function running permanently on the device side, is created to sequentially extract tasks and launches them to the SM asynchronously. Then, dispatching information obtained from the dispatch module is integrated into the revised kernel; therefore, thread blocks of certain kernels can run on the designated SM. Finally, the revised kernels are launched with different CUDA streams [13] to exploit the concurrent scheduling features of the hardware scheduler.

In our proposed smCompactor, we use a "fulfill and retreat" strategy to control the dispatch of the thread blocks. The strategy keeps trying to dispatch the task (thread block) until it locates the specific SM and until the number of thread blocks matches the number in the dispatching information. As introduced above, the NVIDIA hardware thread block scheduler may dispatch thread blocks to any of the SMs according to its resource usage. Since the details of the NVIDIA hardware scheduler are not available to the public, the distribution of thread blocks is random to the users. However, the "fulfill and retreat" strategy takes advantage of the persistent kernel model where thread blocks are treated as tasks to use an alternative means to solve the problem by trying to dispatch the task (thread block) contiguously. We note that the

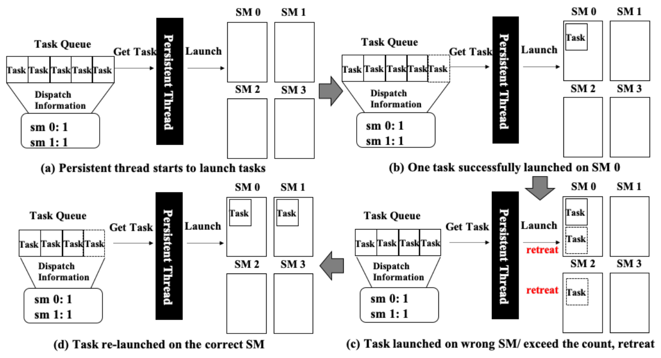

Task Queue
Task
Dispatch Information
sm 0: 1
sm 1: 1
Get Task
Persistent Thread
Launch
SM 0
SM 1
SM 2
SM 3
(a) Persistent thread starts to launch tasks
(b) One task successfully launched on SM 0
retreat
(d) Task re-launched on the correct SM
(c) Task launched on wrong SM/ exceed the count, retreat

Figure 8: Details of fulfill and retreat mechanisms

persistent kernel mode is necessary because original kernels are launched inside a kernel (persistent thread) and current task queue status needs to be synchronized between the task (consumer) and the persistent thread (producer). A CUDA device synchronization variable can be modified and testified by both of them without accessing host memory since both of them are running on the device.

Figure 8 illustrates the details of the "fulfill and retreat" mechanism. As shown in Figure 8(a), the persistent thread starts to dispatch tasks to the SMs according to the dispatching information. Currently, two tasks need to be dispatched to SM0 and SM1, respectively. In the beginning, as shown in Figure 8(b), the persistent thread successfully dispatches one task to SM0 by the hardware scheduler. In this case, the persistent thread pops this task from the queue and continues to dispatch the next one. As shown in Figure 8(c), the next task is dispatched to SM2 (wrong SM in this case), which violates the dispatch information. As a result, the task needs to retreat and the persistent thread dispatches again. After dispatching, at this time, the task is located on SM0, which exceeds the threshold of the thread block count. Thus, the task should be treated again. Finally, as shown in Figure 8(d), the task is successfully dispatched to SM1, and the persistent thread pops the task from the queue and prepares to dispatch the next one until the queue is empty.

3.6 Runtime Compiler Module

The runtime compiler module compiles the modified kernels which are converted by the kernel transform module into an executable form. It works in four stages. First, the runtime compiler module creates a unique CUDA context since as mentioned above, NVIDIA Hyper-Q technology enables concurrent scheduling with CUDA streams where the kernels should belong to the same CUDA context. Second, it compiles the source code into ptx form. Third, it allocates device memory for each kernel in that CUDA context. Fourth, it launches the kernels on the GPU.

The runtime compiler module contains one main thread and several child threads. The main thread creates the unique CUDA context and receives the modified kernel from the previous module. Each child thread obtains the main context and corresponding kernel from the main thread, compile the kernel into a ptx file, allocate device memory, and launching the kernel under the main context.

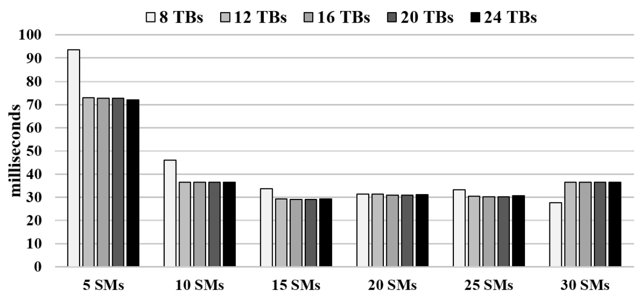


Figure 9: Execution time of BlackScholes with the different number of thread blocks (TBs) on the different number of SMs.

4 EVALUATION

In this section, we evaluate the performance of smCompactor on several real-world applications with different scenarios. The evaluations are executed on a real GPU system, which consists of an NVIDIA Titan Xp GPU card, Intel Xeon E5-2683 CPU with 14 physical cores, and 64GB DDR3 memory. The NVIDIA Titan Xp GPU belongs to the Pascal architecture, which consists of 30 SMs, with 65536 registers and 64KB shared memory in each SM. The whole system runs on the Ubuntu 16.04 operating system, with Linux kernel 4.4.180. The NVIDIA device driver version we used is 384.130, with the NVIDIA CUDA toolkit 9.1.

4.1 Evaluation Methodology

We evaluate our proposed smCompactor with the following performance metrics. 1) real & normalized kernel execution time, 2) resource utilization in terms of the number of active SMs and thread blocks for each workload under a multitasking environment. Table 1 shows the target evaluation workloads. They are from the NVIDIA CUDA 9.0 Samples and Rodinia GPU benchmark suite [20]. Each application varies in its degree of parallelism, number of registers, and shared memory usages. For each application, we run evaluations on its original CUDA, MPS, Slate [2], smCompactor. Since slate is not an open-source framework, we implement it. All of the MPS, Slate, and smCompactor support GPU resource spatial sharing, while the original CUDA supports GPU resource time sharing. The reason we chose Rodinia and CUDA samples as our evaluation targets is that they have appropriate GPU memory consumption. Since currently GPU memory cannot be oversubscribed, appropriate GPU memory usage can avoid Out-of-Memory error in the multitasking environment. We can also consider using Unified Memory (UM), however, there are performance issues on the UM and we will take it as our future work.

4.2 Performance on Different Thread Block Counts on Different Number of SMs

In this section, we evaluate smCompactor by running a single workload with a different number of thread block counts on a different number of SMs. We choose FDTD3d and BlackScholes as the evaluation targets in this section since each of them represents a different resource usage pattern. As shown in Table 1, FDTD3d consumes a large number of registers and shared memory with a small number of thread blocks, while BlackScholes has low resource usage but a large number of thread blocks.

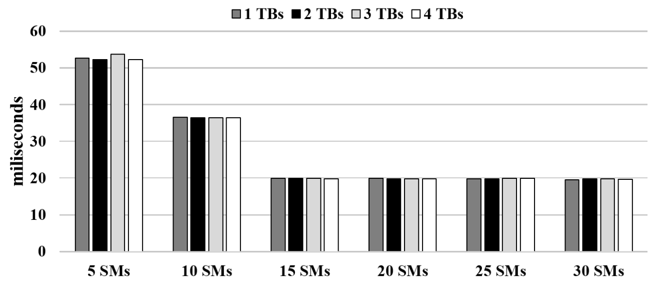


Figure 10: Execution time of FDTD3d with a different number of thread blocks (TBs) on different number of SMs.

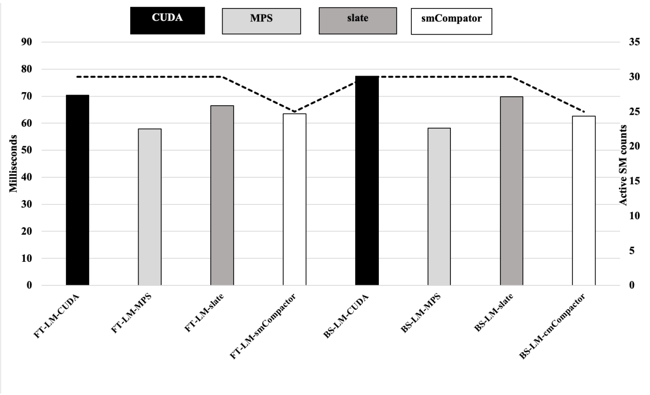


Figure 11: Execution time and active SM counts of running in different scenarios (Bar: kernel execution time; Curve: Number of activated SMs).

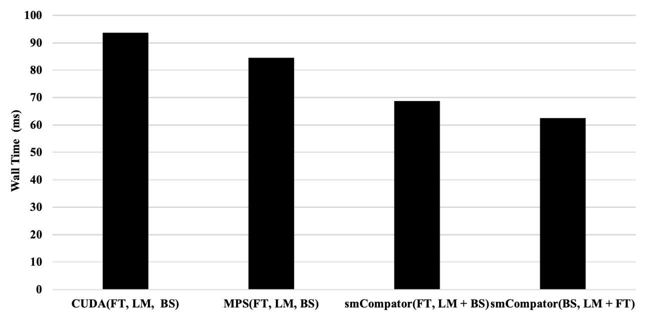
Figure 9 illustrates the kernel execution time of BlackScholes. From the figure, the execution time varies according to the change in the number of thread block and active SMs. The best performance (27.6 ms) can be achieved when 30 SMs were activated and launching eight TBs on each of them, and the second-best performance (28.02 ms) is achieved when 15 SMs were activated with 16 TBs on each of them. In this case, the kernel execution time is almost the same as the execution time on the original CUDA (27.31 ms). The performance can be almost saturated when only half of the whole SMs were activated with an appropriate number of thread blocks launched on each of them, leaving sufficient room for improvement in resource utilization.

Figure 10 shows the case of FDTD3d. The performance is also saturated when 15 SMs are activated. However, different from the case of BlackScholes, the performance variation is small when the number of thread blocks launched on each SM increased from one to four due to its large resource usage of each thread block. Because the thread blocks are actually scheduled by the hardware scheduler, which allows thread blocks to be executed only when the resource is available.

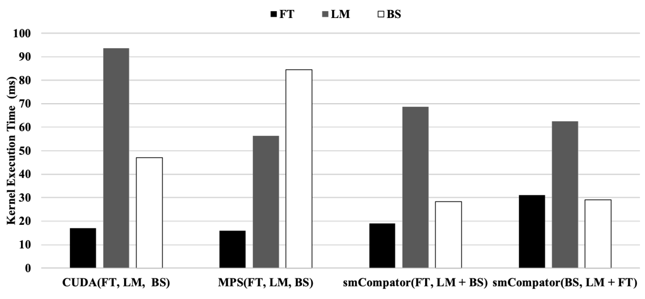
In summary, for either the workload that has a large or small resource usage, its optimal performance can be achieved without all SMs which are activated if an appropriate combination of the number of thread blocks and the active SM counts are found. However, compared with resource-intensive kernels, those small kernels have a higher possibility to find the optimal performance with fewer active SMs since their performance varies greatly when the number of thread blocks launched on each SM changes.

Workloads	#ThreadBlock	#Threads/Block	Registers/Block	Shared Memory (byte)
FDTD3d (FT)	288	512	57344	3840
nn (NN)	32768	256	4608	0
BlackScholes (BS)	2343750	128	2944	0
lavaMD (LM)	1000	128	7168	7200

Table 1: TARGET EVALUATION WORKLOADS



(a) Wall execution time



(b) The execution time of each workload

Figure 12: Execution time of co-locating three workloads with different strategies

4.3 Performance on Concurrent Kernel and Resource Sharing

In this section, we evaluate the performance of two co-launching kernels with different strategies. We select FDTD3d, BlackScholes, and lavaMD as our evaluate target since their execution times are similar to each other. The evaluation is conducted with the original CUDA, MPS, Slate, and smCompactor. We evaluated eight combinations among all the workloads. For the Slate and smCompactor, we evaluate all the combinations and present the best performance on the result.

Figure 11 demonstrates the kernel execution time and the number of active SMs in all eight cases. The time presented in the figure is the execution time of the last finished workload. The original CUDA provides the baseline for the comparison. As shown in Figure 11, MPS can efficiently schedule two concurrent kernels, outperforming the original CUDA version. In that case, the improvements can be 17% and 24% for the FT-LM and BS-LM combination, respectively. The slate also outperforms the original CUDA in both cases. However, it suffers performance downgrades of 14.8% and 22.5% compared with MPS in the cases of FT-LM, and BS-LM, respectively.

	SM 0 - SM 14	SM 15 - SM 19	SM 19 - SM 24	SM 25 - SM 29
FT-LM	FT:1,LM:1	FT:0,LM:8	FT:0,LM:8	FT:0,LM:0
LM-BS	BS:12,LM:4	BS:12,LM:4	BS:0,LM:4	BS:0,LM:0

Table 2: Number of thread blocks on each SM

	SM 0 - SM 14	SM 15 - SM 19	SM 19 - SM 24	SM 25 - SM 29
FT-LM+BS	FT:1,LM:1	FT:0,LM:2,BS:12	FT:0 LM:2,BS:12	FT:0,LM:0,BS:16
LM-BS+FT	BS:12,LM:4	BS:12,LM:4	BS:0,LM:1,FT:1	BS:0,LM:1,FT:1

Table 3: Number of thread blocks on each SM

smCompactor also outperforms the baseline in both cases. Meanwhile, it can enhance the performance against Slate by 10% and 16% when concurrently running FT-LM and BS-LM, respectively. It should be noted that resource utilization can be increased when the workloads are running with smCompactor. Even though the execution time of the FT-SM and BS-LM combination is slightly longer than that of the MPS cases, it only uses 83% and 66% of the whole SMs to achieve this performance, saving the SMs for the upcoming workloads. The number of thread blocks launched for each SM is shown in Table 2.

To demonstrate that those resources that saved by smCompactor can also be used to execute other kernels without performance degradation, we launch a third workload on those unused SMs for both the FT-LM and BS-LM cases. Particularly, we additionally launch thread blocks of BlackScholes (BS) mainly on the remaining five SMs for the FT-LM case, and one thread blocks of FT on the remaining 10 SMs for the BS-LM cases. Table 3 depicts the details of the thread block distribution on every SM. Compared to the previous experiment configuration as shown in Table 2, we slightly tune up the thread block number of LM to achieve an overall better performance.

Figure 12(a) demonstrates the wall time of executing all three workloads, which is the kernel execution time of the last finished workload. In the case of co-launching BS on the SMs remained by FT and LM with our proposed smCompactor, the execution time can be decreased with 26% and 18% compared to CUDA and MPS, respectively. The performance gains can be even larger in the case of co-launching FT on the SMs remained by BS and SM, which are 33% and 26% compared to the CUDA and MPS.

We analyze the results by presenting the kernel execution time of each kernel in different cases as shown in Figure 12(b). In the CUDA case, since the kernels are scheduled in a time-sharing manner, kernels are executed sequentially, leading to the longest wall time. In the MPS case, BS is blocked by FT and LM, leading to a longer execution time (84.5 ms) compared to its solo run case (27.1 ms). Compared to the result of MPS shown in Figure 11, we can tell that as more kernels are launched in parallel with the MPS, the possibility that one or some of them be blocked increases.

On the other hand, the wall time on smCompactor, which depends on the execution time of LM, is slightly increased in the case of smCompactor (FT, LM + BS) compared to the case of smCompactor (BS, LM + FT). This is because of the reduction of thread block counts of LM on several SMs, for providing more resources to the BS. However, the wall time of smCompactor in both cases still outperforms the case of CUDA and MPS, since the kernels can be executing in parallel without blocking.

4.4 Overhead of smCompactor

We finally evaluate the overhead of the proposed system by executing a single application with the original CUDA, MPS, Slate, and smCompactor. We measure the kernel execution time instead of the whole application execution time due to that for some applications, the part running on the host side costs thousands of times more than the kernel execution, which interferes with the accuracy of the evaluation. Besides, to make an appropriate comparison, we configure both the Slate and smCompactor to use all the GPU resources.

The result shows smCompactor has up to 7% overhead compared to the original CUDA case, which is similar to Slate. The overhead is due to the persistent thread model, where the user kernels nested in the dispatcher kernel (persistent thread), and both smCompactor and Slate adopt this concept. However, the case of NN is an exception, where the overhead is about 1.75 times than the baseline. This is due to NN's extremely small kernel execution time. Compared to the kernel execution time of tens of millisecond for other workloads, the execution time of NN is only 0.33 milliseconds, which increases the overhead proportion. As a result, our proposed smCompactor has a tiny impact on those kernels with relatively longer kernel execution time; however, with small kernels, the impact could be notable.

5 RELATED WORK

The research of GPU multitasking was conducted from various perspectives to better utilize GPU resources: from bottom hardware-based implementation support to top software-based support. In terms of the hardware-level approaches, there are attempts to apply the CUDA stream, Hyper-Q, and MPS using simulations, and models were suggested to support kernel preemption and scheduling [17–19, 22]. Park et al. proposed a preemption-based approach [18] to control the overhead of multitasking on the GPU. This approach is based on the flush operation that can preempt the SM with a new kernel. However, preemption can only occur when the thread blocks are at an idempotent state, which limits the functionality. They also proposed a dynamic resource management strategy [19] for efficient utilization of multitasking GPU; it uses SM as its scheduling unit and implemented with a simulator. However, since the functionality needed to implement these strategies is not provided by the real-world GPU, these studies are implemented with a simulator, which may have different features compared the real-world GPUs. Xu et al. proposed [24], a dynamic intra-SM slicing strategy to maximize the performance of concurrent kernels running. This strategy uses an analytical method for calculating resource partitioning across different kernels and assigns the thread blocks of each kernel to the target SM.

On the other hand, most of the software support studies are based on the persistent thread model [6, 8]. Bo et al. [23] first proposed the technology to circumvent the limitations of the hardware scheduler and to allow flexible program-level control scheduling. Slate [2] handles concurrent kernels from arbitrary applications at runtime and integrates workload-awareness into scheduling decisions, however, it focuses on avoiding interference between different kernels and was scheduled based on SM unit.

Our study is in line with these works [2, 23, 24] in terms of the investigation of the technique considering sharing GPU resources among multiple kernels. However, there are certain differences between our study and the previous studies. In [24], all the functionalities proposed are implemented via a hardware simulator, since the NVIDIA GPU does not provide the necessary technologies. In contrast, we implement our proposed smCompactor on the real-world NVIDIA GPU by using software technologies. Slate [2] focuses on avoiding interference by isolating each kernel on different SMs and [23] mainly focuses on improving the performance of a specific kernel by increasing the locality of each kernel. However, our proposed work aims to improve overall resource utilization by enabling as many workloads as possible to run concurrently while reducing the kernel execution time by aggregating thread blocks to exploit intra-SM resources.

6 CONCLUSIONS

Currently, GPU multitasking technology is not efficient enough considering the performance and resource utilization. The main issue is that the scheduling of thread blocks depends on the hardware scheduler which cannot detect the relation between performance and resource usage patterns. This makes it difficult to improve resource utilization while maintaining performance. In this paper, we proposed smCompactor, a fine-grained thread block scheduling framework, which can improve the resource utilization while reducing the kernel execution time in a multitasking environment by dispatching any number of thread blocks to specific SMs. The evaluation results demonstrate that our proposed smCompactor has minimal overheads. Moreover, near-optimal performance can be obtained with fewer SMs by managing thread blocks. For the multitasking cases, the performance gains against the original CUDA and MPS are up to 33% and 26%, respectively.

7 ACKNOWLEDGEMENTS

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2015M3C4A7065646, 2016M3C4A7952587). This work was also supported by the National Research Foundation of the Korea government (MSIT) (NRF-2018R1C1B5085640)

REFERENCES

[1] Andy Adinets. 2014. Dynamic Parallelism. Retrieved Nov 28, 2019 from https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/

[2] T. Allen, X. Feng, and R. Ge. 2019. Slate: Enabling Workload-Aware Efficient Multiprocessing for Modern GPGPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 252–261. https://doi.org/10.1109/IPDPS.2019.00035

[3] Amazon AWS 2017. https://aws.amazon.com/cn/nvidia/

[4] Mohammad Beheshti Roui, S. Kazem Shekofteh, Hamid Noori, and Ahad Harati. 2020. Efficient scheduling of streams on GPGPUs. *Journal of Supercomputing* (February 2020).

[5] Google Cloud 2017. https://cloud.google.com/gpu

[6] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. *A study of persistent threads style GPU programming for GPGPU workloads.* IEEE.

[7] Arthur L Delcher Amitabh Varshney Michael C Schatz, Cole Trapnell. 2007. High-throughput sequence alignment using Graphics Processing Units.. In *BMC Bioinformaticsvolume 8, Article number: 474.*

[8] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 308–317.

[9] NVIDIA. 2009. NVIDIA Fermi Architecture. Retrieved Nov 28, 2019 from https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf

[10] NVIDIA. 2013. NVIDIA HyperQ. Retrieved Nov 28, 2019 from hhttp://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf

[11] NVIDIA. 2017. NVIDIA TITAN Xp. Retrieved Nov 28, 2019 from https://www.nvidia.com/en-us/titan/titan-xp/

[12] NVIDIA. 2019. CUDA context. Retrieved Nov 28, 2019 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#contex

[13] NVIDIA. 2019. CUDA stream. Retrieved Nov 28, 2019 from https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html

[14] NVIDIA. 2019. NVIDIA HyperQ. Retrieved Nov 28, 2019 from https://docs.nvidia.com/deploy/mps/index.html

[15] NVIDIA. 2019. NVIDIA NVCC. Retrieved Nov 28, 2019 from https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html

[16] NVIDIA. 2019. NVIDIA NVProf. Retrieved Nov 28, 2019 from https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[17] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 407–418.

[18] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 593–606.

[19] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. 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*. 527–540.

[20] et al S.che. 2018. Rodinia. Retrieved Nov 28, 2019 from https://rodinia.cs.virginia.edu/doku.php

[21] Summit 2018. https://www.olcf.ornl.gov/summit/

[22] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 193–204.

[23] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 119–130.

[24] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. 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)*. IEEE, 230–242.

[25] Qiu Z. Fan, F. 2004. GPU Cluster for High Perforamnce Computing. In *2004 IEEE/ACM conference on Supercomputing 2004*. ACM, 47.