Efficient GPU Spatial-Temporal Multitasking

Yun Liang, *Member, IEEE*, Huynh Phung Huynh, *Member, IEEE*, Kyle Rupnow, *Member, IEEE*, Rick Siow Mong Goh, *Member, IEEE*, and Deming Chen, *Senior Member, IEEE*

Abstract—Heterogeneous computing nodes are now pervasive throughout computing, and GPUs have emerged as a leading computing device for application acceleration. GPUs have tremendous computing potential for data-parallel applications, and the emergence of GPUs has led to proliferation of GPU-accelerated applications. This proliferation has also led to systems in which many applications are competing for access to GPU resources, and efficient utilization of the GPU resources is critical to system performance. Prior techniques of temporal multitasking can be employed with GPU resources as well, but not all GPU kernels make full use of the GPU resources. There is, therefore, an unmet need for spatial multitasking in GPUs. Resources used inefficiently by one kernel can be instead assigned to another kernel that can more effectively use the resources. In this paper we propose a software-hardware solution for efficient spatial-temporal multitasking and a software based emulation framework for our system. We pair an efficient heuristic in software with hardware leaky-bucket based thread-block interleaving to implement spatial-temporal multitasking. We demonstrate our techniques on various GPU architecture using nine representative benchmarks from CUDA SDK. Our experiments on Fermi GTX480 demonstrate performance improvement by up to 46% (average 26%) over sequential GPU task execution and 37% (average 18%) over default concurrent multitasking. Compared with the state-of-the-art Kepler K20 using Hyper-Q technology, our technique achieves up to 40% (average 17%) performance improvement over default concurrent multitasking.

Index Terms—GPU, spatial, temporal, multitasking, resource allocation

1 Introduction

The continuous demand for increased performance in latency, throughput, and power/energy efficiency has been a driving factor in the adoption of GPUs for computation acceleration. This rapid adoption of GPUs has led to a corresponding proliferation of applications that employ them—a single machine may have many applications competing for access to GPU resources, and the rise of datacenter and cloud-computing environments with resources shared by many simultaneous users has led to an even larger scale of many applications competing for access to resources.

Using NVIDIA's terminology, a GPU is composed of multiple streaming multiprocessors (SMs), each of which has multiple streaming processor (SP) cores. The SP cores within an SM share large amounts of registers and memory resources; in total a GPU architecture with some SMs may support tens of thousands of simultaneously executing threads. Due to this computational power, GPUs have been widely used for acceleration [1], [2], [3].

- *Y. Liang is with the Center for Energy-Efficient Computing and Applications, School of EECS, Peking University, 5 Yiheyuan Road, Haidian District, Beijing 100871, P.R. China. E-mail: ericlyun@pku.edu.cn.*
- *H.P. Huynh and R.S.M. Goh are with the A*STAR Institute of High Performance Computing, Computing Science Department, 1 Fusionopolis Way, #16-16 Connexis North, Singapore 138632. E-mail: {huynhph, gohsm}@ihpc.a-star.edu.sg.*
- *K. Rupnow is with the Advanced Digital Science Center, 1 Fusionopolis Way #08-10 Connexis North, Singapore 138632. E-mail: k.rupnow@adsc.com.sg.*
- *D. Chen is with the University of Illinois at Urbana-Champaign, 1308 W Main St Urbana, IL 61801. E-mail: dchen@illinois.edu.*

Manuscript received 16 Jan. 2013; revised 2 Jan. 2014; accepted 27 Feb. 2014. Date of publication 23 Mar. 2014; date of current version 6 Feb. 2015. Recommended for acceptance by S. Ranka. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2014.2313342

Programming models such as CUDA and OpenCL have made GPUs widely accessible for general purpose computing. Programmers write data-level parallel tasks (*kernels*) that will execute on the GPU—for performance optimization, programmers assume exclusive access to GPU resources. With time-shared access to GPUs, this assumption is reasonable: each application gets exclusive access while executing a kernel, and GPU access is time-shared at the granularity of kernel execution latency. However, not all GPU applications require full use of GPU resources [4]. Depending on the architecture's achievable memory bandwidth, GPU applications may saturate performance with only a fraction of SM resources. Thus, for such applications the GPU's resources may also be shared spatially. Concurrent execution of more than one kernel at a time may improve resource utilization and overall performance.

Spatial multitasking (concurrent kernel execution) is not a new idea: NVIDIA's architecture has rudimentary concurrency support in hardware since the Fermi architecture [5]. NVIDIA's implementation allows a user to specify that certain kernels are independent and can execute simultaneously, but does not allow the user to specify the relative resource allocation between concurrently executing kernels. The Kepler architecture [6] improves Fermi model somewhat by introducing multiple independent kernel queues. If the executing kernel from one queue does not consume all resources, other queues can use the remaining resources. Still, such a system does not have the scheduling and allocation flexibility to consider the relative importance of kernels or differences in the bandwidth consumption and resource requirement. Indeed, we will demonstrate that even with the most promising Hyper-Q feature, the default concurrent multitasking on Kepler can not efficiently utilize the resource and improve the performance.

We present the case for an improved spatial-temporal multitasking model. In our proposed model many simultaneous applications can submit kernels to independent queues (similar to the Kepler architecture). However, in our model the device scheduler would make the spatial-temporal multitasking decisions based on the kernel behavior properties, and leaky-bucket based management [7] of the thread-block execution queues would implement the scheduling decision. Thus, we propose modest changes to three portions of the GPU execution hierarchy:

- Compile-time profiling of each kernel's computation latency and bandwidth use properties.
- Software implementation of concurrency modeling and exploration in GPU device scheduler.
- Hardware implementation of leaky-bucket based Quality of Service (QoS) management of thread-block execution queues.

Profiling of kernel properties and concurrency modeling and exploration are all software-based so we can directly implement them. However, current GPU hardware does not allow us to make our proposed hardware changes—therefore, we use a software framework to emulate the GPU execution behavior. Our emulation system can effectively interleave multiple kernels' execution in the same manner that leaky-bucket based queue management would. The kernels are transformed by the emulation framework but still executed and measured on real GPU hardware. Emulation system requires additional (offline) code modifications that would not be necessary if the proposed hardware were available. Although we do not implement the leaky-bucket algorithm in hardware for this paper, it has been widely employed in prior hardware implementations in network processing [7], and we assume that this functionality is also feasible for implementation in the GPU.

In this paper we present an efficient spatial-temporal multitasking model for a single GPU. Tasks submitted to the GPU are from multiple independent applications, with one application per queue. Kernel dependencies within an application are handled implicitly by queue ordering. Thus, the first kernels from multiple application queues constitute a set of independent kernels. Within this set, a kernel may execute concurrently with other kernels or sequentially. Temporal multitasking determines the number of execution phases, where kernels in the same phase execute concurrently. Spatial multitasking determines the relative SM resource allocation for the kernels in a phase. Because there are many possible spatial-temporal schedules, we develop fast and accurate performance estimation and an efficient heuristic for exploring options; this portion of the solution would be in the GPU device scheduler. To enforce the driver's spatial and temporal schedule decisions, we use leaky-bucket based thread block interleaving.

This paper contributes to the state-of-the-art in GPU optimization with

- A software-hardware solution for efficient GPU multitasking, which allows kernels from multiple applications to share the GPU spatially and temporally.
- A software heuristic paired with fast and accurate performance estimation metric for exploring possible spatial-temporal multitasking schedules which is within 6% of the optimal (in the cases where the optimal are feasible to compute).
- A hardware leaky-bucket based thread-block interleaving method to implement the spatial-temporal multitasking.
- A software emulation framework that can implement leaky-bucket based thread-block interleaving on any GPU platform whether natively supported or not.

We demonstrate our technique using nine representative benchmarks from CUDA SDK. Our technique achieves performance improvement of up to 46% (average 26%) over sequential GPU task execution and 37% (average 18%) over the default Fermi GTX480 concurrent multitasking implementation. Compared with the state-of-the-art Kepler K20 using Hyper-Q technology, our technique achieves up to 40% (average 17%) performance improvement over default concurrent multitasking.

2 Motivation

Intuitively, multitasking is useful when a GPU kernel's performance saturates while some GPU resources remain unused. This happens in two situations: the first, and less common case is when a compute-bound kernel has fewer thread blocks than SM resources—it cannot use the entire GPU simply because there is not enough work to do. The second case is the common case: a kernel has sufficiently high memory bandwidth use that the achievable bandwidth is saturated with a subset of SMs.

Fig. 1 shows how the performance and memory bandwidth scale with the number of SMs for nine applications on NVIDIA GTX480. The details on how the profile data is collected are described in Section 3. As shown, the memory-bound kernels (e.g., *VectorAdd*, *FastWalshTransform*, etc.) fail to scale linearly with the number of SMs. Thus, when one kernel cannot fully utilize the GPU's resources, spatial multitasking can improve system performance by allocating the un-utilized or under-utilized resources to other kernels.

In Fig. 2, we illustrate the potential benefit of spatial-temporal multitasking by showing the execution latency for each possible spatial-temporal schedule, and NVIDIA's concurrent interface for the example of *BlackScholes* and *VectorAdd*. For a set of two kernels, there are two possible temporal schedules: (1) two kernels execute sequentially, or (2) two kernels execute concurrently. For concurrent execution, there are different possible spatial SM allocations between the two kernels. NVIDIA Fermi GTX480 architecture [5] contains 15 SMs. Thus, there are 14 different spatial SM allocations for two kernels. The execution time of spatial-temporal multitasking solutions are collected using the emulation framework to be described in Section 4.

We observe that the performance of the two kernels depends significantly on the SM allocation between the two kernels. Both our spatial multitasking and the default concurrent multitasking improve on the sequential GPU task execution through improved resource utilization. However, our spatial multitasking supports flexible SM allocation to better fulfill different kernel needs. In comparison, the default Fermi multitasking only provides an interface to specify the simultaneous running kernels, but does not

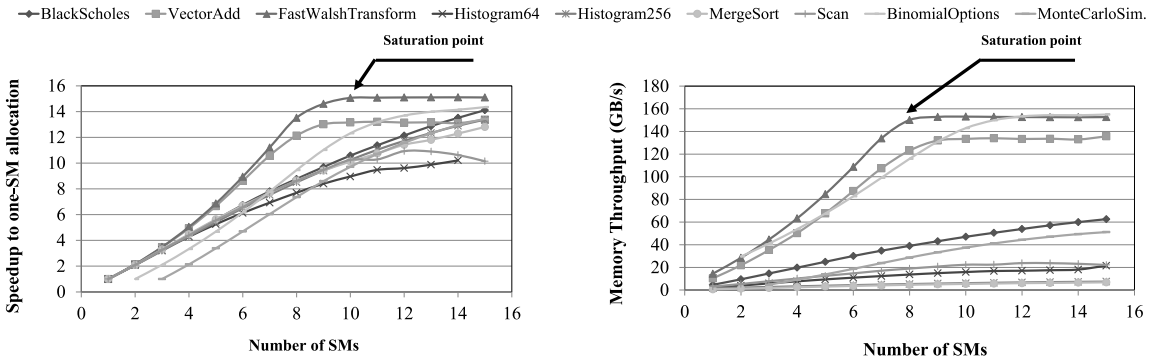


Fig. 1. Speedup and global memory bandwidth versus number of SMs. For speedup, the results are normalized to one SM.

allow users to specify the SM allocation. Similarly, Kepler architecture allows multiple CPU cores to launch work on the GPU simultaneously, but again doest not provide SM allocation flexibility. By exploring various SM allocations, the chosen optimal SM allocation achieves higher performance than the default multitasking. Note that the exact multitasking implementation details (e.g., temporal and spatial scheduling/SM allocation algorithms) are not disclosed by NVIDIA. It is possible that NVIDIA's default concurrent multitasking is not equivalent to any of the spatial-temporal multitasking solution in our search space.

Given a large number of kernels, as we will demonstrate in the experiments section, temporal multitasking will also affect the performance. As the number of kernels in the set of independent kernels grows, the number of possible spatial-temporal multitasking solutions grows rapidly. This leads to both opportunities for improved performance and challenges in quickly and accurately choosing the appropriate spatial-temporal multitasking.

3 Spatial and Temporal Multitasking

In our model, kernels from multiple independent applications are submitted to the GPU for execution as shown in Fig. 3. This is analogous to Kepler's Hyper-Q that uses multiple task queues and allow concurrent kernel execution from multiple applications. Note that Kepler's Hyper-Q indeed increases kernel level parallelism compared to Fermi, but does not improve scheduling and allocation flexibility or consider bandwidth, computation and resource use as what we do in our solution. Our technique is complementary to existing GPUs, but is a proposed solution for future GPUs.

Different spatial allocations can have very different performance (see Fig. 2). Thus, a critical requirement for spatial-temporal multitasking is to understand the behavior of kernels under different resource allocation. Such behaviors can be understood via profiling during compilation; we will discuss our profiling technique later in this section. The first kernels from multiple application queues constitute a set of independent kernels ready for spatial-temporal scheduling. In practice, applications may start and finish at different times. Thus, the set of kernels changes dynamically depending on the applications and system behavior. There are many possible spatial-temporal schedules and we must quickly identify a good spatial-temporal schedule at runtime. The determined spatial-temporal multitasking is enforced by the hardware leaky-bucket based thread block interleaving (described in Section 4).

Now, we discuss the challenges related to spatial-temporal multitasking. We consider the set of independent kernels $\mathcal{K}$ that consists of N kernels $\mathcal{K} = \{k_1, \ldots, k_N\}$, and we assume there are M homogeneous SMs in the GPU.

Temporal multitasking. Given the kernel set $\mathcal{K}$, we must schedule all the kernels in it. We define a set of one or more concurrently executing kernels as a *phase*, and all of the kernels in $\mathcal{K}$ must be divided into N or fewer phases. Each kernel has different computation and memory requirements, and thread structures. The number of temporal multitasking

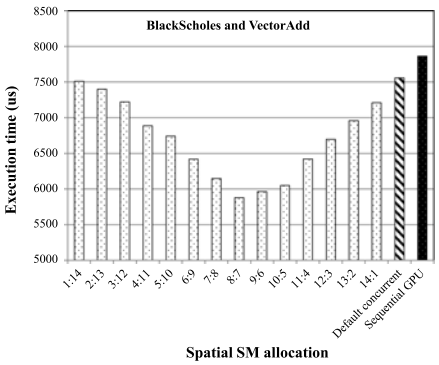


Fig. 2. Impact of different spatial-temporal multitasking. *a:b* on the *x*-axis represents the allocated SMs to the two kernels, respectively. The *default concurrent* bar represents the default NVIDIA's concurrent multitasking supported by GTX480.

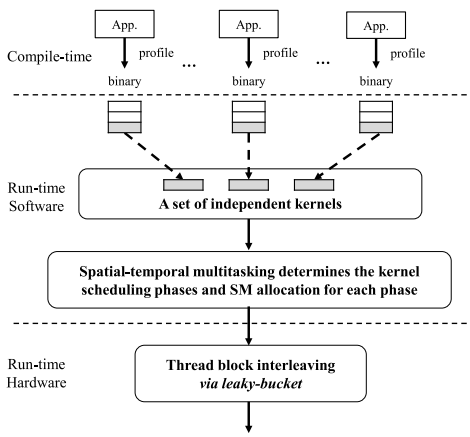


Fig. 3. Spatial-temporal multitasking system overview.

solutions for N kernels is Bell number, which is asymptotic bounded by $O(e^{N(ln(N))})$ [8]. See Appendix A for the details of the problem complexity.

Spatial multitasking. Given a set of concurrently executing kernels (e.g., the kernels in the same phase), find the SM allocation (distribution of GPU resources among the kernels) that performs the best. The number of possible solutions of allocating M SMs to n kernels is $\binom{M-1}{n-1}$. For example, in a GTX480 (15 SMs), with two kernels executing concurrently there are 14 ways of allocating SMs in which each kernel is allocated at least one SM. See Appendix A for the details of the problem complexity.

Performance estimation. To select a good spatial-temporal multitasking schedule, it is neither feasible nor desirable to exhaustively test all the solutions within an online scheduler. Therefore, it is also important to have performance estimation that can quickly estimate performance of candidate solutions to obviate the need to empirically measure the performance on GPU hardware.

To target these problems, we develop a performance model-guided heuristic approach with low overhead. In order to determine the efficiency of our heuristic, we also implement an optimal (exhaustive) solution that compiles and measures actual runtime of each possible spatial-temporal schedule. In the experiments, we demonstrate that our heuristic is close to the optimal in terms of performance improvement, but it is important to emphasize that the exhaustive solution is not suggested or desirable for integration into the scheduler.

3.1 Optimal Solution

For the set of independent kernels as defined above, there are many possible temporal schedules; with different resource requirements and scaling properties, each combination of kernels performs differently. Thus, the optimal solution must examine all possible temporal schedules, and for each phase of each schedule, examine all possible spatial schedules/SM allocations. The result of the exhaustive evaluation is a spatial-temporal schedule of $\mathcal{K}$ and the corresponding SM allocation for the kernels in each of the schedule's phases.

Spatial multitasking. Given a set of kernels in a phase, the optimal solution needs to enumerate and measure each possible SM allocation, and select the allocation that yields the minimum execution latency. For each candidate solution, we use the leaky-bucket framework (Section 4) to measure its performance. See Algorithm 2 in Appendix B for the details.

Temporal multitasking. With the optimal Spatial Multitasking (SM allocation) algorithm, we can find the best SM allocation for any kernel set in a phase; now, we must explore all the possible temporal schedules. For example, with three independent kernels $\mathcal{K} = \{k_1, k_2, k_3\}$, there are five different ways to schedule the kernels into three or fewer phases ($\{k_1, k_2, k_3\}$, $\{k_1\}|\{k_2, k_3\}$, $\{k_2\}|\{k_1, k_3\}$, $\{k_3\}|\{k_1, k_2\}$, $\{k_1\}|\{k_2\}|\{k_3\}$), where kernels in brackets are scheduled concurrently as a single phase, and $|$ separates two schedule phases. Phases cannot overlap execution, therefore the phase ordering does not affect the temporal schedule's latency. Then, with the optimal latency of each phase, the sum of phase latencies can be easily computed to find the optimal latency of a given temporal schedule. The optimal spatial-temporal schedule is then simply the schedule that results in the minimum sum latency. The optimal solution must enumerate all the temporal schedules and selects the best schedule. Moreover, we only consider the valid schedules that no phase have more than M independent kernels as each kernel must be allocated at least one SM. See Algorithm 3 in Appendix B for the details.

3.2 Heuristic Approach

The optimal solution is infeasible for a system that makes these scheduling decisions online. Thus, in this section we develop an efficient heuristic approach that is suitable for integration within a device scheduler. We name our efficient spatial-temporal multitasking as *STM*.

The keys to an efficient heuristic are two-fold: first, we develop a fast performance estimation technique to reduce the cost of evaluating each solution; second, we develop an efficient heuristic to reduce the number of solutions evaluated by excluding sub-solutions previously proven to be sub-optimal. Because the key reason for spatial multitasking is that some kernels may saturate memory bandwidth with fewer than the maximum available SMs, it is important that our performance estimation must estimate memory bandwidth use. As input to our performance estimation, we use profile data of each kernel for both latency and memory bandwidth as shown in Fig. 1. We also need to profile the requested global memory size of each kernel as the sum of global memory size for the kernels in the same phase must fit in the GPU global memory.

For each kernel k_i, we first use CUDA profiler [9] to measure its requested global memory size, G_i. In this work, we assume the global memory is statically allocated for the entire kernel before kernel starts execution. Note that the global memory size does not vary with the number of allocated SMs as all the thread blocks have to be executed regardless how many SMs are used. But the latency and memory bandwidth of a kernel vary with the number of allocated SMs as shown in Fig. 1. To collect them, we use a dummy kernel approach. More clearly, we execute each kernel k_i concurrently with a dummy kernel d_i that has computation but does not progress k_i's computation or consume memory bandwidth. The dummy kernel d_i has the same number of threads per thread block as the kernel k_i and the same thread block latency as the thread blocks of k_i, but the total number of thread blocks varies in order to create the desired SM allocation for k_i. Let us use TB_i to represent the number of thread blocks of kernel k_i. If we want to measure the latency and memory bandwidth of kernel k_i given m SMs, then we need to create $\lceil \frac{TB_i \times (M-m)}{m} \rceil$ thread blocks for d_i, where M is the number of SMs in the GPU. Then, the thread blocks of k_i and d_i are interleaved similar to the mechanism employed to emulate leaky-bucket (Section 4).

The profiling process is then a three step procedure as follows:

1) Measure kernel k_i with exclusive GPU access to compute average thread block execution latency.

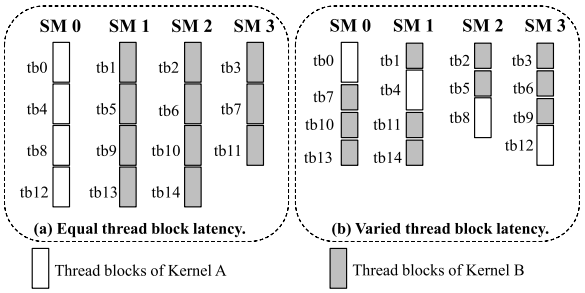


Fig. 4. Dynamic round robin with equal and varied thread block latency. There are four SMs in this example.

2) Tune dummy kernel d_i's thread block latency to the average latency of k_i as measured.
3) For each SM allocation (totally $M-1$ allocations for two kernels k_i and d_i), profile with an appropriate number of interleaved dummy thread blocks.

Each of the measurement uses the CUDA profiler in order to measure the latency and memory bandwidth. Thus, for each kernel k_i, its latency profile R_i and bandwidth profile B_i are gathered for each possible SM allocation. $R_i[m]$ and $B_i[m]$ return the latency and achieved memory bandwidth of kernel k_i given an allocation of m SMs, respectively. This profiling is performed only once per kernel, and is suitable for inclusion in the compilation process, and a reasonable prerequisite for spatial-temporal multitasking as shown in Fig. 3. For the applications with control flow divergence [10], thread block latency and memory bandwidth may be dependent on the input data. For such cases, a more detailed profiling may be necessary. For this work, we use the same input for profiling and evaluation.

We now use the profile data to estimate the latency of concurrently executing kernels. First, we will consider only the latency profile data: the latency of a phase depends on the kernels in the phase, their SM allocation, and the total number of thread blocks for each kernel. For kernel k_i, we compute its per-thread-block latency given an allocation of m SMs as $\frac{R_i[m]\times m}{TB_i}$. The behavior of NVIDIA's thread block scheduler is unknown, but previous studies show that a dynamic round robin schedule that assigns thread blocks in order of their thread block id correlates well to actual kernel performance [11], [12]. Therefore, after the initial thread block assignment to fill the SMs, remaining thread blocks are inserted into a queue and dispatched to SMs as they become available. Fig. 4 shows an example of dynamic round robin with equal and varied thread block latency. Thread blocks of kernel A and B are interleaved to enforce the SM allocation (details described in Section 4). The estimated total latency is the finishing time of the last thread block. In the following, we use $DRR()$ to represent the estimated latency using the dynamic round robin schedule.

For our heuristic approach, we use a constructive approach. Thus, some of the sub-solutions that we explore will use fewer SMs than the total available SMs. Our performance estimation is used to estimate the performance of both sub-solutions and final solutions. First, we define the set of kernels for concurrent execution:

Definition 1 (Concurrent Kernel Set Configuration). *A concurrent kernel set configuration $\mathcal{C}$ is a set of 2-tuples: $\{\langle k_1, sm_1\rangle, \ldots, \langle k_n, sm_n\rangle\}$, where the set of kernels $\{k_1, \ldots, k_n\} \in 2^{\mathcal{K}}$. sm_i is the number of SMs allocated to kernel k_i and $sm_i < M$.*

Based on the definition, the concurrent kernel set may use only a portion of the total available SMs (e.g., the sum of sm_i may be less than M), or use all available SMs. In both cases, we need to evaluate the performance improvement of concurrent execution given a particular set of kernels and their SM allocation. Given a concurrent kernel set configuration $\mathcal{C}$, we compute its execution latency improvement compared to sequential execution as follows:

$$imp(\mathcal{C}) = seq\_lat(\mathcal{C}) - con\_lat(\mathcal{C}), \tag{1}$$

where $seq\_lat(\mathcal{C})$ is the sequential execution latency of the kernels in $\mathcal{C}$ given the total number of SMs used by the configuration $\mathcal{C}$ and $con\_lat(\mathcal{C})$ is the concurrent execution latency of the kernels in $\mathcal{C}$.

Let $\mathcal{C} = \{\langle k_1, sm_1\rangle, \ldots, \langle k_n, sm_n\rangle\}$. The total used SMs of $\mathcal{C}$ is $SM_{\mathcal{C}} = \sum_{i=1\ldots n} sm_i$. We compute $seq\_lat(\mathcal{C})$ as follows:

$$seq\_lat(\mathcal{C}) = \sum_{i=1}^{i=n} R_i[SM_{\mathcal{C}}]. \tag{2}$$

For $con\_lat(\mathcal{C})$, we use dynamic round robin as shown in Fig. 4 to estimate the performance, but we also need to consider the effects of memory bandwidth which might affect the performance. When concurrently executing a set of kernels in $\mathcal{C}$, their requested memory bandwidth may be higher than the maximum bandwidth of the GPU device. Thus, we define a penalty factor $penalty(\mathcal{C})$ as follows:

$$penalty(\mathcal{C}) = \begin{cases} \frac{\sum_{i=1\ldots n} B_i[sm_i]}{Pb}, & \text{if } \sum_{i=1\ldots n} B_i[sm_i] > Pb, \\ 1, & \text{otherwise,} \end{cases} \tag{3}$$

where Pb is the peak memory bandwidth of the GPU device. That is, if the requested bandwidth use of $\mathcal{C}$ is greater than the achievable bandwidth, all kernels are uniformly slowed down in proportion to the excess demand.

Then, the final latency is estimated as

$$con\_lat(\mathcal{C}) = \begin{cases} \infty, & \sum_{i=1\ldots n} G_i > Gb, \\ DRR(\mathcal{C}) \times penalty(\mathcal{C}), & \text{otherwise.} \end{cases} \tag{4}$$

If the requested global memory size of the kernels in $\mathcal{C}$ exceeds the GPU global memory limit (Gb), such execution will be aborted in CUDA. For this case, we set the concurrent execution latency of $\mathcal{C}$ to ∞; this gives worse improvement compared to sequential execution according to Equation (1). Otherwise, the latency of concurrent execution is estimated as the product of the $DRR()$-computed latency and the penalty factor.

Using performance estimation metric $con\_lat(\mathcal{C})$, we can efficiently estimate performance of sub-solutions or final solutions. Now, we need to reduce the number of solutions evaluated. Our heuristic approach based on dynamic programming uses a constructive approach for generating solutions that can exclude sub-solutions previously proven to be sub-optimal. Note that the improvement is calculated based on the number of SMs used by this configuration, not the

total SMs; this important detail is the key to allow constructive building of configurations.

Algorithm 1 describes our heuristic approach. Overall, it iteratively selects the set of kernels for concurrent execution in a phase and their corresponding SM allocation. In each iteration, the dp_select subroutine selects a concurrent kernel set configuration $\mathcal{C}$ with the maximum execution latency improvement and the kernels in $\mathcal{C}$ forms a phase. Then, the set of kernels in $\mathcal{C}$ are removed from the set of kernels to be scheduled and we proceed to the next iteration. This process continues until all the kernels are chosen. Algorithm 1 uses the metric defined in Equation (1) to compare different solutions.

Algorithm 1: Heuristic Approach

```
1  let K = 𝒦;
2  phase = 0;
3  while |K| > 0 do
4  |   𝒞 = dp_select(K) ;
5  |   remove the kernels in 𝒞 from K ;
6  |   temporal_schedule[phase] = 𝒞;
7  |   phase = phase + 1 ;
8
9  function(dp_select(K = {k_1, ..., k_n}))
   // M is the total number of SMs, n is the number of kernels in K ;
10 for j ← 1 to M do
11 |   Config[1][j] = {⟨k_1, j⟩};
12 |   Improve[1][j] = 0;
13
14 for i ← 2 to n do
15 |   for j ← 1 to M do
16 |   |   Improve[i][j] = 0;
17 |   |   for m ← 0 to j do
18 |   |   |   temp = imp(Config[i − 1][j − m] ∪ {⟨k_i, m⟩}) ;
19 |   |   |   if temp > Improve[i][j] then
20 |   |   |   |   Improve[i][j] = temp;
21 |   |   |   |   Config[i][j] = Config[i−1][j−m] ∪ {⟨k_i, m⟩}
   |   |   |   |   ;
22 |   |   |
23 |   |
24 |
25
26 return Config[n][M] ;
```

Given a set of kernels $K = \{k_1, \ldots, k_n\}$, we use dynamic programming to return a concurrent kernel set configuration $\mathcal{C}$ with maximum execution latency improvement ($imp(\mathcal{C})$ Equation (1)). This dynamic programming implementation is based on tables for possible configurations ($Config[][]$) and improvements ($Improve[][]$) that are populated constructively. The tables are built from top to down and left to right. Thus, $Config[i][j]$ returns the kernel concurrent set configuration that achieves the maximum execution latency improvement given j SMs for the first i kernels and $Improve[i][j]$ is the corresponding execution latency improvement. By definition,

$$Improve[i][j] = \max_{0 \le m < j} imp(Config[i-1][j-m] \cup \{k_i, m\}).$$

Note that although we reuse the allocation decision from prior "best" configurations, the improvement estimate is recomputed using $imp(\mathcal{C})$ sub-routine because the bandwidth requirements and latency variations can affect the overall latency and thus improvement. The resulting solution from dp_select may select a subset of the kernels from K and the remaining unselected kernels will be evaluated again on the next iteration of the dp_select algorithm. As we will demonstrate in the experiments, the runtime of our heuristic approach is very small. In practice, this small overhead can be completely hidden by performing the spatial-temporal multitasking analysis while the GPU processing is taking place for a prior set of kernels.

4 Emulation Framework

After the spatial-temporal multitasking analysis determines the kernel scheduling phases and the SM allocation for each phase, we need to implement those decisions for the kernel execution. In our proposed system, we use hardware leaky-bucket based thread block interleaving to implement the scheduling decisions (Section 4.1). To emulate the leaky-bucket based hardware, we use software emulation (Section 4.2). However, the software emulation is executed and measured on the real GPU hardware.

4.1 Thread Block Interleaving via Leaky-Bucket

To implement the temporal schedule, we can schedule all the thread blocks based on their phase number (e.g., schedule the thread blocks of the kernels in phase i before the thread blocks of the kernels in phase $i + 1$).

To implement the spatial schedule, we use the leaky-bucket algorithm. Historically, the leaky-bucket algorithm [7] has been used for supporting quality of service of multiple streams of network data competing for access to the physical network. This network model can be applied to our situation: each independent kernel has an ordered queue of thread blocks (analog to independent queues of network stream packets), and for quality of service we want to enforce an allocation decision (number of SMs allocated versus total SMs available). We implement the desired SM allocation via thread block interleaving done by a leaky-bucket based thread block interleaving.

The thread block interleaving order determines the relative allocation of SMs. For example, given two kernels A and B, if we wish to allocate nine SMs to A and six SMs to B, we can enqueue nine of A's thread blocks followed by six of B's thread blocks. However, this thread block ordering pattern may have sequences of allocated thread blocks that significantly deviate from the desired allocation, especially if the thread block execution latencies of kernels A and B do not match. The leaky-bucket algorithm performs fine-grained interleaving of the thread-blocks, and can provide a stronger guarantee that any interval of successive allocations (after a startup period) will not vary from the desired allocation proportion by more than a single SM. In the leaky-bucket algorithm, each kernel to be executed concurrently has an independent bucket filling with tokens at a rate of one token per cycle. Each bucket's maximum capacity is computed to correspond to the desired relative allocation rate, where larger capacities mean less frequent allocations. When a bucket reaches its capacity, one thread block is placed in the queue, and the bucket is emptied. The concept of "cycles" for this leaky-bucket algorithm does not denote time; they are used only to determine the interleave order for arbitrary desired allocation.

Fig. 5 shows an example of thread block interleaving via leaky-bucket algorithm for a system with three SMs. In this example, there are two kernels: kernel A has four thread blocks and kernel B has six thread blocks. One

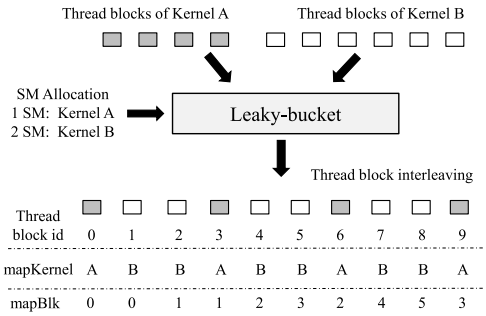


Fig. 5. Thread block interleave via leaky-bucket. $mapKernel$ and $mapBlk$ are used in the software emulation framework.

possible SM allocation is to allocate one SM to kernel A and two SM to kernel B. Based on this SM allocation, the leaky-bucket algorithm interleaves the thread blocks as shown in Fig. 5. In this example, we assume equal thread block latency for all the kernels. If the thread blocks have varied latency, the dynamic round robin algorithm will perform a simple runtime load balancing between SMs, as shown in Fig. 4b. Thus, the average number of SMs it actually occupies is different from the desired SM allocation. To solve the varied latency problem, we can compensate the leaky-bucket filling rate to ensure that the actual allocation meets the desired allocation considering latency variation. In prior networking implementations, this ability allowed for compensation of different packet sizes from different network streams.

4.2 Software Emulation: Code Transformation

For each temporal schedule phase (e.g., a set of concurrently executing kernels), our emulation framework creates a monolithic scheduler that merges the thread blocks of the kernels in the set. The exact behavior of scheduling thread blocks to execute on SMs has not been disclosed by NVIDIA, but as discussed previously a dynamic round robin scheduling policy that assigns thread blocks in the order of their thread block id correlates well to the kernel performance. Thus, the scheduler effectively implements an SM resource allocation corresponding to how the thread blocks of the kernels are interleaved. To emulate the behavior of the hardware leaky-bucket implementation, we use leaky-bucket in advance to determine the thread block interleaving order and the scheduler implements that ordering of the kernels' thread blocks.

The implementation of our software emulation for spatial multitasking involves the creation of the scheduler (e.g., a schedule kernel that invokes the original independent kernels) and source code transformation to both the CPU host and original GPU kernel code. The scheduler merges the parameters of the kernels and assigns the workload based on two mappings—$mapBlk$ and $mapKernel$. These two mappings are created on the host (CPU) before invocation of the scheduler on the GPU. The output of the leaky-bucket algorithm is the $mapKernel$ array. Given a global thread block id bid in the scheduler, $mapKernel[bid]$ returns the kernel id (e.g., kernel A or B). The $mapBlk$ array can be derived from the $mapKernel$ array by tracking the number of thread-blocks already scheduled for each kernel. Thus, $mapBlk[bid]$ returns the corresponding local thread block id in the original kernel. Fig. 5 shows an example of $mapKernel$ and $mapBlk$.

Fig. 6 shows a code example of our software emulation framework using two kernels scheduled concurrently in a phase. The scheduler combines the parameters of the two kernels and requires extra parameters including the thread block and kernel mappings ($mapBlk$ and $mapKernel$), and the original thread block and grid dimension. The scheduler uses an if-else statement to call different kernels based on the kernel mappings. The thread block and kernel mapping step is executed on the CPU before the scheduler is invoked. For the original GPU kernel, we need to replace the current block/grid identifier with the original block/grid identifier and guard the computation using if statement $threadId < blkDim$ to ensure that only the threads in the right range are executed. Appendix C provides more details on the thread structure of the scheduler kernel and the source code transformation.

5 Limitations

In this section, we discuss the limitations of our spatial-temporal multitasking technique for GPUs.

Software emulation framework. The proposed leaky-bucket based thread block interleaving requires hardware

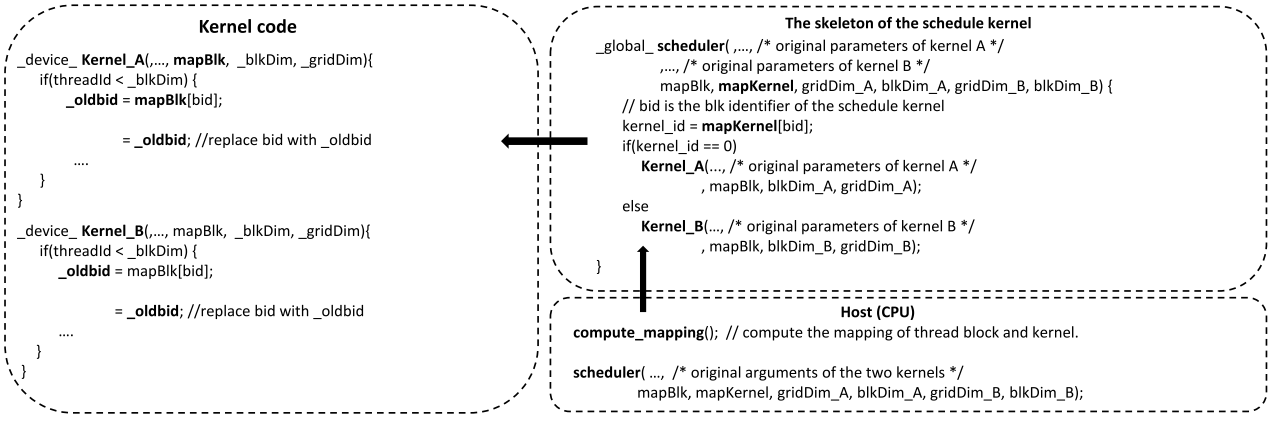


Fig. 6. Software emulation framework.

support. We cannot make the architectural changes to the current GPUs. In order to test our idea, we develop a software framework to emulate the thread block interleaving. However, our software emulation framework suffers from two performance issues. First, we have to use extra variables and arrays (e.g., $mapBlk$, $mapKernel$, $kernel\_id$, etc., as shown in Fig. 6) to store the thread block interleaving results based on leaky-bucket algorithm. The extra variables and arrays increase the register and global memory usage, which may cause potential drop in throughput as the number of simultaneous active threads may reduce. With hardware implementation, dedicated hardware tables and registers would solve this problem.

Second, by statically merging multiple kernels together, we create a monolithic scheduler kernel with maximum resource footprint (e.g., register, shared memory) among any constituent kernel. For example, let us assume our scheudler kernel K merges kernels A and B together; register is the limiting resource for all the three kernels (A, B, K). Let us also assume a thread block of kernel A and B requires 4K and 8K registers, respectively. Hence, a thread block of kernel K requires 8K (max(4, 8)) registers if we ignore the register increase due to the extra variables (e.g., $kernel\_id$) usage. Given the 32K register file capacity on Fermi (GTX480), we can either execute eight thread blocks of kernel A or four thread blocks of kernel B simultaneously. However, for the monolithic kernel K, we can execute only four thread blocks simultaneously. This limitation may cause potential drop in the throughput and overall performance. This problem can be solved with hardware implementation as we do not need to create the monolithic kernel statically; the hardware scheduler can interleave the thread blocks from different kernels with diverse resource requirements at runtime. Note that the software emulation framework would not be part of the proposed system if the hardware implementation of leaky-bucket based queue management is available. In the experiments, we will use the software emulation framework to evaluate our technique and demonstrate performance speedup even with these limitations.

Kernel behaviors. The proposed multitasking technique is useful for a set of concurrent tasks with different behaviors (e.g., memory-bound and compute-bound). Our spatial multitasking technique does not benefit the set of kernels with only one type of behavior as there is no opportunity to improve the resource utilization. In our multitasking algorithm (Algorithm 1), we use sequential execution as the base line implementation. If multitasking does not provide performance benefit, we will choose sequential kernel execution.

Performance models. Our performance estimation in conjunction with heuristic algorithm are used online to determine the multitasking execution. To minimize the runtime overhead, we ignore the effects of performance variation among thread blocks due to control flow divergence, L2 cache contention, etc., in our performance model.

6 Experimental Results

We use nine benchmarks from CUDA SDK [9]: *VectorAdd (va), BinomialOptions (bo), BlackScholes (bs), Histogram256 (h256), MergeSort (mer), Scan (sh), Histogram64 (h64), MonteCarloSim (mci), FastWalshTransform (fwt2).* These applications are representative workloads as they exhibit a variety of behavior in memory bandwidth and compute latency. Our proposed spatial-temporal multitasking technique can exploit such variation among the kernels. But any other applications or workloads with such variations can be used for evaluation as well.

We propose two algorithms for spatial-temporal multitasking: an optimal solution and an efficient heuristic approach. The efficient heuristic approach is labeled *STM* in the following. The optimal solution is only useful for a small set of kernels offline, but our STM solution reduces the search space and performance evaluation overhead significantly that it can be integrated into the device scheduler. Note that we use the emulation framework for our proposed algorithms. The kernels are transformed by the emulation framework but still executed and measured on real GPU hardware. We evaluate our techniques using NVIDIA Fermi GTX480. GTX480 provides interface for concurrent multitasking using CUDA streams. We also test on other GPU architectures including Fermi C2070, Kepler K20 and Kepler GTX680.

In the following, we perform five sets of experiments to evaluate our STM solution: first, we compare STM with sequential GPU kernel execution (e.g., executes GPU kernels sequentially without spatial multitasking) and NVIDIA default concurrent multitasking interface. Second, we compare STM with Optimal. Third, we evaluate the accuracy of our performance estimation technique. Fourth, we show the overhead and scalability of our approach using a large number of kernels. Finally, we show the results of our spatial-temporal multitasking on different GPU architectures—NVIDIA Fermi C2070, Kepler K20 and Kepler GTX680.

Comparison with state-of-the-art solutions. Given kernels with different behaviors (e.g., compute-bound and memory-bound), we create different kernel sets with sizes varying from 2 to 5. We compare our STM solution with sequential GPU kernel execution and the NVIDIA default concurrent multitasking. Fig. 7 shows the performance improvement (percentage) over sequential kernel execution for the optimal solution, STM solution, and default concurrent multitasking on GTX480. The *X*-axis lists the different combinations of kernel sets. Overall, compared to sequential kernel execution, our STM solution achieves up to 46% (on average 26%) performance improvement while the default concurrent multitasking achieves up to 34% (on average 11%).

Compared to sequential kernel execution, both our STM solution and the default concurrent multitasking improve performance by allocating under-utilized resources to other kernels. Furthermore, our STM solution achieves up to 37% (average 18%) performance improvement over the default concurrent multitasking. Our STM solution temporally schedules the kernels into phases and allows flexible SM allocation to better meet different kernels' needs for each phase. Hence, we achieve more performance improvement. The performance improvement varies for different kernel sets. For the kernel sets that achieve high performance improvement, their kernels tend to be

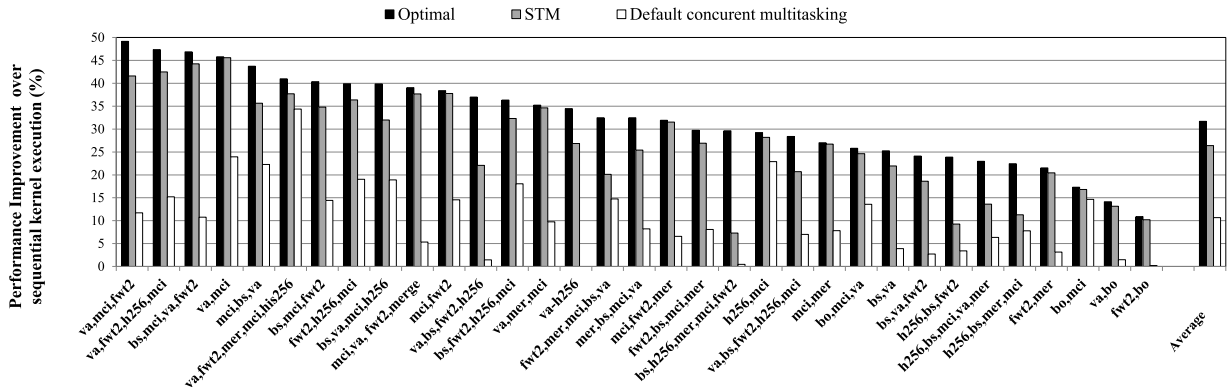


Fig. 7. Performance comparison of different multitasking techniques on GTX480.

complementary in terms of compute and memory resource utilization.

STM versus optimal. We compare our STM solution with the optimal solution for the cases where the optimal is feasible. As shown in Fig. 7, our STM solution tends to be close to the optimal solution; STM on average is 6% away from the optimal solution.

Accuracy of performance metric. The accuracy and efficiency of our STM solution is partially contributed by our accurate performance metric ($con\_lat(C)$ in Section 3.2). Here, we support this claim with concrete experimental results. Fig. 8 compares our estimated performance with actual performance for a three-kernel set. The *Y*-axis shows the execution time while the *X*-axis shows different SM allocations. As shown, our performance estimation accurately predicts the trend well; for each SM allocation our estimation is close to the actual performance, too. For other test cases, the performance is similarly well correlated.

Scalability. Table 1 shows the running time of the optimal and STM solution with different size of kernel sets. The optimal solution is based on exhaustive search and empirical measurement, where we set an upper limit of 24 hours runtime before canceling evaluation. The runtime of optimal solution increases exponentially as the size of kernel sets increases. It takes almost one day for a five-kernel set while our STM solution returns the results within seconds. To prove further the scalability of our STM solution, we increase the size of kernel sets up to 50 by replicating the kernels. Our STM solution still returns results in the order of tens of seconds. This overhead is very low compared to the long runtime of the GPU kernels, representing only an average of 1% of the execution time of the kernels. In practice, this low overhead can be completely hidden by performing the spatial-temporal multitasking analysis while the GPU processing is taking place for a prior set of kernels. Table 1 also shows the average performance improvement of our STM solution over sequential GPU execution. Our STM solution consistently achieves speedup throughout different sizes of kernel sets. As seen by the data in Fig. 7, the speedup can vary depending on the sets of kernels to be scheduled, but these results demonstrate that our STM solution can efficiently divide kernels into spatial-temporal schedules even for large number of kernels. Unfortunately due to the size of the sets it was not feasible to exhaustively determine the optimal schedule for larger problem sets.

Portability. Our spatial-temporal multitasking can be used with any existing GPU architecture. Here, we first evaluate our technique using another NVIDIA Fermi architecture: Fermi C2070. Our STM solution improves performance by up to 46% (average 25%) compared to sequential GPU kernel execution while the default concurrent multitasking achieves up to 26% (on average 13%) improvement. NVIDIA Kepler architecture improves the concurrency model of Fermi by using multiple independent kernel queues called Hyper-Q. We also evaluate our technique by comparing with two different Kepler architectures: Kepler K20 and Kepler GTX680. K20 architecture is featured with Hyper-Q technology. The default concurrent multitasking is implemented using CUDA stream. Compared to default concurrent multitasking, our STM solution achieves by up to 33% (average 18%) and 40% (average 17%) improvement on Kepler GTX680 and Kepler K20, respectively. See

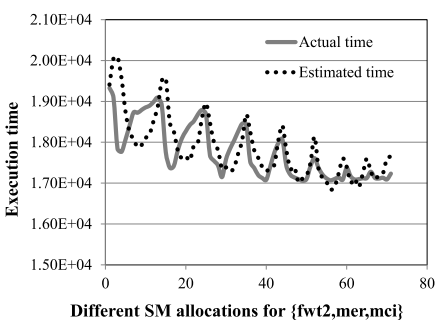


Fig. 8. Accuracy of our performance metrics.

TABLE 1
Optimal versus STM Running Time (Seconds)

No. Kernels	Opt	STM	Avg STM Improvement
2	91.2	0.03	24.8%
3	1141.5	0.2	27.3%
4	8683.9	0.36	28.2%
5	66560.8	0.584	17.4%
10	N.A.	3.948	35.5%
15	N.A.	6.305	17.2%
20	N.A.	13.939	17.3%
30	N.A.	40.387	34.3%
50	N.A.	56.5418	31.4%

Appendix D for the details of experiments on Fermi C2070, Kepler K20 and Kepler GTX680.

7 Related Work

Although GPUs promise high performance, tuning GPU for high performance is not a trivial task [13]. The state-of-the-art of GPU performance modeling and tuning techniques focus on analytic performance models [14], [15], computation optimization [10], [16], [17] and memory access patterns optimizations [18], [19] for a single kernel/task.

There exist very limited studies on multitasking for GPUs. Traditionally, GPUs are time-shared only at the granularity of kernels. Recently, Adriaens et al. demonstrate that not all GPU applications fully use GPU resources [4]. They propose a few heuristics for the spatial multitasking and demonstrate performance improvement compared to sequential kernel execution using GPGPU-sim [20]. In contrast, our multitasking solution allows different kernels to share the GPU resources both spatially and temporally. For spatial multitasking, our solution supports flexible SM allocation through leaky-bucket thread block interleaving. Finally, we evaluate our techniques using a software emulation framework on the real GPU architecture.

Guevara et al. developed a compile time concurrency model [21]. However, in their mechanism, the programmer can not control the SM allocation. Thus, it is only suitable for the small kernels which do not have enough parallelism to fully utilize the GPU. Cederman and Tsigas evaluated different scheduling and load balancing implementations on GPUs [22], [23]. Their implementation provides coarse-grained control over GPU kernels. However, their work did not consider the resource waste of a kernel due to memory bandwidth saturation. In contrast, our technique provides fine-grained thread block interleaving to improve the resource utilization and overall performance. We compare our technique with the state-of-the-art multitasking solution on NVIDIA Fermi and Kepler architectures.

8 Conclusion

GPUs are increasingly important for heterogenous computing due to their tremendous computing power for accelerating data-parallel applications. As more and more applications are using GPUs, it is critical to support efficient multitasking on GPUs. In this work, we propose a software-hardware solution for efficient GPU multitasking, which allows kernels from multiple applications to share the GPU spatially and temporally. We demonstrate the performance improvement of our technique on various GPU architectures using nine representative applications. Our demonstration on the Fermi GTX480 architecture accelerates performance by up to 46 percent (average 26 percent) over sequential GPU kernel execution and 37 percent (average 18 percent) over default concurrent multitasking. Compared with the state-of-the-art Kepler K20 using Hyper-Q technology, our technique achieves up to 40 percent (average 17 percent) performance improvement over default concurrent multitasking.

Appendix A

Problem Complexity

A1. Temporal Multitasking

The number of potential temporal schedules is the sum of the number of schedules for each possible schedule length. For example, with four kernels, the number of schedules would be the sum of the number of schedules of one phase, two phases, three phases and four phases. This value is a Bell number—the number of ways to partition a set of N items into N or fewer sets. The Bell number is most simply defined recursively, where $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_n$. The asymptotic bound of Bell numbers is $O(e^{N(ln(N))})$. Note that there is a restriction that no phase can have more than M independent kernels because each kernel must be allocated at least one SM.

A2. Spatial Multitasking

The number of possible SM allocations scales with the number of kernels, but is limited by the restriction that each kernel must receive at least one SM. If we number the SMs, an SM allocation can be described by the index of transition point between SMs allocated to one kernel and SMs allocated to the next kernel in the phase. Thus, given n $(n \leq M)$ independent kernels in a phase, the number of possible SM allocations is the number of ways of selecting $n-1$ transition points. Therefore, we define the number of possible allocations of M SMs to n kernels as $\binom{M-1}{n-1}$.

Appendix B

Algorithm Details

Algorithm 2 presents the details of the optimal algorithm of spatial multitasking. Given a set of kernels in a phase, Algorithm 2 returns the optimal SM allocation. Algorithm 3 presents the details of the optimal algorithm of temporal multitasking. Given a set of kernels, Algorithm 3 returns the optimal spatial-temporal multitasking schedule.

Algorithm 2: Spatial Multitasking

```
1  let K = {k_1, ..., k_n} be the set of kernels in a phase ;
2  let A be the set of possible SM allocations of K ;
3  min_lat = INF ;
4  foreach a ∈ A do
5      // use the emulation framework to measure the current
       SM allocation ;
6      lat = measure(K, a) ;
7      if lat < min_lat then
8          min_lat = lat;
9          record⟨K, a, min_lat⟩;
10
11
```

Appendix C

Emulation Framework Implementation Details

To launch the scheduler, two important parameters—block and grid dimension need to be specified. The block dimension ($blkDim$) specifies the number of threads per thread

block and the grid dimension ($gridDim$) specifies the number of thread blocks. The total number threads is just the product of $blkDim$ and $gridDim$. The scheduler merges the independent kernels at thread block level. Thus, the number of thread blocks of the scheduler is the sum of thread blocks of all the merged kernels. The number of threads per thread block of the scheduler is the maximum threads per thread block among the kernels.

Algorithm 3: Temporal Multitasking

```
1  let S be the set of possible schedules of K ;
2  best_lat = INF ;
3  foreach s ⊆ S do
4      if is_valid(s) then
5          lat = 0;
6          let P be the set of phases in S ;
7          foreach p ⊆ P do
8              lat = lat + get_minimal_lat(p); // Algorithm 2
9          if lat < best_lat then
10             best_lat = lat;
11             let a be the set of SM allocations of P ;
12             best_sol = ⟨s, a⟩;
13
14
15
```

We need to perform two types of transformations to the original kernel. First, we need to add extra parameters and replace the current block/grid identifier with the previous block/grid identifier. More clearly, three additional parameters including the thread block mapping ($mapBlk$), the original grid ($\_gridDim$) and block ($\_blkDim$) dimension of the kernel are required. Then the statement "$\_oldbid = mapBlk[bid]$" is inserted to map the scheduler's thread block id to the kernel's thread block id's. The transformation then replaces all occurrences of bid in the kernel with $\_oldbid$. Similarly, $blkDim$ and $gridDim$ are replaced with $\_blkDim$ and $\_gridDim$, if they are used.

Second, we surround the computation everywhere except $\_syncthreads()$ with if statement as shown in Fig. 9. With the guarded if statement, only the threads in the actual thread block range are executed if the kernel's actual block dimension is smaller than the merged kernel. $\_syncthreads()$ is not included in the if statement to ensure the correct synchronization among threads.

Kernel code

```
_device_ Kernel(,…, mapBlk,  _blkDim, _gridDim)
{
   if(threadId < _blkDim)
   {
     _oldbid = mapBlk[bid];

             = _oldbid; //replace bid with _oldbid
             ….

   }
   __syncthreads();

   for(……)
   {
      if(threadId < _blkDim)
      {
         ….
      }
      __syncthreads();

      if(threadId < _blkDim)
      {
         ….
      }
   }
```

Fig. 9. Kernel transformations.

APPENDIX D
PORTABILITY EXPERIMENTS

Fig. 10 shows the performance improvement compared to the sequential GPU kernel execution for different kernel sets for Fermi C2070. As shown, both our STM and the default concurrent multitasking consistently improve the performance. But our STM solution achieves more performance improvement compared to the default concurrent multitasking. More clearly, our STM solution improves performance by up to 46% (average 25%) compared to sequential GPU kernel execution while the default concurrent multitasking achieves up to 26% (on average 13%) improvement. The performance improvement is a little different from Fermi GTX480. The difference in improvement comes from the different architecture parameters such as memory clock speed, double-precision floating-point performance, number of SMs and memory bandwidth. Finally, there are a few kernel sets that the default concurrent multitasking only achieves marginal performance improvement. Thus, if the marginal improvement could not compensate the performance overhead of the concurrent multitasking, it is

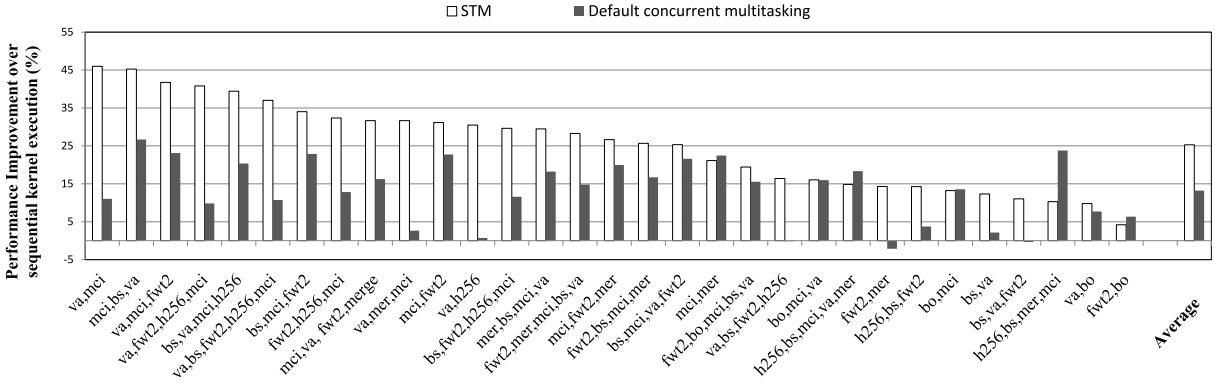


Fig. 10. Performance improvement compared to the sequential kernel execution on NVIDIA Fermi C2070.

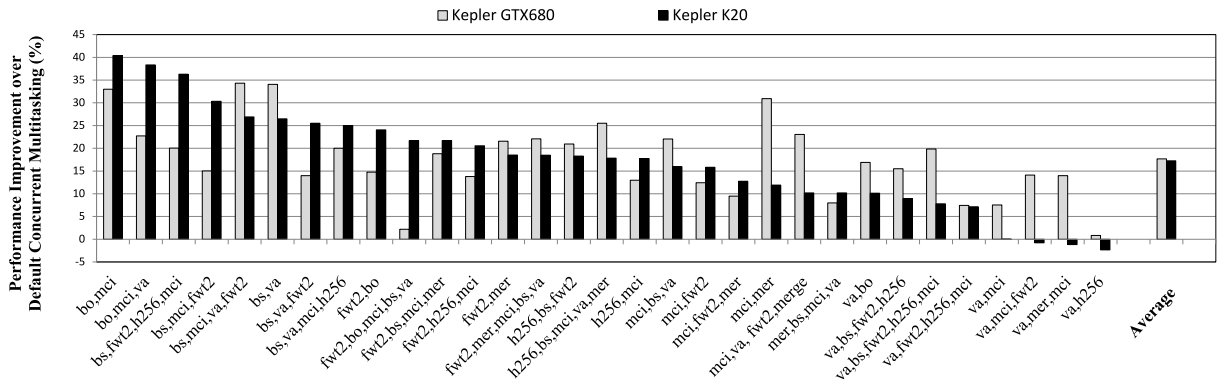


Fig. 11. Performance improvement compared to the default concurrent multitasking on Kepler GTX680 and K20.

possible that the default concurrent multitasking slightly degrades the performance as shown in Fig. 10.

Fig. 11 shows the performance improvement of our STM solution compared to the default concurrent multitasking of Kepler architecture for different kernel sets. As shown, our solution still achieves substantial improvement compared to the default concurrent multitasking of Kepler architecture. More clearly, our solution achieves by up to 33% (average 18%) and 40% (average 17%) improvement on Kepler GTX680 and Kepler K20, respectively.

We also notice that for a few kernel sets, our STM solution is slightly worse than the default concurrent multitasking as shown in Figs. 10 and 11. This is due to the performance overhead introduced by our software emulation framework as discussed in Section 5. However, it is important to emphasize that this is a limitation only for the emulation framework. There will be no such limitation if the hardware implementation is available. Overall, our emulation framework demonstrates substantial speedup even with performance overhead caused by the emulation framework.

Acknowledgments

This work was partially supported by the National Natural Science Foundation of China (No. 61300005) and by the CFAR Center, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by A*STAR of Singapore through Advanced Digital Sciences Center (ADSC) and the Institute of High Performance Computing (IHPC). Huynh Phung Huynh is co-first author.

References

[1] Y. Liang, Z. Cui, S. Zhao, K. Rupnow, Y. Zhang, D. L. Jones, and D. Chen, "Real-time implementation and performance optimization of 3d sound localization on GPUs," in *Proc. Des., Autom. Test Eur. Conf. Exhib.*, 2012, pp. 832–835.

[2] J. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[3] J. S. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. S. Meredith, J. Rogers, P. C. Roth, K. Spafford, and S. Yalamanchili, "Keeneland: Bringing heterogeneous GPU computing to the computational science community," *IEEE Comput. Sci. Eng.*, vol. 13, no. 5, pp. 90–95, Sep./Oct. 2011.

[4] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *Proc. IEEE 18th Int. Symp. High-Performance Comput. Archit.*, 2012, pp. 1–12.

[5] NVIDIA Fermi Architecture. [Online]. Available: http://www.nvidia.com/object/fermi_architecture.html.

[6] NVIDIA Kepler Architecture. [Online]. Available: http://www.nvidia.com/object/nvidia-kepler.html.

[7] J. S. Turner, "New directions in communications (or which way to the information age?)," *IEEE Commun. Mag.*, vol. 24, no. 10, pp. 8–15, Oct. 1986.

[8] N. G. de Bruijin, *Asymptotic Methods in Analysis*. New York, NY, USA: Dover, 1981.

[9] *NVIDIA CUDA Programming Guide, Version 3.2*, NVIDIA, Santa Clara, CA, USA..

[10] Z. Cui, Y. Liang, K. Rupnow, and D. Chen, "An accurate GPU performance model for effective control flow divergence optimization," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 83–94.

[11] S. Ryoo, "Program optimization strategies for data parallel many-core processors," PhD thesis, Univ. of Illinois at Urbana-Champaign, Champaign, IL, 2008.

[12] G. Ruetsch and P. Micikevicius, *Optimizing matrix transpose in CUDA*, NVIDIA. Santa Clara, CA, USA, 2009.

[13] S. Ryoo et al., "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. 13th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2008, pp. 73–82.

[14] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 152–163.

[15] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proc. 15th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2010, pp. 105–114.

[16] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen, "Streamlining GPU applications on the fly: Thread divergence elimination through runtime thread-data remapping," in *Proc. 24th ACM Int. Conf. Supercomput.*, 2010, pp. 115–126.

[17] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2007, pp. 407–420.

[18] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2010, pp. 86–97.

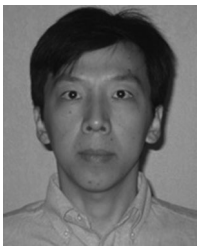
[19] X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2013, pp. 516–523.

[20] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA worloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Performance Anal. Syst. Softw.*, 2009, pp. 163–174.

[21] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the cuda scheduler," in *Proc. Workshop Program. Models Emerging Archit.*, 2009, pp. 69–76.

[22] D. Cederman and P. Tsigas, "On dynamic load balancing on graphics processors," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Symp. Graph. Hardware*, 2008, pp. 57–64.

[23] D. Cederman and P. Tsigas, "On sorting and load balancing on GPUs," *Proc. ACM SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 11–18, Jun. 2009.



Yun Liang received the BS degree in software engineering from Tongji University, Shanghai, China, in 2004, and the PhD degree in computer science from the National University of Singapore in 2010. He worked as a research scientist in Advanced Digital Science Center, University of Illinois Urbana-Champaign between 2010 and 2012. He has been an assistant professor in the School of Electronics Engineering and Computer Science, Peking University since 2012. His current research interests include GPU architecture and optimization, heterogeneous computing, embedded system, and high-level synthesis. He serves as a technical committee member for ASPDAC, DATE, CASES, and so on. He is the TPC subcommittee chair for ASPDAC'13. He received the Best Paper Award in FCCM'11 and Best Paper Award nominations in CODES+ISSS'08 and DAC'12. He is a member of the IEEE.

Huynh Phung Huynh received the PhD in computer science from the National University of Singapore in 2010. He is currently leading HPC group at the Institute of High Performance Computing, A*STAR, Singapore. His research interest include high-performance computing (HPC) research such as developing productivity tools for GPU/many-core computing, and big data analytics. He is a member of the IEEE.

Kyle Rupnow received the PhD degree in electrical engineering from the University of Wisconsin-Madison in 2010. He previously received a Sandia National Laboratories Excellence in Engineering Fellowship and the US National Science Foundation (NSF) Graduate Research Fellowship honorable mention. He is currently a research scientist in ADSC, UIUC, and an assistant professor in Nanyang Technological University, Singapore. He is also received the Gerald Holdridge Award for Tutorial Development and the University of Wisconsin-Madison Capstone PhD Teaching Award. He is a member of the IEEE.

Rick Siow Mong Goh received the PhD degree in electrical and computer engineering from the National University of Singapore. He is the director of the Computing Science Department at the A*STAR Institute of High Performance Computing (IHPC). At IHPC, he leads a team of more than 70 scientists in performing world-leading scientific research, developing technologies to commercialization, and engaging and collaborating with industry. The research focus areas include high-performance computing (HPC), distributed computing, data analytics, interactive interaction technologies, and computational social cognition. His expertise is in discrete event simulation, parallel and distributed computing, and performance optimization and tuning of applications on large-scale computing platforms. He is a member of the IEEE.

Deming Chen received the BS degree in computer science from the University of Pittsburgh, Pennsylvania, in 1995, and the MS and PhD degrees in computer science from the University of California at Los Angeles in 2001 and 2005, respectively. He worked as a software engineer first between 1995 and 1999, and then between 2001 and 2002. He has been an associate professor in the Electrical and Computer Engineering Department of the University of Illinois, Urbana-Champaign, since 2011. He is a research associate professor in the Coordinated Science Laboratory and an affiliate associate professor in the Computer Science Department. His current research interests include high-level synthesis, nanosystems design and nanocentric CAD techniques, GPU optimization, heterogeneous system programming, FPGA synthesis, SoC design, and computational biology. He is a technical committee member for a series of conferences and symposia, including FPGA, ASPDAC, ICCD, ISQED, DAC, ICCAD, DATE, ISLPED, FPL, and so on. He is the TPC subcommittee chair for ASPDAC'09-11 and the TPC track chair/cochair for ISVLSI'09, ISCAS'10-11, VLSI-SoC'11, ICCAD'12, and ICECS'12. He is the general chair for SLIP'12, the CANDE workshop chair in 2011, and the program chair for the PROFIT workshop in 2012. He is an associated editor for the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *ACM Transactions on Design Automation of Electronic Systems*, *IEEE Transactions on Very Large Scale Integration Systems*, *IEEE Transactions on Circuits and Systems I, Journal of Circuits, Systems and Computers*, and the *Journal of Low Power Electronics*. He received the Achievement Award for Excellent Teamwork from Aplus Design Technologies in 2001, the Arnold O. Beckman Research Award from UIUC in 2007, the US National Science Foundation (NSF) CAREER Award in 2008, and five Best Paper Awards in ASPDAC'09, SASP'09, FCCM'11, SAAHPC'11, and CODES+ISSS'13. He received the ACM SIGDA Outstanding New Faculty Award in 2010. He is a senior member of the IEEE.