Slate: Enabling Workload-Aware Efficient Multiprocessing for Modern GPGPUs

Conference Paper · May 2019

DOI: 10.1109/IPDPS.2019.00035

3 authors:

Tyler Allen
Clemson University
9 PUBLICATIONS **52** CITATIONS
SEE PROFILE

Xizhou Feng
Clemson University
37 PUBLICATIONS **2,124** CITATIONS
SEE PROFILE

Rong none Ge
Clemson University
58 PUBLICATIONS **1,894** CITATIONS
SEE PROFILE

2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

Slate: Enabling Workload-Aware Efficient Multiprocessing for Modern GPGPUs

Tyler Allen, Xizhou Feng, Rong Ge
Clemson University
{tnallen, xizhouf, rge}@clemson.edu

***Abstract*—As GPUs now contribute the majority of computing power for HPC and data centers, improving GPU utilization becomes an important research problem. Sharing GPU among multiple kernels is an effective approach but requires judicious kernel selection and scheduling for optimal gains. In this paper, we present Slate, a software-based workload-aware GPU multiprocessing framework that enables concurrent kernels from different processes to share GPU devices. Slate selects concurrent kernels that have complementary resource demands at run time to minimize interference for individual kernels and improve GPU resource utilization. Slate adjusts the size of application kernels on-the-fly so that kernels readily share, release, and claim resources based on GPU status. It further controls overhead including data transfers and synchronization. We have built a prototype of Slate and evaluated it on a system with a NVIDIA Titan Xp card. Our experiments show that Slate improves system throughput by 11% on average and up to 35% at the best scenario for the tested applications, in comparison to NVIDIA Multi-Process Service (MPS) that uses hardware scheduling and the leftover policy for resource sharing.**

***Index Terms*—GPGPU, GPU Multiprocessing, GPU resource sharing, concurrent kernels, kernel scheduling.**

I. Introduction

Optimization of GPU utilization for application performance and system throughput has become a critical open research problem, as GPUs are a major computational resource for today's HPC and data centers. According to the current TOP500 list, five of the seven top systems are accelerated with NVIDIA GPUs [12] and Summit, the top system, draws 95% of its computing power from GPU devices [3]. Nevertheless, GPU computing capabilities are not efficiently utilized. The top five GPU-accelerated systems achieve 65% of the designed peak performance on average when running the highly optimized GPU-friendly LINPACK Benchmark. GPU utilization for real-world applications is much lower. For example, when running the HPCG benchmark, Summit only achieves 1.5% of its peak performance [5].

One reason for the low GPU utilization is that many kernels cannot fully utilize the memory and compute resources on their own all the time [16]. Kernels have various memory access and compute profiles, and have different sizes. Memory intensive applications that are bounded by device memory bandwidth cannot fully utilize the computing capacity. Small kernels do not have sufficient degrees of parallelism to occupy all the processing units. Meanwhile, some compute intensive kernels may use only a portion of the devices' memory bandwidth. Low resource utilization is common when kernels optimized for earlier generations of architectures run on cutting-edge devices.

Sharing GPU resources among multiple kernels with complementary demands can improve GPU utilization. For example, pairing a memory intensive application with a compute intensive application could fully utilize both GPU memory bandwidth and processing units. There are two sharing mechanisms: time slicing—switching among multiple kernels over time, and spatial sharing—executing more than one kernels at a time. A practical GPU sharing framework can use either mechanism or combine them.

To enable and improve GPU resource sharing, recent NVIDIA GPU architectures support two features: Hyper-Q and MPS (Multi-Process Service). Hyper-Q uses multiple hardware work queues to build simultaneous, hardware-managed connections between the host and the GPU device to support concurrent kernel launching on a single GPU [1]. Hyper-Q requires all the work queues belong to a single CUDA context in the same process. To bypass this hardware limitation, MPS introduces a client-server architecture to map multiple clients' CUDA contexts onto a single server CUDA context and then leverages Hyper-Q to build hardware work queues for the clients [15]. MPS uses time-slicing on early architectures and has added spatial sharing on more recent architectures including Pascal and Volta [14].

While MPS extends GPU resource sharing to multiple processes, it neither assesses if the kernels interfere with each other nor evaluates if they can optimize the system throughput. As a result, concurrent kernels may compete for GPU resources like memory bandwidth and cache space, and such contention adversely hurts the performance of individual kernels and degrades system throughput. A workload-aware alternative is needed, instead, to maximize the system throughput and avoid or minimize contention.

In this paper, we present *Slate*, a workload-aware GPU multiprocessing framework that enables concurrent kernels from different processes on modern GPU devices. *Slate* introduces several novel ideas to improve GPU utilization through efficient resource sharing.

First, *Slate* provides a software-based solution that enables spatial sharing of GPU resources among different host processes. It offers a high performance, open-source alternative to NVIDIA MPS to improve GPU utilization and further GPU multiprocessing research.

Second, *Slate* integrates workload-awareness into GPU mul-

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00035

tiprocessing. It selects concurrent kernels with complementary resource demands to minimize resource contention, and uses several techniques to facilitate kernels to share, claim and release resources at run time.

Third, *Slate* scheduling is cost-effective. Scheduling overhead is a major reason why software-based GPU multiprocessing attempts fail to deliver desirable performance on real systems [16]. *Slate* controls the overhead by minimizing schedule-related computation, data transfer, and synchronization.

Finally, we have implemented a *Slate* prototype, which comprises *Slate* runtime, *Slate* library, and APIs, and evaluated it on real GPU platforms. The results on a NVIDIA Titan Xp based system show that *Slate* outperforms MPS by 11% for the set of applications under study.

II. The GPU Resource Sharing Problem

According to the NVIDIA's GV100 white paper, allowing multiple clients to share the same GPU device expedites workload execution by *seven* times [14]. We can interpret this statement in two ways. First, GPU underutilization is serious and common. Second, GPU resource sharing is effective and must be promoted on production systems.

The GPU underutilization can be reproduced using the stream application on our system. Figure 1 illustrates how global memory read performance changes with the number of GPU SMs. Bandwidth first increases quickly and reaches the peak with nine SMs; it does not further increase with SMs. If we pair this kernel with a compute intensive kernel and divide the SMs among them, we can fully utilize both memory bandwidth and computing capacities without noticeably slowing down the execution of each kernel.

However, GPU resource sharing is non-trivial, and we face several challenges to realize it on today's architectures.

Kernel characterization and selection. Concurrent kernels share multiple hardware resources including processing elements, memory bandwidth, and caches. Arbitrary kernel pairing creates contention on these resources, which not only interferes with the execution of individual kernels but also degrades system throughput. Fully utilizing GPU resources requires us to judiciously select concurrent kernels with complementary workload characteristics. In this work, we explore methods of workload characterization and selection, and choose those that balance between accuracy and simplicity for runtime employment.

Scheduling overhead. The scheduler for GPU resource sharing incurs time to select concurrent kernels, allocate GPU resources, and schedule their executions. Any considerable cost will be pronounced because many kernels have a short turn-around time. In this work, we use multiple techniques to control the scheduling overhead and ensure an overall gain from GPU resource sharing.

System support. Current GPU devices use hardware work queues and hardware kernel scheduling without providing flexible controls to external software. The very few vendor-provided tools like NVIDIA MPS [15] are kept as proprietary

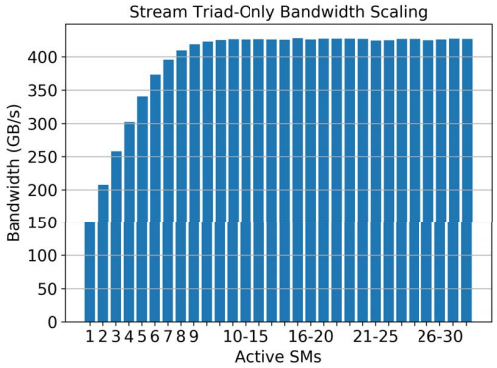

Fig. 1: The performance variation with the number of SMs for the Stream benchmark with a fixed problem size of 6GB on a Titan Xp device. Limited by memory capability, performance reaches the peak at 9 SMs and then flattens.

code. Consequently, we have to build our own framework *Slate*, which utilizes the underlying hardware scheduling.

Although both *Slate* and NVIDIA MPS support GPU multiprocessing, *Slate* distinguishes from MPS by proactively sharing resources among kernels and embracing the concept of workload-aware kernel scheduling.

III. Workload-Aware Kernel Scheduling

Slate provides a software-based solution for GPU multiprocessing and resource sharing. The *Slate* overview is provided in Figure 2. *Slate* accepts kernels from multiple processes, builds a queue for each process and CUDA stream, and schedules them to disjoint SMs.

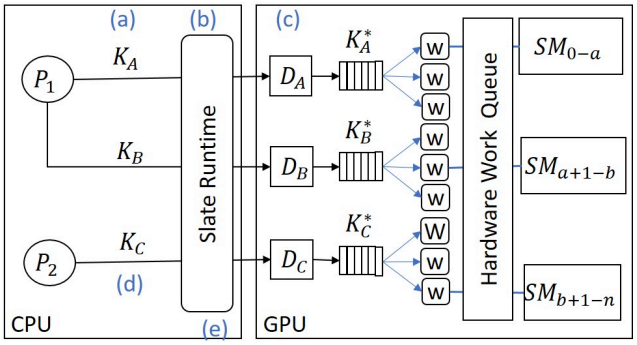

Fig. 2: The *Slate* methodology overview. We highlight several actions in the method as follows: (a) CPU processes launch compute kernels through *Slate* Runtime. (b) *Slate* Runtime (i) provides context funneling to enable concurrent execution of kernels from different CPU processes and (ii) applies kernel transformation to optimize kernel performance and resource utilization. (c) Dispatcher dispatches the transformed kernel, which creates a task queue and binds worker threads to a set of SMs; worker threads retrieve tasks from the task queue. (d) *Slate* Runtime selects complementary kernels to share resource and improve system throughput. (e) *Slate* monitors the system state, notifies the dispatch kernels to dynamically adjusts the kernel sizes.

In addition to supporting concurrent kernels from multiple GPU applications, *Slate* integrates several important techniques to improve the GPU utilization and system perfor-

mance. In this section, we discuss three workload-aware kernel scheduling techniques: kernel transformation, complementary kernel selection, and dynamic kernel resizing.

A. Kernel Transformation

GPU computations are centered on kernels—functions compiled for GPU accelerators and offloaded from the host processes to be executed by many GPU threads. Unlike CPU threads which users have flexible controls, GPU threads are managed by the GPU hardware. Given a kernel's thread blocks, the GPU hardware uses block-oriented scheduling to dispatch the thread blocks to *all* the SMs, preventing resources sharing among multiple precesses.

To overcome the limitation of GPU hardware scheduling, *Slate* transforms user kernels into a format that facilitates efficient kernel task queuing. Specifically, it takes the user kernels and replaces the built-in CUDA variables such as blockIdx and gridDim, and injects necessary device codes to create an additional software layer to manage the order and patterns how kernels are dispatched and executed on GPU devices. The kernel transformation must preserve the semantics of user kernels for computation and be light weight.

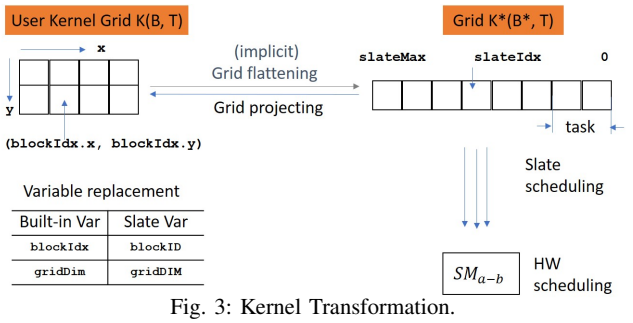

Fig. 3: Kernel Transformation.

1) Kernel Transformation Details: Figure 3 shows the kernel transformation and *Slate* scheduling. *Slate* converts a 2D or 1D grid user kernel $K(B,T)$ to a 1D grid $K^*(B^*,T)$ without modifying the internal structure of the thread blocks. Here B is a 2D or 1D grid and B^* is 1D, and T, the threads per block, is the same in both kernels.

Slate puts the 1D thread blocks of K^* in a queue, and creates a set of persistent workers, each repeatedly pulling the next group of blocks and executing them. To keep track of the queue status, *Slate* uses a scheduling index $slateIdx$, which begins at 0 and finishes at $slateMax$. Unlike GPU hardware that uses a thread block as a scheduling unit, *Slate* groups multiple user thread blocks into a task and schedules the tasks.

K and K^* are isomorphic. Corresponding blocks are identified differently: K uses the built-in variables $gridDim$ and $blockIdx$, and K^* introduces a variable $globIdx$. With the *Slate* scheduling, the values of $gridDim$ and $blockIdx$ no longer correspond to the user-specified grid dimensions. To maintain the user kernel semantics, *Slate* replaces these built-in variables in the user kernel K with its own corresponding variables as shown in Figure 3.

2) Benefits of Kernel Transformation: The kernel transformation process provides several benefits.

First, it enables efficient software-based kernel scheduling, where workers pull the tasks from a queue and execute them.

Second, because the workers process the tasks from the queue in order, they preserve data locality and increase the performance of typical applications.

Third, *Slate* can specify the size of workers and bind them to a designated range of SMs $[\texttt{sm\_low}, \texttt{sm\_high}]$. With this ability, *Slate* can partition the SMs into disjoint subsets and enable spatial sharing among multiple kernels. The detailed implementation of worker-SM binding will be discussed in the following section.

3) Kernel Transformation Overhead: Kernel transformation in previous studies incurs a large cost and is unsuitable for runtime application [16]. In *Slate*, we use two techniques to control the cost of kernel transformation so that it is negligible.

First, *Slate* avoids expensive modulo and division operations when it converts grids, and keeps the inner block geometry unchanged.

Second, *Slate* maintains the intra-block structure, and thus leverages the data locality designed by the kernel developer.

B. Concurrent Kernel Selection

Slate includes a concurrent kernel selection component to determine if an active kernel should share device resources with another kernel. *Slate* makes the decision based on profiles of the kernels and available GPU resources. It chooses concurrent kernels if there exists a candidate kernel with complementary workload characteristics. Otherwise, it runs the active kernel on the entire SMs.

In this study, we say that two kernels are complementary if their concurrent execution has a higher system throughput than their consecutive executions. We use average normalized turnaround time (ANTT) to evaluate throughput. Assume that kernels J_k and J_{k+1} take T_k and T_{k+1} to complete using all the SMs respectively, and T'_k and T'_{k+1} when sharing resource. ANTT is $T = (T_k + T_{k+1})$ for the consecutive solo runs, which is the default situation with CUDA. ANTT is $T' = \max(T'_k, T'_{k+1})$ for the concurrent case if *Slate* or MPS is enabled. $T' < T$ indicates better throughput from concurrent kernels.

1) The Selection Algorithm: Without loss of generality, assume that kernel J_{k-1} has just completed and released its resource, and kernel J_k is active, as illustrated in Figure 4. *Slate* examines if the next kernel J_{k+1} is complementary to J_k. If yes, *Slate* chooses corun (a). Otherwise, *Slate* examines other kernels in the queue. *Slate* runs J_k solo (b) if no complementary kernel is found.

Slate uses kernel profiles to infer at run time if two kernels are complementary. Specifically, it records kernel profiles obtained from its previous runs or offline profiling. To control the decision cost, *Slate* focuses on the most impacting resources, i.e., L2 cache, global memory and compute resource, and uses a heuristic decision method based on demands for them.

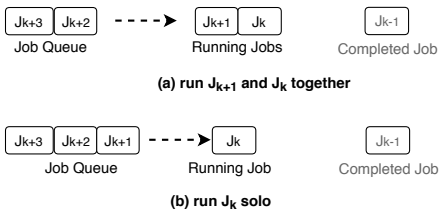

Fig. 4: The *Slate* scheduling decisions: (a) concurrent kernels J_k and J_{k+1} or (b) single kernel J_k.

2) Heuristic Policies: At run time, *Slate* refers to a heuristic policy table in Table I to decide whether a given pair of kernel should share a GPU. This table is derived from empirical results.

Generally, this heuristic method categorizes applications in two characteristics: compute intensity (C) and memory intensity (M). It labels the intensity with three levels—low (L), medium (M), and high (H). *Slate* gives a higher priority to memory intensity over computation intensity. For example, an application of H_M is simply memory intensive, while an application of low-memory (L_M) could be L_C or M_C or H_C. For a given pair of kernels, the heuristic method examines the performances of corun and solo runs, and chooses the one that delivers the higher throughput.

TABLE I: The *Slate* heuristic scheduling policy. The policy is "corun" if the kernels are complementary, and "solo" otherwise.

Benchmark		L_C	M_C	H_C	M_M	H_M
L_M	L_C	corun	corun	solo	corun	corun
	M_C	corun	corun	solo	solo	corun
	H_C	solo	solo	solo	solo	corun
M_M		corun	solo	corun	solo	solo
H_M		corun	corun	solo	solo	solo

C. Dynamic Kernel Resizing

When a concurrent kernel completes or a new kernel arrives, the running kernel should grow or shrink instantly to claim or release resource. Without delay, a completed kernel should release its designated SMs, which the running kernel seamlessly claims. Similarly, if a new kernel arrives, the running kernel shrinks and releases a part of its SMs. Such online kernel resizing is necessary for true GPU multiprocessing and optimized resource utilization.

To realize online kernel resizing, *Slate* dynamically adjusts the spatial SM partition and the worker-SM binding. As introduced earlier, *Slate* transforms the kernels so that it can use a set of persistent workers to execute each user kernel and bind them to a designated range of SMs. *Slate always* sets the size of workers as the maximum number of thread blocks that the designated SMs can support. When the running kernel needs to grow or shrink, *Slate* increases or decreases its designated range of SMs. This involves actions: terminate the old workers and launch a new set of workers. To carry over the kernel progress, *Slate* uses variable $slateIdx$ to keep track of the status of the user task queue. The new set of workers begins with tasks referred by $slateIdx$.

IV. System Design and Implementation

We set several goals in the design of *Slate*. First, *Slate* enables multiprocessing and resource sharing for various GPU applications. Second, *Slate* provides a GPU computing environment where users can write GPU programs and run them as usual. To support the described user experience, *Slate* provides automatic kernel transformation, kernel profiling, concurrent kernel selection, and resource sharing. Ultimately, *Slate* improves system throughput without slowing down individual application execution.

A. System Architecture

Slate is designed with a client-server structure. On the client side is a set of user APIs along with associated library. On the server side is a system runtime (daemon) running on the host that responds to requests from clients and manages workload-aware kernel scheduling. With this architecture, *Slate* can funnel the contexts of kernels from different CPU processes to a single CUDA context, necessary to enable GPU resource sharing among different applications.

1) Slate API: The *Slate* API acts as a wrapper for basic CUDA functions, which the daemon intercepts to funnel contexts and performs kernel transformation and scheduling.

To maintain the semantics of the host-device communications, for each *Slate* API function, the daemon performs some additional operations on top of the corresponding CUDA function. For example, in the case of memory allocation, the daemon allocates a shared buffer, passes its address back to the client, invokes the corresponding CUDA memory allocation call and obtains the returned GPU pointer, and records in a hash table the mapping between the shared buffer address and the GPU pointer. In the case of data transfer between the host and the device, the daemon replaces the shared buffer addresses that clients provide with the corresponding GPU pointers, and then invokes the basic CUDA data transfer. The kernel launch and synchronization cases are simple for which the daemon basically performs the CUDA functions.

In comparison to CUDA, *Slate*'s client-server architecture introduces an additional daemon and extra communications. To control the cost, *Slate* uses multiple channels and adopts a type-based communication strategy. It uses buffers shared between the client and the daemon to store and transfer kernel IO data, which can range from bytes to gigabytes in size [10], [28]. This channel avoids extra memory footprint and data copy, a better option for large data volumes. Another channel is named pipe, which *Slate* uses to communicate API instructions and commands with fast responses.

The *Slate* API is presently provided as a C++ header and shared linkable library for user kernels. We will investigate how *Slate* works for CUDA libraries with diverse compute and memory intensities. In the least, we expect *Slate* can recognize the heavily optimized implementations and run them solo. While we have not investigated how *Slate* can leverage DMA, we anticipate that it does not interfere because *Slate* uses the same host-device data transfer mechanisms as CUDA.

2) Slate Daemon: As the server, the daemon acts as a proxy for CUDA operations so that it funnels them under the same context. This is necessary to enable dynamic co-running that NVIDIA MPS also uses. One limitation is that the server intercepts client requests and relays them, introducing cost. As discussed earlier, *Slate* controls this cost using multiple communication channels. To respond quickly to clients, the daemon creates a session for each application process upon its first *Slate* API call, and keeps the session alive until the process completes. Each session is managed by a separate CPU thread.

More importantly, the daemon realizes workload-aware resource sharing and kernel scheduling. We discuss those functionalities in the following subsections and use daemon and runtime interchangeably.

B. Slate Runtime Internals

The runtime schedules user kernels automatically and transparently using several main modules including code injector, kernel profiler, and kernel scheduler.

The daemon *profiles* kernels at their first time run, and saves the profile data in the kernel profile table. The daemon references the profile data online to decide if it should run the kernels solo or concurrently with others.

At runtime, upon the receipt of a kernel launch from user programs, the daemon uses the code *injector* module to inject code segments, necessary to materialize kernel-SM mapping and kernel scheduling, into the user kernel code. The daemon then puts the transformed kernels in a queue where they wait for the to be dispatched by the *scheduler* to the GPU. Triggered by arrival and completion of kernels, the scheduler refers to the workload selection method to decide launching a new concurrent kernel or resizing the current running kernel.

Slate presently inserts code segments through runtime compilation to provide user transparency. Specifically, provided with a user device code from the *Slate* kernel launch API command, the runtime first uses a FLEX scanner to detect kernels in the original CUDA code and inserts the *Slate* scheduling and resource mapping code. It then uses the NVIDIA Runtime Compiler (NVRTC) to load the kernel onto the device [2]. A compiled kernel image can be further cached for later use by the same user. Alternatively, *Slate* can perform code injection statically using an OMP-like pragma method, which is less transparent.

C. Detailed Implementation of the Slate Runtime

With code injection, *Slate* maps kernels to disjoint SMs and creates separate consistent workers to iteratively execute the kernels. To enable on-the-fly SM partition and kernel corunning, *Slate* keeps track of SM availability and kernel status. Once more SMs become available, it schedules new kernels or resizes running kernels to run on them.

Kernel-SM Mapping. *Slate* maps a user kernel to a range of SMs and ensures that the kernel only runs on these SMs. This kernel-SM mapping is materialized with code injection, as shown in an example in Listing 1. In this example, *Slate* uses two additional kernel arguments: `sm_low` and `sm_high` to specify the lower and upper bounds of the designated SMs respectively. Following the injected variable declarations, *Slate* injects a code segment, which identifies the thread block leader, and assigns it to check if the provisioned SM falls into the designated range. If not, the thread block returns. Otherwise, it stays live to execute the kernel tasks and persists until it finishes them or receives a signal to terminate.

Upon the completion of this code segment, the number of live thread blocks is exactly the maximum number of blocks that the designated SMs can simultaneously support in-registers.

```
//sm_{low,high} are max/min sm id
__global__ void example(const uint sm_low,
                        const uint sm_high, ...) {
    __shared__ uint id, valid_task;
    uint slate_smid;
    // block thread leader
    const int leader = (threadIdx.x == 0 &&
                        threadIdx.y == 0 &&
                        threadIdx.z == 0);
    if (leader) {
        // block id initialization
        id = 0;
        // get SM id
        slate_smid = get_smid();
        // if SM id is in valid range
        valid_task = !(slate_smid < sm_low ||
                       slate_smid > sm_high);
    }
    __syncthreads();
    // entire block quits if invalid SM
    if (!valid_task) {return;}
    //snip
```

Listing 1: **Kernel header modification and code segment** inserted at the beginning of the original user source. It adds arguments to indicate the range of designated SMs. The first thread in the block verifies that the running SMs fall in the range. If not, the entire block quits.

Kernel Scheduling. Provided with the kernel-SM mapping, *Slate* transforms the kernel grid to 1-D, groups the blocks as tasks, and has the live worker threads repeatedly pull tasks until all tasks are complete. This scheduling is implemented with the injected code presented in Listing 2.

By using the 1-D grid, *Slate* places kernel tasks in a queue so that the live threads can execute them iteratively. The iterations are shown as the outer `do-while` loop in the code segment. *Slate* uses the variable `id` to keep track of the queue status. It breaks from the loop if `id` reaches the end of the queue or there is a retreat signal, which is triggered by the arrival or completion of another kernel.

In each iteration, a worker block pulls from the queue a task, which consists of a number ($\texttt{SLATE\_ITERS} \geq 1$) of user defined blocks. This grouping reduces the number of atomic operations on the queue. It also exploits *sequential-task optimization* for regular applications and ensures that multiple user blocks are executed in-order to improve locality. For each user defined block, *Slate* marks it with an 1-D global index, and uses this index to calculate the 1-D or 2-D index that corresponds to the user kernel semantics. This calculated index is used to replace the built-in variable $blockIdx$ in the user kernel source. The other built-in variable that *Slate* replaces is `gridDim`. As *Slate* uses the same inner block geometry as

```
⓪  __shared__ uint3 shared_blockID;
   __shared__ int iters;
   uint globIdx;
①  do {
       if (leader) {
           // pull task atomically
②          globIdx = atomicAdd(&slateIdx, SLATE_ITERS);
           // clamp iterations if last block
           iters = min(SLATE_ITERS, slateMax - globIdx);
           id = globIdx + SLATE_MAX;
           shared_blockID.x= globIdx % gridDimm.x - 1;
           shared_blockID.y = globIdx / gridDimm.x;
       }
       __syncthreads();
       uint3 blockID = {shared_blockID.x,
                        shared_blockID.y, 1};
       const register int local_iters = iters;
       // loop over the acquired block ids in task
③      for (int slate_count = 0; slate_count < local_iters;
            ++slate_count) {
④          ++blockID.x;
           if (blockID.x == gridDim.x) {
               // roll over to next Y index
               blockID.x = 0;
               ++blockID.y;
           }
⑤    // ORIGINAL USER CODE, built-in variables replaced
       }
       // while no signal and available tasks
   } while(!retreat && id < slateMax);
```

Listing 2: ***Slate* Scheduling Code** injected to kernel sources for 2D grids. Workers iteratively execute tasks ①. In each iteration, they pull a group of blocks ② and execute the blocks in-order ③. After computing the corresponding user defined block index ④, *Slate* replaces with it the built-in variables in the original user code and execute the user kernel ⑤.

the user kernel, it is lighter-weight than [16], which requires extra high-cost calculations to obtain the thread indices.

Dynamic Kernel Resizing. To optimize resource utilization, *Slate* dynamically adjusts the spatial SM partition and kernel-SM binding when new concurrent kernels arrive and running kernels complete. In the case of kernel arrival, *Slate* releases a part of the designated SMs of the current running kernel and reduces its SM range. In the case of kernel completion, *Slate* assigns the newly available SMs to the current running kernel and increase its SM range. This dynamic spatial sharing requires that kernels are able to grow and shrink online.

Slate dynamically resizes kernels using a special *dispatch kernel*, which packages together the user kernel and its designated SM range. To launch a user kernel, *Slate* instead launches a dispatch kernel, which in turn launches the main user kernel to its designated SMs and persists through its entire execution. Should the kernel-SM binding be adjusted before the user kernel tasks complete, the dispatch kernel terminates the previous launched user kernel and re-launches it to the adjusted SM range.

The dispatch kernel keeps track of user kernel progress and carries it over in re-launches. This is done through variable `slateIdx` and kernel (re-)launches in a loop as shown in listing 3. `slateIdx`'s value is zero before the initial launch but increases as kernel execution progresses. In the kernel re-launched to the adjusted SM range, its updated value points to the beginning of the remaining tasks.

With the dispatch kernel, threads now have three exit conditions. (1) They run on an undesignated SM. In this case, they quit as described in Listing 1. (2) They run on the designated SMs and execute the entire user kernel tasks. In this case, they are workers that are launched once and persist through. (3) They run on the designated SMs and execute a portion of the user kernel tasks. In this case, they are workers but either terminated early or launched late when the available resource changes.

```
extern "C" __global__
void exampleDispatcher(volatile uint* start_sm,
                       volatile uint* end_sm, ...) {
    cudaStream_t s;
    cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
    // global scheduling change flag
    retreat = 0;
    // global task queue
    slateIdx = 0;
    do {
        // launches with sm arguments and user arguments
        example<<<grid, block, SHARED_MEM,
                  s>>>(*start_sm, *end_sm, ...);
        cudaDeviceSynchronize();
        retreat = 0;
    // if job is incomplete, start over
    } while (slateIdx < slateMax);
    cudaStreamDestroy(s);}
```

Listing 3: **Dispatch Kernel.** The dispatch kernel, corresponding to the example code in listing 1, launches the user kernel to a designated range of SMs and re-launches to adjusted ranges if needed.

V. Results and Evaluation

In this section we present the performance of *Slate* for various applications on a real GPU accelerated system. The system consists of an Intel Xeon E5-2670 CPU with 20 physical cores, 64GB DDR3 memory, and an NVIDIA Titan Xp card, which has 30 SMs of the GP102 Pascal GPU architecture and 12 GB DGGRX5 global memory. By default, the Pascal GPU architecture combines the functionality of the L1 and texture caches into a unified L1/Texture cache. The system runs Fedora 26 OS and CUDA toolkit 9.1.

A. Evaluation Methodology

TABLE II: Benchmarks and their profiles. All profile data are acquired using the `nvprof` tool and its event collection [13] when the original programs run solo with CUDA on the system. Memory bandwidth is the sum of the global load and global store bandwidth as reported by `nvprof`. FLOPS are reported as the total number of single-precision floating point operations over the kernel execution time.

Benchmark	Compute Intensity	Memory Intensity	GFLOP/s	Mem. BW (GB/s)
BlackScholes (BS)	Med	Med	161.3	401.49
Gaussian (GS)	Low	Med	19.6	340.9
SGEMM (MM)	High	Med	1,525	403.5
QuasiRandom Generator (RG)	Low	Low	4.2	71.6
Transpose (TR)	Low	High	0.0	568.6

1) Applications: The applications, presented in Table II, are from the NVIDIA CUDA 8.0 samples and the Rodinia benchmark suite [4]. With their diverse resource requirements and workload characteristics, they can form pairs that are complementary or interfering. For each benchmark, we keep its original version and also create a modified version that works with the *Slate* API and runtime.

2) Evaluation Metrics: The ultimate performance metrics we use in evaluation include: application execution time, kernel execution time, and normalized execution time if multiple applications concurrently run on the system. We collect these data when the applications run solo on all SMs and run concurrently in different pairs. To illustrate why execution time differ with scheduling methods, we also present lower level performance data including compute rate in FLOPS and memory bandwidth.

These performance metrics are used to evaluate and compare three scheduling methods: vanilla CUDA, MPS, and *Slate*. Vanilla CUDA uses time slicing, if there are multiple active kernels, and allocates all SM resources to one and switches to another the next time. MPS uses the leftover policy, which, for our applications, only allows concurrent kernels when there are SMs available near the end of a prior kernel's execution. *Slate* uses spatial sharing, but it selects complementary kernels for optimal resource utilization. *Slate* requires the compatible version that invokes the *Slate* API, while the others work with the original version of the applications.

3) Data Collection: To ensure that timing can reliably reflect performance, the timed programs and code segments must run sufficiently long. To meet this requirement for short running programs, researchers typically repeat them in a loop and time the loop. We adopt this method in our evaluation. Specifically, for each application, we first select a proper problem size that is large to reach its highest performance when the original version runs solo with CUDA. We then loop the kernels for a number of repetitions such that the loop takes $\sim$30 seconds to finish. Lastly, we apply the same number of repetitions to the *Slate* compatible version. All timing data that we report in this section are for the looped kernel execution.

We use the `nvprof` tool and its event collection [13] to collect other metrics including IPC, cache accesses, floating point operations, and memory accesses[1]. This collection is non-intrusive to application execution.

B. Single Application Solo Run

We first present solo execution performances with *Slate*, CUDA, and MPS. Solo execution is the base case and common in real world when there is only one application on the system. Solo performance evaluation demonstrates the overhead of these systems. Here we focus on kernel execution time, which excludes the cost of code injection and runtime compilation. Kernel time can highlight the benefits of *Slate*'s software-based kernel scheduling.

Based on our experiment data, *Slate* outperforms vanilla CUDA with a noticeable margin for all but one kernel. *Slate* performs the best with a 28% gain for Gaussian (GS), which has fairly intensive, regular memory accesses and less intensive arithmetic operations.

For solo runs, it is *Slate*'s basic software-based scheduling that leads to the performance improvement, for its resource sharing and dynamic sizing do not take action. The basic scheduling creates a number of worker blocks that the SMs can maximally support simultaneously. This number is much smaller than what users have specified, and thus reduces the thread setup time. In addition, it schedules tasks in-order from a queue, and the tasks are coarser grained than blocks that hardware scheduler uses. This task scheduling has two main advantages. First, it leverages data locality in user kernels thank to the ordered block execution. Second, it reduces the number of synchronization in scheduling. We set the default task size as 10 blocks.

With the default task size, *Slate* under performs for BS for 5% due to load imbalance between workers. However, *Slate* outperforms CUDA for BS by 2% if the task size is set at 1.

TABLE III: Detailed performance metrics of the Gaussian Elimination (GS) benchmark. *Slate* improves L1 performance and leverages the regular access patterns of Gaussian, and reduces memory throttles.

Metric	CUDA	Slate	Δ%
IPC	0.36	0.47	+30
Mem. Access BW (GB/s)	287	396	+38
% Stalls: Mem Throttle	26.1%	0%	-26.1
Execution Time (s)	24.7	18.9	$+28$

To illustrate the performance improvement at lower hardware level, we further compare how performance events differ with scheduling. Here we use GS as the example. Table III presents its hardware events and metrics under the CUDA runtime and *Slate* respectively. Using *Slate* scheduling, GS achieves higher cache hit rate, memory bandwidth, and IPC, confirming that *Slate*'s basic scheduling leads to better data locality and smaller thread setup cost. Note that the IPC improvement is slightly larger than the kernel time reduction, which complies with the fact that the *Slate* version kernel has additionally injected instructions.

C. Concurrent Kernels and Resource Sharing

We now evaluate the scenario where two applications run on the system. We run all possible pairs from the applications with MPS and *Slate*, either of which transparently decides it should run the kernels consecutively or concurrently. Here we report the normalized kernel execution time to focus on the utilization of the GPU resource.

Experimental results indicate that *Slate* performs significantly better than MPS for some pairs and slightly better for others. For the former cases, *Slate* runs the kernels concurrently because it identifies they are complementary. Take the RG-BS pair as an example. RG is of low compute and memory intensity, unable to fully utilize GPU's capacity in processing and memory data transfer. Meanwhile, BS is relatively compute and memory intensive. In contrast, with the leftover policy, MPS basically runs these kernels consecutively because the large number of blocks and threads that are created to hide data access latency prevents spatial sharing. For this RG-BS pair, *Slate* achieves a 30.55% higher throughput than MPS. For the other pairs, *Slate* runs them solo consecutively as MPS does, but outperforms MPS by 8% on average. Such

[1]The corresponding metrics include l2_read/write_throughput, gld_throughput, gst_throughput, flop_count_sp, flop_count_dp

throughput improvement mainly comes from *Slate*'s software-based scheduling, as discussed in section V-B.

TABLE IV: The performance of the BS-RG pair. Compared to MPS, *Slate* increases bandwidth and IPC.

Metric	MPS	Slate	Δ%
Global/L2 Throughput (GB/s)	241	250	+3.84
Load/Store Executed (million)	151	140	−9
Instructions Per Cycle	0.94	1.61	+71.28
Throughput Gain from *Slate*	30.55%		

For the BS-RG pairing, *Slate* greatly improves memory access throughput and execution rate on the device, thank to the kernel-SM mapping and spatial sharing. *Slate* achieves a 71.28% higher IPC than MPS, indicating that it effectively hides the data access latency and better utilizes the processing units, as shown in Table IV.

D. Slate Overhead Evaluation

Here we evaluate the overhead of *Slate* to provide an comprehensive analysis. *Slate* introduces extra operations that incur time cost. These operations are summarized in Table V. Some are inside the application execution but outside the kernel execution, some are inside in the kernel execution.

Because *Slate* collects the kernel profiles at kernels' first time runs and saves the profiles in a table, kernel profiling is considered offline. *Slate* references the table online and this lookup cost is negligible and included in the kernel execution. As discussed earlier, the profiling using `nvprof` is non-intrusive.

TABLE V: *Slate* introduced operations and their scope.

Scope	Operation
Inside kernel exec	Exec of injected instructions Atomic ops on the task queue
Outside kernel exec	Dynamic code injection & compilation Client-daemon communication
Offline	Kernel profiling to build lookup table

1) Cost inside Kernel Execution: Even though *Slate* incurs some extra costs in the kernel execution, it still outperforms vanilla CUDA and MPS, as we see in the discussions of solo and concurrent kernel execution. Nevertheless, here we analyze them to shed light on necessary expenses in *Slate* and potential optimizations. Because these costs cannot be quantified in isolation, we try to provide qualitative estimations.

The cost is due to the code segments inserted into the user kernels in Listings 1 and 2. There are two kinds of injected instructions. The first is those that every thread executes, and the second is those atomic operations for task scheduling and synchronization, which are serial and limit the performance gain from the massive parallelism on CUDA devices.

We use the BlackScholes (BS) kernel to illustrate the overhead. BS has a predictable behavior, which executes 157.5 million instructions per kernel launch for a problem size of $N = 40$ million. The *Slate* version generates about 4 million or 3% more instructions on average.

Slate uses atomic operations to manage the task queue. To reduce the cost of synchronization, *Slate* schedules 10 blocks (1 task) at a time by default. We show in Figure 5 that this granularity improves kernel execution time. GS is the example for this case. Its kernel time almost halves with the task size of 10. Nevertheless, a very large value may cause workload imbalance among the workers. For example, the task size of 10 is worse than the task size of 1 for BS.

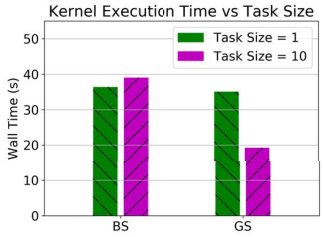

Fig. 5: Effects of task size on kernel execution time with *Slate*.

2) Overhead outside Kernel Execution: *Slate* also introduces costs that are outside the kernel execution but inside application execution. It takes additional time to insert code segments to user kernels and compile them, and handle the client-daemon communications.

Slate's overhead can be quantitatively analyzed from Figure 6. To collect the data, we run single applications with different schedulers, and time their application and kernel executions. With *Slate*, we further time the kernel code injection and dynamic compilation, and communications. In the worst case, *Slate* has the same application execution time as CUDA and MPS. In the best case of GS, *Slate* outperforms them by 28%. Note that MPS generally has a slightly larger application time than CUDA. This is because MPS uses an intermediate daemon in its design.

On average, *Slate* takes 4% of the application execution time to handle the client-daemon communications, and 1.5% to inject codes and compile the kernels. The communication cost is non-negligible, which *Slate* must pay to funnel the contexts from multiple processes due to the lack of GPU hardware interface.

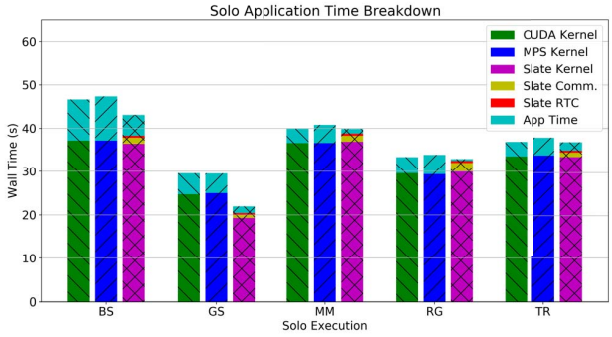

Fig. 6: Application solo execution time with CUDA, MPS and *Slate*. The measured application time is the full bar, and the measured kernel time is the bottom bar. Their difference is the host time, which covers application setup, CUDA basic functions, and data transfers. For *Slate*, we further isolate the time of communication, and code injections and dynamic compilation.

E. Overall Performance with GPU Multiprocessing

Lastly we present the system throughput for multiple processes. We run all possible 15 pairings of the applications and measure the normalized application execution time with

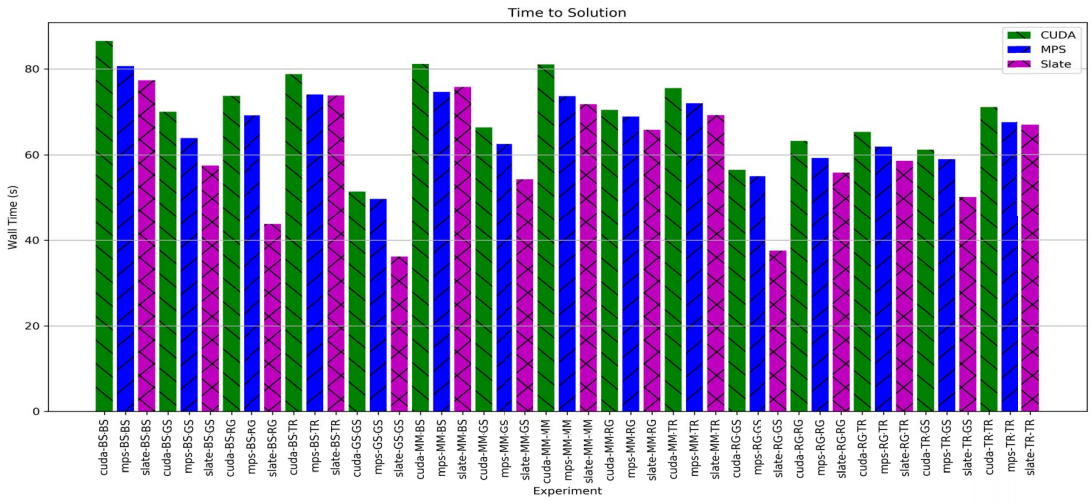

Fig. 7: *Slate* outperforms the CUDA runtime for all application pairings, and outperforms MPS for all but the MM-BS pairing.

vanilla CUDA, MPS and *Slate* respectively. The results are shown in Figure 7. The vanilla CUDA provides the base for comparisons. MPS performs about 6% better than CUDA. MPS cannot effectively corun the large kernels under study; the kernels run consecutively for most of the time. Overall, *Slate* outperforms the vanilla CUDA on all the pairings, and MPS on all but one pairing. On average, *Slate* improves the throughput by 11% over MPS, and 18% over CUDA. In the best case of the `RG-GS` pairing, *Slate* delivers a 35% faster application execution than MPS. The MM-BS pairing is an exception, for which *Slate* under performs by 2% than MPS due to load imbalance for BS with the default task size.

Two techniques in *Slate* contribute most of the performance gain. One is the workload-aware concurrent kernel execution. *Slate* concurrently runs RG with all the other kernels. Of these pairings, BS-RG and RG-GS have the highest throughput improvement, because RG is neither memory nor compute intensive, complementing well with BS and GS that are fairly memory intensive. The pairings of MM-RG and RG-TR have a smaller gain because MM is highly compute intensive and TR is highly memory intensive. The other contributor is the basic software-based scheduling. For all pairings for which *Slate* gains less than 10%, *Slate* runs the kernels consecutively. One special case is GS-GS, for which *Slate* gains 24% throughput with consecutive solo runs. *Slate* significantly improves GS's data locality and memory access performance with the in-order block execution.

These results provide compelling evidence that there is large room for CUDA scheduling optimization, and software-enabled scheduling can effectively overcome the hardware limitations to a great extent.

VI. Related Work

GPU scheduling and concurrent kernels have been studied in several existing bodies of work. These techniques usually take one of three forms: (1) MPS-like that allows applications to run together online without user intervention and explicit resource management, (2) static kernel merging that combines two kernels into a single process at compilation time directly or indirectly, and (3) hardware modification concepts that are evaluated with simulation.

As the proprietary solution with its left-over policy, MPS provides no support for other scheduling policies [15]. To circumvent this limitation, a similar attempt, Mystic, relies on an MPS-like context funneling system [22]. Mystic enables concurrent kernels but does not make scheduling decisions.

Existing software techniques consider two applications at or before compile-time, and many rely on persistent threads to control GPU resources [9]. Gregg et al. developed *KernelMerge*, an OpenCL runtime wrapper that interacts with its own scheduling algorithm [8]. Free Launch used compiler techniques to statically combine a kernel with *child* kernels [6]. Wang et al. developed Kernel Fusion, a source-to-source compiler that can combine certain kernels for specific archetypes, such as map or reduce functions [23]. Wu et al. devised SM-Centric program transformations [26] that statically map jobs to GPU SMs in program source codes. By identifying co-runnable applications and combining them in a single program, SM-centric approach can materialize spatial sharing of the SMs and job co-running. Distinct from this work, *Slate* enables concurrent kernels from arbitrary applications at runtime and integrates workload-awareness into its scheduling decisions.

Hardware modifications have been proposed to introduce different scheduling strategies at the lowest level. Pai et al. proposed to shape CUDA grids with persistent threads [16] to allow concurrent execution of CUDA applications and finite resource control. Similarly, the concepts of spatial partitioning and persistent threads were simulated [16]–[18], [25], [27] or modeled [20], many of which required hardware-enabled kernel preemption [21] that is not yet available on modern cards. Simulations were also performed for process-in-memory capabilities on GPUs [19]. Different from this work, *Slate* enables effective concurrent kernels on today's hardware architectures. *Slate* supports the hardware-assisted kernel selection presented in [24], [29].

Concurrent kernels have been proposed for embedded sys-

tems. Effisha used software transformation techniques to preempt kernels [7] at the cost of kernel progress. Lee et al. developed a runtime application framework for GPU-based embedded systems exclusively for event-driven applications [11]. Their methodology assumed zero latency between host and GPU, unrealistic on real systems.

VII. Conclusion

This paper presents the methodology, design, and evaluation of *Slate*, a workload-aware, cost-effective kernel scheduling and multiprocessing framework for modern GPUs. *Slate* addresses the issue of inefficient utilization of modern GPUs with advanced kernel scheduling and resource sharing. Meanwhile, *Slate* promotes workload-aware runtime design and introduces several techniques to make it practical on real systems.

Using *Slate*, multiple applications can share GPU resources based on their workload characteristics, mutual interference, and system utilization status. Our prototype *Slate* framework has shown up to 35% and an average of 11% throughput improvement over the tested application pairs in comparison with MPS. More importantly, as a software-based solution, *Slate* works on most GPU systems and can be continuously improved. Further, *Slate* is open-source and provides a platform for future GPU multiprocessing research.

Acknowledgements

This work is supported in part by the U.S. National Science Foundation under Grants CCF-1551511 and CNS-1551262. We thank NVIDIA for their donation of the Titan Xp used for this research.

References

[1] Gk110 whitepaper, 2014. https://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-architecture-whitepaper.pdf.

[2] NVRTC - CUDA Runtime Compilation User Guide. Technical report, NVIDIA, 2016. http://docs.nvidia.com/cuda/pdf/NVRTC_User_Guide.pdf.

[3] I. Buck. Reaching the Summit: Accelerated computing powering world's fastest supercomputer, 2018. https://blogs.nvidia.com/blog/2018/06/08/worlds-fastest-exascale-ai-supercomputer-summit/.

[4] S. Che, J. W. Sheaffer, M. Boyer, et al. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, IISWC '10. IEEE Computer Society, 2010.

[5] C. Chen, Y. Du, H. Jiang, K. Zuo, and C. Yang. Hpcg: Preliminary evaluation and optimization on tianhe-2 cpu-only nodes, Oct 2014.

[6] G. Chen and X. Shen. Free launch: Optimizing gpu dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 407–419, New York, NY, USA, 2015. ACM.

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

[8] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent gpgpu kernels. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012.

[9] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing (InPar), 2012*, pages 1–14. IEEE, 2012.

[10] M. Hassaan and I. Elghandour. A real-time big data analysis framework on a cpu/gpu heterogeneous cluster: A meteorological application case study. BDCAT '16, New York, NY, USA, 2016.

[11] H. Lee and M. A. A. Faruque. Run-time scheduling framework for event-driven applications on a gpu-based embedded system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1956–1967, 2016.

[12] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. Top500 list - june 2018, 2018.

[13] NVIDIA. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview, 2010.

[14] NVIDIA. GV100 CUDA hardware and software architectural advances, 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[15] NVIDIA. Multi-process service, 2018. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[16] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013. ACM.

[17] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 593–606, New York, NY, USA, 2015. ACM.

[18] J. J. K. Park, Y. Park, and S. Mahlke. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 527–540, New York, NY, USA, 2017. ACM.

[19] A. Pattnaik, X. Tang, et al. Scheduling techniques for gpu architectures with processing-in-memory capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 31–44, New York, NY, USA, 2016. ACM.

[20] T. Sorensen, H. Evrard, and A. F. Donaldson. Cooperative kernels: Gpu multitasking for blocking algorithms. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 431–441, New York, NY, USA, 2017. ACM.

[21] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, June 2014.

[22] Y. Ukidave, X. Li, and D. Kaeli. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[23] G. Wang, Y. Lin, and W. Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350, Dec 2010.

[24] H. Wang, F. Luo, M. Ibrahim, and et al. Efficient and fair multi-programming in gpus via effective bandwidth management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

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

[26] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 119–130, New York, NY, USA, 2015. ACM.

[27] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, 2016.

[28] J. Zhang, S. You, and L. Gruenwald. Large-scale spatial data processing on gpus and gpu-accelerated clusters. *SIGSPATIAL Special*, 6(3):27–34, Apr. 2015.

[29] X. Zhao, Z. Wang, and L. Eeckhout. Classification-driven search for effective sm partitioning in multitasking gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 65–75, New York, NY, USA, 2018. ACM.