Real Time Executions on GPUs

By

Kiriti Nagesh Gowda

PhD Candidate

Department of Electrical & Computer Engineering

in the Graduate School

Southern Illinois University Carbondale

December 2014

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr. Harini Ramaprasad for the continuous support of my Ph.D. study and research, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me in all the time of research and writing of this proposal document. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Besides my advisor, I would like to thank the rest of my committee. I would like to thank Dr. Shaikh Ahmed for accepting to be my committee chair. My heartfelt gratitude goes to the committee members Dr. Ning Weng, Dr. Ada Chen, and Dr. Mengxia Zhu.

Table of Contents

Chapter 1: INTRODUCTION

This chapter provides a brief introduction to the real time system concepts, a brief review of graphic processing units and the motivation for this proposal. Assumptions on the architectural model, and research plan are also provided.

Real time Systems are systems with both logical and temporal constraints. These systems have its footprint in various fields such as avionics, communication, medical, and defense industry. Real time systems must ensure robustness, safety, and determinism in real time; failing to meet any one of these requirements may result in undesirable consequences.

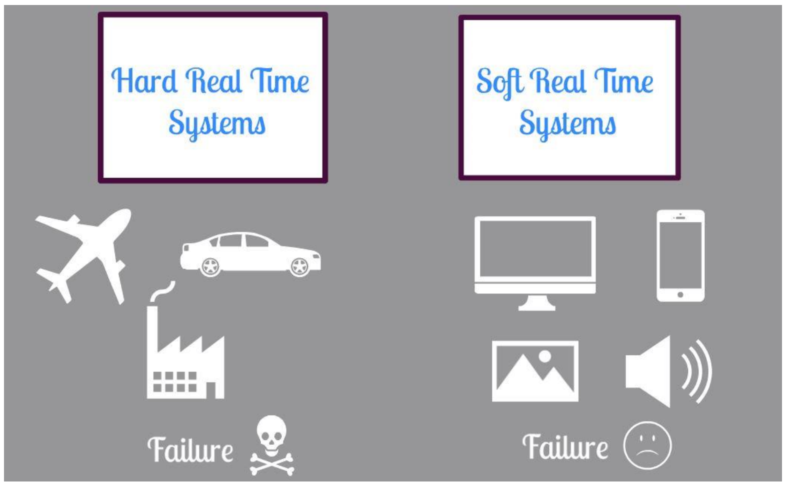


Figure 1: Hard & Soft Real Time Systems

Systems are categorized into hard real time systems, non-real time systems and soft real time systems. Hard real time systems are systems with strict temporal constraints. These systems run hard real time tasks. Soft real time systems are systems composed of soft real time tasks with less strict temporal constraints. A deadline miss in such systems will not result in a system crash, but affects system performance. Non-real time systems are systems with no concept of deadline. These systems run tasks as and when possible.

Figure 2: NVIDIA GPU

Whereas, a graphics processing units (GPU) are designed to operate in a single instruction multiple data (SIMD) fashion. The key application of a GPU is to serve as a graphic accelerator. These graphic acceleration tasks process the same operations independently on large volumes of data. This fashion of SIMD operation is suitable for the acceleration of compute intensive applications.

The architectural aspects of NVIDIA GPU’s differ slightly for GPU’s without Fermi and GPU’s with Fermi both of which are briefly described below

Hardware Model

GPU’s without Fermi has up to 16 multiprocessors per chip and 8 processors (ALU’s) per multiprocessor. During any clock cycle all the processors of a multiprocessor execute the same instruction, but may operate on different data. There is no mechanism to communicate between the different multiprocessor. No synchronization exists to enable communication between multiprocessors.

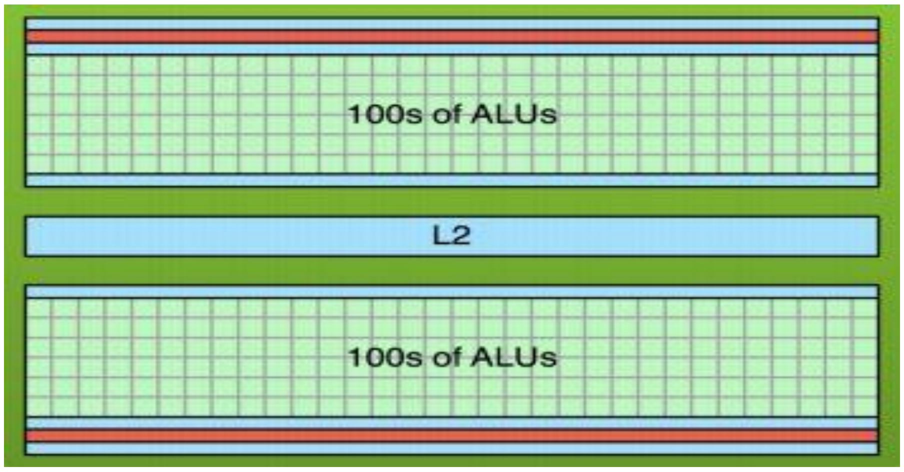


Figure 3: GPU Model

GPU's with Fermi have around 16 Streaming Multiprocessors (SM's) and 32 CUDA enabled cores per SM, each capable of initiating one single precision floating point or integer operation per clock, also has 16 load/store units, four special function units, a warp scheduler and dispatch unit to handle the threads better.

Memory Model

Each multiprocessor of non-Fermi architecture has on-chip memory of the following types:

- One set of local 32 bit registers per processor
- A parallel data cache or shared memory that is shared by all the processors of the multiprocessor. The size of this shared memory per multiprocessor is 16KB and it is organized into 16 banks

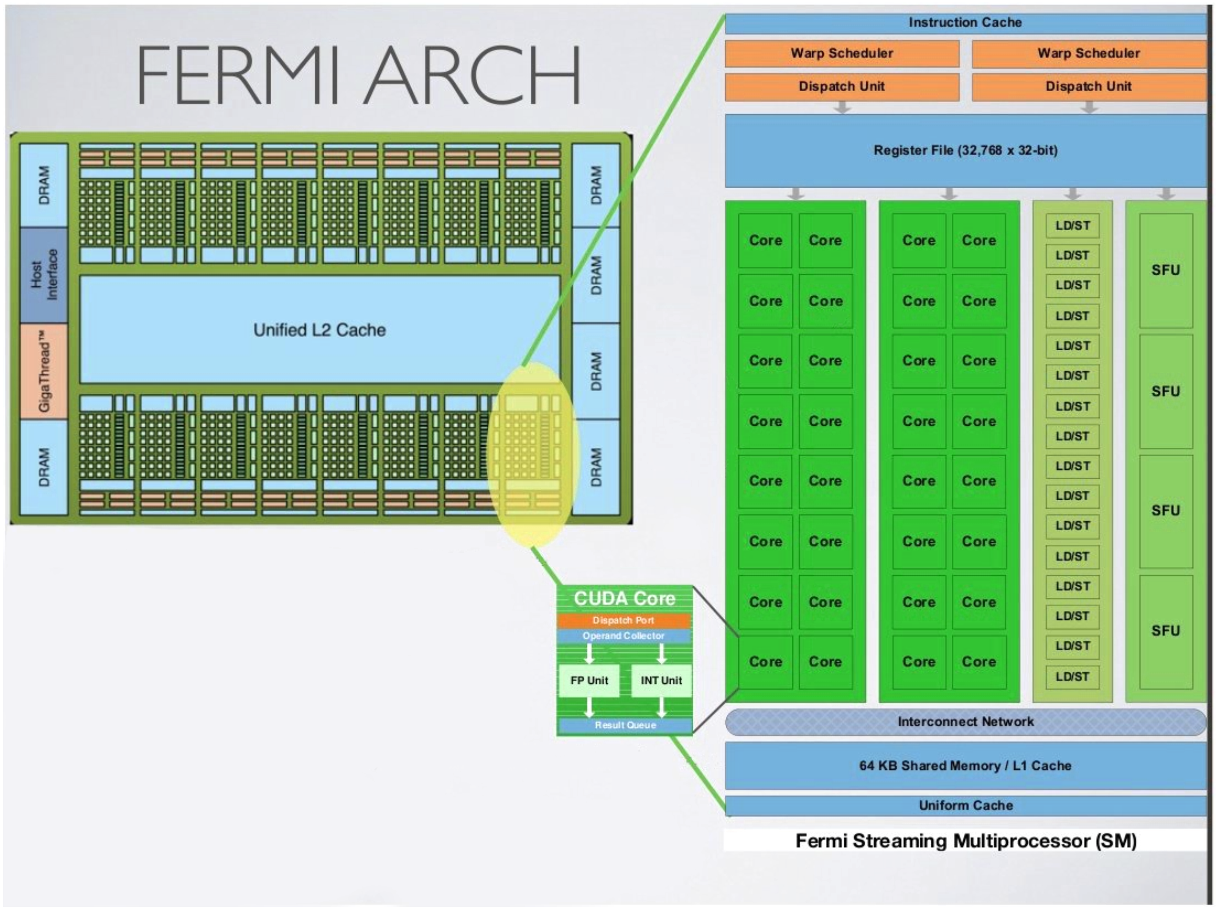


Figure 4: GPU with Fermi Architecture

- A read only constant cache that is shared by all the processors in a multiprocessor, which speeds up reads from the constant memory space
- A read only texture cache that is shared by all the processors in a multiprocessor, which speeds up reads from the texture space. The local and global memory spaces are implemented as read/write regions of the device memory and are not cached.
- Local memory of a processor is used to storing the data structures declared in the instruction executing on that processor; it is in the order of KB's.
- Global memory is read/write memory that is not cached and can take up to 400-600 clock cycles to be read or written, the global memory is in the order of MB's close to a GB.

Each Streaming Multiprocessors in Fermi architecture have the following types:

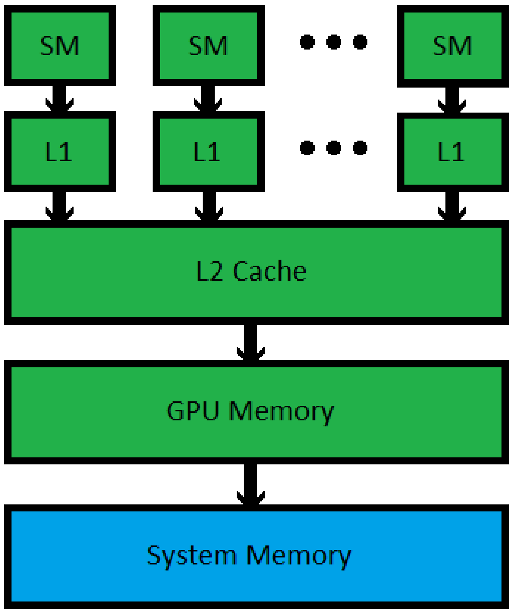


Figure 5: GPU Memory Hierarchy

- 64KB block of high speed on chip memory that can be used either to cache data for individual threads or to share data among several threads and an interface to level 2 cache shared among all 16 SM's.
- The L2 cache is 768KB which is not present in the previous architecture. L2 cache is shared globally and maintained in a coherent manner for all SM's Fermi communicates with the host system via a 16 lane PCI express V2 interface that moves data to and fro at the peak rates of 8GB/sec. It can be addressed and cache directly items stored in the host main memory.
- This has a 64 bit address bit and a global memory on the board up to 6GB.

Programming Model

- CUDA – Compute unified device architecture which is used to interfacing with the GPU devices.

- A GPU devices operates as a coprocessor to the main CPU or host. Data parallel, compute intensive portions of the application running on the host can be off loaded onto the GPU devices.

- A thread block is a batch of threads that can cooperate together by effectively sharing data thought some fast shared memory and synchronize their execution to coordinate memory access.

- User can specify synchronization points in the kernel, where threads in a block are suspended until they reach the synchronization point.

- Threads are grouped into warps, which are further grouped in blocks; threads have individual identity numbers threads ID. All warps composing a block are guaranteed to run on the same multiprocessor and can take advantage of shared memory and local synchronization.
- Each warp consists of same number of threads, called warp size and is executed in a SIMD fashion. The maximum number of threads grouped per thread block is 1024.

- A thread block can be executed by a single multiprocessor at most of 1024 threads can be active in a single multiprocessor at any given time.

Graphics Processing Units (GPUs) are computational powerhouses that were originally designed for accelerating graphics applications. However, in recent years, there has been a tremendous increase in support for general purpose computing on GPUs (GPGPU). GPU based architectures provide unprecedented magnitudes of computation at a fraction of the power used by traditional CPU based architectures. As real-time systems integrate more and more functionality, GPU based architectures are very attractive for their deployment. However, in a real-time system, predictability and meeting temporal requirements are much more important than raw performance. While some real time jobs may benefit from the performance that all cores of the GPU can provide, most jobs may require only a subset of cores in order to successfully meet their temporal requirements. In this document, we propose to study concurrent scheduling of real-time jobs on a GPU based platform and its memory management.

Chapter 2: MOTIVATION

Real-time systems not only have to satisfy logical correctness requirements like any other computing system, but also have to adhere to temporal correctness requirements, typically represented as deadlines for every job within the system. As computational demands of such systems continue to increase in the wake of the ubiquitous presence of real-time systems in today's world, traditional single-core architectures are no longer a viable option for their deployment. As a result, there is a significant body of research that studies the challenges involved in real-time execution on multi-core architectures. However, most of this work focuses on CPU based architectures.

Smart planes for safe air travel
Earthquake-proof building
Energy-efficient power grid
Structurally sound bridges
Smart appliances in energy-efficient homes
Safe highways
Smart car
Intelligent sensors for cleaner water resources
Robotic at work and play

Figure 6: Real Time Systems in our daily life

Recently, researchers have started to explore the use of GPU based architectures in real-time systems. There is strong motivation for enabling real-time execution on GPUs. As noted by Elliott and Anderson [1], there are two fundamental aspects that make GPUs an attractive option for real-time systems.

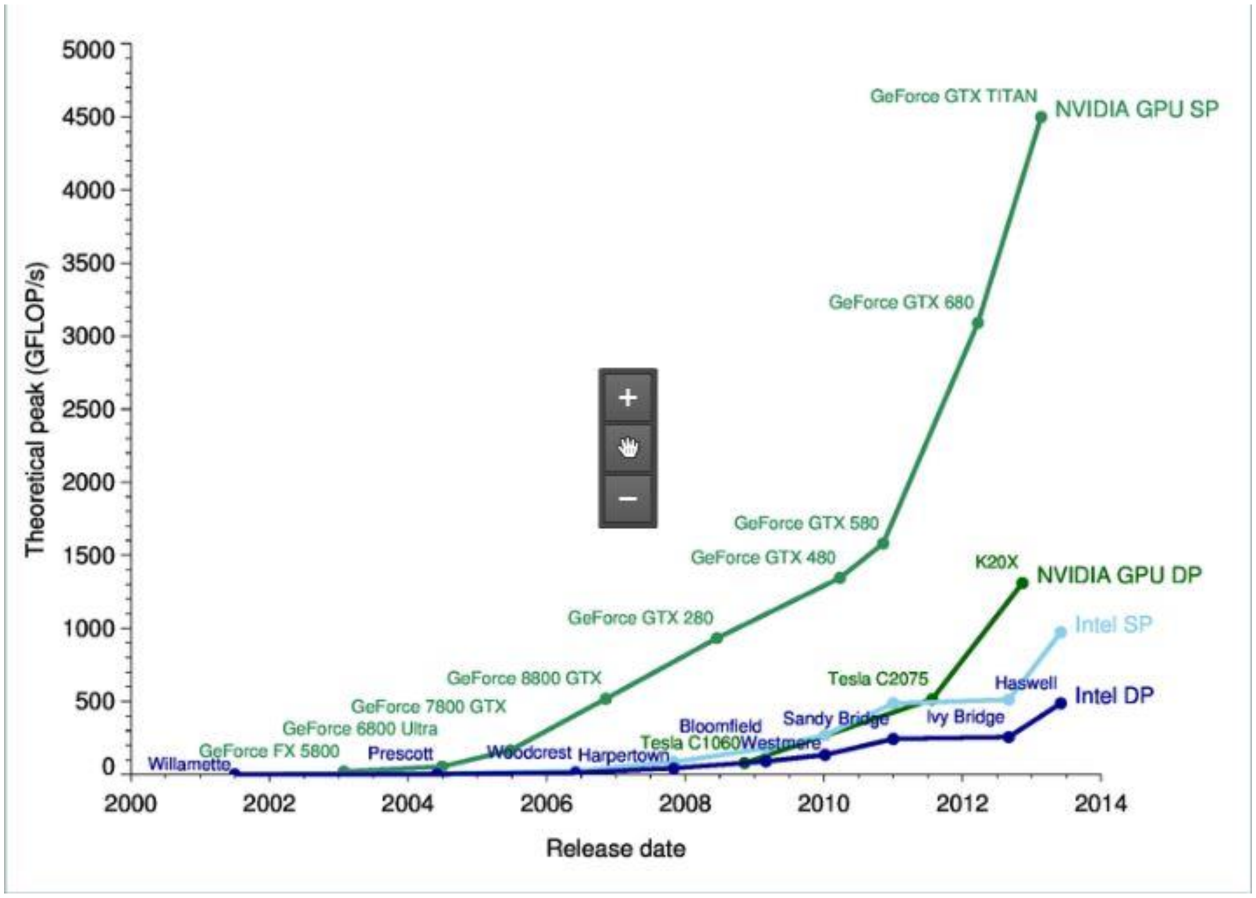
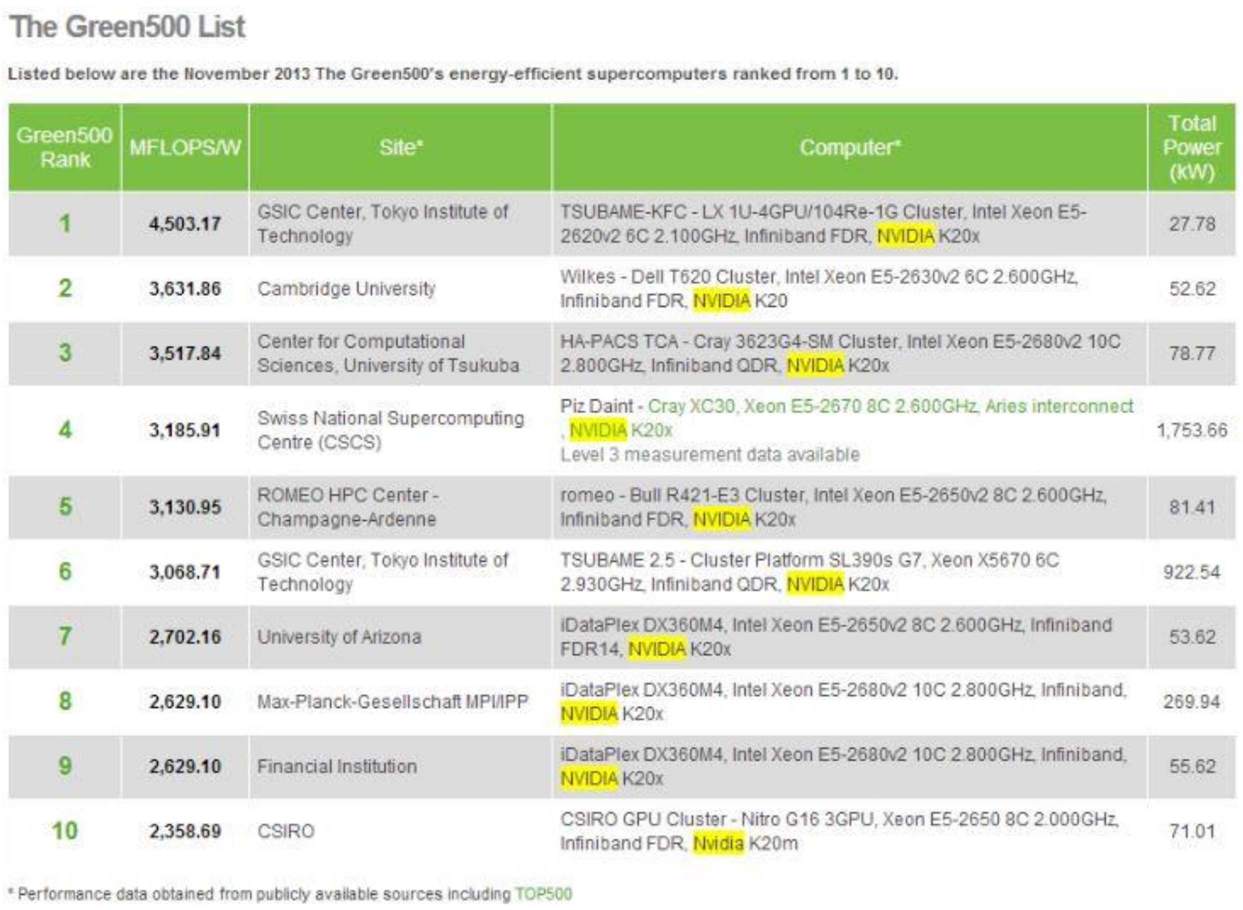


Figure 7: GPU Peak Performance Chart

First, GPUs execute at higher frequencies, thereby accelerating the execution of jobs allocated to it. This could improve system responsiveness. Second, the power needed for a GPU to carry out an operation is much lesser than that needed by traditional CPUs, making it ideal for use in real-time embedded systems. Having said that, execution on GPU architectures has fundamental differences compared to that on CPU based multi-core architectures. First, due to significant hardware and firmware challenges in enabling preemptive execution, GPU execution is assumed to be non-preemptive. Second, GPUs do not provide the degree of controllability of cores that is typically available on CPU based multi-core platforms. In early versions of GPU platforms, only one instruction stream or function represented by a kernel could execute on a GPU at any given time, regardless of GPU utilization. As such, most existing work on enabling real-time execution on GPUs treat the GPU as a black box and focus on developing techniques to determine the order (schedule) for dispatching kernels.



The Green500 List

Listed below are the November 2013 The Green500's energy-efficient supercomputers ranked from 1 to 10.

Green500 Rank	MFLOPS/W	Site*	Computer*	Total Power (kW)
1	4,503.17	GSIC Center, Tokyo Institute of Technology	TSUBAME-KFC - LX 1U-4GPU/104Re-1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x	27.78
2	3,631.86	Cambridge University	Wilkes - Dell T620 Cluster, Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20	52.62
3	3,517.84	Center for Computational Sciences, University of Tsukuba	HA-PACS TCA - Cray 3623G4-SM Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, NVIDIA K20x	78.77
4	3,185.91	Swiss National Supercomputing Centre (CSCS)	Piz Daint - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Level 3 measurement data available	1,753.66
5	3,130.95	ROMEO HPC Center - Champagne-Ardenne	romeo - Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20x	81.41
6	3,068.71	GSIC Center, Tokyo Institute of Technology	TSUBAME 2.5 - Cluster Platform SL390s G7, Xeon X5670 6C 2.930GHz, Infiniband QDR, NVIDIA K20x	922.54
7	2,702.16	University of Arizona	iDataPlex DX360M4, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR14, NVIDIA K20x	53.62
8	2,629.10	Max-Planck-Gesellschaft MPI/IPP	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	269.94
9	2,629.10	Financial Institution	iDataPlex DX360M4, Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, NVIDIA K20x	55.62
10	2,358.69	CSIRO	CSIRO GPU Cluster - Nitro G16 3GPU, Xeon E5-2650 8C 2.000GHz, Infiniband FDR, Nvidia K20m	71.01

* Performance data obtained from publicly available sources including TOP500

Figure 8: GPUs in top10 green supercomputers

More recent GPU platforms such as the NVIDIA Fermi architecture [2] do support concurrent execution of kernels1. In practice, real-time functions may not benefit from the performance that using the entire GPU (i.e., all its cores) can provide. One reason may be simply because it does not require that amount of computational power. A second reason may be that the function is memory bound and cannot effectively utilize all computational elements due to memory transfer delays.

On the other hand, concurrent execution of multiple functions could be tremendously useful in maintaining timeliness of applications. This motivates us to explore the development of policies for concurrent scheduling of kernels. In the current work, we target soft-real-time job execution.

We have done an extensive study on the GPGPU memory hierarchy and have benchmarked various programs, to check the response times, and the usability of the layered structure of the GPUs in a real time application, since we began exploring the capabilities of GPUs on the available hardware.

Chapter 3: EXISTING RESEARCH

Elliott and Anderson present potential real-time applications that can benefit from GPU architectures and discuss the limitations and constraints of current GPU architectures [1]. Several researchers propose policies for scheduling real time jobs on the GPU [4], [5], [6] using a priority-based approach.

In other recent work, researchers target a multi-GPU model, where jobs are dispatched on to an array of available GPUs [7], [6]. Research has been conducted in decoding the GPU driver or managing the GPU as a resource [8], [9]. However, this entire body of work assumes that only one kernel may execute on a GPU at a given time. Recent work has explored concurrent execution of multiple kernels on a single GPU, using hardware (e.g., NVIDIA Fermi architecture) that supports concurrent execution.

Wang et al. discuss considerations involved in such execution and present an experimental evaluation of the performance impact of such execution [3]. This work does not specifically target real-time execution. Junsung et al. present frameworks for supporting adaptive GPU resource management by allowing tasks to use a variable number of cores on the GPU based on their needs and core availability [10]. The authors propose an explicit management mechanism that requires significant programmer involvement and an implicit management mechanism with less programmer involvement.

Currently, the Fermi architecture only allows concurrent execution of kernels that share a common GPU context. However, the fundamental ideas of our work are not affected by or dependent on this limitation. Moreover, there exist software solutions such as context funneling [3] to allow kernels from multiple GPU contexts to execute concurrently.

Chapter 4: RESEARCH PLAN

Our research plan aims to develop a dynamic schedule management framework for soft-real-time jobs on GPU based architectures. Cores on a platform such as the NVIDIA Fermi architecture [2] are organized into clusters, termed streaming multiprocessors (SMs). Cores within each SM share resources (register file, control units, L1 cache, etc.) and execute a common kernel. Our goal is to divide a real-time job into kernels and schedule kernels on the GPU, treating each SM as an indivisible unit. We propose to achieve this goal with minimal programmer involvement.

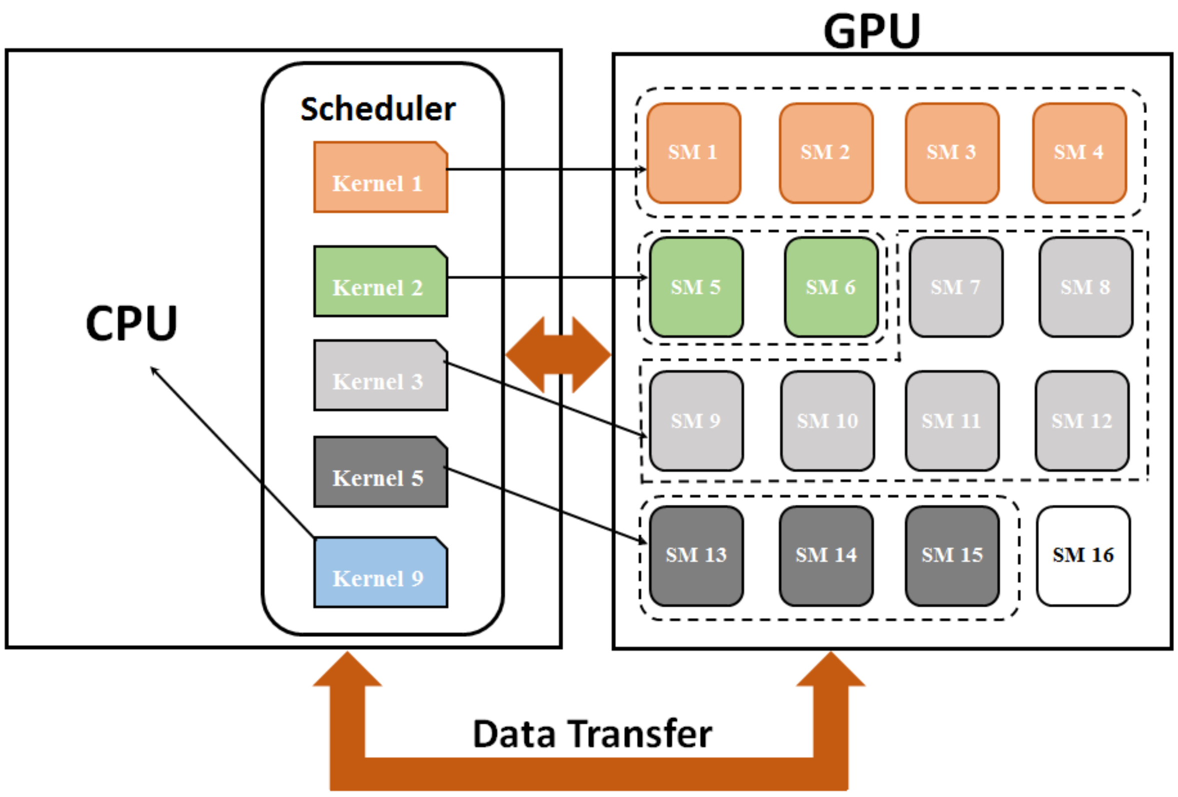


Figure 9: Concurrent Kernel Execution on GPU

In general, a kernel consists of a set of thread blocks. When a kernel is dispatched to the GPU, a work distribution engine in the GPU schedules thread blocks on available SMs. The fundamental idea behind our technique is to divide a kernel into a set of controlled blocks of threads such that the number of threads per block is close to the total number of threads that a single SM can handle concurrently. For example, in the NVIDIA Fermi architecture [2], every SM is capable of supporting 1536 threads, among which 1024 threads can execute concurrently with minimal context switch costs. So, for this architecture, block sizes of 1024 threads are found to be a suitable choice. In this way, no more that one block may reside on a SM at a given time and hence, the number of blocks is a measure of the number of SMs that a kernel will occupy. We have conducted preliminary experimentation on the NVIDIA Fermi platform to verify and evaluate the feasibility of this basic approach towards SM scheduling.

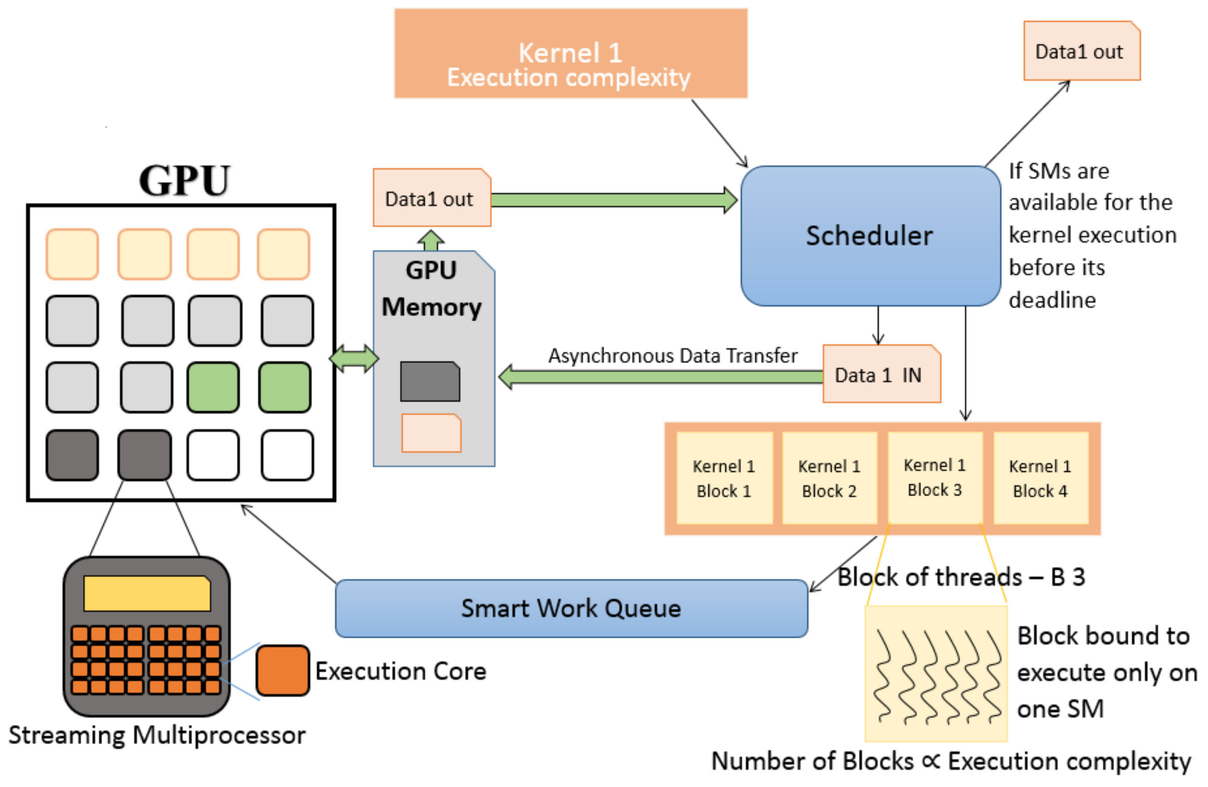


Figure 10: Kernel Scheduling

Thus far, we have found the results to be encouraging. We propose to exploit this basic idea to perform coarse grained scheduling of jobs on SMs. Our work lays emphasis on minimal programmer involvement. To this end, we are developing a dynamic schedule management framework (somewhat similar to the implicit adjustment approach proposed in [10]) that is responsible for 1) keeping track of current and expected SM availability; 2) determining which kernel(s) to dispatch to the GPU at a given time; and 3) determining how many SMs to assign for a given kernel. These decisions will be made based on observed and predicted system state and on job characteristics (expected execution times, deadlines, etc.). Figure 9 depicts our framework. As seen in the figure, our scheduling framework resides on a CPU core and dispatches kernels to the GPU.

Once we successfully implement our framework we want to extend the scheduler to include period jobs, these periodic jobs, whose arrival times are known beforehand can be smartly scheduled on to our GPU allowing the scheduler enough slack to schedule aperiodic or sporadic jobs in between. We are also planning to extend our framework to seamlessly place our data elements in the hierarchal structure of GPU memory units to make the executions more predictable.

Chapter 5: CONCLUSIONS

Graphics Processing Units (GPUs) are capable of providing tremendous computational power under reasonable power/energy budgets. Recent GPU based platforms allow concurrent execution of multiple functions on the GPU. This provides an avenue to explore real-time scheduling on such platforms, trading off raw performance of individual jobs for improved responsiveness and schedulability of job sets. In the research plan described in this paper, we envision the development of a dynamic schedule management framework for soft-real-time job execution on GPU based platforms. In the first phase of the work, we are targeting sets of independent soft-real-time jobs. As part of future work, we plan to extend our framework to support recurring (periodic) soft-real-time tasks and smart memory management.

We propose to use the hardware to explore fine-grained allocation and scheduling options within the GPU. Specifically, we propose to explore the possibility of allocating a subset of streaming multiprocessors to a given task so that multiple sets of streaming multiprocessors may be used for executing multiple tasks in parallel. In this context, we propose to study the trade-offs involved in performance of individual tasks, ability of multiple tasks to better meet their individual time deadlines and the level of controllability and programmability required to achieve this.

With the current results we have on the memory response time, we are trying to explore ways to optimize cache performance, by using the shared or the scratch pad memory available in the streaming multiprocessors, to do fine-grained allocation of data, and smart locking protocols to reduce the cache misses, which would help the tasks to be executed in the allocated time. This could help our efforts in scheduling soft real-time systems task within the GPU.

REFERENCES

[1] G. A. Elliott and J. H. Anderson, “Real-World Constraints of GPUs in Real-Time Systems,” in International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011, pp. 48– 54.

[2] P. N. Glaskowsky, “NVIDIA’s Fermi: the first complete GPU computing architecture,” White paper, 2009.

[3] L. Wang, M. Huang, and T. El-Ghazawi, “Exploiting concurrent kernel execution on graphic processing units,” in International Conference on High Performance Computing and Simulation (HPCS), 2011, pp. 24–32.

[4] G. A. Elliott and J. H. Anderson, “Globally scheduled real-time multiprocessor systems with GPUs,” Real-Time Systems, vol. 48, no. 1, pp. 34–74, 2012.

[5] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, “TimeGraph: GPU scheduling for real-time multi-tasking environments,” in USENIX Annual Technical Conference (USENIX ATC), 2011, p. 17.

[6] G. A. Elliott, B. C. Ward, and J. H. Anderson, “GPUSync: A framework for real-time GPU management,” in Real-Time Systems Symposium (RTSS), 2013, pp. 33–44.

[7] G. A. Elliott and J. H. Anderson, “An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems,” Real-Time Systems, vol. 49, no. 2, pp. 140–170, 2013.

[8] G. Elliott and J. Anderson, “Robust real-time multiprocessor interrupt handling motivated by GPUs,” in Euromicro Conference on Real-Time Systems (ECRTS), 2012, pp. 267–276.

[9] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, “RGEM: A responsive GPGPU execution model for runtime engines,” in Real-Time Systems Symposium (RTSS), 2011, pp. 57–66.

[10] J. Kim, R. R. Rajkumar, and S. Kato, “Towards adaptive GPU resource management for embedded real-time systems,” ACM SIGBED Review, vol. 10, no. 1, pp. 14–17, 2013.