Real-World Constraints of GPUs in Real-Time Systems*

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

Abstract

Graphics processing units (GPUs) are becoming increasingly important in today's platforms as their increased generality allows for them to be used as powerful co-processors. In this paper, we explore possible applications for GPUs in real-time systems, discuss the limitations and constraints imposed by current GPU technology, and present a summary of our research addressing many such constraints.

1 Introduction

The parallel architecture of the graphics processing unit (GPU) often allows data parallel computations to be carried out at rates orders of magnitude greater than those offered by a traditional CPU. Enabled by increased programmability and single-precision floating-point support, the use of graphics hardware for solving non-graphical (general purpose) computational problems began gaining wide-spread popularity in the early part of the last decade [10, 20, 23]. However, early approaches were limited in scope and flexibility because non-graphical algorithms had to be mapped to languages developed exclusively for graphics. Graphics hardware manufactures recognized the market opportunities for better support of general purpose computations on GPUs (GPGPU) and released language extensions and runtime environments,[1] eliminating many of the limitations found in early GPGPU solutions. Since the release of these second-generation GPGPU technologies, both graphics hardware and runtime environments have grown in generality, increasing the applicability of GPGPU to a breadth of domains. Today, GPUs can be found integrated on-chip in mobile devices and laptops [1, 6, 3], as discrete cards in higher-end consumer computers and workstations, and also within some of the world's fastest supercomputers [22].

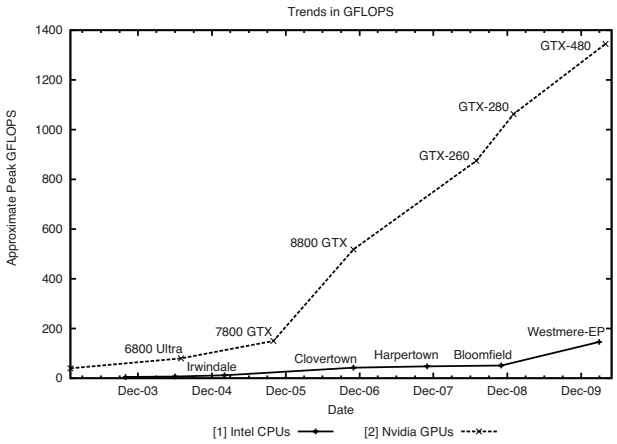

Figure 1: Historical trends in processor performance in terms of approximate peak floating-point operations per second (FLOPS).

There are strong motivations for utilizing GPUs in real-time systems. 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 floating-point performance trends of Intel CPUs and NVIDIA GPUs over much of the past decade [5, 7, 28]. 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]. 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, especially in integrated on-chip designs. This is an ideal feature for embedded and cyber-physical systems.

In this paper we identify several real-time applications that may benefit from the use of GPUs. In particular, we

*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.

[1] Notable platforms include the Compute Unified Device Architecture (CUDA) from NVIDIA [4], Stream from AMD/ATI [2], OpenCL from Apple and the Khronos Group [9], and DirectCompute from Microsoft [8].

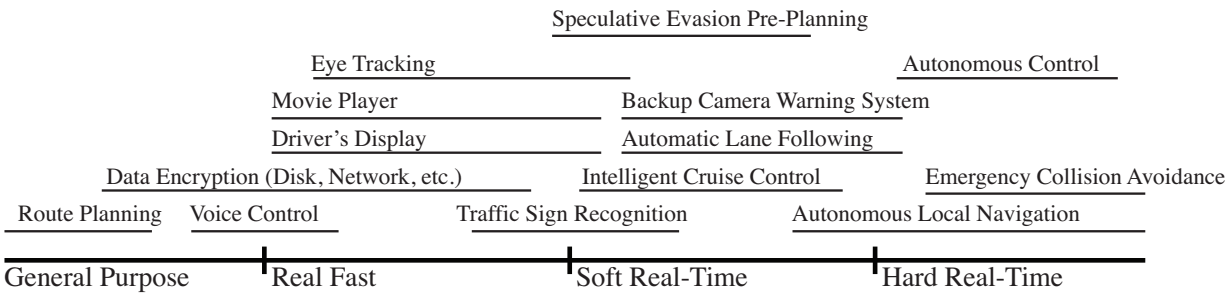

Figure 2: Spectrum of possible temporal requirements for a number of automotive applications that may utilize a GPU. Each feature may cross domains, as indicated by the line beneath each feature name.

note that the use of GPUs appears to be the only economically feasible solution able to meet the processing requirements of advanced driver-assist and autonomous features in future automotive applications. Unfortunately, there are obstacles created by current GPU technology that must be overcome before GPUs can be incorporated into real-time systems. In this paper we discuss several of these obstacles and present a summary of solutions we have found through our research to date. We hope to engage the real-time and cyber-physical systems communities to identify additional applications where the use of GPUs may be beneficial or even necessary. Through further research and the development of a breadth of applications, we hope to inspire GPU manufactures to incorporate features into their products to improve real-time behaviors.

This paper is organized as follows. In the next section, we present several applications where GPUs may be beneficial in real-time systems. In Sec. 3, we present the unique constraints imposed by current GPU technology that pose challenges to the use of GPUs in real-time systems. In Sec. 4, we present a summary of solutions that we have developed that address several of these constraints and allow GPUs to be used in real-time systems. In Sec. 5, we present future directions for our research and discuss what changes may be necessary in current GPU technology to better support real-time systems. Finally, in Sec. 6, we conclude with remarks on the field of real-time GPUs.

2 Real-Time GPU Applications

There are a number of real-time domains where GPUs may be applied. For example, a GPU can efficiently carry out many digital signal processing operations such as multidimensional FFTs and convolution as well as matrix operations such as factorization on data sets of up to several gigabytes in size. These operations, coupled with other GPU-efficient algorithms, can be used in medical imaging and video processing, where real-time constraints are common. Additionally, a particularly compelling application for real-time GPUs is that of automobiles.

GPUs can be used to implement a number of system features in the automotive domain. For user interface features, a GPU may be used to realize rich displays for the vehicle operator and to implement responsive voice-based controls [16], all while possibly driving video entertainment displays for other passengers simultaneously. Further, a GPU can also be used to track the eyes of the vehicle operator [24]. Such tracking could be used to implement a number of safety features. Real-time applications for GPUs in automobiles become even more apparent when we consider driver-assist and autonomous vehicle features. In these platforms, multiple streams of data from video feeds, laser range sensors, and radar can be processed and correlated to provide environmental data for a number of vehicle functions. This data can be used for automatic sign recognition [27], local navigation (such as lane following), and obstacle avoidance [29]. GPUs are well suited to handle this type of workload since these sensors generate enormous amounts of data. Indeed, GPUs are likely the only efficient and cost-effective solution. Moreover, these are clearly safety-critical applications where real-time constraints are important.

Fig. 2 depicts a number of automobile features that could make use of a GPU. These features are plotted along a spectrum of temporal requirements showing our view of the relative need for real-time performance. The spectrum is broken up into four regions: general-purpose, "real-fast,"[2] soft real-time, and hard real-time. Features in the general-purpose region are those that could possibly be supported by general-purpose scheduling algorithms, though may still be a part of a real-time system. The "real-fast" region captures applications that may have general

[2]The term "real-fast" is borrowed from Paul McKenny [26].

quality-of-service requirements or must exhibit low latency behaviors. The soft real-time region has features that may require some level of temporal guarantees, such as bounded deadline tardiness. Finally, the hard real-time region captures features that have strong safety repercussions if temporal guarantees are not met. It should be noted that an automobile manufacture might opt to put conservative real-time requirements on features to both aid in vehicle certification from governmental bodies and to also reduce exposure to legal liability in case of an accident.

3 GPU Constraints

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 current on-chip architectures. With this interface, a GPU is not a system programmable device, i.e., an OS cannot directly schedule or otherwise control a GPU. Instead, a driver in the OS manages all system GPUs. This imposes several constraints on a real-time system. First, due to the fact that high-performance drivers are closed source, a real-time system must be isolated from unknown behaviors of the driver. 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. Second, computer graphics and high-performance computing are the main markets for current GPU technology. The needs of these non-real-time domains drive many hardware and software features. As a result, driver software (and even GPU hardware) is throughput-oriented and optimized for use by a single process at a time; low latency of operations and the sharing of GPUs among processes, very important features for a real-time system, are only supported to a limited degree.

4 Summary of Research

The constraints imposed by current GPU technology do not make it impossible to integrate GPUs into real-time systems. In our research, we have found ways to implement real-time GPU resource arbitration, effectively removing the driver's unknown arbitration methods from impacting real-time tasks. Further, we have developed and implemented methods to manage GPU device interrupts, reducing their impact on real-time tasks by significantly shortening the duration of priority inversions that interrupts may cause.

GPU Resource Management. In [17], we discovered several issues stemming from the above constraints that must be addressed before GPUs can be integrated into a real-time system.[3] First, the execution of a GPU program is non-preemptive. This may cause priority inversions, as high-priority tasks may have to wait until one or more lower-priority GPU-using tasks complete. Second, when a single GPU comes under contention, blocked tasks spin-wait, consuming CPU resources and budget, while waiting for the GPU resource. While spinning may be the best blocking mechanism in a real-time system for short durations [14], this becomes a major limitation for systems with GPUs since the execution of a GPU program can last from tens of milliseconds to several seconds, depending upon the application. This can negatively affect the timely execution of other real-time tasks. Finally, since GPU drivers are designed for general-purpose operating systems,[4] they have no notion of task priority. Thus it is difficult to quantify the effects of a GPU driver has on real-time systems.

Also in [17], we presented two solutions to address these limitations for globally-scheduled multiprocessor real-time systems with a single GPU. The first solution, which we call the Container Method (CM), groups all GPU-using tasks into a container that models a single logical processor. The container is given an execution budget equal to the utilization of the contained GPU-using tasks. A hierarchical scheduler is then used to schedule the real-time system. Non-GPU-using tasks and the GPU container are scheduled on the multiprocessor system using the global earliest-deadline-first algorithm. When the GPU container is scheduled to execute, a first-in-first-out (FIFO) secondary scheduler selects which of the contained GPU-using tasks to run. Because GPU-using tasks are contained within one logical processor and scheduled in FIFO order, it is impossible for the GPU to ever come under contention by multiple tasks since only one GPU-using task may execute at a time and must execute to completion before another may commence. As a result, the major limitations of non-preemptivity, spin-waiting, and lack of respect for scheduling priority become non-issues. While this solution may work well for many real-time systems, the FIFO scheduling of GPU-using tasks can be inefficient since these tasks often require normal CPU execution time as well—there is no reason why these tasks cannot execute simultaneously as long as the GPU is not

[3] To date, we have limited our focus to the CUDA platform for NVIDIA GPUs. NVIDIA is widely recognized as the leader in GPGPU products.

[4] Currently, no GPU drivers support GPGPU computation for any commercially available real-time operating system. Further, they do not appear to respect system priorities as used in general-purpose operating systems.

under contention.

To address this inefficiency, we also developed the Shared Resource Method (SRM), which treats the GPU as a shared resource protected by a real-time mutual exclusion locking protocol. This effectively removes the GPU driver from resource arbitration decisions, so spin-waiting never occurs. Further, the use of a real-time locking protocol respects system priorities and uses priority inheritance techniques to reduce durations of priority inversions. This allows us to bound the effects of a GPU in a real-time system.

In schedulability experiments, we found that both the CM and SRM are able to schedule greater computational workloads than pure CPU systems in common cases, where the GPU offers at least a 4x speedup over a single CPU. Indeed, we found that tasks sets with common characteristics and an effective CPU utilization[5] as great as 15.0 could be scheduled on a soft real-time system of four CPUs with a single GPU. Such a system configuration is conceivable for an embedded platform using today's technology.

As part of the research effort described in [17], we implemented the SRM in our Linux-based real-time operating system, LITMUS^{RT} [13]. We found in execution experiments that the SRM offered superior control over deadline tardiness in comparison to a real-time system without the SRM. Indeed, some task sets could not be scheduled at all without the SRM.[6] An example of such a case is depicted in Fig. 3, where the tardiness of a GPU-using task grows unboundedly over time when the default GPU driver behaviors are relied upon. In contrast, there is no observed tardiness when the SRM is in use. This result shows that a real-time mechanism is required to integrate a GPU with available software drivers into a real-time system.

More recently, we have extended the SRM to support systems with a pool of several GPUs [18]. Minimizing blocking time is critically important for GPU systems since GPU programs execute for long, non-preemptive, durations. This may result in conservative system provisioning, which is only exacerbated by large bounds on worst-case blocking time. The development of a new real-time k-exclusion locking protocol, the *Optimal k-Exclusion Global Locking Protocol* (O-KGLP), to efficiently manage the GPU resources in a globally scheduled real-time system was required in order to minimize blocking time. Using techniques inspired by [11, 12], the

[5]We define *effective CPU utilization* of a task set to be the utilization of the task set if it were implemented and scheduled on a CPU-only platform.

[6]In our experiments, we considered task sets to be schedulable on our soft real-time system if the observed tardiness of any task never exceeded its period after two and a half minutes of task set execution.

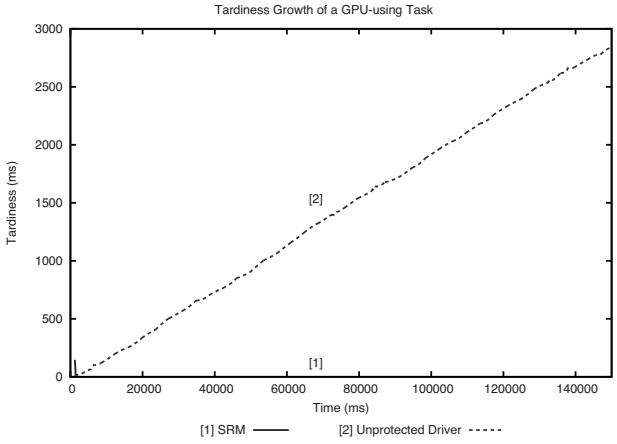

Figure 3: Tardiness growth can be unbounded when the default GPU driver behaviors are in effect, as shown in line [2] for this GPU-using task. In contrast, there was no observed tardiness when the SRM is in use in this particular experimental case.

blocking time GPU-using tasks experience while waiting for a GPU under the O-KGLP is O(m/k), where m is the number of CPUs and k is the number of GPUs. This is an especially interesting result since even informed approaches may lead to less efficient O($m - k$) protocols or O(m/k) protocols that block non-GPU-using tasks.

GPU Interrupt Handling. An interrupt is an asynchronous hardware signal issued from a system device to a system CPU. Upon receipt of an interrupt, a CPU halts the execution of the task it is currently executing and immediately executes an interrupt handler. An interrupt handler is a segment of code responsible for taking the appropriate actions to process a given interrupt. Each device driver, such as a GPU driver, registers a set of driver-specific interrupt handlers for all of the interrupts its associated device may raise. Only after the interrupt handler has completed execution may an interrupted CPU resume the execution of the previously scheduled task.

Interrupts are difficult to manage in a real-time system. Interrupts may occur periodically, sporadically, or at entirely unpredictable moments, depending upon the application. Interrupts often cause disruptions in a real-time system since the CPU must temporarily halt the execution of the currently scheduled task. In uniprocessor and partitioned multiprocessor systems, one may be able model an interrupt source and handler as the highest-priority real-time task in a system [25] or as a blocking source [21], though the unpredictable nature of interrupts in some applications may require conservative analysis. Such approaches can also be extended to multiproces-

sor systems where real-time tasks may migrate between CPUs [15]. However, in such systems the subtle difference between an interruption and preemption creates an additional concern: an interrupted task cannot migrate to another CPU since the interrupt handler temporarily uses the interrupted task's program stack. Stack corruption would occur if a task resumed execution before the interrupt handler completed. As a result, conservative analysis must also be used when accounting for interrupts in these systems too. A real-time system, both in analysis and in practice, benefits greatly by minimizing interruption durations. Split interrupt handling is a common way of achieving this, even in non-real-time systems.

Under split interrupt handling, an interrupt handler only performs the minimum amount of processing necessary to ensure proper functioning of hardware; any additional work that may need to be carried out in response to an interrupt is deferred for later processing. This deferred work may then be scheduled in a separate thread of execution with an appropriate priority. The duration of interruption is minimized and deferred work competes by priority with other tasks for CPU time. This, in essence, describes proper interrupt handling in a real-time system. However, achieving this in practice is actually more complicated.

High-performance GPGPU drivers are commonly only available for general-purpose operating systems, specifically, Microsoft Windows, Mac OS X, and Linux. This limits our ability to implement a real-time operating system that uses GPUs since Windows and Mac OS X are both closed source and not real-time operating systems; thus, they cannot be modified to support robust real-time features such as the CM or SRM. This leaves only Linux. Using the Linux-based LITMUS^{RT}, we can begin to support real-time GPUs. However, being based upon Linux, LITMUS^{RT} also uses Linux's general-purpose method of interrupt handling. Being out of scope for this paper, we do not delve fully into the real-time limitations of interrupt handling in Linux. However, it is sufficient to say that Linux's interrupt handling method can lead to long priority inversions as interrupt handlers may execute for a long duration before returning to normal execution. We recently extended LITMUS^{RT} to better support better real-time handling of GPU interrupts [19].

The real-time handling of GPU interrupts in LITMUS^{RT} required two major features. First, it was necessary for LITMUS^{RT} to identify interrupts raised by specific GPUs. An in-depth study of the GPU driver, in addition to a process of trial and error, was needed to understand the format of data that could be intercepted by LITMUS^{RT}. With this information, the bottom-halves of GPU interrupts were scheduled in separate real-time threads (the second major feature), thus reducing the duration of priority inversions. All of this was accomplished despite the fact that the GPU driver was closed-source. However, merely scheduling interrupt handlers individually is not sufficient if a sporadic task model is to be maintained. Since GPU-using tasks may use GPUs in an asynchronous fashion, as is quite common in high-performance code, it is possible for a GPU interrupt to be raised while the task using the GPU is currently executing. If the interrupt handler and GPU-using task execute simultaneously, then the system deviates from the sporadic task model, which assumes that a particular task executes only on one CPU at a time. Measures had to be taken to ensure that the bottom-half interrupt handler and the GPU-using task that triggered it were never scheduled simultaneously.

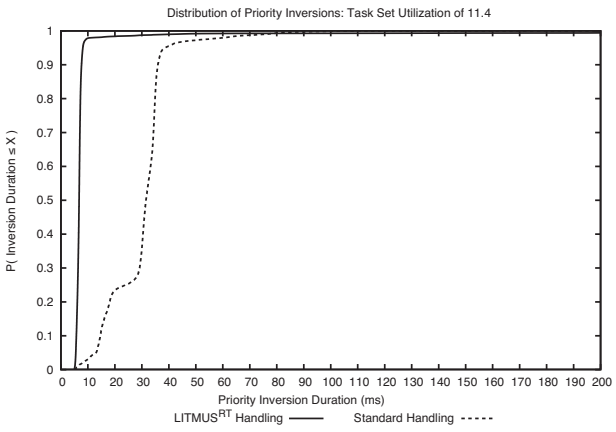

Figure 4: The cumulative distribution of priority inversion durations for this particular task set show that the typical inversion is significantly reduced when LITMUS^{RT} is used to schedule GPU interrupt handlers in comparison to standard Linux interrupt handling.

In experimentation with our extended version of LITMUS^{RT}, we found that both the *number* and *duration* of priority inversions were significantly reduced in nearly every experimental case. A cumulative distribution function of priority inversion durations for a particular task set after two minutes of execution is shown in Fig. 4. As seen, a typical priority inversion is much shorter when LITMUS^{RT} is used to schedule GPU interrupts in comparison to standard Linux interrupt handling. For example, 90% of inversions under LITMUS^{RT} interrupt handling are shorter than $9\mu s$, whereas the 90th percentile exceeds $30\mu s$ under standard Linux interrupt handling.

5 Future Work

To date, our research has primarily been on soft real-time systems. There is still a great deal of work to be done to fully explore this domain. To begin with, we have only explored a subset of systems with global and clustered CPU deadline-based schedulers. The choice of CPU scheduler constrains the choice of locking protocols since not all protocols may be used with all schedulers. It is not clear what combination of scheduler and locking protocol is best able to guarantee soft real-time constraints when an SRM-based approach to GPU management is used. There are many possible scheduler/locking protocol combinations, and we plan to study each in-depth in hopes of conclusively identifying the best configuration.

There are several other problems within the soft real-time domain that have yet to be addressed. One major issue is to discover the best method to allow simultaneous data transmission to/from a GPU while the GPU is busy executing program code. This technique is used very frequently in high-performance GPGPU programming since it can be used to mask communication latencies. However, both the CM and SRM prevent such techniques since GPUs are used exclusively by one task at a time.

Another area to explore is simultaneous GPU program execution. The latest GPU technology allows the simultaneous execution of GPU program code in some limited cases. We have yet to see if this feature may be leveraged to help alleviate the very negative long blocking durations caused by non-preemptive GPU execution.

It is our hope that any solutions developed for the soft real-time domain can also be applied to the hard real-time domain. However, it appears infeasible to implement a hard real-time GPU system using current technology. Firstly, a hard real-time system may require preemption of GPU execution, which is currently not possible. Secondly, it is likely that the I/O interface between the CPU and GPU may introduce latencies too great for hard real-time requirements. Integrated on-chip CPU/GPU technology may need more time to mature to the point where both processors can quickly and seamlessly share the same pool of system memory. Finally, and perhaps the greatest obstacle, GPU drivers should be redesigned with hard real-time constraints in mind. It appears that either the GPU driver must integrate tightly with the operating system, or more likely, it must at least expose a sufficient interface to the operating system that may be used to affect driver behavior. However, the design of such an interface is non-trivial and could require a commitment of GPU manufactures to maintain a degree of stable driver behaviors in addition to agreement from operating system developers. GPU manufactures would be very careful to avoid any behaviors that could limit performance in their greater throughput-oriented markets, although a separate real-time driver could also be implemented. In any case, it is necessary for the real-time and cyber-physical systems communities to show that there is both a market and body of research to build upon for such investments.

6 Conclusion

The use of GPUs as a high-performance co-processor is becoming a mature technology and may be applied in a number of diverse real-time applications. However, this throughput-oriented technology presents challenges to its integration into real-time systems. In this paper, we have presented a summary of our research on this topic to date. We have found that GPU resources in soft real-time systems can be managed through specialized CPU schedulers, or alternatively through efficient locking protocols. We have also implemented methods for real-time handling of interrupts raised by GPU devices, which greatly reduces the duration of priority inversions that interrupts may cause. There is still a great deal of research to be done for GPUs in the soft real-time domain. It is our hope that the development of real-time GPU techniques will help sway GPU manufactures to incorporate hard real-time features into their products.

Acknowledgement: We would like to thank Pinar Muyan-Ozcekil for furthering our understanding of road sign recognition in automotive applications, as well as putting forth the idea of exploiting concurrent GPU program execution as a method to address the limitations of non-preemptive execution.

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] Bringing high-end graphics to handheld devices. Available from: http://www.nvidia.com/object/IO_90715.html.

[4] CUDA Zone. Available from: http://www.nvidia.com/object/cuda_home_new.html.

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

[6] 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.

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

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

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

[10] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. In *SIGGRAPH '03*, pages 917–924, 2003.

[11] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In submission.

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

[13] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS^{RT}: A status report. In *9th Real-Time Linux Workshop*, pages 107–123, 2007.

[14] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *14th Real-Time and Embedded Technology and Applications Symposium*, pages 342–353, April 2008.

[15] B. Brandenburg, H. Leontyev, and J. Anderson. An overview of interrupt accounting techniques for multiprocessor real-time systems. *Journal of Systems Architecture*, 57(6):638–654, 2010.

[16] Paul R. Dixon, Tasuku Oonishi, and Sadaoki Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Computer Speech and Language*, 23(4):510–526, 2009.

[17] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. In *18th International Conference on Real-Time and Network Systems*, pages 197–206, 2010.

[18] G. Elliott and J. Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. Technical Report TR11-003, Department of Computer Science, University of North Carolina at Chapel Hill, 2011.

[19] G. Elliott, C.-H. Sun, and J. Anderson. Real-time handling of GPU interrupts in LITMUS^{RT}. Technical Report TR11-002, Department of Computer Science, University of North Carolina at Chapel Hill, 2011.

[20] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *SIGGRAPH '03*, pages 92–101, 2003.

[21] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *14th Real-Time Systems Symposium*, pages 212–221, 1993.

[22] V. Kindratenko and P. Trancoso. Trends in high-performance computing. *Computing in Science Engineering*, 13(3):92–95, 2011.

[23] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03*, pages 908–916, 2003.

[24] M. Lalonde, D. Byrns, L. Gagnon, N. Teasdale, and D. Laurendeau. Real-time eye blink detection with GPU-based SIFT tracking. In *Canadian Conference on Computer and Robot Vision*, pages 481–487, 2007.

[25] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[26] P. E. McKenney. "Real Time" vs. "Real Fast": How to choose?, 2009.

[27] 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.

[28] 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.

[29] S. Thrun. GPU technology conference keynote, day 3, 2010. Available from: http://livesmooth.istreamplanet.com/nvidia100923/.