Globally Scheduled Real-Time Multiprocessor Systems with GPUs [*]

Glenn A. Elliott and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Graphics processing units, GPUs, are powerful processors that can offer significant performance advantages over traditional CPUs. The last decade has seen rapid advancement in GPU computational power and generality. Recent technologies make it possible to use GPUs as co-processors to CPUs. The performance advantages of GPUs can be great, often outperforming traditional CPUs by orders of magnitude. While the motivations for developing systems with GPUs are clear, little research in the real-time systems field has been done to integrate GPUs into real-time multiprocessor systems. We present two real-time analysis methods, addressing real-world platform constraints, for such an integration into a soft real-time multiprocessor system and show that a GPU can be exploited to achieve greater levels of total system performance.

1 Introduction

The computer hardware industry has experienced rapid growth in the graphics hardware market during this past decade, with fierce competition driving feature development and increased hardware performance. One important advancement during this time was the programmable graphics pipeline. Such pipelines allow program code, which is executed on graphics hardware, to interpret and render graphics data. Soon after its release, the generality of the programmable pipeline was quickly adapted to solve non-graphics-related problems. However, in early approaches, computations had to be transformed into graphics-like problems that a graphics processing unit (GPU) could understand. Recognizing the advantages of general purpose computing on a GPU, lan-

[*]Work supported by NSF grants CNS 0834270, CNS 0834132, and CNS 1016954; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

guage extensions and runtime environments were released by major graphics hardware vendors and software producers to allow general purpose programs to be run on graphics hardware without transformation to graphics-like problems.[1]

Today, GPUs can be used to efficiently handle data-parallel compute-intensive problems and have been utilized in applications such as cryptology [25], supercomputing [3], finance [11], ray-tracing [13], medical imaging [39], video processing [36], and many others.

There are strong motivations for utilizing GPUs in real-time systems to perform general purpose (non-graphical) computation. Most importantly, their use can significantly increase computational performance. For example, in terms of theoretical floating-point performance, GPUs offer greater capabilities than traditional CPUs. This is illustrated in Fig. 1, which depicts the trends in growth of floating-point performance between Intel CPUs and NVIDIA GPUs over much of the past decade [6, 8, 35]. Growth in raw floating-point performance does not necessarily translate to equal gains in performance for actual applications. However, a review of published research shows that performance increases commonly range from 4x to 20x [4], though increases of up to 1000x are possible in some problem domains [12]. Tasks accelerated by GPUs may execute at higher frequencies or perform more computation per unit time, possibly improving system responsiveness or accuracy.

GPUs can also carry out computations at a fraction of the power needed by traditional CPUs. This is an ideal feature for embedded and cyber-physical systems. Further power efficiency improvements can be expected as processor manufacturers move to integrate GPUs in on-chip designs [1, 7]. On-chip designs may also signify a fundamental architectural shift in commodity processors. Like the shift to multicore, it appears that the availability of a GPU may soon be as common as multicore is today. This further motivates us to investigate the use of GPUs in real-time systems. However, it is not immediately self-evident how this should be done given the unique characteristics of a GPU in a host system.

[1]Notable platforms include the Compute Unified Device Architecture (CUDA) from NVIDIA [5], Stream from AMD/ATI [2], OpenCL from Apple and the Khronos Group [10], and DirectCompute from Microsoft [9].

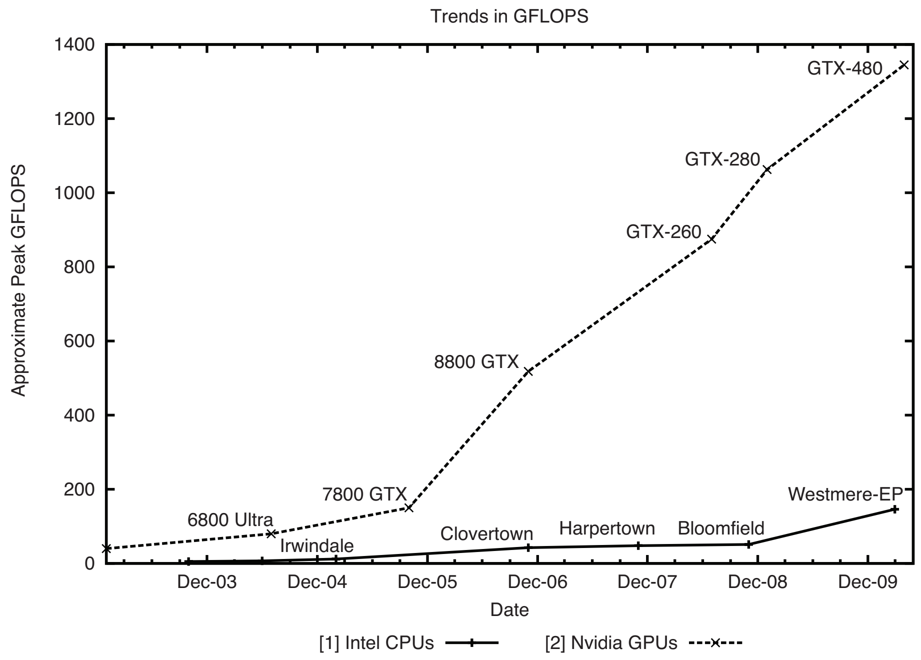


Figure 1: Historical trends in processor performance in terms of approximate peak floating-point operations per second (FLOPS).

A GPU that is used for computation is an additional processor that is interfaced to the host system as an I/O device, even in on-chip architectures. An I/O-interfaced accelerator co-processor, like a GPU or digital signal processor, when used in a real-time system, is unlike a non-accelerator I/O device. In work on real-time systems, the use of non-accelerator devices, such as disks or network interfaces, has been researched extensively [26], with issues such as contention resolution and I/O response time being the primary focus. While these are also concerns for GPUs, the role of the device in the system is different. A real-time system that reads a file from a disk or sends a packet out on a network uses these devices to perform a functional requirement of the system itself. Further, these actions merely cause delays in execution on the CPU; the operations themselves do not affect the actual amount of CPU computation that must be performed. This is not the case for a GPU co-processor as its use accelerates work that could have been carried out by a CPU and does not realize a new functional feature for the system. The performance of a real-time system with a GPU co-processor is dependent upon three inter-related design aspects: how traditional device issues (such as contention) are resolved; the extent to which the GPU is utilized; and the gains in

CPU availability achieved by offloading work onto the GPU.

In this paper, we consider the use of GPUs to perform general purpose computation in *soft* real-time multiprocessor systems, where processing deadlines may be missed but deadline tardiness must be bounded. Our focus on soft real-time systems is partially motivated by the prevalence of application domains where soft real-time processing is adequate. Such a focus is further motivated by fundamental limitations that negatively impact *hard* real-time system design on multiprocessors. In the multiprocessor case, effective timing analysis tools to compute worst-case execution times are lacking due to hardware complexities such as shared caches. Also, in the hard real-time case, the use of non-optimal scheduling algorithms can result in significant utilization loss when checking schedulability, while optimal algorithms have high runtime overheads. In contrast, many global scheduling algorithms are capable of ensuring bounded deadline tardiness in soft real-time systems with no utilization loss and with acceptable runtime overheads. One such algorithm is the *global earliest-deadline-first* (G-EDF) algorithm [20]. Under G-EDF, all schedulable threads of execution share a single ready-queue for all CPUs and threads may migrate between them. In contrast, under partitioned schedulers, threads of execution are statically assigned to individual CPUs and no migrations are allowed.

As G-EDF can be applied to ensure bounded tardiness with no utilization loss in systems without a GPU, we consider it as a candidate scheduler for GPU-enabled systems. We note however, that existing G-EDF analysis has its limitations. Specifically, most analysis is *suspension-oblivious* [17], in that it treats any self-suspension (be it blocking to obtain a lock or waiting time to complete an I/O transaction) as execution time on a CPU. In contrast, *suspension-aware* analysis allows part, or all, of a self-suspension to be treated as available CPU time. Analytical methods exist for *partitioned earliest-deadline-first* (P-EDF) and *partitioned rate-monotonic* (P-RM) scheduling that are suspension-aware, so these scheduling approaches may also be good candidate scheduling algorithms for systems with a GPU. However, partitioned approaches also have limitations. Firstly, partitioned schedulers may suffer utilization loss due to bin-packing-like issues.

In pathological cases, this loss can be nearly as great as 50%. P-RM also suffers additional utilization loss as rate-monotonic scheduling is not optimal on a uniprocessor under most real-time task models. Secondly, systems with a resource shared among partitions often must use a form of *priority-boosting* to provide reasonable bounds on the time a computation may wait before acquiring the shared resource. Priority-boosting temporarily grants the holder of a resource maximum scheduling priority, which may negatively affect the timely execution of work which would normally have been scheduled, including work that does not even require the shared resource. Lastly, it is not entirely clear what an optimal partitioning for workloads using a GPU may be. Should work requiring a GPU be isolated in one partition or scattered amongst them? For these reasons, we defer an investigation of partitioned scheduling for systems with a GPU, which would be non-trivial, to future work in order to keep our initial study tractable.

The use of G-EDF scheduling, and associated suspension-oblivious analysis, implies that the interval of time a task suspends from a CPU to execute on a GPU must also be charged as execution on a CPU. Under these conditions, it appears that a GPU may be useless if work cannot be offloaded from the CPUs. However, a GPU is an *accelerator* co-processor; it can perform more work per unit time than can be done by a CPU. Therefore, there may still be benefits to using a GPU even if CPU execution charges must mask suspensions. In this paper, we determine the usefulness of a GPU in a soft real-time multiprocessor system by answering the following two questions: (Q1) How much faster than a CPU must a GPU be to overcome suspension-oblivious penalties and schedule more work than a CPU-only system? And, (Q2) how much work should be offloaded onto a GPU to make the most efficient use of both the system CPUs and GPU?

Related Work. The problem of arbitrating access to a shared GPU resource has been considered in both real-time and non-real-time domains. In studies concerning graphical displays, prior studies have focused on the behaviors of the platformÕs window manager, such as the ubiquitous X server. In older systems, graphics operations are serialized through the window manager process, indirectly mediating access to the GPU. Non-real-time and real-time solutions have sought to change the execution behavior of the window manager either by intercepting and scheduling the calls made by client applications into the window manager [19] or by changing the scheduling priority of the window manager itself [38, 33, 28]. However, in modern systems, the window manager is often completely bypassed since applications may communicate directly with the GPU for the sake of performance. Indeed, no window manager need be involved when a GPU is used for general purpose computation.

More complete solutions for arbitrating access to a GPU can be realized through the interception and individual scheduling of the low-level commands applications send to the GPU. This approach is taken in [21] where a round-robin scheduler is used to approximate a fair allocation of the GPU for non-real-time applications. A similar approach is taken in [29] for the real-time domain, where commands are instead scheduled by fixed priority. This level of resource arbitration is finer-grained than the solutions presented in this paper and is an interesting approach. However, such solutions presently require the use of open-source drivers, which often do not offer the same level of performance as closed-source counterparts and can be expected to lag behind closed-source drivers in terms of support, features, and performance (at least until GPU manufactures more actively contribute to open-source development).

Outside of graphics, both the hardware and software models of systems with digital signal processors (DSPs) bear similarities to those with GPUs used for general purpose computation. The primary difference is that the DSP often avoids the complication of large complex drivers (though drivers may still need to be used). The use of DSPs was examined in [24] for static priority systems with a single CPU and single DSP. Similar work was performed for dynamic priority systems in [32], also for single-CPU/single-DSP systems. In both studies, access to a non-preemptible DSP is arbitrated by the use of real-time locking protocols (a technique also used in this paper). Unfortunately, the suspension-aware solutions presented in these studies cannot be easily extended to globally-scheduled multiprocessor platforms, and remains an open problem [30].

Finally, some early work has been done to determine the worst-case execution time of programs running on a GPU [37], though such in-depth analysis may not be necessary in soft real-time systems.

Contributions. The work presented in this paper differs from the aforementioned studies in that we seek to develop solutions that may be applied to globally scheduled multiprocessor systems (G-EDF scheduled systems, specifically) that adapt to closed-source drivers, and to also understand how a GPU may be leveraged to increase overall system computational performance while maintaining soft real-time scheduling guarantees. The contributions of this paper are as follows. We first profile common usage patterns for GPUs and explore the constraints imposed by both the graphics hardware architecture and the associated software drivers. We then present a real-time task model that is used to analyze the widely-available platforms of four-, eight-, and twelve-CPU systems with a *single* GPU. With this model in mind, we propose two real-time analysis methods, which we call the *Shared Resource Method* and the *Container Method*, with the goal of providing predictable system behavior while maximizing processing capabilities and addressing real-world platform constraints. The second method attempts to ameliorate suspension-oblivious analysis penalties that can arise in the first method, though this comes at the expense of other penalties. We compare these methods through schedulability experiments to determine when benefits are realized from using a GPU. Additionally, we present an implementation-oriented study that was conducted to confirm the necessity of real-time controls over a GPU in an actual real-time operating system environment. The paper concludes with a discussion of other avenues for possible real-time analysis methods and considers other problems presented by the integration of CPUs and GPUs.

2 Usage Patterns and Platform Constraints

Before developing any real-time analysis approach it is worthwhile to first examine the usage patterns of GPUs in general purpose applications as well as the constraints imposed by hardware and software architectures. As we shall see, these real-world characteristics cannot be ignored in a

Problem Type	**Average Exe. Time**	**Comm. Overhead**	**Comm. To Exe. Ratio**
Eigenvalue Computation			
512x512	$6.74ms$	$0.21ms$	3.12%
2048x2048	$23.58ms$	$0.21ms$	0.89%
4096x4096	$42.73ms$	$0.23ms$	0.54%
2D Convolution			
512x512	$1.50ms$	$0.47ms$	31.33%
2048x2048	$14.28ms$	$5.71ms$	39.99%
4096x4096	$55.91ms$	$22.53ms$	40.30%
Matrix Multiplication			
512x512	$1.89ms$	$1.79ms$	94.71%
2048x2048	$60.15ms$	$60.11ms$	99.93%
4096x4096	$431.55ms$	$431.42ms$	99.97%
2D Fluid Simulation			
512x512	$170\mu s$	—	—
2048x2048	$290\mu s$	—	—
N-Body Simulation			
1024 particles	$210\mu s$	—	—

Table 1: Observed GPU kernel execution times and communication overheads on a GTX-295 NVIDIA graphics card.

holistic system point-of-view. We begin by examining GPU execution environments.

A GPU-using program runs on system CPUs and may invoke one or more GPU programs, called a *kernels*, to utilize the GPU. Similar to a remote procedure call, the execution of a kernel is triggered by a function call made by the GPU-using program. Kernel execution time varies from application to application and can be relatively long. To determine likely execution-time ranges, we profiled sample programs from NVIDIA's CUDA SDK on a GTX-295 NVIDIA graphics card. We found that n-body simulations run on the order of $10 - 100\mu s$ per iteration on average while problems involving large matrix calculations (multiplication, eigenvalues, etc.) can take $2-500ms$ on average. Table 1 contains a summary of observed GPU execution times for several basic operations.

The execution times for these sample programs also include the time required to transmit data between the CPU and GPU, which we called the *communication overhead*. These overheads were measured using high-resolution time stamps placed before and after synchronous memory transfers. Like a remote procedure call, the CPU code that invokes a GPU kernel must also transmit input data. Likewise, the CPU code must also pull results back from the GPU. The time a job spends transferring data between the CPU and GPU is entirely application dependent. We found that data transfers can take anywhere from less than 1% to as much as 90% or more of the total execution time; this can be seen in the "Communication To Execution Ratio" column of Table 1.[2] The 2D Fluid Simulation and N-Body Simulation programs maintain simulation state on the GPU, so communication overheads are negligible.

The results in Table 1 indicate that relatively long GPU access times are common. Additionally, the I/O-based interface to a GPU co-processor introduces several additional unique constraints that need to be considered. First, a GPU cannot efficiently access main memory directly, so memory is most often copied between host and GPU memory. Memory is transferred over the bus (PCIe) explicitly or through DMA to explicitly-allocated blocks of main memory (in integrated graphics solutions, the GPU may use a partitioned section of main memory and data must be copied to and from this GPU-reserved memory space). Thus, memory between the host and GPU is non-coherent between synchronization points. Second, kernel execution on a GPU is non-preemptive: execution of the kernel must be run to completion before another kernel may begin. Third, kernels may not execute concurrently on a GPU even if many of the GPU's parallel sub-processors are unused.[3] Finally, a GPU is not a system programmable device, i.e., a general OS cannot schedule or otherwise control a GPU. Instead, a driver in the OS manages the GPU. This last constraint bears additional explanation.

At runtime, the host-side program sends data to the GPU, invokes a GPU program, and then waits for results. While this model looks much like a remote procedure call, unlike a remote

[2]The sample NVIDIA CUDA SDK programs were modified to use pinned memory, which prevents these memory segments from being potentially paged to disk. The use of pinned memory can significantly reduce communication overheads as the system can take advantage of direct memory access (DMA) data transfers. For example, the communication-to-execution ratio for the eigenvalue program increases to about 30% without it.

[3]NVIDIA's Fermi architecture allows limited simultaneous execution of kernels as long as these kernels are sourced from the same host-side context/thread. In this work, we will not consider such uses.

RPC-accessible system, the GPU is unable to schedule its own workloads. Instead, the host-side driver manages all data transfers to and from the device, triggers kernel invocations, and handles the associated interrupts. Furthermore, this driver is closed-source since the vendor is unwilling to disclose proprietary information regarding the internal operations of the GPU. Also, driver properties may change from vendor to vendor, GPU to GPU, and even from driver version to version. Since even soft real-time systems require provable analysis, the uncertain behaviors of the driver force integration solutions to treat it as a black box.

Unknown driver behaviors are not merely speculative but are a real concern. We performed an evaluation of an NVIDIA CUDA driver in a real-time environment to better understand and illustrate these limitations.

A workload was created consisting of ten simultaneously executing tasks, each of which would periodically compute the Fast Fourier Transform, using the GPU, of a random 1024x1024 matrix every $60ms$. Executing in isolation, each FFT took $2.5ms$ to compute on average, though execution times as little as $2ms$ were occasionally observed. Each computation was given an execution budget of $5ms$. Each task was configured in the CUDA runtime environment to suspend from CPU execution while it waited for the GPU to compute the FFT. Each task computed 100 FFTs before terminating.

This workload was run in LITMUS$^{\text{RT}}$ [18], UNC's real-time Linux testbed, under G-EDF scheduling. Our test platform was an Intel Core i7 quad-core system with an NVIDIA GTX-295 graphics card.[4] The system CPUs operate at 2.67GHz with an 8MB shared cache. The NVIDIA 190.53 64-bit Linux proprietary driver was used on the platform without modification. NVIDIA's CUDA 2.3 SDK provided the CUDA runtime environment. No display of any kind was used. Thus, the GPU was used exclusively for CUDA computations without interference from other processes.

The execution timing properties for the FFT workload are given in Table 2. The worst-case response time measurements are consistent with those that would be expected from a FIFO-order

[4]The GTX-295 actually provides two independent GPUs on a single card, though only one GPU was used in this work.

Average Response Time	$13.5ms$
Standard Deviation	$7.232ms$
Max Response Time	$25ms$
Min Response Time	$2ms$
Reported CPU Utilization	1.368

Table 2: The execution profile of ten simultaneously executing tasks using the GPU to compute the FFT of a random 1024x1024 matrix.

lock prioritization. The worst-case response time of an FFT computation in this case is given by the worst-case delay plus its own execution time, or $9 \cdot 2.5 + 2.5 = 25ms$. This suggests that the NVIDIA driver uses a FIFO-ordered queue itself to prioritize GPU access requests. However, there are several critical limitations to the NVIDIA driver's solution. Firstly, the NVIDIA driver does not implement priority inheritance. This is clear since LITMUS$^{\text{RT}}$ has its own unique notions of priority and the NVIDIA driver is clearly unaware of LITMUS$^{\text{RT}}$. Secondly, though each application suspended CPU execution while it used the GPU to compute the FFT, the average CPU usage as reported by the UNIX command `top` was 1.368, though it should have been less than approximately 0.4. This is because ten tasks executed for $2.5ms$ every $60ms$, so CPU usage should have been about $10 \cdot (2.5/60) \approx 0.4$ even if tasks did not suspend from the CPU while waiting for results from the GPU. This indicates that some FFT programs were likely busy-waiting within the GPU driver to either trigger kernel execution or send/receive data. Indeed, LITMUS$^{\text{RT}}$ detected jobs that exceeded their execution budgets of $5ms$, confirming that the GPU driver added additional execution time to the jobs.

The uncontrollable spinning when the GPU is under contention, despite runtime controls instructing tasks to suspend instead of spin, prevents the maximization of CPU resources. The lack of predictable real-time synchronization (i.e., the lack of priority inheritance) further makes it difficult to bound these spinning durations. It is very hard to make any real-time timing guarantees as a result. These serious behavioral deficiencies of the driver in a real-time environment must be resolved before a GPU can be used as a shared I/O device in a real-time system.

3 Task Model and Scheduling Algorithms

Real-time analysis offers several methods for describing the workload of a real-time system. This paper analyzes mixed task sets of CPU-only and GPU-using tasks with the synchronous implicit-deadline periodic task model, as it adequately describes many common workloads and has well-understood analytical properties.

A synchronous implicit-deadline periodic *task set*, T, consists of a set of recurrent *tasks*, $T_1, \cdots, T_n$, some of which may access a (single) GPU. We let $G(T)$ denote the set of GPU-using tasks in T. Each task, T_i, is described by three parameters: its *period*, p_i, which measures the separation between task invocations, known as *jobs*, and also the (implicit) relative deadline of each such job; its *worst-case CPU execution time*, e_i, which bounds the amount of CPU processing time a job must receive before completing; and its *worst-case GPU execution time*, s_i, which bounds the amount of GPU processing time required by one of its jobs. For a simple GPU-using job which only uses the GPU only once, s_i captures the interval of time between a kernel invocation and the signaling of its completion to the driver. A more complex GPU-using job may use the GPU several times until job completion. In this case, s_i captures the sum of the intervals of time between individual kernel invocations. Like worst-case CPU execution, s_i is unique to each task. Preliminary work [37] has been done to upper-bound GPU kernel execution time, though empirical tests are sufficient for many soft real-time systems. For tasks that do not use the GPU, $s_i = 0$. The *utilization* of task T_i is given by $u_i = e_i/p_i$ and the *system utilization* is given by $U = \sum u_i$.

Suppose that the GPU has a speed-up factor of f, i.e., it can complete the work required by a task f times faster than a CPU. Then a task's *effective utilization* is obtained by considering a CPU-only version of it of the same functionality. This CPU-only version may have a differing worst-case execution time. Let us denote the time a job of T_i spends communicating with the GPU, which is already a part of e_i, by $comm_i$. Then, the effective utilization of task T_i is given by

$$u_{\mathit{eff}_i} = (e_i - comm_i + f \cdot (comm_i + s_i))/p_i. \tag{1}$$

For example, suppose that a GPU-using task has parameters $T_i(p_i = 10, e_i = 4, s_i = 2)$ and $comm_i = 1$. If $f = 16$, then $u_{\mathit{eff}_i} = (4 - 1 + 16 \cdot (1 + 2))/10 = 5.1$.

Similar to system utilization, we also define *effective system utilization* by $U_{\mathit{eff}} = \sum u_{\mathit{eff}_i}$. We use U_{eff} to help in comparing GPU-using systems to corresponding CPU-only ones.

It is important to note that published research on GPU-accelerated algorithms often derive the speed-up offered by the GPU by comparing against *multi*-threaded implementations for multicore CPU platforms. In contrast, we specify the speed-up as a measure against *single*-threaded CPU implementations. We opted to conservatively specify speed-up in this manner to provide a generalized baseline for comparisons against CPU-only real-time systems since the number of CPUs may change from system to system. A single-threaded measure of speed-up also allows us to avoid the additional complexities posed by multi-threaded real-time tasks, the analysis of which would require the consideration of schedulers for multi-threaded jobs, such as gang-schedulers [27].

We wish to make the most efficient use of CPU and GPU resources while supporting soft real-time constraints. One might consider real-time methods for supporting heterogeneous systems, where a system has processors of differing capacities or capabilities. Unfortunately, due to a GPU's unique constraints, direct approaches for heterogeneous systems [14, 15, 23] do not immediately apply. These methods either require a partitioning of tasks amongst heterogenous processors (a GPU-using job must execute on both a CPU and GPU for particular phases of execution) or require that jobs may be preempted and migrated between processors (a GPU cannot be preempted). However, as noted earlier, previous work [20] has shown that G-EDF can ensure bounded tardiness in ordinary multiprocessor systems (without a GPU) without system utilization constraints (provided the system is not overutilized). Thus, it is the primary scheduling algorithm considered in this paper.[5] As noted earlier, G-EDF is a *global* scheduler, meaning that jobs share a single ready queue and can migrate between processors. G-EDF prioritizes work by job deadline, scheduling

[5]Some have recently speculated that the *earliest-deadline-zero-laxity* (EDZL) algorithm may be better suited to accounting for self-suspensions (caused, for example, by using a GPU) [30], though actionable results have yet to be presented, so better suspension accounting remains an open problem.

jobs with the earliest deadlines first.

4 Analysis Methods

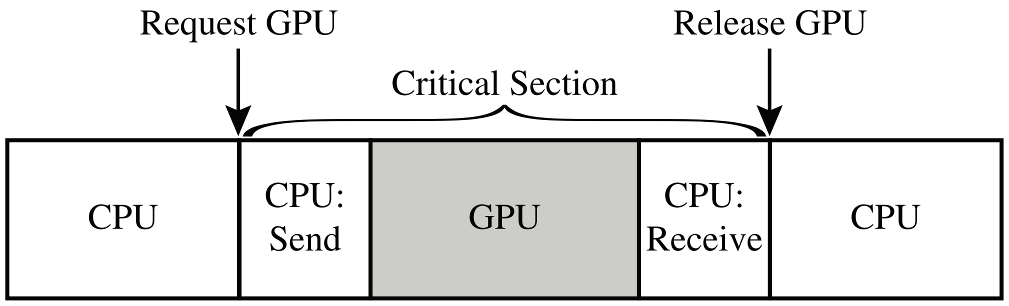
We consider two methods for analyzing mixed task sets of CPU-only and GPU-using tasks on a multiprocessor system with a single GPU: the *Shared Resource Method* and the *Container Method*. Fundamental differences between these methods stem from how GPU execution time is modeled and how potential graphics hardware driver behaviors are managed.

4.1 Shared Resource Method

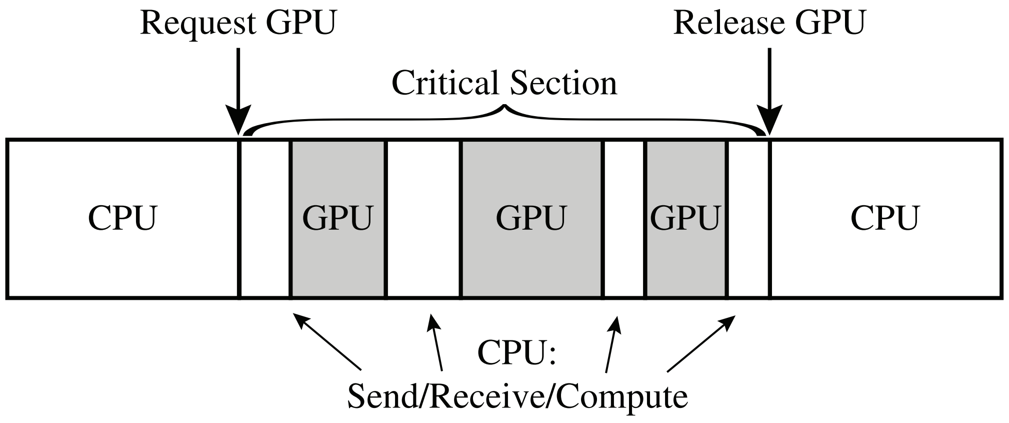
It is natural to view a GPU as a computational resource shared by the CPUs of a multiprocessor system. This is the approach taken by the Shared Resource Method (SRM), which treats the GPU as a globally-shared resource protected by a real-time semaphore protocol.

The execution of a simple GPU-using job goes through several phases. In the first phase, the job executes purely on the CPU. In the next phase, the job sends data to the GPU for use by the GPU kernel. Next, the job suspends from the CPU when the kernel is invoked on a GPU. The GPU executes the kernel using many parallel threads, but kernel execution does not complete until after the last GPU thread has completed. Finally, the job resumes execution on the CPU and receives kernel execution results when signaled by the GPU. Optionally, the job may continue executing on the CPU without using the GPU. Thus, a GPU-using job has five execution phases as depicted in Fig. 2. This model of a GPU-using job is a generalization of potentially more complex execution patterns. A more complex job may execute multiple kernels, communicating with the GPU and performing intermediate computations (perhaps determining which kernels to run based on results from previous kernels) between invocations.[6] However, despite this extra complexity, we may still model this pattern with beginning and ending points. That is, when a job begins using the GPU and when it is finishes using the GPU. The actual execution pattern within these framing points

[6]For performance, GPU operations may be performed asynchronously by the GPU-using job. This allows several GPU operations to be batched together and treated as a single operation, reducing the number of times the job must suspend to wait for GPU results. No changes to our task model are necessary to support this type of operation.



(a) Simple GPU-using job.



(b) Complex GPU-using job.

Figure 2: Execution phases of GPU-using jobs. A job obtains exclusive use of the GPU and the GPU is not released until all GPU-related operations have completed. (a) A simple GPU-using job runs one kernel with data transmissions occurring before and after kernel invocation. (b) A complex job may run several GPU kernels and perform several data transmissions and intermediate computations on the CPU.

does not affect suspension-oblivious analysis since the entire duration is treated as CPU execution time.

We can remove the GPU driver from resource-arbitration decisions and create a suitable model for real-time analysis through the use of a real-time semaphore protocol. Contention for a GPU may occur when a job attempts to communicate with it or run GPU kernels. We resolve this contention with a synchronization point between the first and second phases to provide mutual exclusion through the end of the fourth phase; this interval is called a *critical section* and denoted for each task T_i by cs_i.

The synchronization process to protect the critical sections is run entirely on the CPU and may be implemented using the same types of primitives, such as mutexes, often used to protect traditional shared resources and data objects. A GPU-using job requests use of the GPU by first attempting to acquire a mutex designated to protect the GPU. The job enters its critical section once it has acquired the mutex. The job relinquishes the GPU by releasing the mutex. This approach ensures that the driver only services one job at a time, which eliminates the need for knowing how the driver (which, again, is closed-source) manages concurrent GPU requests. Further, no modifications to, or hooks into, the driver are necessary.

We may consider several real-time multiprocessor locking protocols to protect the GPU critical section. Such a protocol should have several properties. First, it must be usable under G-EDF scheduling. Second, it must allow blocked jobs to suspend since critical-section lengths may be very long (recall Table 1). A spin lock would consume far too much CPU time. Third, the protocol must support priority inheritance so blocking times can be bounded. Finally, the protocol need not support critical-section nesting or deadlock prevention since GPU-using tasks only access one GPU. Both the "long" variant of the *Flexible Multiprocessor Locking Protocol* (FMLP-Long) [16] and the more recent global *O(m) Locking Protocol* (OMLP) [17] fit these requirements. Neither protocol is strictly better than the other for all task sets since priority-inversion-based blocking (per lock access), denoted by b_i, is *O(n)* under the FMLP-Long and *O(m)* under the OMLP, where n is the number of tasks and m is the number of CPUs. Thus, we allow the SRM to use whichever protocol yields a schedulable task set.

The FMLP-Long uses a single FIFO job queue for each semaphore, and GPU requests are serviced in a first-come first-serve order. The job at the head of the FIFO queue is the lock holder. A job, J_i, of task $T_i \in G(T)$ may be blocked by one job from the remaining GPU-using tasks. Formally,

$$b_i = \sum_{G(T)\setminus\{T_i\}} cs_j. \qquad (2)$$

The global OMLP uses two job queues for each semaphore: FQ, a FIFO queue of length at most m; and PQ, a priority queue (ordered by job priority). The lock holder is at the head of FQ. Blocked jobs enqueue on FQ if FQ is not full and on PQ, otherwise. Jobs are dequeued from PQ onto FQ as jobs leave FQ. Any job acquiring an OMLP lock may be blocked by at most $2m - 1$ jobs. Thus, we may loosely bound the blocking time for tasks in $T_i \in G(T)$ with the formula

$$b_i = (2m - 1) \cdot cs_{max}. \tag{3}$$

However, Brandenburg et al. have presented in an appendix of [17] a tighter derivation of the worst-case priority-inversion-based blocking that jobs in a *hard* real-time system may experience while waiting for a resource using the OMLP under G-EDF scheduling. This derivation is summarized by the following two theorems:

Theorem 1 ([17]). *If $|G| \leq m$, then J_i is blocked for at most*

$$b_i = \sum_{T_j \in G \setminus \{T_i\}} cs_j. \tag{4}$$

Theorem 2 ([17]). *Let the* worst-case task interference *be given by the formula*

$$tif(T_i, T_j) = \{cs_{j,l} \mid \lceil (p_i + rt_j)/p_j \rceil\}, \tag{5}$$

where rt_j is the worst-case response time of job J_j, and the worst-case request interference *be given by the formula*

$$A = xif(T_i) = \bigcup_{T_j \in G \setminus \{T_i\}} tif(T_i, T_j). \tag{6}$$

Let A_{max} be the $2m-1$ jobs in A with the longest critical sections. Then a job J_i of T_i is blocked for at most

$$b_i = \sum_{J_j \in A_{max}} cs_j. \tag{7}$$

Theorem 1 describes the case when the number of resource-using tasks is at most m. The FIFO queue, FQ, may hold all pending requests simultaneously in this case. Since FQ is FIFO-ordered, J_i is worst-case blocked for the duration of the critical section of one job for each task in $G \backslash \{T_i\}$.

Theorem 2 captures the case when pending requests may spill out into PQ. Determining worst-case blocking in this instance is more difficult since J_i may be delayed by more than one job generated by the same task. Eq. (5) gives the set of critical sections generated by T_j that may interfere with J_i. Eq. (6) gathers the interference from all possible tasks. Finally, Eq. (7) specifies the maximum time J_i may be blocked by all requests; the maximum number of requests that may delay J_i provides an upper bound on the total number critical sections that must be completed before J_i is unblocked. This upper bound is $2m-1$. We refer the reader to [17] for a more detailed description of the OMLP and these two theorems.

In the case of *soft* real-time systems scheduled with G-EDF, such as the one considered in this paper, a revised form of Eq. (5) must be used since worst-case response times are not readily available. This can be done by replacing the term p_i in Eq. (5) by $p_i + x_i$ and the term rt_j by $p_j + x_j$, where x_i and x_j are tardiness bounds computed using bounded tardiness analysis [20, 22]. This yields the formula

$$tif_{soft}(T_i, T_j) = \{cs_{j,l} \mid \lceil (p_i + x_i + p_j + x_j)/p_j \rceil\}. \tag{8}$$

However, the tardiness analysis used to compute x_i and x_j is dependent upon the execution requirement, including blocking time, of every task in the task set, creating a co-dependence between determining blocking times and tardiness. A fixed-point iterative method must be used to determine a stable b_i before determining task set schedulability.

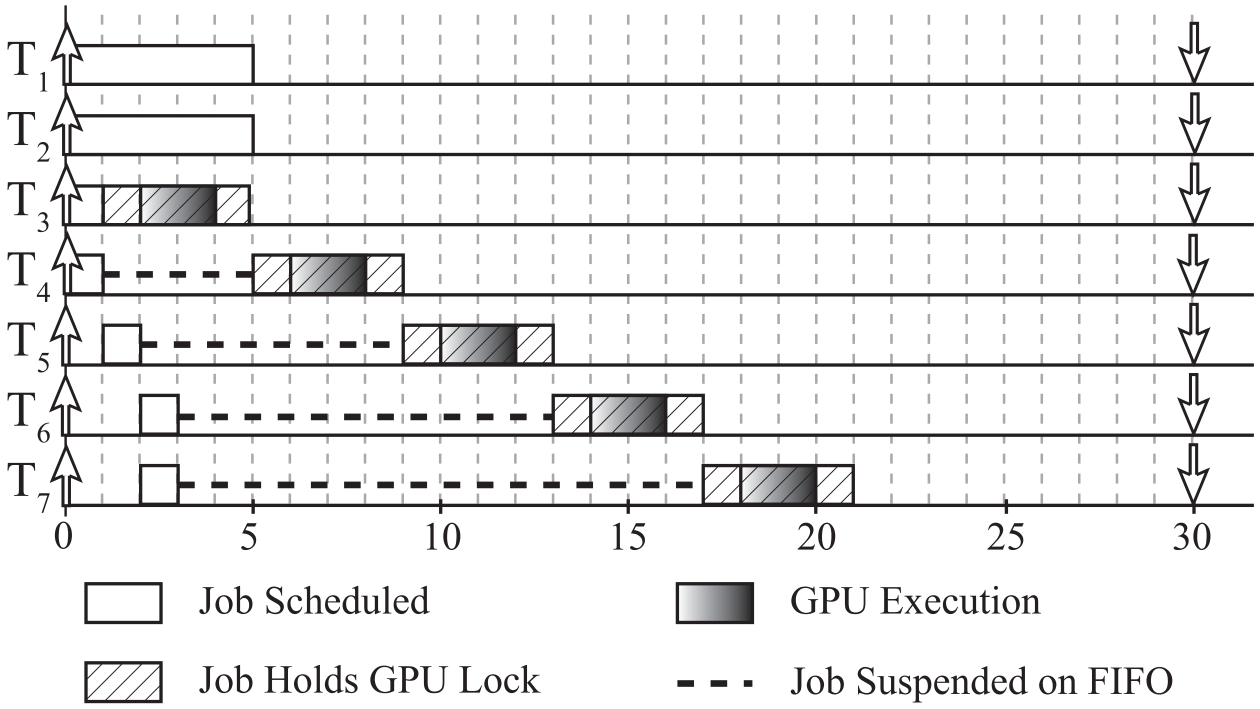


Figure 3: Schedule for the example task set under the SRM on a four-processor single-GPU system.

Soft schedulability under the SRM is determined by the following two conditions. First,

$$e_i + s_i + b_i \leq p_i \tag{9}$$

is required by the tardiness analysis for G-EDF [20]. Second, the condition

$$U = \sum(e_i + s_i + b_i)/p_i \leq m \tag{10}$$

must also hold. This is the soft G-EDF schedulability condition required by [20] to ensure bounded tardiness. Like all suspension-oblivious tests, we must analytically treat suspension due to both blocking and GPU execution as execution on the CPU. Note that no schedulability test is required for the GPU co-processor since a job's mutually exclusive GPU execution is masked by fictitious CPU execution. Still, the suspension-oblivious nature of this test is a limiting characteristic as is seen in Sec. 5.

Example. Consider a mixed task set with two CPU-only tasks with task parameters ($p_i = 30, e_i = 5, s_i = 0$) and five GPU-using tasks with parameters ($p_i = 30, e_i = 3, s_i = 2, cs_i = 4$) to be scheduled on a four-CPU system with a single GPU. The CPU-only tasks trivially satisfy Ineq. (9). The FMLP-Long is best suited for this task set and the blocking term for every GPU-using task is $\sum cs_k = 16$ as computed by Eq. (2). Tasks in $G(T)$ satisfy Ineq. (9) since $3+2+16 = 21 \leq 30$. Ineq. (10) also holds since $U = 2 \cdot (5/30) + 5 \cdot ((3+2+16)/30) \approx 3.83 \leq 4$. Therefore, the task set is schedulable under the SRM.

A schedule for this task set is depicted in Fig. 3. T_1 and T_2 are the CPU-only tasks. Observe that the last scheduled job in the figure completes at time 21, well before its deadline. The computed system utilization of approximately 3.83 is quite close to the upper bound of 4.0 used in Ineq. (10), which suggests a heavily-utilized system. However, the schedule in Fig. 3 shows that the suspension-oblivious analysis is quite pessimistic (mostly due to blocking-term accounting) given that the system is idle for much of the time. In fact, only one CPU is utilized after time 5. The performance of the GPU must overcome these suspension-oblivious penalties if it is to be a worthwhile addition to a real-time multiprocessor system.

4.2 Container Method

The SRM may be overly pessimistic from a schedulability perspective due to heavy utilization penalties arising from the blocking terms introduced by the use of a multiprocessor locking protocol. Methods that lessen such penalties may offer tighter analysis. The Container Method (CM) is one such approach.

In many cases, a *single* GPU will limit the total actual CPU utilization (where suspension effects are ignored) of tasks in $G(T)$. For example, if all tasks in $G(T)$ perform most of their processing on the GPU, then the total actual CPU utilization of these tasks will be much less than 1.0 when the GPU is fully utilized. However, if we consider *suspension-oblivious utilization* (so-utilization), given by the formula $u_{so_i} = (e_i + s_i)/p_i$, then the actual GPU utilization and so-utilization, will both be close to 1.0. This fact inspires the CM, which avoids heavy suspension-oblivious penalties

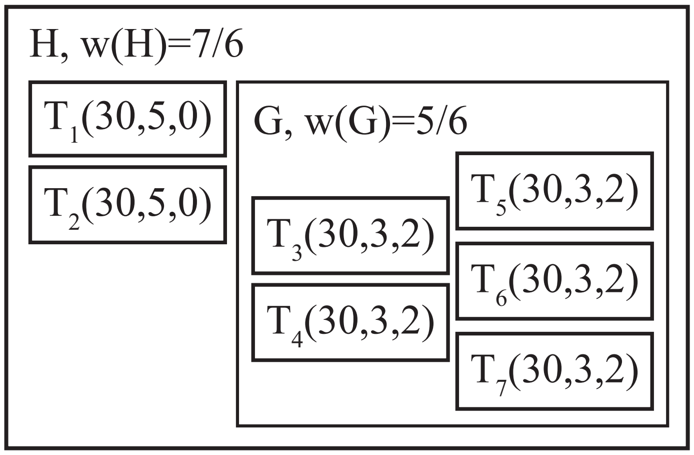


Figure 4: Container decomposition of an example mixed task set.

by removing contention for the GPU resource through the isolation of $G(T)$ to a single (logical) processor.

It was shown in [31] that bandwidth reservations, or *containers*, may be used to support soft real-time guarantees in multiprocessor systems. In a container-based system, a task set is organized into a hierarchical collection of containers. Each container may hold tasks or child containers. A container C is assigned an *execution bandwidth*, $w(C)$, equal to the sum of the utilizations (in our case, so-utilizations) and bandwidths of its child tasks and containers, respectively.

For our GPU-enabled multiprocessor system, we place all CPU-only tasks in a root container, H, and all tasks of $G(T)$ in a child container G of H. In implementation, it is assumed that the system designer has configured the system to place tasks in their proper containers, or that tasks may be able to self-organize at initialization time through the use of one or several system calls.

Under the CM, suspensions are treated as execution time and contribute to task utilizations. This suspension-oblivious analysis allows the CM to also support GPU-using jobs with complex usage patterns, just like the SRM. A container decomposition of the example task set given in Sec. 4.1 is shown in Fig. 4. Observe that the tasks in $G(T)$ are isolated in container G with a bandwidth of $5/6$, the total so-utilization of the tasks in $G(T)$. Container G and the CPU-only tasks are contained within H, which has a bandwidth of $7/6$.

Containers provide *temporal isolation* by hierarchically allocating execution time to contained tasks and containers. If each container schedules its contained tasks and containers using a *window-constrained* scheduling algorithm,[7] such as the G-EDF, then bounded tardiness can be ensured with no utilization loss [31]. The CM exploits both this and the ability to apply different (window-constrained) schedulers to subsets of jobs.

We schedule the children of H with G-EDF and schedule the children of G with uniprocessor FIFO, which is a window-constrained algorithm that prioritizes jobs by release time. All GPU contention is avoided through the use of the FIFO scheduler, which eliminates preemptions, assuming jobs do not self-suspend or self-suspensions are analytically treated as CPU execution. This ensures that the GPU is always available to the highest-priority (according to FIFO) GPU-using job which is implicitly granted exclusive access to the GPU because the job itself is scheduled. However, in implementation it is not necessary for G to suspend or idly consume CPU resources while the GPU is in use. Instead, G may schedule other contained jobs, provided that the GPU critical sections are protected by a simple release-ordered semaphore. This ensures that the highest-priority (by FIFO) job may be scheduled immediately, without conflict, when it is ready to run. This work-conserving approach would reduce observed tardiness, though this is not captured by our analysis here.

Soft schedulability of a task set under the CM is determined by the following conditions. First,

$$w(G) \leq 1 \tag{11}$$

is required to ensure that G is schedulable with bounded tardiness on a uniprocessor. Second,

$$w(G) + \sum_{T_i \notin G(T)} u_i \leq m \tag{12}$$

must also hold. This condition ensures that the root container can be scheduled by G-EDF on m

[7]A *window-constrained* scheduling algorithm prioritizes a job by a time point contained within an interval window that also contains the job's release and deadline.

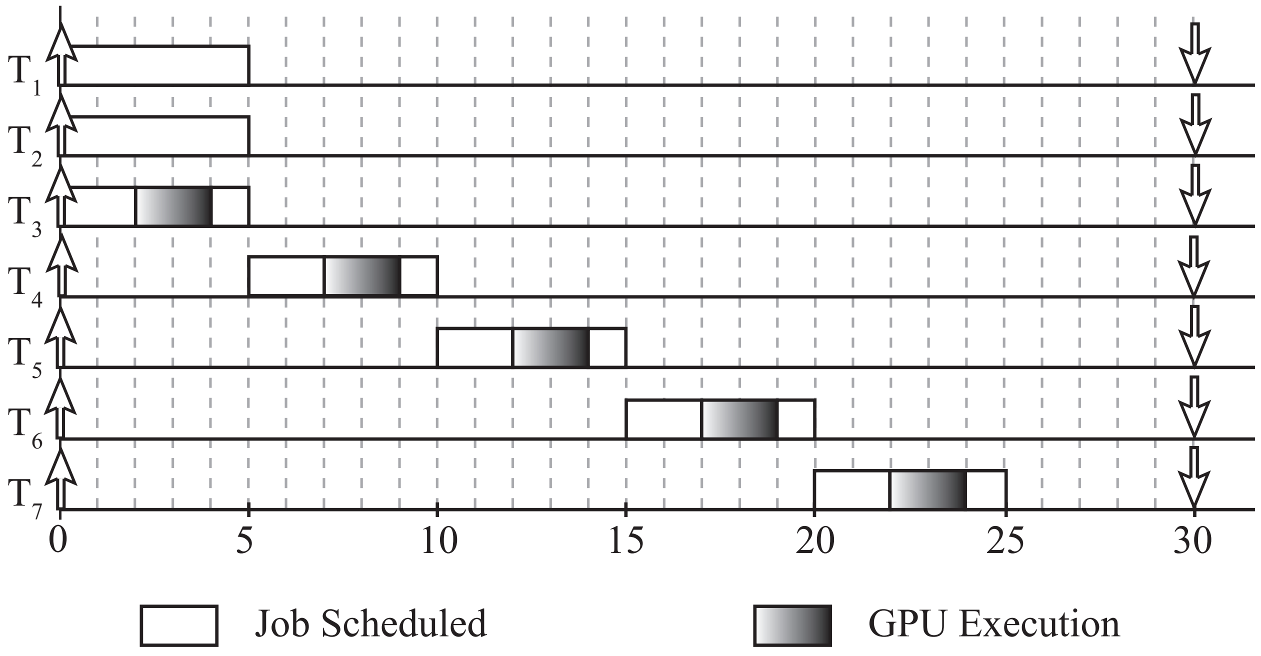


Figure 5: Schedule for the example task set under the CM on a four-processor single-GPU system.

CPUs with bounded tardiness.

Example. Consider the same mixed task set from Sec. 4.1. Ineq. (11) is satisfied since the container bandwidth is $w(G) = 5 \cdot ((3+2)/30) \approx 0.83 \leq 1$. Ineq. (12) also holds as $U = 2 \cdot (5/30) + 5 \cdot ((3+2)/30) \approx 1.16 \leq 4$. Therefore, the task set is schedulable under the CM. A schedule for this task set is depicted in Fig. 5. T_1 and T_2 are the CPU-only tasks. Note that the last scheduled job completes at time 25.

The SRM enforces more permissive constraints on the GPU while the CM enforces more permissive constraints on the CPUs. This trade-off is reflected in both the schedulability tests and example schedules of these methods. Due to the mutually exclusive ownership of the GPU, there may exist only one job within its critical section ready to be scheduled on any CPU at any given time. This implies that system GPU utilization under the SRM can be bounded by the formula

$$\sum_{T_i \in G(T)} cs_i/p_i \leq 1 \tag{13}$$

for schedulable task sets. This measure includes CPU execution time within critical sections since entire critical sections must execute in sequence. Comparing the SRM's and the CM's measures of GPU utilization for the previous example, we find the SRM's GPU utilization (Ineq. (13)) is approximately 0.67 while the CM's (Ineq. (11)) is approximately 0.83; the SRM's CPU constraint (Ineq. (10)) is approximately 3.83 while the CM's (Ineq. (12)) is approximately 1.16. Such trade-offs are not merely limited to the tightness of analytical bounds, but are actually reflected in task set schedules, as can be observed in Figs. 3 and 5. While the CM enforces more permissive CPU utilization constraints, the GPU-using jobs complete later under the CM. This corresponds directly to the CM's higher measure of GPU utilization.

5 Evaluation of Theoretical Schedulability

We carried out SRM- and CM-related schedulability experiments to help answer the two questions raised at the beginning of this paper: (1) How much faster than a CPU must a GPU be to overcome suspension-oblivious penalties and schedule more work than a CPU-only system? And, (2) how much work should be offloaded onto a GPU to make the most efficient use of both the system CPUs and GPU?

5.1 Experimental Setup

To better understand the schedulability of mixed task sets, we randomly generated task sets with varying characteristics, testing them for schedulability on idealized systems (with no OS or hardware overheads) using the schedulability tests described in Sec. 4. Though we ignore system overheads, we may still observe important schedulability trends, make comparisons between the SRM and the CM, and gauge the degree to which a GPU must speed up computations to make their inclusion in a soft real-time system under suspension-oblivious analysis worthwhile.

Task sets were randomly generated using varied characteristics with values for parameters inspired by multimedia and computer vision applications, those where real-time GPUs may be readily applied. These domains often perform image processing algorithms such as FFTs and matrix operations (such as those given in Table 1) at various frame rates.[8]

The task set characteristics varied by the following: three task utilization intervals, three period intervals, three GPU usage patterns, and ten GPU percentage task shares. *Utilization intervals* determine the range of utilization for individual tasks and were $[0.01, 0.1]$ (*light*), $[0.1, 0.4]$ (*medium*), and $[0.5, 0.9]$ (*heavy*). *Period intervals* determine the range of task periods for individual tasks and were $[3ms, 33ms]$, $[15ms, 60ms]$, and $[50ms, 250ms]$. The *GPU usage pattern* determines how much of the execution time of each GPU-using task is spent using the GPU (this duration is equivalent to critical section length); 25%, 50%, and 75% were used, in line with common CPU/GPU workload distributions. Finally, the *GPU percentage task share* is the ratio of GPU-using tasks to the total number of tasks; increments of 10% were used to test GPU task percentages from 0% to 100%. A schedulability experiment scenario was defined by any permutation of these four parameters for systems with four, eight, and twelve CPUs, yielding a total of 810 scenarios. We may safely ignore the effect of communication overheads because both the SRM and CM use suspension-oblivious analysis. That is, the SRM analysis only considers the length of the critical section, not the work distribution inside it. Likewise, the CM analysis only uses the sum of execution and suspension durations, $e_i + s_i$, and not the parameters individually.

We generated random task sets for each scenario in the following manner. First, we selected a total system utilization cap uniformly in the intervals $(0, 4]$, $(0, 8]$, and $(0, 12]$, capturing the possible system utilizations of a platform with four, eight, and twelve CPUs (respectively) and a single GPU when suspension-oblivious analysis is used. We then generated tasks by making selections uniformly from the utilization interval and period interval according to the given scenario. We derived execution times from these selections. We added the generated tasks to a task set until the set's total utilization exceeded the utilization cap, at which point the last-generated task was discarded. Next, we selected tasks for $G(T)$ from the task set; we determined the number of GPU-

[8]Common workload profiles were solicited from research groups at UNC that frequently make use of CUDA. A poll was also informally taken at the NVIDIA CUDA online forums. Similar timing characteristics were later confirmed in the domain of computer vision for real-time automotive applications [34].

using tasks by the GPU task percentage of the scenario. We then assigned the same GPU usage pattern to each task in $G(T)$ according to the scenario. We made cursory tests of CPU and GPU utilization to ensure that the CPUs and GPU were not implicitly overutilized. Finally, we discarded task sets with only one GPU-using task since this case is uninteresting as the GPU does not require resource arbitration. We tested a total of 5,000,000 task sets for each scenario.

We tested the SRM and the CM according to the schedulability conditions already described in Sec. 4.

5.2 Results

A representative subset of graphs resulting from our schedulability experiments is presented in this section to show the schedulability properties of the SRM and the CM and to demonstrate their advantages over pure CPU-only systems.[9] We are limited by reasonable page constraints from presenting results for all 810 scenarios.[10] The presented subset of scenarios was selected because they best utilized *both* the GPU and the CPUs, illustrating seen trends more broadly.

Schedulability results for task sets with a GPU percentage task share ranging from 20% to 30% are shown in Fig. 6 for a four-CPU system; 10% to 20% for an eight-CPU system in Fig. 8; and up to 10% for a twelve-CPU system in Fig. 9. These task percentage ranges were selected for each of the respective systems as they represent the GPU percentage task share scenario where schedulable effective system utilizations were maximized in general.

The graphs are organized to show trends as functions of per-task utilization (across the rows) and GPU usage pattern (down the columns). Columns correspond to GPU usage patterns of 25%, 50%, and 75%. Likewise, rows correspond to light, medium, and heavy per-task utilizations. For readability, each figure is broken up by column across several pages.

We were unable to generate schedulable heavy task sets in some scenarios for the four-CPU case for the GPU percentage task share range 20% to 30% (Fig. 6 insets (h) and (i)). We were also

[9]Please note that some graphs appear to be missing data points at lower and upper system utilization ranges. This is caused by the occasional inability to generate task sets meeting particular scenario constraints. This was usually due to the inability to generate a task set with at least two GPU-using tasks under the given constraints.

[10]Graphs for all scenarios are available at `http://www.cs.unc.edu/~anderson/papers.html`.

unable to generate any heavy task sets meeting scenario constraints for the twelve-CPU system with the GPU percentage task share range up to 10% and are not shown. Instead, schedulability results for heavy task sets with the GPU percentage task share ranges 30% to 40% (Fig. 7) and 10% to 20% (Fig. 10) are presented as supplements for the four- and twelve-CPU systems, respectively.

Each figure plots schedulability for four assumed GPU speed-up factors (2x, 4x, 8x, and 16x) applied uniformly to each GPU-using task. The x-axis in each figure gives the effective system utilization and extends up to the maximum schedulable effective system utilization (for the greatest tested speed-up factor) achievable by an optimal scheduler, which can schedule all CPUs and the GPU to maximum capacity. For example, a four-CPU system with a speed-up factor of 16x has maximum schedulable effective system utilization of $4 + 1 \cdot 16 = 20.0$.

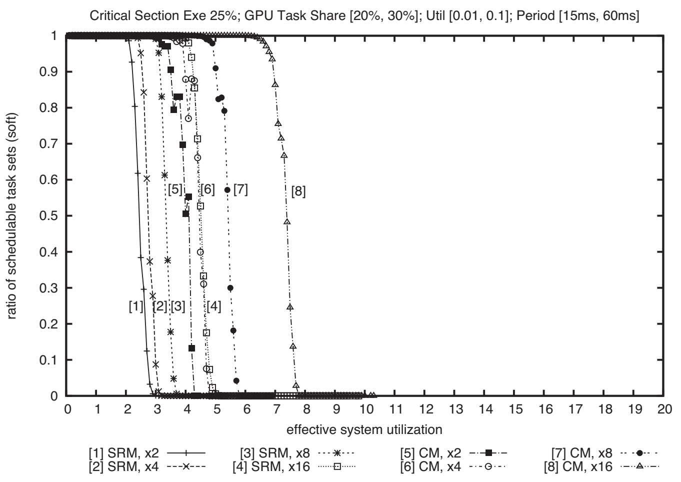
Performance Gains

The presented test results allow us to draw conclusions on how much a system may benefit from a GPU. They also help us answer (in the context of this experimental framework) our original question of how fast a GPU must be to overcome penalties from suspension-oblivious analysis.

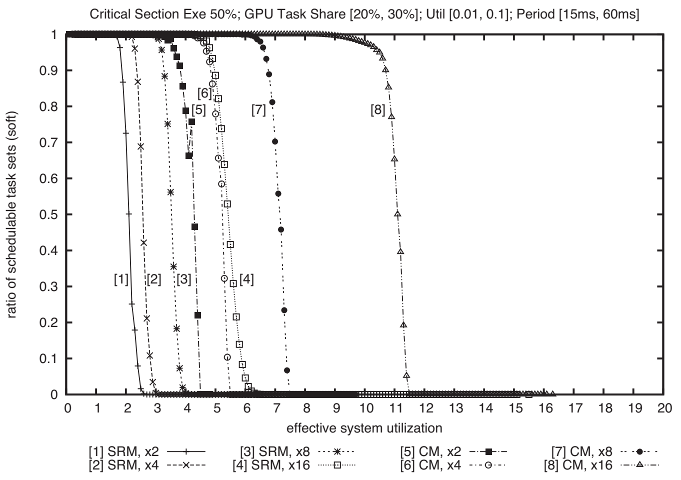
Observation 1. *A GPU can be used to achieve effective system utilizations much greater than the number of CPUs.* In Fig. 6(d) we see that an effective system utilization of nearly 8.0 is achievable on a four-CPU system when GPU-using tasks only use the GPU for 25% of their execution time. This increases to 15.0 when the GPU is used for 75% of the GPU-using tasks' execution time (Fig. 6(f)). These maximum schedulable utilizations are two to three times greater than what a four-CPU system can achieve without a GPU. Similarly, a GPU can give schedulable effective utilizations up to approximately 20.0 (Fig. 8(i)), a 2.5 times increase in maximum schedulable effective system utilization, for an eight-CPU system, and effective utilizations as great as 23.0 (Fig. 9(f)), nearly twice the CPU-only capacity, for a twelve-CPU system.

Observation 2. *A GPU usually allows a four-CPU system to schedule more work than possible with only CPUs when the GPU offers a speed-up of 4x or greater.* The schedulability curves for the CM with a speed-up of 4x or greater show that task sets with effective system utilizations greater than 4.0, excepting heavy task sets, are schedulable as seen in Fig. 6. Not all task sets with effective system utilizations of 4.0 are schedulable under the CM with a GPU speed-up of 4x when the critical sections are only 25% of execution time, as seen in insets (a) and (d) of Fig. 6, though roughly 80% and 90% of these task sets are still schedulable (excepting heavy task sets) in these insets respectively; furthermore, effective system utilizations of up to 4.5 are possible in each of these scenarios. The SRM provides similar benefits with a speed-up of 4x for heavy task sets in Fig. 6(g) and Fig. 7 (which, as mentioned, supplements Fig. 6). The SRM is able to achieve effective system utilizations greater than 5.0. In most scenarios, either the CM or the SRM may be used to achieve effective utilizations greater than 4.0 with a GPU speed-up of 4x. *Helping us answer question Q1 stated in Sec. 1, we observe that in the context of this experimental framework, a GPU must offer about a 4x speed-up over a CPU on a four-CPU system to overcome suspension-oblivious penalties and schedule more work than a CPU-only system with four CPUs.*

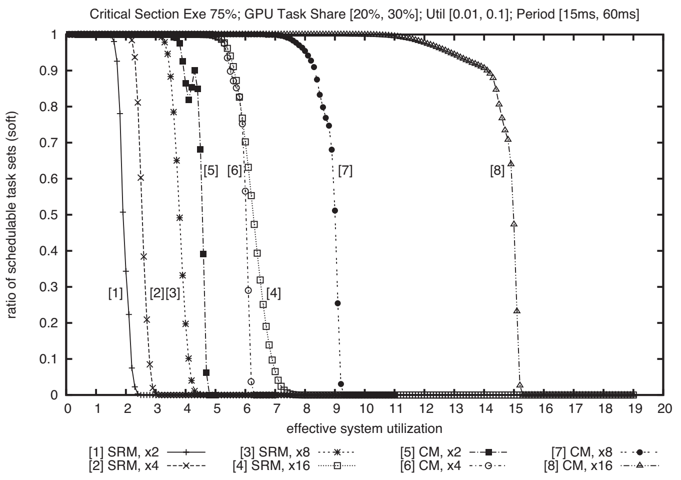
Observation 3. *A GPU usually allows an eight-CPU system to schedule more work than possible with only CPUs when the GPU offers a speed-up of 8x or greater, though a speed-up of 4x frequently offers benefits.* The schedulability curves for the CM with a speed-up of 8x or greater show that task sets with effective utilizations greater than 8.0, excepting heavy task sets, are schedulable as seen in Fig. 8. Not all task sets with effective utilizations of 8.0 are schedulable under the CM with a GPU speed-up of 8x when the critical sections are only 25% of execution time, though roughly 60% of these task sets are still schedulable in insets (a) and (d) of Fig. 8; furthermore, effective utilizations of 9.0 to 9.5 are possible under these conditions. The SRM provides similar benefits with a speed-up of only 4x for heavy task sets in insets (g), (h), and (i) of Fig. 8, where



(a) GPU Usage 25%

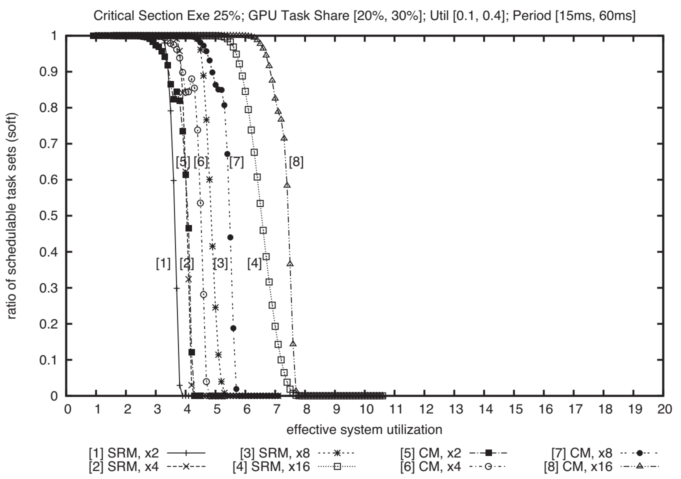


(b) GPU Usage 50%

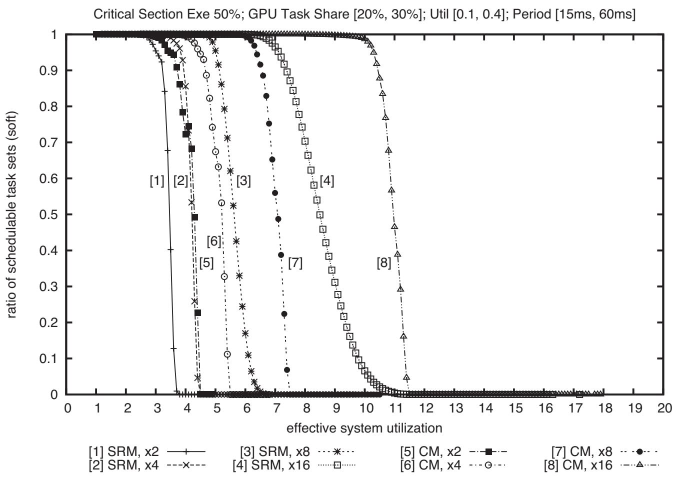


(c) GPU Usage 75%

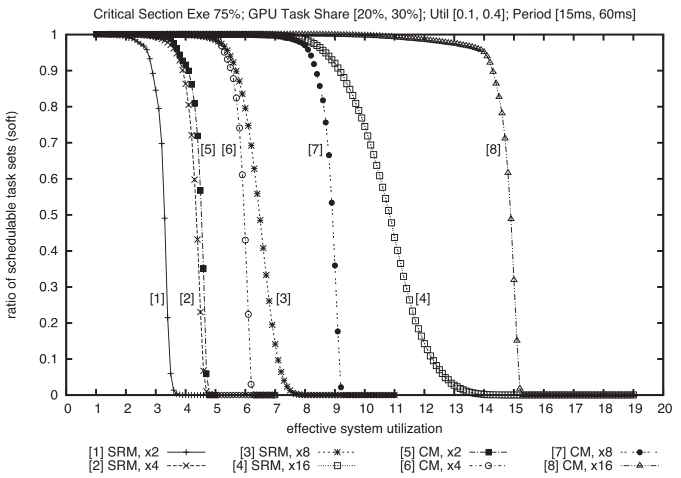
Figure 6: Four-CPU System: Light Task Sets, GPU Percentage Task Share 20–30%



(d) GPU Usage 25%

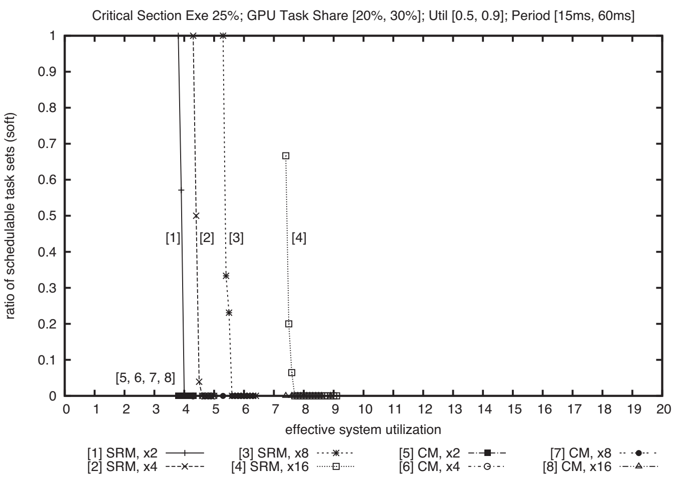


(e) GPU Usage 50%

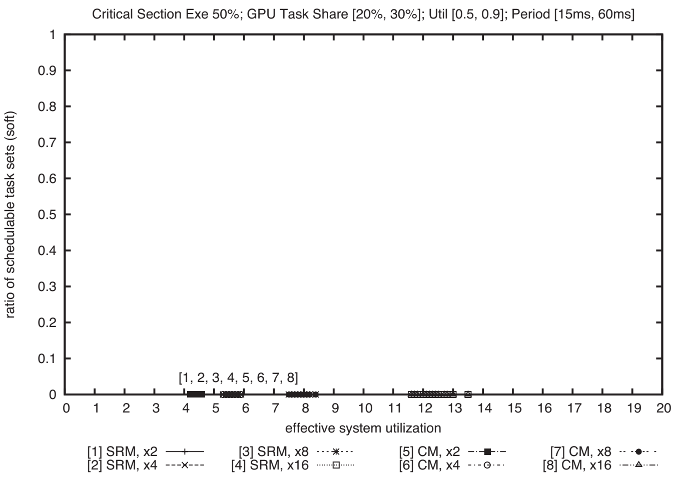


(f) GPU Usage 75%

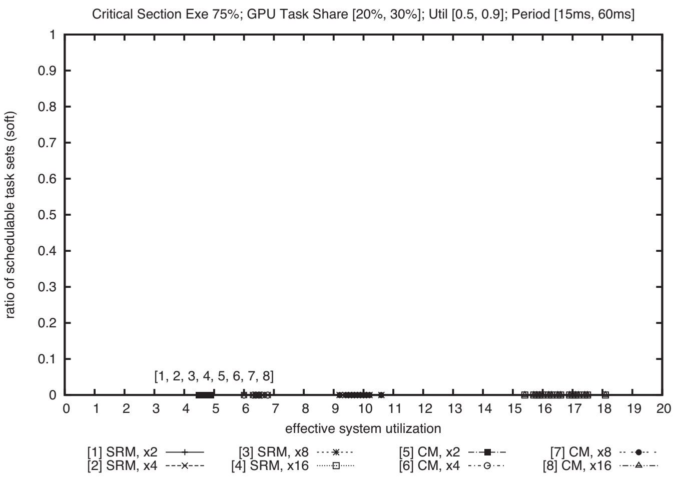
Figure 6: (continued) Four-CPU System: Medium Task Sets, GPU Percentage Task Share 20–30%



(g) GPU Usage 25%

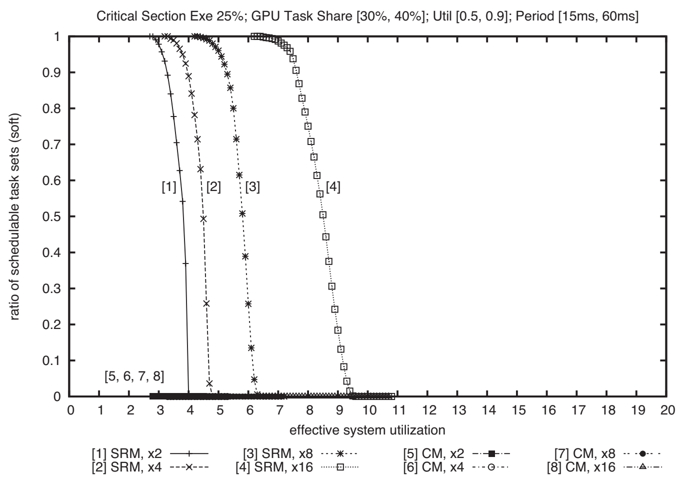


(h) GPU Usage 50%

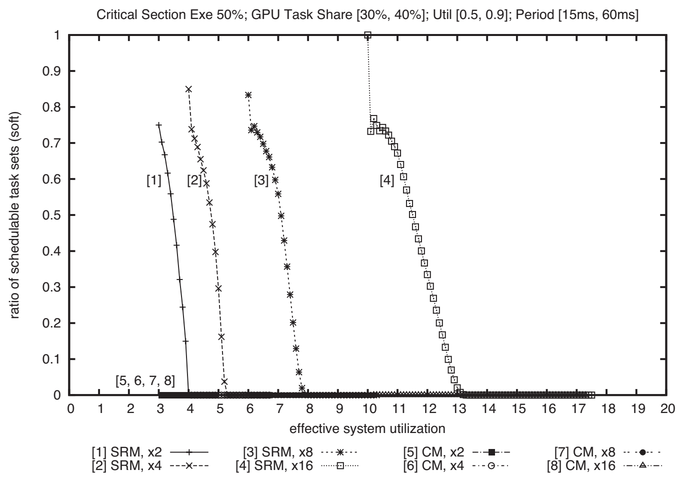


(i) GPU Usage 75%

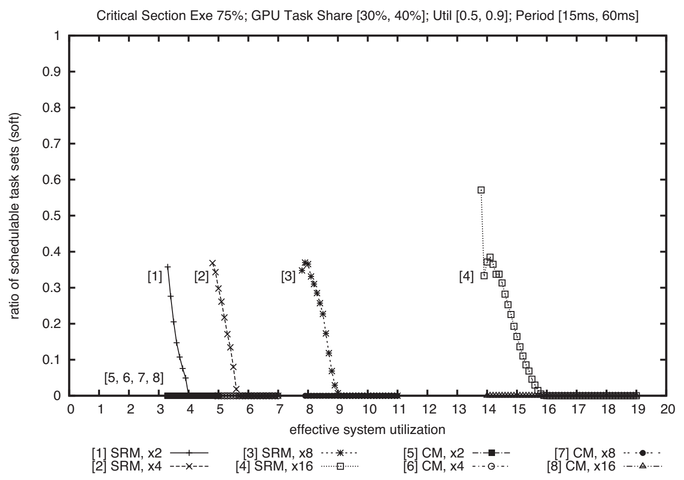
Figure 6: (continued) Four-CPU System: Heavy Task Sets, GPU Percentage Task Share 20–30%



(a) GPU Usage 25%



(b) GPU Usage 50%



(c) GPU Usage 75%

Figure 7: Supplemental, Four-CPU System: Heavy Task Sets, GPU Percentage Task Share 30–40%

effective system utilizations greater than 8.0 are possible. Either the CM or the SRM may be used to achieve effective system utilizations greater than 8.0 with a speed-up of 8x. Further, only a speed-up of 4x is required in many cases. *Helping us answer question Q1 stated in Sec. 1, we observe that in the context of this experimental framework, a GPU must offer between 4x and 8x speed-up over a CPU on an eight-CPU system to overcome suspension-oblivious penalties and schedule more work than a CPU-only system with eight CPUs.*

Observation 4. *A GPU usually allows a twelve-CPU system to schedule more work than possible with only CPUs when the GPU offers a speed-up of 4x or greater.* We may make many of the similar observations we have made in Obs. 2 and Obs. 3 in Fig. 9 and the supplemental Fig. 10. However, the small required speed-up of 4x is somewhat unexpected. This small speed-up factor is the result of the small percentage of GPU-using tasks (which ranges up to 10%, but at least includes two in number) within the task set. Task sets are easier to schedule and allow greater levels of CPU-only utilization when there are few GPU-using tasks. This is explained in more detail in Obs. 9. *Helping us answer question Q1 stated in Sec. 1, we observe that in the context of this experimental framework, a GPU must offer about a 4x speed-up over a CPU on an twelve-CPU system to overcome suspension-oblivious penalties and schedule more work than a CPU-only system with twelve CPUs.*

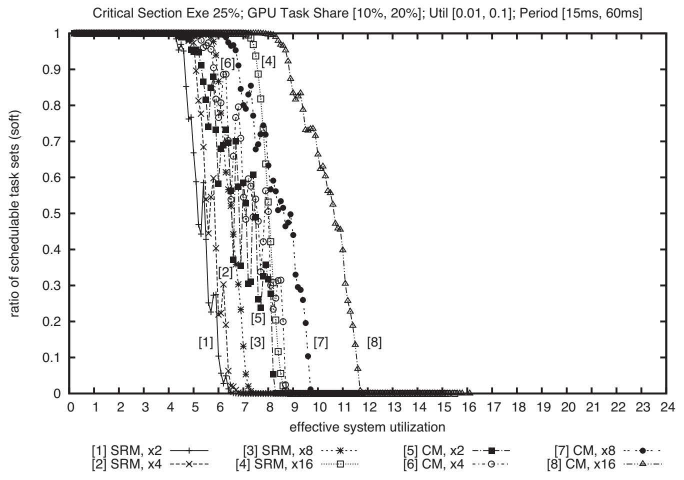
The SRM vs. the CM

We may also compare the performance of the SRM and the CM. Neither method is best in all scenarios.

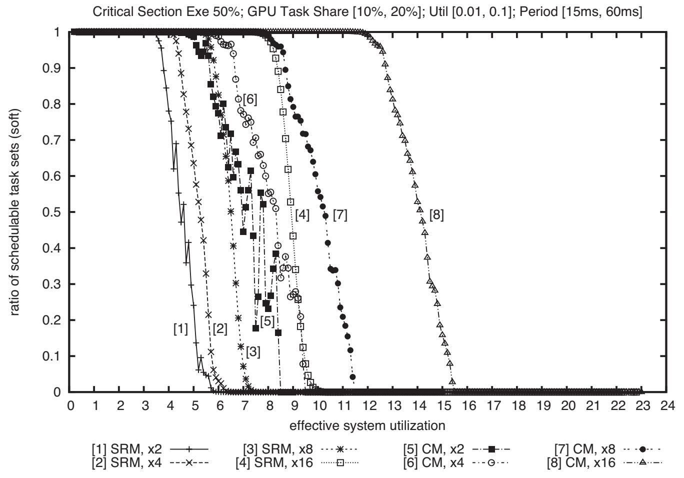
Observation 5. *The Container Method frequently offers better schedulability than the Shared Resource Method.* The CM can often schedule task sets the SRM cannot for all tested systems as illustrated by the large differences in the schedulability curves seen in insets (a), (b), (c), (e), and (f) of Figs. 6, 8, and 9. In the SRM, each task in $G(T)$ incurs an execution penalty of either $|G(T)|-1$ critical sections under the FMLP, or up to seven, 15, and 23 critical sections (recall that Eq. (7)

includes up to $2m - 1$ terms) for the four-, eight-, and twelve-CPU systems, respectively, under the OMLP. If the constraint given by Ineq. (9) is not violated, then there is still a good chance that the constraint of Ineq. (10) will be, especially at higher system utilizations. The CM clearly benefits from avoiding the inclusion of blocking terms in its schedulability analysis, despite the fact that its GPU utilization condition (Ineq. (11)) includes more CPU execution time.

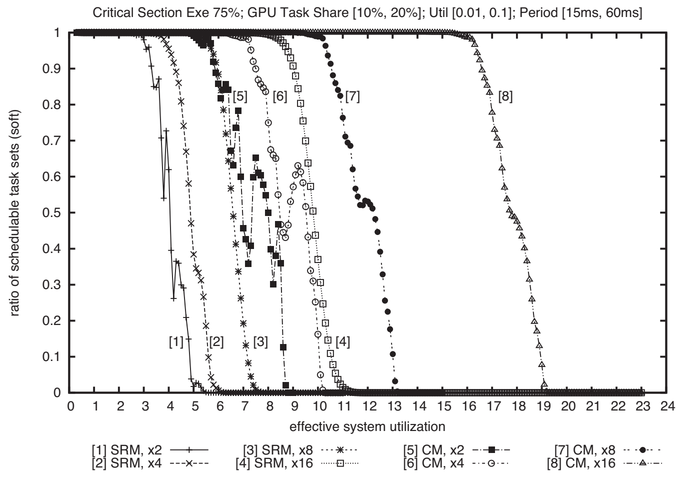
Observation 6. *The Shared Resource Method improves as per-task utilizations increase.* Observe in Fig. 8 (eight-CPU system) how the schedulability curves for the SRM improve across insets (a), (d), and (g). For example, the SRM with a speed-up of 16x may schedule task sets with effective utilizations of roughly at most 13.0 in inset (a). This cap increases to about 18.0 in inset (g), though the percentage size of the critical sections have remained constant. The SRM benefits from increased per-task utilizations since it reduces the total number of tasks in a given task set and hence also reduces the number of tasks in $G(T)$. This improves schedulability since fewer GPU-using tasks result in smaller cumulative blocking-term penalties. Similar observations may be made in Figs. 6 and 9 for the four- and twelve-CPU systems, respectively.



(a) GPU Usage 25%

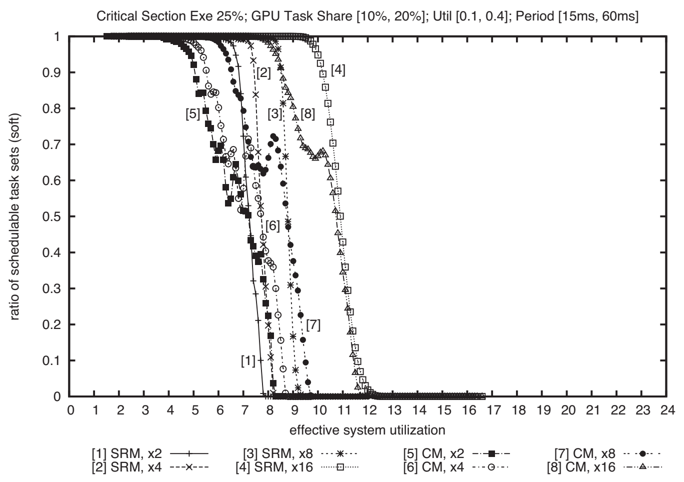


(b) GPU Usage 50%

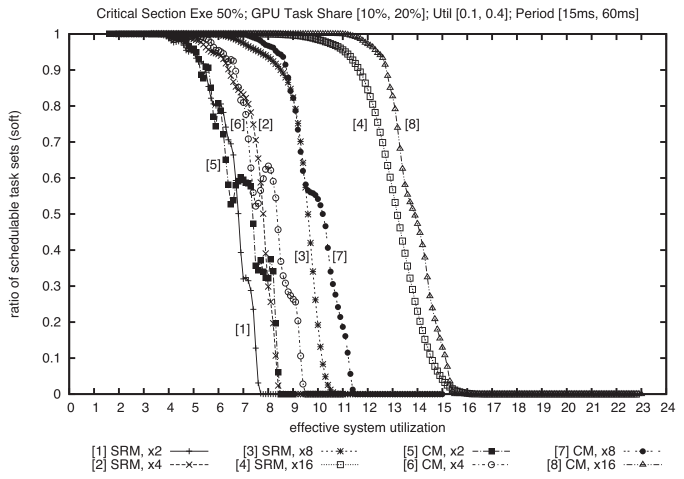


(c) GPU Usage 75%

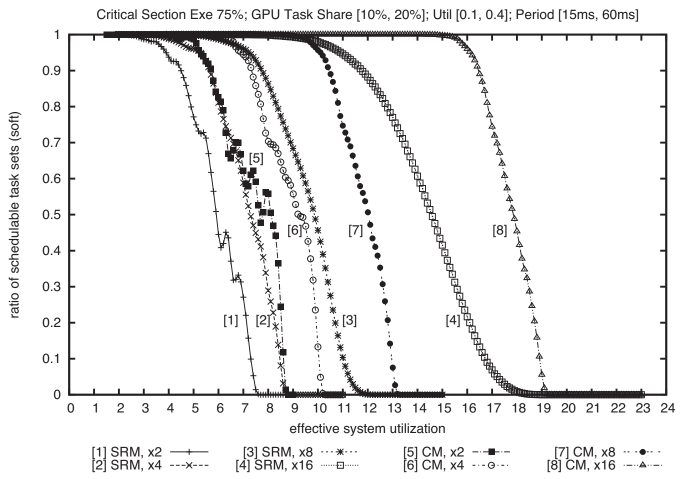
Figure 8: Eight-CPU System: Light Task Sets, GPU Percentage Task Share 10–20%



(d) GPU Usage 25%

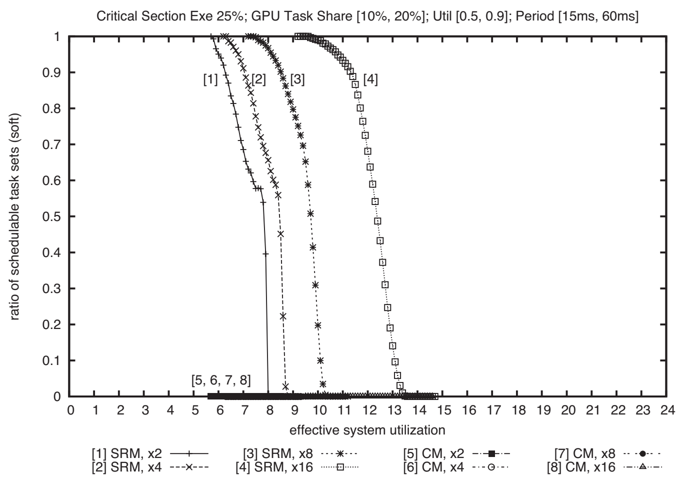


(e) GPU Usage 50%

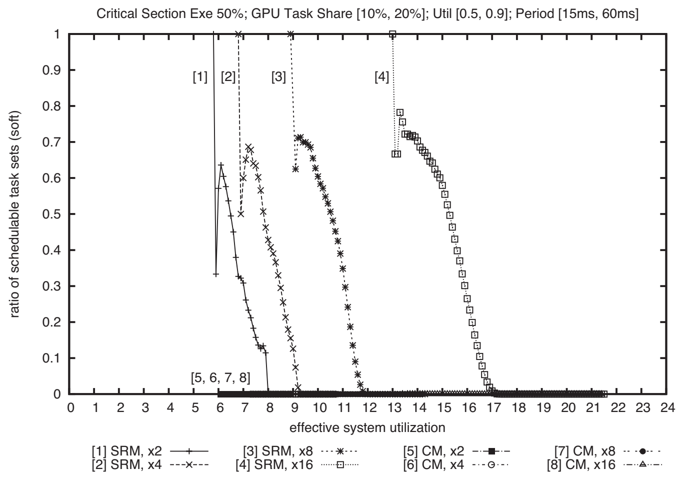


(f) GPU Usage 75%

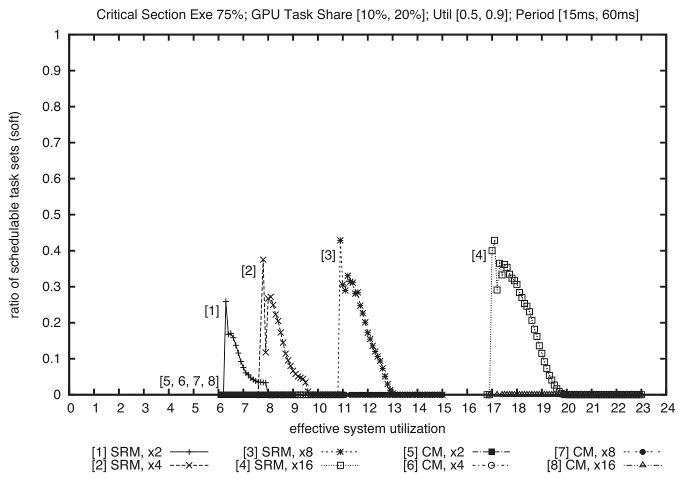
Figure 8: (continued) Eight-CPU System: Medium Task Sets, GPU Percentage Task Share 10–20%



(g) GPU Usage 25%



(h) GPU Usage 50%



(i) GPU Usage 75%

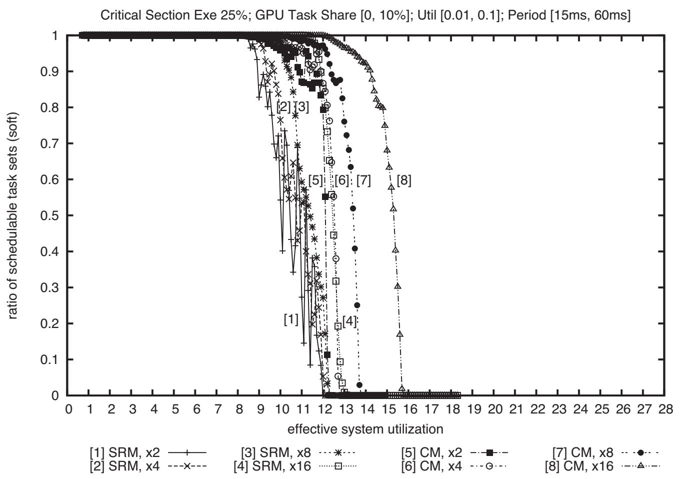
Figure 8: (continued) Eight-CPU System: Heavy Task Sets, GPU Percentage Task Share 10–20%

Observation 7. *The Container Method cannot schedule task sets with per-task utilizations greater than 0.5.* The CM cannot schedule any heavy task set due to its strict container bandwidth constraints as seen in the insets (g), (h), and (i) in Figs. 6 (four-CPU system) and 8 (eight-CPU system) as well as insets (a), (b), and (c) in supplemental Fig. 7 (four-CPU system) and supplemental Fig. 10 (twelve-CPU system). Recall that the condition given by Ineq. (11) must be met for a task set to be schedulable under the CM. A heavy task set is schedulable under the CM only if $G(T)$ contains two tasks with utilizations equal to 0.5. However, the occurrence of this case is highly improbable since utilizations are chosen at random. This illustrates that the CM may suffer from bin-packing limitations with heavy tasks.

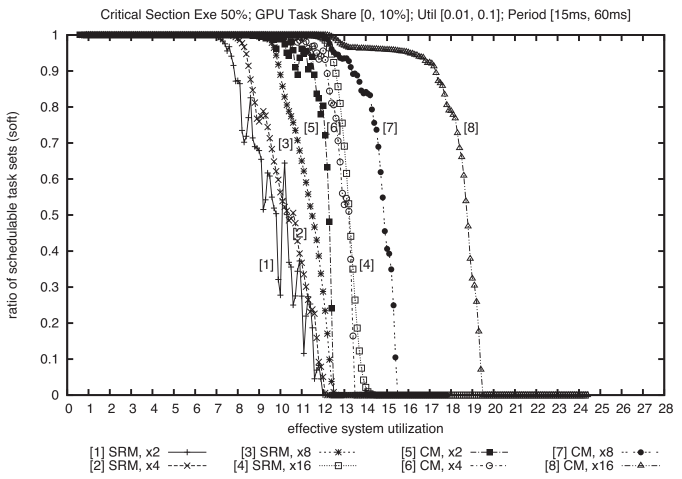
Observation 8. *The Container Method is best suited for systems with medium or light per-task utilizations.* While the CM may be used to varying degrees of success it frequently offers better schedulability than the SRM (Obs. 5). This margin is often significant, as is best observed in Fig. 6 (four-CPU system), inset (c), where the CM, with a speed-up of 16x, can schedule task sets with effective utilizations as great as 15.0 in comparison to the SRM at 7.0. Further, in cases where the SRM offers better schedulability than the CM (Fig. 7(d)), the CM is still competitive.

Most Efficient Offloading of Work

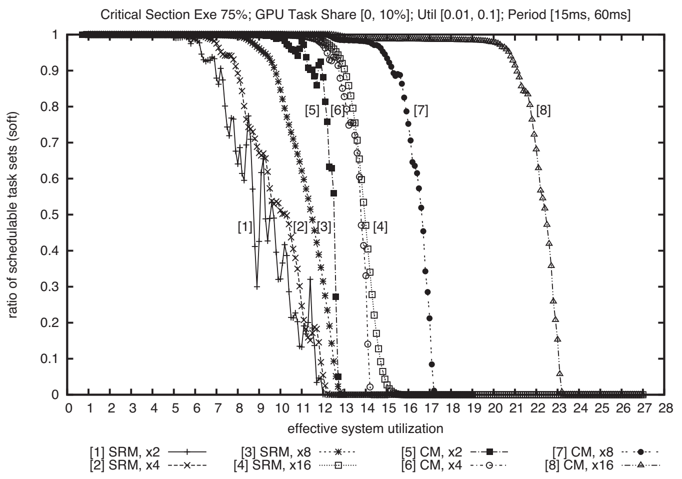
Additional observations may be made when we closely compare the effect the GPU percentage task share has on the SRM and the CM. Figs. 11, 12, and 13 make these comparisons for the four-, eight-, and twelve-CPU systems, respectively. Inset (a) plots schedulability curves for the SRM, and inset (b) for the CM, in each of these figures. We can easily observe the effect of GPU percentage task share when all schedulability curves for all task shares are plotted together. For example, Fig. 11(a) plots schedulability under the SRM for a speed-up of 16x with all tested GPU percentage task shares for medium task sets with critical section lengths of 75% of execution. We can observe in this figure that the GPU percentage task share with the greatest possible effective system utilization is $[30\%, 40\%]$ (though a GPU percentage task share of $[20\%, 30\%]$ initially has better performance) as this curve is farthest to the right, obtaining greater degrees of schedulable effective system utilizations.



(a) GPU Usage 25%

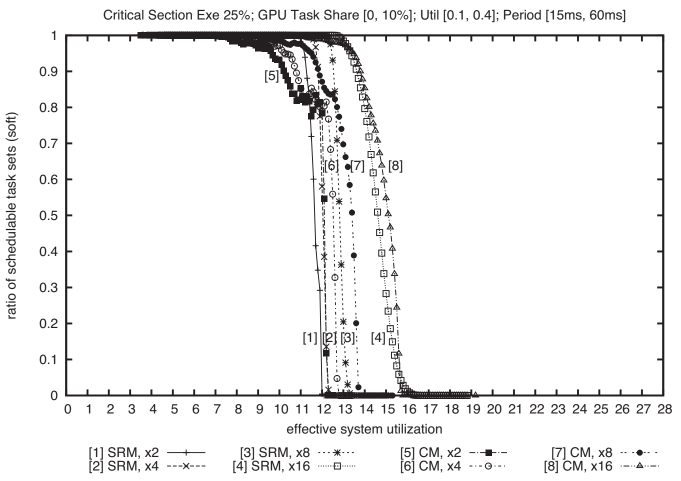


(b) GPU Usage 50%

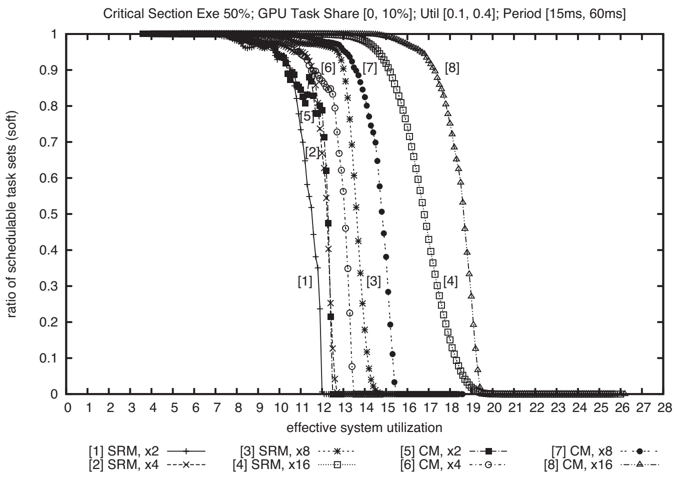


(c) GPU Usage 75%

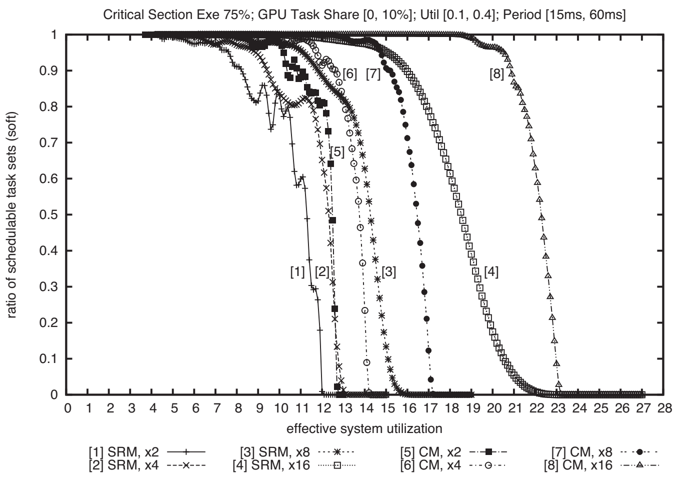
Figure 9: Twelve-CPU System: Light Task Sets, GPU Percentage Task Share 0^{+}–10%



(d) GPU Usage 25%

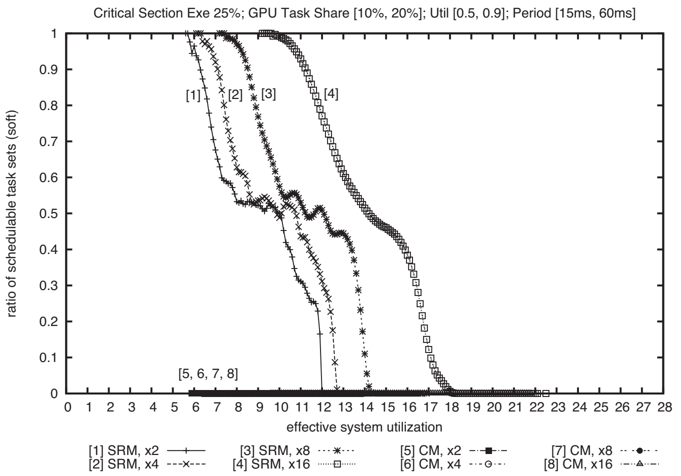


(e) GPU Usage 50%

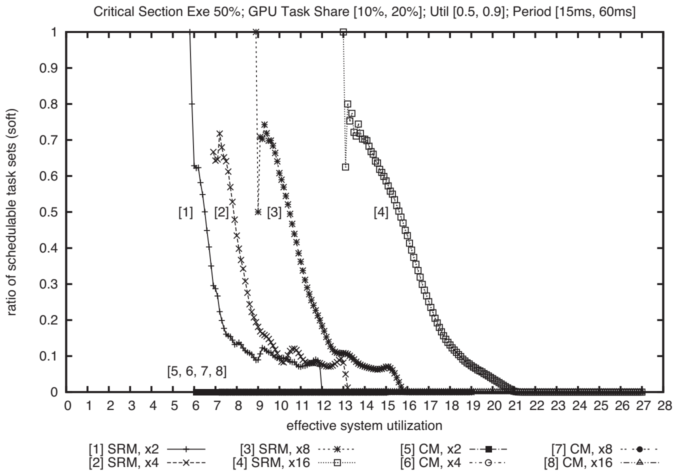


(f) GPU Usage 75%

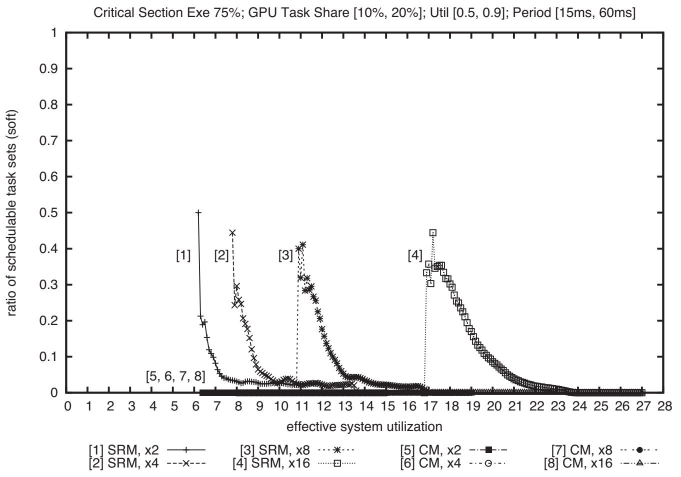
Figure 9: (continued) Twelve-CPU System: Medium Task Sets, GPU Percentage Task Share 0^{+}–10%



(g) GPU Usage 25%

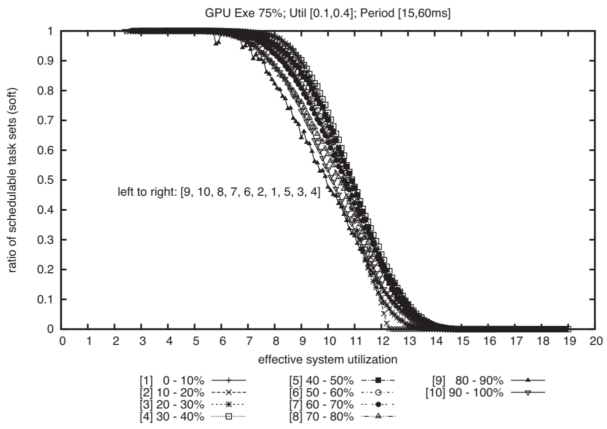


(h) GPU Usage 50%

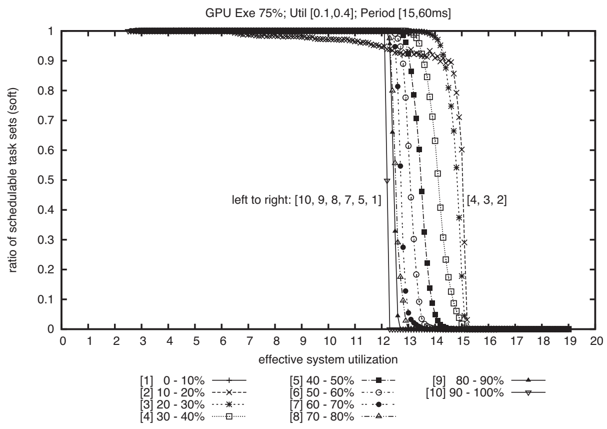


(i) GPU Usage 75%

Figure 10: Supplemental, Twelve-CPU System: Heavy Task Sets, GPU Percentage Task Share 10–20%

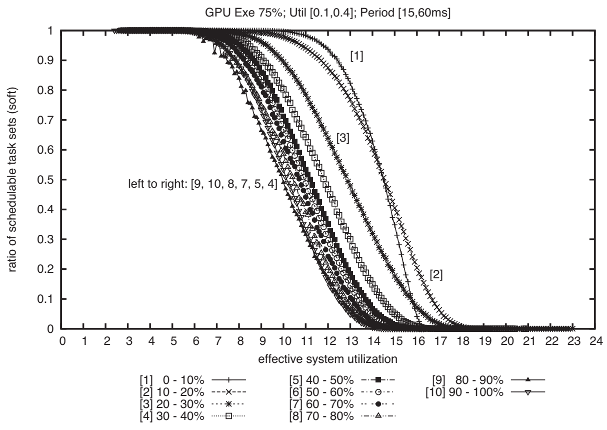


(a) SRM

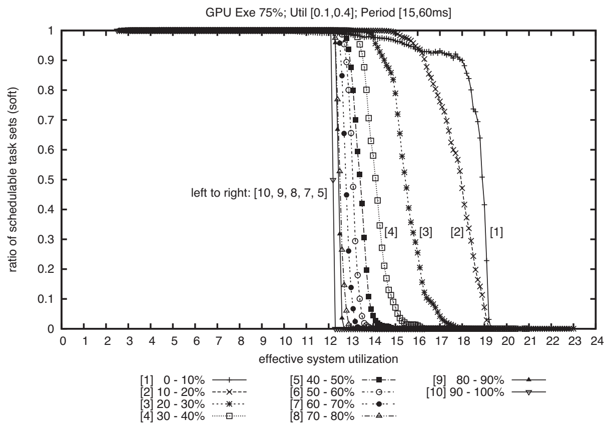


(b) CM

Figure 11: Four-CPU System: The effect of GPU percentage task share.



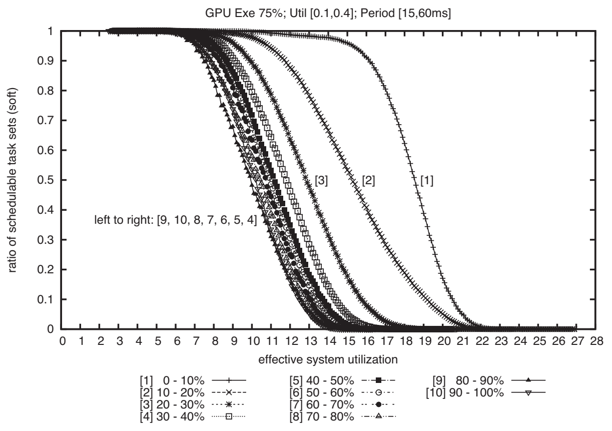
(a) SRM



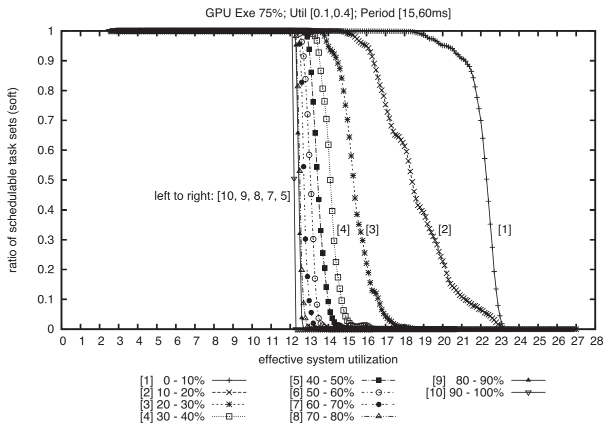
(b) CM

Figure 12: Eight-CPU System: The effect of GPU percentage task share.

Observation 9. *Schedulable effective system utilizations are maximized when GPU percentage task share is roughly* $1/m$*, yielding a balance between GPU-using and CPU-only tasks.* It may be observed in every inset of Figs. 11, 12, and 13 that the curve with greatest possible effective system utilizations are usually for GPU percentage task share ranges that include $1/m$. Indeed, it is for this reason the subset of figures in Figs. 6, 8, and 9 were selected for presentation, where the lower



(a) SRM



(b) CM

Figure 13: Twelve-CPU System: The effect of GPU percentage task share.

and upper interval bounds on GPU percentage task share include, or are very close to, the value $1/m$. For example, consider the eight-CPU system where we find that $1/m = 1/8 = 12.5\%$. The GPU percentage task share interval [10%, 20%] includes 12.5% and we see that it is this interval in Fig. 12(a) that reaches the greatest level of schedulable effective system utilizations.

This behavior can be understood when we consider GPU utilization and suspension-oblivious analysis. Due to our randomized task set generation methods, on average, GPU-using tasks of a given percentage share also contribute an equal or greater share towards the total suspension-oblivious system utilization when execution times are inflated by suspension times (note that this is not effective system utilization). Recall from our discussion of the CM (Sec. 4.2) that the GPU is fully utilized when the GPU-using tasks' total suspension-oblivious system utilization is close to 1.0, or one CPU. Thus, the GPU-using tasks maximize the GPU when the GPU percentage task share is about $1/m$, or the utilization share of one CPU. If GPU task percentage share is much less than $1/m$, then the GPU is likely not fully utilized and maximum processing capabilities are not realized. If the GPU task percentage share is much greater than $1/m$, then the GPU is likely overutilized and thus the task set is unschedulable, or the GPU is fully utilized and the number of CPU-only tasks must be significantly decreased (likely resulting in decreased CPU utilization given fixed per-task utilization constraints) to maintain the high GPU task percentage share ratio.

This observation answers (in the context of this experimental framework) question Q2, regarding how much work should be offloaded onto a GPU to make the most efficient use of both the system CPUs and GPU.

Observation 10. *The GPU may become a bottleneck.* This observation may be obvious, but it is important to keep in mind when developing a real-time system with a GPU co-processor. This is because GPUs are rarely viewed as system bottlenecks due to their role as an accelerator. However, the GPU can become overutilized like any other resource, leaving the CPUs relatively idle. This can be thought of as a corollary of Obs. 9. This bottleneck effect is best observed in Fig. 13(b) (twelve-CPU system) for the GPU percentage task share of 90–100% where the schedulability ratio drops suddenly from 100% to 0% when effective system utilization is greater than 12.0.

This concludes our theoretical analysis of the SRM and the CM, but we have yet to determine if these solutions are applicable to real systems. Are they truly necessary? What behaviors might we encounter? We performed implementation-based experiments to help answer these questions.

6 Real-World Implementation

We implemented the SRM with the OMLP in LITMUS$^{\text{RT}}$ (described in detail in [18]), a UNC-produced Linux-based testbed for real-time schedulers. We did this to both evaluate the practical performance characteristics of our solution and, more importantly, to show that *unguarded GPU device driver access is not viable for a real-time system*—some real-time control is necessary.

We generated synthetic workloads using the same techniques and task set parameters as in Sec. 5.1 and ran them on our test platform, the same Intel Core i7 quad-core system described in Sec. 2. The synthetic task sets were executed with G-EDF scheduling for a duration of 2.5 minutes under two scenarios: one with the SRM and one without. Each job executed dummy busy-wait code to simulate computations for the task-specified durations of e_i and s_i for the CPU and GPU, respectively; jobs arrived periodically, as specified by p_i. GPU-using jobs running under the SRM scenario had to first acquire the OMLP lock (via a LITMUS$^{\text{RT}}$ system call) protecting the

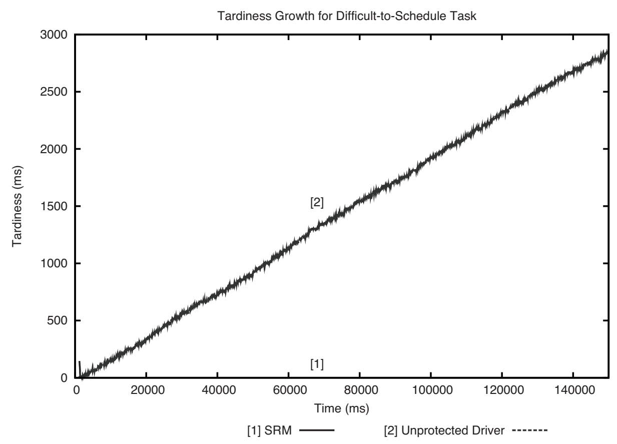
	Average Response Time		Average Tardiness	
Category	SRM	Driver	SRM	Driver
Easy	25.00%	24.95%	0.02%	0.00%
Difficult	29.33%	34.89%	0.17%	4.64%
Unable	92.79%	134.50%	91.97%	133.50%

Table 3: Response time and tardiness statistics for the SRM and unguarded driver. “Average Response Time” refers to the average “response time/task period” ratio over all task sets. Ê“Average Tardiness” refers to the average “tardiness/task period” ratio over all task sets. Smaller values are better.

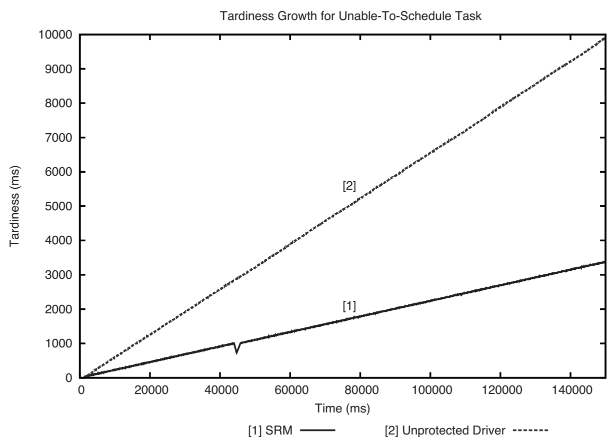
GPU before beginning use of the GPU. We made measurements for response time and tardiness. A total of 400 task sets were tested under each scenario.

A summary of our findings for medium-utilization task sets is shown in Table 3. A large amount of data was generated in these tests and cannot be presented in detail due to page constraints, so only high-level statistics are shown. We use the ratios *response time/task period* and *tardiness/task period* to interpret our data as this allows measurements involving task sets with different period ranges to be compared.

The executed task sets are organized into *easy*-, *difficult*-, and *unable-to-schedule* categories. Easy-to-schedule task sets are those that are deemed schedulable by the theoretical analysis of Sec. 4.1. Difficult-to-schedule task sets are those for which theoretical analysis was unable to determine schedulability, but the observed tardiness of any job of any task T_i never exceeded p_i. While 2.5 minutes of execution cannot *prove* schedulability, it indicates that the task set may be schedulable. We make this assumption here. Unable-to-schedule task sets are those that could not be successfully scheduled (tardiness exceeded p_i during execution) by the implementation—no unable-to-schedule task sets were ever deemed schedulable by the theoretical analysis. We are interested in such cases since soft real-time systems may be provisioned assuming average-case processing requirements. Such a system may at times become temporarily overutilized and we desire reasonable performance under such circumstances.



(a)



(b)

Figure 14: The SRM exhibits superior control over tardiness. (a) Difficult-to-Schedule task set: The task set is only schedulable using the SRM. (b) Unable-to-schedule task set: Though also unschedulable using the SRM, the rate of increasing tardiness is much reduced in comparison to the unprotected driver.

Observation 11. *The Shared Resource Method offers real-time guarantees with little or no observed cost.* The SRM allows GPU-using tasks to be scheduled with real-time guarantees through the use of predictable locking mechanisms, though performance is slightly hindered for easy-to-schedule task sets (as seen in Table 3, average response time and average tardiness are slightly less [better] for the unprotected driver driver in this case). However, the marginally better performance of the unguarded driver for easy-to-schedule task sets comes at the expense of significantly increased CPU utilization since the driver reduces latency through busy-wait spinning as was observed in Sec. 2. It is likely that the CPUs could potentially handle additional processing in such cases. Nevertheless, the driver's spinning and lack of priority inheritance becomes a liability in task sets where resources are more taxed as seen in the greater ratios of the difficult- and unable-to-schedule categories, where the SRM clearly shows better performance with lower average response times and less tardiness.

Observation 12. *The Shared Resource Method is superior at controlling job tardiness.* G-EDF scheduling distributes tardiness relatively equally across all tasks in both the SRM and unguarded driver scenarios. However, tardiness growth is much better controlled under the SRM as can be observed in both insets of Fig. 14. This figure depicts the observed tardiness over time for a task from a difficult-to-schedule (inset (a)) and an unable-to-schedule (inset (b)) task set. The task set in Fig. 14(a) was observed to be schedulable under the SRM, but observed to be unschedulable using the unprotected driver. Here, the task under the SRM experiences no tardiness, while tardiness for the task with the unprotected driver grows indefinitely. This control over tardiness is also exhibited in the unable-to-schedule task sets, as shown Fig. 14(b). Though tardiness grows indefinitely under both methods, the rate of growth is much reduced under the SRM. This behavior is desirable for a soft real-time system that may occasionally become overutilized as it offers better performance during periods of overutilization and will also recover more quickly once this temporary condition is over since the system is less backlogged.

7 Future Work

In future work, we intend to investigate how the SRM may be improved to support the exploitation of asynchronous memory transfers. Discrete graphics cards may support the ability for graphics hardware to send and receive data to one task while the GPU itself performs computations for another. This allows for the masking of communication overheads in a pipelined manner. The current treatment of critical sections precludes the use of such a mechanism.

Another direction we may pursue is support for multi-GPU platforms. Platforms with many GPUs (sometimes heterogeneous) are already available at consumer prices. It is feasible to design a system that could dynamically choose to execute a particular task or job on one of multiple CPUs or on a variety of GPUs. If a SRM-like approach is taken, the locks protecting GPUs become k-exclusion locks, [11] thus adding an extra dimension of complexity. Furthermore, the use of multiple GPUs raises the issue of bus contention; in experimental tests, we have seen such contention more than double the data transfer time between the CPU and GPU in some chipset architectures. Also, execution times of tasks can vary if heterogeneous GPUs with differing capabilities are used.

[11] k-exclusion locks protect a resource or resource pool, allowing up to k simultaneous accesses.

We are also interested in the trade-offs between partitioned, clustered, and global scheduling for systems with GPUs. In the case of partitioned and clustered scheduling, it is not entirely clear what an optimal grouping for tasks using a GPU may be. Each scheduling method introduces its own constraints with regards to schedulability analysis and locking protocols.

Finally, we plan to perform in-depth empirical analysis to determine the gap between the theoretical schedulability results presented in this paper and apparent schedulability in a real system. Rigorous empirical tests should further clarify when a GPU is beneficial in "real world" real-time systems. Such a study would also better quantify the effect of GPU interrupt handling.

8 Conclusion

Recent advances in graphics hardware are enabling the acceleration of computations traditionally carried out on CPUs. The use of such hardware in a real-time system may allow workloads to be supported that are too computationally intensive for CPU-only systems, while also benefiting from reduced power consumption. Through the consideration of current architectural constraints, this paper has presented two methods for integrating GPUs into soft real-time multiprocessor systems: the Shared Resource Method, and the Container Method. Schedulability experiments were presented that assess the schedulability characteristics of each. In the context of the experimental framework and analytical model, it was found that a GPU often need only offer a relatively modest 4x speed-up over a single CPU to schedule greater computational workloads than pure CPU systems in common cases. Additionally, it was also determined both CPU and GPU resources are maximized when GPU-using tasks make up roughly $1/m$ percent, where m is the number of system CPUs, of the tasks in a given task set. The Shared Resource Method was also evaluated through implementation in a Linux-based operating system, exercising manufacturer device drivers, and shown that it exhibited superior runtime characteristics in terms of schedulability and efficient resource utilization in comparison to a similar system that is oblivious to GPU hardware and device driver behaviors.

References

[1] AMD Fusion Family of APUs. Available from: http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf.

[2] ATI Stream Technology. Available from: http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspx.

[3] China's new nebulae supercomputer is no. 2. Available from: http://www.top500.org/lists/2010/06/press-release.

[4] CUDA community showcase. Available from: http://www.nvidia.com/object/cuda_apps_flash_new.html.

[5] CUDA Zone. Available from: http://www.nvidia.com/object/cuda_home_new.html [cited January 12, 2011].

[6] GeForce graphics processors. Available from: http://www.nvidia.com/object/geforce_family.html.

[7] Intel details 2011 processor features, offers stunning visuals build-in. Available from: http://download.intel.com/newsroom/kits/idf/2010_fall/pdfs/Day1_IDF_SNB_Factsheet.pdf.

[8] Intel microprocessor export compliance metrics. Available from: http://www.intel.com/support/processors/xeon/sb/CS-020863.htm.

[9] Microsoft DirectX. Available from: http://www.gamesforwindows.com/en-US/directx/.

[10] OpenCL. Available from: http://www.khronos.org/opencl/.

[11] Parallel computing with SciFinance. Available from: `http://www.scicomp.com/parallel_computing/SciComp_NVIDIA_CUDA_OpenMP.pdf`.

[12] G. Abhijeet and T. Ioane Muni. GPU based sparse grid technique for solving multidimensional options pricing PDEs. In *Proceedings of the 2nd Workshop on High Performance Computational Finance*, pages 1–9, November 2009.

[13] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics*, pages 145–149, August 2009.

[14] S. Baruah. Scheduling periodic tasks on uniform processors. In *Proceedings of the EuroMicro Conference on Real-time Systems*, pages 7–14, June 2000.

[15] S. Baruah. Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, pages 37–46, December 2004.

[16] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, August 2007.

[17] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 49–60, December 2010.

[18] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, December 2006.

[19] S. Childs and D. Ingram. The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. In *Proceedings of the 7th Real-Time Technology and Applications Symposium*, pages 135–, 2001.

[20] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Real-Time Systems*, volume 38, pages 133–189, February 2008.

[21] A. Dwarakinath. A fair-share scheduler for the graphics processing unit. Master's thesis, Stony Brook University, 2008.

[22] J. Erickson, U. Devi, and S. Baruah. Improved tardiness bounds for global EDF. In *Proceedings of the 22nd EuroMicro Conference on Real-Time Systems*, pages 14–23, July 2010.

[23] S. Funk, J. Goossens, and S. Baruah. On-line scheduling on uniform multiprocessors. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 183–202, December 2001.

[24] P. Gai, L. Abeni, and G. Buttazzo. Multiprocessor DSP scheduling in system-on-a-chip architectures. In *Proceedings of the 14th EuroMicro Conference on Real-Time Systems*, pages 231–238, 2002.

[25] O. Harrison and J. Waldron. Practical symmetric key cryptography on modern graphics hardware. In *Proceedings of the 17th Conference on Security Symposium*, pages 195–209, July 2008.

[26] W. Kang, S. H. Son, J. A. Stankovic, and M. Amirijoo. I/O-aware deadline miss ratio management in real-time embedded databases. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 277–287, December 2007.

[27] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 459–468, December 2009.

[28] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Resource sharing in GPU-accelerated windowing systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Application Symposium*, April 2011.

[29] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[30] K. Lakshmanan, S. Kato, and R. Rajkumar. Open problems in scheduling self-suspending tasks. In *Proceedings of the 1st International Real-Time Scheduling Open Problems Seminar*, pages 12–13, July 2010.

[31] H. Leontyev and J. Anderson. A hierarchical multiprocessor bandwidth reservation scheme with timing guarantees. *Real-Time Systems*, 43(1):60–92, September 2009.

[32] R. Pellizzoni G. Lipari. Holistic analysis of asynchronous real-time transactions with earliest deadline scheduling. *Journal of Computer and System Sciences*, 73:186–206, March 2007.

[33] N. Manica, L. Abeni, and L. Palopoli. Qos support in the x11 window system. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 103–112, 2008.

[34] P. Muyan-Ozcelik, V. Glavtchev, J. M. Ota, and J. D. Owens. Real-time speed-limit-sign recognition an embedded system using a GPU. *GPU Computing Gems*, pages 473–496, 2011.

[35] C. Y. Ong, M. Weldon, S. Quiring, L. Maxwell, M. Hughes, C. Whelan, and M. Okoniewski. Speed it up. *Microwave Magazine, IEEE*, 11(2):70–78, 2010.

[36] B. Pieters, C. F. Hollemeersch, P. Lambert, and R. Van de Walle. Motion estimation for H.264/AVC on multiple GPUs using NVIDIA CUDA. In *Applications of Digital Image Processing XXII*, volume 7443, page 74430X, September 2009.

[37] G. Raravi and B. Andersson. Calculating an upper bound on the finishing time of a group of threads executing on a GPU: A preliminary case study. In *Work-in-progress session of the*

16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pages 5–8, August 2010.

[38] J. E. Sasinowski and J. K. Strosnider. ARTIFACT: a platform for evaluating real-time window system designs. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 342–352, 1995.

[39] Y. Watanabe and T. Itagaki. Real-time display on Fourier domain optical coherence tomography system using a graphics processing unit. In *Journal of Biomedical Optics*, volume 14, page 060506, December 2009.