RGEM: A Responsive GPGPU Execution Model for Runtime Engines

Shinpei Kato[†] Karthik Lakshmanan[†] Aman Kumar[‡] Mihir Kelkar[‡]
Yutaka Ishikawa[*] Ragunathan (Raj) Rajkumar[‡]

[†]Department of Computer Science, University of California Santa Cruz
[‡]Department of Electrical and Computer Engineering, Carnegie Mellon University
[*]Department of Computer Science, The University of Tokyo

Abstract

General-purpose computing on graphics processing units, also known as GPGPU, is a burgeoning technique to enhance the computation of parallel programs. Applying this technique to real-time applications, however, requires additional support for timeliness of execution. In particular, the non-preemptive nature of GPGPU, associated with copying data to/from the device memory and launching code onto the device, needs to be managed in a timely manner. In this paper, we present a responsive GPGPU execution model (RGEM), which is a user-space runtime solution to protect the response times of high-priority GPGPU tasks from competing workload. RGEM splits a memory-copy transaction into multiple chunks so that preemption points appear at chunk boundaries. It also ensures that only the highest-priority GPGPU task launches code onto the device at any given time, to avoid performance interference caused by concurrent launches. A prototype implementation of an RGEM-based CUDA runtime engine is provided to evaluate the real-world impact of RGEM. Our experiments demonstrate that the response times of high-priority GPGPU tasks can be protected under RGEM, whereas their response times increase in an unbounded fashion without RGEM support, as the data sizes of competing workload increase.

1 Introduction

The graphics processing unit (GPU) has become one of the most powerful platforms for a wide class of parallel programs, embracing the concept of many-core processors. As of 2011, the peak performance of GPUs (in double precision) reaches 1,000 GFLOPS, which is nearly equivalent to ten times that of traditional multi-core processors [31]. Modern GPUs, such as NVIDIA GeForce GTX 580 [19], integrate more than five hundred processing cores on a chip. Such a rapid growth of GPUs is due to recent advances in programming support for general-purpose computing on GPUs, also known as *GPGPU*, which enables GPUs to be used easily for "compute" programs in addition to graphics programs.

GPGPU solutions are increasingly used in many domains. A recent announcement in June 2011 from TOP500 SUPERCOMPUTER SITES [29] disclosed that three of the top five supercomputers use GPUs. Cloud computing services, such as Amazon EC2 [2], also leverage GPGPU to enhance their data center systems. In addition, the benefit of GPGPU for database and storage systems has been demonstrated by the research community [1, 9, 17]. As seen in the trends, GPGPU is primarily developed for high-performance computing, but is also beneficial for embedded real-time computing, particularly well-suited for the current state of the art in cyber-physical systems that compute a large amount of data obtained through sensors in real-time. A new version of the autonomous vehicle developed by Carnegie Mellon University [30], for example, employs four NVIDIA *Fermi* GPUs to enhance its computing capability required to support autonomous driving tasks, such as vision-based perception, motion planning, localization, and navigation. A case study from Stanford's autonomous vehicle "Stanley" demonstrated that GPGPU can accelerate computer vision tasks in autonomous driving by forty times compared to multi-core solutions [28]. More generally speaking, many cyber-physical systems applications that monitor and control physical environments would benefit from GPGPU, given that environmental data can often be processed independently, and the amount of data to be processed is often massive.

Despite many benefits of GPGPU, there exist few studies that explore how to apply GPGPU to real-time applications. In fact, the current GPGPU programming frameworks impose significant limitations for real-time setups, largely attributed to the fact that GPGPU tasks need to copy data between the device memory and the host memory to perform computation. Since this memory-copy transaction is operated through non-preemptive direct memory access (DMA), it could block other device memory accesses requested by high-priority tasks, and the blocking time increases, as the copied data size increases. There is another issue that a piece of GPU-accelerated code, often referred to as a *kernel*, cannot be preempted till it finishes under the current GPGPU solutions, which could also affect high-priority tasks waiting for the GPU. This non-preemptive nature poses a core challenge for real-time GPGPU.

Figure 1 depicts detailed computation costs for a generic GPGPU matrix multiplication program. Both memory-copy and kernel execution costs increase, as the matrix data sizes increase. It should be noted that data-upload usually takes a longer time than data-download in a matrix multiplication of $A[] \times B[] = C[]$, since a program needs to upload $A[]$ and $B[]$ to compute multiplication, while only the resultant values in $C[]$ need to be downloaded. While the memory-copy cost is often dominated by a data size and a bus interface, the kernel execution cost is more dependent on GPU clock frequency, the number of utilized processing cores, and the algorithm itself.

1052-8725/11 $26.00 © 2011 IEEE
DOI 10.1109/RTSS.2011.13

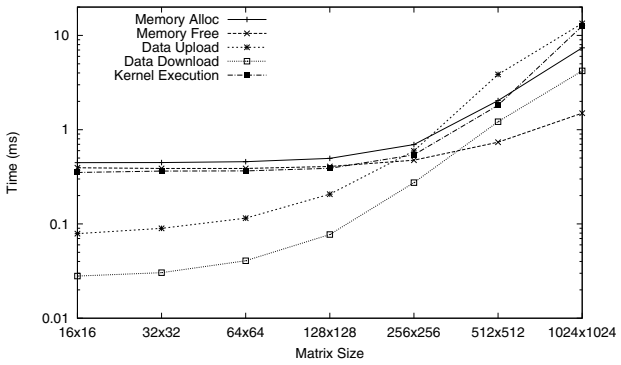


Figure 1. Execution costs for an integer matrix multiplication program on GeForce GTX 480.

In either case, the scale of these execution costs imposed on non-preemptive regions is not trivial. A new GPGPU solution is required for real-time computing.

The contribution of this paper is to develop a responsive GPGPU execution model (RGEM) for improving the response times of high-priority GPGPU tasks in real-time multi-tasking environments. The first feature of RGEM is to split a memory-copy transaction into multiple chunks, providing preemption points at boundaries between the chunks. Hence, the blocking time on the memory-copy transaction is bounded by the length of time taken to copy a chunk of data. The second feature of RGEM is to launch the kernels of different GPGPU tasks one by one based on their priorities, which prevents high-priority GPGPU tasks from getting interfered by concurrent workload once launched. However, the kernel launch itself can still be blocked by preceding low-priority workload launched earlier. This issue is left open for future work. Since RGEM is a user-space runtime solution, neither user applications nor device drivers need modifications. To the best of our knowledge, this is the first piece of work that can protect the response times of GPGPU tasks in real-time multi-tasking environments.

The rest of this paper is organized as follows. We introduce our system model in Section 2. Section 3 provides the design concept of RGEM. Section 4 presents an implementation of a RGEM-based CUDA runtime engine. The advantages of using RGEM are evaluated in Section 5. Section 6 discusses related work. We provide our concluding remarks in Section 7.

2 System Model

The system is composed of a generic multi-core processor and a graphics card. We particularly consider CUDA as the underlying programming model, but the concept of RGEM is applicable to a wide class of GPGPU programming models, such as OpenCL and HMPP. The software stack consists of a device driver, a compiler, and a runtime engine. The compiler generates GPU code and CPU code. The CPU code contains a program to launch the GPU code onto the GPU via the runtime engine and device driver. The GPU code includes at least one kernel, and a set of data used by each kernel is uploaded before and downloaded after the kernel execution. In particular, we assume that GPGPU programs use the following interfaces.

- MemAlloc(size) allocates device memory space of size bytes, and returns a pointer to it. If there is no enough space left for the requested size, the interface call fails.
- MemFree(ptr) frees the device memory space pointed to by ptr, which must have been allocated through the MemAlloc interface.
- MemCopyUpload(dst_addr, src_buf, size) copies data of size bytes from a user-space buffer specified by src_buf in the host memory to the device memory at the address specified by dst_addr. This is a blocking call.
- MemCopyDownload(dst_buf, src_addr, size) copies data of size bytes from the device memory at the address specified by src_addr to a user-space buffer specified by dst_buf in the host memory. This is a blocking call.
- Launch(kernel, arguments) launches the kernel program specified by kernel with the data parameters specified by arguments, which is already loaded as part of the GPU code in the device memory. This is a blocking call.

We assume that GPGPU tasks have fixed priorities, which are by default prioritized over other tasks running on the CPU. They may or may not execute periodically with deadlines. The CPU scheduler in the operating system (OS) dispatches tasks based on their priorities, while the device driver dispatches the requests to access the GPU when received. RGEM is aimed at scheduling these requests before passed to the device driver.

3 System Design

This section presents the design concept of RGEM, which makes high-priority GPGPU tasks responsive in multi-tasking environments. RGEM is a runtime solution to manage GPGPU tasks through the GPGPU programming interfaces introduced in Section 2. Specifically, it uses the MemCopyUpload and the MemCopyDownload interfaces to schedule memory-copy transactions, while the Launch interface to schedule kernel launches, in order to protect the response times of high-priority GPGPU tasks from low-priority interference.

3.1 Memory-Copy Transaction Scheduling

RGEM provides preemptive scheduling of memory-copy transactions. According to the current GPGPU programming frameworks, GPGPU user-space buffers are directly mapped onto OS-space buffers, which are accessible to/from the GPU and the device memory, so that data can be accessed through DMA. Due to this memory-mapped approach, the OS is not aware of memory-copy transactions. RGEM hence manages memory-copy transactions in user-space.

DMA is typically a non-preemptive operation, meaning that blocking times are dependent on the DMA length. This length, however, is not known a priori, and could be infinite in the worst case. For example, malicious and buggy programs could easily produce a large size of memory-copy transactions. The basic idea behind RGEM is to split a non-preemptive memory-copy transaction into multiple chunks to make it preemptive at boundaries between the chunks.

Figure 2 depicts the concept of memory-copy transactions under RGEM. The user buffer is split into multiple chunks by

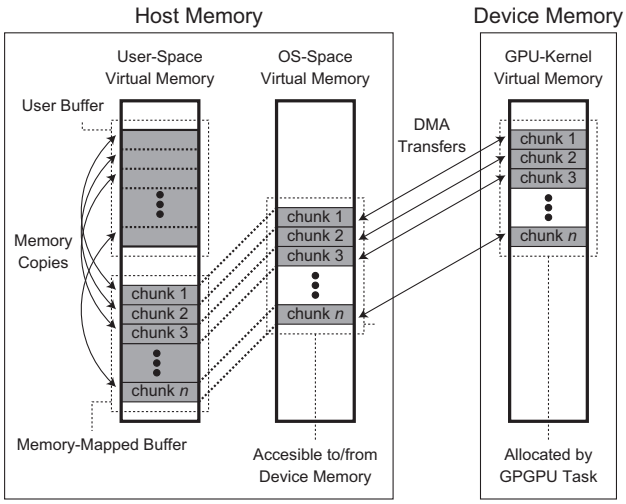


Figure 2. Concept of memory-copy transactions under RGEM.

the runtime engine. This is transparent to the user task. For each chunk k, two memory-copy transactions are atomically coupled: (i) one between the user buffer and the memory-mapped buffer and (ii) the other between the host memory and the device memory via DMA. Making a scheduling decision at a boundary between the chunks, memory-copy transactions become preemptive at the granularity of the given chunk size. The high-level pseudo-code for the MemCopyUpload and the MemCopyDownload interfaces is illustrated in Figure 3. They internally use four RGEM scheduling functions:

- RgemGetChunkSize() returns the chunk size defined by a system designer.
- RgemEnqueue(queue) adds the task into queue.
- RgemDequeue(queue) removes the task from queue.
- RgemSched(queue) suspends the task if queue contains higher-priority tasks.

As illustrated in Figure 3, both the MemCopyUpload and the MemCopyDownload interfaces split the memory-copy transaction into a `nr_chunks` number of chunks. In order to inform the runtime engine that the caller task is performing a memory-copy transaction and hence should be scheduled, the RgemMemEnqueue function is called before the transaction starts. For every chunk copy, the RgemMemSched function is called to create a scheduling point. If there are higher-priority tasks pending for memory-copy transactions, the caller task suspends at this point, and will be awakened later by some other task when its priority becomes the highest among those in the queue. If the operation is to upload, the memory-mapped buffer and its address associated with the source buffer need to be obtained through two helper functions, GetMapBuf() and GetMapAddr(). The memcpy system call is then used to copy data of the chunk size (`ch_size`) from the source buffer to the memory-mapped buffer at the given position (`pos`). This position must be updated every time a chunk is transferred. The data are finally transferred to the device memory using another

```
MemCopyUpload(dst_addr, src_buf, size) {
    ch_size = RgemGetChunkSize();
    nr_chunks = ⌈size/chunk_size⌉;
    pos = 0; copied_size = 0;
    RgemEnqueue(mem_queue);
    for (i = 0; i < nr_chunks; i++) {
        RgemSched(mem_queue);
        copied_size += ch_size;
        if (copied_size > size)
            chunk_size -= copied_size - size;
        src_map = GetMapBuf(src_buf);
        src_addr = GetMapAddr(src_map);
        memcpy(src_map+pos, src_buf+pos, ch_size);
        DMA(dst_addr+pos, src_addr+pos, ch_size);
        pos += ch_size;
    }
    RgemDequeue(mem_queue);
    RgemSched(mem_queue);
}
MemCopyDownload(dst_buf, src_addr, size) {
    ch_size = RgemGetChunkSize();
    nr_chunks = ⌈size/chunk_size⌉;
    pos = 0; copied_size = 0;
    RgemEnqueue(mem_queue);
    for (i = 0; i < nr_chunks; i++) {
        RgemSched(mem_queue);
        copied_size += ch_size;
        if (copied_size > size)
            chunk_size -= copied_size - size;
        dst_map = GetMapBuf(dst_buf);
        dst_addr = GetMapAddr(dst_map);
        DMA(dst_addr+pos, src_addr+pos, ch_size);
        memcpy(dst_buf+pos, dst_map+pos, ch_size);
        pos += ch_size;
    }
    RgemDequeue(mem_queue);
    RgemSched(mem_queue);
}
```

Figure 3. High-level pseudo-code for memory-copy transactions under RGEM.

helper function, DMA(). If the operation is to download, on the other hand, the data must be first transferred from the device memory to the memory-mapped buffer using DMA, and then copied to the user buffer by the memcpy system call. When the transaction completes, the RgemMemDequeue function is called to inform the runtime engine that the caller task need not to be scheduled. Lastly, the RgemMemSched function is called to awaken the highest-priority task in the queue, if any.

3.2 Kernel Launch Scheduling

The second feature of RGEM is to schedule the kernel launches of GPGPU tasks. In the current GPU architectures, as of 2011, multiple kernels from different contexts cannot be executed simultaneously, while those from the same context can upon some architectures, e.g., NVIDIA Fermi. If multiple

```
Launch(kernel, arguments) {
    RgemEnqueue(kern_queue);
    RgemSched(kern_queue);
    SendLaunchCommand(kernel, arguments);
    RgemDequeue(kern_queue);
    RgemSched(kern_queue);
}
```

Figure 4. High-level pseudo-code for kernel launches under RGEM.

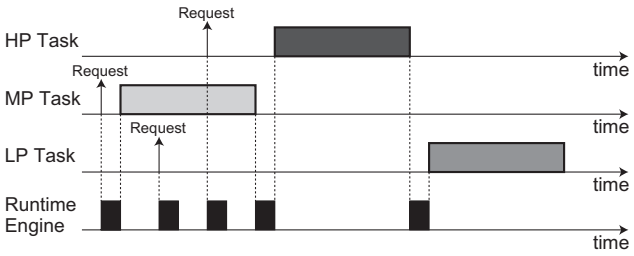


Figure 5. Scheduling example for three kernel launches under RGEM.

kernels from different contexts are loaded at once, they are in turn dispatched by the hardware scheduler. Hence, they can still execute concurrently, if not simultaneously. However, this affects the response times of high-priority GPGPU tasks, since the hardware scheduler does not consider task priorities. The basic idea behind RGEM is to ensure at most one GPGPU task to launch the kernel onto the GPU, and schedule these kernel launches based on task priorities. In fact, such a concept has already been used in *TimeGraph* [16], which is a device-driver solution. RGEM, however, provides this concept at the user-space runtime level, which requires no such specialized device drivers. It should also be noted that RGEM can be used with TimeGraph, as RGEM manages execution flows in user-space, while TimeGraph schedules raw GPU commands in the device driver. Interested readers are encouraged to read [16] about how to schedule GPU commands.

Figure 4 shows the high-level pseudo-code of the Launch interface. It uses the RgemEnqueue, the RgemDequeue, and the RgemSched functions to schedule kernel launches, like memory-copy transaction scheduling presented in Section 3.1. However, the task queues are separate, since memory-copy transactions and kernel launches can be overlapped upon many GPU architectures. It also uses SendLaunchCommand(), a helper function, to launch the kernel already loaded on the GPU with the specified arguments.

Figure 5 illustrates a scheduling example for three GPGPU tasks competing for the GPU to execute their kernels under RGEM. While the medium-priority (MP) task is executing its kernel, the low-priority (LP) and the high-priority (HP) tasks request launching their kernels. If there is no RGEM support, their requests are accepted in their arrival order, and the HP task is blocked by the LP task. However, RGEM enables them to be prioritized when the LP task's kernel completes so that the HP task can respond before the LP task.

4 System Implementation

We now present a prototype implementation of an RGEM-based CUDA runtime engine for Linux. While this paper is focused on CUDA, the concept of RGEM is applicable to other programming models, such as OpenCL and HMPP.

4.1 RGEM Software Stack

Our implementation uses the GPGPU software library and utility tools that we have developed in collaboration with PathScale Inc., where all pieces of software including device drivers, runtime engines, and compilers, are self-made. These library and tools are not open-source, but source-code access may be permitted upon request. More information is available on the PathScale website[1].

In this collaborative project, we have developed a CUDA runtime engine providing both the CUDA Driver API and the CUDA Runtime API [21]. Most standard CUDA programs built by NVIDIA's compiler [22] hence work with our runtime engine. In addition, the performance difference between our solution and NVIDIA's proprietary one is not significant when executing standalone GPGPU programs. Some performance comparisons are reported in [14].

Our runtime engine abstracts GPU device drivers. It can be used with NVIDIA's proprietary driver [20] and PathScale's open-source driver [24]. It is also potentially available with Linux's open-source driver [8]. These drivers manage GPU resources, such as the device memory and execution units, providing *ioctl* interfaces for the user-space runtime engine to access the GPU. For instance, these *ioctl* interfaces are used when creating a GPU context, allocating and freeing device memory space, and launching a kernel. Our runtime engine also abstracts GPU architectures. So far NVIDIA's *Fermi* and *Tesla* architectures have been supported. Other architectures are also planned to be supported in future work.

4.2 RGEM Scheduling Functions

RGEM-based runtime engines require the four scheduling functions introduced in Section 3.1. Although there are several approaches to implementing these functions, we take a user-space inter-process communication (IPC) approach to make our implementation self-contained in user-space. Specifically, we use the POSIX-compliant IPC interfaces to make it widely compatible with many existing systems.

Figure 6 shows the high-level pseudo-code for our RGEM scheduling functions implementation. An argument q for each function indicates an index of the queue to be managed: 0 for memory-copy transactions and 1 for kernel launches. Each task also holds its context ID (0, 1, 2, ...) as a global variable.

Task Descriptors: In order to schedule tasks in user-space, each task must be able to access the status of co-scheduled tasks. Our implementation uses the POSIX *shared memory* API for this purpose. We create a shared memory region to hold an array of task descriptors (`task_descriptors[]` in Figure 6), each of which contains the task's context ID, the priority, the chunk size, and some scheduler-related flags.

[1] http://www.pathscale.com/

```
caller_task = task_descriptors[the context ID];
RgemEnqueue(q) {
    LockQueue(q);
    caller_task.queued[q] = true;
    UnlockQueue(q);
}
RgemDequeue(q) {
    LockQueue(q);
    caller_task.queued[q] = false;
    UnlockQueue(q);
}
RgemSched(q) {
    LockQueue(q);
    caller_task.running[q] = false;
    run_task = GetRunningTask(q);
    hp_task = GetHighestPriorityTask(q);
    if (caller_task.queued[q] == true) {
        if (run_task != null) {
            UnlockQueue(q);
            Suspend();
        } else if (hp_task.prio > caller_task.prio) {
            UnlockQueue(q);
            WakeUp(hp_task);
            Suspend();
        } else
            UnlockQueue(q);
        caller_task.running[q] = true;
    } else if (run_task == null && hp_task != null){
        UnlockQueue(q);
        WakeUp(hp_task);
    } else
        UnlockQueue(q);
}
```

Figure 6. High-level pseudo-code for our RGEM scheduling functions.

Queuing: Our implementation of the RgemEnqueue and the RgemDequeue functions simply sets and clears a flag in the corresponding task's descriptor, and the task is scheduled only if this flag is set. The task descriptors must be accessed exclusively so that the runtime scheduler operates correctly. We use the POSIX *semaphore* API for this purpose, creating a mutex particularly. Tasks must acquire this mutex before they access the task descriptors, and also release it later. The LockQueue() and the UnlockQueue() functions in Figure 6 represent these procedures.

Scheduling: The RgemSched function is a scheduling point. It suspends and awakens tasks based on their priorities. The POSIX standard provides two APIs, *signal* and *message queue*, suitable for these operations. Our implementation uses the message queue, since it is a blocking call that can easily realize suspend-resume operations, while the signal is a non-blocking call that awakens tasks in their signal handlers. More specifically, the Suspend() and the WakeUp() functions listed in Figure 6 use the msgrcv() and the msgsnd() system calls to suspend and awaken tasks respectively. The procedure of the RgemSched functions is as follows.

First, it obtains a task with the "running" state and another with the highest priority in the queue, using helper functions, GetRunningTask() and GetHighestPriorityTask(). The caller task is scheduled only if it exists in the queue. It must suspend regardless of its priority, if some other task already has the running state. If there is no such running task, but some higher-priority tasks exist, the caller task awakens the highest-priority task, and then suspends. The caller task is qualified to execute, only if no task has the running state, and no higher-priority tasks are pending. On the other hand, if the caller task is not queued, it just awakens the highest-priority task, if exists in the queue. This scheduling procedure ensures that at most one task in the queue is allowed to execute on the GPU, thus satisfying the RGEM specification.

Daemon Process: Our implementation creates one user-space daemon process that monitors the tasks running under RGEM. In initialization, it creates the shared memory space, semaphore, and message queue used by the runtime engine. It also repeatedly reads the chunk size of each task written in a specification file ("`/etc/rgem/#PID/chunk_size`"), and reflects it to the value return by the RgemGetChunkSize function. In consequence, system designers can manage the granularity of chunks without applying any modifications to application programs.

5 Evaluation

This section evaluates the advantages of using RGEM, as compared to a traditional GPGPU execution model that does not employ real-time support. Specifically, we compare our prototype RGEM-based CUDA runtime engine to PathScale's CUDA runtime engine whose non-real-time execution logic closely matches that of NVIDIA's proprietary runtime engine. Response times are primarily focused on in our evaluation, since most real-time applications are response-time sensitive. Performance overheads imposed by RGEM are also identified.

5.1 Experimental Setup

Our evaluation is conducted on a machine comprising an NVIDIA's GeForce GTX 480 graphics board and an Intel's Core 2 Duo processor. The Linux kernel Version 2.6.37 and NVIDIA proprietary driver Version 270.41.06 [20] are used as part of the underlying OS. Our microbenchmark programs, *Matrix Multiplication* and *Linear Search*, are written using the CUDA Driver API [21] and compiled by the NVIDIA CUDA Compiler (NVCC) [22]. These GPGPU programs, in fact, play an important role in real-time applications. For example, the Matrix Multiplication program is used in image processing, while the Linear Search program is used in path planning, in an autonomous vehicle [30].

In our experiments, the microbenchmark program tasks are scheduled as *real-time* processes in the Linux kernel, using the `SCHED_FIFO` scheduling policy, to shield them from Linux's background jobs. Among the microbenchmark program tasks, the Matrix Multiplication task is assigned a higher priority, since image processing must meet frame-rates of input data from sensors, while path planning is more exhaustive. No modifications are applied to the Linux kernel, device drivers, and CUDA programs.

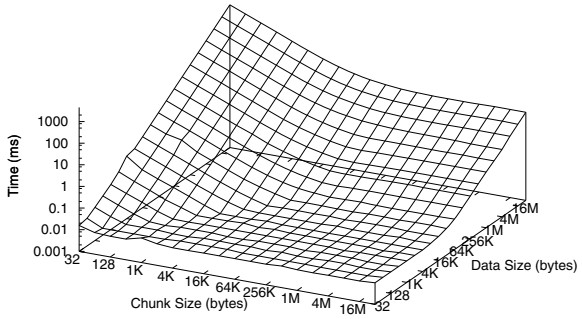


Figure 7. Impact of the chunk size and data size on upload performance.

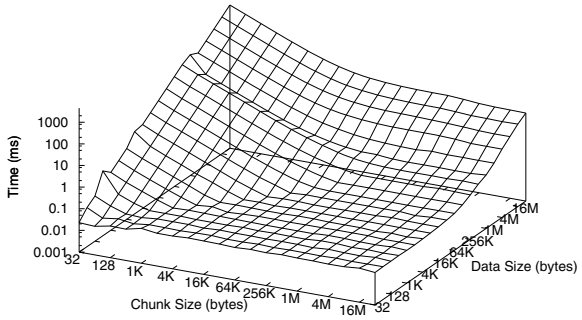


Figure 8. Impact of the chunk size and data size on download performance.

5.2 Experimental Results

Implication of Chunk and Data Sizes: We first identify performance overheads caused by splitting data into chunks, in order to derive an appropriate chunk size. Figure 7 shows the impact of the chunk size and data size on data-upload to the device memory from the host memory, where both the chunk size and data size are set to be powers of two. The time taken to upload the data increases, as the chunk size decreases and/or the data size increases, *i.e.*, the number of chunks increases. Interestingly, the upload cost increases proportionally with a log-scale of ten of the data size when the chunk size is small (*e.g.* 32KB), while it remains almost constant until the data size reaches a certain amount when the chunk size is large (*e.g.*, 16MB). Specifically, a data size of 64KB seems to be a transition boundary. We consider that there might exist some memory transfer block at the hardware level that internally splits transactions around 64KB, though a detailed investigation needs to be conducted. The same effect is observed in data download performance shown in Figure 8.

The above measurement allows us to derive the overhead for copying a chunk of data between the device memory and the host memory. To remove the concern of dependency on any hardware memory transfer blocks, we focus on data sizes greater than 64KB. If there is no overhead, the time taken to copy a certain amount of data between the device memory and

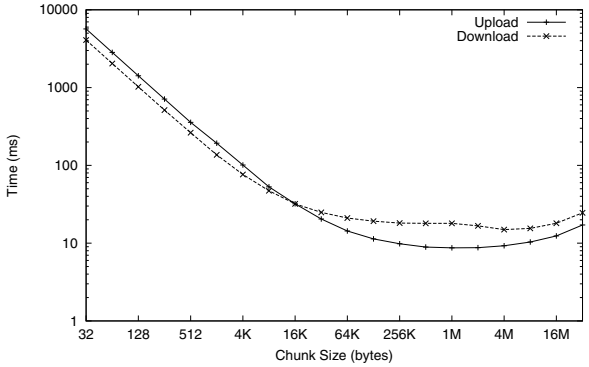


Figure 9. Response times for data-upload and data-download as a function of the chunk size.

the host memory should be the same for any chunk size. Looking at the upload cost for a data size of 16MB in Figure 7, however, it depends on the chunk size. Hence, there exists an overhead. According to our measurements, uploading a data block of 16MB takes about 3892ms with a chunk size of 32B, while it takes about 15ms with a chunk size of 32KB. Let O_U be the overhead cost for uploading one chunk, and U_{16M} be the time taken to upload a data block of 16MB excluding the overhead. Now, the following must hold true:

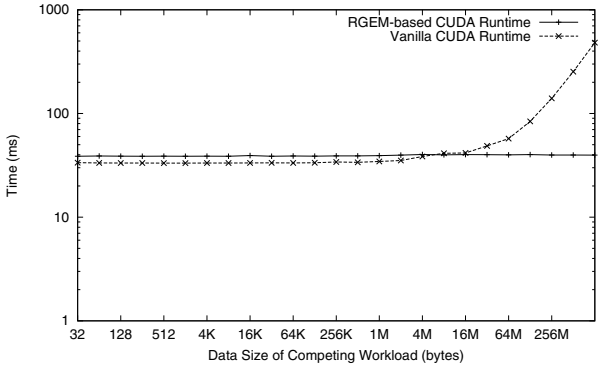
$$\begin{aligned} U_{16M} + 16\text{M}/32 \times O_U &= 3892 \\ U_{16M} + 16\text{M}/32\text{K} \times O_U &= 15 \end{aligned}$$

Hence, $O_U \simeq 0.007ms$ and $U_{16M} \simeq 11.210ms$ are obtained. The time U_1 taken to upload a data block of 1b is also found to be $U_1 = 11.210ms/16\text{M} \simeq 6.681\text{E-}7ms$.

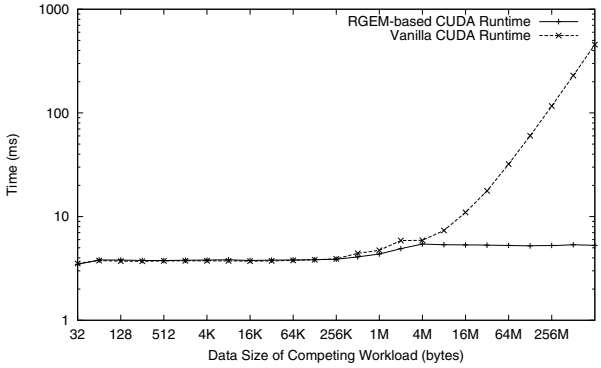
The same analysis can be applied to the download time. Our measurement shows 4088ms for downloading a data block of 16MB with a chunk size of 32B, while it takes 21ms with a chunk size of 32KB. Let O_D be the overhead cost to download one chunk, and D_{16M} and D_1 be the times taken to download a data block of 16MB and 1b, excluding the overhead, respectively. Solving the equations like above, $O_D \simeq 0.008ms$, $D_{16M} \simeq 17.025ms$, and $D_1 \simeq 1.015\text{E-}6ms$ are obtained.

These analytic results point to a conclusion regarding the appropriate chunk size based on the given data size. If the chunk size is too small, the runtime engine would suffer from the overhead, while it can preempt memory-copy transactions in a fine-grained way. If the chunk size is too big, meanwhile, the runtime engine would not be responsive in memory-copy transactions, while the overhead might be negligible. What causes the difference between the upload and download costs also needs additional investigation, but we expect that the write operation to the host memory, *i.e.*, data download, faces more bus contentions, since background jobs and OS daemons may update some data in the host memory due to some inputs from external resources such as networks or disks.

Implication of Response Times and Overheads: We next obtain an appropriate chunk size that addresses the trade-off between response times and overheads in our setup. Figure 9 depicts the response times of the Matrix Multiplication task with regards to data-upload and data-download for different



(a) 1024×1024 integer matrix.



(b) 256×256 integer matrix.

Figure 10. Response times as a function of data sizes of competing workload.

chunk sizes, when the Linear Search task with a data size of 16MB is contending as a low-priority task. If the chunk size is smaller than 1MB, a shorter response time is obtained by a smaller chunk size due to less overhead. However, once it reaches around 1MB, a response time starts increasing as the chunk size increases, since the blocking time introduced by the low-priority Linear Search task has more impact than the overhead. This is an important observation for our RGEM-based runtime engine to address the trade-off between response times and overheads. A difference is observed between the upload and download costs again, and we believe that it is largely attributed to hardware-level issues, including bus contentions and cache effects, which needs more investigations.

For the rest of experiments, we set 1MB to be the chunk size uniformly for all tasks so that we can maximize the advantage of using our RGEM solution.

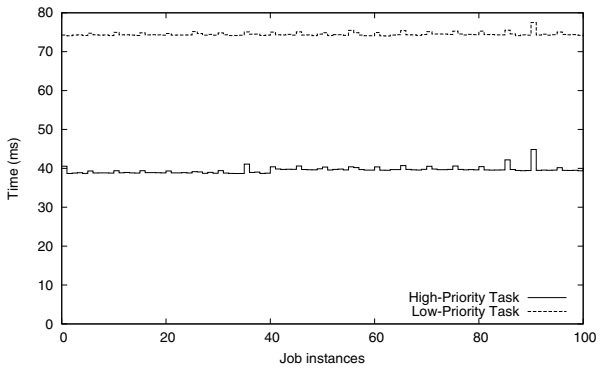
Response-Time Improvement: We now demonstrate how our RGEM-based CUDA runtime engine could improve the response times of high-priority GPGPU tasks, as compared to a vanilla CUDA runtime engine without real-time support.

Figure 10 (a) shows the response times of the high-priority Matrix Multiplication task with a data size of 4MB (1024×1024 integer), competing with the low-priority Linear Search task for multiple cases where the Linear Search task have different data sizes from 32B to 512MB. The average execution time of this Matrix Multiplication task in stand-alone is about 33ms, and it tries to execute periodically at every interval of 50ms. The Linear Search task has variable execution times depending on its data size, and executes an exhaustive search as fast as possible. According to the results, the Matrix Multiplication task running under our RGEM-based CUDA runtime engine is provided with stable execution times around 39ms. However, when running under PathScale's vanilla CUDA runtime, the execution time eventually hits 481ms for the case where the Linear Search task has a data size of 512MB, while it is kept around 34ms if the contending data size is less than 4MB. Hence, our RGEM-based CUDA runtime engine incurs some overhead, but it is less than 15% of the execution time, and the response time of high-priority activities is protected from the interference of low-priority activities. On the other hand, the response time is affected in unbounded fashion without RGEM support, as the contending data size increases.

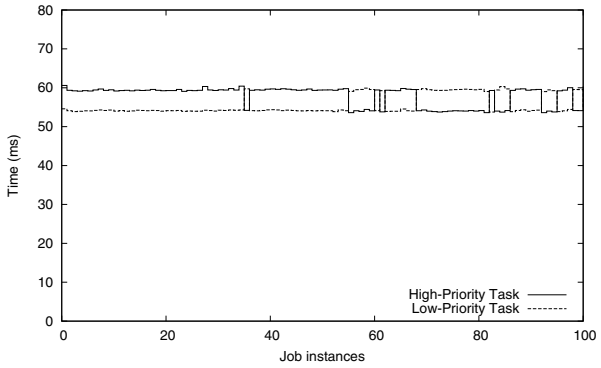
Figure 10 (b) shows the results of a similar setup to the above, but the data size of matrix multiplication is reduced to 256KB (256×256 integer). The overhead imposed by our RGEM-based CUDA runtime engine is now negligible, since a fewer number of chunks is needed to complete memory-copy transactions for matrix multiplication. In addition the blocking time caused by the Linear Search task holds the same, while the execution time of the Matrix Multiplication task itself is much smaller in this setup. Hence, there is more impact on response times, as the contending data size increases.

The above results indicate that RGEM is more beneficial for protecting high-priority GPGPU tasks with a smaller size of data from low-priority workload with a larger size of data. If the contending data size is small enough, RGEM could cause more impact on response times than its prioritization benefit due to the overhead, but the performance penalty is trivial as compared to task execution times, and thus acceptable.

So far we have studied the response times of "high-priority" tasks. We now focus on the interference between multiple tasks for such a case that the response times of "low-priority" tasks are also considered. Figure 11 shows the response times of two instances of the 1024×1024 Matrix Multiplication task on a time (on each job), executing periodically at an interval of 50ms with different priorities. Since the same two tasks are compared, we can study how much they affect each other. According to the results, our RGEM-based CUDA runtime engine clearly prioritizes the two tasks, while they interfere with each other without RGEM support. Since our setup starts the execution of the low-priority task slightly earlier than the high-priority task, the response time of the high-priority task is likely to be longer under the vanilla CUDA runtime engine due to the blocking time introduced on data-download at the end of each job. This blocking time does not appear under our RGEM-based CUDA runtime engine due to its preemption support. It should also be noted that the average sum of the response times of the high- and low-priority tasks for each period is about 114ms under our RGEM-based CUDA runtime engine, while is about 113ms under the vanilla CUDA runtime engine. This observation implies that the overhead imposed by RGEM is acceptable. In consequence, RGEM needs not to compromise throughput very much, though response times can be significantly improved over a traditional GPGPU execution model, if the chunk size is chosen properly.

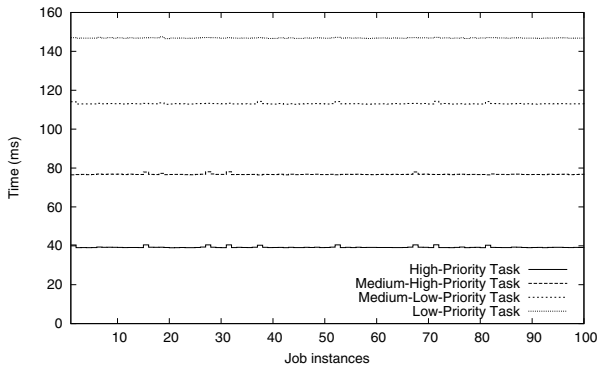


(a) Our RGEM-based CUDA runtime.

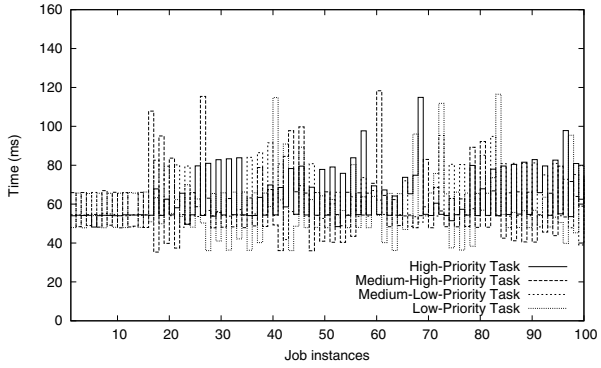


(b) A vanilla CUDA runtime without real-time support.

Figure 11. Interference between two GPGPU tasks with different priorities.



(a) Our RGEM-based CUDA runtime.



(b) A vanilla CUDA runtime without real-time support.

Figure 12. Interference between four GPGPU tasks with different priorities.

In order to evaluate the scalability of our RGEM-based CUDA runtime engine, we next compare the response times of four instances of the same 1024×1024 Matrix Multiplication task, where each CPU core executes two tasks in a partitioned manner. According to the results shown in Figure 12, the four tasks are clearly prioritized under RGEM, whereas there is a significant interference among them without RGEM support. It is also interesting to observe that the response times of high-priority tasks are maintained at almost the same level as the previous case where two tasks run concurrently. This means that our RGEM-based CUDA runtime engine can schedule kernel launches effectively, even though multiple tasks request kernel launches concurrently.

6 Related Work

GPU resource management is not yet well-studied, but is becoming a hot topic nowadays. TimeGraph [15, 16] is a novel approach to real-time GPU resource management using a GPU command scheduler at the device-driver level. GERM [4] also supports a GPU command scheduler at the device-driver level, which manages fairness rather than timeliness. GPU command schedulers could be alternative to RGEM, but pay non-trivial overheads due to queuing and dispatching for all commands. RGEM is more suitable for GPGPU in that tasks are queued and dispatched only when they copy data and launch kernels. There is also a CPU scheduling approach that handles GPU executions as critical sections [7]. PTask [26] is another novel approach to GPU resource management that abstracts GPUs by OSes through a dataflow programming model. These OS approaches, however, require users to use new APIs. Due to the fact that detailed specifications of GPU architectures from vendors are not fully public, the functionality and performance of the existing approaches with low-level device drivers and OSes are limited. Some open-source projects are trying to identify GPU resource management schemes [14].

User-space GPU resource management is more commonly studied, particularly for virtualized systems [6, 12, 13, 18]. These systems, however, rely on the existing runtime solution eventually, and would therefore suffer from blocking times on memory-copy transactions and GPGPU kernel executions in real-time setups. RGEM, on the other hand, is a runtime model with real-time support. It could even provide a reliable runtime engine underlying these GPU-virtualized systems. There are also compile-time and application-level solutions [5, 11, 27], which allow user-space tasks to manage GPU resource usage. Since these solutions require modifications or recompilations of GPGPU programs using specific compilers, algorithms, and APIs, a generality of programming needs to be compromised. RGEM is beneficial in that legacy GPGPU programs work without any modifications; our prototype implementation of RGEM is aligned with the CUDA programming model.

The runtime performance of RGEM depends highly on the underlying IPC mechanisms. While we have used POSIX-compliant IPC mechanisms (shared memory, semaphores, and message queues) in this paper, it is worth investigating how to apply different models and solutions that meet real-time needs more tightly [10, 23, 25].

7 Conclusion

We have presented RGEM, a responsive GPGPU execution model, which improves the response times of high-priority GPGPU tasks in real-time multi-tasking environments. We have also provided a prototype implementation of an RGEM-based CUDA runtime engine to evaluate the benefit of RGEM for real-world GPGPU applications. Our experimental results have demonstrated that the response times of high-priority GPGPU tasks can be successfully protected under RGEM, whereas their response times can be arbitrarily affected by low-priority activities without RGEM support. We believe that the contributions of RGEM are significant in facilitating GPGPU solutions for real-time applications.

In future work, we will explore preemption support for a GPGPU kernel execution to achieve a fully-preemptive GPGPU execution model. This requires additional work on firmware implementations for the microcontroller on the GPU. We will also seek for coordinated resource management with the runtime engine and the device driver.

Acknowledgment

We thank PathScale Inc for having provided their tools and software for our system implementation and experiments.

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 ACM International Symposium on High Performance Distributed Computing*, pages 165–174, 2008.

[2] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-Emptive Scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[4] M. Bautin, A. Dwarakinath, and T. Chiueh. Graphics Engine Resource Management. In *Proc. of the Annual Multimedia Computing and Networking Conference*, 2008.

[5] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao. Dynamic Load Balancing on Single- and Multi-GPU Systems. In *Proc. of the IEEE International Parallel and Distributed Processing Symposium*, 2010.

[6] M. Dowty and J. Sugeman. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[7] G. Elliott and J. Anderson. Real-Time Multiprocessor Systems with GPUs. In *Proc. of the International Conference on Real-Time and Network Systems*, pages 197–206, 2010.

[8] FreeDesktop. Nouveau Open-Source GPU Driver. `http://nouveau.freedesktop.org/`.

[9] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A GPU Accelerated Storage System. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing*, pages 167–178, 2010.

[10] R. Govindan and D. Anderson. Scheduling and IPC Mechanisms for Continuous Media. In *Proc. of the ACM Symposium on Operating Systems Principles*, pages 68–80, 1991.

[11] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling Task Parallelism in the CUDA Scheduler. In *Proc. of the Workshop on Programming Models for Emerging Architectures*, pages 69–76, 2009.

[12] V. Gupta, A. Gavrilovska, N. Tolia, and V. Talwar. GViM: GPU-accelerated Virtual Machines. In *Proc. of the ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24, 2009.

[13] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proc. of the USENIX Annual Technical Conference*, 2011.

[14] S. Kato, S. Brandt, Y. Ishikawa, and R. Rajkumar. Operating Systems Challenges for GPU Resource Management. In *Proc. of the International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–32, 2011.

[15] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar. Resource Sharing in GPU-accelerated Windowing Systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pages 191–200, 2011.

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

[17] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A.D. Nguyen, T. Kaldewey, V.W. Lee, S.A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD/PODS Conference*, 2010.

[18] H.A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-Independent Graphics Acceleration. In *Proc. of the ACM/Usenix International Conference on Virtual Execution Environments*, pages 33–43, 2007.

[19] NVIDIA. GeForce GTX 580. `http://www.nvidia.com/`.

[20] NVIDIA. Linux X64 (AMD64/EM64T) Display Driver. `http://www.nvidia.com/object/linux-display-amd64-270.41.06-driver.html`.

[21] NVIDIA. NVIDIA CUDA Programming Guide Version 3.0. `http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf`.

[22] NVIDIA. NVIDIA CUDA Toolkit Version 3.2. `http://developer.nvidia.com/cuda-toolkit-32-downloads`.

[23] S. Oikawa and H. Tokuda. Efficient Timing Management for User-Level Real-Time Threads. In *Proc. of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pages 27–32, 1995.

[24] PathScale. PSCNV Open-Source GPU Driver. `https://github.com/pathscale/pscnv/`.

[25] R. Rajkumar, M. Gagliardi, and L. Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proc. of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pages 66–75, 2011.

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

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

[28] S. Thrun. GTC Closing Keynote. `http://livesmooth.istreamplanet.com/nvidia100923/`, 2010.

[29] TOP500 supercomputer sites. `http://www.top500.org/`, 2011.

[30] C. Urmson, J. Anhalt, H. Bae, D. Bagnell, C. Baker, R. Bittner, T. Brown, M. Clark, M. Darms, D. Demitrish, J. Dolan, D. Duggins, D. Ferguson, T. Galatali, C. Geyer, M. Gittleman, S. Harbaugh, M. Hebert, T. Howard, S. Kolski, M. Likhachev, B. Litkouhi, A. Kelly, M. McNaughton, N. Miller, J. Nickolaou, K. Peterson, B. Pilnick, R. Rajkumar, P. Rybski, V. Sadekar, B. Salesky, Y-W. Seo, S. Singh, J. Snider, J. Struble, A. Stentz, M. Taylor, W. Whittaker, Z. Wolkowicki, W. Zhang, and J. Ziglar. Autonomous Driving in Urban Environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[31] Wikipedia. FLOPS. `http://wikipedia.org/wiki/FLOPS`.

Appendix – Blocking Delay Analysis

Although RGEM reduces the blocking delays during memory copies and kernel launches, it cannot fully eliminate these delays as they are artifacts of the non-preemptive nature of the

DMA transfer and GPGPU kernel execution. In this section, we provide an analysis to obtain upper bounds on such blocking delays, which can be incorporated into classical fixed-priority scheduling response-time analysis [3]. We assume that host-device memory copies are bi-directional, and the GPGPU kernel execution can be done in parallel with the memory-copy operation. We also assume that the RgemSched interface is performed at the highest priority so that it is not preempted when selecting the task, while it is assigned back the original priority when performing the memory-copy or the kernel-launch operations.

Before describing the analysis, we first introduce the necessary notation and terminology below.

- τ_i represents a task with period T_i.
- UT_i and DT_i denote the total amount of data to be uploaded and downloaded by each instance of task τ_i respectively.
- CS denotes the chunk size specified by RGEM.
- $EQ_i(q)$ and $DQ_i(q)$ denote upper bounds on the the time required to enqueue and dequeue task τ_i to queue q respectively.
- q_m denotes the memory-copy queue.
- q_k denotes the kernel-launch queue.
- $UL(b)$ and $DL(b)$ denote upper bounds on the time taken to upload and download a data block of b bytes respectively.
- O denotes an upper bound on the overhead for setting up a single DMA transfer.
- S denotes an upper bound on the overhead of a single scheduler invocation, which includes the time taken to pick the highest-priority task.
- χ is an upper bound on the context switching cost.
- KN_i is the total number of kernels launched by τ_i.
- $KC_{i,j}$ is the worst-case execution time of the j^{th} GPGPU kernel launched by task τ_i.
- KC_i denotes the total worst-case execution time of all the GPGPU kernels launched by each job of task τ_i, i.e., $KC_i = \sum_{j=1}^{KN_i} KC_{i,j}$

First, we develop an upper bound on response time $RU_i(b)$ for task τ_i to upload b bytes to device memory.

Consider that there are n tasks in the system, τ_1 through τ_n, which could upload data to device memory, including the task τ_i under consideration. Each job of a task in GPGPU applications is typically composed of data upload to device memory, followed by a series of kernel launches, and completes with data download from device memory. Without loss of generality, we consider the tasks to be arranged in non-decreasing order of periods and increasing order of priorities.

We can compute $RU_i(b)$ as:

$$RU_i(b) = EQ_i(q_m) + \lceil \frac{b}{CS} \rceil (S + O + BU_i + UL(CS) + \chi) + DQ_i(q_m) \quad (1)$$

BU_i is an upper bound on the delay incurred by task τ_i for a single chunk upload. Equation 1 follows from the pseudo-code of the memory-copy operation under RGEM given in Figure 3.

An upper bound BU_i is computed iteratively using the convergence ($BU_i^0 = UL(CS)$):

$$BU_i^{n+1} = UL(CS) + \sum_{h=1}^{i-1} \lceil \frac{BU_i^n}{T_h} + 1 \rceil \lceil \frac{UT_h}{CS} \rceil (S + O + UL(CS) + \chi)$$

This follows from the fact that there could already be a non-preemptive memory upload of at most chunk size CS from a lower-priority task happening when τ_i requests its memory upload, which results in a blocking term of $UL(CS)$. The second term captures the interference from the memory uploads of higher-priority tasks τ_h.

We can similarly compute an upper bound on response time $RD_i(b)$ for task τ_i to download a data block of b bytes from device memory to host memory.

$$RD_i(b) = EQ_i(q_m) + \lceil \frac{b}{CS} \rceil (S + O + BD_i + DL(CS) + \chi) + DQ_i(q_m) \quad (2)$$

BD_i is an upper bound on the delay incurred by task τ_i for a single chunk download. Equation 2 also follows from the pseudo-code of the memory-copy operation under RGEM given in Figure 3.

An upper bound BD_i is computed iteratively using the convergence ($BD_i^0 = DL(CS)$):

$$BD_i^{n+1} = DL(CS) + \sum_{h=1}^{i-1} \lceil \frac{BD_i^n}{T_h} + 1 \rceil \lceil \frac{DT_h}{CS} \rceil (S + O + DL(CS) + \chi)$$

This follows from the fact that there could already be a non-preemptive memory download of at most chunk size CS from a lower-priority task happening when τ_i requests its memory download, which results in a blocking term of $DL(CS)$. The second term captures the interference from the memory uploads of higher-priority tasks τ_h.

Now, we can similarly derive an upper bound on the response time RK_i for the cumulative kernel execution times for a task τ_i. Note that the individual kernel executions themselves are non-preemptive, similar to the memory transfer operations.

$$RK_i = EQ_i(q_k) + \sum_{j=1}^{KN_i} (S + O + BK_i + KC_{i,j} + \chi) + DQ_i(q_k) \quad (3)$$

BK_i is an upper bound on the kernel launch delay incurred by task τ_i for a single kernel. An upper bound on BK_i is obtained iteratively using the convergence ($BK_i^0 = \max_{l=(i+1)}^{n} (\max_{j=1}^{KN_l} KT_{l,j})$):

$$BK_i^{n+1} = \max_{l=(i+1)}^{n} (\max_{j=1}^{KN_l} KT_{l,k}) + \sum_{h=1}^{i-1} \lceil \frac{BK_i^n}{T_h} + 1 \rceil (S + O + KC_h + \chi)$$

This follows from the fact that when τ_i issues the kernel launch, there could be a kernel from a lower priority task τ_l that is already executing on the GPU in a non-preemptive fashion. We need to consider the worst case of such a blocking and then compute an upper bound on the maximum interference from the higher priority tasks, as captured by the second term.