

Article

Exploring Graphics Processing Unit (GPU) Resource Sharing Efficiency for High Performance Computing

Teng Li *, Vikram K. Narayana and Tarek El-Ghazawi

NSF Center for High-Performance Reconfigurable Computing (CHREC), Department of Electrical and Computer Engineering, The George Washington University, 801 22nd Street NW, Washington, DC, 20052, USA; E-mails: vikramkn@ieee.org (V.N.), tarek@gwu.edu (T.E.-G.)

* Author to whom correspondence should be addressed; E-mail: tengli@gwu.edu, Tel. +1 202-670-3675.

Received: 20 September 2013; in revised form: 23 October 2013 / Accepted: 31 October 2013 / Published: 19 November 2013

Abstract: The increasing incorporation of Graphics Processing Units (GPUs) as accelerators has been one of the forefront High Performance Computing (HPC) trends and provides unprecedented performance; however, the prevalent adoption of the Single-Program Multiple-Data (SPMD) programming model brings with it challenges of resource underutilization. In other words, under SPMD, every CPU needs GPU capability available to it. However, since CPUs generally outnumber GPUs, the asymmetric resource distribution gives rise to overall computing resource underutilization. In this paper, we propose to efficiently share the GPU under SPMD and formally define a series of GPU sharing scenarios. We provide performance-modeling analysis for each sharing scenario with accurate experimentation validation. With the modeling basis, we further conduct experimental studies to explore potential GPU sharing efficiency improvements from multiple perspectives. Both further theoretical and experimental GPU sharing performance analysis and results are presented. Our results not only demonstrate the significant performance gain for SPMD programs with the proposed efficient GPU sharing, but also the further improved sharing efficiency with the optimization techniques based on our accurate modeling.

Keywords: GPU; resource sharing; SPMD; performance modeling; high performance computing

1. Introduction

Recent years have seen the proliferation of Graphics Processing Units (GPUs) as application accelerators in High Performance Computing (HPC) Systems, due to the rapid advancements in graphic processing technology over the past few years and the introduction of programmable processors in GPUs, which is also known as GPGPU or General-Purpose Computation on Graphic Processing Units [1]. As a result, a wide range of HPC systems have incorporated GPUs to accelerate applications by utilizing the unprecedented floating point performance and massively parallel processor architectures of modern GPUs, which can achieve unparalleled floating point performance in terms of FLOPS (Floating-point Operations Per Second) up to the TeraFLOP (Tera Floating-point Operations Per Second) barrier [2,3]. Such systems range from clusters of compute nodes to parallel supercomputers. While computer clusters used by academia are increasingly equipped with GPU co-processors for accelerating applications [4,5], the contemporary offerings from supercomputer vendors have begun to incorporate professional GPU computing cards into the compute blades of their parallel computer products; examples include the latest Cray XK7 [6], Cray XK6 [7] and SGI (Silicon Graphics, Inc.) Altix UV [8] supercomputers. Yet, more notably, one of the most powerful supercomputers currently ranking second in the world [9], Titan [10], is equipped with 18,688 NVIDIA GPUs and thereby able to deliver a sustained 17.59 PFLOPS (Peta Floating-point Operations per Second) LINPACK (LINear equations software PACKage) performance [9].

Development of parallel applications for any of these GPU-based heterogeneous HPC systems requires the use of parallel programming techniques composed of both CPU and GPU parallel programs to fully utilize the computing resources. Among the different parallel programming approaches, the most commonly followed programming approach is the Single-Program Multiple-Data (SPMD) model [11]. Under the SPMD scenario, multiple processes execute the same program on different CPU cores, simultaneously operating on different data sets in parallel. Techniques of message passing, such as MPI (Message Passing Interface) [12] are often deployed to achieve the SPMD parallelism effectively with the required inter-processor communication. By allowing autonomous execution of processes at independent points of the same program, SPMD serves as a convenient, yet powerful, approach for efficiently making use of the available hardware parallelism.

With the introduction of hardware accelerators, such as GPUs, as co-processors, HPC systems are exhibiting an architectural heterogeneity that has given rise to programming challenges not previously existing in traditional homogeneous parallel computing platforms. With the SPMD approach used for programming most of the homogeneous parallel architectures, directly offloading the program instances on to GPUs is not feasible, due to the different Instruction Set Architectures (ISAs) of CPUs and GPUs. Moreover, GPUs are primarily suited for the compute-intensive portions of the program, serving as co-processors to the CPUs in order to accelerate these sections of the parallel program. The “single program” requirement of SPMD, therefore, means that every program instance running on the CPUs must necessarily have access to a GPU accelerator. In other words, it is necessary to maintain a one-to-one correspondence between CPUs and GPUs, that is, the number of CPU cores must equal the number of GPUs. However, due to the proliferation of many-core microprocessors in HPC systems, the number of CPU cores generally exceeds the number of GPUs, which is also true for all four

GPU-based supercomputers in the top 30 list, as shown in Table 1 [9,10,13–15]. Therefore, the problem of system computing resource underutilization with the SPMD approach is general across GPU-based heterogeneous platforms and urgently needs to be solved with the increasing number of CPU cores in a single CPU, due to the fast advancement of multi-/many core technologies.

Table 1. Graphics Processing Unit (GPU)-based supercomputers in the top 30 List.

Supercomputer (Ranking)	# of CPU Cores	# of GPUs	CPU/GPU Ratio
Titan (2 nd)	299,008	18,688	16
Tianhe-1A (10 th)	102,400	7,168	14.3
Nabulae (16 th)	55,680	4,640	12
Tsubame2.0 (21 st)	17,984	4,258	4.2

However, even though the number of physical GPUs cannot match the number of CPU cores found in contemporary HPC systems, modern high-end GPU architecture is designed as massively parallel and composed of up to thousands of processing cores [3]. Moreover, since GPU programs are composed of parallel threads executed in parallel on these many processing cores physically, it is possible to achieve the execution concurrency of multiple GPU programs on the single GPU. For example, the NVIDIA Fermi [16] architecture consisting of up to 512 Streaming Processor (SP) cores allows the concurrent execution of up to 16 GPU kernels [17]. The increasing parallel computation capabilities of modern GPUs enable the possibility of sharing a single GPU to compute different applications or multiple instances of the same application, especially when the application problem size and parallelism is significantly smaller than the inherent parallelism capacity of the GPU.

In this paper, targeting the problem of resource underutilization under the SPMD model, we propose to efficiently share the GPU resources among the microprocessors through streaming for modern heterogeneous systems. Our proposed sharing approach is to provide multiple folds of execution concurrencies. The sharing of a single GPU from a SPMD parallel program can be further divided into process-level and thread-level sharing, depending on the SPMD program. Note that here, we are primarily focused on the SPMD programs for which all processes/threads carry out identical GPU tasks. Therefore, our further discussions related to the SPMD programs are concentrated on identical GPU kernels from multiple processes/threads. For process-level GPU sharing, our previous work [18] provided a GPU virtualization approach to eliminate Fermi [16] or earlier GPUs' sharing inefficiency among multiple processes by providing a virtualization layer to the processes, while all GPU kernels from SPMD processes are launched from a single daemon process (virtualization layer) through Compute Unified Device Architecture (CUDA) streams [17] to achieve concurrent kernel execution and kernel concurrency with GPU I/O (Input/Output). In other words, Fermi or earlier series of GPUs do not natively support inter-process concurrencies, since multiple GPU contexts [17] are created for processes and current execution features can only happen within a single GPU context. Only till very recently, the NVIDIA Kepler series of GPUs [19] provide Hyper-Q hardware support along with the CUDA proxy feature, which allows multiple processes to share a single GPU context. Thus, in order to achieve the efficient process-level GPU sharing for the Kepler series of GPUs, programmers can utilize Hyper-Q and CUDA proxy support directly. For thread-level GPU sharing, the latest CUDA release [17] provides

necessary supports to share a single GPU context among multiple threads for both Fermi and Kepler series of GPUs. Thus, thread-level GPU sharing can be directly achieved by using streaming execution. Therefore, from an overall perspective, to achieve GPU sharing, our proposed sharing approach is to launch multiple GPU kernels from multi-processes/threads using CUDA streaming execution within a single GPU context, while the single context requirement is met by launching kernels from a single process, such as our virtualization implementation.

By describing the efficient GPU sharing approach, in this paper, we provide both theoretical and experimental analysis of the proposed GPU sharing. On the theoretical side, we formally define that multiple identical GPU kernels from a SPMD program can share the GPU under four GPU sharing scenarios (depending upon the kernel profiles): Exclusive Space, Non-Exclusive Space, Space/Time and Time Sharing. Along with the defined scenarios, we propose a series GPU sharing execution models as our analytical basis to analyze possible performance improvements under the four sharing scenarios. On the experimentation side, we utilize multiple GPU benchmarks to verify the accuracy of the proposed modeling. We further employ multiple benchmarks to study the performance characteristics of each sharing scenario. As a further step, we perform several experimental studies on exploiting possible sharing efficiency improvements based on these performance characteristic studies and modeling analysis. Multiple perspectives of optimization are being considered for different sharing scenarios, ranging from the problem/kernel size and parallelisms of the SPMD program to optimizable sharing scenarios. Based on these factors, we provide experimental optimization analysis and achieve an optimized I/O concurrency for kernels under Time Sharing, a better Streaming Multiprocessor (SM) utilization for kernels under Exclusive Space Sharing and an optimized parallelism strategy for parallel programs. We also propose the concept of Sharing Scenario Casting as a means to switch the possible SPMD sharing scenario with improved performance, as well as a general optimization guideline with our sharing scenario analysis. Our further experimental results also demonstrate the significantly achievable performance advantage with efficient GPU sharing through streaming execution, along with the fine-grained GPU sharing modeling analysis.

The rest of this paper is organized as follows. Section 2 provides an overview of related work on GPU resource sharing in heterogeneous HPC systems and related fields. A background of the GPU architectural and programming model, as well as the GPU kernel execution flow is given in Section 3, followed by a formal definition of the proposed GPU sharing scenarios and the analytical GPU sharing execution model in Section 4. A series of experimental analysis and results, as well as performance optimization strategies, are presented and discussed in Section 5, which is followed by our conclusion in Section 6.

2. Related Work

There are several streams of research endeavors targeting GPU resource sharing within the HPC field. One direction of such research has focused on the problem of GPU device underutilization. Guevara *et al.* [20] proposed a GPU sharing approach that allows GPU requests from multiple processes to be intercepted at run-time. They also introduced a GPU kernel merging approach to merge two kernels into a single kernel for reduced sharing overheads. By using the similar kernel merging approach

to reduce GPU resource underutilization, Saba *et al.* [21] presented an algorithm that allocates GPU resources for tasks based on the resource goals and workload size. While they target a different problem of a time bound algorithm that optimizes the execution path and output quality, the employed GPU sharing approach is to merge the kernels similarly. Although the kernel merging approach is useful to increase the GPU resource utilization in the scenario of GPU sharing, it needs compiler-level support with merged kernel generation. Moreover, with merged kernels, multiple kernels are “invoked” simultaneously; therefore, it does not exploit the possibility to hide data transfer overhead with kernel execution, which can be achieved through concurrent kernel execution and data transfer on the current GPU architecture. Moreover, the latest GPU architecture further provides concurrent kernel execution support as a straightforward approach to eliminate possible resource underutilization, which makes the kernel-merging approach currently unnecessary.

Two other categories of solutions for GPU sharing target two different levels of GPU sharing. One focuses on thread-level GPU sharing. Peters *et al.* [22] proposed a method to share a single NVIDIA GPU device within a GPU compute node among multiple host threads. They provide a persistent kernel that has been initialized and executing continuously under a single GPU process (context). Meanwhile, the host thread acts as a management thread to accept requests from other threads, with the requested GPU functions essentially using a single thread-block on the GPU. While devices that do not support concurrent kernel execution can also benefit from kernel concurrencies with this solution, the persistent kernel limits the total blocks to be executed, and the programming efforts to fit the application into the single-block size is also another limitation. The other category of solutions is to provide process-level GPU sharing. Our previous work [18] presented a GPU virtualization infrastructure that provides a virtual SPMD model by exposing multiple virtual GPU interfaces to the processors. The virtualization infrastructure allows for multiple processes to share the GPU using a single GPU context and to concurrently execute GPU kernels, as well as to achieve concurrency between data transfer and kernel execution. This is achieved by using a centralized running process, known as the Virtualization Manager (VM), to take requests and allocate GPU resources, as well as to provide synchronizations. Another example is described by the S_GPU project [23], which uses a custom software stack to ensure the time-sharing of the GPU between MPI processes. In their implementation, each MPI process inserts GPU commands into a stream object, and when the process initiates the execution of the stream, all the enqueued GPU commands are executed in the desired sequence. Although S_GPU may have drawbacks in terms of always time-sharing the GPU and GPU context-switch overheads, their work is complementary to our previous work and may be combined as a general process-level GPU sharing implementation approach. In addition, our previous work in [24] preliminarily models the GPU sharing, which lays the analytical basis for this paper.

Another direction of research on GPU resource sharing has focused on efficient GPU resource sharing within the cloud/virtual machine environment. Ravi *et al.* [25] proposed a GPU sharing framework for the cloud environment, based on that they provided a GPU kernel consolidation algorithm to combine kernel workloads within the cloud and, thus, achieved improved throughput, while considering both GPU space and time sharing, from a coarse and general perspective. However, their analysis only considered a limited case of GPU sharing when two kernels are present. Thus, further sharing efficiency cannot be achieved unless a detailed device-level GPU sharing analysis is conducted and more kernels are

considered. Similarly, [26–28] also focused on approaches to provide access to GPU accelerators within virtual machines. Nevertheless, these approaches only enable time sharing of the GPU at a coarse level of granularity, while incurring context-switch overheads among processes, due to the native process-level sharing approach. Moreover, they are not particularly suited for GPU sharing within SPMD programs, since it would require the use of a dedicated virtual machine for every SPMD process within a multi-core node, leading to significant overheads.

3. Background of GPU Computing

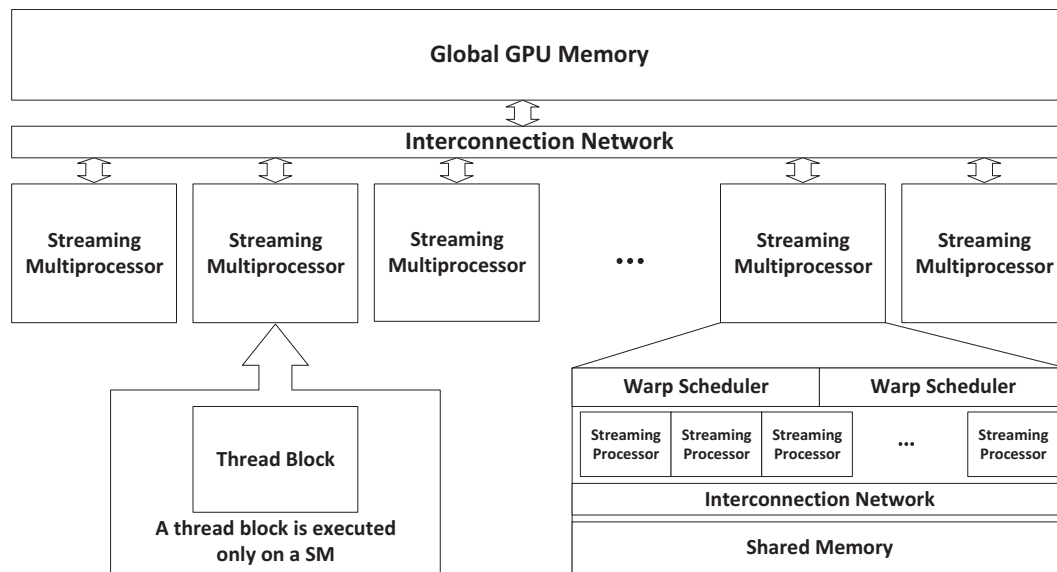
In this section, we present a general overview of the GPU computing architecture, on which our further analysis will be based. We will describe both the GPU programming model and the architectural model of current GPU technology. Some advanced features and the device-level execution flow of modern GPUs are also discussed to support our further analysis.

3.1. Programming Models

The current two prevalent GPU programming models are Compute Unified Device Architecture (CUDA) [17], developed by NVIDIA, and Open Computing Language (OpenCL) [29], provided by Khronos Working Group. For convenience, we use CUDA terms in the rest of this paper. Currently, both models follow a master-slave flow, under which the CPU master process sends input data to GPU device memory, launches the GPU kernel, waits for the kernel to finish and retrieves the output data. The kernel executed on the GPU follows Single-Instruction, Multiple-Thread (SIMT) to achieve the parallelism. Threads are the lightest execution unit of the kernel function. Each GPU kernel is launched per grid, which can be decomposed into thread blocks consisting of a number of threads. The provided three levels of thread hierarchies require the programmer to write the kernel for a single thread and specify the total number of parallel threads. The thread hierarchies are then decided based on the data and resource considerations. Note that, here, we are mainly concentrating on the programming styles and features provided by CUDA. Therefore, we will use the CUDA terminology for the rest of the paper.

3.2. An Architectural Model

Modern GPUs are composed of massively parallel processing units and hierarchies of memories. Figure 1 shows a top-down architectural overview of the NVIDIA Fermi [16] GPUs. The GPU is composed of 16 Streaming Multiprocessors (SMs) as the first processing unit hierarchy, each of which is further composed of 32 Streaming Processors (SPs) as the second hierarchy. For the memory hierarchies, all SMs on chip share the global device memory, and each SM has its private shared memory, which is shared by all SPs within each SM. Each SP holds a number of registers for each thread. Within each SM, there are two thread schedulers, each of which schedules a warp (a batch of 32 threads) at one time to be executed on the SPs. Multiple warps are allowed to co-exist on the same SM. Especially when threads have high memory access latency, increasing the warp occupancy by having multiple warps co-exist on a single SM simultaneously can improve the overall execution performance.

Figure 1. A GPU architecture overview: Fermi.

3.3. GPU Device-Level Execution Flow

When a kernel is launched, each block can only be executed on a single SM. Multiple blocks can reside on the same SM only if resources permit, and the resources include the shared memory usage, the register usage of each thread and the total number of warps within the block. Since each SM has a limited number of registers, a fixed size of shared memory and a maximum number of warps that can co-exist and be scheduled, only multiple blocks meeting these constraints can be scheduled within a single SM. However, blocks are only limited to a single kernel. In other words, blocks from different kernels cannot be concurrently scheduled in conventional GPU devices. Nevertheless, current CUDA devices with a computing capability higher than 2.0 (Fermi) support concurrent kernel execution, which allows different kernels to be launched from the same process (GPU program) using CUDA streams. Concurrent kernel execution allows blocks from different kernels to be scheduled simultaneously. Furthermore, by using asynchronous CUDA streams, concurrent data transfer and kernel execution can also be achieved among multiple streams, each of which carries a GPU kernel. Note, however, that GPU kernels launched from independent CPU processes cannot be concurrently executed, since each process creates its own GPU context, and CUDA kernels from a different GPU context cannot achieve execution concurrency. In other words, to efficiently execute multiple GPU kernels on a single GPU, kernels should be launched within the same GPU context within a single process.

4. GPU Sharing Scenarios

4.1. GPU Sharing Approach with Streams for SPMD Programs

For a given SPMD program, the program parallelism can be expressed at two different levels: process-level and thread-level. In other words, while the SPMD program can be written using process-level parallelism, such as MPI [12], or thread-level parallelism, such as OpenMP (Open

Multi-Processing) [30], each process or thread from the given SPMD program composed of both CPU and GPU codes needs access from the GPU, and thus shares the GPU on the process/thread level. In this paper, for both process/thread-level GPU sharing, we propose an efficient sharing approach utilizing the advanced concurrency features of modern GPUs through streaming execution. In other words, as explained earlier, modern GPUs can execute multiple GPU kernels simultaneously while establishing concurrent kernel execution and GPU I/O transfer between different kernel executions. These features can thus be utilized by SPMD programs executing many identical GPU kernels.

Currently, concurrent kernel execution support from CUDA specifically on Fermi GPU or newer families uses CUDA streams to achieve kernel parallelism, with each kernel being launched through a CUDA stream. However, as we mentioned earlier, in order to achieve the kernel-level concurrency, all kernels (streams) need to be launched within the same process (GPU context), since each process accessing the GPU creates its own GPU context. For a SPMD parallel program to efficiently share the GPU, multiple processes or threads thus need to be able to establish the required inter-process/thread kernel execution concurrency using CUDA streams, depending on the level (process/thread) at which the SPMD program is written.

When a SPMD program is composed of parallel processes, each of which carries a GPU kernel execution, the GPU is shared at the process-level. Since multiple processes create multiple GPU contexts, kernel execution cannot natively achieve concurrency among processes for Fermi or earlier GPUs [16]. As we explained earlier, the latest Kepler GPU architecture [19] provides native hardware Hyper-Q support, which allows multiple processes to share a single GPU context using the CUDA proxy server feature. As the purposes of utilizing our GPU virtualization approach (for Fermi or earlier GPUs), as well as using the Hyper-Q feature (for Kepler series of GPUs) are both to meet the single GPU context requirement for efficient GPU sharing, here, we provide a brief description of the GPU virtualization approach addressed by our previous work [18]. Figure 2 shows that all SPMD GPU kernels are executed within the single daemon process using CUDA streams. In other words, since each SPMD GPU kernel is launched with a separate CUDA stream, improved sharing efficiency can be achieved, due to possible execution concurrency among kernels. The GPU virtualization solution also provides the programmers with simple APIs (Application Programming Interfaces) to launch the GPU kernel within each process to realize efficient GPU sharing.

On the other hand, when an SPMD program is composed of parallel threads, each of which carries a GPU kernel execution, the GPU is shared at the thread level. For NVIDIA Fermi or earlier GPUs, different CUDA versions provide varied thread-level GPU sharing support. Before CUDA 4.0, similar to process-level sharing, each thread launching a GPU kernel created its own GPU context with no possible inter-thread kernel execution concurrency. CUDA 4.0 and later versions [17] provide further improved multi-threading concurrency support for Fermi or earlier GPUs, as well as Kepler GPUs. The improved thread-level concurrency support allows multiple threads to share the single GPU context created for the multi-threaded SPMD program launching process, as shown in Figure 3. Thus, as long as each thread within the SPMD program launches its kernel with the CUDA stream natively, efficient GPU sharing (inter-thread concurrency) can be achieved.

Figure 2. Efficient Single-Program Multiple-Data (SPMD) process-level GPU sharing.

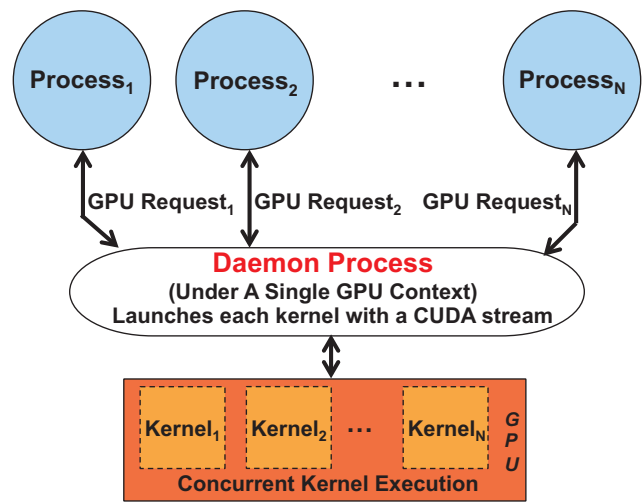


Figure 3. Efficient SPMD thread-level GPU sharing.

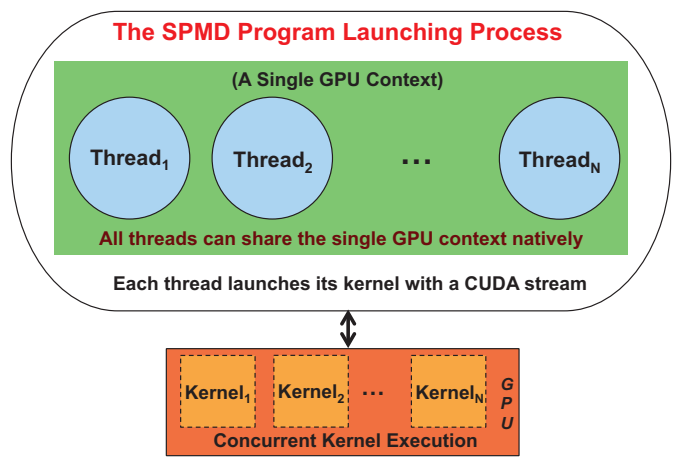
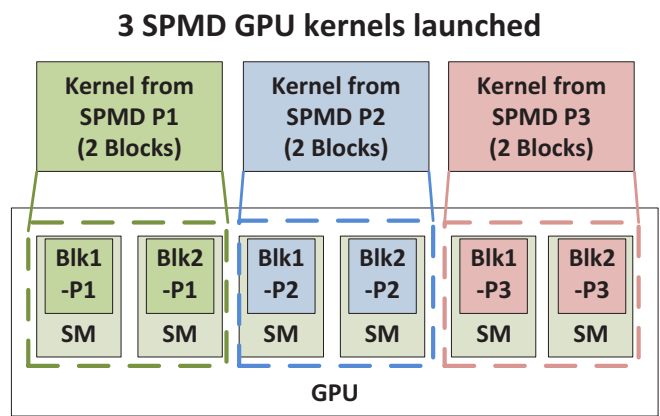


Figure 4. Exclusive Space Sharing scenario.



Therefore, for a given SPMD program to efficiently share the GPU, our sharing approach utilizes CUDA streams to execute the GPU kernel from each process/thread in general. The achievable GPU

sharing efficiency from the SPMD program is specifically determined by the streaming execution efficiency of multiple kernels. Therefore, our further analysis first studies the device-level multi-kernel behavior as the modeling basis when SPMD kernels are all simultaneously executed through CUDA streams.

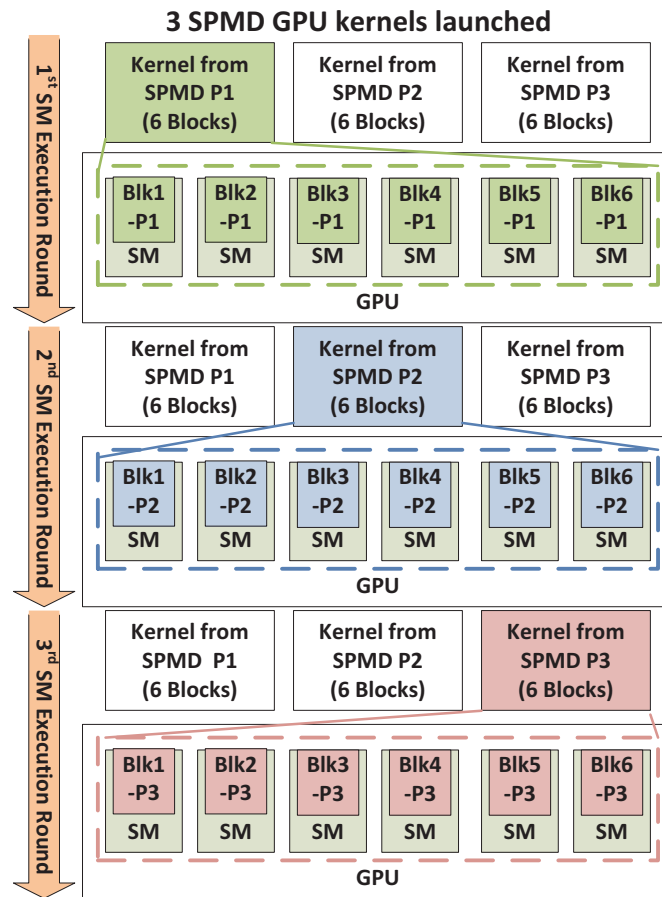
4.2. GPU Sharing Scenarios

In this section, we analyze several scenarios when multiple GPU kernels invoked from CUDA streams share a single, physical GPU. The GPU kernels are identical in function and launched simultaneously, but they operate on different data. In order to study the different scenarios, we analyze them based on the manner in which the kernels utilize the underlying processing cores (SMs) in the GPU, through space-multiplexing, time-multiplexing or hybrid space-time multiplexing [31]. Our analysis for each scenario is based on the device-level multi-kernel concurrency behaviors. As we shall see shortly, the occurrence of different sharing scenarios depends on the profile characteristics of the SPMD GPU kernels.

The primary goal of our analysis is to provide a general and formalized GPU space/time sharing boundary for different SPMD GPU applications, which will, in turn, aid us in estimating the possible performance gains that can be achieved using our sharing approach. Of more significance, insights gained from our analyses are also to be used for fine-tuning the applications to achieve better GPU sharing performance.

As previously described, a typical GPU is composed of multiple SMs, which execute multiple thread blocks. When multiple kernels are simultaneously launched with CUDA streams, blocks from all kernels are to be scheduled for execution on SMs. Based on how blocks from different kernels occupy the SMs, we expect the following four GPU sharing scenarios: Exclusive Space Sharing, Non-Exclusive Space Sharing, Space/Time Sharing and Time Sharing. These terms will be explained shortly. We illustrate the four sharing scenarios by using a simple example GPU that has six SMs that is shared by three SPMD Parallel processes/threads (represented as SPMD_Ps - SPMD_P1; SPMD_P2 and SPMD_P3), which are demonstrated in Figure 4 to Figure 5. In order to generalize the required condition for each sharing scenario, we define several parameters, as shown in Table 2.

When a parallel SPMD program launches multiple, identical GPU kernels to share a single GPU, all kernels are executed simultaneously through CUDA streams within a single GPU context. Kernels are composed of thread blocks, and every thread block can execute only on one SM, as previously explained. We further assume that the GPU hardware assigns all thread blocks to free SMs until every SM is occupied, before assigning additional thread blocks to an SM. Based on this assumption, if the total number of thread blocks from all SPMD kernels does not exceed the number of available SMs (N_{SM}), kernels will execute on independent SMs, resulting in a Space-Sharing scenario.

Figure 5. Time Sharing scenario.**Table 2.** Parameters defined for GPU sharing scenarios.

Symbol	Definition
N_{SM}	# of SMs in the GPU
$N_{regs_per_SM}$	# of registers in each SM
$S_{shm_per_SM}$	The size of shared memory in an SM
$N_{thds_per_warp}$	# of threads per warp
$N_{max_warps_SM}$	The maximum # of warps allowed in an SM
$N_{thds_per_blk}$	# of threads per block
$N_{regs_per_thd}$	# of registers per thread
$N_{regs_per_blk}$	$N_{thds_per_blk} \times N_{regs_per_thd}$
$S_{shm_per_blk}$	The size of shared memory per block
$N_{warps_per_blk}$	$N_{thds_per_blk} / N_{thds_per_warp}$
$N_{blks_per_SM}$	# of blocks per SM within one SM round
$N_{max_blks_per_SM}$	The maximum # of blocks per SM
$N_{blks_per_knl}$	# of blocks per SPMD kernel
N_{SPMD_P}	# of SPMD parallel processes/threads sharing the GPU

4.2.1. Exclusive Space Sharing

As shown in Figure 4, as an example, each SPMD kernel has two blocks of threads, and the total of six blocks from all three kernels are launched on six available SMs, respectively, under concurrent kernel execution. Thus, kernels from different SPMD_Ps can co-exist on the GPU and be processed by different SMs simultaneously. We term this Exclusive Space Sharing, which will occur under the following condition.

Exclusive Space Sharing Condition.

$$N_{blks_per_knl} \times N_{SPMD_P} \leq N_{SM} \quad (1)$$

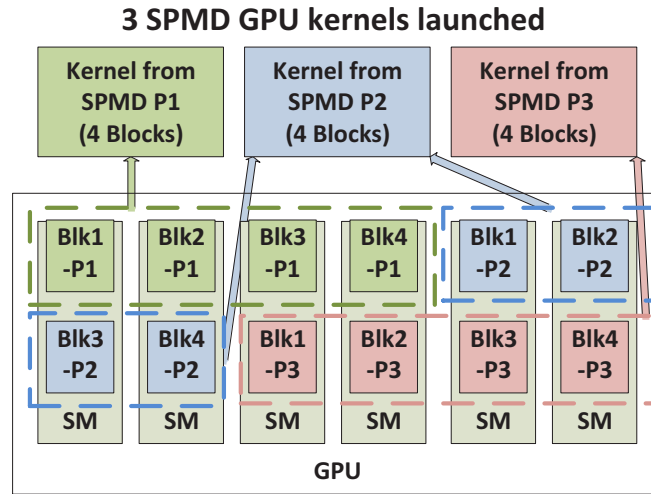
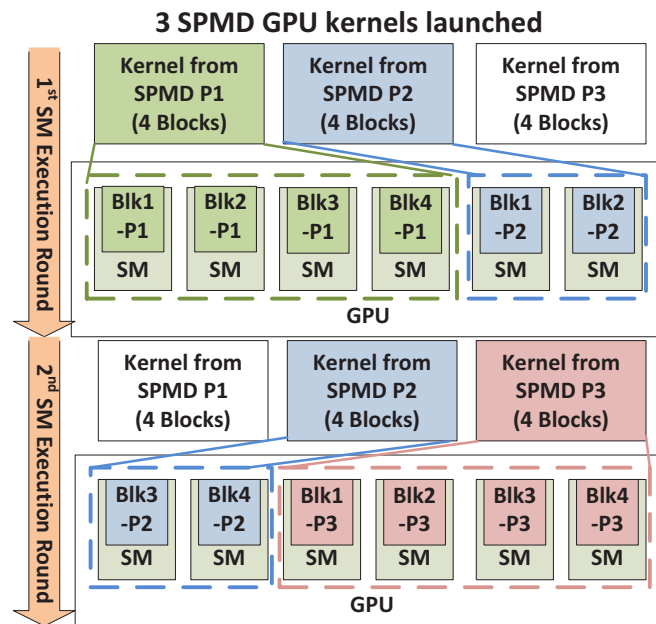
4.2.2. Non-Exclusive Space Sharing

If each GPU kernel requires a larger number of thread blocks and consequently does not satisfy condition (1), we can no longer have exclusive sharing of the SMs by GPU kernels. Instead, more than one thread block (from different kernels) will be assigned to an SM. In other words, warps from thread blocks belonging to different kernels will execute on the same SM in an interleaved fashion. Note that warps from the same or different thread blocks always execute sequentially within an SM. However, switching between warps is very fast and often aids execution by allowing memory access latencies from one warp to be hidden by the execution of another warp [16]. Hence, the scenario considered here qualifies as space-sharing. Nevertheless, each SM is not exclusively used by one kernel; we therefore term this case Non-Exclusive Space Sharing.

In this scenario, the number of thread blocks mapped to a single SM depends on the availability of resources. Specifically, it is limited by three factors: (a) the register usage of all blocks mapped to an SM must not exceed the available registers within the SM, $N_{regs_per_SM}$; (b) the shared memory usage of all blocks mapped to the SM must not exceed the available shared memory in the SM, $S_{shm_per_SM}$; and (c) the total number of warps mapped to an SM must not exceed the warp capacity of the SM, $N_{max_warps_SM}$. The influence of these three factors is captured by the following expression for the number of blocks that can be mapped to an SM:

$$N_{max_blks_per_SM} = \text{Minimum} \left(\left\lfloor \frac{N_{regs_per_SM}}{N_{regs_per_blk}} \right\rfloor, \left\lfloor \frac{S_{shm_per_SM}}{S_{shm_per_blk}} \right\rfloor, \left\lfloor \frac{N_{max_warps_SM}}{N_{warps_per_blk}} \right\rfloor \right) \quad (2)$$

The Non-Exclusive Space Sharing scenario described here occurs only when the number of blocks that can be mapped to an SM is greater than unity, that is, $N_{max_blks_per_SM} > 1$. On the other hand, if only one block can be mapped to an SM based on (2), then the SM will have to be time-shared between thread blocks, which is a different scenario described later. An illustration of non-exclusive space sharing is given in Figure 6. After blocks from SPMD_P1 fill four SMs, only two blocks from SPMD_P2 can be scheduled on the remaining two free SMs. Since all SMs still have room for more blocks, the remaining two blocks from SPMD_P2 and all blocks from SPMD_P3 can be scheduled to co-execute on the same six SMs.

Figure 6. Non-Exclusive Space Sharing scenario.**Figure 7.** Space/Time Sharing scenario.

In addition to supporting more than one block per SM, the Non-Exclusive Space Sharing case occurs only if the number of thread blocks from all kernels is large enough so that it violates (1). However, in order to ensure that an excessively large number of blocks does not force time-sharing, the total number of thread blocks from all kernels needs to be limited. This limiting number is the maximum number of thread blocks that can be handled by all SMs taken together, $N_{max.blks_per_SM} \times N_{SM}$. These conditions may be summarized as follows.

Non-Exclusive Space Sharing Conditions.

$$N_{max.blks_per_SM} > 1 \quad (3)$$

$$N_{SM} < N_{blks_per_knl} N_{SPMD_P} \leq N_{max.blks_per_SM} N_{SM} \quad (4)$$

4.2.3. Space/Time Sharing

Irrespective of the number of thread blocks supported by an SM, if the total number of thread blocks is so large that it exceeds the RHS (Right Hand Side) of (4), the available SMs will have to be time-shared through multiple rounds of SM executions. As an example, consider the scenario in Figure 7, which uses $N_{max.blks.per.SM} = 1$ for simplicity.

Here, the first four blocks from SPMD_P1 occupy four SMs and leave two SMs available for other SPMD_Ps in the first execution round. Only two blocks from SPMD_P2 can therefore execute and space-share the GPU with SPMD_P1. Meanwhile, the other two blocks from SPMD_P2 have to be executed in the second round, thus time-sharing the GPU with blocks from SPMD_P1. We thus observe that within an execution round, there is space sharing between different kernels, and across multiple execution rounds, time sharing occurs. As a result, both space sharing and time sharing co-exist, and we call this scenario the Space/Time Sharing scenario. Note that the space-sharing that is exhibited within an execution round may be exclusive or non-exclusive, as described earlier.

The conditions for Space/Time Sharing may be listed as below. Firstly, the total number of thread blocks must be large enough to violate (4). Secondly, the number of thread blocks from a single kernel must be smaller than the total block capacity $N_{max.blks.per.SM} \times N_{SM}$; otherwise, one kernel occupies the entire execution round, which indicates that time-sharing predominates between different kernels, and we classify that accordingly under time-sharing. The two conditions for space/time sharing are summarized as the following.

Space/Time Sharing Conditions.

$$N_{blks.per.knl} N_{SPMD.P} > N_{max.blks.per.SM} N_{SM} \quad (5)$$

$$N_{blks.per.knl} < N_{max.blks.per.SM} N_{SM} \quad (6)$$

4.2.4. Time Sharing

We classify the execution to be in a Time-Sharing scenario when (a) multiple execution rounds are required to process all kernels, as exemplified by (5) and (b) the number of thread blocks within a single kernel is large enough to occupy at least one execution round. In other words, the number of blocks in a single kernel must exceed the single-round thread block capacity of the GPU, $N_{max.blks.per.SM} \times N_{SM}$. These conditions may be summarized as follows.

Time Sharing Conditions.

$$N_{blks.per.knl} N_{SPMD.P} > N_{max.blks.per.SM} N_{SM} \quad (7)$$

$$N_{blks.per.knl} \geq N_{max.blks.per.SM} N_{SM} \quad (8)$$

An illustration is provided in Figure 5, with $N_{max.blks.per.SM} = 1$, $N_{blks.per.knl} = 6$ and $N_{SM} = 6$. The six blocks in each kernel fully occupy six SMs, which forces the three kernels to be executed sequentially in

separate rounds. Note that if this example had seven blocks for every kernel, there would be some space sharing occurring between SPMD_P1 and SPMD_P2 during the second execution round. However, for simplicity, we classify this under Time Sharing, because the number of execution rounds is at minimum equal to the number of N_{SPMD_P} , and a large kernel generally executes many rounds.

Note that our analysis of the four sharing scenarios only considers the execution phases of the kernels, without consideration of data transfers between the CPU and the GPU. Accurate performance estimates can be achieved only when the I/O transfers are taken into account for all the SPMD_Ps. Using the analysis of different sharing scenarios as the foundation, we build the SPMD-based GPU sharing and execution model as described next.

4.3. GPU Sharing and Execution Model

As an analytical basis, we model the execution of a GPU program to consist of the following stages: GPU device initialization; transferring input data into GPU memory; executing the GPU kernel and transferring the result data to the main memory, as shown in Figure 8. In addition, we define the necessary analytical parameters in Table 3.

Figure 8. GPU execution stages.

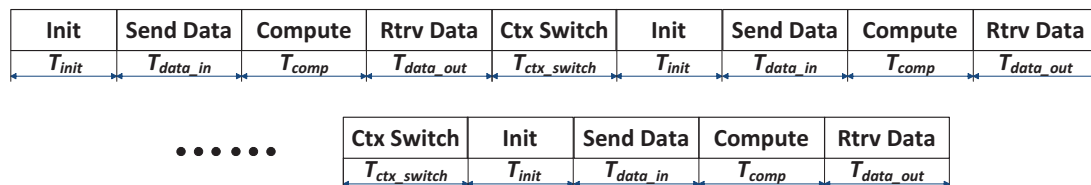
Initialization	Send Data	Compute	Retrieve Data
T_{init}	T_{data_in}	T_{comp}	T_{data_out}

As the first step, we set our performance evaluation baseline as the native GPU sharing, wherein multiple processes share the GPU in a sequential manner, while process-level sharing incurs context-switch overhead. This is because each process creates its GPU context, and there is no concurrency possible among multiple GPU contexts. Here, we are primarily modeling the process-level native sharing baseline, since thread-level sharing can be achieved under a single GPU context natively with the latest CUDA support. Thus, for process-level native sharing, as shown in Figure 9, we model an average context-switch overhead by switching from one process to another, as well as an average GPU initialization overhead, due to the fact that the GPU resource needs to be initialized in each new context. With this execution model, we derive the T_{native_sh} accordingly, as shown in (9):

Table 3. Parameters defined for analytical modeling.

Symbol	Definition
T_{data_in}	The time to transfer input data into the GPU memory
T_{data_out}	The time to transfer the result data into the main memory
T_{comp}	The time for the GPU kernel computation
T_{init}	The time overhead for the GPU resource to be initialized
T_{ctx_switch}	The average context-switch overhead between processes (process-level)
T_{native_sh}	The total time to execute all SPMD process using the native sharing approach
T_{sh_sce}	The total execution time for a given sharing scenario
T_{ex_sp}	The total execution time for Exclusive Space Sharing
$T_{n_ex_sp}$	The total execution time for Non-Exclusive Space Sharing
T_{sp_tm}	The total execution time for Space/Time Sharing
T_{tm}	The total execution time for Time Sharing
$T_{tm_io_i}$	The total execution time for I/O-intensive applications
T_{SM_str}	The SM time stretch of adding a block per SM
$T_{SM_str}(N_{blks_per_SM})$	The SM time stretch of adding a block per SM with current $N_{blks_per_SM}$
R_{SM}	The total number of SM execution rounds
$T_{full_rnd_str}$	The time stretch of one full SM execution round
$T_{fs_rnd_str}$	The stretch of adding the first SM execution round to the full one
$T_{ls_rnd_str}$	The stretch of the last SM execution round (may not be full)
$T(N_{blks_per_SM})$	The execution time when there are $N_{blks_per_SM}$ blocks per SM

$$T_{native_sh} = N_{SPMD_P}(T_{init} + T_{data_in} + T_{comp} + T_{data_out}) + (N_{SPMD_P} - 1)T_{ctx_switch} \quad (9)$$

Figure 9. The native sequential GPU sharing (process level).

Different from the native sharing approach, our sharing approach through streaming execution achieves inter-process parallelism using CUDA streams. Three types of overlappings are possible: (a) the overlapping of the execution of multiple concurrent kernels; (b) the overlapping of kernel execution with either the GPU input or output data transfer; and (c) the overlapping of the input data transfer with the output data transfer. These possible overlappings are to be employed in our further modeling. However, the latest release of CUDA supports streams being programmed in two styles aimed at achieving either kernel execution concurrency or I/O concurrency [17]. When targeting I/O concurrency (Programming

Style-2 or PS-2), both input and output I/O can be inter-overlapped and also overlapped with kernel execution, although the execution phases of different kernels cannot overlap. On the other hand, if streams are programmed to achieve kernel concurrency (Programming Style-1 or PS-1), the output data transfer can overlap with only a very small portion of kernel execution, while concurrent execution of multiple kernels, as well as execution overlapping with input data transfer can be maintained. The latter programming style (PS-1) is adopted by us for analyzing GPU sharing scenarios, unless otherwise specified.

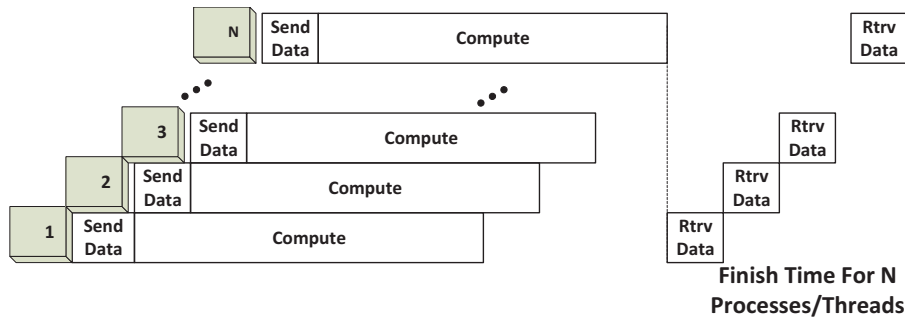
For modeling our SPMD Execution scenario, we bear in mind that there is the possibility of overlapping computation phases from multiple kernels only if the I/O transfer time is sufficiently small. To elaborate, consider the case when multiple kernels are initiated from CUDA streams, each requiring input data transfer, execution and output data transfer. Since input data transfers need to occur sequentially, if we have $T_{data.in} \geq T_{comp}$, the input data transfer of a subsequent kernel can only finish after the computation phase of the earlier kernel. In other words, the computation phases cannot overlap, and we categorize this as I/O-intensive. Only time-sharing can thus occur for I/O-intensive cases. We therefore begin our analysis for the compute-intensive case, for which $T_{data.in} < T_{comp}$ with PS-1 adopted.

4.3.1. Execution Model for Compute-Intensive Applications

We first lay out the GPU execution and sharing models for compute-intensive applications. We make the following assumption for the model: I/O in each direction consumes the full I/O bandwidth and cannot be overlapped with another I/O transfer in the same direction; the output I/O cannot be overlapped with kernel execution, due to the aforementioned CUDA feature for the chosen programming style (PS-1). Furthermore, we assume that kernels finish execution following the order in which they are initiated; in other words, the output data transfer sequence follows the kernel launching sequence, even though output needs to wait until all kernels have finished execution, as mentioned earlier. This last assumption on the sequence does not affect the estimation of total execution time.

Exclusive Space Sharing. As discussed earlier, Exclusive Space Sharing allows all kernels to achieve complete concurrency, due to the fact that all kernel blocks are executed on different SMs. With CUDA streams, input data transfer can also be overlapped with kernel execution. However, the retrieve data stages have to wait until all computing stages have finished, since streams are programmed for concurrent kernel execution. Thus, we define the execution model as in Figure 10 and derive the total execution time as given in (10) below:

$$T_{ex.sp} = N_{SPMD.P}(T_{data.in} + T_{data.out}) + T_{comp} \quad (10)$$

Figure 10. Execution model for exclusive space sharing.

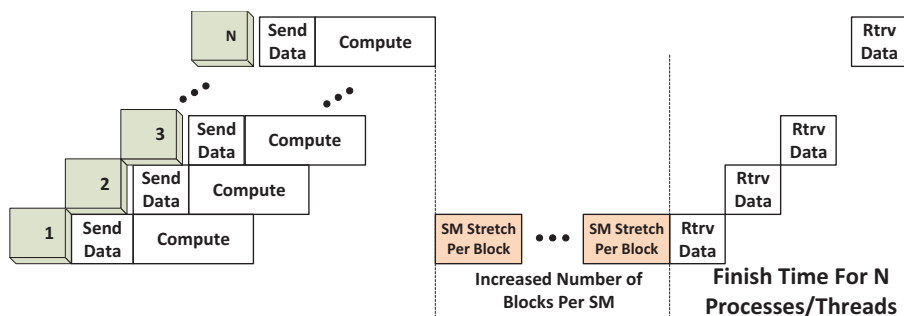
Non-Exclusive Space Sharing. Non-Exclusive Space Sharing allows blocks from all kernels to reside in all SMs simultaneously without moving to the next SM execution round. However, scheduling more blocks on each of the single SM stretches the execution time of each SM, compared with when only one kernel is executed on the GPU. We use the term “SM time stretch” to denote increased execution time when the number of blocks per SM increases. We define T_{SM_str} as the time stretch when the number of blocks per SM increases by one. T_{SM_str} can be expressed as a function of the number of blocks per SM ($N_{blks_per_SM}$), as shown in (11). Depending upon the nature of the GPU kernel, T_{SM_str} can be derived empirically in our following experimentation section.

$$T_{SM_str} \equiv T_{SM_str}(N_{blks_per_SM}) = T(N_{blks_per_SM} + 1) - T(N_{blks_per_SM}) \quad (11)$$

Thus, under Non-Exclusive Space Sharing, the total time stretch of the execution time component, T_{comp} , is the SM time stretch when the number of blocks per SM increases with the added ($N_{SPMD_P} - 1$) SPMD kernels. We describe the model behavior of this scenario in Figure 11, the basis upon which we derive the execution time in Equation (12) below. The number of blocks per SM with only one kernel executing is: $N_1 = \left\lceil \frac{N_{blks_per_knl}}{N_{SM}} \right\rceil$. Under SPMD execution with N_{SPMD_P} processes, the number of blocks executing on each SM increases to $N_2 = \left\lceil \frac{N_{blks_per_knl} \cdot N_{SPMD_P}}{N_{SM}} \right\rceil$.

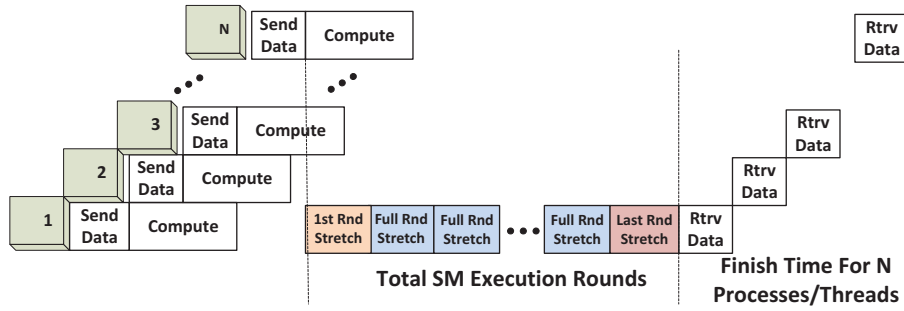
The total execution time is therefore:

$$T_{n_ex_sp} = N_{SPMD_P}(T_{data_in} + T_{data_out}) + T_{comp} + \sum_{i=N_1}^{N_2-1} T_{SM_str}(i) \quad (12)$$

Figure 11. Execution model for non-exclusive space sharing.

Space/Time Sharing. Under Space/Time Sharing, when each SM executes one or more blocks, the increased block number per SM due to the added kernels from other processes can be accounted for using the same T_{SM_str} as previously defined. The time taken for the execution phase is given by the sum of the execution phases in each computation round. An illustration of this case under the SPMD scenario is given in Figure 12.

Figure 12. Execution model for space/time and time sharing.



However, since the execution phases of the different kernels are intermingled, they cannot be separately depicted in the figure. Instead, for ease of calculation, we show the execution times for the different rounds that need to be added to the execution time of the first kernel, shown in Figure 12 for SPMD_P1. The added components consist of the following:

1. $T_{fs_rnd_str}$: This term captures the stretch in the execution time of the first kernel during the first round. The first execution round will definitely execute according to its full capacity; that is, each SM will execute $N_{max_blks_per_SM}$ thread blocks. Accordingly, the stretch is computed based on an increase in the number of blocks per SM from $N_1^{fs} = \left\lceil \frac{N_{blks_per_knl}}{N_{SM}} \right\rceil$ to $N_2^{fs} = N_{max_blks_per_SM}$. Therefore:

$$T_{fs_rnd_str} = \sum_{i=N_1^{fs}}^{N_2^{fs}-1} T_{SM_str}(i) \quad (13)$$

2. $T_{full_rnd_str}$: This term quantifies the execution time of every execution round that occurs subsequent to the first round. Effectively, this is the stretch in execution time that occurs due to the presence of the particular execution round. Again, the round is expected to execute according to its full capacity, with $N_{max_blks_per_SM}$ blocks per SM. The execution time of the round is computed as a stretch that occurs when the number of blocks per SM is increased from $N_1^{full} = 0$ to $N_2^{full} = N_{max_blks_per_SM}$. We therefore have:

$$T_{full_rnd_str} = \sum_{i=N_1^{full}}^{N_2^{full}-1} T_{SM_str}(i) \quad (14)$$

Note that the final execution round may not be able to run at its full capacity and is therefore modeled separately, as explained next.

3. $T_{ls_rnd_str}$: This term captures the time duration of the final execution round, since the last round may not have the required number of thread blocks to occupy SMs at their maximum capacity. If

there are a total of R_{SM} rounds, the number of thread blocks available for the final round, defined as F , is:

$$F = N_{blks_per_knl} \cdot N_{SPMD_P} - (R_{SM} - 1) \cdot N_{max_blks_per_SM} \quad (15)$$

which is based on the assumption that all the earlier ($R_{SM}-1$) rounds will execute SMs with their maximum capacity. The time duration of the final round, computed as a stretch resulting from increasing the number of SMs per block from $N_1^{ls} = 0$ to $N_2^{ls} = \left\lceil \frac{F}{N_{SM}} \right\rceil$, is therefore:

$$T_{ls_rnd_str} = \sum_{i=N_1^{ls}}^{N_2^{ls}-1} T_{SM_str}(i) \quad (16)$$

The total number of execution rounds in the foregoing analysis is computed by dividing the total number of thread blocks from all kernels by the maximum SM capacity that is possible in each round:

$$R_{SM} = \left\lceil \frac{N_{blks_per_knl} \cdot N_{SPMD_P}}{N_{SM} \cdot N_{max_blks_per_SM}} \right\rceil \quad (17)$$

The total execution time for the SPMD application can now be computed by referring to Figure 12:

$$T_{sp_tm} = N_{SPMD_P}(T_{data_in} + T_{data_out}) + T_{comp} + T_{fs_rnd_str} + (R_{SM} - 2) T_{rnd_str} + T_{ls_rnd_str} \quad (18)$$

Time Sharing. We define Time Sharing for compute-intensive applications (under PS-1) to be the scenario when a single kernel is large enough to be executed for one or more SM round. While merely many more SM execution rounds are present, time sharing can be modeled the same as Space/Time Sharing, as shown in Figure 12.

Accordingly, the execution time for the Time-Sharing scenario can be derived as in (19) below, similar to (18):

$$T_{tm} = N_{SPMD_P}(T_{data_in} + T_{data_out}) + T_{comp} + T_{fs_rnd_str} + (R_{SM} - 2) T_{rnd_str} + T_{ls_rnd_str} \quad (19)$$

4.3.2. Execution Model for I/O-intensive Applications (Always Time Sharing)

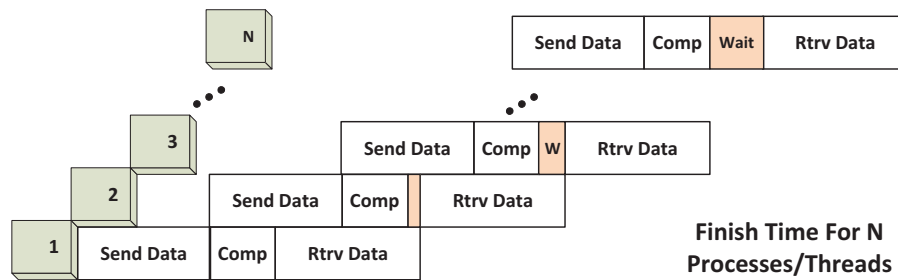
For I/O-intensive applications, the data transfer time dominates and leaves no kernel execution concurrency. As described previously, this scenario occurs when $T_{data_in} \geq T_{comp}$. In other words, computations will always occur in a time-shared fashion. Since we are sure that kernel executions cannot overlap, our sharing approach adopts the stream programming style aimed at I/O concurrency (PS-2), as previously described. This allows overlap to occur between output data transfer and kernel execution, while maintaining concurrency between input transfer and kernel execution, as well as bidirectional I/O capability.

These features are captured in the execution model shown in Figure 13; both T_{data_in} and T_{data_out} can be inter-overlapped, as well as overlapped with T_{comp} , while T_{data_out} can only be sequential and possibly

waits, as shown in the figure. In other words, the output “wait” happens when $T_{data_in} < T_{data_out}$ and *vice versa*. Thus, by combining two conditions, the total time can be derived as shown in (20):

$$T_{tm_io_i} = N_{SPMD_P} \text{Max}(T_{data_in}, T_{data_out}) + T_{comp} + \text{Min}(T_{data_in}, T_{data_out}) \quad (20)$$

Figure 13. Execution model for I/O-intensive application.



4.4. Theoretical Performance Gains

Our modeling analysis provides estimates of total execution time under different sharing scenarios. Using our baseline as described in (9), the performance gain in terms of speedup can be derived accordingly using T_{native_sh}/T_{sh_sce} , which provides the theoretical performance estimate for process-level GPU sharing using our sharing approach through streaming execution.

5. Experimental Analysis and Performance Results

In this section, based on the theoretical analysis provided previously, we experimentally analyze the proposed GPU sharing scenarios in terms of efficiencies and approaches targeting further performance improvements. All of our experiments are conducted on our GPU computing node, which is equipped with an NVIDIA Tesla C2070 GPU consisting of 14 SMs running at 1.15 GHz and 6 GB of device memory. The node also has dual Intel Xeon X5570 quad-core processors (8 cores at 2.93 GHz) with 48 GB of system memory and runs under Ubuntu 11.04 with the 2.6.38-8 Linux kernel. CUDA 4.0 has been used as the GPU development environment.

5.1. Experimental Validation of the Sharing Model

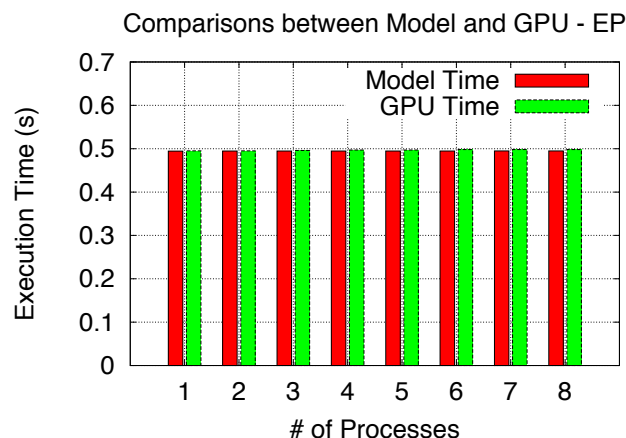
We first conduct several benchmarks to experimentally verify the proposed modeling analysis. Here, we utilize five application benchmarks from different scientific fields representing each of the sharing scenarios. Each benchmark is preliminarily profiled with necessary parameters, as shown in Table 4. By also considering the SM specification of NVIDIA C2070 ($N_{max_warps_SM}=48$; $S_{shm_per_SM}=48\text{KB}$; $N_{regs_per_SM}=32\text{K}$), we are able to derive $N_{max_blks_per_SM}$ for each application. The profiled five benchmarks are used to cover all described scenarios, so that each benchmark is used to verify a specific sharing scenario, as also shown in Table 4.

Table 4. Profiling results of GPU kernel benchmarks.

	Embarrassingly Parallel (EP)	Black–Scholes (BS)	Electrostatics (ES)	Multi–Grid (MG)	Vector Multiplication (VecM)
Class	Comp-intensive	Comp-intensive	Comp-intensive	Comp-intensive	I/O-intensive
Problem Size	M=24	3.92M options/64 iterations	100'000 atoms	Class W	16M floats/15 iterations
Grid Size(Blocks)	1	14	48	4096	16'000
Block Size(Threads)	128	128	256	64	1024
T_{data_in} (ms)	0	8.67	0.03	6.71	36.20
T_{comp} (ms)	494.49	423.29	210.457	50.51	11.98
T_{data_out} (ms)	0.013	18.37	1.97	12.82	15.51
$N_{regs_per_thd}$	35	16	24	52	10
$S_{shm_per_blk}$ (Byte)	0	0	0	0	0
$N_{max_blks_per_SM}$ from eq. (2)	7	8	5	8	1
Profiled Sharing Scenario	Exclusive Space	Non-Exclusive Space	Space/Time	Time	Time

As we described, the proposed sharing scenarios can be achieved both with process-level and thread-level parallelisms, while SPMD kernels are launched with CUDA streams on both levels. In this section, we utilize process-level SPMD program emulation as the example approach for implementation and verification. In other words, we emulate the SPMD program by launching the same GPU benchmark kernel on multiple processes simultaneously using our implemented GPU virtualization infrastructure. Here, we primarily compare the GPU time with the execution time analyzed through the model. The GPU time refers to the time that all processes spend on sharing the GPU within the single Daemon Process, which is provided by our GPU virtualization approach. The model results are derived by using the previously defined equations with the profiling results for each sharing scenario, respectively. With each process being set with affinity to a CPU core to the maximum number of 8 cores, we vary the number of emulated SPMD processes from 1 to 8.

We first utilize the NASA advanced supercomputing Parallel Benchmarks (NPB) [32] Embarrassingly Parallel (EP) GPU kernel(smallest class) [33], which is written using one single block merely for the purpose of Exclusive Space Sharing verification. Since each single block EP only takes a single SM out of the total of 14 SMs, all processes exclusively space-share the GPU when the number of processes increase from 1 to 8. The verification comparison is shown in Figure 14, and all 8 model cases perfectly match the GPU results.

Figure 14. Model validation (Exclusive Space).

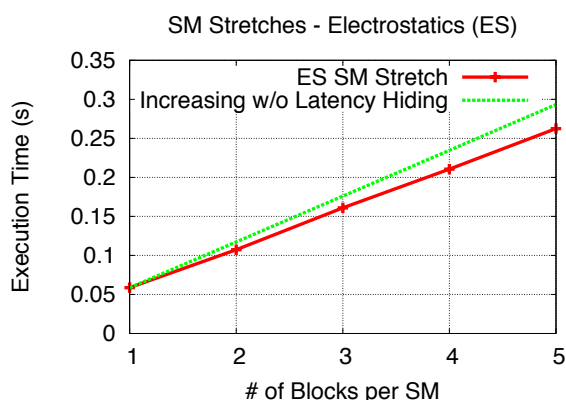
To verify both Non-Exclusive Space and Space/Time Sharing scenarios, we respectively utilize Black–Scholes (BS) [34], a European option pricing benchmark, and the fast molecular electrostatics algorithm (ES), which is a part of the molecular visualization program, Visual Molecular Dynamics (VMD) [35].

Two initial microbenchmarks are conducted to analyze the T_{SM_str} , for both BS and ES, respectively, when $N_{blks_per_SM}$ increases. This is performed by increasingly feeding the same benchmark kernel blocks into the GPU until the number of blocks per kernel reaches $N_{max_blks_per_SM}$, while at each x-axis point, ensuring that all SMs hold the certain same number of blocks. As shown in Figure 15 and 16, the execution time of BS and ES are plotted for each number of blocks per SM (1 to 8 for BS and 1 to 5 for ES). In our modeling analysis, we defined T_{SM_str} as a function of the number of increased blocks per SM. Note that many warps (blocks) being executed within an SM execution round is to hide the GPU memory access latencies, even though warps are executed sequentially. Thus, T_{SM_str} with adding one/two more blocks in the SM does not linearly double/triple the execution time. As comparisons to T_{SM_str} , in both figures, we use “Increasing Without Latency Hiding” lines to represent the theoretical scenario when all block execution is sequential and no memory latency is hidden, so as to show the effectiveness of memory hiding for both kernels.

Figure 15. SM stretch (Black–Scholes).



Figure 16. SM stretch (ES).



Since Non-Exclusive Space Sharing (BS) only involves T_{SM_str} within a single SM execution round, the modeled execution time for BS can be derived from Figure 15, and the model validation comparison is shown in Figure 17. To validate Space/Time Sharing using ES, we utilize the previous analysis to determine the number of SM execution rounds and corresponding $T_{full_rnd_str}$, $T_{fs_rnd_str}$ and $T_{ls_rnd_str}$, while the time for each round is derived from T_{SM_str} in Figure 16. Thus, the execution time from the model can be determined and compared with experimental results, as shown in Figure 18. Both validation comparisons show a good model accuracy for both Non-Exclusive Space and Space/Time Sharing scenarios.

Figure 17. Model validation (Non-Exclusive Space).

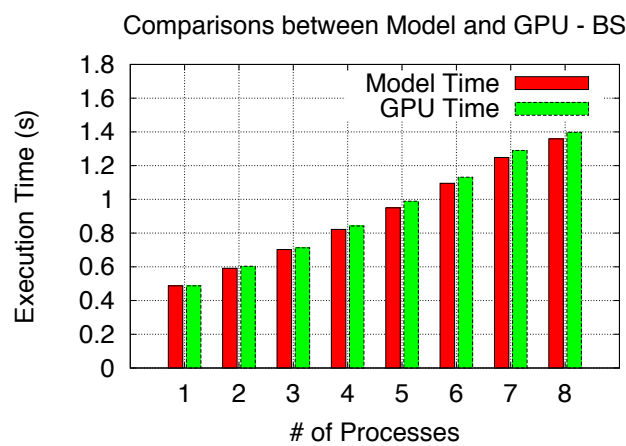
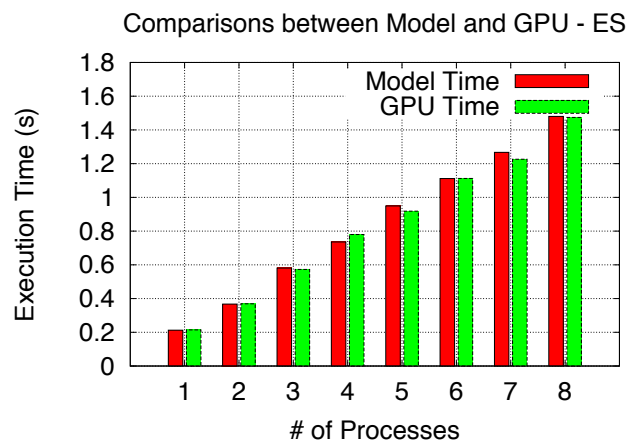


Figure 18. Model validation (Space/Time).



In verifying the Time Sharing scenario, we utilize our NPB Multi-Grid (MG) kernel (Class W with 4,096 blocks) [33] and, thus, only time shares of the GPU among multiple processes. As our Time Sharing model follows the Space/Time Sharing model, similarly, the comparison results of the model and the GPU are described in Figure 19 and also demonstrate good agreement. Figure 20 shows the Time Sharing scenario for I/O-intensive applications, for which we use a simple Vector Multiplication benchmark. While our model assumes only I/O overlapping, the derived model results match the GPU

nicely. Therefore, in Table 5, we summarize the average model deviations (averaged from 1 to 8 processes) for each sharing scenario, as a general model accuracy demonstration with less than 5% deviation for all cases.

Figure 19. Model validation (Time).

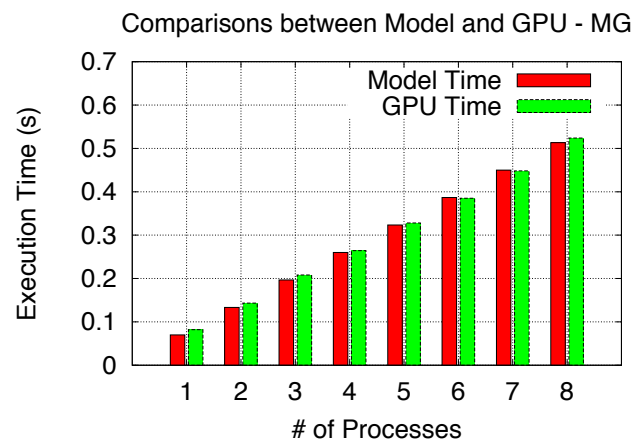


Figure 20. Model validation (I/O-intensive).

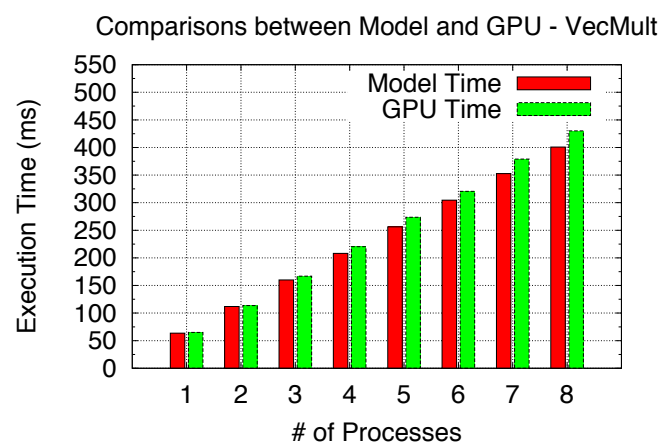


Table 5. Average model deviations for all scenarios (averaged from 1 to 8 processes).

Exclusive Space (EP)	Non-Exclusive Space (BS)	Space/Time (ES)	Time (MG)	Time-I/O-I (VecM)
0.42%	2.73%	1.92%	4.10%	4.76%

5.2. Performance Prediction from the Model

One purpose in using the model is to analyze the possible performance gain with GPU sharing. With the previous verified model accuracy, we use several benchmarks with the aid of the model to demonstrate the sharing efficiency that can be achieved through varied sharing scenarios. This is done by varying the SPMD kernel sizes (problem sizes in general) to extend each benchmark kernel

across different sharing scenarios. Here, we utilized the same aforementioned EP, BS and MG kernel benchmarks, but with different problem sizes. For each of the three benchmarks, four different problem sizes for each SPMD GPU benchmark kernel are employed to represent four sharing scenarios. We first provide detailed profiling for each of the 12 benchmarks as our basis for the model, as shown in Table 6. Further similar microbenchmarks are conducted to analyze the SM stretch for EP and MG, as shown in Figure 21 and 22, with theoretical linear increasing without latency hiding as the comparison. The previous analyzed SM stretches of BS and ES are also utilized as shown in Figure 15 and 16. Therefore, the model results for the 12 benchmarks can be derived. As shown in Figure 23, 24 and 25, the modeled execution time under GPU sharing is compared with modeled sequential time for all four sharing scenarios. Please note that the modeled sequential time does not consider T_{init} and T_{ctx_switch} . This is due to the fact that T_{init} can be hidden and T_{ctx_switch} does not apply to thread-level sharing. Thus, the sequential comparison baseline provides the lower bound of performance gain, due to efficient GPU sharing.

Table 6. Benchmark profiles across varied sharing scenarios.

	Scenario	Problem Size (Kernel)	Grid Size	$T_{data_in}(ms)$	$T_{comp}(ms)$	$T_{data_out}(ms)$	$N_{max_blks_per_SM}$	SM Rounds	$N_{blks_per_SM}$ of last round
EP1	Exclusive Space	M=24	1	0	494.49	0.013	7	1	1
EP2	Non-Exclusive Space	M=28	16	0	558.01	0.013	7	1	2
EP3	Space/Time	M=30	64	0	950.87	0.013	7	1	5
EP4	Time	M=31	128	0	1823.33	0.013	7	2	3
BS1	Exclusive Space	0.28M Options	1	1.04	413.13	1.99	8	1	1
BS2	Non-Exclusive Space	3.92M Options	14	10.32	469.65	17.98	8	1	1
BS3	Space/Time	19.6M Options	70	48.24	808.25	92.62	8	1	5
BS4	Time	98M Options	350	306.01	3641.87	466.93	8	4	1
MG1	Exclusive Space	4x4x4	1	0.088	0.491	0.107	8	1	1
MG2	Non-Exclusive Space	28x8x4	14	0.096	0.685	0.135	8	1	1
MG3	Space/Time	28x14x12	84	0.142	1.191	0.192	8	1	6
MG4	Time	28x32x16	224	0.242	2.763	0.340	8	2	8

Figure 21. SM stretch (EP).

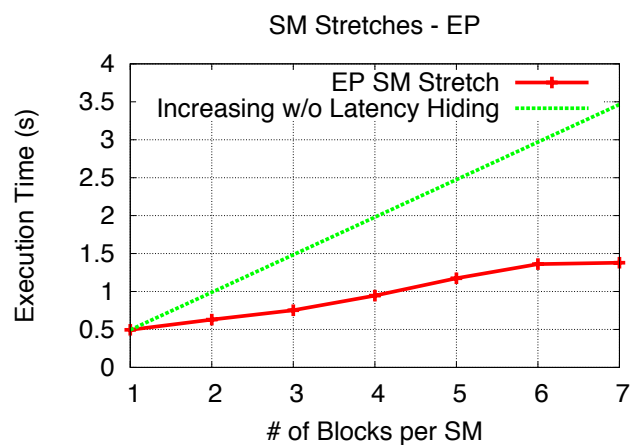


Figure 22. SM stretch (MG).

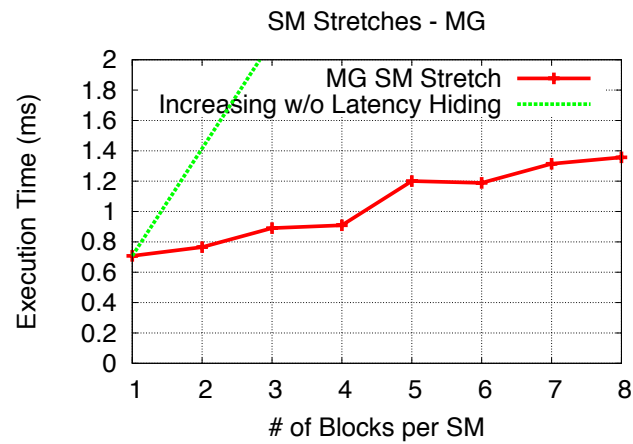


Figure 23. Performance from the model (EP).

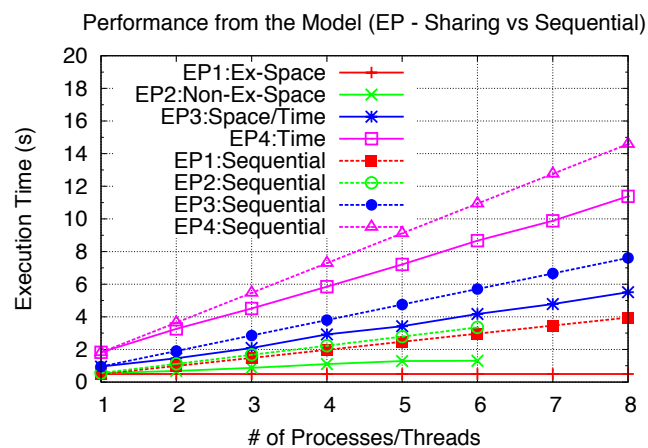


Figure 24. Performance from the model (BS).

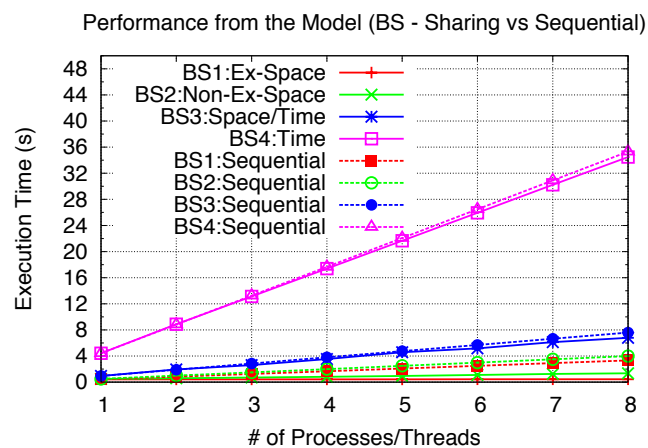
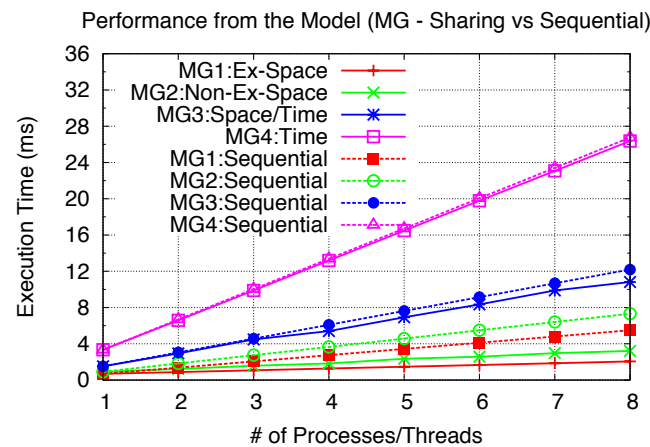


Figure 25. Performance from the model (MG).

In Figure 23, we are able to demonstrate the achievable performance gain under GPU sharing with the increasing SPMD parallelism. For EP2, we only increase the parallelism to 6, since the Non-Exclusive Space Sharing will cross to Space/Time Sharing when the parallelism reaches 7. For all other EP kernels, varying the parallelism from 1 to 8 will always achieve the specified sharing scenario. Further modeled performance improvements are also demonstrated in Figures 24 and 25 for the BS and MG kernel with the same parallelism variation from 1 to 8. Thus, the achievable speedups of all three benchmarks (parallelism of 6 for EP2, 8 for all others) can be summarized in Table 7.

Table 7. Theoretical achievable speedups.

Benchmark	Exclusive Space	Non-Exclusive Space	Space/Time	Time
EP	7.99	2.56	1.38	1.28
BS	7.61	2.93	1.12	1.03
MG	2.67	2.28	1.13	1.02

Our described sharing modeling uses the CUDA stream programming style targeting kernel execution parallelism (PS-1). No I/O concurrency between SPMD kernels can be achieved. Thus the profiles of the kernel itself in terms of I/O time percentage are one of the determinants of sharing performance improvements. Other than I/O time, the problem size of each kernel determines the number of SMs and SM execution rounds to be occupied and, thus, also determines the performance gain under GPU sharing.

Exclusive Space Sharing can achieve complete inter-kernel parallelism when the kernel problem size is small; thus, with very little I/O for both EP1 and BS1. Close to 8-times speedups can be achieved. Since MG1 has relatively higher I/O intensity, 2.67-times speedups can be achieved.

In Non-Exclusive Space Sharing, all kernels are to be executed within a single SM round. Even with more blocks sequentially sharing the SM execution within a round, the total execution time of the SM round can still be much less than the sequential execution time of each single block, as we have already demonstrated in Figure 15, 16, 21 and 22. Depending on the kernel nature (memory access pattern, *etc.*), significant performance gain still can be achieved in Non-Exclusive Space Sharing. Moreover,

according to our model analysis, if blocks from one kernel leave more space sharing opportunities, more performance gains can be achieved. We can observe from 2- to 3-times speedups for three benchmarks with Non-Exclusive Space Sharing.

Under Space/Time and Time Sharing, multiple kernels are executed across many SM rounds. The performance gains depend upon the number of “left-over” blocks from the first SM round (Space/Time) and the last SM round (Time). In other words, the amount of “Space Sharing” determines the performance potential. For EP3, a great percentage of blocks from the second kernel can space-share with the first kernel, resulting in 38% performance gain. Similar results are also demonstrated with BS3 and MG2. For EP4, the second kernel can still space-share part of the first kernel’s last SM round, plus only 2 SM rounds for each kernel; thus, it is able to provide 28% improvement. For both BS4 and MG4, only 2–3% of improvement can be achieved. In the case of MG4, each kernel takes 2 full SM rounds, leaving no possible space (SM round)-sharing for the next kernel. In the case of BS4, the last execution round only use 1 block per SM. However, since the total number of SM rounds is 4, which involves 3 full rounds, the improvement potential is also comparatively small. Thus, while Space/Time Sharing can still achieve speedup with some “Space Sharing”, Time Sharing can achieve limited speedup with only sharing the last SM round of the previous kernel.

5.3. Sharing Efficiency Exploration and Improvement Potential Analysis

In this section, we will utilize our model to provide optimization considerations targeting improved GPU sharing efficiency. We will consider a few schemes, including providing further I/O concurrency for specific Time Sharing scenarios, optimizing the SPMD parallelisms as well as providing kernel-level optimizations and Sharing Scenario Casting. For each of the optimization schemes, applications are to be provided and analyzed with further benchmarking as an optimization performance demonstration.

5.3.1. Time Sharing: Concurrent Kernel Execution vs. Concurrent I/O

The previously analyzed sharing scenarios are based on the CUDA stream Programming Style (PS)-1, which aims at concurrent kernel execution for all compute-intensive applications. From the modeling performance results, we are able to observe considerable kernel execution concurrency for Exclusive/Non-Exclusive Space and Space/Time Sharing. However, Time Sharing cannot achieve much performance improvements, especially with large kernel size and many SM execution rounds. Under PS-1, a very limited I/O concurrency can be achieved. Meanwhile, very limited kernel execution concurrency can happen under Time Sharing; choosing an optimum CUDA stream programming style for a specific SPMD GPU program is essential.

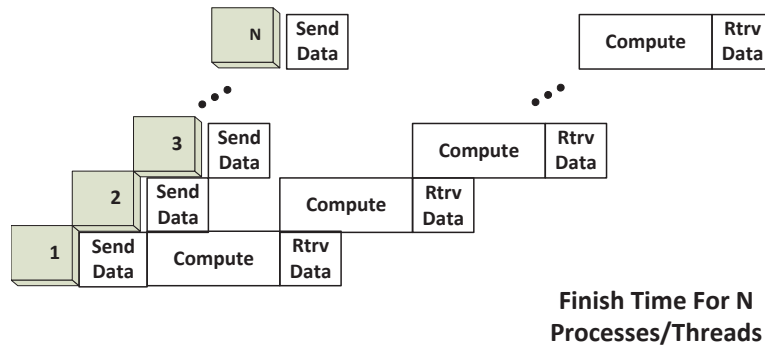
Therefore, if the SPMD kernels time share the GPU under PS-1 and I/O, time is not negligible. When time-sharing cannot achieve much performance gain with modeling analysis, we choose to use PS-2 targeting I/O concurrency to achieve an optimized sharing efficiency as one of the optimization strategies using the model.

Under this optimization strategy, the execution model can only achieve sequential T_{comp} , due to PS-2. However, the non-negligible $T_{data.in}$ and $T_{data.out}$ can be both inter-overlapped and overlapped with T_{comp} , as shown in Figure 26. We use T_{tm} and $T_{tm_op_io}$ to represent the execution time of Time Sharing under

PS-1 and optimized I/O concurrency under PS-2, respectively. As T_{tm} can be derived using previous Equation (19), $T_{tm_op_io}$ can be derived with Equation (21).

$$T_{tm_op_io} = T_{data_in} + N_{SPMD_P}T_{comp} + T_{data_out} \quad (21)$$

Figure 26. Comp-intensive kernels: concurrent I/O.



Thus, for a given SPMD GPU benchmark that has been profiled as Time Sharing and achieves low performance gain under GPU time sharing, with the help of our analytical model, Equation (21) will possible give the optimized achievable performance gain under PS-2 *versus* PS-1 (Time Sharing) given by Equation (19).

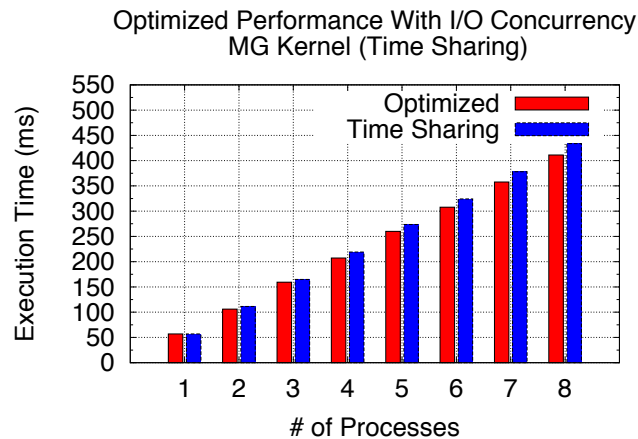
By utilizing the MG benchmark with the size of W (64x64x64, 40 iterations and 4,096 blocks per kernel) shown in Table 4, we here demonstrate the achievable performance improvement from I/O concurrency under Time Sharing with the effective Programming Style suggestion from the model. With profiling results in Table 4, comparing Equation (21) with Equation (19) gives us a theoretical potential performance gain (N_{SPMD_P} from 2 to 8) when switching to PS-2. As the model suggests that PS-2 can bring more performance benefits, we conduct the same MG benchmark on the GPU to study the actual performance gain with varied N_{SPMD_P} . The GPU virtualization implementation is used for process-level SPMD program GPU sharing as the example experimentation. By increasing the number of processes (N_{SPMD_P}) from 2 to 8, we are able to achieve 6% performance gain with concurrent I/O compared with pure Time Sharing, as shown in Figure 27. Thus, considering kernels with the Time Sharing scenario with our modeling analysis, choosing an optimized programming style leads to an improved GPU sharing efficiency.

5.3.2. Exclusive Space Sharing: Utilization of SMs vs. Optimizable SPMD Kernel Grid Size

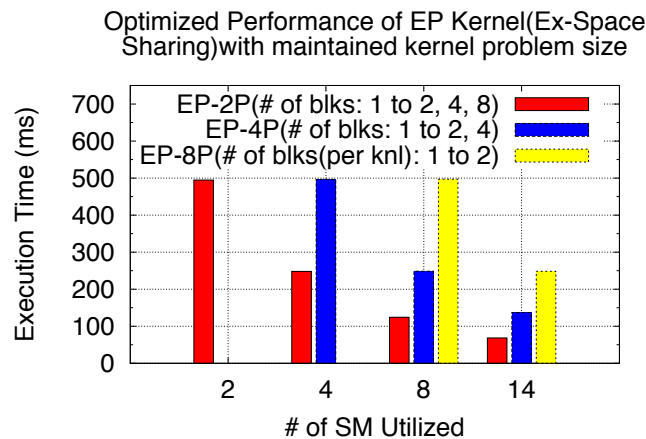
While Exclusive Space Sharing applies to kernels that execute small problem sizes, each SPMD kernel block is exclusively executed on one or more SMs. As a parallel SPMD program is to execute a certain big problem size and distribute it across processes/threads, under Exclusive Space Sharing, a GPU kernel from each process/thread exclusively takes one or more SM. From what we understand from the model, an optimized performance could be achieved only when all SMs are utilized. This is especially helpful in improving the SPMD program, which does not fully utilize all SM resources when written. Thus, here, we use kernel EP, shown in Table 4, to demonstrate the achievable performance gain

with the optimized problem/kernel size per process/thread, while maintaining the same total problem size. Here, we consider three benchmark cases with the same EP kernel shown in Table 4, but launched with a different number of processes, as shown in Table 8.

Figure 27. I/O concurrency optimization for MG.



Our optimization principle is to make the SPMD program utilize all GPU SM resources by optimizing the kernel problem/grid sizes, while maintaining the kernel problem sizes. In other words, our optimization technique is to increase the number of blocks per SPMD kernel by reducing the computation workload of each kernel block, so that more block-level parallelism can be obtained, and thus, the initial unused SMs can be utilized. Figure 28 shows the performance comparison for the 3 benchmarks in Table 8 with optimized kernel block sizes. The three EP SPMD benchmarks have different total problem sizes (with the varied number of parallel processes), but the same kernel problem size. Please note here that we still use the process-level emulated SPMD program as the benchmark study example, and the GPU execution times are experimentally derived through our virtualization implementation. As shown in Figure 28, the shared GPU execution time for EP-2P can be reduced with up to 7.2 times speedups when all SMs has been utilized, while EP-4P and EP-8P can achieve 3.6 and 2 times speedups, respectively. These speedups depend highly on the number of unutilized SMs when SPMD programs exclusively space-share the GPU. Meanwhile, since the computation intensity of the GPU kernel threads primarily depends on the kernel itself, not all kernels can be optimized for more blocks with a maintained problem size. Here, our benchmark is purely to demonstrate the optimization technique based on our previous modeling analysis and potential performance improvements. We also note that the original Exclusive Space Sharing scenario might be switched to Non-Exclusive Space Sharing (when the total number of blocks from all SPMD kernels exceeds N_{SM}); we will define this as Sharing Scenario Casting in the following analyses. However, here, our main purpose is to optimize utilizing all SM resources for an SPMD program under Exclusive Space Sharing.

Figure 28. SM utilization optimization for EP.**Table 8.** Benchmarks profiles - exclusive space sharing and kernel size optimization.

	Processes	Problem Size (Kernel)	# of Blocks per Kernel	Problem Size (SPMD Program)	# of SM Utilized
EP-2P	2	M=24	1	M=25	2 out of 14
EP-4P	4	M=24	1	M=26	4 out of 14
EP-8P	8	M=24	1	M=27	8 out of 14

5.3.3. Maintained SPMD Problem Size/Sharing Scenario vs. SPMD Parallelisms

There are cases when partial SM resources are not used in Exclusive Space Sharing. Here, we focus on the scheme in which all SM resources are utilized by SPMD kernels. In other words, the total number of GPU kernel blocks from an SPMD program are large enough to cover at least N_{SM} . In this scheme, we are carrying out studies on the impact of SPMD parallelism on the performance of GPU sharing in general, when the total SPMD program problem size is maintained. The SPMD parallelism is denoted as N_{SPMD-P} . As a motivation, our previous modeling analysis assumes that the T_{comp} of all kernels starts executing simultaneously and that the I/O hiding (concurrency between T_{data_in} and T_{comp}) is offset by the assumption that the first T_{data_out} is waiting on the last T_{comp} to finish. Thus, the SPMD parallelism might affect the GPU sharing performance. We note that the sharing scenario is also maintained here with the maintained total number of kernel blocks, due to the maintained SPMD problem size.

Table 9. Benchmarks profiles - maintained SPMD problem sizes vs. SPMD parallelisms.

	Processes	Problem Size (Total)	Problem Size (Kernel)	# of Blocks per Kernel	Scenario
BS-1P-NES	1	31.36M	31.36M	112	Non-Exclusive Space (NES)
BS-2P-NES	2	31.36M	15.68M	56	Non-Exclusive Space (NES)
BS-4P-NES	4	31.36M	7.84M	28	Non-Exclusive Space (NES)
BS-8P-NES	8	31.36M	3.92M	14	Non-Exclusive Space (NES)
MG-1P-TS	1	28x32x16	28x32x16	224	Time Sharing (TS)
MG-2P-TS	2	28x32x16	28x16x16	112	Time Sharing (TS)
MG-4P-TS	4	28x32x16	28x16x8	56	Time Sharing (TS)
MG-8P-TS	8	28x32x16	28x8x8	28	Time Sharing (TS)

Since all SMs are utilized (Exclusive Space Sharing excluded), we consider two representative sharing scenarios here: Non-Exclusive Space with one SM round and Time Sharing with multiple SM rounds. We utilize two benchmarks, Black–Scholes and MG, to demonstrate the performance of an SPMD program with a maintained SPMD problem size, but varied SPMD parallelisms and GPU kernel problem sizes. Two benchmark profiles are shown in Table 9. Process-level SPMD programs are still used as the study example with all GPU sharing time benchmarked using our virtualization implementation. Figure 29 and 30 demonstrate the performance gain by increasing the SPMD parallelism in terms of the number of processes in our benchmark context for both BS and MG. We can see 16% and 14% performance gain here for BS and MG, respectively, by increasing the number of processes from 1 to 8. This is primarily due to the currency between $T_{data.in}$ and T_{comp} with the increased SPMD parallelism. Therefore, for a SPMD program to solve a maintained big problem size, our experimental studies demonstrate that increasing the SPMD parallelism (such as the number of parallel processes) can improve the GPU sharing efficiency with the aid of further I/O hiding.

Figure 29. Performance: maintained SPMD problem size (BS).

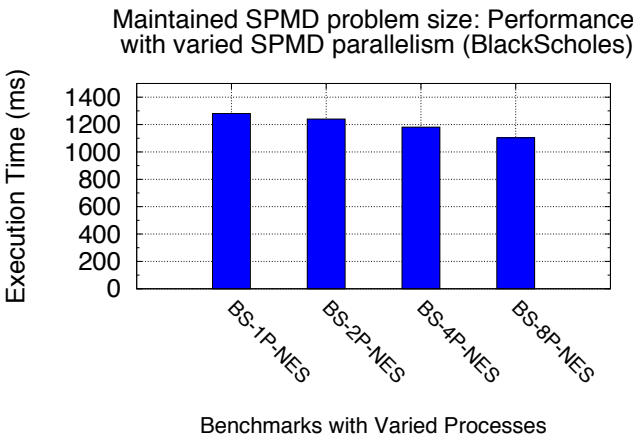
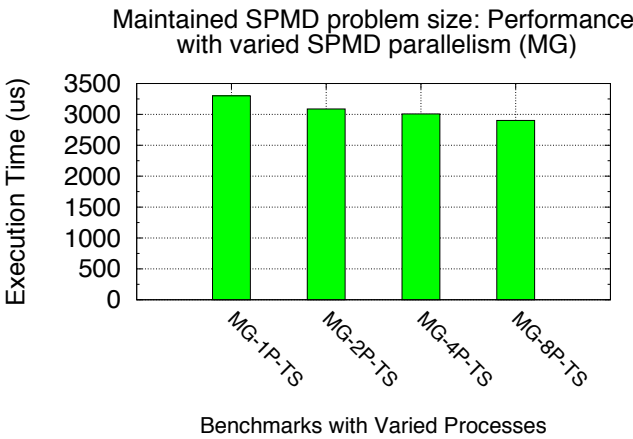


Figure 30. Performance: maintained SPMD problem size (MG).



5.3.4. Sharing Scenario Casting: Maintaining Both SPMD Problem Sizes and SPMD Parallelisms

For a given SPMD parallel program, since each GPU kernel holds the same number of blocks, the corresponding sharing scenario is determined by maintaining both SPMD problem size and $N_{SPMD.P}$. As we discussed earlier in optimization for Exclusive Space Sharing, we increased the number of blocks per kernel to allow unutilized SMs to be utilized. In other words, we increased the number of blocks for each kernel while maintaining each kernel's problem size. This is done by possibly changing the computation intensity of each thread to allow more threads/blocks involved to solve the problem of the same size.

Table 10. Benchmarks profiles - sharing scenario casting.

	Processes	Problem Size (Total)	Problem Size (Kernel)	# of Blocks per Kernel	Thread Compute Intensity	Scenario
EP-NES-Orig (Original)	8	M=28	M=25	8	2^{13}	Non-Exclusive Space (NES)
EP-ST-Cast1	8	M=28	M=25	16	2^{12}	Space/Time (ST)
EP-ST-Cast2	8	M=28	M=25	32	2^{11}	Space/Time (ST)
EP-ST-Cast3	8	M=28	M=25	64	2^{10}	Space/Time (ST)
EP-TS-Cast	8	M=28	M=25	128	2^9	Time Sharing (TS)
ES-ST-Orig (Original)	8	800K atoms	100K atoms	48	1.2K	Space/Time (ST)
ES-NES-Cast	8	800k atoms	100k atoms	8	4.8K	Non-Exclusive Space (NES)
ES-ST-Cast	8	800k atoms	100k atoms	24	2.4K	Space/Time (ST)
ES-TS-Cast	8	800k atoms	100k atoms	96	0.6K	Time Sharing (TS)

With the same technique, here, we focus on the possible optimization by using Sharing Scenario Casting for a given SPMD program with all SMs already utilized. We specifically consider the case of sharing scenarios with a single SM round (Non-Exclusive Space Sharing) and with multiple SM rounds (Space/Time or Time Sharing). Two benchmarks are utilized: the original EP under Non-Exclusive Space Sharing (EP-NES-Orig) and the original Electrostatics under Space/Time Sharing (ES-ST-Orig), as shown in Table 10. For both benchmarks, we cast the sharing scenarios by modifying the kernel sizes (number of blocks), while maintaining the kernel problem size. In other words, the compute intensity of each kernel thread is adjusted accordingly. With a varied kernel thread-level compute intensity set for each benchmark, we are able to cast EP-NES-Orig up to Space/Time and Time Sharing, as well as EP-ST-Orig from Space/Time down to Non-Exclusive Space and up to Time Sharing. In Figures 31 and 32, we compare the performance of scenario casting for both benchmarks. The similar process-level SPMD program is launched with the support of a GPU virtualization infrastructure, and thus, the time that all kernels spend on sharing the GPU is experimentally derived accordingly. As we can see from both figures, casting the sharing scenario to Time Sharing can bring a certain amount of performance gain : 63.1% for EP and 93.9% for ES. This is due to the increased number of warps being simultaneously scheduled within a SM execution round, due to the increased kernel sizes. On the other hand, casting ES-ST-Orig back to Non-Exclusive Space Sharing reduces the performance by 3.53-times. Thus we demonstrate that increasing the number of blocks per SPMD kernel with casting up to the Time Sharing

scenario can achieve improved GPU sharing performance. We here also note that the changeability of compute intensity for each kernel thread highly depends on the kernel coding. In other words, only certain kernels written with thread-level intensity control (such as a certain number of iterations) can be optimized using Sharing Scenario Casting. Therefore, not all kernels' sharing scenario can be cast. Here, we merely use EP and BS as the demonstrations of our Sharing Scenario Casting optimization technique to achieve an improved GPU sharing efficiency.

Figure 31. Sharing Scenario Casting (EP).

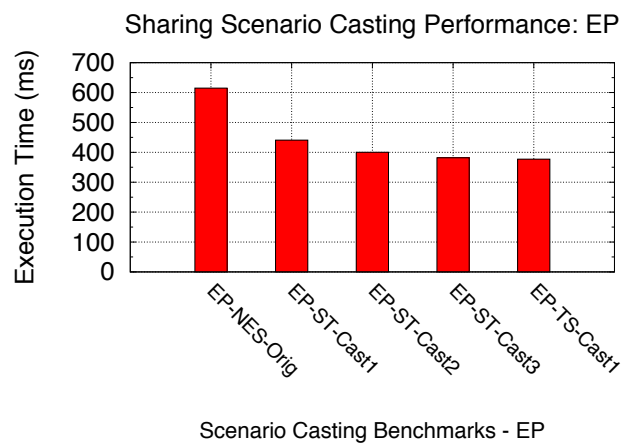
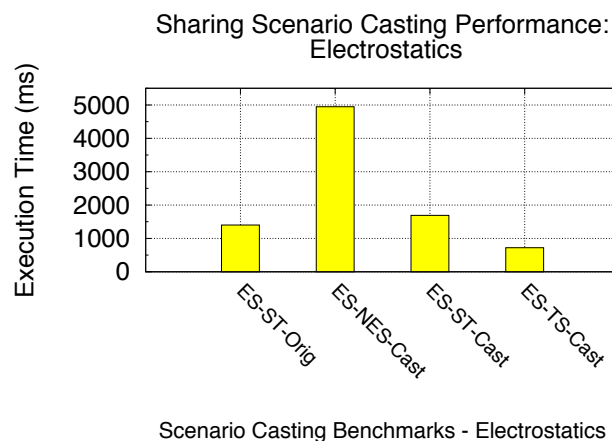


Figure 32. Sharing Scenario Casting (ES).



5.4. Performance Gains with GPU Sharing for SPMD Programs

As we theoretically analyzed the performance gain using GPU sharing under multiple sharing scenarios previously with the model, we experimentally demonstrate the performance advantage of using our GPU sharing approach for SPMD programs in this section. In other words, stream-level (CUDA streams) GPU sharing among kernels coming from different SPMD processes/threads achieves certain speedups over non-sharing (sequential kernel execution) depending on the kernel profiles and corresponding sharing scenarios. As an experimental setup, we here use the five aforementioned benchmark kernels in Table 4 and launch each benchmark with 8 processes. Since process-level

GPU sharing is achieved through our GPU virtualization infrastructure, we here empirically derive the actual total time that all SPMD processes spend on executing the GPU kernels within the single Daemon Process. The comparison baseline is the actual total time that all SPMD processes sequentially execute GPU kernels within the Daemon Process with no inter-kernel concurrency being achieved. Figure 33 demonstrates the actual experimental performance advantage when using GPU sharing over non-sharing, and Figure 34 shows the experimental speedups for all five benchmarks when launched with 8 processes. While both figures demonstrate up to 7.97-/2.68-times speedups for EP/BS under Exclusive/Non-Exclusive Space Sharing, ES and MG under Space/Time and Time Sharing can achieve 15% and 4% performance gain, respectively. Furthermore, I/O-intensive benchmark VecM under time sharing achieves 36% performance gain under GPU sharing. The experimental results also show a good agreement with the previously analyzed performance gain through the model. Thus, we are able to demonstrate both achievable and accurately analyzable performance gain with GPU sharing for SPMD programs.

Figure 33. Performance: GPU sharing vs. non-sharing.

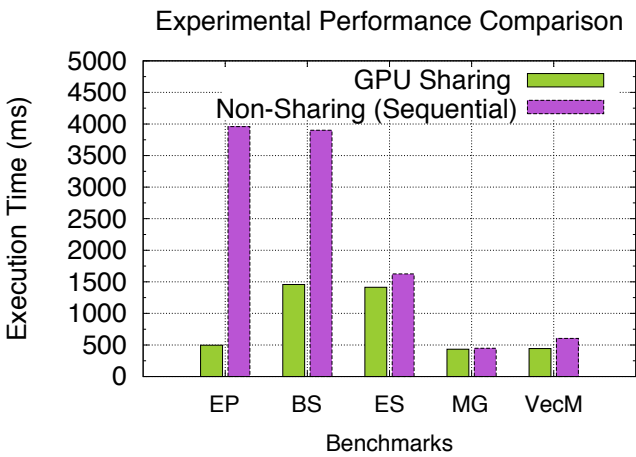
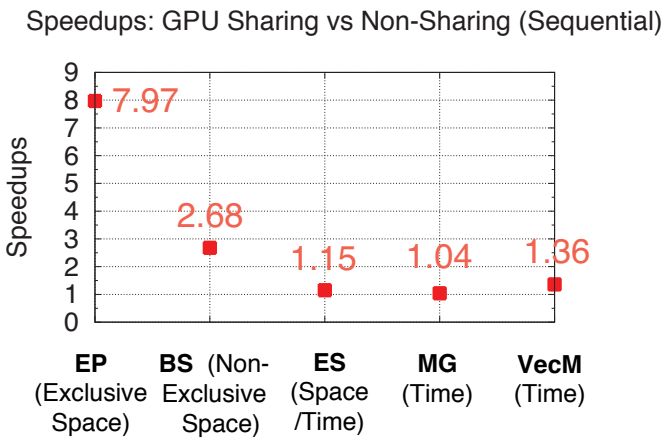


Figure 34. Experimental speedups with GPU sharing.



6. Conclusions

In this paper, we conducted extensive studies on providing efficient GPU resource sharing for the HPC heterogeneous platforms under the SPMD programming model. We formally defined efficient GPU sharing conditions and analyzed a series of GPU sharing scenarios from pure space-sharing to multiplexed time-sharing. A series of GPU sharing execution models have been introduced for each of the sharing scenarios, and we provide a theoretical prediction of the attainable performance gain over the non-sharing scenario. Initial performance benchmarking was conducted to validate the accuracy of the proposed sharing scenario modeling, followed by the detailed performance analysis for each of the sharing scenarios using varied benchmark profiles. Based on our theoretical analysis, we further experimentally studied potential performance improvements for GPU sharing from multiple perspectives, including: the problem size of the SPMD program; the GPU kernel size; and the SPMD parallelisms and sharing scenarios' switching/optimizations. Based on these factors, we proposed optimization techniques for different sharing scenarios, including the optimization of further I/O concurrency, SM utilization of the GPU, the SPMD parallelism and Sharing Scenario Casting. While our optimization strategies provided necessary coding and optimization suggestions to the parallel programmers, the utilized benchmarks for each strategy demonstrated a certain amount of performance gain compared to native SPMD program/kernel coding under GPU sharing. Moreover, our experimental studies also demonstrated the achievable performance gain and the advantage of using our stream-level GPU sharing approach for SPMD programs to eliminate resource under-utilization, as well as providing good agreement with the theoretical modeling analysis on the achievable performance gain under varied GPU sharing scenarios.

Acknowledgments

This work was supported in part by the I/UCRC (Industry & University Cooperative Research) Program of the National Science Foundation under Grant Nos. IIP (Industrial Innovation and Partnerships) - 1161014 and IIP - 1230815.

Conflicts of Interest

The authors declare no conflicts of interest.

References

1. GPGPU Webpage. <http://www.gpgpu.org> (accessed on 20 September 2013).
2. NVIDIA Corp. NVIDIA Tesla GPU Computing: Revolutionizing High Performance Computing. Available online: <http://www.nvidia.com/docs/IO/43399/tesla-brochure-12-lr.pdf> (accessed on 20 September 2013).
3. Advanced Micro Devices, Inc. AMD Firestream 9350 Datasheet. Available online: http://www.amd.com/us/Documents/FireStream_9350_Datasheet.pdf (accessed on 20 September 2013).
4. Fan, Z.; Qiu, F.; Kaufman, A.; Yoakum-Stover, S. GPU Cluster for High Performance Computing. In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing; IEEE, 2004; p. 47.

5. Kindratenko, V.; Enos, J.; Shi, G.; Showerman, M.; Arnold, G.; Stone, J.; Phillips, J.; Hwu, W. GPU Clusters for High-Performance Computing. In Proceedings of the IEEE International Conference on Cluster Computing and Workshops; IEEE, 2009; pp. 1–8.
6. Cray Inc. Cray XK7 Brochure. Available online: <http://www.cray.com/Assets/PDF/products/xk/CrayXK7Brochure.pdf> (accessed on 20 September 2013).
7. Cray Inc. Cray XK6 Brochure. Available online: <http://www.cray.com/Assets/PDF/products/xk/CrayXK6Brochure.pdf> (accessed on 20 September 2013).
8. SGI Corp. SGI GPU Compute Solutions. Available online: <http://www.sgi.com/pdfs/4235.pdf> (accessed on 20 September 2013).
9. Top 500 Supercomputer Sites Webpage. <http://www.top500.org> (accessed on 20 September 2013).
10. Titan Webpage in the Oak Ridge National Lab. <http://www.olcf.ornl.gov/titan> (accessed on 20 September 2013).
11. Darema, F. The SPMD Model: Past, Present and Future. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*; Cotronis, Y., Dongarra, J., Eds.; Springer: Berlin-Heidelberg, Germany, 2001; Chapter 1.
12. Gropp, W.D.; Lusk, E.L.; Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*; MIT Press, 1999.
13. China National Supercomputer Center in Tianjin Webpage. <http://nsc-tj.gov.cn> (accessed on 20 September 2013).
14. Nebulae Specification in Sugon Webpage. <http://www.sugon.com/en/> (accessed on 20 September 2013).
15. GSIC, Tokyo Institute of Technology. Tsubame Hardware Software Specifications. Available online: http://www.gsic.titech.ac.jp/sites/default/files/TSUBAME_SPECIFICATIONS_en_0.pdf (accessed on 20 September 2013).
16. NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Available online: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf (accessed on 20 September 2013).
17. NVIDIA Corp. NVIDIA CUDA C-Programming Guide Ver. 5.5. Available online: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (accessed on 20 September 2013).
18. Li, T.; Narayana, V.K.; El-Araby, E.; El-Ghazawi, T. GPU Resource Sharing and Virtualization on High Performance Computing Systems. In Proceedings of the 40th International Conference on Parallel Processing; IEEE, 2011; pp. 733–742.
19. NVIDIA Corp. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Available online: <http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf> (accessed on 20 September 2013).
20. Guevara, M.; Gregg, C.; Hazelwood, K.; Skadron, K. Enabling Task Parallelism in the CUDA Scheduler. In Proceedings of the Workshop on Programming Models for Emerging Architectures (PMEA); Citeseer, 2009.
21. Mangharam, R.; Saba, A. Anytime Algorithms for GPU Architectures. In Proceedings of the IEEE Real-Time Systems Symposium (IEEE RTSS) 2011, 29 November - 2 December 2011, Vienna, Austria, 2010; Article No. 31.

22. Peters, H.; Koper, M.; Luttenberger, N. Efficiently Using a CUDA-enabled GPU as Shared Resource. In Proceedings of the IEEE 10th International Conference on Computer and Information Technology (CIT), 2010; IEEE, 2010; pp. 1122–1127.
23. S_GPU Project Webpage. <http://sgpu.ligforge.imag.fr> (accessed on 20 September 2013).
24. Li, T.; Narayana, V.K.; El-Ghazawi, T. Accelerated High-Performance Computing Through Efficient Multi-Process GPU Resource Sharing. In Proceedings of the 2012 ACM International Conference on Computing Frontiers (CF'12); ACM, 2012; pp. 269–272.
25. Ravi, V.; Becchi, M.; Agrawal, G.; Chakradhar, S. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In Proceedings of the 20th International Symposium on High-Performance Parallel and Distributed Computing; ACM, 2011; pp. 217–228.
26. Gupta, V.; Gavriloavska, A.; Schwan, K.; Kharche, H.; Tolia, N.; Talwar, V.; Ranganathan, P. GViM: GPU-accelerated Virtual Machines. In Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing; ACM, 2009; pp. 17–24.
27. Shi, L.; Chen, H.; Sun, J. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. In Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS); IEEE, 2009; pp. 1–11.
28. Giunta, G.; Montella, R.; Agrillo, G.; Coviello, G. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In Proceedings of the 16th International Euro-Par Conference, Ischia, Italy, 31 August - 3 September 2010; D'Ambra, P., Guarracino, M. Talia, D., Eds.; Springer: Berlin-Heidelberg, Germany, 2010; pp. 379–391.
29. Khronos OpenCL Working Group. The OpenCL Specification Ver. 2.0. Available online: <http://www.khronos.org/registry/cl/specs/opencl-2.0-opencl.pdf> (accessed on 20 September 2013).
30. Dagum, L.; Menon, R. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* **1998**, *5*, 46–55.
31. Gummaraju, J.; Coburn, J.; Turner, Y.; Rosenblum, M. Streamware: Programming General-Purpose Multicore Processors Using Streams. In Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems; ACM: New York, NY, USA, 2008; pp. 297–307.
32. Bailey, D.H.; Barszcz, E.; Barton, J.T.; Browning, D.S.; Carter, R.L.; Dagum, L.; Fatoohi, R.A.; Frederickson, P.O.; Lasinski, T.A.; Schreiber, R.S.; Simon, H.D.; Venkatakrishnan, V.; Weeratunga, S.K. The NAS Parallel Benchmarks. *Int. J. High Perform. Comput. Appl.* **1991**, *5*, 63.
33. Malik, M.; Li, T.; Sharif, U.; Shahid, R.; El-Ghazawi, T.; Newby, G. Productivity of GPUs Under Different Programming Paradigms. *Concurr. Comput. Pract. Exp.* **2012**, *24*, 179–191.
34. Black, F.; Scholes, M. The Pricing of Options and Corporate Liabilities. *J. Polit. Econ.* **1973**, *81*, 637–654.
35. Visual Molecular Dynamics Program Webpage. <http://www.ks.uiuc.edu/Research/vmd> (accessed on 20 September 2013).