Pagoda: A GPU Runtime System for Narrow Tasks

TSUNG TAI YEH, Purdue University
AMIT SABNE, Microsoft
PUTT SAKDHNAGOOL, National Electronics and Computer Technology Center
RUDOLF EIGENMANN, University of Delaware
TIMOTHY G. ROGERS, Purdue University

Massively multithreaded GPUs achieve high throughput by running thousands of threads in parallel. To fully utilize the their hardware, contemporary workloads spawn work to the GPU in bulk by launching large tasks, where each task is a kernel that contains thousands of threads that occupy the entire GPU.

GPUs face severe underutilization and their performance benefits vanish if the tasks are narrow, i.e., they contain less than 512 threads. Latency-sensitive applications in network, signal, and image processing that generate a large number of tasks with relatively small inputs are examples of such limited parallelism.

This article presents Pagoda, a runtime system that virtualizes GPU resources, using an OS-like daemon kernel called MasterKernel. Tasks are spawned from the CPU onto Pagoda as they become available, and are scheduled by the MasterKernel at the warp granularity. This level of control enables the GPU to keep scheduling and executing tasks as long as free warps are found, dramatically reducing underutilization. Experimental results on real hardware demonstrate that Pagoda achieves a geometric mean speedup of 5.52X over PThreads running on a 20-core CPU, 1.76X over CUDA-HyperQ, and 1.44X over GeMTC, the state-of-the-art runtime GPU task scheduling system.

CCS Concepts: • **Software and its engineering** → *Compilers*; *Runtime environments*;

Additional Key Words and Phrases: GPU runtime system, utilization, task parallelism

ACM Reference format:
Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2019. Pagoda: A GPU Runtime System for Narrow Tasks. *ACM Trans. Parallel Comput.* 6, 4, Article 21 (November 2019), 23 pages.
https://doi.org/10.1145/3365657

1 INTRODUCTION

GPGPU computing has demonstrated an ability to accelerate a substantial class of compute-intensive applications [8, 37]. These applications are often composed by a high degree of parallelism, where iterations of large parallel loops are executed on the GPU. The significant performance improvement is shown in this type of applications, since GPU's hardware resources are fully utilized by launching enough concurrent threads.

This article was based on "Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks", published in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.

Authors' addresses: T. T. Yeh (corresponding author) Purdue University, 465 Northwestern Ave, West Lafayette, IN, USA; email: yeh14@purdue.edu; A. Sabne; email: sabne@microsoft.com; P. Sakdhnagool; email: putt.sakdhnagool@nectec.or.th; R. Eigenmann; email: eigenman@udel.edu; T. G. Rogers; email: timrogers@purdue.edu.

2329-4949/2019/11-ART21 $15.00
https://doi.org/10.1145/3365657

The GPU's performance benefits start to diminish as the degree of parallelism lessens. Conventionally, large parallel loops are offloaded to the GPU, while retaining the execution of smaller ones on the CPU. The main thesis of this article is that, despite having a smaller degree of parallelism, applications can benefit from using the GPU, provided that the involved task (or CUDA kernel) count is sufficiently high. Each such task, called a *narrow task*, has limited parallelism (which we empirically defined as < 500 data parallel threads).

Narrow tasks emerge in a number of scenarios. One of such applications are often shown in latency-driven, real-time workloads. For example, Ethernet is limited by Maximum Transmission Unit (MTU) and carries out small Ethernet packets (maximum IEEE 802.3 Ethernet packet size is 1500 bytes [1]), resulting in packets (tasks) with low parallelism. Ethernet routers can receive many tasks from many sources in a quick succession and require immediate processing. These workloads have been characterized as having mixed task and data parallelism [33, 34]. Second, *irregular* applications can exhibit narrow tasks. These applications often contain varying amounts of computation among different threads, and/or among loop iterations. To reduce load imbalance, these applications are often represented using many tasks with low degrees of parallelism [25]. Irregular workloads may also arise in multi-programmed environments. Different applications with low degrees of parallelism can be co-executed on a node to exploit all the computing resources.

GPU underutilization is the key reason why narrow tasks are conventionally executed on CPUs and ignored by GPU application developers frequently. This article presents Pagoda,[1] a runtime system that greatly improves GPU utilization in the presence of narrow tasks. Pagoda introduces novel elements of a massively parallel OS to virtualize and dynamically schedule GPU core resources at warp granularity, enabling hundreds of tasks to execute concurrently.

Prior work has identified the issue of GPU underutilization [9, 15, 26, 38]. One approach to solve this problem is to statically fuse multiple smaller tasks [9, 38] to accumulate a large kernel. Advanced approaches [26, 29] use a concurrent kernel mechanism, monitoring and time-slicing their execution at runtime to obtain fair sharing. These static approaches require the programmer to fuse tasks *manually* and none of them have been shown to work beyond ten concurrent tasks. These mechanisms also require static knowledge of the kernels to be fused, which is not always possible in multi-programmed or real-time environments. Additionally, individual tasks in a fused tasks receive the same on-chip resource allocation, e.g., shared memory and registers, thereby limiting occupancy based on the resource requirements of the largest task.

Dynamic (runtime) solutions can mitigate the above issues of static fusion. NVIDIA's current-generation GPUs employ HyperQ [21], which allows 32 tasks to be concurrently executed on one GPU. However, we observed that narrow tasks can still cause underutilization, as 32 such tasks may not occupy the entire GPU.

We argue that software mechanisms are needed to achieve flexible kernel concurrency. Prior work, *GPU enabled Many-Task Computing* (GeMTC) [15], presents a runtime task-scheduling mechanism, where a task executes as a single *threadblock*. *Threadblocks* are sets of threads constituting the GPU kernel. Because GPU architectures limit the concurrent threadblock count, executing narrow tasks in GeMTC may result in poor utilization. In addition, GeMTC uses batch-based task execution, which results in delayed task launching and load imbalance since the completion time of a batch is determined by its longest running task.

Pagoda is designed to overcome these issues. The programmer replaces certain CUDA API calls with equivalent Pagoda calls in the host and device codes, retaining the functionality of the CUDA programming model. Unlike static solutions, the programmer does not have to tediously fuse the available tasks. Pagoda achieves high utilization by continually running a *MasterKernel*, which

[1]Download Pagoda from http://bit.ly/2hmmY5p.

controls the execution of all GPU *warps* in software. In Pagoda, tasks are spawned by the CPU as soon as they become available, without batching. On the GPU side, the MasterKernel virtualizes the GPU's resource allocation and threadblock scheduling mechanism to allow individual warps to make progress as soon as resources are available.

There are three key challenges that must be addressed when attempting to launch and run thousands of short-running tasks on a GPU, the combination of which no previous work has solved.

First, CPU-GPU communication overhead must be minimized, while allowing the GPU to asynchronously schedule new tasks on each Streaming Multiprocessor (SM). Launching thousands of short-running tasks increases the importance of minimizing the time it takes for each task to begin execution on the GPU. Since the CPU and GPU must coordinate task spawning and scheduling over the PCIe bus, which currently has no support for atomic operations, the handshaking required is expensive or impossible if a traditional data structure, such as a queue [19], is used. Previous work that required OS-like co-ordination over PCIe [13, 32] solved consistency issues using a producer-consumer model but did not have to optimize the system for many, short running tasks. To support narrow tasks, Pagoda presents *TaskTable*, a novel data structure aimed at limiting communication overhead and enabling asynchronous GPU task pulls.

The second challenge is to keep the overheads involved in task spawning and scheduling low. Minimizing both the copying of task parameters and the search for free GPU resources is important when task execution times are short. To limit these overheads, Pagoda performs task scheduling in parallel and pipelines task spawning, scheduling and execution to overlap their operation.

The third issue is supporting native CUDA functionality such as shared memory usage and efficient threadblock synchronization. Since Pagoda's MasterKernel overrides native support for this functionality, we introduce low-overhead software mechanisms to provide it.

In summary, the following contributions are made to enable the efficient execution of narrow tasks on GPUs. This article

- introduces a continuous task spawning mechanism to reduce CPU-GPU synchronizations and obtains a high spawn rate.
- presents a software mechanism to schedule multiple tasks on the GPU in parallel, and describes a pipelining scheme to overlap several task processing stages.
- describes software solutions for dynamic shared memory management and sub-threadblock synchronizations.
- methods in a new runtime system, called Pagoda. Pagoda achieves geometric mean speedup of 5.52x over PThreads running on a 20-core CPU, of 1.76x over CUDA-HyperQ, and of 1.44x against GeMTC.

2 GPU PROGRAMMING AND ARCHITECTURE

The GPU cores are organized into 28 Streaming Multiprocessors (SMs).[2] Each SM has 128 CUDA cores and can concurrently schedule up to 64 warps. A *Warp* is the basic Single Instruction, Multiple Thread (SIMT) work unit, which comprises 32 threads that march in lockstep, executing the same instruction. Each SM has a 96KB on-chip programmer managed cache, known as *shared memory* and 64K, 32-bit registers.

In the CUDA programming model, the programmer organizes parallel work in *kernels*. Threads of a kernel are grouped into *threadblocks*. Multiple threadblocks can reside on each SM, the maximum number being 32. The threadblock size is limited to 1024 threads, or 32 warps. Each SM can hold up to 2048 concurrent threads. Both the shared memory and registers of an SM are partitioned

[2]We use terminology from the NVIDIA Pascal Titan X architecture.

Table 1. Pagoda Programming API

CUDA Function	Pagoda Function	Caller	Return Value	Arguments	Description
kernel<<<>>>	taskSpawn	CPU	taskId	#threads, #threadblocks, shared memory, sync flag, kernel pointer, kernel args	Spawn a task from CPU onto Pagoda
cudaEventSynchronize	wait	CPU		taskId	Wait until the specified task has finished
cudaEventQuery	check	CPU	true if the task is done, else false	taskId	Returns the status of the task
cudaDeviceSynchronize	waitAll	CPU			Wait until all tasks in Pagoda have finished
threadIdx	getTid	GPU	thread Id		Get the thread Id of this thread
syncthreads	syncBlock	GPU			Synchronize all threads in the block
__shared__ char *arr	getSMPtr	GPU	32-byte aligned char pointer		Get shared mem pointer for the threadblock

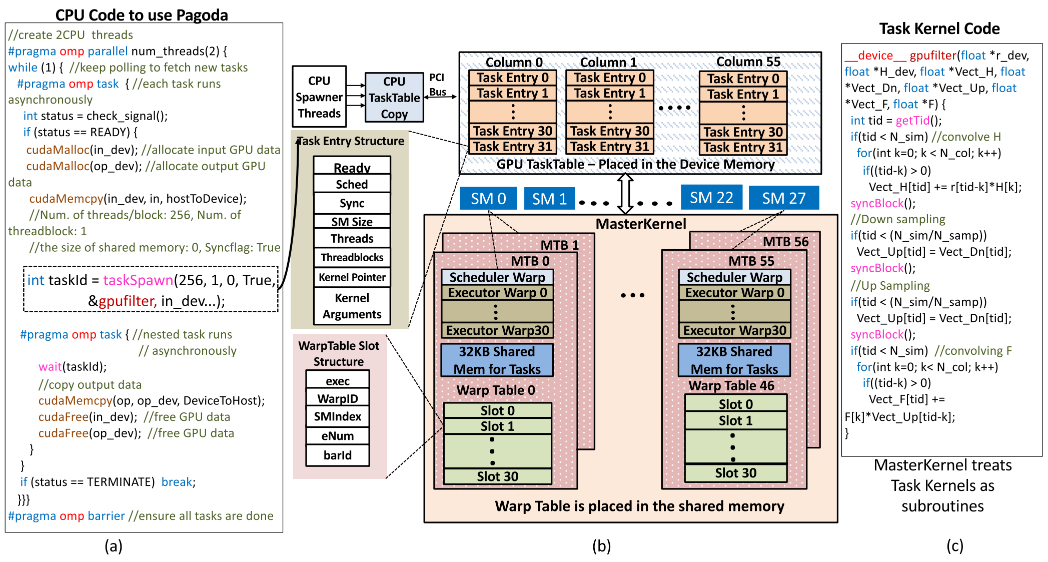
among the executing threadblocks. There is not a CUDA primitive for global, kernel-wide synchronization; however, threads in a threadblock can use the *__syncthreads()* function as a barrier.

A way of measuring the GPU utilization is *occupancy*. Occupancy is the ratio of the total number of resident GPU warps divided by the maximum number of warps that can co-exist in the GPU (i.e., 64 × the number of SMs in the GPU). The kernel occupancy is affected by three factors, namely, (i) size of threadblocks, (ii) kernel's register count, and (iii) size of the requested shared memory. Balancing these three factors requires programmer expertise, making high occupancy often difficult to achieve. In one scenario of narrow tasks, where one task contains 256 threads, or 8 warps. If only one task is executed on the GPU at a time, the occupancy would be $(8/(64 \times 28)) \times 100\% = 0.45\%$. With HyperQ, 32 kernels may co-execute, meaning that 32 narrow tasks can run simultaneously. The achieved occupancy then would still be low, i.e., $(8 \times 32/(64 \times 28)) \times 100\% = 14.29\%$.

3 PAGODA PROGRAMMING PRIMITIVES

Programmers are required to use Pagoda API functions in their applications to access the Pagoda runtime system. Pagoda follows programming primitives of CUDA programming model to tailor new Pagoda APIs. Pagoda API functions shown in Table 1 override the corresponding CUDA functions. The Pagoda API functions belong to the following two categories:

CPU-Side API. The *taskSpawn* function launches a task from the CPU onto Pagoda. The programmer specifies the number of threads per threadblock, and the number of threadblocks as arguments. The programmer also specifies the kernel to execute, along with the parameters. The size of the shared memory needed per threadblock in bytes may be specified. The *sync* flag indicates if threadblock-level synchronization is necessary for the task. *TaskSpawn* is a non-blocking function. The CPU can synchronize with the spawned task(s) using *wait* and *waitAll* functions, or can check the task status with *check* function. One difference in functionality with respect to CUDA is that Pagoda returns a *taskID* for each task. The taskIDs are essential to use functions such as *wait*.


CPU Code to use Pagoda

```
//create 2CPU  threads
#pragma omp parallel num_threads(2) {
while (1) {  //keep polling to fetch new tasks
  #pragma omp task  { //each task runs
asynchronously
    int status = check_signal();
    if (status == READY) {
     cudaMalloc(in_dev); //allocate input GPU data
     cudaMalloc(op_dev); //allocate output GPU
data
      cudaMemcpy(in_dev, in, hostToDevice);
      //Num. of threads/block: 256, Num. of
threadblock: 1
      //the size of shared memory: 0, Syncflag: True

     int taskId = taskSpawn(256, 1, 0, True,
                 &gpufilter, in_dev...);

   #pragma omp task { //nested task runs
                    // asynchronously
      wait(taskId);
      //copy output data
      cudaMemcpy(op, op_dev, DeviceToHost);
      cudaFree(in_dev);  //free GPU data
      cudaFree(op_dev);  //free GPU data
    }
  }
  if (status == TERMINATE)  break;
 }}}
#pragma omp barrier //ensure all tasks are done
```

(a)

(b)

Task Kernel Code

```
__device__ gpufilter(float *r_dev,
float *H_dev, float *Vect_H, float
*Vect_Dn, float *Vect_Up, float
*Vect_F, float *F) {
 int tid = getTid();
 if(tid < N_sim) //convolve H
  for(int k=0; k < N_col; k++)
   if((tid-k) > 0)
    Vect_H[tid] += r[tid-k]*H[k];
 syncBlock();
 //Down sampling
 if(tid < (N_sim/N_samp))
  Vect_Up[tid] = Vect_Dn[tid];
 syncBlock();
 //Up Sampling
 if(tid < (N_sim/N_samp))
  Vect_Up[tid] = Vect_Dn[tid];
 syncBlock();
 if(tid < N_sim)  //convolving F
  for(int k=0; k< N_col; k++)
   if((tid-k) > 0)
    Vect_F[tid] +=
F[k]*Vect_Up[tid-k];
}
```

MasterKernel treats Task Kernels as subroutines

(c)

Fig. 1. Pagoda runtime system overview. The source task kernel and CPU code require few changes to an equivalent CUDA code. The MasterKernel design is shown for NVIDIA Pascal Titan X GPU. The 56 MasterKernel threadblocks (MTBs) have 1024 threads each. TaskTable is mirrored on both the CPU and GPU. The CPU threads spawn tasks into the CPU TaskTable, which are then sent to the GPU counterpart. Scheduler warps inside each MTB find free executor warps to launch tasks on. The WarpTable performs bookkeeping for each executor warp.

GPU-Side API. Since Pagoda virtualizes the GPU resources, the CUDA-based shared memory allocation and threadblock synchronization cannot be directly used. Pagoda allocates shared memory and barriers for each threadblock when it gets scheduled, and the API provides functions that allow the threadblock to obtain a pointer to its shared memory, and to perform barrier synchronizations. Pagoda also offers a function to obtain the threadId of the current thread.

Figure 1(a) shows a possible implementation of Pagoda host code, while Figure 1(c) shows the corresponding device code for FilterBank. The two CPU threads spawn tasks and wait for their completion. Calling *wait()* in a nested task allows the CPU thread to progress, without getting blocked. One key distinction from CUDA is that the task kernels are written as *__device__* functions, instead of *__global__*. In CUDA, GPU functions declared as __global__ are called kernels and are called from the host. Functions declared as __device__ can only be called from the device side.

4 PAGODA RUNTIME SYSTEM

This section first describes the design of MasterKernel that achieves GPU resource virtualization. Next, the Pagoda task spawning mechanism is described in Section 4.2. Pagoda task spawning mechanism employs TaskTable, a novel data structure that allows simultaneous updates from both the CPU and GPU, and is mirrored in both their memories. TaskTable drastically reduces the CPU-GPU handshaking communication by allowing lazy aggregate updates. Last, the section presents Pagoda's GPU scheduling mechanism that parallelizes the scheduling process, and overlaps various task processing stages.

4.1 Resource Virtualization via MasterKernel

The MasterKernel is a *__global__* CUDA kernel, offered by all GPU resources such as warps, shared memory, and registers. The MasterKernel is launched at the start of Pagoda runtime and is continually executed until the end of the computation of entire tasks. The kernels of tasks are translated into __*device*__ kernels of the MasterKernel. The MasterKernel allocates its own resources to these sub-kernels dynamically in Pagoda.

Figure 1 describes our MasterKernel design on the NVIDIA Pascal Titan X GPU. The MasterKernel acquires all warps of each SM (64) by launching two, 32-warp threadblocks, called MTBs (MasterKernel Threadblocks). Each MTB statically allocates 32 KB shared memory, which later gets assigned to different tasks. The MasterKernel uses the remaining shared memory of the SM to store some of the scheduling data sturctures. The register count of each thread is capped at 32 (using -maxrregcount) to ensure 100% occupancy for the MasterKernel.

In Figure 1, the first warp of each MTB is called a *scheduler warp*, while the remaining 31 warps are called *executor warps*. The scheduler warp is responsible for scheduling tasks on the executor warps in the MTB. It also manages shared memory allocations and barriers. The MasterKernel contains two scheduling data structures. The first one, called *TaskTable*, is mirrored on the CPU and GPU, and is used for task spawning. Each entry in the TaskTable holds a task. The GPU TaskTable receives online updates from the CPU TaskTable, and is therefore placed in the GPU device memory. The second data structure is called *WarpTable*. Each MTB contains its own WarpTable, which is placed in the shared memory. Every WarpTable contains 31 slots to maintain the status of each executor warp.

4.2 Continuous Task Spawning

Scheduling algorithms often involve queues that accumulate tasks, where processing elements pull tasks from the queue [32]. To simultaneously schedule several tasks, multiple pulls must take place in a synchronous/atomic manner, which has long been recognized as a critical source of overhead [19]. Performing global synchronizations or atomic operations on GPUs is extremely expensive. Therefore, to reduce this contention, one solution would be to use multiple queues, and only let a smaller set of GPU threads pull from each queue. Even this solution is impractical. As the CPU-GPU memories are discrete, before the CPU could spawn a task on a GPU queue, it must gather the queue *head* and *tail* pointers from the GPU. Such handshaking is expensive because it requires data copies over the PCIe bus. Another way to spawn tasks is to use a batch-based mechanism [15], where the CPU sends a batch of tasks to the GPU. However, such mechanisms are susceptible to load imbalance across tasks.

The Pagoda design therefore employs TaskTable, a data structure that as we will show, drastically reduces the amount of CPU-GPU handshaking. TaskTable is a 2D array and has multiple columns and rows. Each TaskTable entry contains the following fields describing the task: (1) number of threadblocks, (2) number of threads in a threadblock, (3) task kernel pointer, (4) size in bytes of the shared memory allocation required per threadblock, (5) a flag indicating whether the task needs thread-block-level synchronization, (6) task inputs, (7) a *ready* field, and (8) a *sched* flag. Each TaskTable column corresponds to an MTB; The scheduler warp in that MTB schedules tasks in the column's entries onto the executor warps of that MTB. Having multiple rows in the TaskTable allows for high availability of tasks to schedule. Pagoda uses 32 TaskTable rows per MTB.

4.2.1 TaskTable Operation. When a task is launched via the Pagoda API (a call to *taskSpawn*), the tasks's parameters must be copied into an entry in the CPU TaskTable, then the entry must be copied to the GPU for scheduling. Since this copy has to occur while the MasterKernel is in flight, a *ready* field is necessary to indicate the finishing of the copy to the GPU. A straightforward way of implementing this, where the task's parameter data and the ready flag are copied in one *cudamemcopy* transaction, cannot work because the PCIe bus does not guarantee that the parameters will arrive in the GPU memory before the ready flag. One solution would be to simply split it into

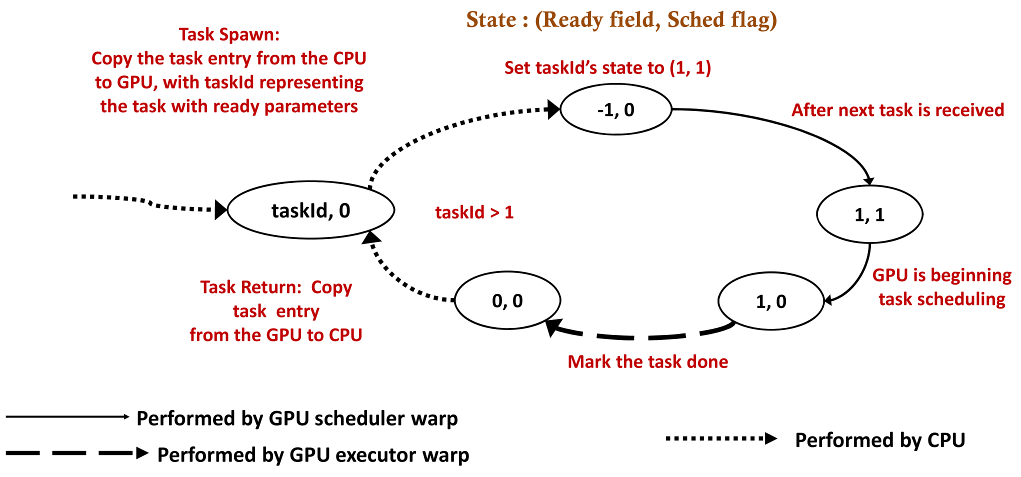

State : (Ready field, Sched flag)
Task Spawn: Copy the task entry from the CPU to GPU, with taskId representing the task with ready parameters
Set taskId's state to (1, 1)
-1, 0
After next task is received
taskId, 0
taskId > 1
1, 1
GPU is beginning task scheduling
Task Return: Copy task entry from the GPU to CPU
0, 0
1, 0
Mark the task done
Performed by GPU scheduler warp
Performed by GPU executor warp
Performed by CPU

Fig. 2. TaskTable State Diagram. The CPU only touches TaskTable entries with reset *ready* fields, when a task gets scheduled to warps, and the GPU only touches TaskTable entries with non-zero *ready* fields, allowing for simultaneous TaskTable updates from the CPU and GPU. The *sched* flag determines when the task gets scheduled on GPU warps.

two *cudamemcopy* transactions, one for the parameters, and another for the ready flag. However, this doubles the parameter copying overhead, significantly reducing Pagoda performance. To solve this issue, we pipeline the launching of tasks. The launch of a task prompts a copy of its parameters to the GPU, as well as a pointer indicating which task had its parameters copied in the previous *cudamemcopy* transaction. In the steady-state, we achieve 1 *cudamemcopy* per task table entry and the CUDA streams API guarantees that the parameters are copied before the task is scheduled.

The task arrival rate in a real system can be hard to predict and we do not want a task to wait indefinitely after its parameters are copied. To alleviate this issue, Pagoda sends out an empty update to the device when the interval between two tasks arriving crosses a threshold.

4.2.2 Task Spawning Example. Each task's state comprises its *ready* field and *sched* flag. The *ready* field of each TaskTable entry can be in one of four states: 0 meaning the task is not ready, -1 meaning the task's parameters have been copied to the task table, 1 meaning the task is being considered for scheduling on the GPU, or it can be a taskID which is an integer > 1. The taskID provides the necessary indirection to implement the pipelining, indicating which task has already had its parameters copied to the GPU. The *sched* flag has two states: 1, indicating that the task is ready to begin scheduling on an MTB, and 0 meaning otherwise. Figure 2 presents a task's state diagram.

Figure 3 presents an example execution of task A (TA). When the *taskSpawn* function is executed by the CPU, Pagoda finds a TaskTable entry with a reset *ready* field and copies the task's parameters into the entry. Since TA is the first task, the CPU sets the *ready* field to -1. For all subsequent tasks, it sets the ready field to the taskId of the last spawned task, e.g., during task B (TB) spawn, TA is set as the *ready* field. The taskIds generated by Pagoda are references to entries in the TaskTable. Next, the CPU resets the *sched* flag, and copies the entry to the GPU. If the *ready* field is a taskID, i.e., > 1, the continually polling scheduler warp for the TaskTable column (S2) sets the state of the previous task (TA) to (1, 1). The ready flag reserves taskID, which are brought from another task located in different MTBs. Both tasks will access the same taskID stored in the device memory in a serial manner. Next, S2 sets the state of the current task to (-1, 0) (see Algorithm 1, lines 5-13). S2

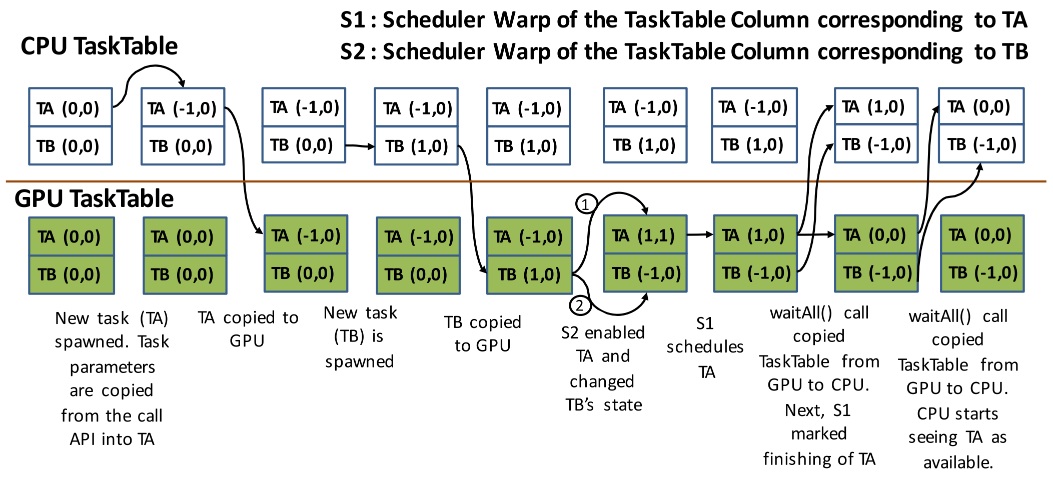

S1 : Scheduler Warp of the TaskTable Column corresponding to TA
S2 : Scheduler Warp of the TaskTable Column corresponding to TB
CPU TaskTable
GPU TaskTable
TA (0,0)
TB (0,0)
TA (-1,0)
TB (1,0)
TA (1,1)
TB (-1,0)
TA (1,0)
1
2
New task (TA) spawned. Task parameters are copied from the call API into TA
TA copied to GPU
New task (TB) is spawned
TB copied to GPU
S2 enabled TA and changed TB's state
S1 schedules TA
waitAll() call copied TaskTable from GPU to CPU. Next, S1 marked finishing of TA
waitAll() call copied TaskTable from GPU to CPU. CPU starts seeing TA as available.

Fig. 3. Example execution of task TA. TA gets scheduled only after TB is spawned. Our design allows for the CPU and GPU TaskTable entries to contain mis-matching values.

waits for the state of TA to be (−1, 0) before changing it to (1, 1). This is achieved through polling orchestrated with CUDA *threadfence* calls. Now, S1, the scheduler warp for TA, finds that TA has a set *sched* flag, and hence schedules TA. To do so, S1 first resets the *sched* flag and then finds executor warps for the task. Once the task execution is finished, the last finishing executor warp for the task sets the *ready* field to 0, marking the end of the task's execution, and freeing up the task entry TA. If the CPU spawner thread observes no new tasks come in, it copies back the status of the last task, i.e., TB, and if it is (−1, 0), then sets it to (1, 1) and copies it to the GPU, ensuring the successful execution of the last task.

Lazy Aggregate TaskTable Updates. The above mechanism allows both the CPU and GPU to simultaneously update the TaskTable. As the CPU only spawns a task if the *ready* field is reset, the CPU can keep spawning as long as it finds an entry with a reset *ready* field. Similarly, since the GPU only edits TaskTable entries with non-zero *ready* fields, it can keep scheduling as long as it finds a task entry with a set *sched* flag. When the *ready* fields of all CPU-side TaskTable entries are non-zero, the CPU can no longer spawn tasks. In that case, it copies back all the TaskTable entries from the GPU, thereby updating all *ready* fields. It can then realize which tasks have finished, and launch new tasks in entries with reset *ready* fields. The CPU therefore receives updates from GPU TaskTable in a *lazy aggregate* manner.

This laziness greatly reduces the number of handshaking communication calls. Furthermore, aggregated (bulk) copying achieves better data transfer bandwidth on the PCIe bus. The *wait* and *waitAll* functions return only when the *ready* field(s) of the corresponding task(s) in the TaskTable is/are reset. The laziness of TaskTable updates may block these functions if the CPU is not spawning more tasks; these functions therefore use a timeout, after which they enforce a copy-back of the involved TaskTable entries.

Because the CPU overwrites the TaskTable while the MasterKernel is in flight, coherence issues may arise. We therefore marked the TaskTable as `volatile`, and validated that *cudaMemCopy()* transfers from the CPU to the GPU update the GPU L2 cache, which is shared between all SMs. Data written to global memory by the GPU is also visible to the CPU. The GPU L1s are write-through and *cudaMemCopy()* calls that transfer data from the GPU to the CPU read from the GPU's L2. This behavior has also observed by other work [13, 32] that requires updates to GPU memory from the

Table 2. WarpTable Entry Fields

warpId	maintains the warp ID of the warp, for the current task. It is used to generate the threadID in the *getTid()* function.
eNum	refers to the task entry in the TaskTable, which is being executed by the warp. This reference allows each warp to obtain the task kernel arguments.
SMindex	indicates the shared memory starting location for the corresponding threadblock.
barId	maintains the barrier ID that the warp should synchronize on. It is only valid for tasks that request threadblock synchronization.
exec	acts as a flag for the warp to begin task execution. It is also used to query the warp status.

CPU, while kernels are in flight. The host side calls *cudaStreamSynchronize()* to ensure the CPU and GPU are in sync after the host copies back the GPU task table.

4.3 Concurrent Task Scheduling

Task scheduling in Pagoda involves finding free resources (warps, shared memory) on which to execute a task. All warps of a given task execute in the same MTB. This design stems from the fact that narrow tasks need less threads than those available in an MTB.

We found that task spawning and scheduling are high-overhead operations, especially for narrow tasks, which can be short running. To mitigate this issue, Pagoda overlaps the three task processing stages, namely, spawning, scheduling, and execution. Second, multiple scheduler warps across different MTBs schedule tasks concurrently, lowering the time to execution for each task.

Two Pagoda data structures facilitate task spawning and scheduling in parallel: the multi-row TaskTable and the per-MTB WarpTable. While the CPU is spawning tasks on a TaskTable row, scheduler warp(s) on the GPU may schedule tasks from the remaining rows. The status of each executor warp is stored in a WarpTable entry, whose fields are described in Table 2.

Algorithm 1 describes the operation of the scheduler and executor warps. The scheduler warp (Lines 2–28) scans the corresponding column in the TaskTable, and when it finds an entry with a set *sched* flag (Line 14), it attempts to schedule the task. It begins by resetting the *sched* flag. If the task requires shared memory or synchronization, then the scheduler first allocates shared memory/barrier (Lines 19–24) for them (Section 5.1) and performs scheduling for each individual threadblock of the task. If neither shared memory nor synchronization are required, then execution is based solely on available warp slots (Line 28).

The executor warps remain idle until the *exec* flag in their WarpTable slot is set. Once this flag is set, they execute the task (Line 33). Afterwards, they release the shared memory and the synchronization barriers (Lines 36–39). Last, they reset the *ready* flag in the corresponding TaskTable entry, and reset the *exec* flag in the WarpTable element, marking the warp to be free (Lines 41–43). In order for this mechanism to work, the scheduler and executor warps must have a consistent view of the WarpTable, which is achieved by the *threadfences*. The scheduler warp does not explicitly monitor the end of a task execution. Hence, it cannot free the shared memory used by the task's threadblocks immediately after they finish execution. The executor warps cannot themselves deallocate the shared memory, since it may lead to inconsistencies if the scheduler warp is simultaneously allocating the shared memory. To overcome this issue, the last executing warp of each threadblock requesting shared memory marks the shared memory region to be freed, and before performing any future shared memory allocation, the scheduler warp first deallocates all memory blocks marked for freeing (Line 22). The allocation/deallocation mechanism is described in Section 5.1.

ALGORITHM 1: Pagoda Task Scheduling : Each MTB Executes this Algorithm

```
Input: gTaskPool - column of task entries in the TaskTable belonging to the given MTB,
       numEntriesPerPool - #rows in the TaskTable,ctr[numEntriesPerPool] and
       doneCtr[numEntriesPerPool] - counters allocated in shared memory, tid - threadID
while (1) do
    if $tid < warpSize$ then // scheduler warp does this
        for (i = 0; i < numEntriesPerPool; i++) do
            entry $\leftarrow$ gTaskPool[i]
            taskId = entry.ready
            if $taskId > 0$ then
                prevEntry $\leftarrow$ taskTable entry for taskId
                if $prevEntry.ready \neq -1$ then
                    threadfence()
                    continue
                else
                    prevEntry.ready $\leftarrow$ 1
                    prevEntry.sched $\leftarrow$ 1
            if entry.sched then // check if sched flag is set
                entry.sched $\leftarrow$ 0
                doneCtr[i] $\leftarrow$ ctr[i] $\leftarrow$ getNumWarps(entry)
                if $entry.SMSize > 0 \vee entry.sync$ then // schedule warps per threadblock
                    for (j = 0; j < entry.numTB; j++) do
                        if (entry.sync) then  barId $\leftarrow$ getBarId()
                        if $(entry.SMSize > 0)$ then
                            do
                                deallocMarkedSM() // avoids deadlocking
                                retVal $\leftarrow$ allocSM(entry.SMSize, &index)
                            while (retVal == false)
                        ctr[i]$\leftarrow$ getNumWarpsPerTB(entry)
                        pSched(ctr[i]$\times$j, i, index, barId, &ctr[i])
                else // schedule all warps
                    pSched(0, i, 0, 0, &ctr[i])
    else // executor warps do this
        if (warpTable[warpId].exec) then
            entryId $\leftarrow$ warpTable[warpId].eNum
            tEntry $\leftarrow$ gTaskPool[entryId]
            *(tEntry.funcPtr))(tEntry.args) // warp executes the task
            if (laneId == 0) then
                if lastWarpInBlock() then // only 1 thread per threadblock performs this
                    if $(tEntry.SMSize > 0)$ then // dealloc SM
                        markSMForDealloc(warpTable[warpId].SMindex)
                    if (tEntry.sync) then
                        releaseBarId[tEntry.barId]
                threadfence_block()
                if (atomicDec(&doneCtr[entryId]))) then
                    tEntry.ready $\leftarrow$ 0; // free the task entry
                warpTable[warpId].exec $\leftarrow$ 0 // warp is free now
```

ALGORITHM 2: Parallel Warp Schedule Function

```
Input: tid - thread number, baseWarpId - base warp number getting scheduled, eNum - number of the
       TaskTable column entry, index - starting address of the shared memory for the threadblock,
       barId - barrier Id for the threadblock, warpCtr - count of the number of warps to be scheduled
Function pSched (baseWarpId, eNum, index, barId, warpCtr)
    threadDone ← 1 // private per thread
    i ← tid; // private per thread
    while (1) do
        if (i < numEntriesPerPool) then
            threadDone ← 0
            if (!warpTable[tid].exec) then
                if (id ← (atomicDec(warpCtr)) >= 0) then
                    warpTable[i].warpId ← id + baseWarpId
                    warpTable[i].eNum ← eNum
                    warpTable[i].SMindex ← index
                    warpTable[i].barId ← barId
                    threadfence_block()
                    warpTable[i].exec ← 1
        if (*warpCtr <= 0) then threadDone ← 1
        if (__all(threadDone == 1) == true) then // Synchronize threads in the scheduler warp
            break
        i ← i + 32
        if (i > numEntriesPerPool) then i ← tid
```

Scheduling is performed by the threads of the scheduler warp in parallel, through the *PSched* function (Algorithm 2). Note that the scheduler warps across different MTBs operate concurrently. The threads in the scheduler warp find free executor warps for a given task by checking their *exec* flags. If a free warp is found, a counter holding the number of warps that are yet to be scheduled is decremented atomically. If the result is positive, then the corresponding warp is scheduled (Lines 7–14). This counter resides in the GPU shared memory, speeding up the atomic operations. Note that both branches on lines 7 and 8 are divergent, i.e., different threads may have different branch outcomes. The threads with a false branch outcome may repeatedly execute the outer *while* loop, in spite of the other threads finding free warps. To remedy this problem, all threads in the scheduler warp must be synchronized after each iteration of the *while* loop. We achieve this using *__all()*, a CUDA warp-level vote function, as opposed to the usual CUDA API for synchronization, *__syncthreads()* which will synchronizes all MTB threads. At last, there is a +32 in at the end of the loop because there are 32-threads in the scheduler warp and on the next iteration of the while(1) loop, the next 32 warps from the task are evaluated.

5 SUPPORTING NATIVE CUDA FUNCTIONALITY

As tasks in Pagoda are launched by the MasterKernel, native CUDA shared memory and synchronization management cannot be used. This section describes how Pagoda supports these functionalities.

5.1 Shared Memory Management

CUDA lacks support for software-driven dynamic shared memory allocation once a kernel has been launched. Tasks in Pagoda piggyback on the MasterKernel, and hence cannot directly use the shared memory. A need therefore arises for software management of the shared memory. Each

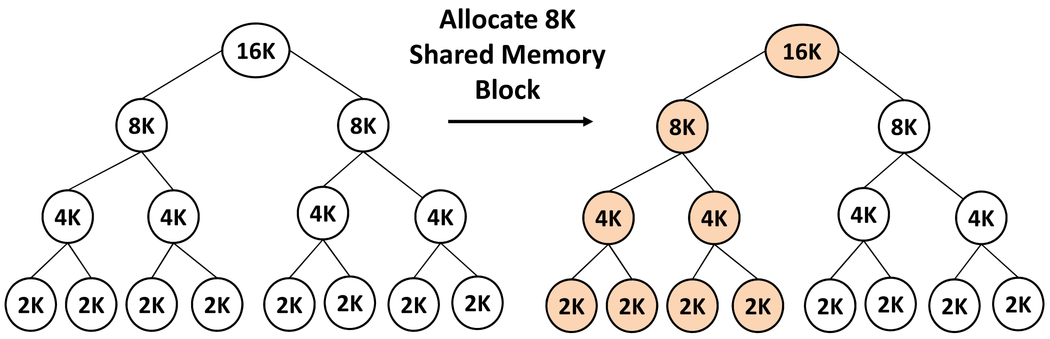


Fig. 4. Allocating 8K of shared memory in Pagoda. The value in each node represents the size of the shared memory block. Note that not all levels of the tree are shown here. The white nodes are free blocks and the shaded nodes are allocated blocks.

MTB reserves shared memory when it starts execution, and allocates this memory to threadblocks of one or more tasks, and frees it after the tasks finish execution.

Our software allocator/deallocator manages small, contiguous regions of shared memory with low overhead. Unlike many general-purpose allocators that rely on freelists [40], Pagoda's algorithm is motivated by the buddy system mechanism [14] to reduce overhead. Threads of the scheduler warp in the MTB are responsible for performing the allocations and deallocations.

Data Structure. The memory blocks are represented as nodes in a tree, as shown in Figure 4. This tree is arranged as an array in the shared memory itself, allowing fast access. Each level in the tree corresponds to memory blocks of a given size. The lowest node in the tree represents 512 bytes of memory, which is the smallest allocation granularity in our mechanism. The parent of a given node represents a memory block twice as large. Thus, the total number of nodes in the tree is 128, small enough to fit in the shared memory. A marked node means the block is allocated, otherwise, it is free. An invariant of this data structure is that if a node is marked, then its parent must be marked as well.

Allocation. Figure 4 shows a case where a completely free tree receives an 8K allocation request. The first step is to find the tree level at which node sizes are no smaller than the request. The static mapping of blocks allows our mechanism to search for a free node on such a level of the tree, an operation which is performed in parallel by the threads of the scheduler warp. One of these threads that finds such a free node marks it. The next operation is to mark all descendants and ancestors of this node. Since the tree contains only 128 nodes, threads of the scheduler warp each check four nodes, and mark them if they are either the descendants or ancestors of the allocated node.

Deallocation. Figure 5 shows an example where a block of 4K needs to be freed. First, the threads of the scheduler warp work in parallel to unmark all descendants of this node. Next, the first thread of the scheduler warp unmarks the node itself, and keeps going up the tree unmarking the parent as long as the sibling node is unmarked as well. Recall that both allocation and deallocation are carried out only by the scheduler warp, and hence no locking is necessary while performing them.

5.2 Sub-Thread Block Synchronization

CUDA __*syncthreads()* synchronizes threads within a threadblock. If this function is used directly within the Pagoda kernel code, the synchronization may lead to undefined behavior. This would

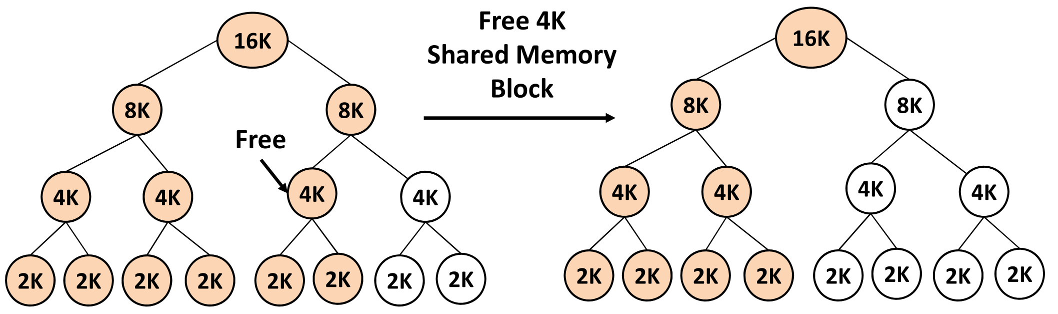


Fig. 5. Deallocating 4K of shared memory in Pagoda. Ancestors of the current node are marked free only if the sibling is free.

occur because the MTB may be running two different threadblocks simultaneously, and hence all threads in the MTB may not reach the *__syncthreads()* barrier.

A naive solution to this issue would force all threadblocks running on the MTB to reach the same barrier. However, this would lead to excessive wait times in threadblocks that do not require synchronization. Pagoda presents a sub-threadblock barrier, where only the threads of a given threadblock can synchronize. Pagoda achieves this using named barriers (using *bar.sync* instruction) in the PTX programming model [24]. Each threadblock of a task that annotates the synchronization requirement in the TaskTable entry is provided a unique barrier ID during the scheduling of the threadblock (Algorithm 1, Line 19). When a threadblock encounters the *syncBlock()* function, this barrier ID is used for synchronization. The PTX model allows for only 16 such barriers. The Pagoda design therefore needs to recycle these IDs once the threadblock is finished. We assume no kernels having less than 64 threads in one narrow task. Hence, 16 named barriers are enough for most narrow task applications.

6 EVALUATION

The following subsections detail our experimental setup and results.

6.1 Experimental Setup

The GPU experiments are run on a node with an NVIDIA Pascal Titan X GPU, which contains 3584 1417MHz GPU cores with 12GB RAM. The machine runs Ubuntu 16.04, with 24GB RAM and an Intel Core-i7 4.0GHz quad-core CPU. All experiments were run on NVIDIA Pascal Titan X GPU except Section 6.8. We enabled 32 concurrent kernels in the HyperQ by setting the CUDA_DEVICE_MAX_CONNECTIONS environment variable to 32. All CUDA and Pagoda benchmarks are compiled using *nvcc* from CUDA 9.0, with the *-O3* option. The MasterKernel, along with all task kernels, are forced to use at most 32 registers in the Pagoda versions. The PThreads and sequential programs are compiled with *gcc -O3* and are executed on two hyperthreaded Intel Xeon E5-2660 v3 CPUs each having 10 cores running at 2.6GHz.

Table 3 details the applications used in this study. We chose benchmarks from various application domains, such as signal and image processing, network security, and scientific computing where narrow tasks arise often. Table 4 shows the workload characteristics of the benchmarks.

6.2 Runtime Performance

Figure 6 compares the performance of narrow task applications on different CPU (PThread), and GPU runtime systems (CUDA-HyperQ, GeMTC, and Pagoda). Pagoda achieves geometric mean speedup of 1.76X over CUDA HyperQ programs, 1.44X over GeMTC, and 5.52X over the pThread

Table 3. Benchmark Description

MB	Mandelbrot sets are used in fractal analysis [7]. Each pixel value of the image is calculated in parallel; however, the required computation per pixel is highly irregular. Therefore, computation over each pixel is represented as a task that has low degree of parallelism.
MM	This is a standard matrix multiplication implementation, refactored from the NVIDIA SDK samples [23]. We used small matrix sizes, with each multiplication running as a task to simulate the behaviour seen in an earthquake engineering simulator [17]. The behavior arises from concurrent simulation of various structures, each of which is represented by different but small matrix sizes.
FB	Filterbank is a signal processing algorithm that separates input signals into multiple sub-signals with a set of filters. Multiple radios generate signals, processing each of them represents a task. Each task contains small amount of parallelizable computation.
BF	Beam former is a signal processing method used to control the direction of signal reception and transmission. Many independent signal beams receive inputs asynchronously. Processing individual inputs generate a narrow task.
SLUD	This is a sparse matrix solver using multi-frontal method [16]. A matrix is divided into multiple regular sub-matrices. Sparse LUD is represented as a task-based application owing to the irregularity in the computation size among different iterations of a parallel loop.
3DES	It is used to encrypt electronic data [6]. Network routers encrypt multiple packets as they arrive, each of which is represented as a narrow task. We use NetBench [18] to generate varied sizes of network packets that 3DES encrypts.
DCT	The Discrete Cosine Transform (DCT) [22] is commonly used for compression, e.g., JPEG (image), MP3 (audio), and MPEG (video) use it. Online surveillance systems gather image streams from multiple cameras, and operate on images from different streams in parallel [11]. Processing each image represents a narrow task.
CONV	Convolution filters [20] are used in blur and edge detection mechanisms in image processing. Each filter operation represents a task, which operates in parallel across pixels.
MPE	Pagoda is able to run multi-programmed workloads, where multiple applications generate narrow tasks asynchronously. To evaluate such a setup, we built a multi-programmed benchmark of our own. Multi-programmed environments often encounter heterogeneity in workloads. To simulate that, we chose (1) 3DES and Mandelbrot, which contain irregular computations, (2) Filterbank, which requires threadblock-level synchronization, and (3) Matrix multiplication, which uses shared memory. Each of the benchmarks contained 8K tasks, totalling 32K tasks.

benchmarks. The performance metric in Figure 6 is calculated over the entire execution time, including the time of both compute and data copy in the GPU benchmarks. We injected 32K narrow tasks in each application in Figure 6, except SLUD (273K). Each task is composed by 128 parallel threads in the GPU benchmarks. The small number of narrow tasks does not fully utilize GPU resources. Increasing the concurrent task counts is helpful for the computational throughtput of narrow tasks. Therefore, the key reason why Pagoda can outperform other runtime systems is the high GPU utilization. GeMTC increases the GPU utilization by launching work in batches and tasks are run within its SuperKernel. We could not implement SLUD in GeMTC; GeMTC needs the

Table 4. Benchmark Characteristics

Benchmark	Source	Task Type	Input Set per Task(each task is one image, signal, matrix or network packet)	Num Tasks	% Time spent in data copy (CUDA-HyperQ)	% Time spent in computation (CUDA-HyperQ)	May benefit from Shared Memory	Requires threadblock synchronization	Default Register Count
Mandelbrot (**MB**)	Quinn [28]	Irregular	64 × 64 images	32K	24	76	✗	✗	28
MatrixMul (**MM**)	CUDA SDK [23]	Regular	64 × 64 matrix	32K	51	49	✓	✓	30
FilterBank (**FB**)	StreamIt [36]	Regular	Signals of width 2K	32K	35	65	✗	✓	21
BeamFormer (**BF**)	StreamIt [36]	Regular	Signals of width 2K	32K	13	87	✗	✗	34
Sparse LU Decomposition (**SLUD**)	OpenMP Task Suite [5]	Irregular	32 × 32 matrix	273K	3	97	✗	✗	17
3DES	NIST [6]	Irregular	Network packets sized 2K-64K	32K	74	26	✗	✗	26
DCT8x8 (**DCT**)	CUDA SDK [22]	Regular	128 × 128 images	32K	81	19	✓	✓	33
Image Convolution (**CONV**)	CUDA SDK [20]	Regular	128 × 128 images	32K	30	70	✗	✗	25

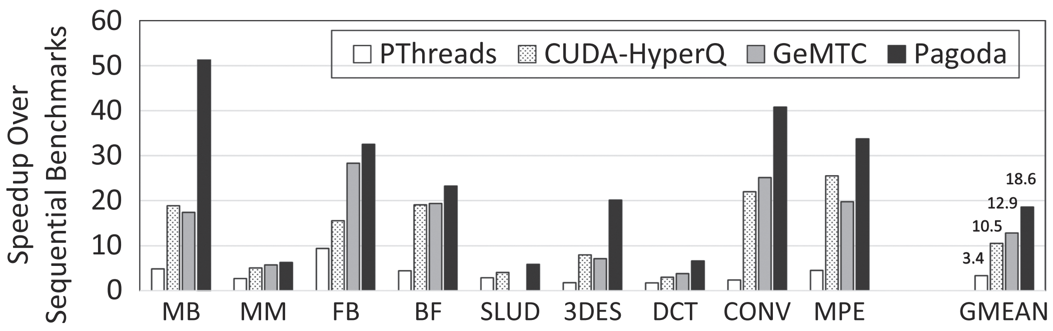


Fig. 6. Overall Performance Comparison. All applications of this experiment were run on NVIDIA Pascal GPU. Speedup is normalized to the sequential CPU applications. The number of tasks in each benchmark is constant (32K), except SLUD, which contains 273K tasks. Each GPU task uses 128 threads. The measurement of execution time contains both data copy and compute times. Pagoda significantly outperforms CUDA-HyperQ (1.76x), 20-core PThreads (5.52x), and GeMTC (1.44x) because of the high GPU utilization.

number of tasks to be pre-defined, which is not the case in SLUD. Both GeMTC and Pagoda can reach 100% GPU occupancy. However, the average performance of GeMTC is 18% less than Pagoda. The reason is due to the complex task queuing and lock-step communication of GeMTC. GeMTC performs worse than CUDA-HyperQ in MB, MPE, and 3DES becuase these applications contain irregular workloads. For a fair comparison with a CPU execution, we have tried to make use of Python-based thread pooling, OpenMP for data parallelism, pThreads-based task parallelism. We found the pThreads implementation obtained the best results, which are included in Figure 6.

6.3 Pagoda Performance Scalability

Pagoda is able to run on different types of NVIDIA GPUs after changing the number of SMs in its system configuration. Figure 7 presents the performance results of Pagoda on NVIDIA Maxwell

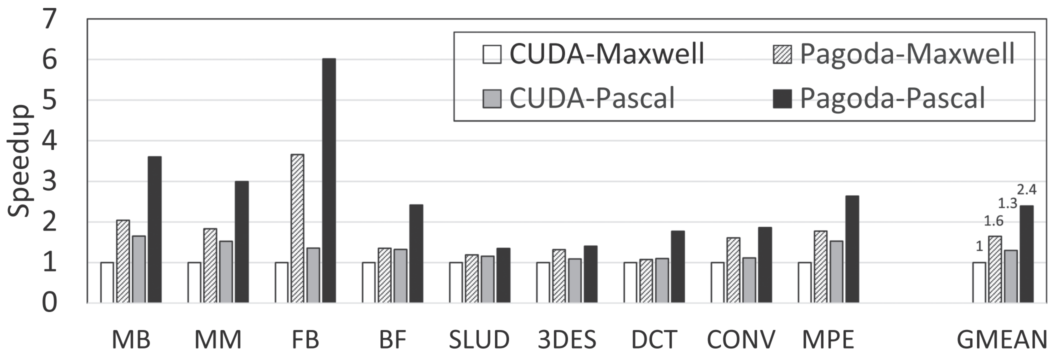


Fig. 7. Pagoda Performance Scalability. CUDA-Maxwell and CUDA-Pascal indicates CUDA-HyperQ applications are run on NVIDIA Maxwell and Pascal Titan X GPU. Pagoda achieves 2.4X speedup compared to CUDA-Maxwell by running benchmarks on NVIDIA Pascal GPU.

and Pascal Titan X GPU. This section aims to figure out the performance scalability of Pagoda from NVIDIA Maxwell to Pascal GPU. Each application in Figure 7 includes 32K tasks, 128 threads in one task and the execution time calcuation does not count the time of data copy in. The number of cores on NVIDIA Pascal Titan X GPU is 16% higher than on the Maxwell architecture. Additionally, its compute frequency is 53% higher than on the Maxwell GPU. The average performance speedup of Pagoda-Pascal is 2.39X and Pagoda-Maxwell achieves a 1.64X speedup compared to CUDA-Maxwell in Figure 7. The increased number of compute cores and speed on Pascal GPU results in this performance improvement. CUDA HyperQ programs attach tasks in different CUDA streams and these tasks are executed when there are available GPU resources. However, the limited number of task queues on the GPU constrains the degree of task concurrency. As a result, most of the performance benefit of the CUDA-Pascal applications only comes from the higher GPU speed and is 30% faster than on the Maxwell GPU.

6.4 Sensivity Analysis for Task Load Imbalance

Task load imbalance impacts the performance and response time of the individual tasks. This section presents the performance results of tasks composed of various computations in static task fusion, CUDA-HyperQ and Pagoda. Static task fusion combines multiple tasks into a monolithic large task [9, 38]. This method is good for tasks consisting of the same computational work, since the task fusion method can decrease the CPU-GPU communication overhead and increases the degree of parallelism within one fused task (kernel). However, this regular workload is not always seen in the real world. In this experiment, we created tasks comprising various input sizes and thread counts by using a pseudo-random generator. Each application contains 32K tasks. The threadblock size was fixed in the fused kernel, and the threadblock size is 256 in this experiment. We chose this number heuristically, since selecting the best thread count per task is infeasible in static fusion. The threadblock size in some sub-tasks in a fused kernel are smaller than 256, and this waste is unavoidable. The SLUD application cannot be fused because the number of tasks is not known statically.

Figure 8 demonstrates the speedup of static task fusion and Pagoda compared to CUDA-HyperQ. Pagoda gains 1.8X speedup and the static task fusion method is about 10% slower than CUDA-HyperQ applications. There are two reasons for the slowdown shown in the static task fusion benchmarks. First, the longest task in a fused kernel dominates the execution time. This situation is often shown in compute-intensive applications such as MB, FB and CONV in Figure 8. Second,

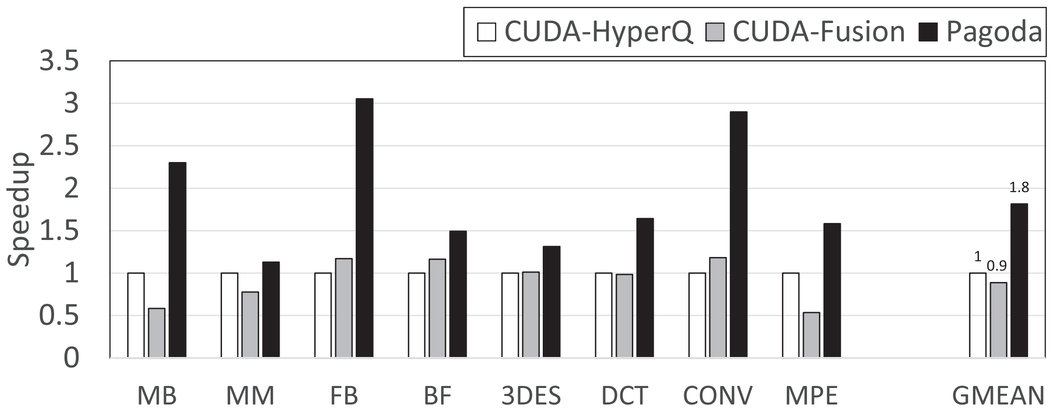


Fig. 8. Performance Comparison of Static Fusion, CUDA-HyperQ and Pagoda with irregular tasks. Dynamic task spawning mechanism in Pagoda obtains high performance even with irregular workloads.

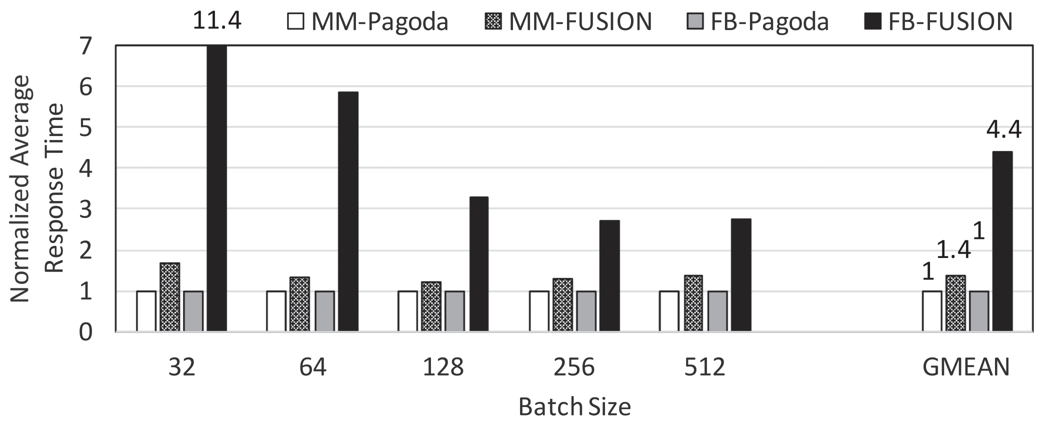


Fig. 9. Average Latency of Tasks. Pagoda achieves much lower latency compared to static fusion.

the underutilization is shown in a fused kernel because the threadblock size of each sub-task can be different. In contrast to the static fusion method, Pagoda launches tasks in quick succession without combining tasks in a batch. Additionally, Pagoda follows the availability of the GPU hardware resources to allocate tasks. Pagoda's task allocation mechanism can fully utilize the GPU and satisfy dynamic task workloads.

6.5 Task Latency Analysis

Figure 9 compares the normalized average response time between Pagoda and static fusion applications. The average response time of static fusion benchmarks is 40% and 440% longer than Pagoda shown in Figure 9. The batch size in Figure 9 means the number of threadblocks in one fused kernels. The average response time measures the total task execution time over the number of injected tasks. Each benchmark in this experiment contains 16K irregular tasks with various threadlblocks and input sizes. Each task in a statically fused kernel and in a batch-based system such as GeMTC, is completed until all tasks in the fused kernel or batch have done the work. Thus, their response time increases with the the number of tasks per batch.

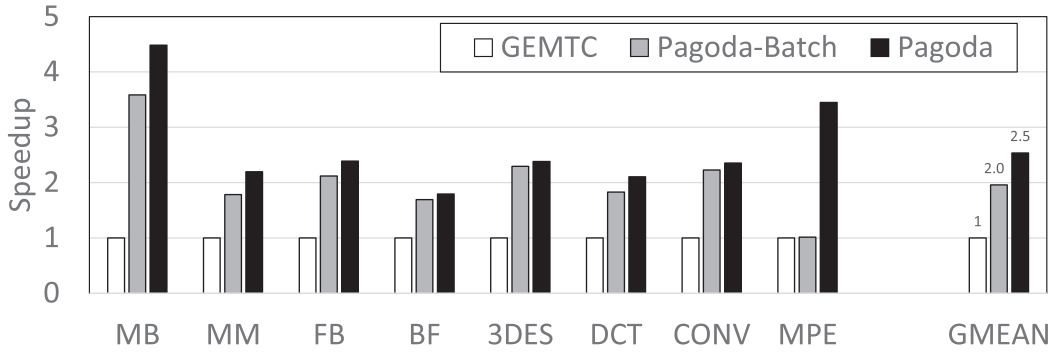


Fig. 10. Benefit of Pagoda Continuous Spawning and Concurrent, Pipelined Task Processing. Pagoda performs both continuous task spawning and concurrent, pipelined task processing. Pagoda-batching only performs task processing. GeMTC performs neither. Pagoda outperms GeMTC in all cases.

In Figure 9, the average response time of BF-Fusion decreases with growing batch size. The reason is that the large batch size increases the degree of parallelism in the fused kernel. The parallelism helps the performance of the fused kernel and response time. However, this growth trend does not always increase linearly and slows down when the batch size is over 256 in FB-Fusion in Figure 9. Threadblocks specified in the fused kernel can be over the pre-defined threadblock counts on GPUs. Over-subscribing threadblocks in the fused kernel increases the contention of resources and slows down the performance speedup. Instead of batching tasks to increase parallelism, Pagoda launches tasks successively on the GPU. Pagoda allocates tasks to the GPU dynamically based on the availability of hardware resources and decreases the resource contention.

6.6 Lock-Step Communication Overhead

To understand the benefit of continuous task spawning, this experiment creates a Pagoda version that spawns tasks in batches. This batch Pagoda version does not spawn tasks until all tasks in the previous batch are done. Figure 10 compares the performance of Pagoda, Pagoda-batch, and GeMTC. Each application in Figure 10 contains 32K tasks and their threadblock size is 128. In Figure 10, Pagoda gains 2.53X speedup compared to GeMTC. This performance improvement comes from the concurrent task scheduling in Pagoda. GeMTC's complicated task queue method hindered the task concurrency within its batch. In addition, on average, the Pagoda-batch implementation incurs 29% overhead because of its lock-step task launch mechanism. Pagoda overlaps the task spawning and execution to achieve this speedup compared to the Pagoda-batch alternative. In Figure 10, CONV only gets 5% performance benefit from continuous task spawning because its regular, extremely short running task. However, MPE demonstrates the exceptionally high benefit in the presence of unbalanced tasks.

6.7 Pagoda Task Scheduling Overlead Analysis

Figure 11 shows the overhead of Pagoda task scheduling for various threadblock and input size. In Figure 11, the threadblock size of the HyperQ applications is 256, and both benchmarks contain 32K tasks. In the CUDA-HyperQ programs, the GPU hardware distributes threadblocks of one task (kernel) to other SMs. However, Pagoda only uses 31 executor warps from the MasterKernel ThreadBlock (MTB) to feed the request of one task. Hence, some warps in one task must wait for executor warps to finish. This case occurs in the big task comprising a large number of threadblocks in Pagoda. As shown in Figure 11, Pagoda obtains 3% runtime overhead in MM task as its input size

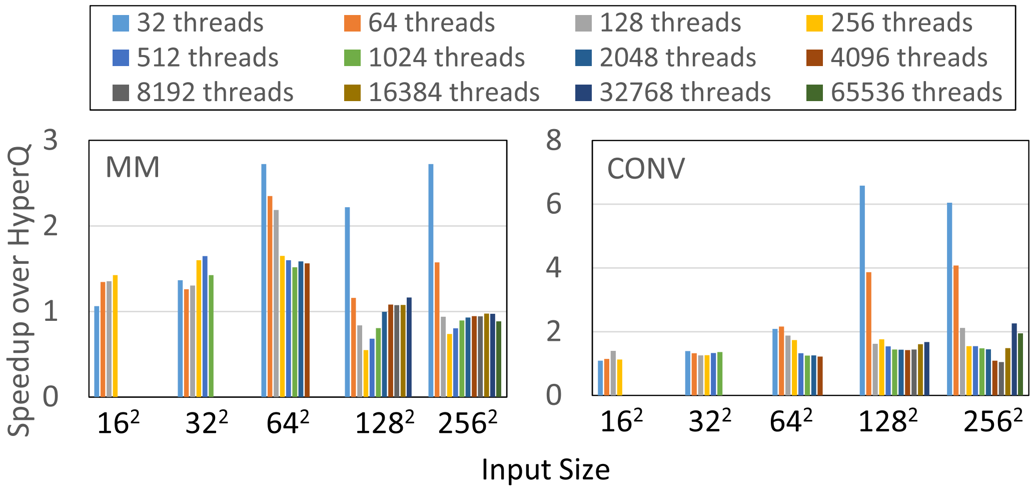


Fig. 11. Effects of varying threads per task for different input size.

is 256 × 256 and 32K threads. The speedup of HyperQ programs increases linearly with increasing of threadblock size from 32 to 512. The increase of parallelism in one task facilitates this speedup in HyperQ applications. However, the speedup of HyperQ does not improve beyond their threadblock size of 1024, as the GPU is fully utilized. On the other hand, the performance of Pagoda does not fluctuate with changing threadblock size, since Pagoda increases the number of concurrent tasks in small threadblock size configurations. Furthermore, 256 × 256 input size and 64K threads of CONV, the speedup of Pagoda improves again. We attribute this behavior to the warp-level scheduling in Pagoda versus the threadblock-level scheduling in CUDA. CUDA prevents a new threadblock from launching until all warps of the previous threadblock finish, where as Pagoda can schedule a warp from a new threadblock as soon as another warp completes.

6.8 Effects of Varying Threads in Narrow Tasks

Figure 11 presents the speedup of Pagoda after varying the number of threads in narrow tasks. For small threads, Pagoda outperforms HyperQ in all input sizes. For large thread counts, Pagoda may still outperform HyerQ because its finer grain of scheduling.

6.9 Effects of Sub-Threadblock Synchronization

Pagoda demonstrates the impact of the sub-threadblock synchronization on the MM and DCT applications. Both applications use shared memory and sub-threadblock synchronization. Pagoda performs software management of the GPU shared memory, as described in Section 5.1. To compare the obtainable performance benefits from the use of shared memory, we show performance results on the DCT and MM benchmarks. These two codes can potentially benefit from the use of shared memory. We created two versions for each: with and without using shared memory. Table 5 compares the speedups achieved by these versions over the CUDA-HyperQ versions, which also use the shared memory. The shared memory requirement may reduce the achieved occupancy; yet, Pagoda shared memory versions achieve performance benefits. None of the other static-fusion or runtime batching solution offer shared memory utilization, and miss out on such benefits.

Table 5. Compute Performance Comparison of Tasks Run in Pagoda with and without Shared Memory Allocation

Benchmark	Pagoda with Shared Memory		Pagoda without Shared Memory	
	Speedup over HyperQ using Shared memory	Achieved Occupancy	Speedup over HyperQ using Shared memory	Achieved Occupancy
DCT	1.13x	25%	1.02x	97%
MM	1.47x	97%	1.32x	97%

Each version runs 32K tasks. DCT tasks have 64 threads, MM tasks contain 256 threads. Only the compute time is compared. The shared memory usage offers considerable benefits.

7 RELATED WORK

Task-based models [2, 4] employ a runtime system which governs task executions on various engines, such as CPUs and GPUs. These systems, however, always execute narrow tasks on CPUs, believing that their low parallelism degree cannot overcome the overhead of memory copies.

Static task fusion is the preliminary approach to deal with GPU underutilization. Wang et al. [38] present a mechanism where such fusion achieves higher utilization, resulting in energy benefits. KernelMerge [9] statically fuses kernels, and explores round-robin and fair-partitioned execution schemes for these kernels. The GPU programming models, such as CUDA and OpenCL, allocate same resources to each thread. Therefore, the resource usage in static fusion schemes gets limited by the requirements of the most resource-hungry task. A more sophisticated approach is therefore to perform fusion at the runtime. Two approaches [26, 29] perform kernel consolidation leveraging the GPU concurrent kernel executions. They launch multiple concurrent kernels, where resources not being used by one kernel can be yielded to another. The first approach [29] relies on a threadblock-level launching scheme. The second approach [26] presents a compiler scheme that transforms kernels so that they can automatically support any threadblock configuration. This ability helps in finding the best sharing configuration for different kernels. Zhong and He [42] present an approach where a large kernel is split into independent smaller kernels that co-execute to achieve better utilization. Kato et al. [12] propose a software scheduler at the device driver layer to prevent interference among concurrently running GPU applications, trading off response latency for throughput. However, all these approaches are restricted by the 32 kernel limit imposed by the CUDA-HyperQ, and fail to efficiently execute narrow tasks.

Runtime systems that virtualize GPU resources can naturally overcome the hardware-imposed kernel limit. Additionally, they offer low execution latencies compared to static fusion. Closest to our work is GeMTC [15]. Like the MasterKernel in Pagoda, GeMTC runs a SuperKernel that virtualizes GPU resources. The use of large dameon-like kernels is similar to persistent threading [10]. Unlike the MasterKernel, the SuperKernel does not guarantee an occupancy of 100%, and therefore may face underutilization. Second, the GeMTC design uses a single FIFO queue for its batch-based task launching scheme, resulting in significant task scheduling overhead. Third, GPU-specific functionalities, such as the shared memory and threadblock-level synchronization, remain unsupported.

GPU researchers have exploited pipelining [30] to overlap data transfers with kernel computations. The distinguishing factor in the Pagoda pipelined task processing is that it overlaps spawning, which comprises CPU finding a free task entry and performing a data copy, with GPU scheduling, which is only a sub-part of the overall task processing. Yang et al. [41] showed that fusing cross-kernel threadblocks can obtain better shared memory performance. By contrast, the Pagoda shared memory management schedules threadblocks as long as shared memory is found at runtime.

Prior research has explored preemptive hardware techniques to improve GPU utilization in the presence of concurrent low-occupancy kernels [27, 35, 39]. In contrast to these works, which require hardware changes, Pagoda provides a software only solution that runs on contemporary GPU hardware and could be applied to any future GPU hardware that supports the CUDA programming model.

Virtualizing GPU resources has also been explored to improve GPU utilization via multi-tenancy in cloud computing. Sengupta et al. [31] focus on virtualizing the GPU as a whole in a cloud with multiple GPUs. Becchi et al. [3] study a virtual memory system that isolates the memory spaces of concurrent kernels and allows kernels whose aggregate memory footprint exceeds the GPU's memory capacity to execute concurrently. By contrast, Pagoda virtualizes the compute resources of a single GPU at the granularity of a warp.

8 CONCLUSION

This article has presented Pagoda, a GPU runtime system that overcomes underutilization in the presence of narrow tasks. Pagoda virtualizes GPU resources via MasterKernel, a continually executing daemon on the GPU. Pagoda launches tasks on the GPU as long as some free warps are available. Unlike previous work, Pagoda supports most functionality of the native CUDA model. A key distinction in Pagoda is the task spawning and scheduling mechanism. It contains a novel data structure, called TaskTable, that greatly reduces CPU-GPU handshaking during task spawning. Pagoda achieves concurrent task scheduling, and overlaps task spawning, scheduling, and execution through pipelining. The experimental evaluation showed that Pagoda achieves a geometric mean speedup of 1.76x over CUDA- HyperQ, 1.44x over GeMTC, and 5.52x over 20-core CPU PThreads. The evaluation also showed that Pagoda can outperform static fusion schemes by 1.79x, and achieves much lower latency per task. We believe that the Pagoda design makes it easy to exploit GPUs for applications exhibiting narrow tasks, and will encourage porting of many non-traditional workloads to GPUs.

REFERENCES

[1] IEEE Standards Association. 2012. 802.3-2012-IEEE Standard for Ethernet. [Online]. Available: http://standards.ieee.org/findstds/standard/802.3-2012.html (accessed March. 5, 2018).

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.* 23, 2 (Feb. 2011), 187–198. DOI: https://doi.org/10.1002/cpe.1631

[3] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. 2012. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*. ACM, New York, 97–108. DOI: https://doi.org/10.1145/2287076.2287090

[4] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. 2012. Productive programming of GPU clusters with Ompss. In *Proceedings of the 2012 IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*. 557–568. DOI: https://doi.org/10.1109/IPDPS.2012.58

[5] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona Openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in Openmp. In *Proceedings of the 2009 International Conference on Parallel Processing (ICPP'09)*. IEEE Computer Society, Washington, DC, 124–131. DOI: https://doi.org/10.1109/ICPP.2009.64

[6] PUB FIPS. 1999. 46-3: Data encryption standard (DES). *National Institute of Standards and Technology* 25, 10 (1999), 1–22.

[7] Fraqtive. 2016. [Online]. Available: http://fraqtive.mimec.org/ (accessed January 5, 2019).

[8] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. 2008. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE Press, 2.

[9] Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron. 2012. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*. USENIX Association, Berkeley, CA, 10–10. http://dl.acm.org/citation.cfm?id=2342788.2342798

[10] Kunal Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of persistent threads style GPU programming for GPGPU workloads. In *Proceedings of the Symposium on Innovative Parallel Computing (InPar'12)*. IEEE, 1–14.

[11] A. S. Kaseb, E. Berry, Y. Koh, A. Mohan, W. Chen, H. Li, Y. H. Lu, and E. J. Delp. 2014. A system for large-scale analysis of distributed cameras. In *Proceedings of the 2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 340–344. DOI: https://doi.org/10.1109/GlobalSIP.2014.7032135

[12] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, 2–2. http://dl.acm.org/citation.cfm?id=2002181.2002183

[13] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. 2014. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 201–216. http://dl.acm.org/citation.cfm?id=2685048.2685065

[14] Kenneth C. Knowlton. 1965. A fast storage allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623–624. DOI: https://doi.org/10.1145/365628.365655

[15] Scott J. Krieder, Justin M. Wozniak, Timothy Armstrong, Michael Wilde, Daniel S. Katz, Benjamin Grimmer, Ian T. Foster, and Ioan Raicu. 2014. Design and evaluation of the GeMTC framework for GPU-enabled many-task computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC'14)*. ACM, New York, 153–164. DOI: https://doi.org/10.1145/2600212.2600228

[16] Joseph W. H. Liu. 1992. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Rev.* 34, 1 (March 1992), 82–109. DOI: https://doi.org/10.1137/1034004

[17] F. McKenna. 2011. OpenSees: A framework for earthquake engineering simulation. *Computing in Science Engineering* 13, 4 (July 2011), 58–66. DOI: https://doi.org/10.1109/MCSE.2011.66

[18] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. 2001. Netbench: A benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*. IEEE Press, 39–42.

[19] Adam Morrison and Yehuda Afek. 2013. Fast concurrent queues for x86 processors. *SIGPLAN Not.* 48, 8 (Feb. 2013), 103–112. DOI: https://doi.org/10.1145/2517327.2442527

[20] NVIDIA. 2007. Texture-based Separable Convolution. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/C/html_x64/Image_Processing.html. (accessed January 5, 2019).

[21] NVIDIA. 2012. Hyper-Q Example. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf (accessed January 5, 2019).

[22] NVIDIA. 2012. The White Paper of Discrete Cosine Transform for 8x8 Blocks with CUDA. [Online]. Available: http://www.math.uaa.alaska.edu/~ssiewert/a385_doc/dct8x8.pdf (accessed January 5, 2019).

[23] NVIDIA. 2015. CUDA. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (accessed January 5, 2019).

[24] NVIDIA. 2016. PTX. [Online]. Available: http://docs.nvidia.com/cuda/parallel-thread-execution/ (accessed January 5, 2019).

[25] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. 2013. The case for tiny tasks in compute clusters. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*. USENIX, Berkeley, CA. https://www.usenix.org/conference/hotos13/case-tiny-tasks-compute-clusters.

[26] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, 407–418. DOI: https://doi.org/10.1145/2451116.2451160

[27] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative preemption for multitasking on a shared GPU. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, 593–606. DOI: https://doi.org/10.1145/2694344.2694346

[28] Michael J. Quinn. 2003. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

[29] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat Chakradhar. 2011. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC'11)*. ACM, New York, 217–228. DOI: https://doi.org/10.1145/1996130.1996160

[30] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. 2013. Scaling large-data computations on Multi-GPU accelerators. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS'13)*. ACM, New York, 443–454. DOI: https://doi.org/10.1145/2464996.2465023

[31] Dipanjan Sengupta, Raghavendra Belapure, and Karsten Schwan. 2013. Multi-tenancy on GPGPU-based servers. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*. 3–10.

[32] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2013. GPUfs: Integrating a file system with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, 485–498. DOI:https://doi.org/10.1145/2451116.2451169

[33] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. 1993. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*. ACM, New York, 13–22. DOI:https://doi.org/10.1145/155332.155334

[34] Jaspal Subhlok and Gary Vondran. 2000. Optimal use of mixed task and data parallelism for pipelined computations. *J. Parallel Distrib. Comput.* 60, 3 (2000), 297–319. DOI:https://doi.org/10.1006/jpdc.1999.1596

[35] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*. IEEE Press, Piscataway, NJ, USA, 193–204. http://dl.acm.org/citation.cfm?id=2665671.2665702

[36] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *Compiler Construction*. Springer, 179–196.

[37] Vasily Volkov and James W. Demmel. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2008 (SC 2008)*. IEEE, 1–11.

[38] Guibin Wang, Yisong Lin, and Wei Yi. 2010. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *Green Computing and Communications (GreenCom), Proceedings of the 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing (CPSCom)*. 344–350. DOI:https://doi.org/10.1109/GreenCom-CPSCom.2010.102

[39] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. 2015. Simultaneous multikernel: Fine-grained sharing of GPGPUs. *IEEE Computer Architecture Letters* PP, 99 (2015), 1–1. DOI:https://doi.org/10.1109/LCA.2015.2477405

[40] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management (IWMM'95)*. Springer-Verlag, London, UK, 1–116. http://dl.acm.org/citation.cfm?id=645647.664690

[41] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. 2012. Shared memory multiplexing: A novel way to improve GPGPU throughput. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, 283–292. DOI:https://doi.org/10.1145/2370816.2370858

[42] Jianlong Zhong and Bingsheng He. 2014. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (June 2014), 1522–1532. DOI:https://doi.org/10.1109/TPDS.2013.257

Received March 2018; revised March 2019; accepted May 2019