Response-Time Bounds for Concurrent GPU Scheduling

Ming Yang and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

***Abstract*—Graphics processing units (GPUs) have been receiving increasing attention in the real-time systems community as a potential solution for hosting workloads like those found in autonomous-driving use cases that require significant computational capacity. Allowing multiple programs to access a GPU concurrently can enable the GPU to be more efficiently utilized, if each individual program is incapable of occupying all GPU resources. In this work, we summarize the basic scheduling rules for concurrent GPU scheduling in NVIDIA GPUs. We define a task model for GPU scheduling based on these scheduling rules. In ongoing work, we are attempting to obtain response-time bounds for tasks under this model.**

I. Introduction

Graphics Processing Units (GPUs) have been receiving increasing attention in the real-time systems community as a means for accelerating computationally intensive workloads. Some of the more promising use cases for GPUs can be found with respect to autonomous driving, where image- and sensor-processing computations must be supported. Several GPU management frameworks have been created that provide real-time guarantees for tasks using a GPU. These frameworks ([2], for example) typically view a GPU as a "black box" that can be accessed by only one task at a time. However, with GPU core counts continually increasing, a single small-scale kernel (*i.e.*, GPU program) may not fully utilize all of a GPU's resources. For example, average resource usage for programs in the Parboil2 suit is only around 30% [7]. To avoid GPU-related capacity loss, sharing a single GPU among multiple small kernels becomes necessary. To address this issue of GPU under-utilization, NVIDIA GPU introduced a feature called *concurrent kernel execution (CKE)* in the Fermi GPU architecture. This feature allows kernels from the same *GPU context* to execute concurrently on a GPU.

In this paper, we summarize the concurrent GPU scheduling rules used on NVIDIA GPUs, as verified in prior work by our group using synthetic benchmarks [5]. As we shall see, these scheduling rules provide a notion of parallel execution that is a hybrid of the gang task model [6] and the malleable task model [1]. In the gang model, each task is multi-threaded and all of a task's threads must be scheduled to run together. The malleable task model is similar, except that a task's threads may execute on any available processors and do not have to commence execution together. In concurrent GPU scheduling, both of these notions of execution occur together, but at different granularities. Such a combination introduces challenges with respect to real-time schedulability analysis. In this paper, we formally present a task model for concurrent GPU scheduling based on the above-mentioned rules, and consider the issue of deriving response-time bounds under this model.

II. Background

We limit attention in this paper to NVIDIA GPUs.

CUDA programming. A program using a GPU uses the *compute unified device architecture (CUDA) API* provided by NVIDIA [4]. Using CUDA, parallelism-related attributes of kernels can be specified. Each kernel is issued as a group of *threads* that execute on a GPU. Threads of one kernel are arranged into multiple same-size *blocks*. Each kernel is a *grid* of such blocks. The number of threads per block (*block size*) and the number of blocks per grid (*grid size*) of a kernel are defined via parameters passed to the CUDA call that launches that kernel.

Concurrent kernel execution. With concurrent kernel execution, multiple kernels from the same GPU context are allowed to execute concurrently on a GPU. Multiple CPU threads[1] within the same CPU process usually share a single *GPU context*, which is conceptually a virtual address space in the GPU hardware. A GPU accepts kernels for execution based on the available thread resources on the GPU and the block size of the kernel.

GPUs treat *blocks* as schedulable entities. All threads in a block are scheduled as a gang. A GPU has a limited number of threads. When a block is scheduled, that block occupies a number of threads equal to the block size of the corresponding kernel. A block cannot be selected for scheduling unless the GPU has a sufficient number of unoccupied threads. Different blocks of the same kernel may be scheduled at different times. At this granularity, GPU scheduling is more like the malleable task model.

Streams and queues. Programmers can define precedence relationships among kernels using *streams*. A stream contains a sequence of kernels that execute in order; kernels from different streams can execute out of order or concurrently. Programmers are suggested to not rely on any ordering of kernels executed in different streams for program correctness [4]. A GPU uses *hardware work*

[1] Not to be confused with the GPU *hardware* threads mentioned above.

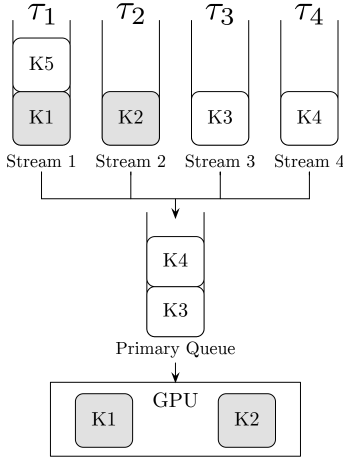

Fig. 1: Streams and queues

queues in scheduling kernels. All NVIDIA GPUs since the Kepler architecture have up to 32 such queues. In this work, we therefore assume that 32 queues are available and the number of streams is at most 32. As shown in Fig. 1, we let each task τ_i have a unique stream. Thus, in the rest of this paper, the terms task, stream, and hardware work queue can be used interchangeably without ambiguity. Each GPU context has a *primary queue* (or channel [9]) shared among streams. The enqueue and dequeue rules for these queues are described in the next section.

III. GPU SCHEDULING

In this section, we summarize the GPU scheduling rules for concurrent kernels launched from one multi-threaded CPU process. The acceptance of a kernel for execution by the GPU depends on when the kernel becomes the head of its stream queue and the availability of the GPU resources it needs to commence execution. Generally, these resources include GPU threads, registers, and shared memory, but we only consider thread resources in this work. Current GPUs limit block sizes to be at most 1,024 threads. Also, the NVIDIA Jetson TX1 [3], which is marketed for embedded use cases, requires the total number of occupied threads to be at most 4,096. We will assume both of these limits on thread resources in this work.

The basic GPU scheduling rules are as follows:

R1. A block of a kernel in the primary queue is assigned to the GPU for execution if:
   a) that kernel is at the head of the primary queue, and
   b) the number of unoccupied threads on the GPU is at least the kernel's block size.

R2. A kernel is enqueued on the primary queue when it becomes the head of its stream queue.

Example 1. Consider task set $\{\tau_1, \tau_2, \tau_3, \tau_4\}$, where each task τ_i has a unique stream, Stream i. As shown in Fig. 1, the kernels are labeled increasingly by their arrival orders as heads of their respective stream queues. A schedule for these kernels is shown in Fig. 2. The GPU is fully idle with

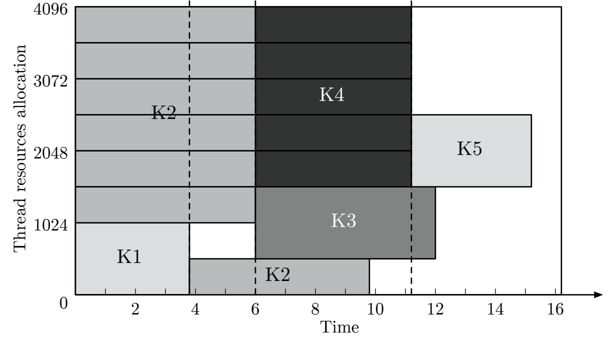

Fig. 2: Schedule example

4,096 unoccupied threads at the beginning. Then K1 starts executing and occupies 1,024 threads—it consists of one block and has a block size of 1,024. At the same time, K2 begins executing concurrently. K2 has seven blocks and a block size of 512 threads, so six of its seven blocks can execute concurrently with K1. The queues in Fig. 1 show the state of the system at this point in time. Note that K1 and K2 are no longer in the primary queue. They will remain in their stream queues until they complete. Their stream-queue entries are shaded to indicate they are currently executing. When K1 finishes, by Rule R1a, K3 is still blocked waiting for K2. Once the last block of K2 is assigned to the GPU, K3 is blocked until time 6, by Rule R1b. K5 is not inserted into the primary queue until K1 finishes, by Rule R2. For K5, the condition of Rule R1 is satisfied when K4 finishes.

These rules are a reworking of the stream scheduling rules for the Fermi GPU architecture defined in [8], with the additional consideration of the Hyper-Q feature introduced since the Kepler GPU architecture. Comprehensive experiment verifying these rules on the NVIDIA TX1 have been presented in prior work by our group [5].

In the next section, we present a task model based on the rules above. In this task model, we only consider the kernel at the head of each task's stream queue. Any kernels behind that at the head can be viewed as not being released yet. With this assumption, we can summarize the rules above in one property:

(P1) Kernels are executed in FIFO order by release time. A kernel cannot be scheduled until all earlier kernels are completely scheduled, *i.e.*, all blocks of the earlier kernels have been assigned to the GPU.

IV. SYSTEM MODEL

We consider the problem of scheduling a set τ of n independent sporadic *tasks* $\{\tau_1, \tau_2, \tau_3, \cdots, \tau_n\}$ on a single GPU. Each task τ_i releases *jobs* (kernels) repeatedly with minimum separation time between consecutive jobs equal to T_i, which is called the *period* of τ_i. We denote the total execution time workload for τ_i as C_i. The *utilization* of task τ_i is defined as $u_i = C_i/T_i$. The *total utilization* of the task system τ is defined as $U_{sum} = \sum_{\tau_i \in \tau} u_i$.

The $k^{th}(k \geq 1)$ job of τ_i is denoted $\tau_{i,k}$. The release time of the job $\tau_{i,k}$ is denoted $r_{i,k}$ and its (absolute) deadline $d_{i,k}$ is computed as $r_{i,k} + T_i$. Denoting the completion time of $\tau_{i,k}$ as $f_{i,k}$, its *response time* is defined as $R_{i,k} = f_{i,k} - r_{i,k}$. A task's response time is the maximum of the response time of any of its jobs. Each task is sequential, *i.e.*, jobs cannot be executed concurrently (in our model, jobs correspond to kernel executions, and a task submits all of its kernels to one stream, which orders those kernels). According to Property P1, jobs are prioritized by their release times with arbitrary tie-breaking.

As discussed in Sec. II, the degree of parallelism afforded to a job is determined by configuring parameters that define its thread-block hierarchy. Hence, we associate two more parameters with each task τ_i, namely g_i and b_i: g_i denotes the *grid size* of each job of τ_i, *i.e.*, the number of blocks in the grid of the job (we assume each job has only one grid, so this is equivalent to the number of blocks per job); b_i denotes the *block size* of the job, *i.e.*, the number of threads in each block. For brevity, we denote τ_i's parameters using the notation $\tau_i = (C_i, T_i, g_i, b_i)$.

All blocks of a job $\tau_{i,k}$ complete the same amount of work, so each block has a workload of $\frac{C_i}{g_i}$. Each block executes as b_i parallel GPU threads, so it finishes within $\frac{C_i}{g_i b_i}$ time units.

We denote the maximum block size limited by the GPU as b_{lim}, which is equal to $1,024$ for all currently available NVIDIA GPUs. We denote the largest block size of the task system τ as b_{max}. We denote the total number of GPU threads on the GPU as m.

Example 2. Consider the task set $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ from Example. 1. We have $\tau_1 = (1024 \cdot 3.8, 8, 1, 1024)$, $\tau_2 = (3584 \cdot 6, 16, 7, 512)$, $\tau_3 = (1024 \cdot 6, 17, 1, 1024)$, and $\tau_4 = (2560 \cdot 11.2, 18, 5, 512)$. Note that tasks can have utilization larger than 1.0, as a task can occupy multiple GPU threads. For instance, in the example here, $u_1 = 486.4$. This task set has total utilization $U_{sum} = 3784.7 < 4096 = m$.

The task model above is admittedly simplistic in several ways. As noted earlier, we are currently only considering available GPU threads as resources required by kernels and ignoring GPU registers and shared memory. Also, actual GPU-using workloads consist of a mixture of CPU code and GPU code, and typically, a task would submit kernels that are not all the same. Our eventual goal is to consider a richer model that is more realistic, but for now, we are restricting our attention to the simple model defined above.

V. Current progress

Total utilization restriction. Due to the usage of gang scheduling at the block level, a block has to be postponed if there are not enough unoccupied GPU threads to accommodate it. The following theorem shows that this leads to the need for a utilization restriction if response times are to be bounded. Let b_{max} denote the largest block size in the system. Assume that $b_{max} > 1$, for otherwise

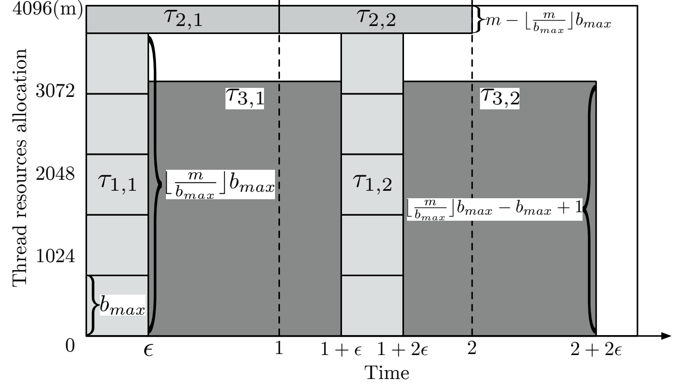

Fig. 3: Unbounded response time for τ_3.

($b_{max} = 1$), our task model becomes the malleable task model, for which responses can be bounded with no utilization restriction.

Theorem 1. There exists a task system τ for which $U_{sum} \geq m - b_{max} + 1$ such that response times are unbounded.

Proof. We construct such a task system $\tau = \{\tau_1, \tau_2, \tau_3\}$. Let $k = \lfloor \frac{m}{b_{max}} \rfloor b_{max}$, which is the maximum number of threads for a job with block size b_{max} that can be scheduled at a time. Define $\tau_1 = (\epsilon \cdot k, 1, \frac{k}{b_{max}}, b_{max})$, $\tau_2 = (m - k, 1, m - k, 1)$, and $\tau_3 = (k - b_{max} + 1, 1, k - b_{max} + 1, 1)$. Note that $U_{sum} = \frac{\epsilon \cdot k}{1} + \frac{m-k}{1} + \frac{k - b_{max}+1}{1} = \epsilon \cdot k + m - b_{max} + 1$, so $\lim_{\epsilon \to 0+} U_{sum} = m - b_{max} + 1$. As shown in Fig. 3, $\tau_{3,1}$ is blocked for time ϵ, by Rule R1b, because all GPU threads are fully occupied by $\tau_{1,1}$ and $\tau_{2,1}$ in time interval $[0, \epsilon)$. $\tau_{3,1}$ finishes at time $1 + \epsilon$ with response time $1 + \epsilon$. Considering later jobs, the response time of τ_3 keeps increasing. Specifically, although the $\lim_{\epsilon \to 0+} U_{sum} = m - b_{max} + 1$, job $\tau_{3,j}$'s response time is $R_{3,j} = 1 + \epsilon \cdot j$, where $j \geq 1$. ☐

VI. Conclusions

In this paper, we defined a task model for GPU scheduling and showed that response times can be unbounded in this model if total utilization exceeds a certain limit. In ongoing work, we are attempting to prove that response times are indeed bounded in this model if total utilization is at most this limit.

References

[1] S. Collette, L. Cucu, and J. Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 2008.

[2] G. Elliott, B. Ward, and J. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS '13*.

[3] NVIDIA. NVIDIA Tegra X1 Whitepaper. Online at link, 2015.

[4] NVIDIA. NVIDIA CUDA Toolkit Documentation. Online at link, 2017.

[5] N. Otterness, M. Yang, T. Amert, J. Anderson, and F.D. Smith. Inferring the scheduling policies of an embedded CUDA GPU. In *OSPERT '17*, in submission.

[6] J. K. Ousterhout et al. Scheduling techniques for concurrent systems. In *ICDCS '82*.

[7] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *ACM SIGPLAN Notices*, 2013.

[8] S. Rennich. Webinar: CUDA C/C++ streams and concurrency. Online at link, 2011.

[9] H. Zhou, G. Tong, and C. Liu. Gpes: a preemptive execution system for gpgpu computing. In *RTAS '15*.