Towards Adaptive GPU Resource Management for Embedded Real-Time Systems

Junsung Kim and Ragunathan (Raj) Rajkumar

Department of Electrical and Computer Engineering
Carnegie Mellon University

Shinpei Kato

Department of Computer Science
University of California, Santa Cruz

Abstract

In this paper, we present two conceptual frameworks for GPU applications to adjust their task execution times based on total workload. These frameworks enable smart GPU resource management when many applications share GPU resources while the workloads of those applications vary. Application developers can explicitly adjust the number of GPU cores depending on their needs. An implicit adjustment will be supported by a run-time framwork, which dynamically allocates the number of cores to tasks based on the total workload. The runtime support of the proposed system can be realized using functions which measure the execution times of the tasks on GPU and change the number of GPU cores. We motivate the necessity of this framework in the context of self-driving technologies, and we believe that our frameworks for GPU programming are useful contributions given the increasing emphasis on parallel heterogeneous computing.

1 Introduction

Graphics processing units (GPUs) are becoming more and more commonplace in many application domains widely ranging from high-performance computing to embedded mobile computing. For example, three of the top five supercomputers on the TOP500 list [12], announced as of March 2012, use GPUs to accelerate computations, while recent tablets, such as ASUS Eee Pad Transformer Prime, also leverage embedded GPUs, like Tegra 3 [9], to enhance performance under power constraints. This trend is expected to continue.

One notable application domain of GPUs is automotive engineering. Modern automobiles employ several tens of processing units. Further advances in safe-driving features, such as adaptive cruise control, stop-and-go cruise control, lane keeping, and assisted lane change, would require even larger computing capabilities. For vehicles to become fully or semi-autonomous, a multitude of computer vision, sensor fusion, signal processing, and graphics sub-systems must operate and communicate in real-time. Given their highly data-parallel and compute-intensive workloads, parallel computing is a useful solution. As technology stands today, the GPU is the most well-suited platform. In fact, NVIDIA GPUs will be used for infotainment systems platforms in future product lines of BMW vehicles [10].

Automatic safety features require smart planning and intelligent processing of data obtained from many sensors equiped in the vehicle, including LIDAR (LIght Detection And Ranging), radar, camera, and ultrasonic sensors. One common characteristic in these types of processing is that GPUs can accelerate their processing speeds significantly. For example, autonomous driving should ideally follow the best path among many potential paths, whose calculations can happen in parallel. Calculating as many potential paths to follow as possible will yield better quality of driving. As a matter of fact, CMU's autonomous vehicle team showed that their motion planning algorithm was sped up by 40 times [6], using an NVIDIA GTX 260 GPU that integrates 192 compute cores on a chip.

In addition to motion planning, a perception algorithm as well as sensor data processing can benefit from the GPU. The perception system of a self-driving car should be able to detect, classify and track the obstacles around itself. Various types of sensors will generate voluminous amount of information that must be processed in order to understand the vehicle's surroundings. For example, a self-driving car at CMU manages 1536 objects from LIDAR sensors before they are fused with other types of sensor data. There has been on-going research using GPU to build a perception system [1], and their GPU implementation yielded 30,000-times-faster performance compared to the case of using only one CPU.

As described above, there has been research on applying GPU to different applications on self-driving cars, and it is clear that GPU can provide great benefits on realizing safer or self-driving car technologies. However, not much research has been done when those technologies are deployed together on self-driving cars, where the loads of

each application dynamically vary depending on the environment. The period and the computation time of the planning algorithms for autonomous driving highly depend on the vehicle speed, so the planning algorithms can be heavily loaded when the car is (say) on a highway. The load of the perception algorithms mainly depends on the number of obstacles around the car. Hence, the perception system requires more computing resources when the car is driving in an urban area. Therefore, an intelligent method of sharing many cores on GPU would be essential when we use GPUs on self-driving cars. For example, if a self-driving car has a 96-core GPU, the planning algorithm of the car can use 72 cores on the highway and use 12 cores in the urban environment. A self-driving car [13] requires tens of tasks, and the dynamic core management should be fulfilled across all tasks if those tasks utilize GPU.

In this paper, we present two conceptual frameworks for GPU applications to adjust their task execution times given current workload conditions. These frameworks enable smart GPU resource management when many applications share GPU resources while the workloads of those applications vary. These frameworks support both explicit and implicit adjustment. With support for explicit adjustment, application developers can adjust the number of GPU cores depending on their needs. A run-time framwork will dynamically allocate the number of cores to tasks based on current workloads. The runtime support of our proposed system can be realized using functions which measure the execution times of the tasks on GPU and change the number of cores.

The rest of paper is organized as follows. Section 2 describes how our proposed system is modeled. Section 3 presents the methods for adaptively managing GPU resources, and we conclude our paper in Section 4.

2 System Model

We assume real-time embedded systems that contain CPU and one or more GPUs as compute devices. An application task starts execution on the CPU, and offloads its data-parallel compute-intensive workload onto the GPU when needed. Once offloaded onto the GPU, the task becomes non-preemptive due to many reasons. In fact, it is technically possible to preempt the running task on the GPU by loading and restoring its context, but it requires additional firmware, runtime, and OS support, and the preemption cost would be non-trivial due to a very large set of GPU registers and states. We, hence, restrict our attention to a non-preemptive execution model for GPU computing. GPUs may also pose some constraints in multi-tasking. Even the NVIDIA Fermi architecture [8], one of the most popular GPU product lines, allows only one context to use GPU resources at once, though this context may spawn multiple GPU kernels (jobs) simultaneously. In other words, if task-level parallelism is required, the entire system must run in the same context. We, however, believe that this constraint will not limit the concepts we describe in this paper. The same GPU context can be used to exploit concurrent parallel job executions, to serve at least as a proof of concept. We also expect that future product lines will remove this concern.

We consider real-time applications where each task runs in a periodic or sporadic manner under deadline constraints. Such a task set may include motion planning and vision-based perception in state-of-the-art autonomous driving vehicles, where the periods often correspond to frame-rates, and the deadlines occur at the end of the period. We also presume that the computing demand of each task is highly variable. For example, the performance of planning and perception tasks is usually governed by the number of objects, the size of data, and the desired quality of output. These workloads are also very parallelizable using the GPU. The contributions of this paper are not limited to autonomous driving tasks but are also generally applicable to highly variable workloads running on the GPU.

3 Adaptive GPU Resource Management

In this section, we describe adaptivity support for GPU applications. We particularly focus on solving resource allocation problems. The goal is to support embedded real-time systems that exhibit highly variable workloads. Since GPUs integrate a large number of cores on a chip, we aim to enable the execution of highly variable workloads in a timely manner by adjusting allocated cores at runtime.

Several approaches have been studied for adaptive GPU resource management. Some work [3, 4, 5] took time-driven approach that controls timings and the duration of time allowed to access GPU resources, *i.e.*, scheduling and reservation. In these time-driven approaches, application tasks need not to be aware of what is happening in GPU resource management, because it is handled by the OS or runtime scheduler. However, they can not manage task execution times. They also limit the number of contexts that can access the GPU simultaneously to remove performance interference. Therefore, GPU resources could be wasted if a running context does not fully use compute cores.

We consider a different approach than previous work that enables GPU applications to adjust their task execution times. The number of cores used in the program is a major factor that affects the execution time. Hence, we explore how to adjust the core allocation at runtime. It is important to note that the programmer is typically responsible for allocating the number of cores (or threads mapped to cores) in GPU programming. In order to adjust the number of cores at runtime, it is essential to provide the programmer with an

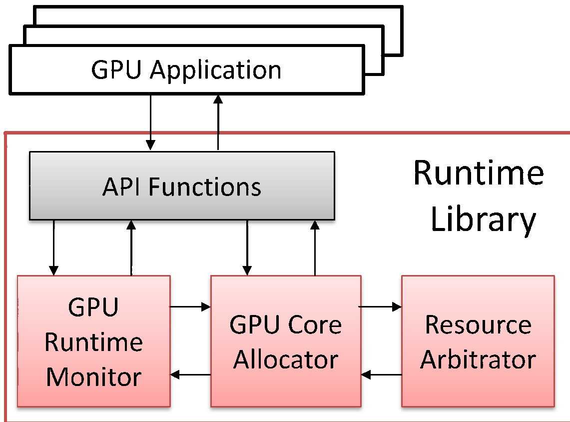

Figure 1. The proposed conceptual architecture for GPU resource management.

interface to obtain the information on the number of cores available or allocated for the program at runtime. The programmer is then responsible for making the program adaptive to the number of cores.

In the following, we present two frameworks that could be used to implement the proposed approach. We plan to implement a real system as a proof-of-concept, leveraging open-source software [2]. The proposed architecture is also illustrated in Figure 1.

3.1 Explicit Adjustment

In our explicit adjustment framework, the programmer is responsible for adjusting the number of cores to relax or tighten the computing demand. There will be no adjustment unless the programmer explicitly takes an action. A typical usage of this framework with periodic real-time tasks is as follows.

At the end of each period, the programmer calls a function provided by our framework that returns the latest task execution time. The programmer next calls either of the following two API functions. One increases the number of cores to be used by the next GPU execution to speed up the program. The other decreases it to slow down the program. This framework is usable in practice because the programmer often knows the desired task execution time to meet the frame-rate or deadline. It is also flexible in that the programmer can determine when to increase or decrease the number of cores.

A downside of this framework is that a task may misbehave and interfere with other contending application tasks, if the programmer fails to call the API functions correctly. We can cap the maximum number of cores available for an individual task to prevent it from abusing GPU resources, but the adaptivity of computing depends on the programmer, and outside system control.

3.2 Implicit Adjustment

Our second approach to adaptive resource management is an implicit adjustment framework. In this framework, the number of cores to be allocated for the program is set by the runtime system. Hence, the adaptivity of computing does not really depend on the programmer. If the program is not aware of this framework, however, it may fail to run, since the number of core allocated for the program may be different from what the program assumes.

The programmer specifies the desired task execution time as a set point before the task starts. If this set point is not specified, the runtime system tries to derive it internally as time goes by. When the task uses the GPU, the runtime system consistently updates the number of cores available for the corresponding task in the next period based on the previous execution time records. It is still the programmer's duty to check the number of available cores before offloading the computation onto the GPU.

This implicit adjustment framework is more preferable to the explicit adjustment framework, as it can enforce adaptive GPU resource management. However, it requires consensus in the programming model that the number of cores allocated for the program could be changed every time it is offloaded onto the GPU, and the programmer must be aware of it to make the program work. We claim that this is a natural trade-off between the generality of programming and needed adaptivity of service.

3.3 Runtime System Support

The runtime system provides the API for real-time GPU programmers. In order to support adaptive GPU resource management, we must provide some additional API functions.

- Our adaptive GPU resource management frameworks require a function to measure the execution time of each job running on the GPU. This function is easy to implement. Since we assume that job execution on the GPU is non-preemptive, the amount of time in run-to-completion can be accounted as job execution time. This accounting method is also known to work from previous studies [5, 11]. For the explicit adjustment framework, this function must be exposed to the programmer, while it is used internally by the implicit adjustment framework.

- We also need several functions to change the number of cores to be allocated for the program. Some existing programming languages for GPGPU, *e.g.*, CUDA [7],

provide the API to allow the programmer to specify the shape of the grid structure and the number of threads mapped to compute cores. We can use this API as it is, or provide a corresponding API if the underlying programming language does not support it.

In addition to these API functions, the runtime system must be able to detect when the program is offloaded onto the GPU and when it is completed on the GPU. Since the programmer calls a specific API function to launch the GPU program in most GPU programming models, it is very easy to record the start time of GPU execution. The detection of the completion time of GPU execution, on the other hand, is not straightforward. We would need to use an interrupt to notify the runtime system of the completion of GPU execution. Polling on a particular register is an alternative, but it would not be suitable for latency-sensitive real-time systems, as previous work demonstrated [5].

Finally, runtime system support must be integrated with the API so that the programmer can make use of our frameworks under a single unified programming model. We plan to extend our CUDA runtime library developed in previous work [2] to support our adaptive frameworks. While this is our planned prototype implementation, and our frameworks can also be integrated with other programming models beyond CUDA.

4 Summary

In this paper, we have discussed adaptivity requirements in embedded real-time systems with GPUs, and presented two frameworks for adaptive GPU resource management. We conjecture that the generality of programming may need to be compromised to achieve adaptivity of resource allocation on the GPU. Nonetheless, adaptive resource management is a key solution in optimizing performance under resource-constrained environments. We believe that our frameworks for GPU programming are useful contributions in this line of work, given the increasing emphasis in highly parallel heterogeneous computing.

References

[1] J. Ferreira, J. Lobo, and J. Dias. Bayesian real-time perception algorithms on gpu. *Journal of Real-Time Image Processing*, 6:171–186, 2011. 10.1007/s11554-010-0156-7.

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

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

[4] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 57–66, 2011.

[5] 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.

[6] M. McNaughton, C. Urmson, J.M. Dolan, and Jin-Woo Lee. Motion planning for autonomous driving with a conformal spatiotemporal lattice. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4889 –4895, may 2011.

[7] NVIDIA. CUDA C Programming Guide. `http://developer.nvidia.com/nvidia-gpu-computing-documentation`.

[8] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi (Whitepaper). `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[9] NVIDIA. Tegra 2 and Tegra 3 super chip processors. `http://www.nvidia.com/object/tegra-superchip.html` seen on March 7th, 2012.

[10] NVIDIA Press. NVIDIA GPUs to Be the Infotainment Centerpiece Across BMW's Next-Generation of Cars. `http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&version=live&prid=704317&releasejsp=release_157&xhtml=true` seen on March 7th, 2012.

[11] C. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating system abstractions to manage GPUs as compute devices. In *Proc. of the ACM Symposium on Operating Systems Principles*, 2011.

[12] Top500 Supercomputing Sites. `http://www.top500.org/`.

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