7-2012

Supporting Preemptive Task Executions and Memory Copies in GPGPUs

Kyoung-Don Kang
Binghamton University--SUNY, kang@binghamton.edu

Can Basaran

Supporting Preemptive Task Executions and Memory Copies in GPGPUs

Can Basaran and Kyoung-Don Kang
Department of Computer Science
State University of New York at Binghamton
{*cbasaran, kang*}*@cs.binghamton.edu*

Abstract

GPGPUs (General Purpose Graphic Processing Units) provide massive computational power. However, applying GPGPU technology to real-time computing is challenging due to the non-preemptive nature of GPGPUs. Especially, a job running in a GPGPU or a data copy between a GPGPU and CPU is non-preemptive. As a result, a high priority job arriving in the middle of a low priority job execution or memory copy suffers from priority inversion. To address the problem, we present a new lightweight approach to supporting preemptive memory copies and job executions in GPGPUs. Moreover, in our approach, a GPGPU job and memory copy between a GPGPU and the hosting CPU are run concurrently to enhance the responsiveness. To show the feasibility of our approach, we have implemented a prototype system for preemptive job executions and data copies in a GPGPU. The experimental results show that our approach can bound the response times in a reliable manner. In addition, the response time of our approach is significantly shorter than those of the unmodified GPGPU runtime system that supports no preemption and an advanced GPGPU model designed to support prioritization and performance isolation via preemptive data copies.

1 Introduction

Processor technology has significantly been advanced. Especially, GPUs provide massive parallelism for low costs. For example, an NVIDIA Fermi GPU provides 512 cores [23]. GPUs are increasingly used for general purpose computations as they are becoming more programmable and flexible. For example, NVIDIA's CUDA [15] and ATI's OpenCL [12] support general purpose GPU programming. At the same time, the demand for high performance real-time computing is increasing in cyber physical systems that deal with large amounts of real-time sensor data. For example, GPUs provide an order of magnitude speedup over multicore solutions for computer vision tasks in autonomous driving [4]. Despite the powerful features, related work on the application of GPUs to real-time computing is scarce [5, 10, 14].

Unfortunately, applying GPGPU technology to real-time computing is not straightforward. In fact, supporting real-time scheduling for GPUs is a challenging problem that requires significant research efforts. For example, it is largely unknown how to analyze the schedulability of GPU tasks or provide a reliable yet not overly pessimistic estimates of the worst case execution times in GPUs [5, 10]. In this paper, we focus on a specific piece of the problem, i.e., supporting a basic capability for preempting a memory copy, and a GPU job, called kernel, execution for periodic soft real-time tasks.

A GPU, called a device, is a coprocessor to the hosting CPU. Thus, data have to be copied between the CPU and GPU memory. Since the memory copy operation is performed via non-preemptive DMA (Direct Memory Access), a high priority task can be blocked due to a memory copy transaction of a low priority task. In addition, a kernel cannot be preempted once it starts running in the GPU. Therefore, a high priority task can suffer from priority inversion. The resulting real-time performance penalty can be serious especially when a high priority task is blocked by a large low priority kernel that does complex computations using big data. Kato et al. [10] have developed a novel approach called RGEM (Responsive GPGPU Execution Model) that divides data into a set of fixed size chunks to allow preemption between the consecutive chunk copies. In this way, their approach significantly decreases the response time of high priority tasks, while supporting enhanced performance isolation between tasks with different priorities. However, their approach supports no kernel preemption. As a result, a high priority kernel may be blocked by a low priority kernel already running in the GPU.

To address these issues, we present a new approach, called PKM (Preemptive Kernel Model), which supports 1) preemption of kernels that implement periodic real-time

This work was supported, in part, by NSF grant NSF CSR-1117352

tasks assigned fixed priorities, 2) efficient preemptive memory copies between the CPU and GPU memory, and 3) concurrent processing of memory copies and periodic kernels. In our approach, a job, i.e., a periodic task instance, is implemented as a GPU kernel. To make a job preemptive, PKM divides one kernel into a set of subkernels where a subkernel is executed by a specified number of thread blocks called a subgrid in this paper. In PKM, a job suspends itself after finishing a current subkernel, if a high priority job is waiting.

PKM supports preemptive memory copies by dividing a memory copy transaction into a series of copies of smaller data chunks, similar to [10]. However, different from [10], PKM allows concurrent executions of memory copies and kernels. Because a job with the highest priority cannot start running in the GPU before the data to process is copied to the GPU memory, a low priority kernel that already has data to process in the GPU memory can be executed, while the memory copy transaction for the highest priority task is being performed or vice versa. By overlapping data copies and job executions, we aim to further reduce the response time and unnecessary blocking between memory copies and kernel executions. In addition, unlike [10], PKM directly passes input sensor data from the operating system address space to the GPU memory via DMA to eliminate unnecessary data copies between the operating system and user address space. In this way, PKM further reduces the delay for memory transactions, while decreasing the memory consumption.

To show the feasibility of our approach, we have implemented a prototype PKM system in an NVIDIA GeForce GTX 460 GPU. The experimental results show that our approach bounds the response times in a reliable manner. Also, in an experiment, the response time of a high priority task in PKM is approximately $\frac{1}{10}$ and $\frac{1}{100}$ of those measured in RGEM [10] and basic CUDA, which provide preemptive data copies and no preemption respectively. Thus, we heave experimentally verified that PKM is substantially more cost-effective than the two methodologies representing the state of the art.

The rest of the paper is organized as follows. GPU backgrounds and our preemptive model are discussed in Section 2. PKM design and implementation are discussed in Section 3. Performance evaluation is given in Section 4 and related work is discussed in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2 Preemptive GPGPU Model

In this section, basic GPGPU model based on the CUDA architecture is discussed. Based on the model, the approaches taken by PKM for preemptive kernel executions and memory copies are described.

2.1 System Model and Backgrounds

PKM aims to improve the performance of periodic soft real-time GPGPU tasks. In PKM, a task set consists of $n(\geq 1)$ tasks where a task τ_i $(1 \leq i \leq n)$ is associated with a period T_i and fixed priority P_i. A task is assumed to execute the same function to process input data, such as sensor readings and audio/video data, at every period. PKM is designed and implemented based on the NVIDIA CUDA [15] architecture using tools provided by the CUDA library [17]. Unlike [10], we do not modify underlying GPU device drivers that could be specific to certain GPU hardware, and different versions of GPU libraries or toolkits. Instead, PKM is designed to run in the user space and, therefore, applicable to various GPUs as long as the basic CUDA functionalities used in this paper are supported by them.

A CUDA application has a non-interruptible entry function, called a kernel, which invokes other (non-kernel) GPU device functions. To execute a kernel, data have to be uploaded from the CPU to GPU memory. After completing a kernel, computational results have to be downloaded from the device to the host memory. It is not allowed to preempt a running kernel. Neither is it possible for the host to communicate with a kernel executed in the device. A GPGPU has a number of SIMT (Single Instruction Multiple Threads) multiprocessors that host and switch between GPU threads. Thus, a GPU can accommodate thousands or even more live threads. The number of processing elements and maximum number of active threads that can run at once vary among GPUs [18]. The latter may also vary even in one device depending on the availability of resources at runtime.

CUDA provides a combination of hardware and software techniques such as DMA, streams, and events to enhance the performance of GPU applications [17]. DMA significantly improves the performance of host-device memory copies as it eliminates page faults and operating system overhead. A stream is a group of CUDA operations that need to be executed sequentially in the device. However, operations from different streams can be interleaved. In PKM, we divide a kernel into subkernels to support fine grained preemption between subkernels, if necessary, to avoid priority inversion due to non-preemptive kernel execution in CUDA. Further, we support an asynchronous, nonblocking memory copy using CUDA streams and pinned memory in the host to execute subkernels and memory copies of different streams, i.e., periodic jobs in a concurrent manner. For example, while stream A does a memory copy, stream B can execute its subkernel or vice versa. CUDA also provides events to query the state of asynchronous GPU transactions. Using events, the host can check the status and progress of streams without being blocked. PKM extensively use streams and events to

improve the response time of real-time tasks.

In PKM, a static set of periodic tasks are compiled together with the scheduler. PKM is completely implemented in the user space as a single process to avoid context switches between tasks. Tasks are implemented as C++ classes that inherit the `Task` base class and implement the `Task::run_job()` template function, which is invoked by the scheduler at specified intervals. In the `Task::run_job()` function, a periodic task instance (i.e., a job) requests the GPU to execute a series of memory copies and a kernel.

2.2 Fine-grained preemption of large kernels

Since an active CUDA kernel cannot be preempted, a large kernel can take up all the resources in the GPU for a considerable amount of time once it starts running. A CUDA kernel consists of one or more grids. A user-specified grid needed to execute a kernel consists of a number of blocks where a block consists of a number of threads. A CUDA programmer has to define the number of grids G, the number of blocks per grid B, and the number of threads per block R for a kernel. In PKM, a large kernel submitted by a user task is partitioned into subgrids that consist of a fixed number of CUDA blocks. PKM receives a complete kernel execution request from a user and divides it into subgrids to run subkernels. A system administrator explores an appropriate *subkernel size* S (the number of blocks per subgrid) via profiling to support fine grained kernel preemption in PKM with acceptable overhead. Essentially, there is a tradeoff between fine grained preemption and overhead. Smaller subkernels support more fine-grained preemption, but they are subject to more overhead. On the other hand, larger subkernels experience less overhead at the cost of more gross-grained preemption, which may increase the potential for priority inversion. Since an appropriate value of S are specific to an application and a device, we profile the relation between the subkernel size and slowdown due to partitioning a single kernel into multiple subkernels.

Given S and the other aforementioned parameters, PKM computes the number of subkernels: $M = \lceil G \times B/S \rceil$ where each block has R threads. For example, assume that a job requests a grid of 1024 blocks to execute a kernel. PKM divides this job into 4 subgrids, if S = 256 blocks. PKM processes a single subgrid of the job that currently has the highest priority. When a higher priority job arrives, it preempts the current job after finishing its subkernel currently being executed. To analyze this tradeoff, we have designed a matrix multiplication application that has a single grid. For varying sizes of input matrices, we measure the response time of the task for the decreasing subkernel size and the increasing number of subkernels. The results of the experiments for profiling is given in Table 1. We observe from the table that larger tasks better tolerate finer grained partitions in terms of the overhead. The 4096x4096 matrix multiplication task takes around 750ms when executed as a single kernel. When it is divided into 256 subkernels, it experiences only 9% increase in the response time as shown in Table 1. Notably, this is one of the desirable features of PKM. A large kernel, which may cause severe priority inversion and a significant real-time performance penalty as a result, is readily divisible into a series of subkernels with acceptable overhead. On the other hand, it is desirable to partition a relatively small kernel into a small set of subkernels that does not introduce a large increase in the response time. In this way, PKM calculates the subgrid size S for each task, starting from the highest priority task. Specifically, for each task, PKM chooses the largest S that results in a preemption interval that is equal to or shorter than the minimum of the periods of the higher priority tasks, if any.

Data Size	B	M	S	Slowdown
1024x1024	4096	4	1024	0%
		16	256	26%
		64	64	340%
2048x2048	16384	4	4096	0%
		16	1024	10%
		64	256	32%
		128	128	250%
4096x4096	65536	4	16384	0%
		16	4096	2%
		64	1024	8%
		256	256	9%
		512	128	42%

Table 1. Subkernel size vs. slowdown for matrix multiplication (B: #blocks/grid, M: #subkernels, and S: #blocks/subgrid)

2.3 Preemption of large memory transactions

PKM runtime system divides a kernel or a memory transaction into a series of smaller units. Given a memory copy request, PKM queues the request with the total data size. (Each task in PKM has two queues for scheduling kernels and memory transactions. A detailed description of the queue management in PKM is given in Section 3.) Each time the request is serviced, PKM copies only a smaller chunk of data and decreases the request size by the size of the portion, similar to [10]. Thus, the data chunk size affects the granularity of preemption and corresponding overhead. A smaller chunk size supports more fine-grained pre-

A subkernel and a subgrid used to run a subkernel are used interchangeably in this paper.

emption between memory transactions potentially increasing the overhead or vice versa. In this paper, we experimentally picked 1MB as the chunk size, since it provides fine granularity with the shortest response time for memory transactions among the tested various chunk sizes.

In PKM, different from [10], PKM overlaps a kernel execution with memory transactions as discussed before. Further, data are directly written to the DMA buffer without making an extra copy to the user space buffer and copying the data back to the DMA buffer. Removing extra data copies are important especially for processing large real-time sensor data such as audio/video streams, since extra copies will increase both the memory consumption and delay for memory transactions. We are unaware of any prior work that supports preemptive kernels and memory copies, while supporting overlapped processing of kernels and memory transactions for real-time applications.

3 System Design and Implementation

In this section, the data structures to model tasks and scheduling queues are described. Also, our approach to scheduling of preemptive kernels and memory transactions is discussed.

3.1 Data Structures

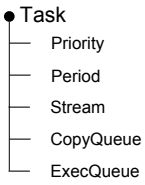


Figure 1. Data structure of a task in PKM

The task data structure used by PKM is shown in Figure 1. It contains attributes to specify the priority and period of a task. For each task, it also has a CUDA stream handle, a queue for copy transactions, and another queue for kernels. In PKM, each periodic real-time task is associated with a separate CUDA stream. Each task represented by a stream has its own private queues for its kernels and memory transactions separately.

The data structures used for copy and execution queue entries are shown in Figure 2. They have common fields for a timestamp and a CUDA event handle. In PKM, timestamps are only used to order memory/kernel operations within a task. There is no total or partial order among the timestamps of different tasks. Thus, a memory operation or kernel without any unfinished preceding operation of a task is eligible to run next. Among the eligible operations of

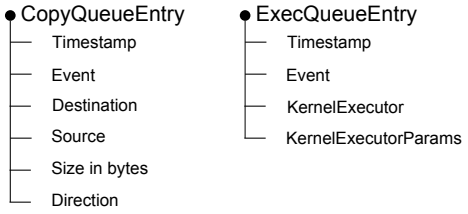


Figure 2. Data structure of a kernel or memory copy request

different tasks, PKM schedules a memory copy and kernel with the highest priority.

The CUDA event handle in Figure 2 is used mainly to check the completion of an operation after it is scheduled. This event handle is also used to gather response time statistics. In addition to these common attributes, each entry has parameters specific to the operation it is representing. A memory copy entry contains the source and destination pointers, total data size of the request, and the direction of the copy, i.e., upload from host to device or download from device to host. A single copy entry is created for each user data copy operation. The size, destination, and source pointers are updated as the data request is processed chunk by chunk. A kernel execution entry stores a user supplied execution object that is responsible for passing the executable with parameters (if any) and keeping track of the completion of the kernel.

Our approach greatly simplifies the design of the system-wide scheduler. The PKM scheduler only has to pick the highest priority memory copy and kernel that are ready to run immediately and submit them to the device. Having two separate queues for copy and execution transactions allows PKM to easily check each task for pending transactions of a particular type in non-ascending order of priority. Therefore, our approach does not require complex scheduling schemes, system-wide queues, message passing, or synchronized queue access mechanisms. The overhead of our scheduling is bounded by the cost of sorting the pending tasks and searching through them to find the highest priority kernel and memory operations ready to run. Pending tasks can be sorted using a min-heap. An update of a min-heap is performed in $O(\log n)$ time when the size of the task set is n. Thus, the runtime overhead for scheduling is bounded by $O(n)$ time due to the search for a kernel or memory transaction with the highest priority when a DMA request or subkernel execution finishes.

Notably, in a periodic task model, it is unnecessary for a job to continually check whether a higher priority job has arrived. Considering the periods of tasks, our scheduler analyzes offline when a task can be preempted by a higher priority task. For an arbitrary pair of tasks with different

priority levels, the scheduler computes offline when a low priority job needs to check an arrival of a higher priority job by computing the least common multiplier of the periods of the two tasks. As a result, each job knows when it has to suspend itself, if necessary, to avoid blocking a higher priority job. The scheduler has to compute this information for only one hyper period of the task set and repeatedly apply the computed result in the following hyper periods. Therefore, the offline computation is performed in O(n^2) time for n tasks.

3.2 Concurrent Scheduling of Copy and Kernel Operations

```
1 Task::run_job()
2 {
3   PKMemCpy(dev_dst, host_src, size, UPLOAD);
4   PKExec(KernelFunc(KernelParameters));
5   PKMemCpy(host_dst, dev_dst, size, DOWNLOAD);
6 }
```

Listing 1. Specification of a user task

A user defines a job (i.e., a kernel) using a number of functions provided by PKM. The pseudo code in Listing 1 is an example job launcher of a periodic task. A job typically generates a sequence of a data copy (upload), kernel execution, and data copy (download) requests through the PKM runtime system. If `run_job()` is called at time t determined according to the task period, two entries are inserted into the copy queue and one entry into the kernel queue of the task with timestamps $t, t+2$, and $t+1$, respectively.

```
1 PKScheduler::sched()
2 {
3   task_list.sort_by_priority();
4   if (current copy transaction is completed)
5     foreach (task in task_list)
6       if (task.isReady(COPY_QUEUE))
7       {
8         CopyParams cp = task.deque_copy(
              CHUNK_SIZE);
9         launch COPY on task.stream using cp;
10        break;
11      }
12  if (current exec transaction is completed)
13    foreach (task in task_list)
14      if (task.isReady(EXEC_QUEUE))
15      {
16        KernelParams kp = task.deque_exec(
              subGridSize);
17        launch KERNEL on task.stream using kp;
18        break;
19      }
20 }
21
22 boolean Task::isReady(queue)
23 {
24   if (queue == CopyQueue)
25     other_queue = ExecQueue;
26   if (queue == ExecQueue)
27     other_queue = CopyQueue;
28   return (!queue.empty() and
29     queue.front().precedes(other_queue.front()));
30 }
```

Listing 2. Pseudo code for scheduling

The scheduling function of PKM, i.e., `sched()` in Listing 2, makes necessary scheduling decisions when an instance of a periodic task is created via a call of `Task::run_job()` or a previously scheduled copy operation or subkernel completes. PKM maintains tasks in priority order in a list called the `task_list`. Each scheduling decision picks a copy operation and a subkernel that is associated with the highest priority and eligible to run next.

Example. Figure 3 shows three tasks with high, medium, and low priority scheduled using the algorithm in Listing 2. The PKM scheduler schedules the first entry in the copy queue of the high priority task, i.e., HP(10). The scheduler then looks for a kernel execution transaction. Although the kernel execution queue of the highest priority task is not empty, the scheduler will not schedule H(12) until the preceding data copy operations HP(10, 11) complete. Thus, it skips to the next task MP, which is not ready to execute, because the kernel execution transaction MP(9) is waiting for copy transactions MP(6, 7, 8) and these in turn are currently blocked by HP(10). Hence, the scheduler considers the lowest priority task LP and schedules the execution transaction LP(16) which is not waiting for any copy transaction. This decision completes the first scheduling round of the snapshot shown in Figure 3. The next decision is made when HP(10) or LP(16) completes. For the clarity of presentation, in this example, it is assumed that an arbitrary data copy and subkernel execution complete at the same time. Thus, the PKM scheduler concurrently schedules HP(11) and LP(17) transactions next and proceeds with remainder of the schedule shown in Figure 3.

3.3 Design of Preemptive Kernels

Preemptive memory copies are relatively easy to support due to the incremental nature of data copies between the host and device. However, supporting preemptive kernels is more complex than providing preemptive memory copies. Depending on the availability of resources, a GPU may execute a user-defined grid of thread blocks either at once or divide the grid into subgrids and execute them one by one. Note that a user cannot control the procedure. Neither is the procedure clearly known. Moreover, a grid executing a kernel is non-preemptive even if it is run as a series of subgrids by the device.

To support parallel programming, however, CUDA guarantees the consistency of the block index, blockIdx, used in a user kernel. For example, if a user specifies a grid of 1024 blocks, blockIdx ranges from 0 to 1023, even

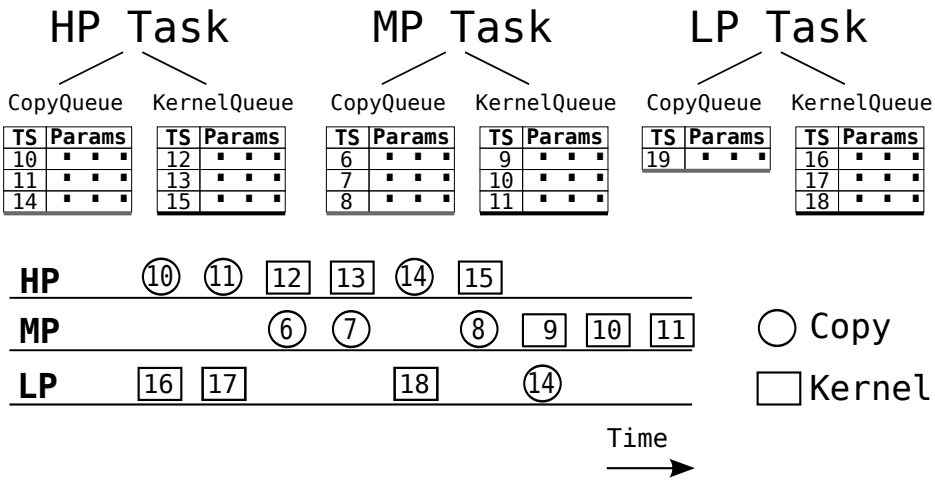


Figure 3. PKM scheduling example (HP: High Priority, MP: Middle Priority, LP: Low Priority, TS: Timestamp, Params: Parameters)

if the underlying CUDA runtime system divides the grid into, for example, 4 subgrids of 256 blocks each and execute the subgrids one by one. For instance, a statement such as **char** `*block_data = src[blockIdx * 1024]` partitions data into chunks so that each CUDA block receives a specific chunk of 1024 bytes of data based on its block index. However, if we were to manually schedule the same kernel in four grids of 256 blocks to enable preemption, the previous assignment statement will not execute correctly, since CUDA is unaware that these four smaller grids are correlated. As a result, the blocks in each of the four grids will be assigned 0 - 255 block indexes. Thus, one cannot support preemptive kernels by manually partitioning a grid into subgrids. Even if the approach works, it is onerous to require a user to redesign and re-implement the kernel to support fine grained preemption, which may not be a user's main concern.

To address the problem, PKM provides a new feature called *subBlockOffset*. In the previous example, PKM computes a list of subBlockOffsets (0, 256, 512, 768) and passes it to the kernel when each grid is launched. In this way, the k^{th} subkernel ($0 \leq k \leq 3$ in this example) receives an offset of $k \times S$ where S is the subkernel size (256 in the example). PKM then computes `subBlockIdx = blockIdx + subBlockOffset` for the kernel and use the computed subBlockIdx to perform the data assignment discussed before; that is, PKM replaces the previous assignment statement with **char** `*block_data = src[`*subBlockIdx* `* 1024]`. To write a preemptive kernel, a user only has to use the PreemptiveKernel constructor provided by PKM. PKM automatically divides a grid to subgrids, where the size of a subgrid (a subkernel) is determined offline by profiling the subgrid size vs. overhead relation as discussed before. In summary, to divide a kernel to subkernels and invoke them without requiring a user to manually partition a kernel, PKM implements kernel preemption by launching subkernels and maintaining a `subBlockIdx` which is used by the real-time tasks instead of CUDA auto-variable `blockIdx`.

4 Performance Evaluation

In this section, we have implemented PKM and RGEM [10]. We compare their performance to basic CUDA that provides no preemption. Especially, we measure the response time that is important for real-time computing. The system used in the experiments has an NVIDIA GeForce GTX 460 GPU, AMD Athlon II X4 630 CPU, 4GB RAM, and 500GB hard disk running Linux 2.6.32.21 kernel.

For performance evaluation, we use two micro-benchmarks: matrix multiplication and linear search, similar to RGEM [10]. Matrix multiplication (MM) task represents a compute intensive task such as processing video sensor data. Linear search (LS) task is more I/O intensive, it scans its input data and produces a small list of matches. It models filtering large sensor data to pick important ones. While it is relatively easy to implement LS as a series of independent small kernels, MM requires direct support for preemptive kernels due to incremental nature of LS. In this paper, we implement an instance of a periodic LS task as a series of subkernels for PKM and RGEM. Specifically, each LS subkernel uploads 1MB of data, processes the data, and downloads the result. However, only PKM supports preemptive subkernels for matrix multiplication. In our experiments, PKM subgrid size for MM is 1024 blocks.

4.1 Priority Inversion vs. Response Time

In this experiment, we aim to measure the response time of a high priority (HP) task when there is a competing task with a low priority (LP). A HP task does 1024x1024 MM at every 50ms. An LP task is either LS or MM. We measure the response time of the HP task as the input data size of an LP task increases. Each data point in Figure 4 is the average response time of the HP task when the HP task is executed every 50ms and a low priority LS or MM task is run periodically for a specific size of the input data for 100s. The input data sizes of the low priority LS task used for these experiments are 512KB, 1MB, 2MB, ..., 512MB. The data sizes of the low priority MM used for the experiments are 256x256, 512x512, 768x768, ..., 4096x4096. As we use bigger data, we also increase the period of the low priority LS task from 20ms to 500ms. When a low priority MM task is used to generate competing workloads, we extend the period from 20ms to 800ms to avoid overloading the GPU. By doing these extensive experiments, we intend to observe the impact of potential priority inversion on the performance of the HP task. We have derived 90% confidence intervals; however, we have omitted them since they are less than 1%.

In Figure 4(a), we show the response time of the high priority MM task when the competing workload is LS. When the lower priority task is LS, the response time of the HP task in both RGEM and PKM is nearly constant as shown in Figure 4(a). The response time of the HP task is 6ms for PKM and 14ms for RGEM. By supporting kernel preemption in addition to preempting data chunk copies, and overlapping execution of these, PKM achieves over 2x speedup and up to an order of magnitude performance enhancement for the high priority task compared to RGEM and CUDA, respectively. When the input data of the low priority LS task is only 512KB, CUDA supports the shortest response among the tested approaches as shown in Figure 4(a), because it does not have any overhead for supporting preemptive memory copies or kernel executions. However, the response time of the HP task in CUDA increases rapidly as the input size of the LS increase due the non-preemptive nature and resulting priority inversion. PKM achieves substantial performance enhancement compared to CUDA and RGEM, because it can preempt low priority kernel executions as well as memory transactions. Although RGEM can preempt low priority data chunk copies, it cannot preempt a low priority kernel. As a result, it shows better performance than CUDA does, while providing worse performance than PKM does.

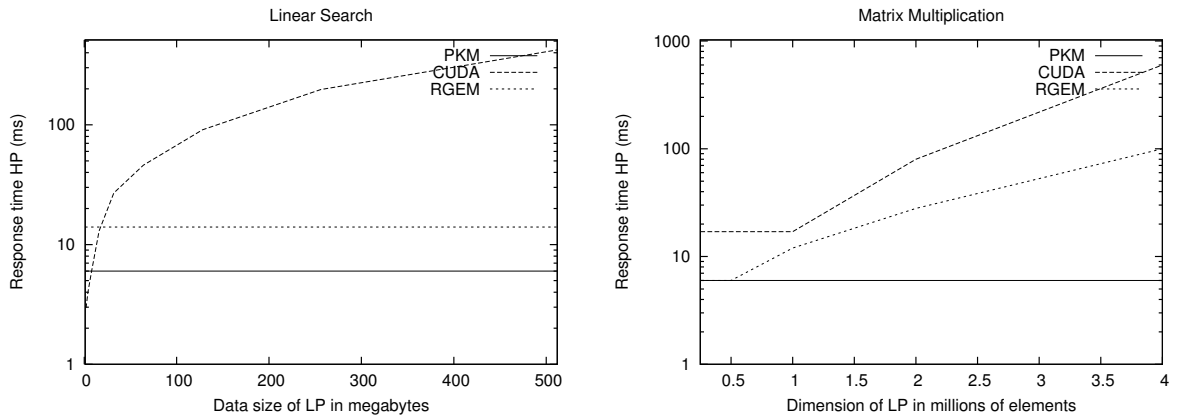
In Figure 4(b), we show the response time of the high priority MM task when the competing workload is another MM task with lower priority. The average response time of the HP task in PKM is stable around 6ms as shown in Figure 4(b). In RGEM, however, the response time increases from approximately 6ms to 100ms, since it suffers from priority inversion. Its performance is degraded compared to that in Figure 4(a), since the low priority MM task is a single kernel. The basic CUDA with no support for preemption shows the largest response time increase, reaching approximately 600ms. Hence, the response time of RGEM and basic CUDA is up to roughly 16x and 100x higher than that of PKM.

Task	Type	Period	Data Size
τ_0	MM	50ms	1024 x 1024
τ_1	LS	100ms	32MB
τ_2	MM	100ms	1024 x 1024
τ_3	MM	600ms	2048 x 2048
τ_4	LS	1200ms	96MB

Table 2. Experimental Settings

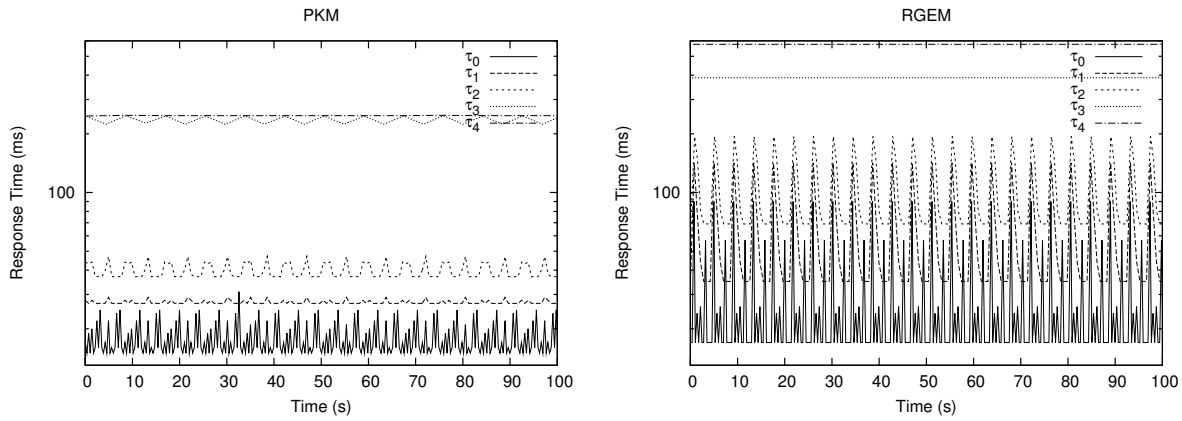
4.2 Preemptive vs. Non-Preemptive Kernels

In this experiment, we measure the performance of PKM and RGEM for a set of periodic tasks $\tau_0 - \tau_4$ listed in descending fixed priority order in Table 2. Tasks τ_0, τ_2, and τ_3 are single kernel MM applications that are preemptive only under PKM, while τ_1 and τ_4 are LS applications. The performance results are shown in Figure 5 and summarized in Table 3. We show the average response time with 90% confidence intervals and also report the longest observed response time for each task. On average, PKM completes an instance of the highest priority task τ_0 in 17.75 $\pm$ 0.57ms. The maximum observed response time of τ_0 is bounded by 25ms in PKM in Table 3. For every task, PKM substantially reduces the response time compared to RGEM, while bounding the response time in a relatively more reliable manner with considerably smaller fluctuations compared to RGEM as shown in Figure 5. Since high priority tasks finish early in PKM, lower priority tasks experience less frequent preemption during data copies and kernel executions. As a result, the performance of lower priority tasks also enhance. Notably, in RGEM, the maximum response time of τ_0 is 90ms, which is longer than the period of τ_0 (50ms). In contrast, PKM does not show such an undesirable behavior by supporting preemptive kernels unlike RGEM. Moreover, all the task instances finish before the task periods. Overall, PKM significantly decreases the response time and its variations by supporting preemptive kernels and memory copies, while multiplexing memory copies and kernel executions of different tasks.



(a) Linear Search (b) Matrix Multiplication

Figure 4. Priority Inversion vs. Response Time: Response time of a high priority task (1024 x 1024 matrix multiplication with 50ms period) for increasing competing loads



(a) PKM (b) RGEM

Figure 5. Preemptive vs. Non-Preemptive Kernels: Response time of PKM and RGEM

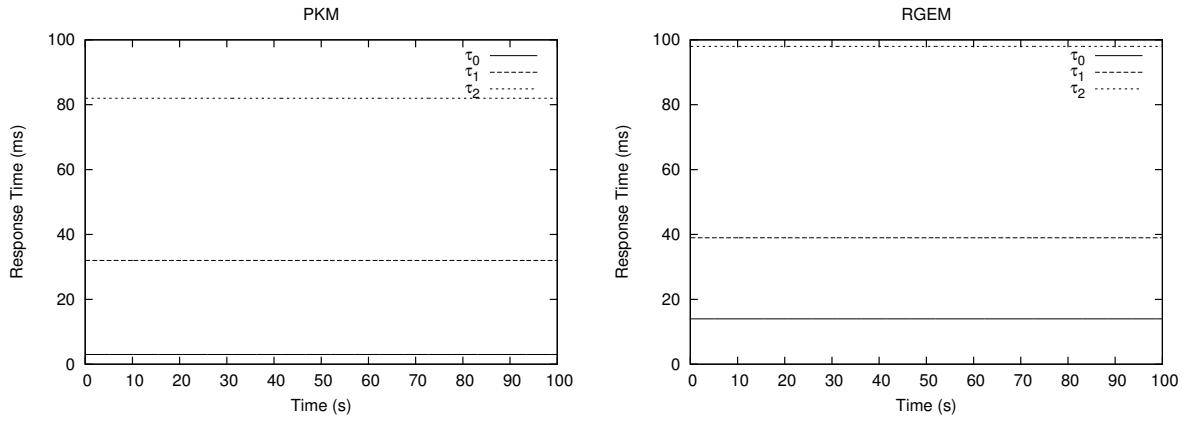
4.3 Response Times under No Priority Inversion

Finally, we compare the performance of PKM and RGEM in terms of memory management. For this purpose, we generate three tasks τ_0, τ_1, and τ_2 listed in descending priority order. A periodic instance of task τ_0 processes a 1024 x 1024 matrix multiplication, while tasks τ_1 and τ_2 are linear search tasks periodically processing 64MB and 128MB data, respectively. They are assigned 50ms, 100ms, and 100ms periods. Since the periods are harmonic, they are released at every 100ms at which the jobs are executed in priority order. Thus, priority inversion due to non-preemptive kernels is eliminated. By doing this, we aim to favor RGEM and evaluate the efficiency of PKM's memory copy mechanism and overlapped processing of memory and kernel operations.

Figure 6 shows the performance results. Given the harmonic tasks, the response time of each task in PKM and RGEM is nearly constant. Thus, the confidence intervals are almost zero in this set of experiments. The response time of the highest priority task τ_0 is 6ms and 14ms in PKM and RGEM, respectively. Also, the response times of τ_1 and τ_2 in PKM is considerably shorter than that in RGEM as shown in Figure 6. Since a kernel does not block for memory transactions of the other tasks and vice versa, PKM shows considerable enhancement in terms of the response time compared to RGEM.

Task	PKM AVG	RGEM AVG	PKM Max	RGEM Max
τ_0	17.75±0.57ms	28.52±3.73ms	25ms	90ms
τ_1	27.36±0.11ms	60.79±6.41ms	29ms	149ms
τ_2	39.75±0.59ms	103.96±8.01ms	47ms	194ms
τ_3	235.54±1.92ms	388.00±0.00ms	248ms	388ms
τ_4	248.03±0.02ms	576.00±0.00ms	249ms	576ms

Table 3. Preemptive vs. Non-Preemptive Kernels: Maximum and average response time with 90% confidence intervals



(a) PKM (b) RGEM

Figure 6. Response time of PKM and RGEM with no priority inversion

5 Related Work

GPUs are employed to support high performance for various applications including scientific applications [7, 3], cryptography [22], intrusion detection [19], bioinfomatics [13], databases [2], and storage systems [1]. However, the detailed architectures of GPUs are not fully known to the public. As a result, scheduling and resource management for real-time support is challenging [10, 11, 5, 6, 16].

TimeGraph [11] is a GPU command scheduler at the device driver level. It supports prioritization and isolation for real-time applications. TimeGraph is executed to schedule every GPU command. RGEM [10] is subject to less overhead, since it is run only when data are copied and kernels are launched. PKM is also executed for data copies and kernel launches. PKM provides enhanced real-time support by supporting preemptive kernel executions in addition to preemptive memory copies. Moreover, PKM overlaps the processing of memory copies and kernels using CUDA streams. It provides these features in the user space leveraging basic CUDA capabilities without requiring any additional support from the underlying device drivers or operating system.

Research efforts have been made to integrate GPUs as part of real-time multiprocessor systems [5, 6]. One of their approaches, called Shared Resource Model, considers GPU executions as critical sections. Another model, called Container Method, provides less pessimistic analysis for real-time scheduling. A brief analysis of the response time bound in RGEM is given in [10]. Also, quality adaptive anytime algorithms are developed to produce lower quality results early, if necessary, to support the timeliness of real-time queries when the GPU is overloaded [14]. Our contribution is providing more efficient methods to support preemptive GPU computing, which can be used as a vehicle to improve the responsiveness and schedulability of real-time GPU tasks. Therefore, our work is complementary to these approaches. For example, the response time bound analysis in [10] can be extended to compute the bound for PKM by considering the impact of preemptive kernels. Our approach could also be integrated with an adaptive scheme, such as [14], to gracefully adapt the quality of service under overload. A thorough investigation of these research issues is reserved for future work.

PTask [21] is a novel approach designed to support GPU resource management via a data flow model in the operating system. GViM [8] is a GPU-accelerated virtual machine (VM) manager that provides GPU resource management through CUDA APIs. It enables VMs to time-share a GPU. Pegasus [9] also provides novel approaches to sharing

a GPU to increase the utilization. Ravi et al. [20] improves GPU-accelerated VM technology by supporting inter-VM concurrent kernel executions. However, research on global CPU-GPU scheduling for sharing a GPU as a coprocessor to increase the performance of multi-processor systems mainly focuses on fairness rather than prioritization [6, 21].

6 Conclusion and Future Work

In this paper, we present a new approach to supporting fully preemptive execution of soft real-time tasks in GPGPUs via preemptive kernel execution and data copies between the host and device. Moreover, our approach simultaneously runs a computational job and memory transaction to reduce the delay. For performance evaluation, we have designed and implemented a prototype system for preemptive data copies and job executions in a GPGPU. The experimental results show that our approach can bound the response times in a reliable manner, while achieving an up to two orders of magnitude shorter response time. In terms of both the average and maximum observed response time, our approach consistently outperformed the tested baselines. In the future, we will further improve PKM, while investigating other research issues such as QoS adaptation of real-time tasks in GPUs and admission control.

References

[1] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, 2008.

[2] P. Bakkum and K. Skadron. Accelerating SQL Database Operations on a GPU with CUDA. In *Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[3] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Transactions on Graphics*, 22:917–924, 2003.

[4] Christopher Urmson et al. Autonomous driving in urban environments: Boss and the Urban Challenge. *Journal of Field Robotics Special Issue on the 2007 DARPA Urban Challenge, Part I*, 25(1):425–466, 2008.

[5] G. Elliott and J. Anderson. Real-Time Multiprocessor Systems with GPUs. In *International Conference on Real-Time and Network Systems*, 2010.

[6] G. Elliott and J. Anderson. Globally Scheduled Real-Time Multiprocessor Systems with GPUs. *Real-Time Systems, special issue on selected papers from the 19th International Conference on Real-Time and Network Systems*, 48(1):34–74, 2012.

[7] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2008.

[8] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-Accelerated Virtual Machines. In *ACM Workshop on System-level Virtualization for High Performance Computing*, 2009.

[9] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *USENIX Annual Technical Conference*, Portland, OR, June 2011.

[10] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines . In *IEEE Real-Time Systems Symposium*, 2011.

[11] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX Annual Technical Conference*, 2011.

[12] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 2008.

[13] S. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 9:1–9, 2008.

[14] R. Mangharam and A. A. Saba. Anytime Algorithms for GPU Architectures. In *IEEE Real-Time Systems Symposium*, 2011.

[15] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[16] J. Nickolls and W. Dally. The GPU Computing Era. *IEEE Micro*, 30(2):56 –69, 2010.

[17] NVIDIA. *CUDA Programming Guide 3.2*. NVIDIA, 2009.

[18] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26:80–113, 2007.

[19] Q. Qian, H. Che, R. Zhang, and M. Xin. The Comparison of the Relative Entropy for Intrusion Detection on CPU and GPU. *Australasian Conference on Information Systems*, 2010.

[20] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework. In *ACM Conference on High Performance Distributed Computing*, 2011.

[21] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *ACM Symposium on Operating Systems Principles*, 2011.

[22] R. Szerwinski and T. Güneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2008.

[23] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, 2011.