creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. DISSERTATION

Optimizing GPU System for Efficient Resource Utilization of General Purpose GPU Applications in a Multitasking Environment

멀티 태스킹 환경에서 GPU를 사용한 범용적 계산 응용의 효율적인 시스템 자원 활용을 위한 GPU 시스템 최적화

August 2020

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Qichen Chen

Ph.D. DISSERTATION

Optimizing GPU System for Efficient Resource Utilization of General Purpose GPU Applications in a Multitasking Environment

멀티 태스킹 환경에서 GPU를 사용한 범용적 계산 응용의 효율적인 시스템 자원 활용을 위한 GPU 시스템 최적화

August 2020

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Qichen Chen

Optimizing GPU System for Efficient Resource Utilization of General Purpose GPU Applications in a Multitasking Environment

멀티 태스킹 환경에서 GPU를 사용한 범용적 계산 응용의 효율적인 시스템 자원 활용을 위한 GPU 시스템 최적화

지도교수 염헌영

이 논문을 공학박사 학위논문으로 제출함

2020 년 7 월

서울대학교 대학원

전기·컴퓨터 공학부

진계신

진계신의 공학박사 학위논문을 인준함

2020 년 6 월

위 원 장	엄 현 상	(인)
부위원장	염 헌 영	(인)
위 원	김 진 수	(인)
위 원	이 재 욱	(인)
위 원	손 용 석	(인)

Abstract

Recently, General Purpose GPU (GPGPU) applications are playing key roles in many different research fields, such as high-performance computing (HPC) and deep learning (DL). The common feature exists in these applications is that all of them require massive computation power, which follows the high parallelism characteristics of the graphics processing unit (GPU). However, because of the resource usage pattern of each GPGPU application varies, a single application cannot fully exploit the GPU system's resources to achieve the best performance of the GPU since the GPU system is designed to provide system-level fairness to all applications instead of optimizing for a specific type. GPU multitasking can address the issue by co-locating multiple kernels with diverse resource usage patterns to share the GPU resource in parallel. However, the current GPU multitasking scheme focuses just on co-launching the kernels rather than making them execute more efficiently. Besides, the current GPU multitasking scheme is not open-sourced, which makes it more difficult to be optimized, since the GPGPU applications and the GPU system are unaware of the feature of each other. In this dissertation, we claim that using the support from framework between the GPU system and the GPGPU applications without modifying the application can yield better performance. We design and implement the framework while addressing two issues in GPGPU applications. First, we introduce a GPU memory checkpointing approach between the host memory and the device memory to address the problem that GPU memory cannot be over-subscripted in a multitasking environment. Second, we present a fine-grained GPU kernel management scheme to avoid the GPU resource under-utilization problem in a

multitasking environment. We implement and evaluate our schemes on a real GPU systems. The experimental results show that our proposed approaches can solve the problems related to GPGPU applications than the existing approaches while delivering better performance.

Keywords: GPU System, Multitasking, Memory Management, Checkpointing, GPU Resource Utilization

Student Number: 2011-24087

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphics processing unit (GPU) is originally designed for efficiently manipulating computer graphics and image processing, which are mostly depending on fast geometry computing powers. Rather than general computing operations, geometry computing prefers more parallel performance, thus leading to the current GPU architecture that within extremely high parallelism. Besides, its high parallelism also can be applied to general purpose computing, thus the concept of general purpose computing on graphics processing unit (GPGPU) has been put forward. Recently, GPGPU has been widely adopted in several fields where exascale computational power [1] is needed. Among those domains that GPU has been widely used, high-performance computing (HPC) and deep learning (DL) are the ones that draw the most attention. Besides, within the increase of problem complexity and input scale for HPC and DL applications, GPUs are widely used in large-scale computing environments, such as cloud computing environment [2–4], supercomputers, and clusters. As introduced in the top 500 list released in November 2019 [5], the world's most powerful supercomputer

Summit [6] also uses the CPU/GPU cluster structure.

However, according to [7], *Summit* can only achieve 65% of designed performance when running GPU-friendly benchmarks. The performance could even be much worse when executing general applications. For example, *Summit* can only achieve 1.5% of the designed performance when running high performance conjugate gradients (HPCG) benchmark [8]. Therefore, efficient utilization of GPU resources becomes more important than ever, especially current GPGPU applications can not exploit GPU resources. Our goal is to optimize the GPU system to allow multiple GPGPU applications to efficiently utilize GPU resources while achieving the best performance.

1.1 Motivation

GPGPU workloads are designed to monopolize the whole GPU, thus the intra-GPU resources, such as the registers, shared memory, stream processors, and device memory bandwidth are usually under-utilized in such an environment [9–18]. One solution to improve resource utilization is to enable GPU multitasking, where multiple GPGPU workloads with different resource utilization patterns can share the GPU resources. However, there are several problems when GPU multitasking is enabled. We address the following problems in this dissertation, which are all related to GPU resource utilization under a multitasking environment.

GPU Memory Cannot be Over-subscripted To handle HPC and DL problems, tens of thousands of GPUs are deployed in clusters. However, the state-of-the-art cluster management systems, such as Kubernetes [19], Slurm [20], and Torque [21] are incapable of scheduling various applications to share a single GPU. Particularly, applications must acquire an exclusive lock before their

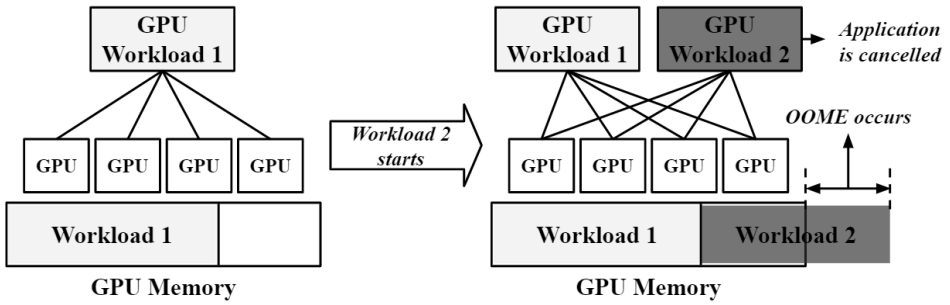


Figure 1.1 Out-of-Memory error when multiple applications share the GPU.

execution to ensure safety, which results in a decrement of GPU utilization. One reason that most cluster management systems cannot handle multiple concurrent executions on the same GPU is that GPU memory oversubscription can cause applications to unexpectedly fail.

Unlike CPU memory that can be swapped into the disk, GPU memory is a type of resource that cannot be oversubscribed, if the instant GPU memory requirement of all the GPU workloads exceeds the physical capacity, an out-of-memory(OOM) error will occur and cause the application to fail. In addition, the GPU device driver will not track the GPU memory usage of each workload, making OOM occurs easier under a multitasking environment. Figure 1.1 shows an example of how OOM error occurs when multiple GPU workloads execute in parallel. GPU memory request of workload 2 exceeds the physical can cause itself to be terminated with a failure.

One key observation regarding HPC workloads is that they do not always utilize the GPU resources fully during its execution time. Figure 1.2 demonstrates the GPU utilization and GPU memory usage of MUMmerGPU [22][1]

[1]MUMmerGPU is an open-source high-throughput parallel pair-wise local sequence align-

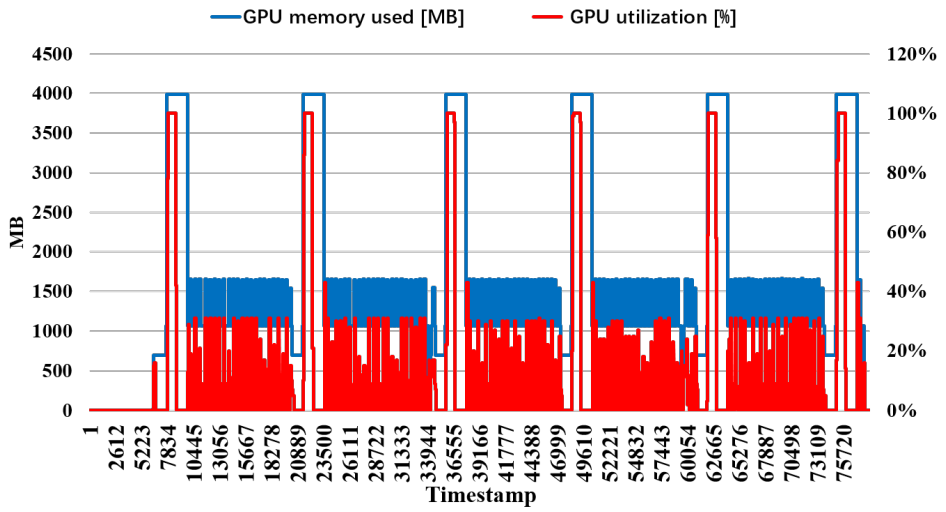


Figure 1.2 GPU & GPU memory utilization of MUMmergpu.

during its execution time, we track the variation of GPU memory usage and GPU utilization every 100 millisecond. From Figure 1.2, we have noticed that both GPU utilization and GPU memory usage are not always maintained at a high level during its lifetime. Instead, it periodically spikes rapidly and then falls down to reach a steady level for a relatively long period in this procedure.

We have also observed that HPC workloads are not originally designed for sharing the GPU with other workloads. For example, GPU memory will not be deallocated until the entire application is finished, even if the CPU and GPU computations are interleaved. Figure 1.2 also reveals this fact by depicting that a certain amount of GPU memory is always being occupied even though the kernels are not being executed. This is reasonable when the workload is executed on the GPU alone, because maintaining data in GPU memory can avoid freeing, allocating, and copying data between the host memory and the device memory,

ment program that runs on commodity GPUs

which will introduce overhead and downgrade the performance. However, when the GPU is shared among multiple workloads, as the GPU memory is a resource that cannot be oversubscribed, always maintaining the data in the GPU memory will result in fewer workloads sharing the GPU, leading to low GPU utilization. Currently, GPU cluster management systems do not schedule the applications according to runtime GPU memory utilization; Instead, each application is executed with a pre-defined maximum GPU memory requirement value.

To handle these issues, previous studies [23–26] investigates the solution of sharing a physical GPU among multiple applications by creating a copy of CUDA application programming interfaces (APIs) to virtualize the GPU and provide them to each application. [27] introduced a solution for using remotely located GPUs in the virtualization system. However, the performance may be downgraded owing to the overhead. In addition, [26] introduce a new solution that captures every GPU memory resource allocation/deallocation call information from each application and schedules these applications based on this information. Basically, it executes the maximum possible number of GPU memory allocation calls, which can efficiently solve the GPU memory oversubscription problem, unless the total allocation amount exceeds the physical capacity. However, it has a potential deadlock issue when GPU memory allocation/deallocation occurs frequently. In this dissertation, we claim that GPU memory oversubscription problem can be handled by a checkpoint based mechanism efficiently, it also can solve the deadlock problem remained by the previous research.

GPU Computational Resource Under-Utilization

There rarely exists a GPU kernel that can use all the GPU resources at a high level during its whole execution time [28], while currently most of the GPU workloads are designed to monopolize the GPU. NVIDIA provides Hyper-Q [29]

and MPS [30] technology to enable GPU multitasking, however, MPS cannot exploit GPU resource efficiently either, since it is still unaware of the relation between resource usage pattern and the performance of each kernel. Apart from this, kernels run with MPS are not preempted-able, which makes small kernels have to wait for large ones to finish if they are launched later, downgrading the quality of service (QoS). To handle these issues, previous studies [31–33] proposed both hardware and software based strategies to improve the performance when multiple kernels run in parallel with or without MPS. Among those, slate [34] introduced a workload-aware kernel based scheduling system to improve performance when multiple workloads share the GPU. It efficiently avoids the interference caused by co-locating kernels by isolating them into separate SMs. However, improving the GPU resource usage by supporting as many kernels running in parallel as possible while maintaining the performance is not taken into consideration.

Besides, we observed the fact that the best performance of GPU workloads may be achieved even when only a part of the SMs are used, thus monopolizing the whole GPU can lead to a waste of resources. We explore this issue by evaluating the kernel execution time variation as the number of thread blocks launched per SM and the number of active SMs changes on an NVIDIA Titan Xp GPU [35]. The target workloads are BlackScholes (BS), lavaMD (LM), and FDTD3d (FT), which belongs to NVIDIA CUDA samples and Rodinia benchmark [36] respectively. Figure 1.3 shows the details that are compared to use all 30 SMs, both BlackScholes and FDTD3d achieve their best performance where 15 SMs are activated with up to 12 thread blocks and 1 thread block per SM, respectively; Meanwhile, lavaMD also can achieve 95% of its best performance when 20 SMs are activated with up to 8 thread blocks launched on each of them. In this dissertation, we claim that GPU computational resources under-

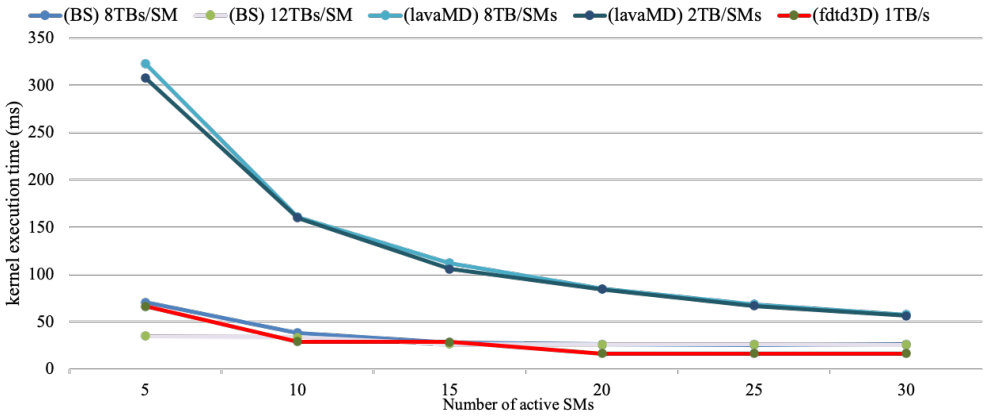


Figure 1.3 Kernel execution time varies with launched thread block per SM and active SM.

utilization issues in the multitask environment can be solved by designating a specific number of thread blocks to particular SMs, meanwhile the performance can be improved.

1.2 Contribution

The contributions are summarized as follows:

- We present a new GPU multitasking framework to show that GPU utilization can be improved by tailoring the scheduling framework to the features of GPU workload. It introduces GPU memory checkpoint techniques to migrate the temporarily irrelevant GPU memory contents to the host memory to create space for more kernels running at the same time. It also solves the GPU memory allocation deadlock problem by introducing a temporary checkpoint for the occupied GPU memory.

- We introduce a new GPU multiprocessing framework that shows GPU utilization can be improved by tailoring the scheduling framework to the intra-SM resource consumption features of the GPU workloads. In

particular, our work focuses on the interference and complementarity of intra-SM resource usage of each kernel. Our proposed mechanism aims at making full use of all computational resources, including the shared memory, registers, and stream processors, to improve the system throughput by supporting as many workloads as possible that can share the GPU at the same time. Meanwhile, the performance of each workload will be maintained at a reasonable level

- We implement and evaluate both of the framework prototypes on a real NVIDIA Titan Xp-based system. The evaluation result shows that our proposed framework can efficiently solve GPU memory oversubscription problems. It also improves the GPU computational resource utilization while improving performance compared to state-of-the-art technologies.

1.3 Outline

This dissertation is structured as follows:

- **Chapter 2** covers the background about GPU memory management system, GPU computational resource management system and system support GPU multitasking to improve the resource utilization.

- **Chapter 3** introduces FlexGPU, our checkpoint-based scheme that solves GPU memory over-subscription problem under the multitask environment. We first explain the problems of existing GPU memory management mechanisms under multitask environment. Then we describe the details of design and implementation of our proposed new scheme and evaluate it on the real GPU system with GPGPU benchmark and real-world workload.

- **Chapter 4** introduces smCompactor, our thread block based fine-grained GPU multitasking scheme. We start by explaining the problems of GPU

computational resources under-utilization under existing multitask environments. Then we propose the details of our scheme. Finally, we evaluate our scheme on a real GPU system with GPGPU workloads.

- **Chapter 5**summarizes and concludes the dissertation. It also points out the directions for future work.

Chapter 2

Background

Our approaches heavily rely on many functions support by the NVIDIA GPU device driver. In this chapter, we explain the background of GPU system and the functionality that supported by the device driver for efficient resource utilization to help understand the rest of the dissertation.

2.1 Graphics Processing Unit (GPU) and CUDA

Initially, the GPU was developed for graphics work, such as 3D rendering, which is required for parallel computing [37] so that the GPU can achieve parallel execution. However, as the investigations from several studies demonstrated the general-purpose utilization of GPUs, it has been optimized to accelerate various HPC applications, such as deep learning (DL) and big data analysis in cloud environments.

NVIDIA CUDA [38] was released in February 2007, which is an extensively used parallel computing platform, which provides APIs for GPGPUs. CUDA extents ANSI C language and provides APIs for easy using. It allows us to access

the parallel computation elements of the GPU directly. There are two kinds of programming interferes, which are Driver API and Runtime API. The Runtime API is a high-level API that is implemented on top of the low-level Driver API. The Runtime API is easy to use because it does not need explicit initialization and management. However, The Driver API can perform better performance since it supports fine-grained CUDA context control and module loading. In CUDA, the *host* is the CPU and its memory and the *device* is the GPU and its memory. The code running on the host can manage the memory on both the host and the device and launches *kernels*, which are functions executed on the device. These kernels are executed by several GPU threads in parallel [38]. GPU utilization can be measured using the NVIDIA system management interface and nvprof [39], which are both profiling tools provided by NVIDIA.

2.2 Checkpoint and Restart

Checkpoint and restart [40] is a classic strategy that works when the contents are permanently lost. At certain firms and super computing centers, it is a common practice to break up long-running computational programs into several batches. When these long-running programs are intentionally stopped after a checkpoint, it can restart from the previous checkpoint. Traditional checkpoint and restart methods save the checkpoints in underlying storage systems. However, in our research, we apply the checkpoint and restart theory in our design such that checkpoint saves the contents of applications running in the GPU in the host memory and restarts them when it is necessary.

2.3 Resource Sharing Model

There are two resource sharing models when using the NVIDIA GPU, the time-sharing and spatial-sharing model. Originally, multiple kernels that run on the

same GPU but without different CUDA streams are scheduled using the time-sharing model. In the time-sharing model, time budgets are assigned to each GPU kernel. Each kernel can utilize GPU resources only when both of its time budget and required resources are available. After its time budget expired, other kernels will be 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 same GPU are scheduled depending only on their resource usage, rather than their time budgets. In this way, multiple independent kernels can be simultaneously executed on different SMs and SPs, as long as their resource requirements are satisfied.

2.4 CUDA Context

The CUDA context [41] holds all the management data to control and use the device, such as list of allocated memory, loaded modules that contain devices code, etc. There are two ways to handle CUDA context during kernel execution, which are described in detail as follows.

Context Switching

Each host instance will create its own CUDA context when accessing a GPU device. All resources and actions performed within the CUDA driver API are encapsulated inside this CUDA context. When the warp scheduler selects the running threads from one warp to another, it leads to a context switch. GPU architectures use context switch to hide the memory latency between the warps. However, since contexts are private to each control host, it is impossible to share resources among different contexts even though they belong to the same device.

Context Funneling

Unlike context switching, context funneling is implemented as a client-server structure. The server thread creates a GPU context and shares it with other client threads. Specifically, NVIDIA MPS is an implementation of context funneling. In this case, since only one context exists, per-context objects, which include device memory objects, now can be shared among different application threads. Meanwhile, kernels originally launched from different host threads can be executed in parallel. However, the performance gain of concurrent kernel execution still varies depending on the features of resource consumption of different kernels.

2.5 GPU Thread Block Scheduling

On existed NVIDIA Fermi [42] and Kepler [43] GPUs, a GigaThread Engine is responsible for thread block scheduling. Since there is very little public information available about the detail of thread block scheduling, [28] reveals the details through sophisticated experiments. It uses a round-robin thread block scheduling where the thread blocks are assigned to each SM in a round-robin manner and assign the maximum number of thread blocks. The maximum number depends on resource usage of each workload, including the amount of registers, shared memory, etc. Once a thread block finishes its execution, the hardware thread block scheduler assigns another thread block to that particular SM, until all thread blocks have been assigned.

2.6 Multi-Process Service with Hyper-Q

The NVIDIA MPS [30] is a binary-compatible client-server runtime implementation of the CUDA API. It is designed to enable cooperative multi-process

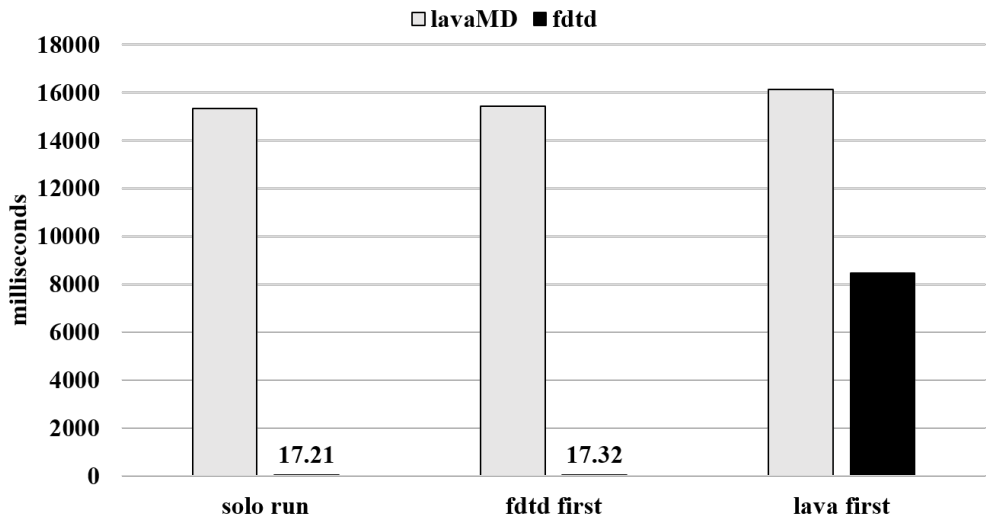


Figure 2.1 Performance varies when launch sequence changes.

CUDA applications in a concurrent manner using Hyper-Q. Starting from Fermi architecture [42], Hyper-Q enables multiple CPU threads or processes to launch work on a single GPU simultaneously, therefore allowing the connections for both CUDA streams [44] and MPI processes. Hyper-Q allows CUDA kernels to be processed concurrently on the same GPU, which can improve performance when GPU resources are underutilized.

While the concurrent scheduling of Hyper-Q is limited to a single CUDA context, MPS with Hyper-Q could collect the contexts from different applications and map them into a single one. In the Pascal architecture [45], MPS works in a client-server architecture. The control daemon of MPS acts as the server that 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 together.

However, since the design concept of MPS is merging CUDA contexts from different applications into a single one, to exploit concurrent scheduling features of Hyper-Q; the relation between performance and resource usage pattern of each kernel is not taken into consideration, let alone with resource utilization. In addition, MPS does not support dynamic parallelism [46], which is currently widely used due to its additional parallelism that can be exposed to GPU scheduler and load balancer. Besides, MPS client processes may allocate memory from different partitions of the same GPU virtual address space, causing an out-of-range write or triggering an error [30]. It may even block the later launched kernel until the previously launched one finishes its execution, decreasing the QoS and resource utilization severely if the previous kernel is not resource intensive with a long execution time. We designed an experiment to verify this property by staggering the launch time of two workloads with a huge difference in their kernel execution time. The workloads we chose in this experiment are FDTD3d and lavaMD, which are from the CUDA sample and the Rodinia benchmark [36] respectively. Figure 2.1 demonstrates the details of this experiment. When FDTD3d is launched before lavaMD, both of their kernel execution time is similar to the solo run cases; however, when lavaMD is launched first, the execution time of FDTD3d extends to 8000 milliseconds, which is 460 times longer than the solo run case, since MPS does not start FDTD3d immediately while lavaMD is running.

Chapter 3

Checkpoint based solution for GPU memory oversubscription problem

3.1 Motivation

Currently, HPC workloads are designed to monopolize GPU resources to achieve good performance. However, according to our profiling result which is presented in Figure 1.2, the GPU resources are not fully used during the workloads' lifetime, meanwhile, the GPU memory management strategy of each workload is not suit for the multitasking environment. Previous study addressed the GPU memory oversubscription issues can be solved by making the GPU memory allocation request wait until the physical capacity becomes available, however we observed that it will cause a deadlock situation. Figure 3.1 demonstrates how GPU memory allocation deadlock appears on a NVIDIA TITIAN Xp GPU when four instances of MUMmerGPU were executed simultaneously by following the scheduling mechanism proposed in [26], we track the GPU memory

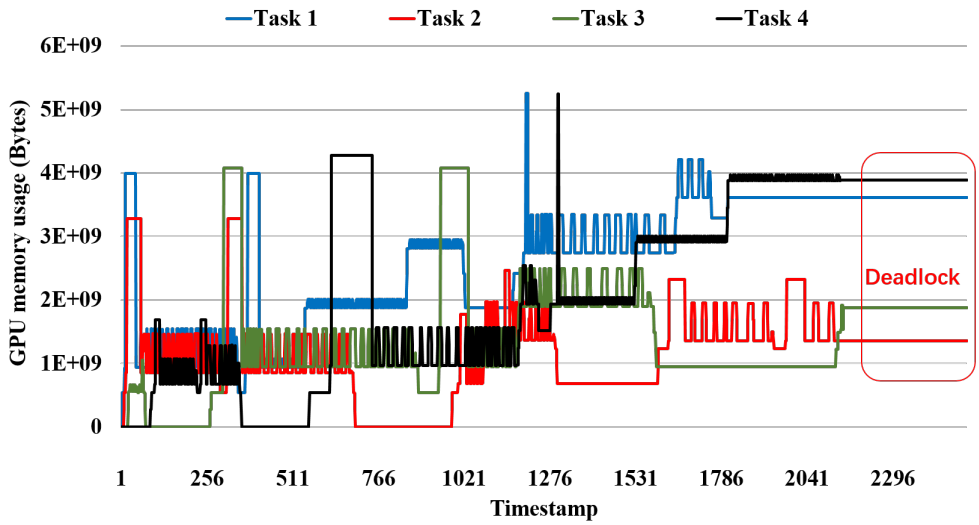


Figure 3.1 GPU memory allocation deadlock and breakdown on 4 MUMmergpu instances

usage every 100 milliseconds. The performance remains the same when up to a maximum of three instances are executed simultaneously; However, a deadlock occurs when four instances are executed in parallel. From Figure 3.1, we can observe that at the final phase of the execution, each instance falls into a situation where their memory usages become stagnant.

We have clearly illustrated this deadlock situation in Figure 3.2. At the final phase, instances one to four occupied 3.61 GB, 1.36 GB, 1.88 GB, and 3.88 GB, respectively. As the GPU we used contains a total memory of 12 GB, the free space at that moment is 1.27 GB. Then, these instances request a GPU memory allocation of 3.36 GB, 3.09 GB, 3.51 GB, and 2.51 GB, respectively. However, as the free space cannot be allocated to any of the instances and each instance requires the requested GPU memory space to continue its execution, the system falls into a deadlock state.

Inspired by this observation, we propose FlexGPU, which can rearrange the

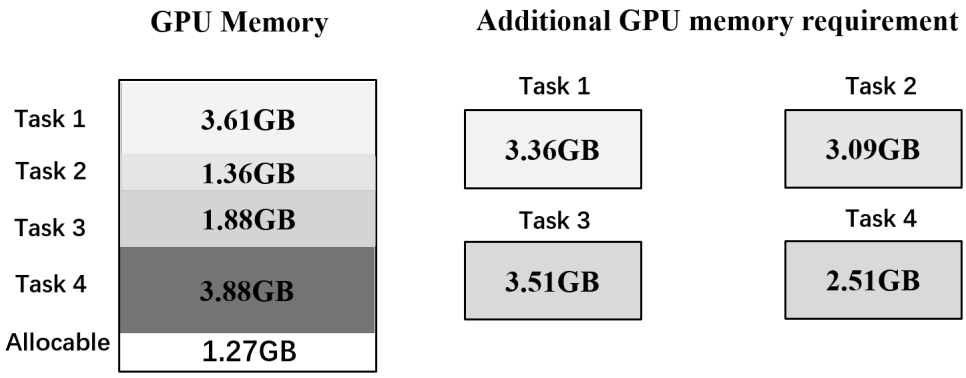


Figure 3.2 Breaking down of the deadlock situation

issue time of each kernel as well as its related GPU memory allocation according to their computational requirements.

3.2 Related Work

GPU scheduling and GPU resource sharing under a multitasking environment recently has been researched in several works.

GPU scheduling: GPU scheduling techniques can usually classified into 1) concepts within hardware modification that implemented with simulation. 2) concepts that using verified GPU/CUDA features to schedule the execution of application within software middle-ware.

Hardware modification can enable various of scheduling policies in the low level. [9] proposed a solution to enable concurrent CUDA application running by using persistent kernel method to re-shape the CUDA grids. [32] and [47] also evaluated the persistent kernel methods by using the simulation, since some of them needs kernel preemption, which is still not available in current GPU model.

On the other hand, [48] provided a scheduling mechanism for loading con-

current applications on to the nodes of the cluster. The throughput was improved; however, the intra-node scheduling of jobs was not considered. [49, 50] provide a load-balancing-based finer-grained job scheduling in multiple GPUs-based environment. However, they only focused on the uniform distribution of the workload to the CPU instead of improving GPU utilization. Slate [34] presented a solution that scheduled the GPU kernel according to their features. Our work is in line with this method; however, we also considered the GPU memory oversubscription issue, which was not included in [34]

GPU resource sharing: Park et al [51] proposed a solution to dynamically allocate the computational resources in a GPU and evaluated with simulation. [52] proposed a solution to maintain the latency for latency sensitive kernels when launching with batched kernels by ensuring the computational resource for latency sensitive kernels with higher priority. ConVGPU [26] is a solution designed for sharing GPU memory among Docker containers and GaiaGPU [23] is a solution that considers GPU computing resources as well as memory. Our study is in line with these works in terms of the investigation of the technique considering GPU resources in a GPU-virtualized environment. The previous studies focused on sharing GPU resources on the environment; however, there are certain differences between our study and the previous studies. Firstly, in ConVGPU [26], when the scheduler receives a request that exceeds the GPU memory capacity from the container, it denies the request or pauses the container. In contrast, we handle the same request in a more flexible way with checkpointing and ensure better performance in terms of overall throughput and response time. We also prevent the potential GPU memory allocation deadlock situation, which can occur in ConVGPU [26]. GaiaGPU [23] is based on the device plugins in existing resource management software; however, we have implemented our solution without any software installation. Further, the

target applications of [23] are neural network algorithm workloads (e.g., Modified National Institute of Standards and Technology database, Alexnet) on a DL framework. Conversely, we evaluate our solution with workloads on other HPC frameworks.

3.3 Design and Implementation

There are two goals in our design of FlexGPU. First, FlexGPU supports efficient CUDA kernel level multitasking and resource sharing among multiple workloads. Second, FlexGPU can track the GPU memory requirements dynamically and solve the GPU memory oversubscription problem among several multiple applications; then, the kernels can be executed simultaneously. Our approach is based on a client-server structure, where the communication is implemented by using a UNIX socket. The *CUDA API wrapping module* captures each GPU memory allocation call and kernel launch call from the application, while a *scheduler module* acts as a server to decide the scheduling result. The wrapping module is also responsible for reclaiming the GPU memory temporarily by copying the corresponding content to the host memory and restoring them when it is necessary.

Meanwhile, the scheduler continues to monitor the GPU memory usage and the kernels being executed while accepting the memory allocation, deallocation, and kernel launch information from each application. It decides when the kernel should be issued and the related GPU memory should be allocated according to the profiling result. In addition, when the kernel requires more GPU memory, which is beyond the physical capacity, the scheduler also can decide whether to postpone the issue or to trigger the GPU memory checkpointing. In the rest of this section, we present the details of the design and implementation of each module.

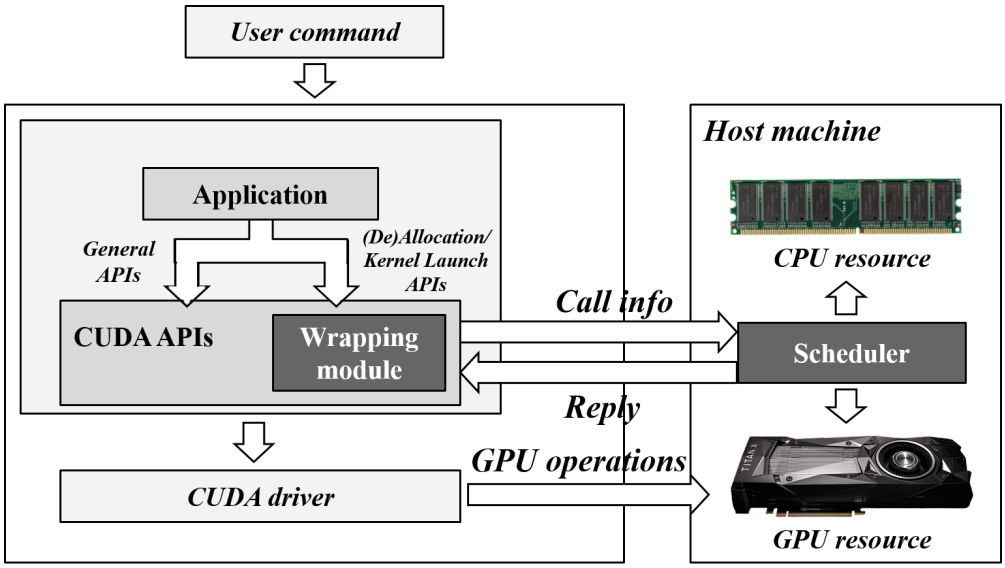


Figure 3.3 Overall system design.

3.3.1 System Design

The overall system architecture is as described in Figure 3.3. The wrapping module is loaded as an LD_PRELOAD [53] library when the application starts its execution. It partially intercepts the CUDA GPU memory allocation and kernel launch calls. After the calls are intercepted, it analyzes them and extracts the corresponding information then forwarding them to the scheduler. Meanwhile, the intercepted calls will be on hold until receiving a reply from the scheduler.

The scheduler continues to track the information of all the running kernels and the kernels that must be launched, and decides whether a kernel can be issued or which part of the allocated memory should be checkpointed to the temporary memory for making space for other kernels.

When the wrapping module receives the reply, it continues the operations of either allocating the necessary memory and launching a kernel or checkpointing

some of its GPU contents directly through the original CUDA driver APIs to the GPU.

3.3.2 CUDA API wrapping module

An HPC application uses the CUDA runtime and driver APIs to communicate and control the NVIDIA GPU. These APIs achieve multiple functionalities from the launch of CUDA kernels to the allocation of resources in the GPU and the transfer of data between the device and the host. Among these APIs, *cuMemAlloc*, *cudaMalloc*, *cudaFree*, and *cuMemFree* are used to allocate device memory, and *cuMemcpyDtoH*, *cuMemcpyHtoD*, and *cudaMemcpy* are used to transfer the data between the CPU memory and the GPU memory; further, *cudaLaunch* is used to launch a CUDA kernel function. As the CUDA API is not an open source, we cannot modify the function itself. Instead, we develop the CUDA API wrapping module to capture the CUDA API calls and replace them with our own implementation. This CUDA API wrapping module intercepts a part of the CUDA API calls, which are mostly GPU memory allocation, deallocation and kernel launch calls. To make sure all the kernels from different application run in parallel, the wrapping module also issues the kernels with different CUDA streams [44]. Since the NVIDIA GPU with Hyper-Q can only schedule streams in the same CUDA context concurrently, we also merge the contexts from different application into one context in the FlexGPU runtime.

The basic concept of implementing such an interception module is the process of adding the module name into the *LD_PRELOAD* [53] environment variable. *LD_PRELOAD* is a list of additional, user-specified, executable and linkable format-based shared objects, which are loaded before all other objects. This feature can be used to selectively override functions in other shared objects and the objects are searched for and added to the link map according to

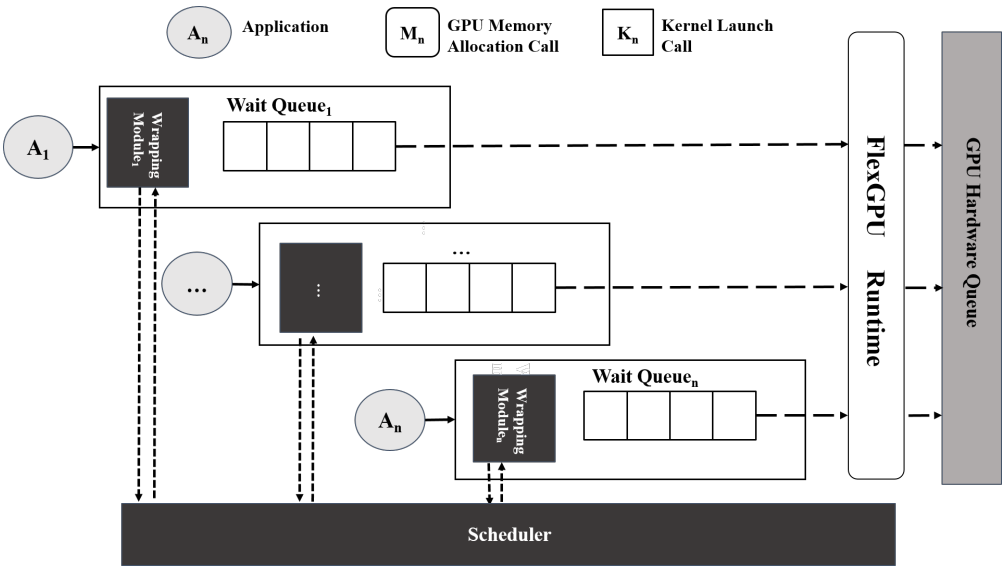
the left-to-right order specified in the list.

Figure 4.1(a) demonstrates the architecture of FlexGPU. Each application is bound to a specific wrapping module, which contains a wait queue inside. Each wrapping module sends related information to the scheduler and waits for a reply. Figure 4.1(b) shows the control flow of FlexGPU. When an application allocates a GPU memory and launches a kernel through the wrapping module, it intercepts both of them and encapsulates them into one wrapped kernel launch call. We group the GPU kernel launch and GPU memory allocation calls together because if we consider them separately, the preceding allocated memory and a delayed kernel launch will decrease the GPU utilization as other kernels may not be launched owing to the exceeding GPU memory physical capacity. There is a wait queue in each of the wrapping modules; the grouped kernels that have not been issued wait in the queue. It should be noted that even though interleaved execution can be performed on kernels from different applications, the launch sequence of kernels from the same application must be preserved; thus we design this queue to maintain the launch sequence.

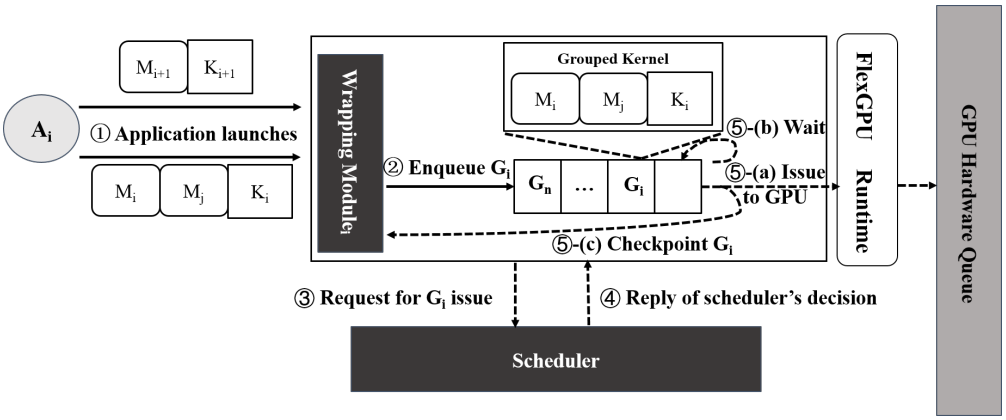
In detail, when the applications try to allocate GPU memory, launch a kernel or free GPU memory, the wrapping module automatically captures the allocating size, GPU device pointer, kernel name and its process id.

When the applications are attempting to allocate GPU memory, launch a kernel, or free the GPU memory (①), the wrapping module automatically captures the allocating size, GPU device pointer, kernel name, kernel parameters as well as its process ID, encapsulate them into a grouped kernel and pushes it into its wait queue (②). This information is then forwarded to the scheduler (③). Then the scheduler makes a decision according to the current state of the system and sends replies to each wrapping module of those decisions.

The replies from the scheduler indicate 3 kinds of cases, kernel launch ap-



(a) Overview of FlexGPU architecture



(b) Control flow of FlexGPU

Figure 3.4 System architecture and control flow of FlexGPU

proved, kernel launch denied, and checkpointing the previous allocated GPU memory by the wrapping module (④). In the first case, the wrapping module will allocate GPU memory and launch a kernel by using the original CUDA APIs (⑤-(a)). In the second case, grouped kernel will be on hold in the waiting queue for the next launch opportunity (⑤-(b)). In the third case, part of the previously allocated GPU memory would be check-pointed to make space for subsequent kernel launching. The reply is forwarded back to the wrapping module and makes the wrapping module start the checkpoint procedure (⑤-(c)).

Algorithm 3.1: Checkpoint Algorithm

Variable: $vector_{gpu}$ and $vector_{cpu}$ which type is $pair(start, size)$
Output: out

```
 1: function record_gpu_usage_info(start, size);
 2: pair = make_pair(start, size)
 3: vector_gpu.push(pair)
 4: end function

 5: function checkpoint_data(void);
 6: while (!vector_gpu.is_empty()) do
 7:     pair_gpu = vector_gpu.pop()
 8:     size = pair_gpu.size
 9:     start = cpu_mem_alloc(size)
10:     copy_data_gpu_to_cpu(start, size)
11:     pair_cpu = make_pair(start, size)
12:     vector_cpu.push(pair_cpu)
13:     gpu_mem_free(size)
14: end while
15: end function

16: function restore_checkpointed_data(void);
17: while (!vector_cpu.is_empty()) do
18:     pair = vector_cpu.pop()
19:     copy_data_cpu_to_gpu(pair.start, pair.size)
20: end while
21: end function
```

The wrapping module is also responsible for temporarily checkpointing previous allocated GPU memory contents into the host memory and emptying the space for the subsequent running kernels. Details of how checkpoint proce-

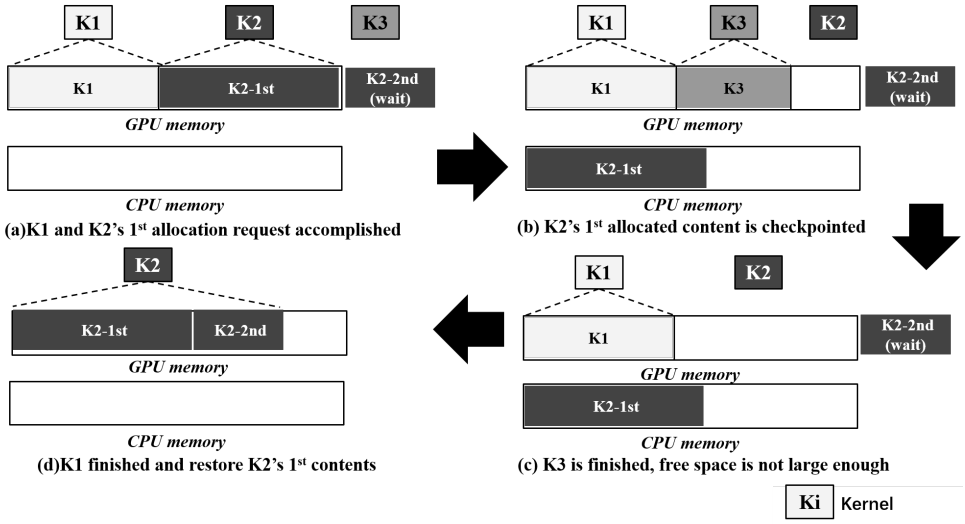


Figure 3.5 Example of GPU memory contents checkpoint

dure works are as follows. When it receives the checkpointing reply, it initially allocates a new memory address in the host memory; then, it transfers the corresponding contents stored in the GPU memory to the host memory by calling *cuMemcpyDtoH*. Meanwhile, to maintain consistency, we also store the information regarding the source and destination addresses as well as the process ID of the target kernel for each content movement individually. Finally, the wrapping module frees the occupied GPU memory of certain applications by calling the original *cuMemFree*. Then, when certain kernels from the checkpointed application are required to be issued, it initially checks whether all its necessary parameters have already been allocated; if some of them were checkpointed to the host memory, then the wrapping module brings them back to the GPU memory by calling the original *cuMemcpyHtoD* APIs and then the kernel is launched. Later, when the environment becomes suitable for suspended kernel to execute again, they will be awakened and continues their execution after the checkpoint module brings their transferred data back to the GPU memory by

calling the original *cuMemcpyHtoD* API (line 16 to 18).

When the application is required to release its GPU memory, because the wrapping module has stored the start address, allocated size, and device pointer of each allocation, it sends the size of the GPU memory that is going to be freed to the scheduler and calls the original *cuMemFree* to complete the deallocation when it gets a confirmed reply. In addition, when the application has finished its entire execution, the checkpointed module detects this fact as its destruction function is called. Then, it sends the information to the scheduler to deallocate the GPU memory used by the current application.

Figure 3.5 illustrates how GPU memory contents checkpointing works when several kernels try to allocate their own contents. In Figure 3.5(a), K1, K2 and K3 are three running kernels. K1 has already allocated its contents on GPU memory, while the GPU memory allocation requests of K2 comprise two separate parts. The remaining size of GPU memory after K1 allocates its own request can only satisfy the first GPU allocation request of K2. As a result, the second GPU memory allocation request is denied and a GPU memory checkpoint procedure should be started according to the decision of the scheduler. In Figure 3.5(b), the wrapping module checkpoints K2's 1st allocated contents to the host memory and makes free space for K3 to fulfill its GPU memory request. In Figure 3.5(c), K3 has finished its execution and freed its occupied GPU memory, however, since the remaining GPU memory cannot afford both of the 1st and 2nd GPU memory request for K2, the scheduler decides to wait without any action. Finally, in Figure 3.5(d), after K1 finished its execution, there are sufficient GPU memory space for both of the GPU memory request of K2, and then the wrapping module restores previous checkpointed contents and make the kernel execute again.

Label	GPU utilization(%)	execution time	Device pointer1	Device pointer2	etc.
Kernel 1	20	30ms	0x10000000	0x110000000	NULL
Kernel 2	30	10ms	0x10120000	0x112000000	0x1410000
Kernel 3	50	5ms	0x10140000	0x114000000	0x1530000

Table 3.1 Scheduling information

3.3.3 Scheduler

The *scheduler* schedules kernels according to the maximum available GPU memory of the system, current GPU core utilization, GPU core occupancy of each kernel, and GPU memory requirements of each kernel. Before each application starts running, the scheduler requires a unique ID for each wrapping module that is embedded in each application. The scheduler decides whether each kernel should be launched, each GPU memory allocation request should be executed, rejected, or delayed. It also decides whether previously allocated GPU memory contents should be checkpointed to the host memory or not.

The scheduler traces all GPU memory allocations of each kernel as well as the current GPU memory utilization. It also tracks the current GPU core utilization of the system at each kernel launching time and the GPU utilization of each kernel. We use nvProf [39] to profile each achieved occupancy of the kernel before executing the application. Meanwhile, the scheduler also tracks the size of the GPU memory that each launching kernel requires by storing the corresponding parameters that are transferred by the wrapping module during its launching time. Table 4.1 shows an example of the related information that the scheduler uses to schedule the kernels. This table is maintained by the scheduler and stored in the host memory. By referencing the table, the scheduler is aware of how much free GPU memory can be allocated and whether the kernel can be picked to run concurrently with other kernels.

Figure 3.6 demonstrates the entire procedure of the scheduling mechanism.

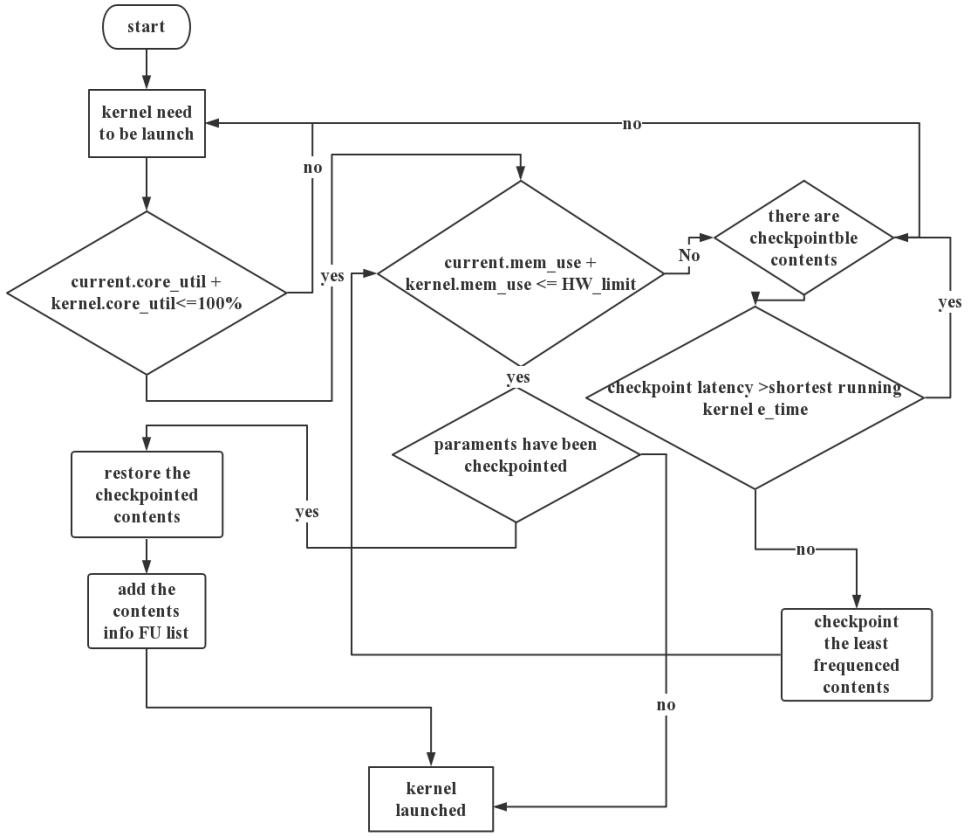


Figure 3.6 Scheduling procedure

The scheduler is triggered when any of the applications attempt to launch a kernel or when the launched kernel is finished. The kernel launching call as well as the related GPU memory allocation requests are intercepted by the wrapping module and forwarded to the scheduler. The scheduler initially checks whether the achieved occupancy of the system will exceed 100% or if the kernel will demonstrate a negative impact on the existing running kernel when the current kernel is launched. Demonstrating a negative impact implies that the running time while executing the kernels together is even longer than while executing them sequentially.

If the current kernel requires a large amount of computational resources or if it will demonstrate a negative impact on the current running kernels, the scheduler will delay its launch for a retry at the subsequent time slot. Conversely, the scheduler will check its GPU memory requirements to verify whether the GPU memory would be oversubscribed if the kernel is launched. If the GPU memory oversubscription occurs, the scheduler will crosscheck the reference table of current running kernels to see if there are any checkpointable contents.

We define "checkpointable contents" as the GPU memory contents that are not currently used by any of the running kernels. If there are no checkpointable contents currently, the scheduler will deny the kernel issue request and the request is on hold until the subsequent time slot. However, if checkpointable contents exist, then the scheduler checks the size of all these candidates and calculates the transfer time for their migration. The transfer latency can be tested by using the bandwidth test utilities provided by the NVIDIA CUDA.

Within the estimated migration time of each target kernel, the scheduler checks whether there is any kernel that has a shorter estimated migration time than the minimum execution time of the current running kernels. If some checkpoint candidates match this condition, then the scheduler finally selects the least frequently used contents as the final checkpoint target. The least frequently used contents are selected as the target because we considered the overhead when these contents are restored from the host memory. After the checkpoint target is selected, the scheduler sends a reply to the corresponding wrapping module. Then, the wrapping module executes the checkpoint procedure, which was introduced in the previous sections. When the target content is safely checkpointed to the host memory, the scheduler checks whether the free GPU memory size is sufficient for the execution request of the target kernel; If the freed

GPU memory size is still smaller than the required size, the scheduler repeats the procedure mentioned above. When enough GPU memory is available, the scheduler can allocate the memory and launch the kernel.

3.4 Evaluation

In this section, we initially present the evaluation of the overheads of FlexGPU. Secondly, we present an analysis of the performance by using the popular GPU benchmark and real-world workload. Finally, we demonstrate the performance gain of workloads consisting of multiple applications.

3.4.1 Evaluation setup

Our evaluations are executed on a machine that consists of two Intel(R) Xeon(R) E5-2683 CPUs and a 64 GB RAM. The GPU used was the NVIDIA Titan Xp, which features the Pascal architecture. There are 3,840 NVIDIA CUDA cores running at 1.6 GHz. Further, it is armed with 12 GB of graphics double data rate 5X memory. In addition, it supports NVIDIA Hyper-Q technology, which enables the parallel execution of multiple kernels. We used NVIDIA driver 384.130 with CUDA 9.0.176. The operating system used was Ubuntu 16.04 with Linux kernel 4.4.0.

The workload used includes dwt2d, gaussian, hotspot, lavaMD, nn, nw, and leukocyte, which are from Rodinia GPU benchmark [36]. The GPU utilization and GPU memory usage and the number of kernels launched varies among the different workloads.

We also use two real-world applications as our use cases. The first one is MUMmerGPU [22], which is used for aligning DNA sequences in bioinformatics. In our experiment, the search pattern is a sequence of 4,000 base pairs, which is matched against the reference that contains a complete genome in an alignment

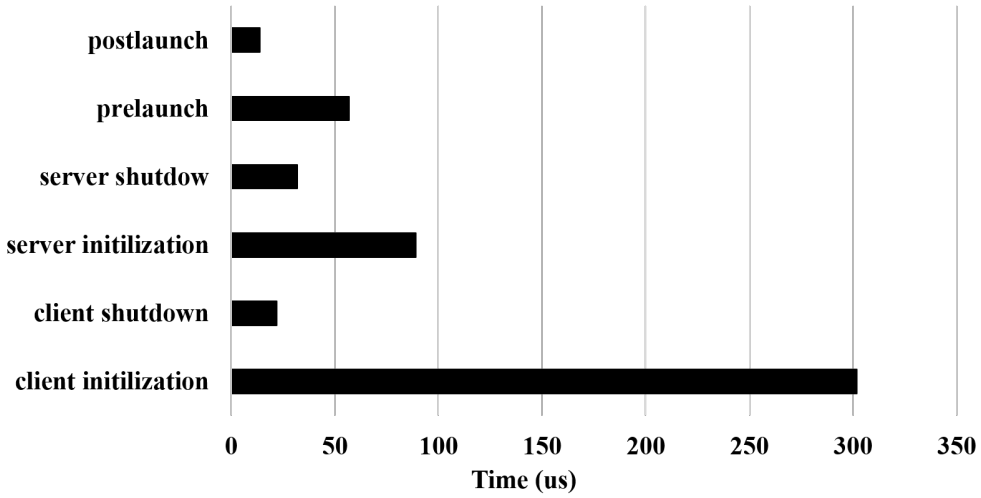


Figure 3.7 Breakdown of overhead in FlexGPU life cycle

of 15,000,000 reads.

The second application is a GPU-basic local alignment search tool (BLAST) [54]. This application searches a database of proteins for a nucleotide; further, the database used in our experiment is referred to as est_human.

3.4.2 Overhead of FlexGPU

Overhead breakdown

Figure 3.7 compares the different stages in the life cycle of FlexGPU. As our approach is based on the client-server structure, the client/server initialization and shutdown stages are required. From Figure 3.7, we can observe that the client and server initializations are relatively costly, even if they are called just once. This is because, in our approach, the wrapping module, which acts as the client, and the scheduler, which is referred to as the server, must manage the necessary information that are stored in their individual memory spaces. Allocating memory space for this information requires a longer time. In addition, as our implementation is based on the UNIX socket, communication initialization

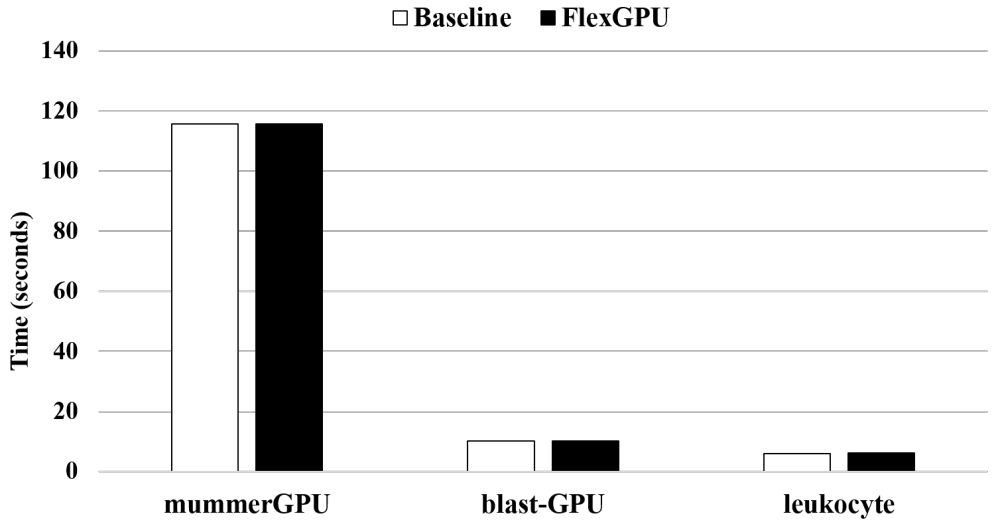


Figure 3.8 Overall overhead of FlexGPU

costs are also counted in this period. Besides, the pre-launch stage consists of a CUDA API interception as well as the allocation of the client request, while the post-launch stage contains the process of storing the relative information. It is to be noted that both the costs of pre-launch and post-launch stages are less than 60 us; Further, although these costs are counted every time when the corresponding CUDA APIs are called, the overhead is still acceptable when compared to the execution time of the application. Consequently, the overhead of our implementation is less than one millisecond. This does not have any impact on long-running applications.

Overall overhead

In this section, we evaluate the overall overhead including the wrapping and communication overheads. We compare the execution time for running one instance of MUMmergpu, GPU-BLAST, and leukocyte, with and without FlexGPU. Here, we only focus on the CUDA API wrapping and communication overheads.

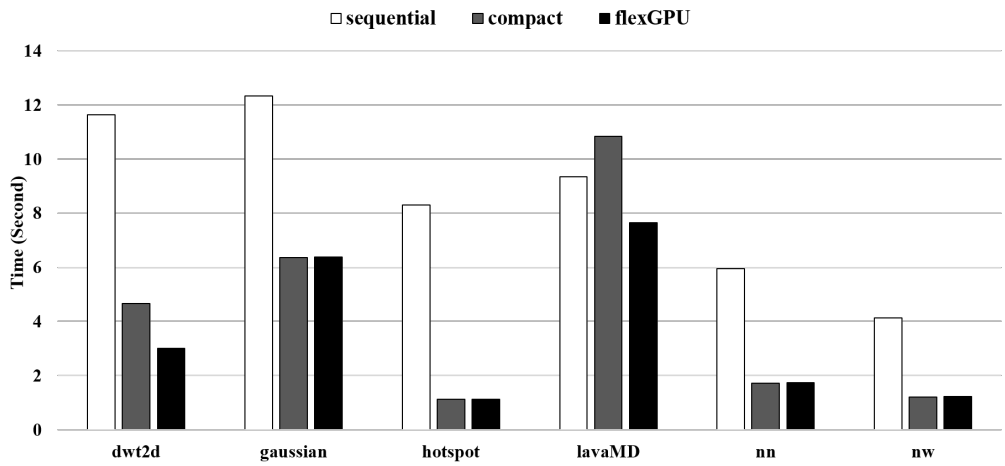


Figure 3.9 Average Latency of running 8 instances of benchmarks selected from Rodinia

As shown in Figure 3.8, owing to the overhead mentioned in the previous section, the overall execution time of FlexGPU is generally longer than the original version. However, the overhead can be neglected as the largest difference is 0.11 second when compared to the executing time of 6.02 second. From this evaluation, we can conclude that although the wrapping and communication overheads exists in FlexGPU, its impact is negligible.

3.4.3 Performance with GPU Benchmark Suits

Figure 3.9 shows the average execution time of six applications from Rodinia benchmarks. The average execution time is calculated by using the total execute time divided by the number of running instances. This evaluation runs in three different scenarios. The first scenario, which is referred as sequential, does not execute with FlexGPU. Instead, the instance of each application is sequentially executed eight times. Using this scenario in our experiment is reasonable because currently most of the GPU applications is designed to monopolize the

Table 3.2 Maximum GPU memory requirement and number of concurrent instance in compact mode

Workloads	Maximum GPU memory(MB)	# of concurrent instances
dwt2d	3757	3
gaussian	301	8
hotspot	183	8
lavaMD	2331	5
nn	171	8
nw	303	8

whole GPU without considering any sharing issues. The second scenario named as compact, also does not use FlexGPU. In this case, we manually pack up to eight instances of one application for concurrent execution. To make sure kernels running in parallel, we modify kernel issue part of each application to ensure each kernel is issued in different CUDA streams [**?**] and merge their CUDA contexts into a single one, taking advantage of NVIDIA Hyper-Q technology. However, owing to the differences in the GPU memory requirement of each application, the number of actual concurrently running instances varies. Table 4.2 demonstrates the maximum GPU memory requirement and the number of concurrent running instances that without OOM occurs of each application in the compact mode. In particular, as dwt2d and lavaMD require over 2,000 MB of GPU memory, the number of concurrent running instances is limited to three and five, respectively. The other four workloads can execute eight instances concurrently.

The third scenario employs FlexGPU and could safely run eight instances of all the applications even including dwt2d and lavaMD. As multiple instances use FlexGPU to run concurrently, the average execution time of all instances is reduced when compared to the first scenario, where eight instances are executed sequentially. We observe that by using FlexGPU, a seven times decrease in the

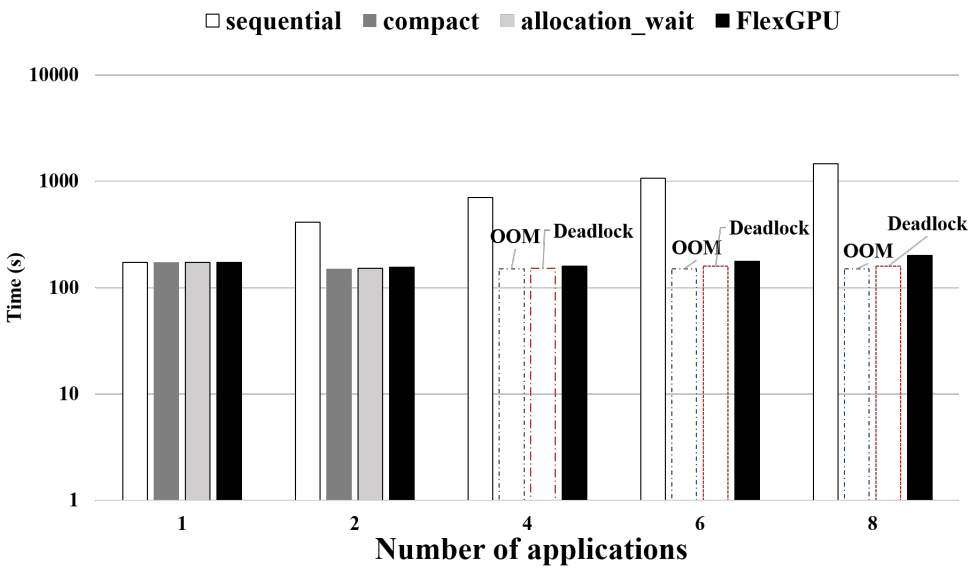


Figure 3.10 Latency of running up to 8 instances of MUMmergpu

execution could be achieved compared to the first case. For the workloads such as gaussian, hotspot, nn and nw, the average execution time of FlexGPU is slightly increased compared to the compact scenario, because of the overhead of FlexGPU itself. However, with dwt2d and lavaMD, the average execution time of FlexGPU is decreased by 35% and 29% respectively. We analyze the reason and determine that although the total execution time of FlexGPU is 72% and 13% (24.06s, 61.23s) longer than the compact scenario (13.97s, 54.18s) due to the overhead of frequent checkpoint operations, the number of concurrent running instances also increased by 2.6 times and 1.5 times compared to the second scenario respectively, leading to the large improvement on average execution time.

3.4.4 Performance with Real-world Workloads

We further evaluate the performance of FlexGPU on real-world workloads. We

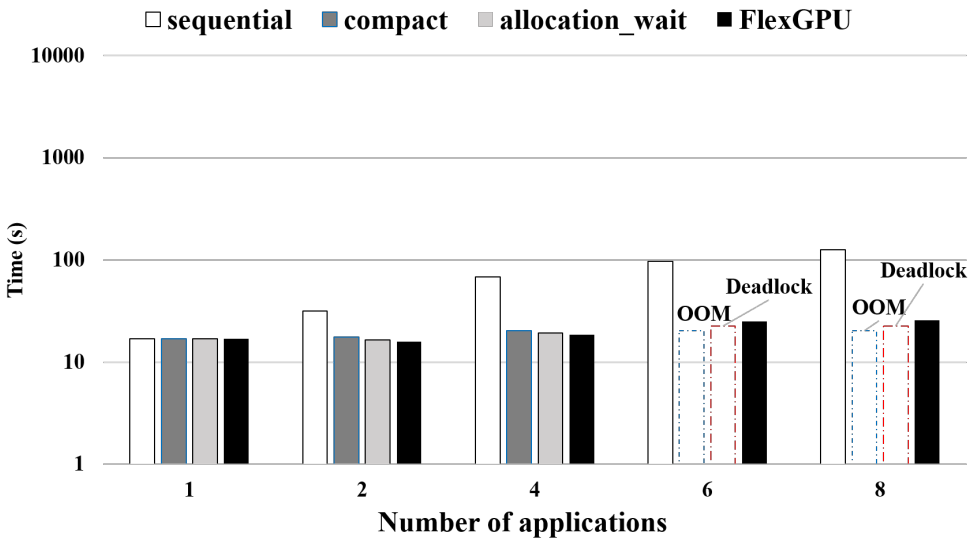


Figure 3.11 Latency of running up to 8 instances of GPU-BLAST

select mummerGPU and GPU-BLAST as our target workloads. Figure 3.10 and Figure 3.11 show the execution time of executing MUMmergpu and GPU-BLAST in four different scenarios. The first one is sequential execution, which sequentially execute each instance one after another. The second one is compact execution, which manually compacts multiple instances to make them run in parallel. However, because there is no additional mechanism to make sure all instance can fulfill their GPU memory allocation request, several instances may suffer in OOM failure. Table 4.3 demonstrates the maximum GPU memory requirement and the number of maximum concurrent running instances for each workloads in this scenario. The third executing scenario is called allocation wait scenario, which delays CUDA memory allocation call to make them wait until there is enough free memory. This is firstly proposed in [26]. However, owing to the deadlock issue we mentioned before, the number of instances that can safely execute in parallel is also limited in this scenario. Finally, we run the

Table 3.3 Maximum GPU memory requirement and number of concurrent instance in compact scenario

Workloads	Max GPU memory(MB)	#of concurrent instances
MUMmergpu	3985	3
GPU-BLAST	2215	5

instances with our proposed FlexGPU.

It is demonstrated in figure 3.10 and figure 3.11 that due to the lack of necessary GPU memory management method, both MUMmerGPU and GPU-blast experience OOM failure when the number of concurrent running instances exceed 3 and 5 respectively in the compact scenario. Meanwhile due to the deadlock issue, the number of safely parallel running instances is also limited in the allocation wait scenario. When Compared to the sequentially running scenario, our proposed FlexGPU achieved a seven times increase in speed for MUMmerGPU and approximately five times increase for GPU-BLAST. By using our proposed FlexGPU, the system can handle at least eight instances of both MUMmerGPU and GPU-BLAST running concurrently.

We also noticed that the execution time of running with FlexGPU rapidly increased when the number of instances become larger than 4 and 6, respectively. This is because in that case, GPU memory checkpoint is triggered and memory transfer overhead occurs.

There is a latency increase of 15% when running eight instances of MUMmergpu in parallel with FlexGPU, compared to the execution of a single instance at a time without FlexGPU. In case of GPU-BLAST, the increase goes up to 51%. This is because in the case of MUMmergpu, the GPU utilization remains 30% during most of the execution time, while it was higher for GPU-BLAST as its kernel is computation intensive. A relatively lower GPU utilization during kernel execution time allows FlexGPU to pack the kernels together and take advantage of the time periods that the GPU remains under utilized,

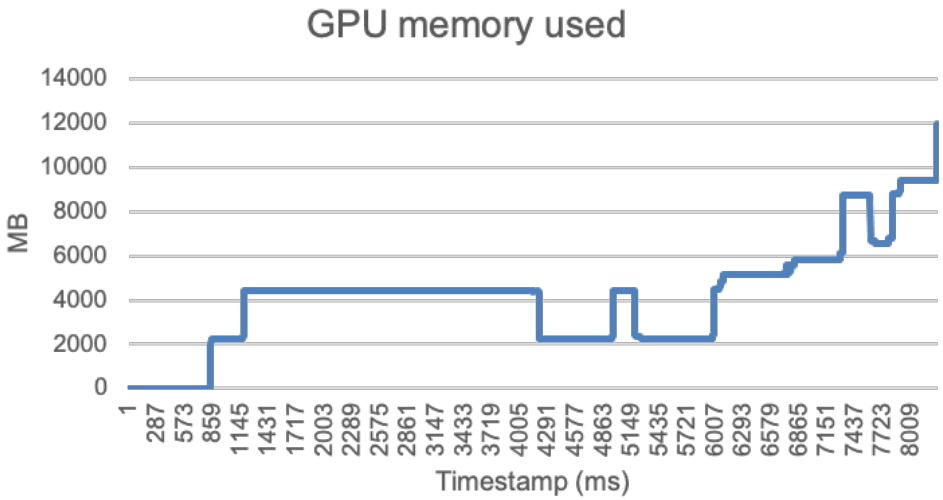


Figure 3.12 Running 3 MUMmerGPU and 3 GPU-Blast sequentially with OOM occurs

But a computation intensive kernel may limit the efficiency.

3.4.5 Performance of workloads composed of multiple applications

To observe how our proposed FlexGPU handles multiple applications executed in parallel, we launch three MUMmergpu instances and three GPU-BLAST instances with the following scenarios: 1) launching simultaneously, 2) delay the kernel launching to avoid GPU memory requirements collision, 3) Using our proposed FlexGPU.

Figure 3.12 shows the GPU memory usage. It can be observed that 601.3 s after the launch, the total amount of GPU memory exceeds the physical capacity and causes the failure of several instances with OOM error. Figure 3.13 demonstrates the delayed launching of resource-intensive kernel. It can be noticed that instead of exceeding the physical capacity, the kernel launch is delayed until the previous kernels have deallocated some of their allocated GPU memory

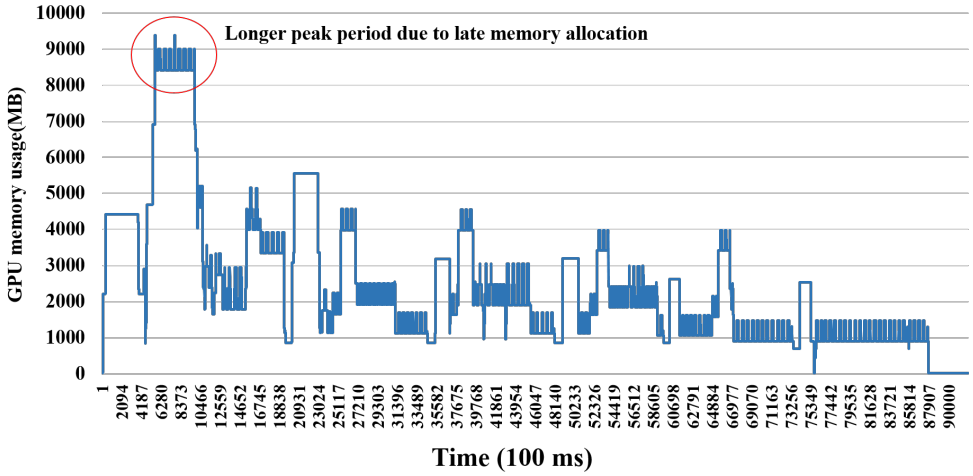


Figure 3.13 Running 3 MUMmerGPU and 3 GPU-Blast Concurrently with delayed kernel launch

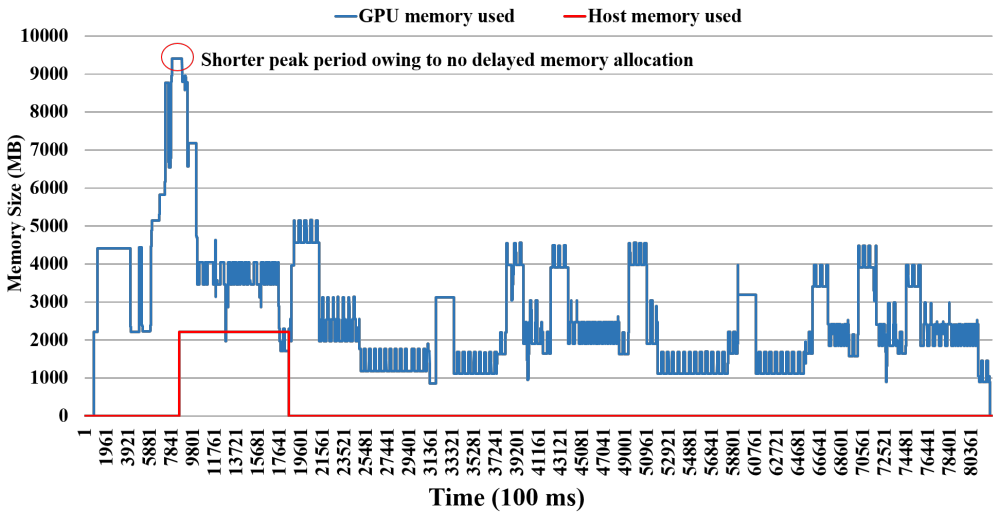


Figure 3.14 Running 3 MUMmerGPU and 3 GPU-Blast Concurrently with FlexGPU

contents. The peak memory usage shown in Figure 3.13 is generally less than 9 GB, which is maintained for a relatively long time. Figure 3.14 shows the details of how FlexGPU handles this situation. It can be noted that 835.2 seconds

after the application is executed, the FlexGPU detects that the current launching kernel would cause the GPU memory to exceed the capacity; consequently, it checkpoints certain unrelated content to the host memory and creates space for the kernel. Subsequently, when the checkpointed part becomes necessary again, it brings it back to the GPU memory and continues the execution of the corresponding kernel.

Both the delayed launching and FlexGPU can prevent OOM failure; however, using delayed launching may decrease the GPU utilization when kernels having lower GPU utilization but higher GPU memory requirements have to be executed sequentially. The experiment shows that in terms of the execution time, FlexGPU demonstrates an improvement of 11% (9,087.9 seconds to 8,111.1 seconds)

3.5 Summary

Currently, GPUs can be underutilized as multiple applications cannot share the GPU, even if some of them are not computation intensive. The primary issue is that applications cannot oversubscribe the GPU memory. Thus, GPU memory allocation requirements that exceed the physical capacity will cause an OOM error. In this paper, we proposed FlexGPU, which schedules the GPU kernels to improve the GPU utilization. It also allows the GPU memory content to be checkpointed and restored to improve the availability of each application; Consequently, it ensures that more GPU applications can share the GPU simultaneously. The evaluation demonstrated that FlexGPU improved the performances up to 7 times and enabled 2.6 times more applications to share the GPU.

Chapter 4

A Workload-aware Fine-grained Resource Management Framework for GPGPUs

4.1 Motivation

Currently, GPGPU workloads are designed to monopolize the GPU resource to achieve good performance, which may lead to a waste of resources. Launching multiple workloads concurrently on GPU was thought to solve this problem. However, NVIDIA GPUs originally only support time-sharing based scheduling [55] of co-located kernels, which lead to a near sequential execution of multiple kernels. To improve this situation, NVIDIA provides Hyper-Q [29] and MPS [30] technology. Hyper-Q technology enables multiple CPU threads or processes to launch work on a single GPU simultaneously. It allows multiple, simultaneous, hardware-managed connections between host and GPU, thus increasing the number of concurrent running kernels. However, to guarantee multiple kernels being scheduled in parallel, Hyper-Q technology requires

kernels to be issued with different CUDA streams [44] from the same CUDA context [41]. In other words, kernels from different applications cannot take advantage of concurrent scheduling enabled by Hyper-Q technology, since each application will create a separate CUDA context during execution. Therefore, NVIDIA proposed MPS to enable kernels from different CUDA context to be concurrently scheduled through Hyper-Q technology. Basically, the MPS client runtime is built into the CUDA Driver library and can be used transparently by any CUDA application, and the server process is the clients' shared connection to the GPU, which maps context created by each client into the one that was created by itself, thus leveraging the concurrent scheduling features of Hyper-Q technology.

In the meantime, according to Figure 1.3, shortest kernel execution time can be achieved when only a part of the SMs are activated as well as their intra-SM resources are not fully occupied. Taking BlackScholes as an example, it can achieve its best performance when 15 SMs are activated with 12 thread blocks launched on each of them. As revealed in table 4.2, 12 thread blocks of BlackScholes requires 35328 registers in each SM. Since NVIDIA Titan Xp has 65536 registers in each SM [35], the remaining 30208 registers can be used to launch 4 thread blocks of lavaMD additionally on the same SM.

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 fewer resources as possible, thus leaving more available resources for other workloads. In addition, we define near-optimal performance as a range between the best performance to a user-defined threshold. The threshold may change according to user requirement, we define it as 90% in our current research.

4.2 Related Work

4.2.1 GPU resource sharing

Multitasking in GPU management was not a sudden burst. A single execution of an application does not use up the entire GPU resources, such as register and memory. To handle this resource under-utilization issue, concurrent kernel execution must be supported to achieve the benefit. The research 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, attempts to apply Stream, Hyper-Q, and MPS to simulations and models were suggested to support kernel preemption and scheduling [9, 32, 51, 56, 57]. Park et al. proposed a preemption based approach [32] to control the overhead of multitasking on GPU. This approach is based on the flush operation that can preempt a 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 [51] 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 are not provided by the real-world GPU, these researches are implemented with the simulator, which may have different features with the real-world GPUs. Simulations were also performed for process-in-memory capabilities on GPUs [58]. Xu et al. proposed [31], 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. Concurrent kernel also have been proposed for embedded systems. Effisha [59, 60] proposed software technologies to

enable kernel preempt, however, their approaches are especially for embedded systems, which are not available on real GPU systems.

On the other hand, most of the software support studies are based on the persistent thread model [61,62], where thread blocks are treated as tasks issued by a persistent running thread. Bo et al. [33] firstly proposed the technology to circumvent the limitations of hardware scheduler and to allow a flexible program-level control scheduling. Slate [34] handles concurrent kernels from arbitrary applications at runtime and integrates workload-awareness into scheduling decisions, however, it focused on avoiding interference between different kernels and was scheduled based on SM unit.

4.2.2 GPU scheduling

As GPU multitasking has been in great demand, many studies start to focus on scheduling policies for task scheduling on GPUs. Mystic [63] circumvents the limitation that MPS does not support other scheduling policies by proposing an MPS-like context funneling system for GPU clusters but does not for a single GPU. Free launch used compiler techniques to statically combine a kernel with child kernels [64]. Wang et al. developed Kernel Fusion [65], a source-to-source compiler that can combine certain kernels for specific archetypes. Researches [28,66,67] proposed two aspects of thread block based scheduling, first, the lazy CTA scheduling (LCS) can restrict the maximum number of thread blocks allocated to the SM, second, Block CTA scheduling (BCS) policy assigns consecutive thread blocks to the same SM. Currently, the NVIDIA does not support the preemption of thread blocks of kernels. FLEP [68] pointed out that the lack of kernel preemption on commodity GPUs can lead to performance and priority inversion problems in multitasking environments. After FLEP transforms kernels into preempt-able forms, programs can be interrupted and yield

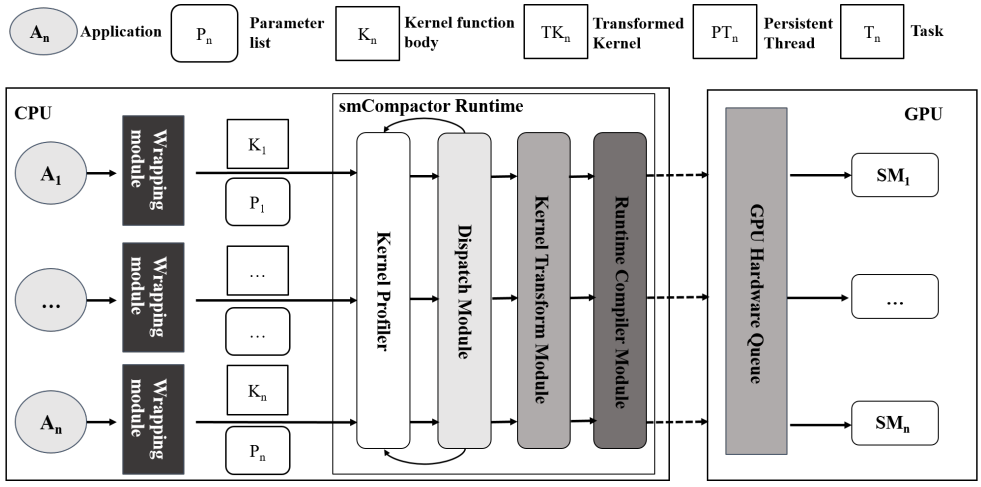
all or parts of SMs. Experiments show flexible preemption policy can enhance overall performance.

4.3 Design and Implementation

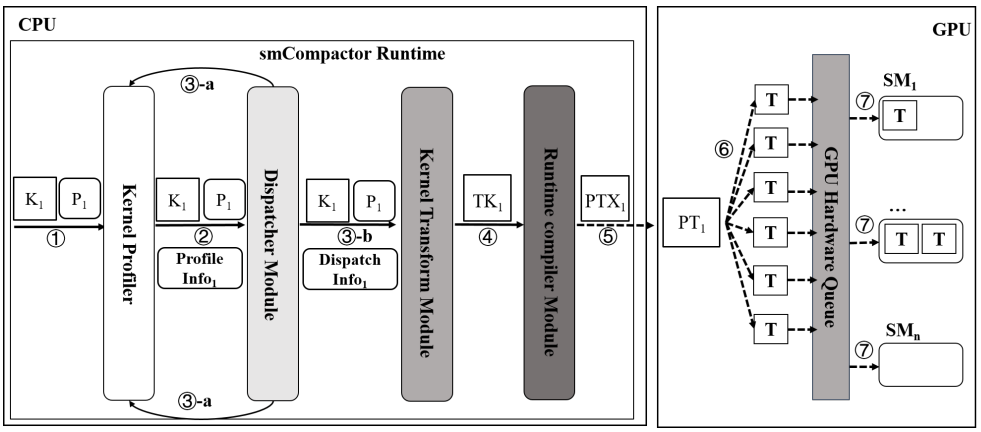
There are several goals in our design of the smCompactor. First, the smCompactor supports multiple kernels running concurrently without using NVIDIA MPS; second, the smCompactor supports efficient thread block level multiprocessing and resource sharing among multiple concurrent kernels, exploiting the GPU resources while maintaining the performance of each individual kernel execution; Third, the smCompactor provides a GPU kernel execution environment that is transparent to the users. Users can write GPU kernels with general CUDA APIs without additional modifications. To achieve our design goal, the smCompactor needs to automatically extract the kernel function and its parameters from the source code, transparently converting the original kernel function into a manually controllable version, profiling the kernel information, including the resource consumption, merging CUDA contexts into a single one, and finally managing the concurrent kernel scheduling according to that information.

4.3.1 System Architecture

Our approach is a client-server structure, based on the persistent thread mode [61]. Figure 4.1(a) demonstrates the architecture of the smCompactor. Each application is bound to a separate CUDA API wrapping module, which acts as the client. The CUDA API wrapper module intercepts the original CUDA kernel functions and their parameter lists for the future scheduling and resource management and transfer these intercepted contents to the server process. The server process runs smCompactor runtime on the host side, modifying the received kernel function body by adding scheduling and resource management



(a) Overview of smCompactor architecture



(b) Control flow of smCompactor

Figure 4.1 System architecture and control flow of smCompactor

related code. Then it uses the NVIDIA runtime compiler (NVRTC) to dynamically compile the revised kernel into ptx files [69], meanwhile merging CUDA contexts created by each client into a single one to exploit the concurrent scheduling feature of NVIDIA Hyper-Q technology. Finally, the smCompactor runtime launches the revised kernels to the GPU hardware queue.

4.3.2 CUDA API Wrapping Module

Originally, CUDA applications use the CUDA runtime and driver APIs to communicate and control the NVIDIA GPU. These APIs achieve multiple functionalities from the launch of CUDA kernels to the allocation of resources in the GPU and transferring of data between the device and the host. Among these APIs, *cuMemAlloc*, *cudaMalloc*, *cudaFree*, and *cuMemFree* are used to allocate and deallocate device memory, and *cuMemcpyDtoH*, *cuMemcpyHtoD*, and *cudaMemcpy* are used to transfer the data between the CPU and the GPU; further, *cudaLaunch* is used to launch a CUDA kernel function. Since in our proposed design, the smCompactor runtime is in charge of transforming the original CUDA kernel functions received from each client into the modified version that can be scheduled, 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 characters by parsing the *entry* parameter used in the *cudaLaunch* function. Besides, for the input and output parameters of the CUDA kernel, we firstly extract them by parsing the *entry*, then we obtain the size needed to be allocated for each parameter

by intercepting CUDA memory allocation functions, such as *cuMemAlloc*, *cudaMalloc*. Finally, we transfer the kernel function source and their parameter list, as well as their sizes, to the smCompactor runtime.

Unlike either the MPS, where device memory allocation occurs in each context and is then mapped to a single one, or recent researches such as [34], where device memory is allocated by the client, in our proposed design; clients only capture the contents and transfer them to the smCompactor runtime daemon. Device memory is allocated in the unique context that the smCompactor runtime daemon created. Since every device memory allocation is executed in the same context, physical addresses represented by device pointers and obtained by these allocation calls will not be in conflict, eliminating the potential GPU memory modification contentions.

4.3.3 smCompactor Runtime

The smCompactor runtime executes as a daemon process. As illustrated in Figure 4.1(a), it consists of a kernel profiler, dispatch module, kernel transform module, and runtime compiler module. Figure 4.1(b) shows the control flow of the smCompactor runtime. When the application calls CUDA API to allocate device memory and launch a kernel, the CUDA API wrapping module intercepts the kernel function body and parameter lists and forwards them to the dispatch module in the smCompactor runtime (①). The kernel profiler begins profiling the kernel when it is firstly executed or its dispatch information changes, and it forwards the profiling result to the dispatch module (②). The dispatch module adds dispatching related information to the kernel function body and keeps forwarding 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 body with dispatching information into a transformed kernel, which can be adopted in the persistent thread model. Then, it forwards the transformed kernel to the NVIDIA runtime compiler module (④). The runtime compiler module compiles the transformed kernel source into a ptx file, and it launches the modified kernel function through the ptx file (⑤). After the transformed kernel launched on the GPU side, a persistent thread will be generated, and then it will dispatch the tasks (⑥) to the specific SMs (⑦) according to the implanted dispatching information. The smCompactor runtime serves an important role in realizing resource utilization and managing current kernel scheduling. We will discuss each module in the smCompactor runtime in detail in the following subsections.

Kernel Profiler

The kernel profiler is in charge of profiling static and runtime features of each kernel. There are two models work with the kernel profiler, offline and online. Offline profiling obtains the static features of the kernel, while online profiling captures dynamic information. The static information includes the degree of parallelism, number of registers, and shared memory size used in each thread, and it is collected for the first time when each kernel is launched. The dynamic runtime features include the kernel execution time, instructions per cycle (IPC), device memory bandwidth, and L2 cache bandwidth. The profiler maintains a table to record the relation between runtime features and the number of activated SMs and thread blocks launched on each SM. The profiler profiles the dynamic information whenever the thread block and assigned SM number changes.

The reason that the smCompactor collects these features is because we use them to calculate how many resources remain in each SM and decide how

Label	SM0	SM1	SM2	...	SM30
Kernel 1	12	12	12	...	0
Kernel 2	5	5	5	...	5
Kernel 3	2	2	2	...	8

Table 4.1 Dispatching information

many more thread blocks can be issued to each SM. The profiler obtains the static features by utilizing NVIDIA CUDA Compiler (NVCC) [70] options while profiling the runtime information via NVProf [39]. Finally, the profiled result will be forwarded to the dispatch module.

Dispatch Module

The dispatch module injects the dispatching information into the users' kernel sources according to the obtained profiling information from the kernel profiler. The dispatch information will be a guideline for how to dispatch thread blocks of different kernels to each SM to maximize resource utilization while achieving near-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. The details are shown as follows; first, the dispatch module decides the number of SMs should be activated and thread blocks should be launched on those SMs for the first kernel according to its profiling result, considering the principle of achieving a near-optimal performance with consuming as fewer resources as possible. In the meanwhile, the dispatch module keeps track of the resource usage of each SM, recording available resources of each SM separately. Then, the dispatch module decides other kernels' dispatching information by preferring to consume resources remained by the previously decided kernel. Our thread block dispatching strategy refers to [71] since every intra-resource can

be treated as a vector and the whole dispatching problem can be transferred into a multi-dimensional bin packing problem. As mentioned in the previous section, every time creation of new dispatch information will lead to re-profile dynamic features of the new combination. The dispatch module can keep tuning the dispatch information according to the feedback from the kernel proflier, for example, activating new SM one at a time after remained resources are used up until all workloads achieve their near-optimal performance. It should be noted that the dispatch information only acts as a guideline for thread block dispatching, thus only a near-optimal performance can be achieved while the best performance cannot be guaranteed. How to schedule the thread blocks more efficiently based on our proposed framework is taken into the consideration as our future work.

The dispatching information contains the thread block & SM mapping data. As shown in Table 4.1, for any kernels coming into the system, the dispatch module provides the number of thread blocks that are supposed to be dispatched on each SM. We should also note that this dispatching information is not fixed; it varies depending on the current co-locating kernels since each kernel has its resource usage feature. Finally, it forwards this dispatching information with the user kernel function body and the parameter list to the kernel transform module.

Kernel Transform Module

As is well known, computations on GPUs are achieved by the kernel function. A kernel function is executed by GPU threads in parallel. Generally, GPU threads are scheduled depending on the hardware scheduler; they are dispatched to each SM in the thread block unit, totally beyond the programmer's control. Thus, improving the resource utilization according to each kernel's feature by manu-

ally controlling the dispatch of GPU threads is difficult to implement. However, [61] proposed the concept of 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 running permanently, picking up tasks from the task queue, and launching the tasks asynchronously.

We adopt this concept in our proposed smCompactor. In particular, the kernel transform module modifies the original kernel into a revised version that can be used in the persistent thread model. To fulfill the transformation, three kinds of modifications are needed. First, the kernel transform module replaces the built-in CUDA variables, such as the *blockIdx* and *gridIdx*, with the persistent thread model related code segments to implement its functionality. Figure 4.2 illustrates how the general user kernel is transformed into a persistent thread aware version. A general CUDA kernel with 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. A persistent thread, which is also running on the device side, sequentially extracts tasks and launches them to SM asynchronously. Second, dispatching information obtained from the dispatch module should be integrated into the revised kernel; therefore, thread blocks of certain kernels can run on the designated SM. Finally, to exploit the concurrent scheduling features of the hardware scheduler, the revised kernel should be launched with different CUDA streams [44]; therefore, a separated CUDA stream should be created and injected into the kernel launch part by the kernel transform module.

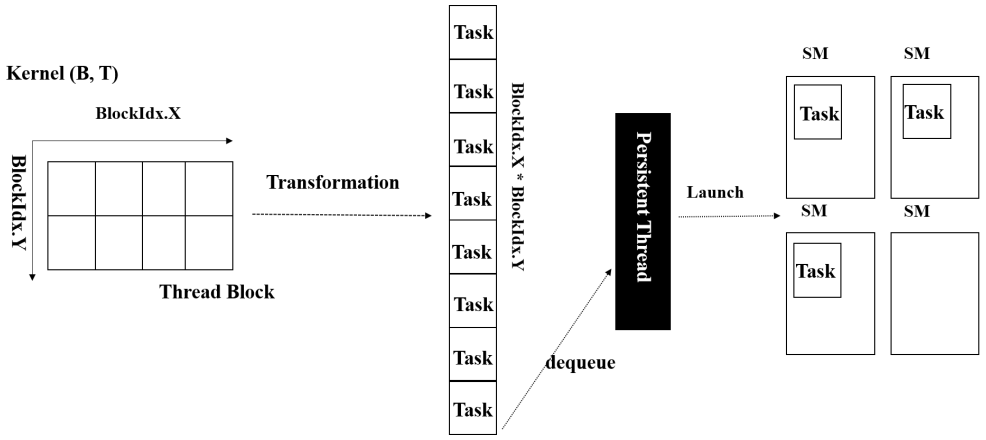


Figure 4.2 Kernel transformation.

Runtime compiler module

The revised kernels converted by the kernel transform module are presented as the source code of the function contents. To launch and execute those kernels, the source code has to be compiled into an executable form. The runtime compiler module works in four stages, first, it creates a unique CUDA context; second, it compiles the source code into ptx form; third, it allocates device memory for each kernel in that CUDA context; and fourth it launches the kernels on the GPU. As mentioned above, NVIDIA Hyper-Q technology enables concurrent scheduling only in the streams where the kernels issued belong to the same CUDA context. Besides, device memory allocation from different CUDA contexts will cause potential memory address conflicts, since different contexts are unaware of each other.

Figure 4.3 demonstrates the details of the runtime compiler module. This module contains one main thread and several child threads. The main thread creates the unique CUDA context and receives the revised kernel source from the previous module. For each child thread, it obtains the main context and

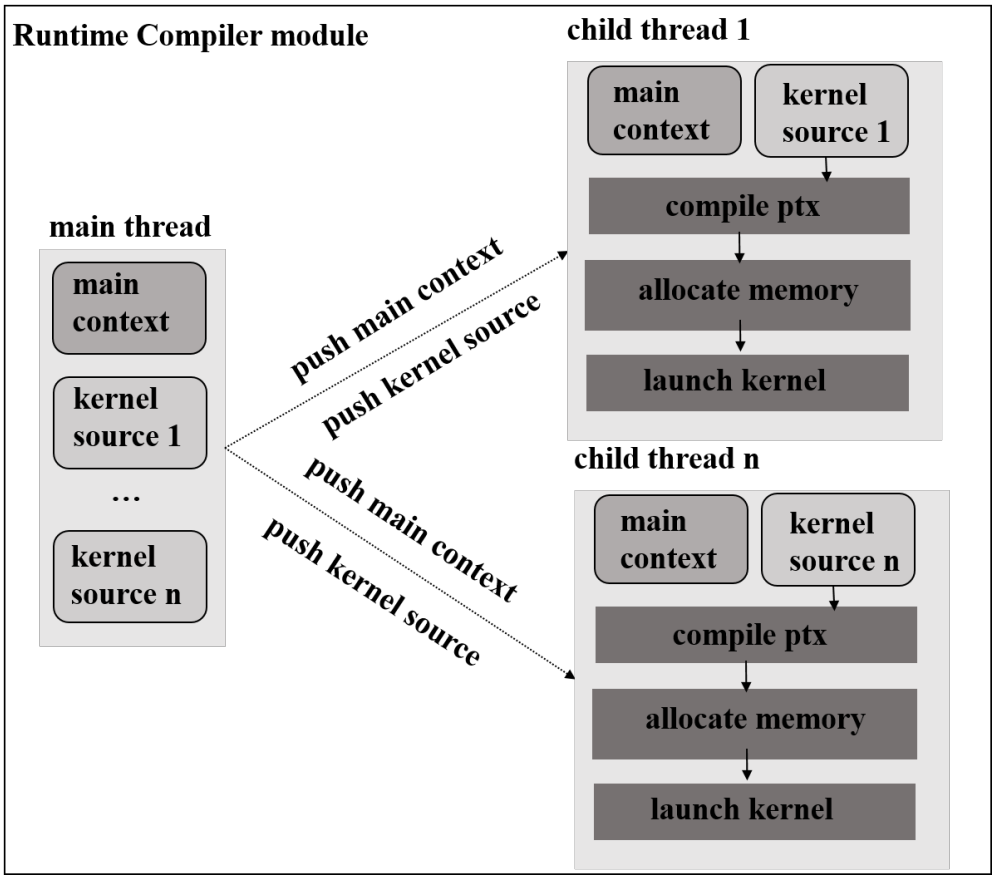


Figure 4.3 Runtime compiler module.

corresponding kernel source from the main thread, then compiles the source into ptx file, allocating device memory and launching the kernel under the main context.

4.3.4 Implementation Details

As mentioned above, the smCompactor adopts the concepts of persistent thread model to manually control the execution of the thread blocks of particular kernels on SMs. This concept shows us the possibility that the launch of thread blocks on the GPU can be handled by the programmer. However, how to assign a certain number of thread blocks of a kernel to specific SMs is still under research.

In our proposed smCompactor, we use a "fulfill and retreat" strategy to reach this target. 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. The persistent thread will keep trying to dispatch the task (thread block) until it locates the specific SM and until the number of thread blocks satisfies the dispatching information.

Figure 4.4 illustrates the details of the "fulfill and retreat" mechanism in our proposed smCompactor. As shown in Figure 4.4(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 4.4(b), the persistent thread dispatches one task, successfully to SM0 by the hardware scheduler. In this case, the persistent

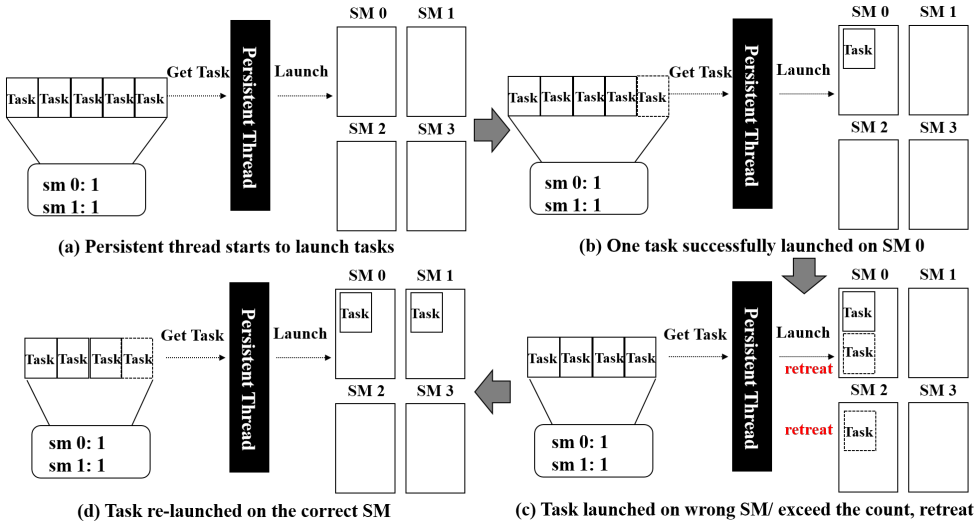


Figure 4.4 Details of fulfill and retreat mechanisms

thread pops this task from the queue and continues to dispatch the next one. However, as shown in Figure 4.4(c), the persistent thread dispatches the next task to SM2, which violates the dispatch information. As a result, the task needs to retreat and the persistent thread causes it to dispatch again. This time, the task is located on SM0, which makes the count of thread blocks running on SM0 exceeds the threshold. This should also be treated as a failed dispatch and the task needs to retreat again. Finally, as Figure 4.4(d) demonstrates, 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.

To implement the retreat functionality, we inject related code segments into each user kernel by the kernel transform module. Algorithm 4.1 reveals the details of these code segment. As shown in line 4, the first thread of the current thread block sequentially checks whether current SM is the target SM (Line 5) and current thread block count exceeds the threshold (Line 7). Any dissatisfaction with these conditions will lead to thread block retreats (Line 19).

Algorithm 4.1: Fulfill and Retreat

```
1: function gpu_kernel_function(parameter1, parameter2, ...)
2: retreat = false
3:
4: if current_thread = first thread of the thread block then
5:     if current_SM = target SM then
6:         add thread_block_count by 1 atomically
7:         if thread_block_count = threshold then
8:             retreat = true
9:         end if
10:    else
11:        retreat = true
12:    end if
13: else
14:    sync_threads
15: end if
16:
17: if retreat = true then
18:    sub thread_block_count by 1 atomically
19:    return
20: else
21:    do computation parts
22:    sub thread_block_count by 1 atomically
23:    return
24: end if
25: end function
```

When both of the conditions are satisfied, the computation is executed (Line 21).

4.4 Analysis on the relation between performance and workload usage pattern

As mentioned in previous sections, the execution time of GPU kennels can varies depends on the resources that be allocated to each kernel. In this section, we analysis the relation between the kernel execution time and intra-GPU resources that allocated to each kernel.

4.4.1 Workload Definition

We firstly definite the GPU workloads into two groups, the computational intensive workloads and the memory intensive workloads. According to [72], *Empirical Criteras* can be used to characterize the workloads. The definition of *Empirical Criteras* is as follow shows.

$$\begin{aligned} EmpiricalCriteras = (&flop_{count\_sp} + flop_{count\_dp} + flop_{count\_sp} + \\ &inst_{integer} + inst_{bit\_convert} + inst_{control})/ \\ &(gld_{transactions} + gst_{transactions}) \end{aligned} \tag{4.1}$$

A large Empirical Criteras value indicates the workload is relatively computational intensive while a small Empirical Criteras value shows the workload is relatively memory intensive. In our work, we use the Empirical Criteras value as one of the matrices to classify GPU workloads.

4.4.2 Analysis on performance saturation

Figure 4.5 shows the kernel execution time of mummerGPU, whose EC value equals to 49.5, on different cases of resource allocation. As the figure shows, the kernel execution time becomes saturated when 15 SMs are used. Since

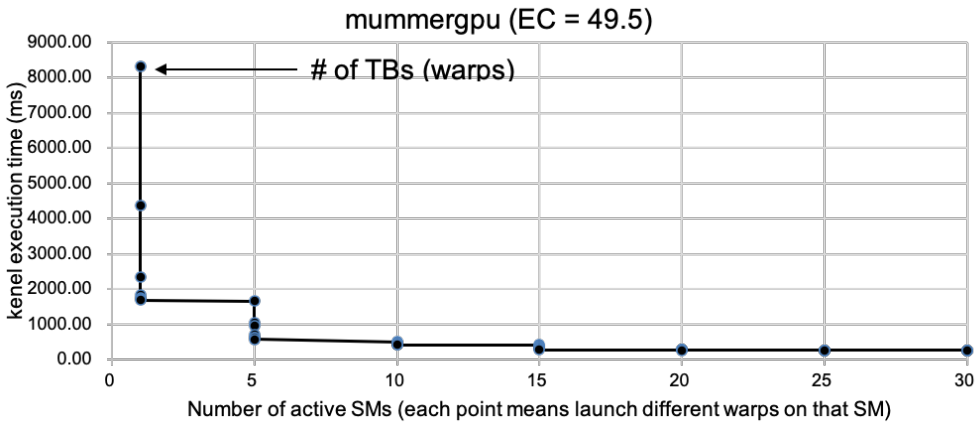


Figure 4.5 MummerGPU kernel execution time with different resource allocated.

the kernel execution time saturation is usually relate to memory bandwidth and parallelism bandwidth, we also profiles the gld_throughput, gst_throught, and achieved_occupancy on different cases of resource allocation. Figure 4.6 and Figure 4.7 show the gld_throughput and achieved_occupancy with different number of thread blocks allocated on different number of SMs, respectively. Compared the Figure 4.6, Figure 4.7 with Figure 4.5, we can tell that the change of kernel execution time of mummerGPU is highly related to memory bandwidth rather than the parallelism. Particularly, the kernel execution time saturated when 15 SMs are used, while the memory bandwidth shows the same tendency.

We also profile the computational workload to analysis the relation between performance and resource usage pattern. As Figure 4.8, Figure 4.9, and Figure 4.10 show, the kernel execution time of computational intensive workload is also highly related to the memory bandwidth rather than the parallelism. Thus, we can use the memory bandwidth as a metrics to predict the best kernel execution time.

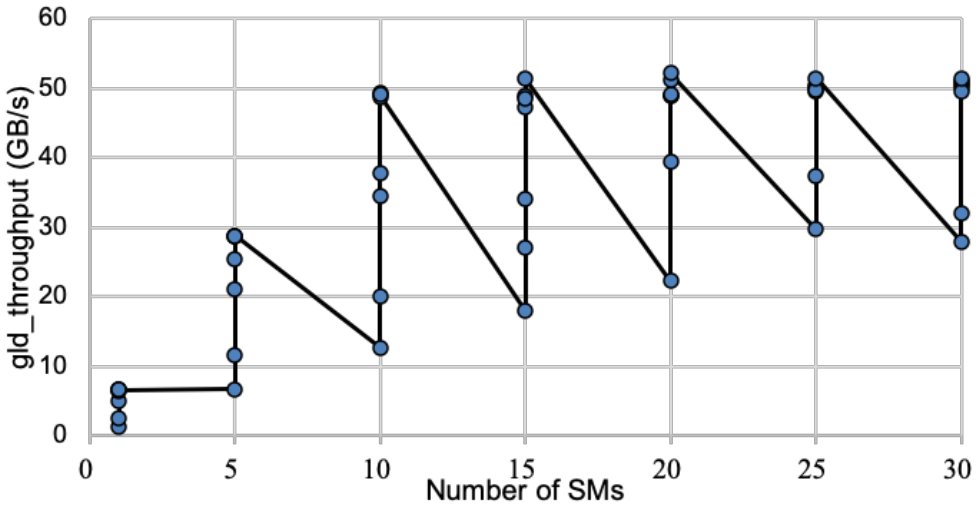


Figure 4.6 MummerGPU gld_throughput with different resource allocated.

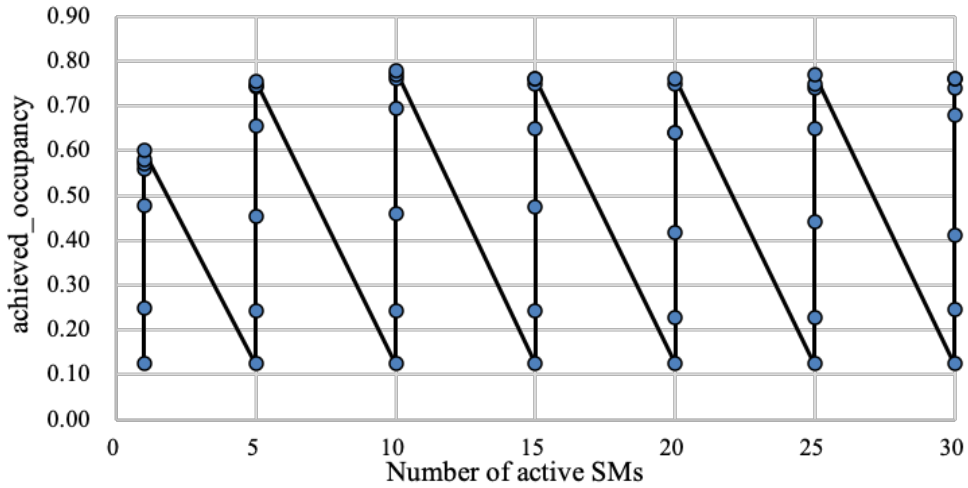


Figure 4.7 MummerGPU achieved occupancy with different resource allocated.

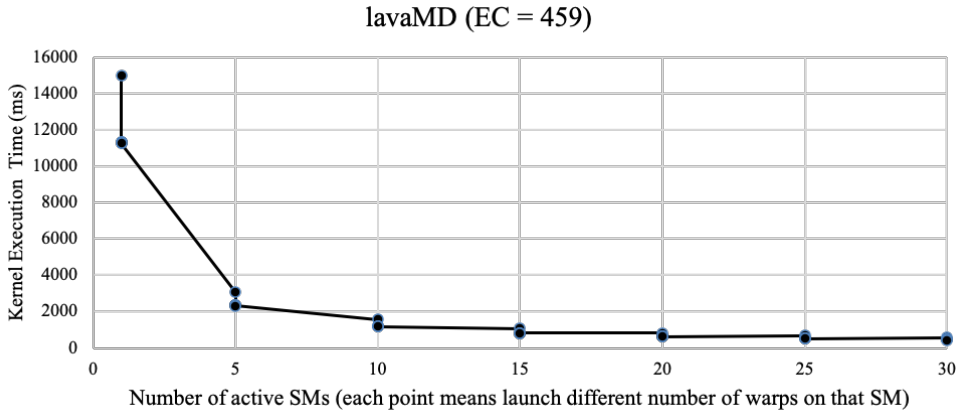


Figure 4.8 lavaMD kernel execution time with different resource allocated.

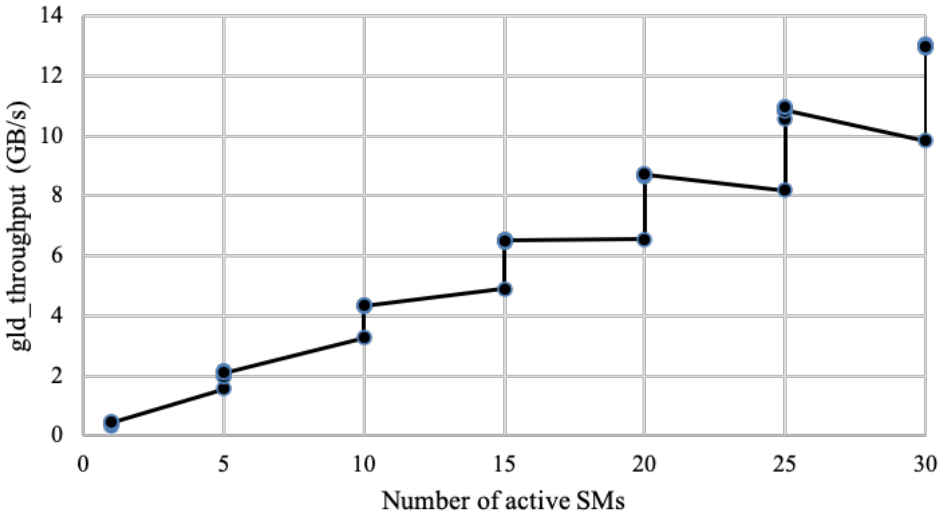


Figure 4.9 lavaMD gld_throughput with different resource allocated.

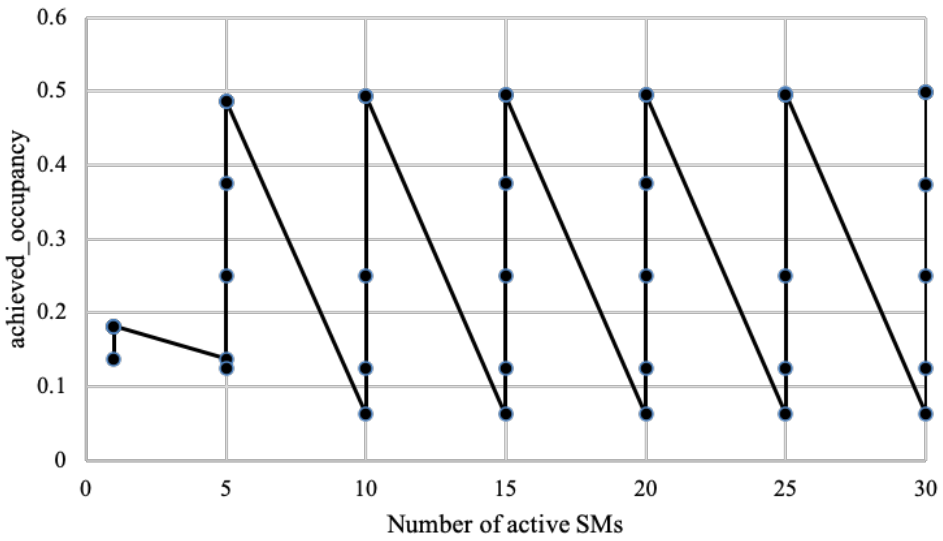


Figure 4.10 lavaMD achieved occupancy with different resource allocated.

Another observation is the resource usage on best kernel execution time of computational workload and memory intensive workload is really different. As Figure 4.6 shows, the memory intensive workload achieves its best performance when part of the SMs are used with large amount of thread blocks issued on each SM. However, As Figure 4.9 shows, computational intensive workload achieves its best performance when all the SMs are used with small amount of thread blocks issued on each SM. Figure 4.11 shows the two situations, and we call the case of memory intensive workload that compact horizontally while the case of computation intensive workload compact vertically

4.4.3 Predict the necessary SMs and thread blocks for best performance

As mentioned in previous sections, the kernel execution time of memory intensive workload and computation intensive workload is highly related to the memory bandwidth. Thus, prediction of the need SMs and thread blocks for

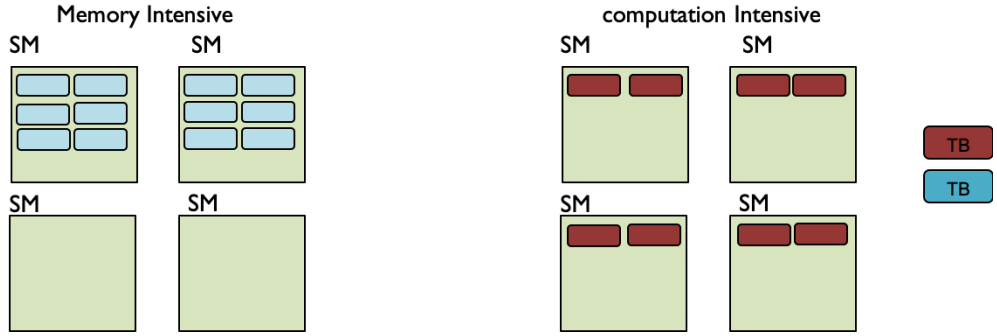


Figure 4.11 Comparison on memory intensive workload and computational intensive workload

achieving best kernel execution time can be transformed into predicting the necessary SMs and thread blocks to get the saturated memory bandwidth.

Figure 4.12, Figure 4.13, and Figure 4.14 show the memory bandwidth of mummerGPU when 1SM, 5SMs, and 10SMs are used, meanwhile, the X-axis in those figures shows the launched number of warps for each cases. From those figures, we can tell that before the memory bandwidth saturates, the tendency of memory bandwidth increase for each cases are similar.

Figure 4.15 demonstrates the increase of memory bandwidth when 1 thread block issued on different number of SMs. From the Figure, we can find that the memory bandwidth increase linearly with the number of SMs increase. As a result, we can use the memory bandwidth profiled in 1SM, and the saturated memory bandwidth to predict necessary SMs and thread blocks per SM.

$$saturated\_throughput = SM\_NUM * start\_throughput * TB\_NUM^{improve\_factor} \tag{4.2}$$

We use Equation 4.2 to describe how to calculate the necessary number of SMs and thread blocks launched on each SM. The saturated throughput means the memory bandwidth when best performance is achieved, the start throughput

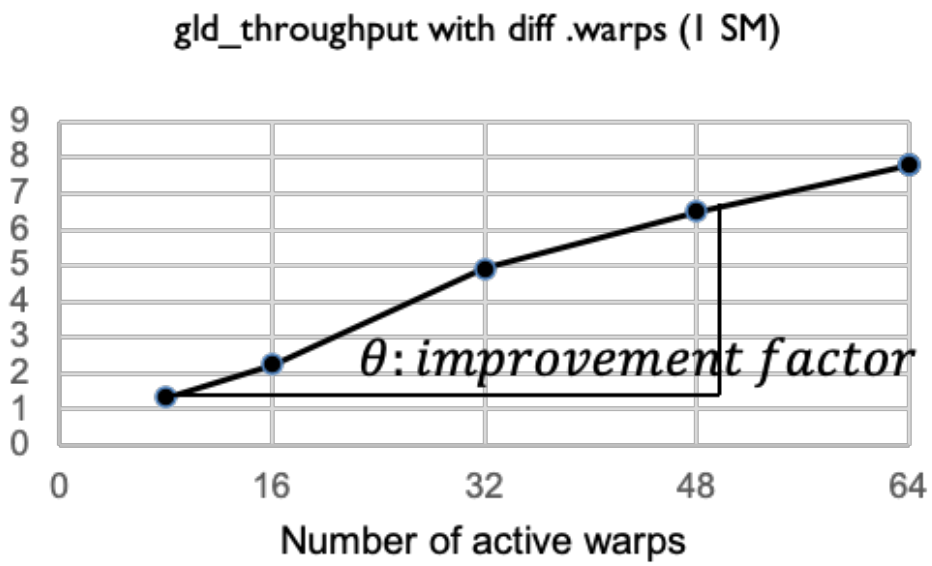


Figure 4.12 memory bandwidth with different number of Warps on 1 SM

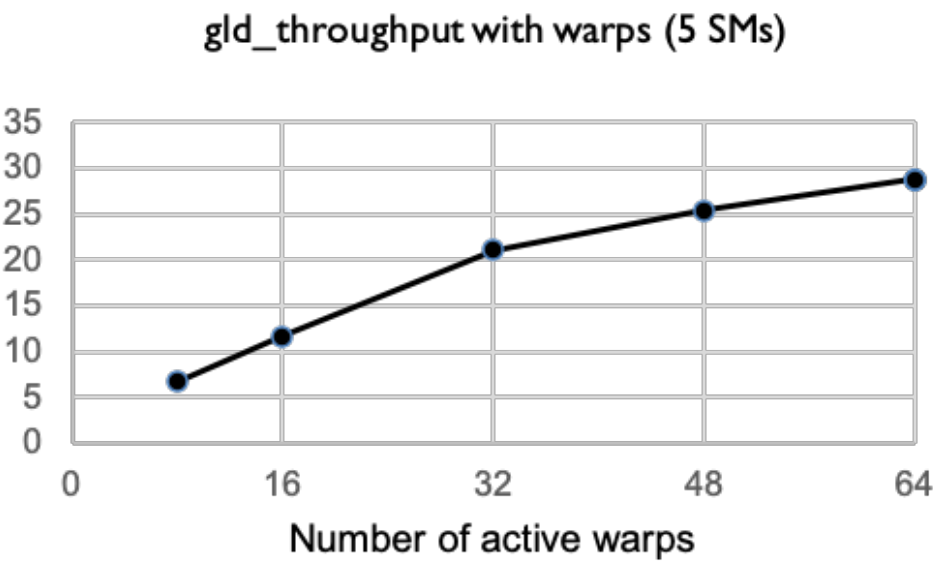


Figure 4.13 memory bandwidth with different number of Warps on 5 SMs

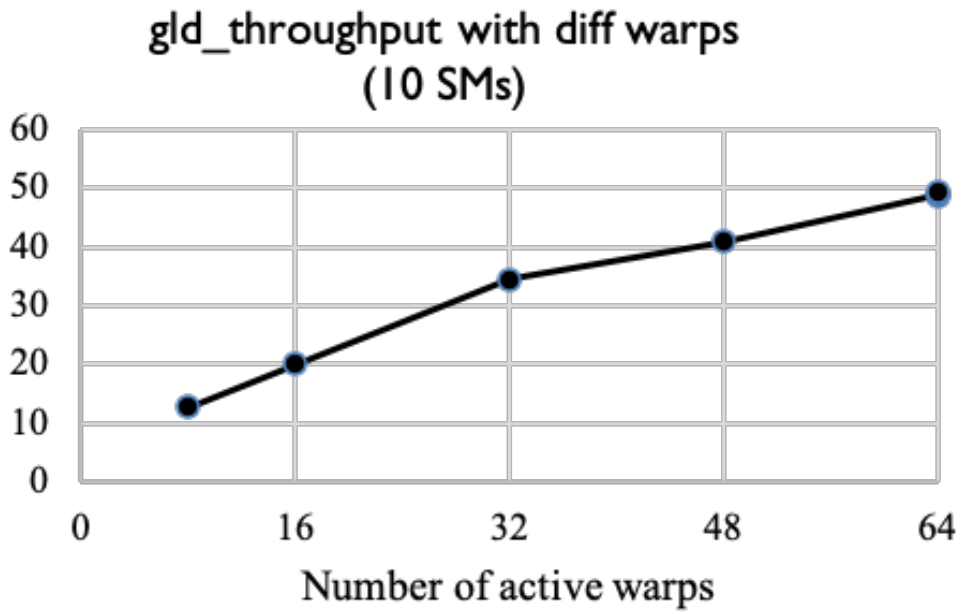


Figure 4.14 memory bandwidth with different number of Warps on 10 SMs

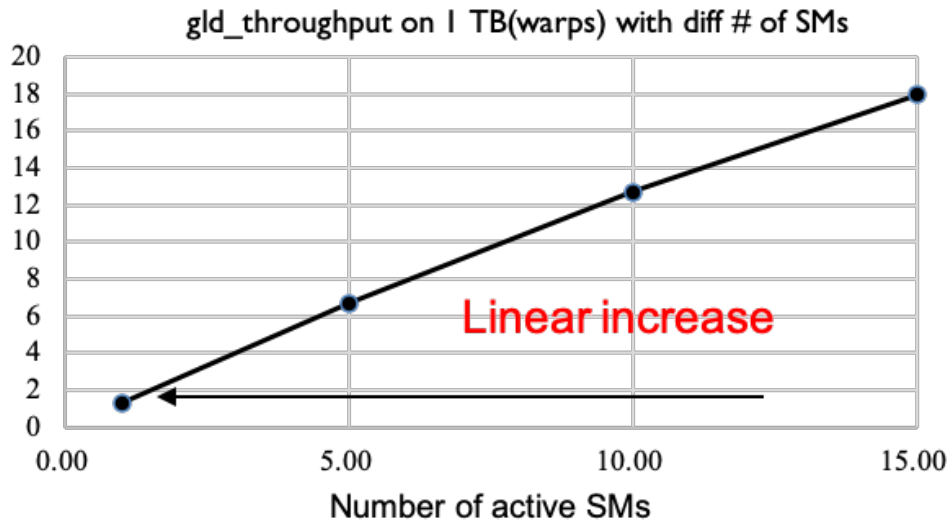


Figure 4.15 memory bandwidth with 1 thread block on different number of SMs

means the memory bandwidth of one thread block issued on 1 SM, and the improve factor indicts the increase tendency that shows in Figure 4.12. All of those metrics can be achieved by profiling workload when all resource are allocated and only 1 SMs are allocated. In addition, the SM count on GPU is limited, for example there are 30 SMs on NVIDIA Titan Xp, and the thread blocks that can be launched is also limited due to the resource usage. As a result, we can use those conditions to predict the necessary number of SMs and thread blocks for each SM that can achieve the best performance.

We also found that the memory bandwidth can be distributed to different number of SMs with different number of thread blocks launched. Taking BlackScholes as example, the saturated bandwidth is 266GB/s, which can be achieved when 15 SMs are activated with 10 TBs launched on each SM. Since 266GB/s can be divided into 96GB/s + 170GB/s, according to our Equa-

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
QuasiRandom Generator (QG)	128	384	15360	0
lavaMD (LM)	1000	128	7168	7200

Table 4.2 Target Evaluation Workloads

tion 4.2, the necessary SMs for 96GB/s and 170GB/s are 5 SMs with 8 TBs and 10SMs with 8TBs, respectively. The real memory bandwidth of 5SMs and 10SMs activated individually (No.0-4, No.20-29), is 262GBs, which is similar to the saturated bandwidth.

Thus, according to our observations, we can flexibly compact the resource space of each workload while maintain their performance, and the profiling effort can be limit to an acceptable level

4.5 Evaluation

In this section, we evaluate the performance of the 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 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.

4.5.1 Evaluation Methodology

Target Evaluation Applications

Table 4.2 shows the target evaluation workloads. They are from the NVIDIA CUDA 9.0 Samples and Rodinia GPU benchmark suite [36]. 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 version, MPS version, slate [34] version, and our proposed smCompactor version. Since the slate is not an open source framework, we implement it according to the details introduces in that paper.

Evaluation Metrics

We evaluate our proposed smCompactor using the following performance metrics. 1) real & normalized kernel execution time, 2) resource utilization in terms of a number of active SMs for each workload.

The original CUDA version uses time-sharing as its scheduling policy. In this instance, one kernel occupies the whole GPU resources during its time slice and switches its control of resources to another kernel at the next time slice. Both MPS and slate enable spatial sharing policy; However, in the case of MPS, it only allows kernels running when there are available resources near the end of the previous kernel's execution if their launch time is slightly staggered, its execution also totally depends on the hardware scheduler, which is beyond the control of users. Slate enables selecting complementary kernels according to the profiled information, and it isolates each kernel on specific SMs. Our proposed smCompactor also enables spatial sharing among the concurrent kernels, in addition, it provides a fine-grained scheduling mechanism, which can control the number of thread blocks on the specific SM to find out the optimal combination of thread block and SM pairs, increasing the resource utilization

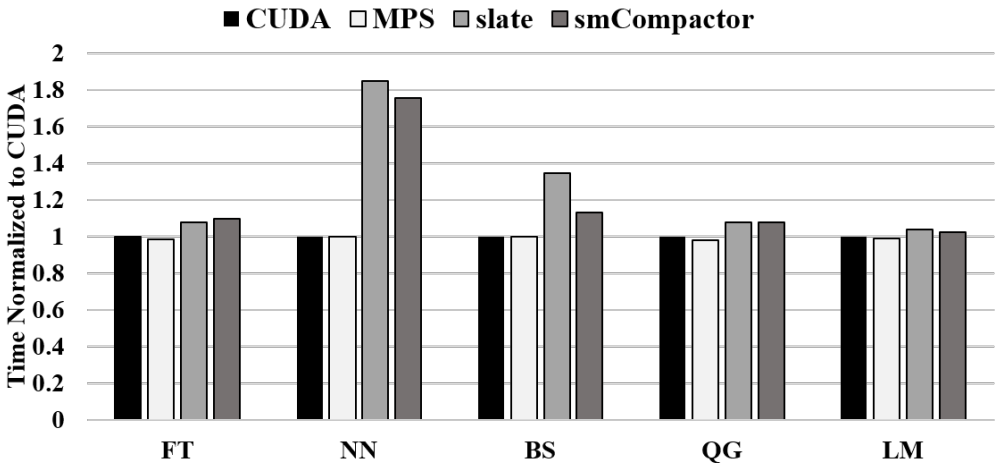


Figure 4.16 Execute time normalized to original CUDA.

while maintaining the performance.

4.5.2 Overhead of smCompactor

We first 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 also configure the slate to use all the SMs and configure smCompactor as not limiting the number of thread blocks running on each SM.

Figure 4.16 shows the execution time of each benchmark running solo in different situations. The result is normalized to the execution time of running the original CUDA. Generally, the smCompactor has up to 7% overhead compared to the original CUDA case, which is similar to the slate. The overhead is due to the persistent thread model, where the user kernels nested in the dispatcher

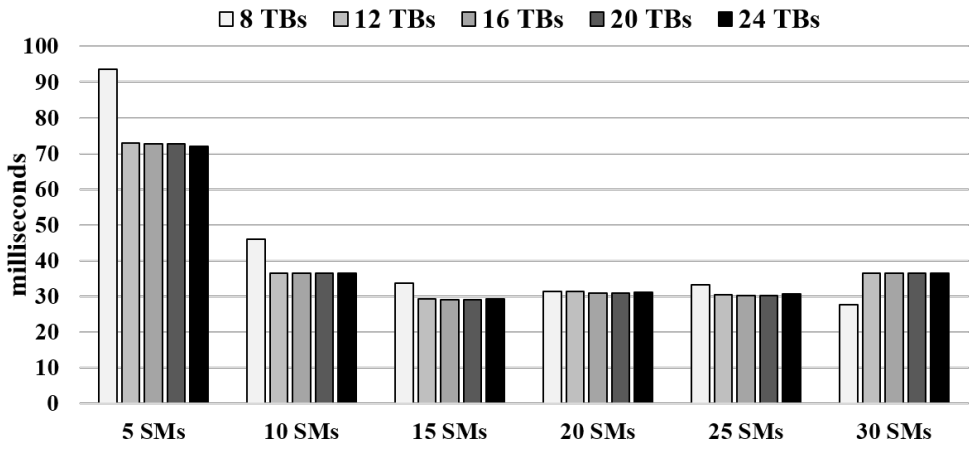


Figure 4.17 Execution time of BlackScholes with the different number of thread blocks (TBs) on the different number of SMs.

kernel (persistent thread), and both the smCompactor and slate adopt this concept. However, the case of NN is an exception, where the overhead is about 1.75 times of 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 millisecond, 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.

4.5.3 Performance with Different Thread Block Counts on Different Number of SMs

In this section, we evaluate smCompactor by running a single workload with different number of thread block counts on 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 4.2, FDTD3d consumes a large amount of registers and shared memory with a small

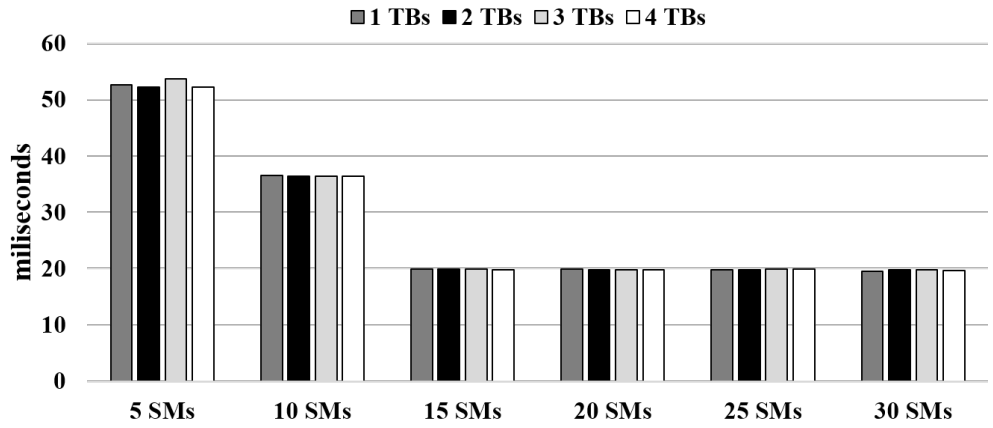


Figure 4.18 Execution time of FDTD3d with different number of thread blocks (TBs) on different number of SMs.

number of thread blocks, while BlackScholes has low resource usage but a large number of thread blocks.

Figure 4.17 illustrates the execution time of BlackScholes when launching different number of thread blocks on different numbers of SMs. From the figure, we can tell that the execution time varies according to the change of the count of thread block and active SMs. The best performance (27.6 ms) can be achieved when 30 SMs were activated and launching 8 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). We can also tell that the performance can be almost saturated when only half of the whole SMs were activated with appropriate number of thread blocks launched on each of them, leaving ample room for improvement in resource utilization.

Figure 4.18 shows the case of executing FDTD3d with different number of thread blocks on different number of SMs. The performance was 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 1 to 4. This is because of its large resource usage of each thread block since 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 near-optimal performance can be achieved without all SMs are activated when an appropriate combination of the number of thread blocks and the active SM counts are found. However, compared to resource consumption intensive kernels, those small kernels have a higher possibility to find the near-optimal performance with less active SMs since their performance varies greatly when the number of thread blocks launched on each SMs changes.

4.5.4 Performance with Concurrent Kernel and Resource Sharing

In this section, we present the kernel execution time of co-launching two applications on the system concurrently with different strategies. We select FDTD3d, BlackScholes, and lavaMD as our target workloads, since their execution times are similar among themselves. The evaluation is conducted with original CUDA, MPS, slate and our proposed smCompactor. We evaluated 8 combinations among all the workloads. In the case of slate, we configure it to use all SMs for the workloads and each workload occupies a multiple of five number of SMs, then we choose the best performance among all the combinations. For example, workload A is dispatched to five SMs, while workload B is dispatched to the remaining 25 SMs and so on.

Figure 4.19 demonstrates the kernel execution time and number of active SMs on all eight cases. The time presented in Figure 4.19 is the execution

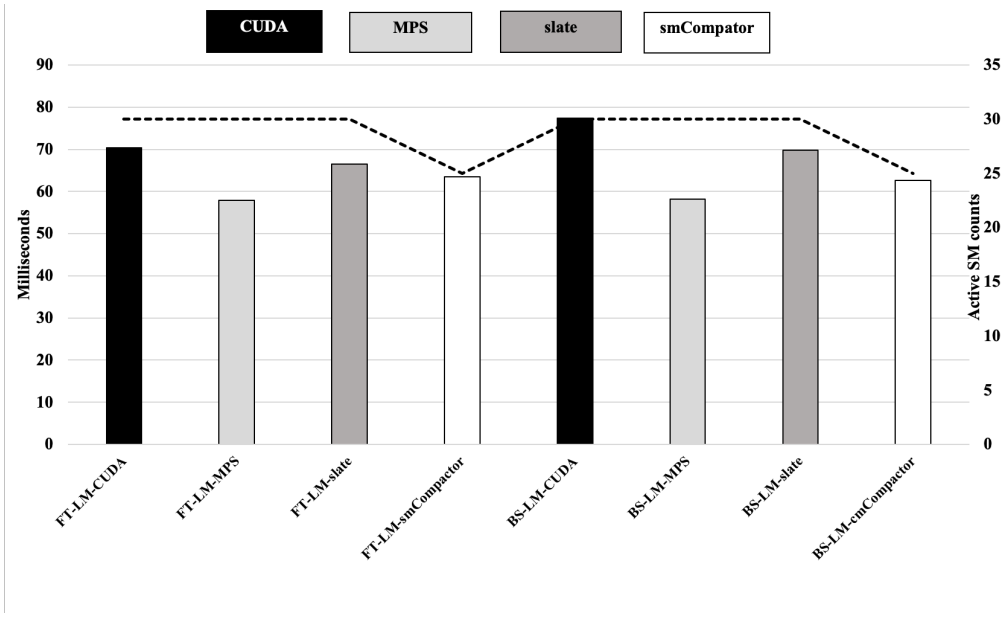


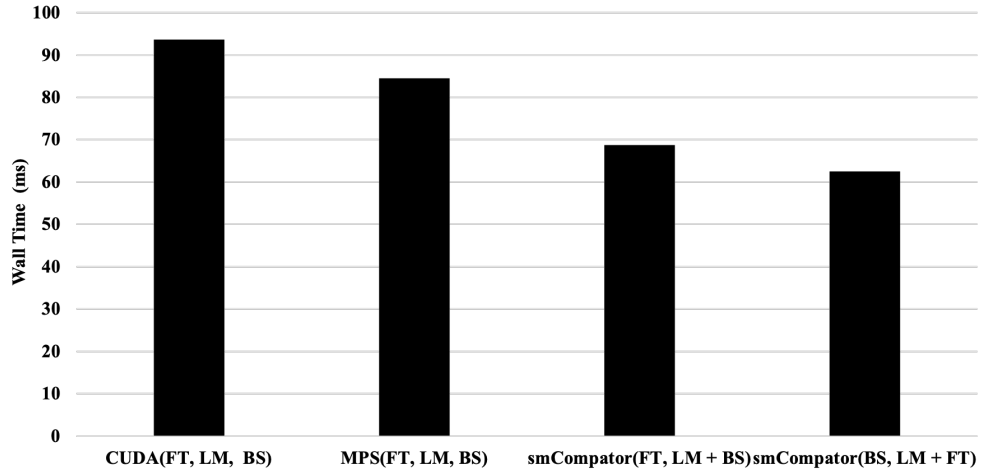
Figure 4.19 Execution time and active SM counts of running in different scenarios.

	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

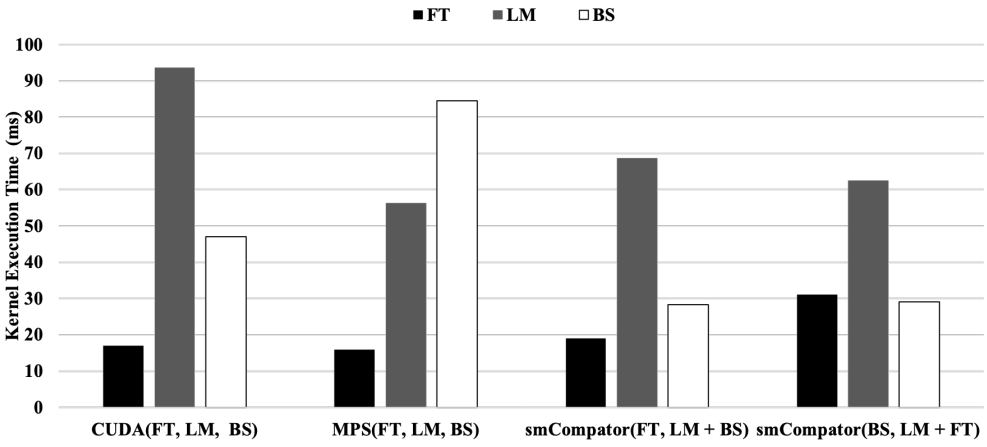
Table 4.3 Number of thread blocks on each SM

time of the last finished workload. The original CUDA provides the baseline for the comparison. As Figure 4.19 shows, 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, respectively. The slate also outperforms the original CUDA in both cases; however, it suffers performance downgrades of 14.8% and 22.5% compared to MPS in the case of FT-LM, and BS-LM respectively.

Our proposed smCompactor also outperforms the baseline in both cases. It can enhance the performance against slate by 10% and 16% when concur-



(a) Wall execution time



(b) The execution time of each workload

Figure 4.20 Execution time of co-locating three workloads with different strategies

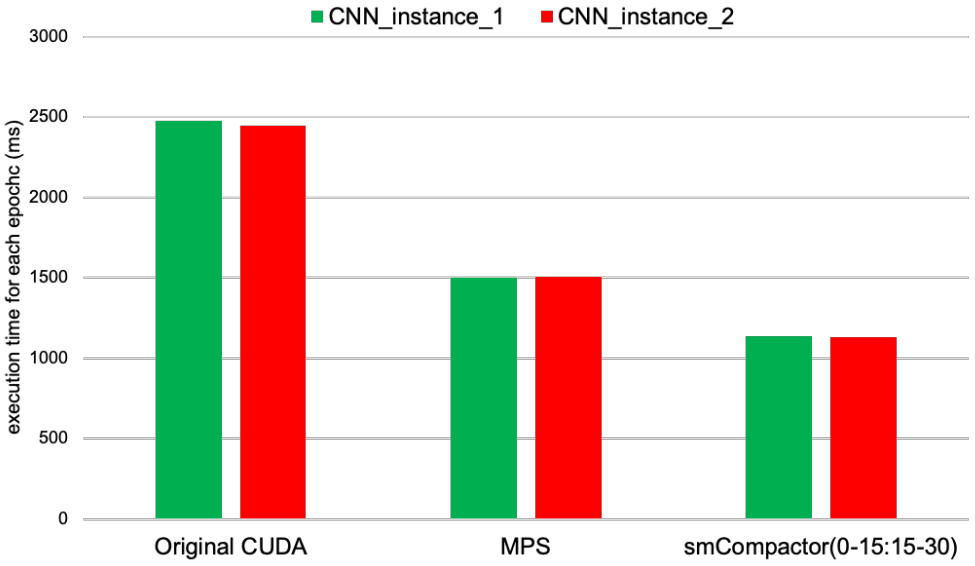


Figure 4.21 Execution time of CNN with different strategy.

	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 4.4 Number of thread blocks on each SM

rently running FT-LM and BS-LM respectively. It should be noted that resource utilization can be increased when the workloads running with our proposed smCompactor. Even though the execution time of the FT-SM and BS-LM combination is slightly longer than the MPS cases, it only uses 83% 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 4.3.

To demonstrate that those saved SMs by using our proposed smCompactor can also be used to execute other kernels without introducing 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

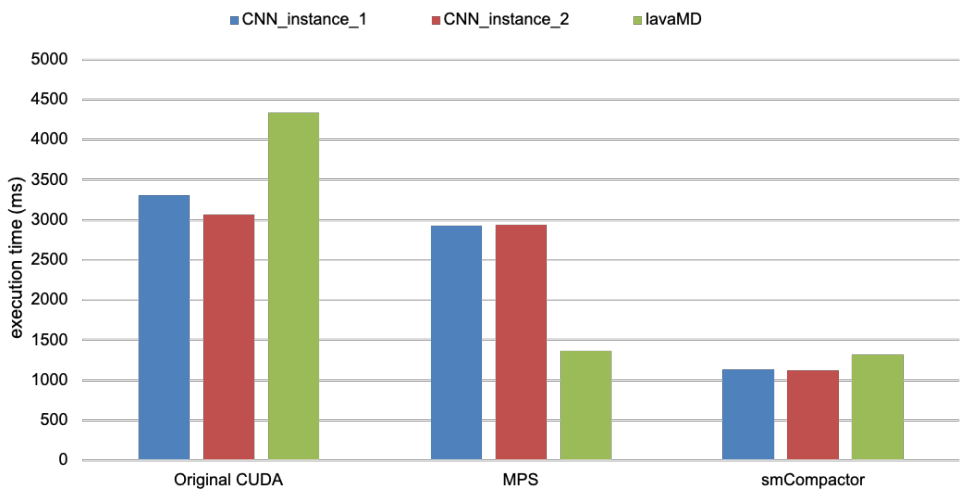


Figure 4.22 Execution time of CNN and lavaMD with different strategy.

of BS mainly on the remaining 5 SMs for the FT-LM case, and thread blocks of FT on the remaining 10 SMs for the BS-LM cases. Table 4.4 depicts the details of the thread block distribution on every SM. Compared to the previous experiment configuration as shown in Table 4.3, we slightly tune up the thread block number of LM to achieve an overall better performance.

Figure 4.20 shows the detailed result of the evaluation. Figure 4.20(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 gain can be even larger in the case of co-launching FT on the SMs remained by BS and SM, which is 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 in Figure 4.20(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 4.19, we can tell that as more kernels are launched in parallel, the possibility increases that one or some of them can be blocked.

On the other hand, the wall time of 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 count 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.

We also test our smCompactor with MPS and the original CUDA case on the CNN workload. Figure 4.21 shows the result. From the Figure, our proposed smCompactor outperform MPS 24% when running 2 instances. In addition, we run the third workload: lavaMD on the remained resources, and Figure 4.22 shows the result, where our proposed smCompactor outperform MPS 61% in terms of CNN and 5% in terms of lavaMD.

4.6 Summary

Currently, GPU resources can be underutilized even when multiple kernels run in parallel. The main issue is that the scheduling of thread blocks depends on the hardware scheduler, which is beyond the control of the users. Besides, the GPU hardware scheduler cannot detect the relation between performance and resource usage, making it difficult to improve resource utilization while

maintaining the performance.

In this paper. we proposed the smCompactor, a fine-grained thread block scheduling framework, which can improve the resource utilization while maintaining the performance by dispatching an arbitrary number of thread blocks to specific SMs. It adopts the concept of the persistent thread model, using a CUDA API wrapping module, dispatching module, kernel transform module, and runtime compiler module to transparently and automatically modify the original user kernel to the revised version.

The evaluation results demonstrate that our proposed smCompactor has minimal overheads. Moreover, near-optimal performance is obtained with fewer SMs by managing the number of thread blocks launched to each SM. For the multiprocessing cases we looked at, the performance gain against the original CUDA and MPS are up to 33% and 26%, respectively.

Chapter 5

Conclusion

Efficient utilization of GPU resources under multitasking environment is becoming ever more significant. However, the exist GPU management system is not transparent to the users and provides limited support for GPGPU workloads to efficiently exploit the GPU resources.

In this dissertation, we have researched two GPU resource-related problems in the multitasking environment with GPGPU workloads in current GPU management system and solved them by designing and implementing a GPU task management framework.

In Chapter 3, we first explore the problems that GPU memory cannot be oversubscribed limited the resource usage under GPU multitasking environment. We also found that researches aiming at solving this problem may cause a GPU memory allocation deadlock situation.

Therefore, we proposed a GPU memory checkpoint based approach that can temporarily migrate unused GPU memory contents to the host memory for maximize the GPU memory utilization.

The evaluation results show that our proposed approach can improve the GPU memory utilization in a multitasking environment and efficiently solve the deadlock problems.

In Chapter 4, We explore the problems that intra-GPU resources cannot be fully exploited when multiple workloads share the GPU in parallel. We also found that the current mechanism for GPU multitasking is not perfect yet.

As a result, we proposed a thread block based fine-grained GPU task management framework. Our proposed framework enables GPU multitasking by merging the GPU context, which is different with the current strategy. It also can designate particular thread blocks to specific SMs thus make the GPU task distribution on SMs become controllable.

Experiment results shows that our scheme can improve the intra-GPU resource utilization by support more workloads running in parallel compared to current mechanism and their performance is also guaranteed.

In the future work, we will merge the two separate schemes into a general framework, which can handle both GPU memory and computational resource at the same time. We also plan to design a task scheduling algorithm for handling GPU tasks more efficiently.

Bibliography

[1] Q. Z. Fan, F, "Gpu cluster for high perforamnce computing," in *2004 IEEE/ACM conference on Supercomputing 2004*, p. 47, ACM, 2004.

[2] "Amazon AWS." `https://aws.amazon.com/cn/nvidia/`.

[3] "Microsoft Azure." `https://www.nvidia.com/en-us/data-center/gpu-cloud-computing/microsoft-azure/`.

[4] "Google Cloud." `https://cloud.google.com/gpu`.

[5] "Top 500 list at Nov 2019." `https://www.top500.org/lists/2019/11/`.

[6] "Summit." `https://www.olcf.ornl.gov/summit`.

[7] C. Chen, Y. Du, H. Jiang, K. Zuo, and C. Yang, "Hpcg: Preliminary evaluation and optimization on tianhe-2 cpu-only nodes," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pp. 41–48, Oct 2014.

[8] "HPCG Benchmark." `https://www.hpcg-benchmark.org/`.

[9] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving gpgpu concurrency with elastic kernels," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 407–418, 2013.

[10] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron, "Fine-grained resource sharing for concurrent gpgpu kernels," in *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, (USA), p. 10, USENIX Association, 2012.

[11] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, (USA), p. 1–11, IEEE Computer Society, 2015.

[12] 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, 2014.

[13] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for gpgpu spatial multitasking," in *IEEE International Symposium on High-Performance Comp Architecture*, pp. 1–12, 2012.

[14] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen, "Efficient gpu spatial-temporal multitasking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 748–760, 2015.

[15] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in gpus: Characterization, impact, and mitigation," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 284–295, 2014.

[16] Q. Xu and M. Annavaram, "Pats: Pattern aware scheduling and power gating for gpgpus," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 225–236, 2014.

[17] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 140–149, 2014.

[18] H. Jeon and M. Annavaram, "Warped-dmr: Light-weight error detection for gpgpu," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 37–47, 2012.

[19] "Kubernetes." `https://kubernetes.io`.

[20] "Slurm." `https://www.schedmd.com/index.php`.

[21] "Torque." `http://www.adaptivecomputing.com/products/torque/`.

[22] A. L. D. . A. V. Michael C Schatz, Cole Trapnell, "High-throughput sequence alignment using graphics processing units.," in *BMC Bioinformaticsvolume 8, Article number: 474*, 2007.

[23] J. Gu, S. Song, Y. Li, and H. Luo, "Gaiagpu: Sharing gpus in container clouds," in *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications*, pp. 469–476, Dec 2018.

[24] L. Shi, H. Chen, J. Sun, and K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2012.

[25] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis, "Virtualization for high-performance computing," *SIGOPS Oper. Syst. Rev.*, vol. 40, p. 8–11, Apr. 2006.

[26] D. Kang, "Convgpu: Gpu management middleware in container based virtualized environment.," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2017.

[27] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing Simulation*, pp. 224–231, June 2010.

[28] 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)*, pp. 260–271, IEEE, 2014.

[29] "NVIDIA Hyper-Q." `http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf`.

[30] "NVIDIA MPS." `https://docs.nvidia.com/deploy/mps/index.html`.

[31] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram, "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)*, pp. 230–242, IEEE, 2016.

[32] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared gpu," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 593–606, 2015.

[33] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 119–130, 2015.

[34] T. Allen, X. Feng, and R. Ge, "Slate: Enabling workload-aware efficient multiprocessing for modern gpgpus," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 252–261, May 2019.

[35] "NVIDIA TITAN Xp." `https://www.nvidia.com/en-us/titan/titan-xp/`.

[36] "Rodinia." `https://rodinia.cs.virginia.edu/doku.php`.

[37] B. Varghese, C. Reaño, and F. Silla, "Accelerator virtualization in fog computing: Moving from the cloud to the edge," *IEEE Cloud Computing*, vol. 5, pp. 28–37, Nov 2018.

[38] "NVIDIA CUDA." `https://docs.nvidia.com/cuda/`.

[39] "NVProf." `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`.

[40] M. Bozyigit and M. Wasiq, "User-level process checkpoint and restore for migration," *SIGOPS Oper. Syst. Rev.*, vol. 35, pp. 86–96, Apr. 2001.

[41] "CUDA Context." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#contex`.

[42] "NVIDIA Fermi Architecture." `https://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIA's_Fermi-The_First_Complete_GPU_Architecture.pdf`.

[43] "NVIDIA Kepler Architecture." `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`.

[44] "CUDA stream." https://docs.nvidia.com/cuda/cuda-runtime-api/group_CUDART_STREAM.html.

[45] "NVIDIA Pascal Architecture." https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/.

[46] "Dynamic Parallelism." https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/.

[47] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 358–369, IEEE, 2016.

[48] V. T. Ravi, M. Becchi, W. Jiang, G. Agrawal, and S. Chakradhar, "Scheduling concurrent applications on a cluster of cpu-gpu nodes," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 140–147, May 2012.

[49] J. I. Agulleiro, F. V'zquez, E. M. Garzón, and J. J. Fern'ndez, "Dynamic load scheduling on cpu-gpu for iterative tomographic reconstruction," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 603–608, July 2012.

[50] L. Chen, O. Villa, and G. R. Gao, "Exploring fine-grained task-based execution on multi-gpu systems," in *2011 IEEE International Conference on Cluster Computing*, pp. 386–394, Sep. 2011.

[51] J. J. K. Park, Y. Park, and S. Mahlke, "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, pp. 527–540, 2017.

[52] C. Yu, Y. Bai, H. Yang, K. Cheng, Y. Gu, Z. Luan, and D. Qian, "Smguard: A flexible and fine-grained resource management framework for gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 12, pp. 2849–2862, 2018.

[53] "LD_PRELOAD." `http://man7.org/linux/man-pages/man8/ld.so.8.html`.

[54] "GPU-BLAST." `http://archimedes.cheme.cmu.edu/?q=gpublast`.

[55] "NVIDIA Fermi White Paper." `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[56] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on gpus," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 193–204, 2014.

[57] T. Sorensen, H. Evrard, and A. F. Donaldson, "Cooperative kernels: Gpu multitasking for blocking algorithms," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 431–441, Association for Computing Machinery, 2017.

[58] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 31–44, 2016.

[59] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling effficient preemptive scheduling of gpu," *SIGPLAN Not.*, vol. 52, p. 3–16, Jan. 2017.

[60] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "Effisha: A software framework for enabling effficient preemptive scheduling of gpu," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, (New York, NY, USA), p. 3–16, Association for Computing Machinery, 2017.

[61] K. Gupta, J. A. Stuart, and J. D. Owens, *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.

[62] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 308–317, 2011.

[63] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 353–362, 2016.

[64] G. Chen and X. Shen, "Free launch: Optimizing gpu dynamic kernel launches through thread reuse," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 407–419, 2015.

[65] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications*

Int'l Conference on Cyber, Physical and Social Computing, GREENCOM-CPSCOM '10, (USA), p. 344–350, IEEE Computer Society, 2010.

[66] Y. Yu, W. Xiao, X. He, H. Guo, Y. Wang, and X. Chen, "A stall-aware warp scheduling for dynamically optimizing thread-level parallelism in gpgpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, pp. 15–24, 2015.

[67] Y. Yu, X. He, H. Guo, Y. Wang, and X. Chen, "A credit-based load-balance-aware cta scheduling optimization scheme in gpgpu," *International Journal of Parallel Programming*, vol. 44, no. 1, pp. 109–129, 2016.

[68] B. Wu, X. Liu, X. Zhou, and C. Jiang, "Flep: Enabling flexible and efficient preemption on gpus," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 483–496, 2017.

[69] "NVIDIA Parallel Thread Execution." `ttps://docs.nvidia.com/cuda/parallel-thread-execution/index.html//`.

[70] "NVIDIA NVCC." `https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html`.

[71] R. Panigrahy, K. Talwar, L. Uyeda, and U. Wieder, "Heuristics for vector bin packing." January 2011.

[72] S. S. N. H. e. a. Beheshti Roui, M., "Efficient scheduling of streams on gpgpus.," in *BJ Supercomput (2020). https://doi.org/10.1007/s11227-020-03209-x*, 2020.

요약

최근 범용 GPU (GPGPU) 응용 프로그램은 고성능 컴퓨팅 (HPC) 및 딥 러닝 (DL)과 같은 다양한 연구 분야에서 핵심적인 역할을 수행하고 있다. 이러한 응용 분야의 공통적인 특성은 거대한 계산 성능이 필요한 것이며 그래픽 처리 장치 (GPU)의 높은 병렬 처리 특성과 매우 적합하다. 그러나 GPU 시스템은 특정 유형의 응용 프로그램에 최저화하는 대신 모든 응용 프로그램에 시스템 수준의 공정성을 제공하도록 설계되어 있으며 각 GPGPU 응용 프로그램의 자원 사용 패턴이 다양하기 때문에 단일 응용 프로그램이 GPU 시스템의 리소스를 완전히 활용하여 GPU의 최고 성능을 달성 할 수는 없다.

따라서 GPU 멀티 태스킹은 다양한 리소스 사용 패턴을 가진 여러 응용 프로그램을 함께 배치하여 GPU 리소스를 공유함으로써 GPU 자원 사용률 저하 문제를 해결할 수 있다. 그러나 기존 GPU 멀티 태스킹 기술은 자원 사용률 관점에서 응용 프로그램의 효율적인 실행보다 공동으로 실행하는 데 중점을 둔다. 또한 현재 GPU 멀티 태스킹 기술은 오픈 소스가 아니므로 응용 프로그램과 GPU 시스템이 서로의 기능을 인식하지 못하기 때문에 최적화하기가 더 어려울 수도 있다.

본 논문에서는 응용 프로그램을 수정 없이 GPU 시스템과 GPGPU 응용 사이의 프레임워크를 통해 사용하면 보다 높은 응용성능과 자원 사용을 보일 수 있음을 증명하고자 한다. 그러기 위해 GPU 태스크 관리 프레임워크를 개발하여 GPU 멀티 태스킹 환경에서 발생하는 두 가지 문제를 해결하였다. 첫째, 멀티 태스킹 환경에서 GPU 메모리 초과 할당할 수 없는 문제를 해결하기 위해 호스트 메모리와 디바이스 메모리에 체크포인트 방식을 도입하였다. 둘째, 멀티 태스킹 환경에서 GPU 자원 사용율 저하 문제를 해결하기 위해 더욱 세분화 된 GPU 커널 관리 시스템을 제시하였다.

본 논문에서는 제안한 방법들의 효과를 증명하기 위해 실제 GPU 시스템에

구현하고 그 성능을 평가하였다. 제안한 접근방식이 기존 접근 방식보다 GPGPU 응용 프로그램과 관련된 문제를 해결할 수 있으며 더 높은 성능을 제공할 수 있음을 확인할 수 있었다.

주요어: GPU 시스템, 멀티 테스킹, 메모리 관리, GPU 자원 관리, 체크포인팅
학번: 2011-24087