Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks

Tsung Tai Yeh[†], Amit Sabne[‡*], Putt Sakdhnagool[†], Rudolf Eigenmann[†], Timothy G. Rogers[†]
School of Electrical and Computer Engineering, Purdue University[†]
Microsoft[‡]
yeh14@purdue.edu, amsabne@microsoft.com, {psakdhna, eigenman, timrogers}@purdue.edu

ABSTRACT

Massively multithreaded GPUs achieve high throughput by running thousands of threads in parallel. To fully utilize the hardware, 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 < 500 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 paper 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. Experimental results demonstrate that Pagoda achieves a geometric mean speedup of 5.70x over PThreads running on a 20-core CPU, 1.51x over CUDA-HyperQ, and 1.69x over GeMTC, the state-of-the-art runtime GPU task scheduling system.

CCS Concepts

•Software and its engineering → Runtime environments;

Keywords

GPU runtime system; utilization; task parallelism

1. INTRODUCTION

GPGPU computing has demonstrated an ability to accelerate a substantial class of compute-intensive applications [36, 7]. These applications have a high degree of parallelism, where iterations of large parallel loops are executed on the GPU. The programs see significant performance benefits because they can fully utilize the GPU's hardware resources by launching enough concurrent threads.

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 paper 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 (< 500 data parallel threads in practice).

Narrow tasks emerge in a number of scenarios. One set of such applications comprises latency-driven, real-time workloads. For example, online sensors that generate small inputs, resulting in tasks with low parallelism. Online sensors can generate many tasks in quick succession and require immediate processing. These workloads have been characterized as having mixed task and data parallelism [33, 32]. Secondly, *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 [24]. 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. This paper 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 [37, 8, 25, 14]. One approach to solve this problem is to statically fuse multiple smaller tasks [37, 8] to accumulate a large kernel. Advanced approaches [28, 25] 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 re-

*Author conducted this work while at Purdue University

PPoPP '17, February 04-08, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3018743.3018754

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

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 is over
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 are over
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

source 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 [20], which allows 32 tasks to concurrently execute on the GPU. However, we show 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) [14], 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 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 (SMM). 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 [18], is used. Previous work that required OS-like co-ordination over PCIe [12, 31] 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 enable the efficient execution of narrow tasks on GPUs. This paper

- 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.
- implements and evaluates the presented methods in a new runtime system, called Pagoda. Pagoda achieves geometric mean speedup of 5.70x over PThreads running on a 20-core CPU, of 1.51x over CUDA-HyperQ, and of 1.69x against GeMTC.

2. GPU PROGRAMMING AND ARCHITECTURE

The GPU cores are organized into 24 Streaming Multiprocessors (SMMs) [2]. Each SMM 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 SMM has a 96KB on-chip programmer managed cache, known as *shared memory* and 64K, 32-bit registers.

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

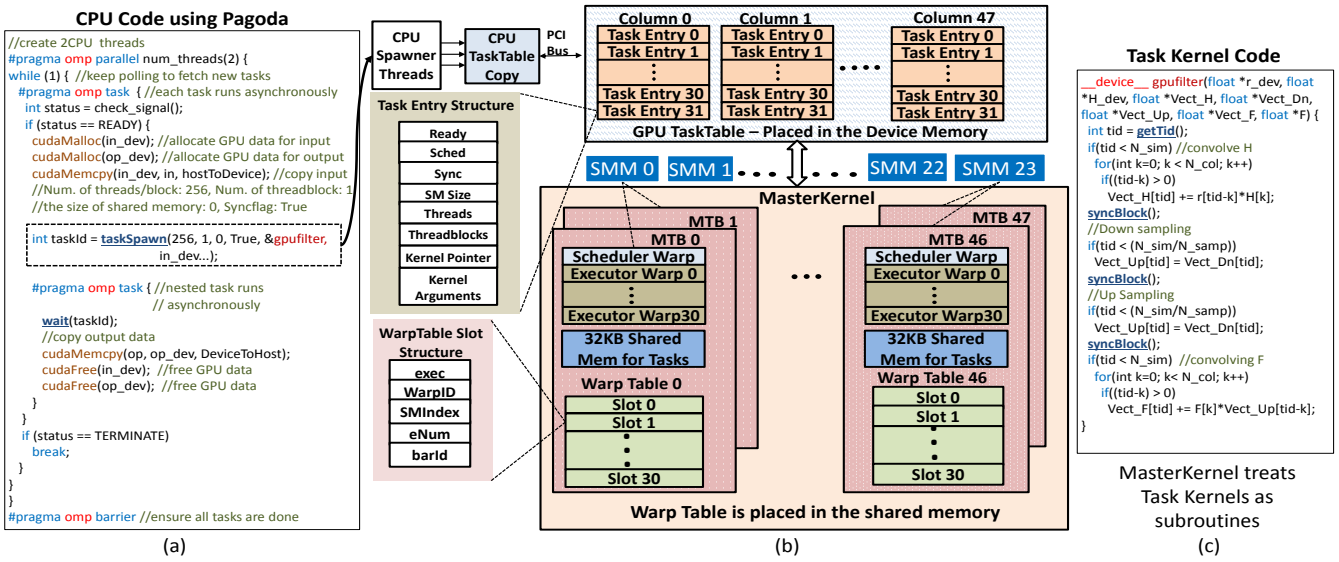


Figure 1: Pagoda runtime system overview: *The source task kernel and CPU code require few changes to an equivalent CUDA code. The MasterKernel design is shown for the Maxwell Titan X GPU. The 48 MasterKernel threadblocks (MTBs) have 1024 threads each. TaskTable is mirrored on both the CPU and GPU. The 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.*

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 SMM, the maximum number being 32. The threadblock size is limited to 1024 threads, or 32 warps. Each SMM can hold up to 2048 concurrent threads. Both the shared memory and registers of an SMM are partitioned among the executing threadblocks. There is no 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 SMMs 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. Consider a scenario of narrow tasks, where one task has 256 threads, or 8 warps. If only one task is executed at a time, the occupancy would be $(8/(64\times24))\times100\% = 0.52\%$. 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. $(8\times32/(64\times24))\times100\% = 16.67\%$.

3. PROGRAMMING WITH PAGODA

Programmers use Pagoda API functions in their applications to access the Pagoda runtime system. Pagoda supports the CUDA programming model, where-by the 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*.

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.

Fig. 1a shows a possible implementation of Pagoda host code, while Fig. 1c 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__`.

4. PAGODA NARROW TASK PROCESSING

This section first describes the MasterKernel design that achieves resource virtualization. Next, it presents the Pagoda task spawning mechanism. The 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 ag-

gregate updates. Lastly, 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 continually executes on the GPU as a CUDA kernel. It acquires all GPU resources, namely warps, shared memory, and registers, that later get allocated to tasks running on Pagoda.

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

The first warp of each MTB is called a *scheduler warp*, while the rest of the 31 warps are known as *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 [31]. 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 [18]. 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 [14], where 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. 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) *ready* field, and 8) *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.

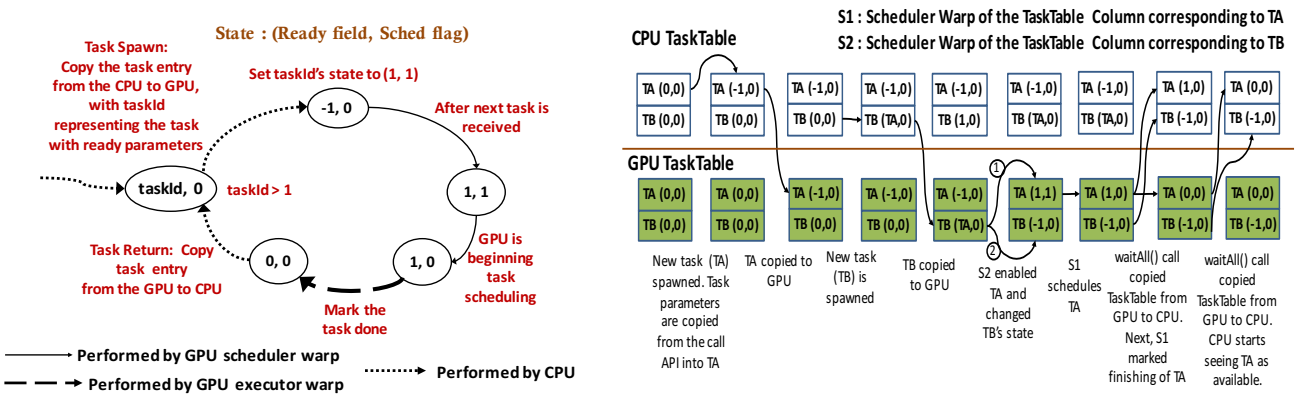
4.2.1 TaskTable Operation

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 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.

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. Fig. 2a presents a task's state diagram.

Fig. 2b presents an example execution of task A (TA). When the *taskSpawn* function is executed by the CPU, Pagoda finds a TaskTable entry with a cleared *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 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 clears 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). Next, S2 sets the state of the current task to (-1, 0) (see Algo. 1, lines 5-13). S2 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 clears 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



(a) ***TaskTable state diagram :*** *The CPU only touches TaskTable entries with cleared* ready *fields, and the GPU only touches TaskTable entries with non-zero* ready *fields, allowing for simultaneous TaskTable updates from CPU and GPU. The* sched *flag determines when a task gets scheduled to warps.*

(b) ***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.*

Figure 2: Obtaining Correct TaskTable Ordering

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 cleared *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 cleared *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 performed extensive micro-benchmarking to ascertain that the in-flight writes by the CPU to the TaskTable are visible to the GPU and vice-versa on two GPU architectures, Tesla K40 and Maxwell Titan X. This behavior is also confirmed by other researchers [12, 31].

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. Secondly, 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). Lastly, 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

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

```
1  while (1) do
2    if tid < warpSize then // scheduler warp does this
3      for (i = 0; i < numEntriesPerPool; i++) do
4        entry ← gTaskPool[i]
5        taskId = entry.ready
6        if taskId > 0 then
7          prevEntry ← taskTable entry for taskId
8          if prevEntry.ready ≠ -1 then
9            threadfence()
10           continue
11         else
12           prevEntry.ready ← 1
13           prevEntry.sched ← 1
14       if entry.sched then // check if sched flag is set
15         entry.sched ← 0
16         doneCtr[i] ← ctr[i] ← getNumWarps(entry)
17         if entry.SMSize > 0 ∨ entry.sync then //
             schedule warps per threadblock
18           for (j = 0; j < entry.numTB; j++) do
19             if (entry.sync) then barId ← getBarId()
20             if (entry.SMSize > 0) then
21               do
22                 deallocMarkedSM() // avoids
                     deadlocking
23                 retVal ← allocSM(entry.SMSize, &index)
24               while (retVal == false)
25             ctr[i]← getNumWarpsPerTB(entry)
26             pSched(ctr[i]×j, i, index, barId, &ctr[i])
27         else // schedule all warps
28           pSched(0, i, 0, 0, &ctr[i])
29   else // executor warps do this
30     if (warpTable[warpId].exec) then
31       entryId ← warpTable[warpId].eNum
32       tEntry ← gTaskPool[entryId]
33       *(tEntry.funcPtr))(tEntry.args) // warp executes the
           task
34       if (laneId == 0) then
35         if lastWarpInBlock() then // only 1 thread per
             threadblock performs this
36           if (tEntry.SMSize > 0) then // dealloc SM
37             markSMForDealloc(warpTable[warpId].SMindex)
38           if (tEntry.sync) then
39             releaseBarId[tEntry.barId]
40         threadfence_block()
41         if (atomicDec(&doneCtr[entryId])) then
42           tEntry.ready ← 0; // free the task entry
43         warpTable[warpId].exec ← 0 // warp is free now
```

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.

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 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

```
1  Function pSched (baseWarpId, eNum, index, barId,
     warpCtr)
2    threadDone ← 1 // private per thread
3    i ← tid; // private per thread
4    while (1) do
5      if (i < numEntriesPerPool) then
6        threadDone ← 0
7        if (!warpTable[tid].exec) then
8          if (id ← (atomicDec(warpCtr)) >= 0) then
9            warpTable[i].warpId ← id + baseWarpId
10           warpTable[i].eNum ← eNum
11           warpTable[i].SMindex ← index
12           warpTable[i].barId ← barId
13           threadfence_block()
14           warpTable[i].exec ← 1
15     if (*warpCtr <= 0) then threadDone ← 1
16     if (__all(threadDone == 1) = true) then //
         Synchronize threads in the scheduler warp
17       break
18     i ← i + 32
19     if (i > numEntriesPerPool) then i ← tid
```

Scheduling is performed by the threads of the scheduler warp in parallel, through the *PSched* function (Algo. 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.

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 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 [39], Pagoda's algorithm is motivated by the buddy system mechanism [13] to reduce overhead. Threads of the scheduler warp in the MTB are responsible for performing the allocations and deallocations.

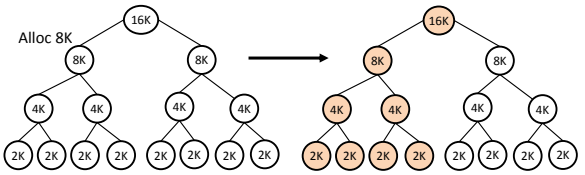


Figure 3: Allocating 8K of shared memory in Pagoda: *The value in each node represents the size of the shared memory block. Note that not all levels of the tree are shown here. The white nodes are free blocks and the shaded nodes are allocated blocks.*

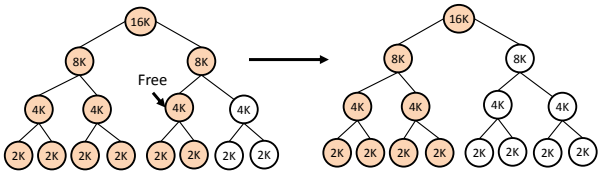


Figure 4: Deallocating 4K of shared memory in Pagoda: *Ancestors of the current node are marked free only if the sibling is free.*

Data Structure: The memory blocks are represented as nodes in a tree, as shown in Fig. 3. 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: Fig. 3 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: Fig. 4 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 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. Pa-goda achieves this using named barriers (using *bar.sync* instruction) in the PTX programming model [23]. 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 (Algo. 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.

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 Maxwell Titan X GPU, which contains 3072 1000MHz GPU cores with 12GB RAM. The machine runs Ubuntu 14.04, with 24GB RAM and an Intel core-i7 4.0GHz quad-core CPU. 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 7.5, with *-O3*. 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 CPUs each having 10 cores running at 2.6GHz.

Table 4 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 3 shows the workload characteristics of the benchmarks.

6.2 Comparison against Runtime Mechanisms

Fig. 5 shows that Pagoda obtains higher performance than other runtime schemes, namely, PThreads on the CPU, CUDA-HyperQ and GeMTC on the GPU. The speedup

Table 3: Benchmark Characteristics

Benchmark	Source	Task Type	Input Set per Task(each task is one image, signal, matrix or network packet)	Number of 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 [27]	Irregular	64 × 64 images	32K	24	76	✗	✗	28
FilterBank(**FB**)	StreamIt [35]	Regular	Signals of width 2K	32K	35	65	✗	✓	21
Beam-Former(**BF**)	StreamIt [35]	Regular	Signals of width 2K	32K	13	87	✗	✗	34
Image Convolution(**CONV**)	CUDA SDK [19]	Regular	128 × 128 images	32K	30	70	✗	✗	25
DCT8x8(**DCT**)	CUDA SDK [21]	Regular	128 × 128 images	32K	81	19	✓	✓	33
Matrix-Mul(**MM**)	CUDA SDK [22]	Regular	64 × 64 matrix	32K	51	49	✓	✓	30
Sparse LU Decomposition(**SLUD**)	OpenMP Task Suite [4]	Irregular	32 × 32 matrix	273K	3	97	✗	✗	17
3DES	NIST [5]	Irregular	Network packets sized 2K-64K	32K	74	26	✗	✗	26

Table 4: Benchmark Description

MB	Mandelbrot sets are used in fractal analysis [6]. 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 [22]. We used small matrix sizes, with each multiplication running as a task to simulate the behaviour seen in an earthquake engineering simulator [16]. The behaviour 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 [15]. 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 [5]. Network routers encrypt multiple packets as they arrive, each of which is represented as a narrow task. We use NetBench [17] to generate varied sizes of network packets that 3DES encrypts.
DCT	The Discrete Cosine Transform (DCT) [21] 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 [10]. Processing each image represents a narrow task.
CONV	Convolution filters [19] 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.

is calculated over the entire execution time, i.e, it measures both compute and data copy times. The performance increase is attributed to the high utilization achieved by Pagoda. The GeMTC versions do not use shared memory, since GeMTC has no support for it. We could not implement SLUD in GeMTC; GeMTC needs the number of tasks to be pre-defined, which is not the case in SLUD. GeMTC launches work in batches, where batch comprises a SuperKernel that runs tasks. The default GeMTC design used 32 threads per SuperKernel threadblock, obtaining only 50% occupancy. We hence modified GeMTC to use more threads; from 64 threads onwards, GeMTC can obtain 100% occupancy. Our evaluation uses 128 threads per task for HyperQ, GeMTC, and Pagoda, which was heuristically chosen because GeMTC performs reasonably well in this configuration (Fig. 7). GeMTC performs worse than HyperQ in MB and 3DES because these applications contain irregular workloads. Speedup achieved by all GPU schemes is low in DCT because the application is data copy bound. For a fair comparison with the CPU, we implemented OpenMP with data parallelism, OS-based task scheduling, Python-based thread pooling, and PThreads-based task parallelism. PThreads obtained the best results, which we include in Fig. 5.

Fig. 6 presents weak-scaling results. It compares execution times when the task count is varied. For low task counts, none of the schemes occupy the entire GPU, and hence HyperQ and GeMTC perform fairly well. However, once the task count grows beyond 512, Pagoda obtains higher performance because of increased utilization.

6.3 Comparison against Static Task Fusion

Static task fusion combines multiple tasks into a monolithic large task [37, 8]. It is clear that such mechanisms perform the best if all tasks start and end together. Benefits from Pagoda come into play when the computations of tasks would not end together, due to irregularities. Here, we compare the speedups achieved by Pagoda, static fusion, PThreads, and CUDA-HyperQ when task compute workloads vary. To obtain such irregularity, we generated input sizes of different tasks in a pseudo-random manner. We

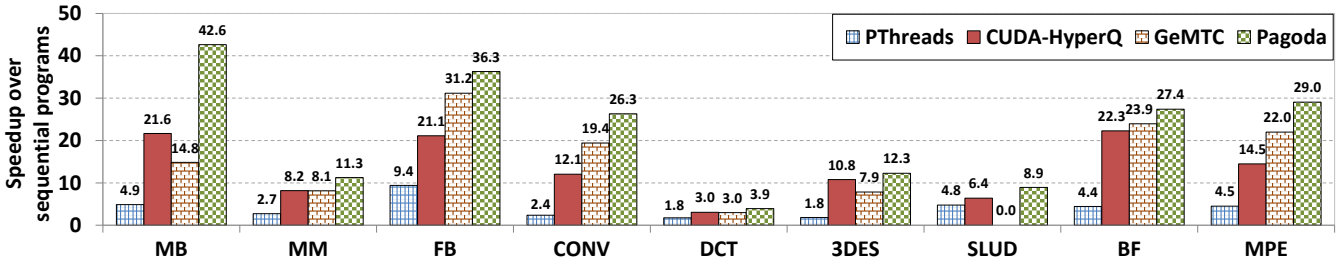


Figure 5: Overall Performance Comparison: *The number of tasks in each benchmark is constant (32K), except SLUD, which contains 273K tasks. Each GPU task uses 128 threads. The measurement of execution time contains both data copy and compute times. Pagoda significantly outperforms CUDA-HyperQ(1.51x), 20-core PThreads(5.70x), and GeMTC(1.69x), owing to the higher utilization.*

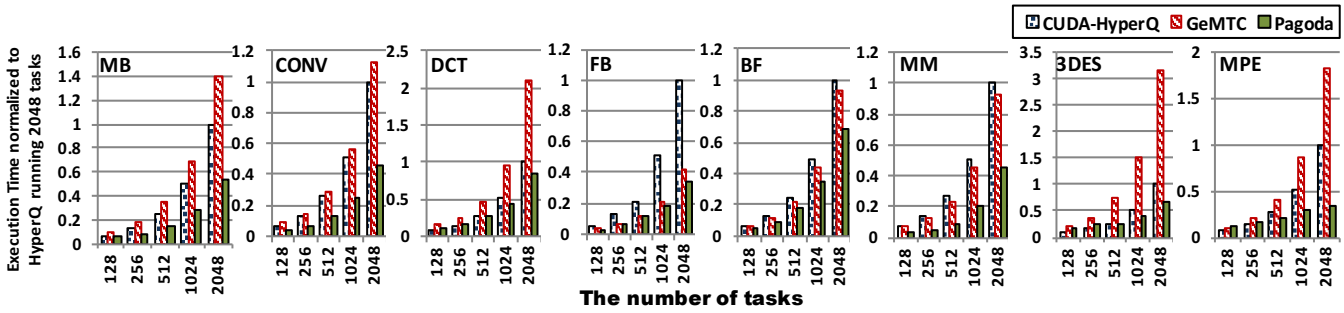


Figure 6: Weak scaling with number of tasks: *Each task contains 128 threads, since that configuration works the best for GeMTC. Pagoda versions run faster than HyperQ and GeMTC beyond 512 tasks. The Pagoda execution time scales well with the number of tasks.*

could not fuse SLUD since the number of tasks is not known statically. Each application contained 32K tasks. Each subtask in the statically fused task uses 256 threads. We chose this number heuristically, since selecting the best thread count per task is infeasible in static fusion. On the other hand, the runtime schemes of Pagoda/CUDA-HyperQ allow for dynamic thread count selection, based on the size of the irregular task. The tasks in Fig. 9 use between 32 and 256 threads for Pagoda/CUDA-HyperQ. The Pagoda performance is superior to all other schemes. Pagoda achieves a geometric mean speedup of 1.79x over static fusion. Note that the static fusion approach represents the upper-bound performance from batch-scheduling systems such as GeMTC, since the execution time is bounded by the longest running task.

Pagoda's performance gain depends on the input size, thread counts per task, and the number of tasks. Fig. 6 showed the impact of the number of tasks. Next, we analyze the impact of thread counts on the GPU compute time, while keeping the task input sizes fixed and varying the work done per thread. Fig. 7 compares benchmark execution times with a varying number of threads per task. No shared memory was used in either of the program versions because GeMTC does not support it. Pagoda outperforms both CUDA-HyperQ and GeMTC on all configurations. The performance benefits of Pagoda over HyperQ decrease with thread count because the underutilization becomes less severe. Generally, Pagoda performance increases with thread counts owing to the increased pipelined processing benefits. Pagoda's performance on FB decreases at higher thread counts because synchronization in that benchmark incurs a higher penalty when more threads must synchronize. The GeMTC performance does not change much with the thread count. This is explained since each batch, or SuperKernel, contains a fixed number of total threads.

6.4 Understanding Pagoda's Utility

Fig. 8 studies Pagoda's compute time versus HyperQ on two representative benchmarks wherein both the input size and thread counts are varied. The HyperQ configurations use 256 threads per threadblock. The evaluation uses 32K tasks. For most inputs, Pagoda gains significant speedups over HyperQ owing to the increased underutilization, until the thread count is greater than 512. Afterwards, Pagoda benefits diminish, since HyperQ can then better utilize the GPU without having to pay the Pagoda scheduling cost. However, in certain cases after further increasing the thread count, e.g., CONV with input size 256x256 and 64K threads, Pagoda starts to significantly outperform HyperQ. We attribute this behavior to the warp-level scheduling in Pagoda against the threadblock-level scheduling in CUDA. CUDA prohibits a new threadblock from launching until all warps of the previous threadblock finish, where Pagoda can schedule a warp from a new threadblock as soon a warp in a running threadblock completes.

6.5 Task Latency Analysis

Fig. 10 compares the average latency obtained per task in Pagoda with that in a statically fused task. We use two representative applications: 3DES, which contains irregular tasks, and MM, which contains regular tasks. Each task in a statically fused kernel, or in a batch-based system such as GeMTC, can only finish when all tasks in the fused kernel or batch have finished. Therefore, their achieved average latency increases with the number of tasks. On the other hand, the average latency of each Pagoda task remains the same for any number of launched tasks.

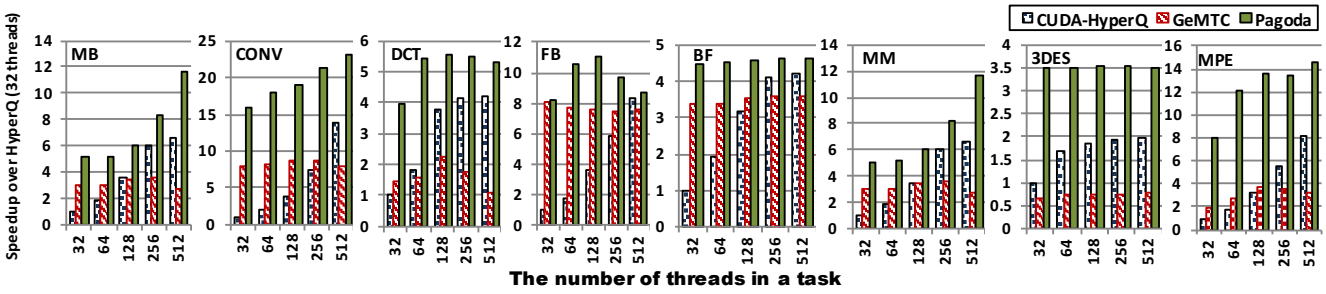


Figure 7: Comparison of computation times with different thread counts per task: *The number of tasks in each benchmark is 32K. The amount of work per task remains constant in all thread configurations. Pagoda achieves geometric mean speedup of 2.29x over HyperQ, and 2.26x over GeMTC at 128 threads. Pagoda's performance benefit over HyperQ decreases as the number of threads per task increases.*

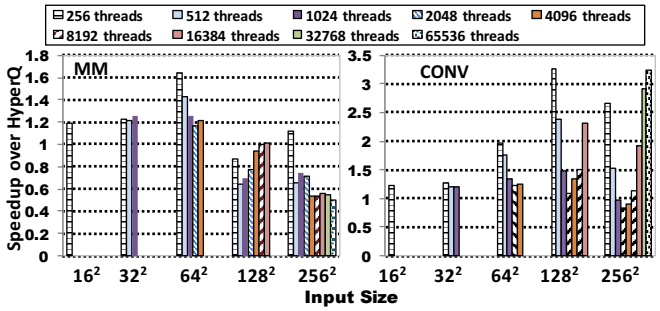


Figure 8: Effects of varying threads per task for different input sizes: *For small thread counts, Pagoda outperforms HyperQ in all input sizes. For large thread counts, Pagoda may still outperform HyperQ owing to its finer grain of scheduling.*

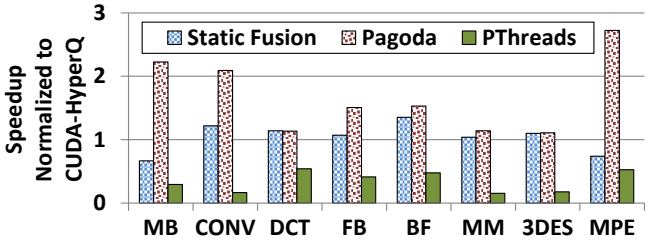


Figure 9: Performance comparison of static fusion, CUDA-HyperQ, PThreads, and Pagoda with irregular tasks: *Dynamic task spawning mechanism in Pagoda obtains high performance even with irregular workloads.*

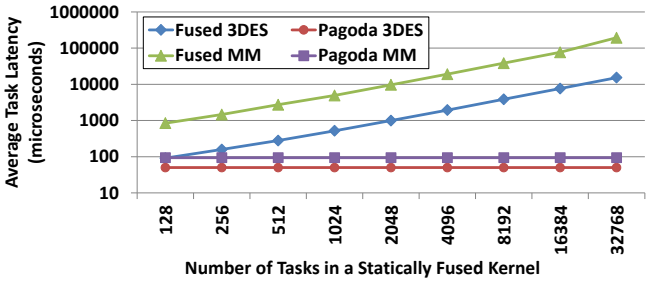


Figure 10: Average Latency of Tasks: *Pagoda achieves much lower latency compared to static fusion.*

6.6 Benefits of the Continuous Spawning and Concurrent Pipelined Task Processing Approach

To compare the performance benefits of continuous spawning and concurrent pipelined task processing in Pagoda, we created a Pagoda version that spawns tasks in batches, i.e., no new tasks are spawned until all tasks in the previous batch finish. This resembles the task spawning in GeMTC; however, it still performs concurrent task scheduling. We chose the same batch size as GeMTC's. Fig. 11 compares GeMTC, Pagoda Batching, and Pagoda, which performs continuous spawning and concurrent, pipelined task processing. The total task count is 32K, and each task contains 128 threads. The performance difference between GeMTC and Pagoda batching demonstrates the benefits achieved by concurrent task processing. The performance difference between Pagoda and Pagoda Batching demonstrates the benefits achieved by continuous, pipelined task spawning. This mechanism is enabled by Pagoda's TaskTable which offers simultaneous CPU task spawning and GPU task scheduling. Continuous spawning and concurrent, pipelined processing achieve significant benefits. CONV is the only benchmark that does not benefit from continuous spawns, owing its regular, extremely short-running tasks. Multi-Programming Environment(MPE) obtains exceptionally high benefits from continuous, pipelined processing due to its unbalanced tasks.

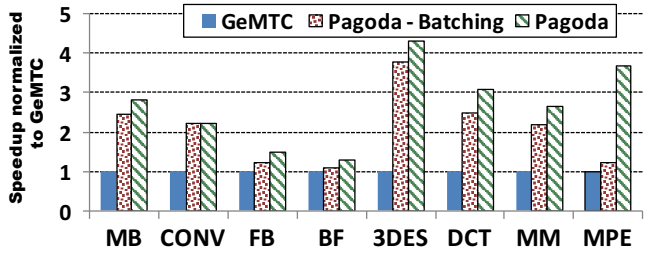


Figure 11: Benefits of Pagoda Continuous Spawning and Concurrent, Pipelined Task Processing: *Pagoda performs both continuous spawning and concurrent, pipelined task processing. Pagoda-batching only performs concurrent task processing. GeMTC performs neither. Pagoda outperforms GeMTC in all cases.*

Table 5: Compute performance comparison of tasks run in Pagoda with and without shared memory allocation: *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.*

Bench-mark	Pagoda with Shared Memory		Pagoda without Shared Memory	
	Speedup over HyperQ using Shared memory	Achieved Occu-pancy	Speedup over HyperQ using Shared memory	Achieved Occu-pancy
DCT	1.35x	25%	1.25x	97%
MM	1.51x	97%	1.20x	97%

6.7 Pagoda Shared Memory Analysis

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.

7. RELATED WORK

Task-based models [1, 3] 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. [37] present a mechanism where such fusion achieves higher utilization, resulting in energy benefits. KernelMerge [8] 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 [28, 25] perform kernel consolidation leveraging concurrent GPU kernel execution. They launch multiple concurrent kernels, where resources not being used by one kernel can be yielded to another. The first approach [28] relies on a threadblock-level launching scheme. The second approach [25] 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 et. al. [41] present an approach where a large kernel is split into independent smaller kernels that co-execute to achieve better utilization. Kato et al. [11] 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 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 [14]. 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 [9]. Unlike the MasterKernel, the SuperKernel does not guarantee an occupancy of 100%, and therefore may face underutilization. Secondly, 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 [29] to overlap data transfers with kernel computations. The distinguishing factor in the Pagoda pipelined task processing is that it overlaps spawning, which comprises the 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. [40] showed that fusing cross-kernel threadblocks can obtain better shared memory performance. Pagoda's 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 [38, 26, 34]. 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. [30] focus on virtualizing the GPU as a whole in a cloud with multiple GPUs. Becchi et al. [2] 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

The paper 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 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 achie-ves a geometric mean speedup of 1.51x over CUDA- HyperQ, 1.69x over GeMTC, and 5.70x 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 easier for narrow task applications to exploit GPUs, and will encourage the growth of non-traditional workloads on GPUs.

9. REFERENCES

[1] C. A. Augonnet, S. Thibault, R. Namyst, and P.-A. W. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, Feb. 2011.

[2] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. 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, pages 97–108, New York, NY, USA, 2012. ACM.

[3] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. AyguadÃl', and J. Labarta. Productive programming of GPU clusters with ompss. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568, May 2012.

[4] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. 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, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

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

[6] Fraqtive. [Online]. Available: http://fraqtive.mimec.org/, 2016. (accessed March 5, 2016).

[7] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 2. IEEE Press, 2008.

[8] C. Gregg, J. Dorn, K. Hazelwood, and K. Skadron. Fine-grained resource sharing for concurrent GPGPU kernels. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[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] A. S. Kaseb, E. Berry, Y. Koh, A. Mohan, W. Chen, H. Li, Y. H. Lu, and E. J. Delp. A system for large-scale analysis of distributed cameras. In *Signal and Information Processing (GlobalSIP), 2014 IEEE Global Conference on*, pages 340–344, Dec 2014.

[11] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[12] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein. GPUnet: Networking abstractions for GPU programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

[13] K. C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, Oct. 1965.

[14] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu. 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, pages 153–164, New York, NY, USA, 2014. ACM.

[15] J. W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Rev.*, 34(1):82–109, Mar. 1992.

[16] F. McKenna. Opensees: A framework for earthquake engineering simulation. *Computing in Science Engineering*, 13(4):58–66, July 2011.

[17] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: A benchmarking suite for network processors. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 39–42. IEEE Press, 2001.

[18] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. *SIGPLAN Not.*, 48(8):103–112, Feb. 2013.

[19] NVIDIA. Texture-based Separable Convolution. [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples/#graphics, 2007. (accessed March. 5, 2016).

[20] NVIDIA. Hyper-Q Example. [Online]. Available: http://docs.nvidia.com/cuda/samples/6_Advanced/simpleHyperQ/doc/HyperQ.pdf, 2012. (accessed March. 5, 2016).

[21] NVIDIA. The White Paper of Discrete Cosine Transform for 8x8 Blocks with CUDA. [Online]. Available: http://docs.nvidia.com/cuda/samples/3_Imaging/dct8x8/doc/dct8x8.pdf, 2012. (accessed March. 5, 2016).

[22] NVIDIA. CUDA. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2015. (accessed March 5, 2016).

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

[24] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica. The case for tiny tasks in compute clusters. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, 2013. USENIX.

[25] 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.

[26] 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.

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

[28] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. 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, pages 217–228, New York, NY, USA, 2011. ACM.

[29] A. Sabne, P. Sakdhnagool, and R. Eigenmann. Scaling large-data computations on Multi-GPU accelerators. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 443–454, New York, NY, USA, 2013. ACM.

[30] D. Sengupta, R. Belapure, and K. Schwan. Multi-tenancy on GPGPU-based servers. In *Proceedings of the 7th International Workshop on Virtualization Technologies in Distributed Computing*, pages 3–10, 2013.

[31] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a file system with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, New York, NY, USA, 2013. ACM.

[32] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 13–22, New York, NY, USA, 1993. ACM.

[33] J. Subhlok and G. Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *J. Parallel Distrib. Comput.*, 60(3):297–319, 2000.

[34] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 193–204, Piscataway, NJ, USA, 2014. IEEE Press.

[35] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[36] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–11. IEEE, 2008.

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

[38] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel: Fine-grained sharing of gpgpus. *IEEE Computer Architecture Letters*, PP(99):1–1, 2015.

[39] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 1–116, London, UK, UK, 1995. Springer-Verlag.

[40] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou. 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, pages 283–292, New York, NY, USA, 2012. ACM.

[41] J. Zhong and B. He. Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1522–1532, June 2014.