

Kernelet: High-Throughput GPU Kernel Executions with Dynamic Slicing and Scheduling

Jianlong Zhong, Bingsheng He

Abstract—Graphics processors, or GPUs, have recently been widely used as accelerators in shared environments such as clusters and clouds. In such shared environments, many kernels are submitted to GPUs from different users, and throughput is an important metric for performance and total ownership cost. Despite recently improved runtime support for concurrent GPU kernel executions, the GPU can be severely underutilized, resulting in suboptimal throughput. In this paper, we propose *Kernelet*, a runtime system to improve the throughput of concurrent kernel executions on the GPU. Kernelet embraces transparent memory management and PCI-e data transfer techniques, and dynamic slicing and scheduling techniques for kernel executions. With slicing, Kernelet divides a GPU kernel into multiple sub-kernels (namely *slices*). Each slice has tunable occupancy to allow co-scheduling with other slices for high GPU utilization. We develop a novel Markov chain-based performance model to guide the scheduling decision. Our experimental results demonstrate up to 31% and 23% performance improvement on NVIDIA Tesla C2050 and GTX680 GPUs, respectively.

Index Terms—GPGPU, Performance Modeling, Task Scheduling, Kernel Slicing, Markov Chain

1 INTRODUCTION

The graphics processing unit (or GPU) has become an effective accelerator for a wide range of applications from computation-intensive applications (e.g., [17], [18], [28]) to data-intensive applications (e.g., [4], [9], [11]). Compared with multi-core CPUs, new-generation GPUs can have much higher computation power in terms of FLOPS and memory bandwidth. Due to their immense computation power and memory bandwidth, GPUs have been integrated into clusters and cloud computing infrastructures. In Top500 list of June 2013, two out of the top ten supercomputers are with GPUs integrated. Amazon and Penguin have provided virtual machines with GPUs. In both cluster and cloud environments, GPUs are often shared by many concurrent GPU programs (or *kernels*) (most likely submitted by multiple users). Additionally, to enable sharing GPUs remotely, a number of software frameworks such as rCUDA [3] and V-GPU [33] have been developed. This paper studies whether and how we can improve the throughput of concurrent kernel executions on the GPU.

Throughput is an important optimization metric for efficiency and the total ownership cost of GPUs in such shared environments. First, many GPGPU applications such as scientific and financial computing tasks are usually throughput oriented [5]. A high throughput leads to high performance and productivity. Second, compared with CPUs, GPUs are still expensive devices. A high throughput not only means a high resource utilization but also a low total ownership cost. That might be one of

the reasons that GPUs are usually deployed and shared to handle kernels from multiple users.

Recently, we have witnessed the success of GPGPU research. However, most studies focus on single-kernel optimizations (e.g., new data structures [10] and GPU friendly computing patterns [6]). Despite the fruitful research, a single kernel usually severely under-utilizes the GPU. This severe underutilization is mainly due to the inherent memory and computation behavior of a single kernel (e.g., random memory accesses and lack of instruction level parallelism). In our experiments, we have studied eight common kernels (details are presented in Section 5). On C2050, their average *IPC* is 0.52, which is far from the optimal value (1.0). Their memory bandwidth utilization is only ranging from 0.02% to 14%.

Recent GPU architectures like NVIDIA Fermi architecture supports concurrent kernel executions, which allows multiple kernels to be executed on the GPU simultaneously if resources are allowed. In particular, Fermi adopts a form of *cooperative* kernel scheduling. Other kernels requesting the GPU must wait until the kernel occupying the GPU voluntarily yields control. Here, we use NVIDIA CUDA's terminology, simply because CUDA is nowadays widely adopted in GPGPU applications. A kernel consists of multiple executions of *thread blocks* with the same program on different data, where the execution order of thread blocks is not defined (known as SPMD (Single Program Multiple Data) execution model). On Fermi GPUs, one kernel can take the entire GPU if it has sufficient thread blocks to occupy all the multi-processors (even though it can have severely low resource utilization). Concurrent execution of two such kernels almost degrades to sequential execution of individual kernels. Recent studies [22] on optimizing concurrent kernels mainly focus on kernels with low occupancy (i.e., the thread blocks of a single kernel

• J. Zhong and B. He are with School of Computer Engineering, Nanyang Technological University, Singapore, 639798.
E-mail: jzhong2@ntu.edu.sg, bshe@ntu.edu.sg

cannot fully utilize all GPU multiprocessors). However, the occupancy of kernels is usually high after single-kernel optimizations in practice.

Individual kernels as a whole cannot realize real sharing of the GPU resources. This paper investigates whether we can slice the kernel into small pieces and then co-schedule slices from different kernels for higher GPU resource utilization. One observation is that GPU kernels conform to the SPMD execution model. A kernel execution can usually be divided into multiple *slices*, each consisting of multiple thread blocks. Slices can be viewed as low-occupancy kernels and can be executed simultaneously with slices from other kernels. The GPGPU concurrent kernel scheduling problem is thus converted to the slice scheduling problem.

With slicing, we have two more issues to address. The first issue is on the slicing itself: what is the suitable slice size? How to perform the slicing in a transparent manner? The smallest granularity of a slice is one thread block, which can lead to significant runtime overhead of submitting too many such small slices onto the GPU for execution. To the other extreme, the largest granularity of the slice is the entire kernel, which degrades to the non-sliced execution. The second issue is to select the slices for co-scheduling in order to maximize GPU utilization.

To address those issues, we develop *Kernelet*, a runtime system with dynamic slicing and scheduling techniques to improve the GPU utilization. Kernelet transparently performs memory management and data transfer management between the main memory and the GPU memory. Given the kernels that are ready to execute on the GPU (i.e., the input data are available on the GPU memory), Kernelet dynamically performs slicing on the kernels, and the slices are carefully designed with tunable occupancy to allow slices from other kernels to utilize the GPU resources in a complementary way. For example, one slice utilizes the computation units and the other one utilizes the memory bandwidth. We develop a novel and effective Markov chain based performance model to guide kernel slicing and scheduling decisions. Compared with existing GPU performance models (e.g., [12], [24]) which are limited to a single kernel only, our model are designed to handle heterogeneous workloads (i.e., slices from different kernels). We further develop a greedy algorithm to always co-schedule slices from the two kernels with the highest estimated performance gain.

We have evaluated Kernelet on two latest GPU architectures (Tesla C2050 and GTX680). The GPU kernels under study have different memory and computational characteristics. Experimental results show that 1) our analytical model can accurately capture the performance of heterogeneous workloads on the GPU, 2) our scheduling increases the GPU throughput by up to 31% and 23% on C2050 and GTX680, respectively, 3) our memory command scheduler overlaps up to 99% of data transfer with kernel execution.

Organization. The rest of the paper is organized as follows. We introduce the background and definition of

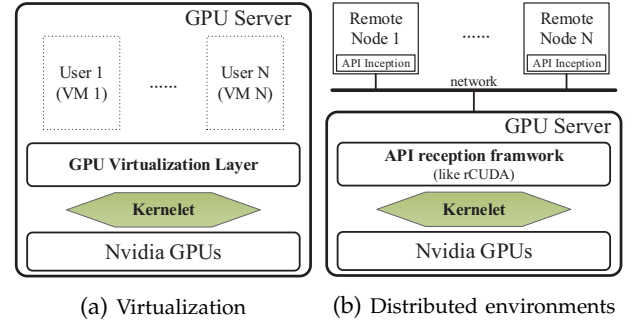


Fig. 1: Application scenarios of concurrent kernel executions on the GPUs.

our problem in Section 2. Section 3 presents the system overview, followed by detailed design and implementation in Section 4. The experimental results are presented in Section 5. We review the related work in Section 6 and conclude this paper in Section 7.

2 BACKGROUND AND PROBLEM DEFINITION

In this section, we introduce the background on GPU architectures, and our problem definition. More details on the NVIDIA GPU architectures can be found in Appendix A of the supplementary file.

2.1 GPU Architectures

This paper focuses on the design and implementation with NVIDIA CUDA. With the introduction of CUDA, a GPU can be viewed as a many-core processor with a set of streaming multi-processors (SM). Each SM has a set of scalar cores, which execute the instructions in the SIMD (Single Instruction Multiple Data) manner. The SMs are in turn executed in the SPMD manner. The program is called *kernel*. Kernelet takes advantage of the concurrent kernel execution capability of new-generation GPUs.

We define the SM occupancy as the ratio of active warps to the maximum active warps that are allowed to run on the SM. Higher occupancy means higher thread parallelism. The aggregated register and shared memory usage of all warps should not exceed the total amount of available registers and shared memory on an SM.

2.2 Problem Definition

Application scenario. We consider two typical application scenarios in shared environments as shown in Figure 1. One is sharing the GPUs among multiple tenants in the virtualized environment (e.g., cloud). As illustrated in Figure 1a, there is usually a GPU virtualization layer integrated with the hypervisor. Figure 1b shows the other scenario, in which GPU servers offer API reception softwares (like rCUDA [3]) to support local/remote CUDA kernel launches. In both scenarios, the GPU faces multiple pending kernel launch requests. Kernelet can be applied to schedule those pending kernels.

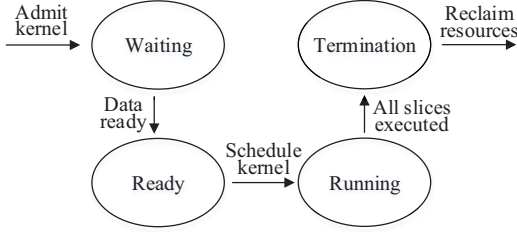


Fig. 2: Kernel state transitions.

Our study mainly considers a single GPU. Kernelet can be extended to multiple GPUs with a workload dispatcher distributing tasks to individual GPUs.

We assume that thread blocks in a kernel are independent with each other. This assumption is mostly true for the GPGPU kernels due to SPMD programming model. Most kernels in NVIDIA SDK and benchmarks like Parboil [27] do not have dependency among thread blocks in the same kernel. This assumption ensures that our slicing technique on a given kernel is safe.

We formally define the terminology in Kernelet.

Kernel. A kernel \mathcal{K} consists of k thread blocks with IDs, $0, 1, 2, \dots, (k-1)$.

Kernel state. A general flow for a GPGPU program consists of three steps. First, the host code allocates GPU memory for input and output data, and copies input data from the main memory to the GPU memory. Second, the host code starts the kernel on the GPU. The kernel performs the task on the GPU. Third, when the kernel execution is done, the host code copies results from the GPU memory to the main memory. Thus, we define the following states for a kernel (as illustrated in Figure 2): (1) *waiting*: the kernel is in the waiting state, if its input data is not totally available on the GPU. Initially, the kernel is in the waiting state. (2) *ready*: all the input data for the kernel execution are available on the GPU. (3) *running*: the kernel is sliced and its slices are being scheduled for execution. (4) *termination*: all slices of the kernel finish execution and its memory resources can be reclaimed.

Slice. A slice is a subset of the thread blocks of a launched kernel. Block IDs of a slice is continuous in the block index space. The size of a slice s is defined as the number of thread blocks contained in the slice.

Slicing plan. Given a kernel \mathcal{K} , a slicing plan $\mathcal{S}(\mathcal{K})$ is a scheme slicing \mathcal{K} into a sequence of n slices ($s_0, s_1, s_2, \dots, s_{n-1}$). We denote the slicing plan to be $\mathcal{K}=s_0, s_1, s_2, \dots, s_{n-1}$.

Co-schedule. Co-schedule cs defines concurrent execution of n ($n \geq 1$) slices, denoted as s_0, \dots, s_{n-1} . All the n slices are active on the GPU.

Scheduling plan. Given a set of n kernels $\mathcal{K}_0, \mathcal{K}_1, \dots, \mathcal{K}_n$, a scheduling plan \mathcal{C} ($cs_0, cs_1, \dots, cs_{n-1}$) determines a sequence of co-schedules in their order of execution. cs_i is launched before cs_j if $i < j$. All thread blocks of the n kernels occur in one of the co-schedules once and only once. A scheduling plan embodies a slicing plan for each kernel.

Co-scheduling profit. We define the performance benefit of co-scheduling n kernels to be the *co-scheduling profit* ($CP = 1 - \frac{1}{\sum_{i=0}^{n-1} \frac{cIPC_i}{IPC_i}}$), where IPC_i and $cIPC_i$

are IPC (Instruction Per Cycle) for sequential execution and concurrent execution of kernel i respectively. Our definition is similar to those in the previous studies on CPU multi-threaded co-scheduling [14], [25].

Problem definition. Given a set of kernels for execution, the problem is to determine the optimal scheduling plan (and slicing) so that the total execution time for those kernels is minimized. That corresponds to the maximized throughput. Given a set of n kernels $\mathcal{K}_0, \mathcal{K}_1, \dots, \mathcal{K}_{n-1}$, we aim at finding the optimal scheduling plan \mathcal{C} for a minimized total execution time of $\mathcal{C}(\mathcal{S}_0(\mathcal{K}_0), \mathcal{S}_1(\mathcal{K}_1), \dots, \mathcal{S}_{n-1}(\mathcal{K}_{n-1}))$. Note, in the shared GPU environment, the arrival of new kernels may trigger re-optimization.

3 SYSTEM OVERVIEW

In this section, we present the rationales on the Kernelet design, followed by a system overview.

3.1 Design Rationale

Since kernels are submitted in an ad-hoc manner, the scheduling decision has to be made in real time. The optimization process should take the newly arrived kernels into consideration. Moreover, our runtime system of slicing and scheduling should be designed with light overhead. The overhead of slicing and scheduling should be small compared with their performance gain.

Unfortunately, finding the optimal slicing and scheduling plans is a challenging task. The solution space for such candidate plans is large. Due to the limited PCI-e bandwidth and GPU memory capacity, different memory management and data transfer orders result in different opportunities for co-scheduling optimizations. For slicing a kernel, we have the factors including the number of slices as well as the slice size. For scheduling a set of slices, we can generate different scheduling plans with co-scheduling slices from different kernels. All those factors are added up into a large solution space. Considering newly arrived kernels makes the huge scheduling space even larger.

Due to real-time decision making and light-weight requirements, it is impossible to search the entire space to get a globally optimal solution. The classic Monte Carlo simulation methods are not feasible because they usually exceed our budget on the runtime overhead and violate the real-time decision making requirement. There must be a more elegant compromise between the optimality and runtime efficiency. Particularly, we make the following considerations.

First, inspired by the classical multi-level design on process scheduling in operating systems [26], we decouple the scheduling decisions into two independent levels: *memory command scheduler* on handling memory management and data transfer commands and *kernel*

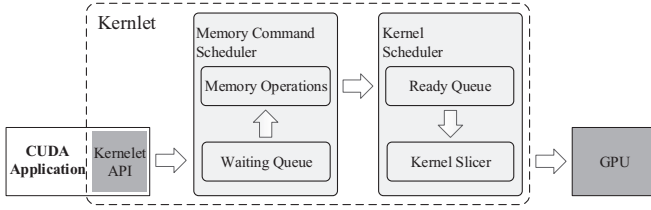


Fig. 3: Design overview of Kernelet.

scheduler on performing co-scheduling on the *ready* GPU kernels. With the two-level scheduler design, the first level aims at fully utilizing the PCI-e bus bandwidth, and the second level aims at fully utilizing the GPU computation resources.

Second, the scheduling considers two kernels only. Previous study [14] on the CPU has shown, when there are more than two co-running jobs, finding the optimal scheduling plan is an NP-complete problem. Following previous studies [14], [22], we make our scheduling decision on co-scheduling two kernels only.

Third, once we choose two kernels to schedule, their slice sizes keep unchanged until either kernel finishes.

3.2 System Overview

We develop Kernelet as a runtime system to generate the slicing and scheduling plans for optimized throughput. Kernelet also automatically manages the device memory to facilitate sharing GPU among multiple applications and exploits opportunities of overlapping PCI-e data transfer with kernel execution.

Figure 3 shows an overview of Kernelet. Kernelet maintains two queue structures: *waiting queue* and *ready queue*. The waiting queue stores kernels in waiting state, and the ready queue stores the kernels in ready state. Initially, kernels are submitted and are temporarily buffered in the waiting queue. The memory command scheduler automatically schedules execution of GPU memory commands. If a kernel has all its input data available on the GPU (i.e., its preceding memory commands in the program are all executed), the kernel is migrated from the waiting queue to the ready queue. The kernel scheduler performs co-scheduling on ready kernels based on our performance model, which estimates the performance of co-scheduling slices from two different kernels in a probabilistic manner. Usually, a kernel is submitted in the form of SASS or PTX code. Once the scheduling plan is obtained, the kernel slicer transforms the kernel code into slices. Then, the slices from the two kernels are co-scheduled and executed on the GPU until either kernel finishes. We will describe the detailed design and implementation of each component in the next section.

4 KERNELET METHODOLOGY

In this section, we first briefly introduce our kernel slicing mechanism. Next, we present details of the kernel scheduler, followed by our performance model. More details on two-level scheduling can be found in Appendix B of the supplementary file.

4.1 Kernel Slicing

The purpose of kernel slicing is to divide a (ready) kernel into multiple slices so that the finer granularity of each slice as a kernel can create more opportunities for time sharing. Moreover, we need to determine the minimum slice size for keeping low slicing overhead (i.e., the difference between the total execution time of all slices and the original kernel execution time). Particularly, we experimentally determine the minimum slice with slicing overhead not greater than $p\%$ of the kernel execution time. In this study, $p\%$ is set to be 2% by default. We focus on the implementation of slicing on PTX or SASS code. Note that, we choose thread blocks as the units for slicing, instead of warps, because warps within the same thread block usually have data dependency with each other, e.g., with the usage of shared memory.

Kernelet performs kernel slicing by transforming a single kernel invocation into a set of invocations. We propose the *index rectification* process to ensure the output is exactly the same as the original kernel invocation. The transformation takes the PTX/SASS code of the kernel as input, and does not require the source code. We give one example and more details on kernel slicing in Appendix B of the supplementary file.

4.2 Scheduling

Algorithm 1 Scheduling algorithm of Kernelet

- 1: Denote \mathcal{R} to be the set of kernels pending for executions;
- 2: **if** A new kernel call \mathcal{K} comes **then**
- 3: Add \mathcal{K} into \mathcal{R} ;
- 4: Perform profiling if the kernel has not been submitted before;
- 5: **while** $\mathcal{R} \neq \text{null}$ **do**
- 6: $\langle \mathcal{K}_1, \mathcal{K}_2, \text{size}_1, \text{size}_2 \rangle = \text{FindCoSchedule}(\mathcal{R})$;
- 7: Denote the co-schedule to be c ;
- 8: Execute c on the GPU;
- 9: **while** \mathcal{R} does not change, and \mathcal{K}_1 and \mathcal{K}_2 both still have thread blocks **do**
- 10: Generate co-schedule according to c and execute it on the GPU;

Proc. $\text{FindCoSchedule}(\mathcal{R})$

Function: generate the optimal co-schedule from \mathcal{R} .

- 1: Generate the candidate space for co-schedules \mathcal{C} ;
 - 2: Perform pruning on \mathcal{C} according to the computation and memory characteristics of input kernels;
 - 3: Apply the performance model (Section 4.3) to compute CP for all the co-schedule in \mathcal{C} ;
 - 4: Obtain the optimal co-schedule with the maximized CP ;
 - 5: Return the result co-schedule;
-

According to our design rationales, our scheduling decision is made on the basis of two ready kernels, to avoid the complexity of scheduling three or more kernels as a whole. Thus, we develop a greedy scheduling algorithm, as shown in Algorithm 1. The scheduling algorithm considers new arrival kernels in Lines 2–4 in the main algorithm. The main procedure calls the procedure *FindCoSchedule* to obtain the optimal co-schedule in Line

6. The co-schedule is represented in four parameters $\langle \mathcal{K}_1, \mathcal{K}_2, size_1, size_2 \rangle$, where \mathcal{K}_1 and \mathcal{K}_2 denote the two selected kernels with slice sizes $size_1$ and $size_2$ respectively.

In Procedure *FindCoSchedule*, we first consider the entire candidate space consisting of co-schedules on pairwise kernel combinations. Because the space may consist of C_n^2 co-schedules (n is the number of kernels for consideration), it is desirable to reduce the search space. Therefore, we perform pruning according to the computation and memory characteristics of input kernels (More details can be found in Appendix B of the supplementary file). After pruning, we apply the performance model (Section 4.3) to estimate the CP for all co-schedules, and pick the one with the maximized CP for execution on the GPU.

4.3 Performance Model

We need a performance model for two purposes: firstly, to select the two kernels for co-scheduling; secondly, to determine the number of thread blocks for each kernel in the co-schedule (i.e., the slice size). Previous performance models on the GPU [12], [24], [30] assume a single kernel on the GPU, and are not applicable to concurrent kernel executions. They generally assume that the thread blocks execute the same instruction in a round-robin manner on an SM. However, this is no longer true on concurrent kernel executions. The thread blocks from different kernels have interleaving executions, which cause non-determinism on the instruction execution flow. It is not feasible to statically predict the interleaving execution patterns for warps from multiple kernels. To capture the non-determinism, we develop a probabilistic performance model to estimate the performance of co-schedule. Our performance model has very low runtime overhead, because it uses a series of simple parameters as input and leverages the Markov chain theory to get the performance of concurrent kernel executions.

Table 1 summarizes the notations used for our performance model.

Since the GPU adopts SPMD model, we use the performance estimation of one SM to represent the aggregate performance of all SMs on the GPU. We model the process of kernel instruction issue as a stochastic process and devise a set of states for an SM during execution. Based on the state transition probabilities of the SM, we develop our Markov chain-based model for single-kernel executions (homogeneous workloads). We then extend the model for concurrent kernel executions (heterogeneous workloads).

For presentation clarity, we begin with our description on the model with the following assumptions, and relax those assumptions at the end of this section. First, we assume that all the memory requests are coalesced. This is the best case for memory performance. We will relax this assumption by considering both coalesced and uncoalesced memory accesses. Second, we assume that

TABLE 1: Notations in the performance model

Para.	Description
W	Maximum number of active warps
$Round$	A warp scheduling cycle that all ready warps are served by the warp scheduler
D_i	Round duration when there are i idle warps
R_m	Memory instruction ratio
$P_{r \rightarrow r}$	Probability that a ready warp remains ready
$P_{r \rightarrow i}$	Probability that a ready warp transits to idle
$P_{i \rightarrow r}$	Probability that an idle warp transits to ready
$P_{i \rightarrow i}$	Probability that an idle warp remains in idle
$N_{r \rightarrow r}$	Number of ready warps that remain ready
$N_{r \rightarrow i}$	Number of ready warps that transit to idle
$N_{i \rightarrow r}$	Number of idle warps that transit to ready
$N_{i \rightarrow i}$	Number of idle warps that remain idle
L	Average memory latency (cycle)
S_i	S_i corresponds the state where i warps are idle on the SM ($i = 0, 1, \dots, W$)
P_{ij}	P is the Markov chain transit matrix. Entry P_{ij} of P represents the probability of transiting from state S_i to state S_j .

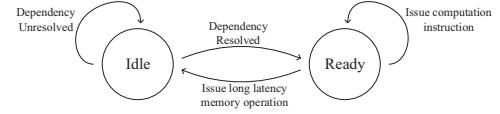


Fig. 4: Warp state transition diagram.

the GPU has a single warp scheduler. We will extend it to the GPU with multiple warp schedulers.

Homogeneous Workloads. We first investigate the performance of a single kernel executed on the GPU and each SM accommodates W active warps at most.

An active warp can be in either of two states: idle or ready. An idle warp is stalled by memory access, while a ready warp has at least one instruction ready for execution. Its state transition is illustrated in Figure 4. When a warp is currently in the ready state, we have two cases for state transitions by definition:

- remaining in the ready state with the probability of $P_{r \rightarrow r} = 1 - R_m$.
- transiting to the idle state with the probability of $P_{r \rightarrow i} = R_m$.

When a warp is currently in the idle state, we also have two cases for state transitions:

- transiting to the ready state with the probability of $P_{i \rightarrow r} = \frac{1}{W-I} = \frac{W-I}{L}$, where I is the number of idle warps on the SM. ($W - I$ warps will be executed in a round, and $W - I$ is also the time duration for the execution of one round.)
- remaining in the idle state with the probability of $P_{i \rightarrow i} = 1 - P_{i \rightarrow r} = \frac{L-W+I}{L}$.

We specifically define the time step and state transition of the Markov chain model to capture the GPU architectural features. Although lack of official documentation, we assume the GPU adopts a round-robin style warp scheduling policy to minimize block synchronization overhead. This assumption is also adopted in previous studies [1], [12]. In each *round*, the warp scheduler polls each warp to issue its ready instructions so all ready

warps can make progress. We model the SM state based on the number of idle warps. We denote S_i to be the SM state where i warps are idle on the SM ($i = 0, 1, \dots, W$). Thus, we consider the state change of the SM in one round and use round as time step in our Markov chain-based model. In each round, every ready warp has an equal chance to issue instructions. In contrast, models for the CPU usually assume that the CPU will keep executing one thread until this thread is suspended and model each cycle as a step.

We use IPC to represent the throughput of the SM. Thus, the number of idle warps on the SM is a key parameter for IPC . More outstanding memory requests usually lead to higher latency because of memory contention [1]. We adopt a linear memory latency model to account for the memory contention effects. We calculate L as $L = a_0 \cdot x + b_0$, where x is the number of outstanding memory requests, and a_0 and b_0 are the constant parameters in the linear model. We follow the previous micro benchmarks on varying the number of outstanding memory requests [1] to obtain a_0 and b_0 .

For homogeneous workload, the probabilities of state transitions are the same for all ready warps in a round. We assume when SM transits from S_i to S_j , $N_{i \rightarrow r}$ idle warps transit to the ready state and $N_{r \rightarrow i}$ ready warps transit to idle state. The following conditions hold by definition.

$$\begin{cases} 0 \leq N_{i \rightarrow r} \leq S_i \\ 0 \leq N_{r \rightarrow i} \leq W - S_i \\ N_{r \rightarrow i} - N_{i \rightarrow r} = S_j - S_i \end{cases} \quad (1)$$

With those constraints, there are multiple possible transitions from S_i to S_j (the transition probability is denoted as P_{ij}). Since the possible transitions are mutually exclusive events, P_{ij} is calculated as the sum of the probabilities of all possible transitions. With all entries of the transition matrix P obtained, we can calculate the steady-state vector of the Markov chain. This is done by finding the eigenvector π corresponding to the eigenvalue one for matrix P [20].

$$\pi = (\gamma_0, \gamma_1, \dots, \gamma_W) \quad (2)$$

In Equation 2, γ_i is the probabilities that the SM is in state S_i in each round, i.e., the probability there are i idle warps in one round. The duration of the time step is $(W - i)$ cycles. In the case $i = W$, the round duration is one, indicating no warp is ready and the SM experiences an idle cycle. Hence, the estimated IPC is the ratio of non-idle cycles given in equation 3, where $\sum_{i=0}^{W-1} \gamma_i (W - i)$ is the total non-idle cycles and γ_W is the total idle cycles.

$$IPC_K = \frac{\sum_{i=0}^{W-1} \gamma_i D_i}{\sum_{i=0}^W \gamma_i D_i} \quad (3)$$

Heterogeneous Workloads. When there are multiple kernels running concurrently, the model needs to keep track of the state of each workload. Although we only consider two concurrent kernels (\mathcal{K}_1 and \mathcal{K}_2) in scheduling, our model can be used to handle more than two kernels.

Suppose there are two kernels \mathcal{K}_1 and \mathcal{K}_2 , and each has w_1 and w_2 active warps respectively ($w_1 + w_2 =$

W). The number of possible states of the SM will be $(w_1 + 1) \times (w_2 + 1)$. The state space is represented as a value pair (p, q) with $0 \leq p \leq w_1$ and $0 \leq q \leq w_2$, where p and q are the numbers of idle warps of \mathcal{K}_1 and \mathcal{K}_2 respectively. We first consider individual workload state transition probabilities based the single kernel model. State transitions of the two kernels are independent with each other, since they are scheduled independently. Thus, the probability that the SM transits from state $(p_i, q_{i'})$ to state $(p_j, q_{j'})$ is the product of the individual transition probabilities.

The steady state vector of our two kernel Markov chain-based model is denoted as $\pi = \{\gamma_{(0,0)}, \gamma_{(0,1)}, \dots, \gamma_{(w_1, w_2)}\}$. With π , we calculate the IPC of each workload using the same method for homogeneous workloads, except the parameters are defined and calculated in the context of two kernels. For example, the round duration is equal to the total number of ready warps of both kernels. Individual IPC s of \mathcal{K}_1 and \mathcal{K}_2 is calculated as the ratio of non-idle cycles for each workload, as shown in Eq. (4) and (5), respectively. The concurrent IPC is the sum of individual IPC s (Eq. (6)).

$$IPC_{\mathcal{K}_1} = \frac{\sum_{i=0}^{w_1-1} \sum_{i'=0}^{w_2} \gamma_{(i,i')} \times (w_1 - i)}{\sum_{i=0}^{w_1} \sum_{i'=0}^{w_2} \gamma_{(i,i')} \times D_{(i,i')}} \quad (4)$$

$$IPC_{\mathcal{K}_2} = \frac{\sum_{i=0}^{w_1} \sum_{i'=0}^{w_2-1} \gamma_{(i,i')} \times (w_2 - i')}{\sum_{i=0}^{w_1} \sum_{i'=0}^{w_2} \gamma_{(i,i')} \times D_{(i,i')}} \quad (5)$$

$$C = IPC_{\mathcal{K}_1} + IPC_{\mathcal{K}_2} \quad (6)$$

With the estimated IPC , we now discuss how to estimate the optimal slice size ratio for two kernels. We define the slice ratio which minimizes the execution time difference of co-scheduled slices as the *balanced slice ratio*. By minimizing the execution time difference, the kernel-level parallelism is maximized. The execution time difference is calculated as ΔT in Eq. (7).

$$\Delta T = \left| \frac{1}{IPC_{\mathcal{K}_1}} \times I_{\mathcal{K}_1} \times P_{\mathcal{K}_1} - \frac{1}{IPC_{\mathcal{K}_2}} \times I_{\mathcal{K}_2} \times P_{\mathcal{K}_2} \right| \quad (7)$$

$I_{\mathcal{K}_i}$ and $P_{\mathcal{K}_i}$ represent the number of instructions per block and the slice size of kernel \mathcal{K}_i ($i = 1, 2$). Since $P_{\mathcal{K}_i}$ is less than the maximal number of active thread blocks, only a limited number of slice ratios need to be evaluated to get the balanced ratio.

Uncoalesced Access. So far, we assume that all memory accesses are coalesced and each memory instruction results in the same number of memory requests. However, due to the different address patterns, memory instructions may result in different amounts of memory requests. On Fermi GPUs, one memory instruction can generate 1 to 32 memory requests. Here we consider the two most common access patterns: fully coalesced access, and fully uncoalesced access. We extend our model to handle both coalesced and uncoalesced accesses by defining three states for a warp: ready, stalled on coalesced access (coalesced idle), and stalled on uncoalesced access (uncoalesced idle). The memory operation latency depends on the memory access type. Since uncoalesced access generates more memory traffic, its latency is higher than that of coalesced access. We also use the linear model to estimate the latency. By identifying the

ratio of coalesced and uncoalesced memory instructions, we can easily extend the two-state model to handle three states and their state transitions. Distinguishing between coalesced and uncoalesced accesses increases the accuracy of our model.

Adaptation to GPUs with multiple warp schedulers.

Our model assumes there is only one warp scheduler. New-generation GPUs can support more than one warp schedulers. The latest Kepler GPU features four warp schedulers per SMX (SMX is the Kepler terminology for SM). We extend our model to handle this case by deriving a single pipeline virtual SM based on the parameters of the SMX. The virtual SM has one warp scheduler, and its parameters such as active thread blocks and memory bandwidth are obtained by dividing the corresponding parameters of the SMX by the number of warp schedulers. This virtual SM can still capture the memory and computation features of a kernel running on the SMX. Experimental results in Appendix C of the supplementary file show that performance modeling on the virtual SM provides a good estimation on the Kepler architecture.

We develop an efficient implementation on reducing the model calculation cost and obtaining input parameters. The details about those issues can be found in Appendix B of the supplementary file.

In summary, our probabilistic model has captured the inherent non-determinism in concurrent kernel executions. First, it simply requires only a small set of profiling results on the memory and computation characteristics of individual kernels. Second, with careful probabilistic modeling, we develop a performance model that is sufficiently accurate to guide our scheduling decision. The effectiveness of our model will be evaluated in the experiments (Section 5).

5 EVALUATION

In this section, we present the experimental results for evaluating Kernelet on latest GPU architectures.

5.1 Experimental Setup

We have conducted experiments on a workstation equipped with one NVIDIA Tesla C2050 GPU, one NVIDIA GTX680 GPU, two Intel Xeon E5645 CPUs and 24GB RAM. We note that C2050 and GTX680 are based on Fermi and Kepler architectures, respectively. One C2050 SM has two warp schedulers, and each can serve half a warp per cycle (with a theoretical *IPC* of one). In contrast, one GTX680 SMX features four warp schedulers and each warp scheduler can serve one warp per cycle (with a theoretical *IPC* of eight considering its dual-issue capability). More details about those two architecture can be found in Appendix C of the supplementary file. Our implementation is based on GCC 4.6.2 and NVIDIA CUDA toolkit 5.0.

Workloads. We choose eight benchmark applications with different memory and computation intensiveness.

Sources of the benchmarks include the CUDA SDK, Parboil Benchmark [27], CUSP [2] and our home grown applications. The benchmark applications include: Pointer Chasing (PC), Sum of Absolute Differences (SAD), Sparse Matrix Vector Multiplication (SPMV), Matrix Multiplication (MM), Magnetic Resonance Imaging - Q (MRIQ), Black Scholes (BS), and Tiny Encryption Algorithm (TEA). Details about the benchmark applications can be found in Appendix C of the supplementary file.

To evaluate the two-level scheduler design in Kernelet, we define a *scale factor* (sf) for each application. We define scale factor equals one, when input data size of the application equals the default size. Given a scale factor of sf , we scale the input data size of the application to be sf times as that with the scale factor of one. We vary the scale factor to evaluate different cases for computation/memory characteristics. Particularly, we are interested in two cases. In *Case I*, all the scale factors equal one ($sf = 1$). The transfer of input data is done once at the beginning and repeatedly accessed by serials of kernel invocations. This case represents the applications where many kernels continuously access the input data that can fit into the GPU memory (e.g., GPU-based query processing in databases [10]). In this case, the data transfer overhead is ignorable (the input data transfer is amortized among many kernel executions and the output transfer is overlapped with the kernel execution). Thus, we can study the sole impact of kernel slicing and co-scheduling. In *Case II*, we evaluate the impact of different scale factors. Particularly, we evaluate the effectiveness of both kernel co-scheduling and the overlapping on PCI-e data transfer and the kernel execution. The GPU memory may not be large enough to accommodate all working sets and the PCI-e data transfers are more frequent than Case I.

To assess the impact of kernel scheduling under different mixes of kernels, we create four groups of kernels namely CI, MI, MIX and ALL (as shown in Table 2). CI represents the computation-intensive workloads, whereas MI represents workloads with intensive memory accesses. MIX and ALL include a mix of CI and MI kernels. ALL has all the eight kernels. In each workload, we assume the application arrival conforms to Poisson distribution. The parameter λ in the Poisson distribution affects the weight of the application in the workload. For simplicity, we assume that all application has the same λ . We also assume λ is sufficiently large so that at least two kernels are pending for execution at any time for a high utilization of the GPU. In the experiments, two instances of each kernel are submitted per second on average ($\lambda = 2$ instances per second). Thus, we fix the total number of instances of kernel executions for all scheduling algorithms and compare their total execution times (from submitting the first kernel till the completing the last kernel).

Comparisons. To evaluate the effectiveness of kernel scheduling in Kernelet, we have implemented the following scheduling techniques:

TABLE 2: Workload configurations.

Workload	Applications
CI	BS, MM, TEA, MRIQ
MI	PC, SPMV, ST, SAD
MIX	PC, BS, TEA, SAD
ALL	PC, SPMV, ST, BS, MM, TE, MRIQ, SAD

- **Kernel Consolidation (Base):** the kernel consolidation approach of concurrent kernel execution [22].
- **Oracle (OPT):** OPT uses the same scheduling algorithm as Kernelet, except that it pre-executes all possible slice ratios for all combinations to obtain the CP and then determines the best slice ratio and kernel combination. In another word, OPT is an offline algorithm and provides the best throughput for the greedy scheduling algorithm.
- **Monte Carlo-based co-schedule (MC):** We adopt the Monte Carlo method to generate the performance distribution of random co-scheduling plans in the solution space. In each Monte Carlo simulation, we randomly pick kernel pairs for co-scheduling with random slice ratios. We denote the result of MC to be $MC(s)$, where s is the number of Monte Carlo runs.

5.2 Results on Model Prediction

We evaluate the accuracy of our performance model in different aspects, including the estimation of IPC s for single kernel and concurrent kernel executions, and CP prediction for concurrent kernel executions. Due to space limitation, we present results on CP in this section, and more experiments can be found in Appendix C of the supplementary file.

For the eight benchmark applications, we run every possible combination of kernel pairs and measure the IPC for each combination. Figure 5 compares the measured and predicted IPC s with the suitable slice ratio given by our model. For all the figures comparing the predicted and measured IPC and CP in this paper and the supplementary file, we also show the two lines ($y = x \pm 0.2$ for C2050 and $y = x \pm 0.8$ for GTX680) to highlight the scope where difference between prediction and measurement is within $\pm 20\%$ of the peak IPC . Note, the theoretical IPC s for C2050 and GTX680 are one and eight respectively. If the result falls in this scope, we consider the estimation well captures the trend of the measurement. Figure 5 shows most of the predictions are with this scope. More results of model prediction can be found in Appendix C of the supplementary file. For different kernel combinations and slicing ratios, our model is able to well capture the trend of concurrent executions for both dynamic and static slice ratios.

5.3 Results on Kernel Scheduling

In this section, we evaluate the effectiveness of our kernel scheduling algorithm by comparing with Base and OPT. To simulate the continuous kernel submission

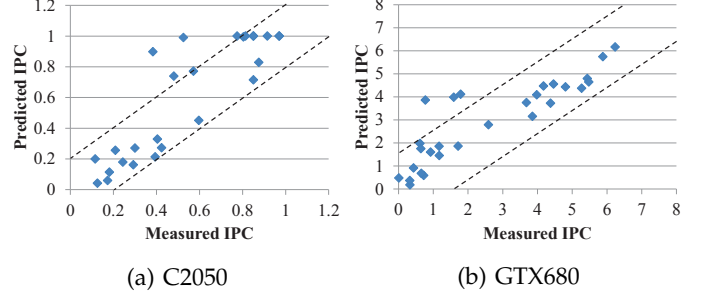


Fig. 5: Comparison between predicted and measured concurrent kernel execution IPC s on two GPUs with optimal slice ratio.

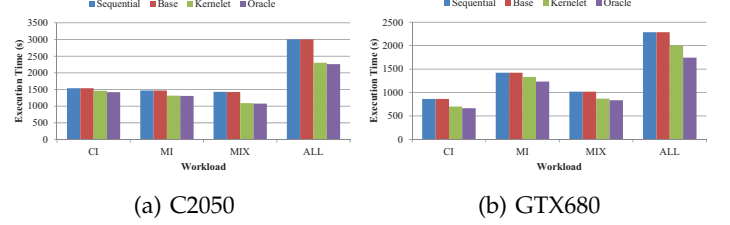


Fig. 6: Comparison between different scheduling methods on both C2050 and GTX680.

process, we initiate 1000 instances for each kernel and submit them for execution according to Poisson distributions. Different scheduling algorithms are applied and the total kernel execution time is reported.

Results for Case I. We first study Case I, where PCI-e data transfer overhead is small. Figure 6 shows the total execution time of executing those kernels on C2050 and GTX680. We also include the performance of sequential execution for reference. On all the four workloads with different memory and computation characteristics, Kernelet outperforms Base (with the improvement 5–31% for C2050 and 7–23% for GTX680). Kernelet achieves similar performance to OPT (with the difference 1–3% for C2050 and 4–15% for GTX680). The performance improvement of Kernelet over Base is more significant on MIX and ALL, because Kernelet have more chances to select kernel pairs with complementary resource usage. Base only slightly improves the sequential execution since each individual kernel has sufficient thread blocks to entirely occupy the GPU. We also show the number of kernel combinations pruned during kernel scheduling with varying pruning parameters in Appendix C of the supplementary file.

We finally study the execution time distribution of the scheduling candidate space. Figure 7 shows the CDF (cumulative distribution function) of the execution time of the $MC(1000)$ (1000 Monte Carlo simulations) on ALL workload. None of the random schedules is better than Kernelet. It demonstrates that random co-schedules hurt the performance with a high probability due to the huge space of scheduling plans.

Results for Case II. In Case II, we can evaluate the impact of memory transfer/computation overlapping and kernel co-scheduling. Figure 8 shows the total ex-

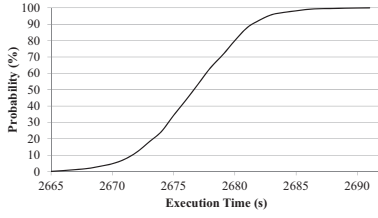


Fig. 7: CDF (cumulative distribution function) of execution time of MC(1000).

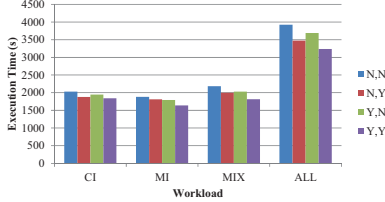


Fig. 8: Execution time comparison of different Kernelet variants on C2050 ($sf = 1$).

ecution time including all memory transfers and kernel executions comparison with different algorithm variants " O_1, O_2 " for $sf = 1$ on C2050. We observed similar results on GTX680. While the total input data size of all kernels is roughly equal to the GPU memory size, each kernel execution accesses different input data. Thus, PCI-e data transfer is necessary for each kernel execution, unlike Case I. Considering many kernel executions, the total working set size of all kernel executions is much larger than the GPU memory size. In our notation, $O_1 = Y, N$ represents the algorithm with and without memory/computation overlapping respectively, and $O_2 = Y, N$ represents the algorithm with kernel co-scheduling in Kernelet and the default execution method, respectively. Thus, $Y - Y$ represents our fully optimized Kernelet system. Overall, Kernelet is up to 9.2%, 13.0%, 16.8%, and 17.5% faster than the other three variants on CI, MI, MIX and ALL, respectively. It is the combined improvement of kernel scheduling and memory transfer/computation overlapping.

Let us analyze the impact of individual techniques in more details. Comparing Y, Y and N, Y , we find that the memory transfer/computation overlapping improves the performance by 9%, thanks to the automated memory management techniques. We divide the data transfer time on PCI-e into two categories, depending on whether the memory transfer is overlapped with GPU kernel execution. In Kernelet, the ratios of the memory transfer that overlaps with GPU kernel execution are

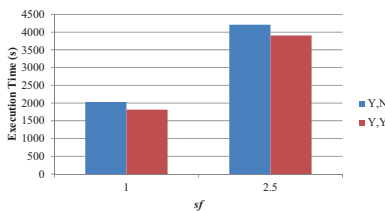


Fig. 9: Impact of kernel co-scheduling for different sf on C2050.

99%, 95%, 97%, and 97% on CI, MI, MIX and ALL for Y, Y , respectively. Most of PCI-e data transfer overhead is hidden by the kernel execution.

Comparing Y, Y and Y, N , we find that kernel co-scheduling achieves a smaller performance improvement compared with those in Case I. The improvement of kernel co-scheduling is 5–12% on the four workloads. Figure 9 shows the performance for different algorithm variants for MIX when $sf = 1$ and $sf = 2.5$. When $sf = 2.5$, the total input data size of all kernels exceeds the GPU memory size. Kernelet consistently improves the performance for varying scale factors. We also note that, when $sf = 2.5$, 95% of the PCI-e data transfer is overlapped with kernel execution in MIX for Y, Y .

6 RELATED WORK

In this section, we discuss the related work in multi-kernel executions on the GPUs. We also review the scheduling on multi-core CPUs in Appendix D of the supplementary file.

GPU architectures have undergone significant and rapid improvements for GPGPU support. Due to lack of concurrent kernel support in early GPU architectures, researchers initially proposed to merge two kernels at the source code level [7]. They have three major disadvantages compared with our approach. First, they will increase the resource usage of each thread block, leading to lower SM occupancy and performance degradation. Second, those approaches require source code, which is usually unavailable in shared environments. Third, it requires two kernels with different block sizes avoiding using barriers within the thread block, otherwise deadlocks may occur.

New-generation GPUs like NVIDIA Fermi GPUs support concurrent kernel executions. Taking advantage of this new capability, a number of multi-kernel optimization techniques [8], [16], [23] have been developed to improve the utilization of GPUs. Ravi et al. [22] proposed kernel consolidation to enable space sharing (different kernels run on different SMs) and time sharing (multiple kernels reside on the same SM) on GPUs. However, kernel consolidation does not have space sharing and has little time sharing when the launched kernels have sufficient thread blocks to fully occupy the GPU. Furthermore, they determined the kernel to be consolidated with heuristics based on the number of thread blocks. In contrast, Kernelet utilizes slicing to create more opportunities for time sharing, and develops a performance model to guide the scheduling decision. Wende et al. [29] exploited space sharing for concurrent execution of kernels with a small number of thread blocks. Peters et al. [21] used a persistently running kernel to handle requests from multiple applications. GPU virtualization has also been investigated [8], [16]. Pai et al. [19] proposed *elastic Kernels* to allow concurrent execution of thread blocks from different kernels. However, their scheduling policies have not considered

the co-scheduling opportunities from different memory/computation characteristics.

Recent studies also address the problem of GPU scheduling when multiple users share one machine, e.g., RGEM [15] and PTask [23] manage the GPU at the operating system level. All those scheduling methods do not consider how to schedule concurrent kernels in order to fully utilize the GPU resources. Automatic memory and data transfer management has been studied [13]. Most studies rely on advanced programming analysis techniques to automate and optimize the data transfer between the GPU memory and the main memory. Those techniques are orthogonal to Kernelet.

As for performance models on GPUs, Hong [12] and Sim [24] proposed analytical models based on the round-robin warp scheduling assumption. All those models are designed for a single kernel. Moreover, they usually require extensive hardware profiling and/or simulation processes. In contrast, our performance model is designed for concurrent kernel executions on the GPU.

7 CONCLUSION

Recently, GPUs have been more and more widely used in clusters and cloud environments, where many kernels are submitted and executed on the shared GPUs. This paper proposes Kernelet to improve the throughput of concurrent kernel executions. Kernelet implements transparent memory management and data transfer techniques on the GPU, creates more sharing opportunities with kernel slicing, and uses a probabilistic performance model to capture the non-deterministic performance features of multiple-kernel executions. We evaluate Kernelet on two NVIDIA GPUs, Tesla C2050 and GTX680, with Fermi and Kepler architectures respectively. Our experiments demonstrate the accuracy of our performance model, and the effectiveness of Kernelet by improving the concurrent kernel executions by 5–31% and 7–23% on C2050 and GTX680, respectively. Our on-going work includes implementing Kernelet into our graph processing system Medusa [31], [32].

8 ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable comments. This work is supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067).

REFERENCES

- [1] S. S. Bagsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu. Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors. In *PPoPP*, 2012.
- [2] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations version 0.3.0. <http://cusplibrary.github.io/>, accessed on May 16th, 2013.
- [3] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana-Orti. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPCS*, 2010.
- [4] W. Fang, B. He, Q. Luo, and N. K. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *IEEE Trans. Parallel Distrib. Syst.*, 22(4):608–620, 2011.
- [5] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53(11), Nov. 2010.
- [6] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC*, 2008.
- [7] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *HotPar*, 2012.
- [8] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. Gvim: GPU-accelerated virtual machines. In *HPCVirt*, 2009.
- [9] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *PACT*, pages 260–269.
- [10] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):1–39, 2009.
- [11] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.
- [12] S. Hong and H. Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *ISCA*, 2009.
- [13] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.
- [14] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*, 2008.
- [15] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *RTSS*, 2011.
- [16] T. Li, V. Narayana, E. El-Araby, and T. El-Ghazawi. GPU resource sharing and virtualization on high performance computing systems. In *ICPP*, 2011.
- [17] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra. Optimizing symmetric dense matrix-vector multiplication on GPUs. In *SC*, 2011.
- [18] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-D FFT kernel for GPUs using CUDA. In *SC*, 2008.
- [19] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving GPGPU concurrency with elastic kernels. In *ASPLOS*, 2013.
- [20] A. Papoulis and R. Probability. *Stochastic processes*, volume 3. McGraw-hill New York, 1991.
- [21] H. Peters, M. Koper, and N. Luttenberger. Efficiently using a CUDA-enabled GPU as shared resource. In *CIT*, 2010.
- [22] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *HPDC*, 2011.
- [23] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP*, 2011.
- [24] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in GPGPU applications. In *PPoPP*, 2012.
- [25] A. Snively and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.
- [26] W. Stallings. *Operating Systems: Internals and Design Principles*, 6/E. Pearson Education India, 2009.
- [27] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. IMPACT technical report, IMPACT-12-01, UIUC, 2012.
- [28] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC*, 2008.
- [29] F. Wende, F. Cordes, and T. Steinke. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. In *SAAHPC*, 2012.
- [30] Y. Zhang and J. D. Owens. A quantitative performance analysis model for gpu architectures. In *HPCA*, 2011.
- [31] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE TPDS*, 99(Preliminary):1, 2013.
- [32] J. Zhong and B. He. Parallel graph processing on graphics processors made easy. *PVLDB (demo)*, 6:1–4, 2013.
- [33] Zillians. V-GPU: GPU virtualization. <http://www.zillians.com/products/vgpu-gpu-virtualization/>, accessed on May 16th, 2013.



Jianlong Zhong received the bachelor degree in software engineering from Tianjin University (2006-2010) and is now a PhD candidate in School of Computer Engineering of Nanyang Technological University, Singapore. His research interests include GPU computing, cloud computing and parallel algorithms. His recent work mainly focuses on parallel graph processing on emerging hardware architectures such as the GPU.



Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in Division of Networks and Distributed Systems, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, cloud computing, and database systems.