Distinguished Dissertations

Markus Steinberger*

An overview of dynamic resource scheduling on graphics processors

Abstract: In this paper, we present a series of scheduling approaches targeted for massively parallel architectures, which in combination allow a wider range of algorithms to be executed on modern graphics processors. At first, we describe a new processing model which enables the efficient execution of dynamic, irregular workloads. Then, we present the currently fastest queuing algorithm for graphics processors, the most efficient dynamic memory allocator for massively parallel architectures, and the only autonomous scheduler for graphics processing units that can dynamically support different granularities of parallelism. Finally, we show how these scheduling approaches help to advance the state-of-the-art in the rendering, visualization and procedural modeling.

Keywords: GPU, scheduling, parallel computing, memory management, task management.

ACM CCS: Computer methodologies → Computer Graphics, Computer systems organization → Architectures → Parallel architectures

DOI 10.1515/itit-2014-1083
Received November 19, 2014; accepted January 28, 2015

1 Introduction

During the last years a paradigm change has taken place in the field of computing. The chip production hit the so-called power wall, rendering increases in clock rate infeasible. To satisfy the ever growing demand for more processing power, chips became increasingly parallel. The first multi-core central processing unit (CPU) simply executed multiple applications in parallel. Unfortunately, such simple strategies are nowadays not sufficient to achieve high performance. Every algorithm must be designed for parallel execution from the start.

***Corresponding author: Markus Steinberger,** Graz University of Technology, Austria, e-mail: steinberger@icg.tugraz.at

The currently most powerful parallel processor is the graphics processing unit (GPU). A current GPU has more than 3000 individual processing cores, which should be fully utilized throughout the entire execution of an algorithm to achieve full performance. Diverse fields, such as genetics, molecular biology, space research, archeology or medicine heavily rely on simulations running on the GPU. Although the use of the GPU is essential in many fields, its efficient use is most often very difficult to achieve. However, the reason for this problem is not to be found at the hardware level, but rather lies with the execution model. In our work, we address this problem and show that a new dynamic scheduling model designed for the execution on massively parallel hardware can turn an inflexible GPU into a device capable of executing highly dynamic algorithms. Using our model, the true execution power of graphics processors becomes available for a wide range of applications, which previously could not be executed on the GPU at all. The details of our solutions can be found in the long version of the theses [6], as well as in focused publications [7–11].

1.1 Limitations of the current execution model

To take a closer look at the problems of the current GPU execution model, it is essential to understand the differences between the CPU and the GPU. While functions on the CPU are executed by a single thread, functions on the GPU, so-called *kernels*, are executed by thousands of threads in parallel. While all threads start executing the same code, they are free to take different execution paths. However, the programmer must keep the special constraints of the graphics hardware in mind. A GPU consists of a number of *multiprocessors*, which execute small groups of threads in lockstep according to a single instruction, multiple data (SIMD) model. Efficient execution on the GPU can only be achieved if all threads within this group choose the same execution paths.

Explicit parallelism
The first limitation of the traditional GPU execution model is that sufficient parallelism must be available in every step of an algorithm. There should be tens of thousands of threads started for each kernel. This thread count must be set up before a kernel is launched. A dynamic adjustment of the executing threads is not possible with the traditional execution model. From a hardware perspective it would be sufficient if at each point in time a large number of coherently executing groups of threads were available. The execution model, however, dictates that thousands of threads need to be started concurrently to achieve good performance. This constraint severely limits the number of algorithms that can be executed on graphics hardware.

Control switches
The second limitation comes from the fact that execution is controlled by the CPU. Most often the output of one kernel forms the input to the next (see Figure 1). This intermediate data needs to be stored in graphics memory, which is a factor 1000 slower than local on chip memory. Often the output of one kernel determines the number of threads to be started in the next kernel. Then, this information needs to be copied from the GPU to the CPU, before the CPU can launch the next kernel. This process is extremely costly, as the GPU remains idle during the slow copy process.

Influence on the execution
The third limitation is that it is not possible to influence the execution of a kernel as soon as it has been submitted for execution. The scheduler on the GPU executes kernels in a simple first-in-first-out (FIFO) manner. Thus, long running background tasks can possible block the GPU while high priority foreground tasks are unpredictably delayed. Ultimately, the absence of priorities does not allow for a concurrent execution of tasks with different characteristics.

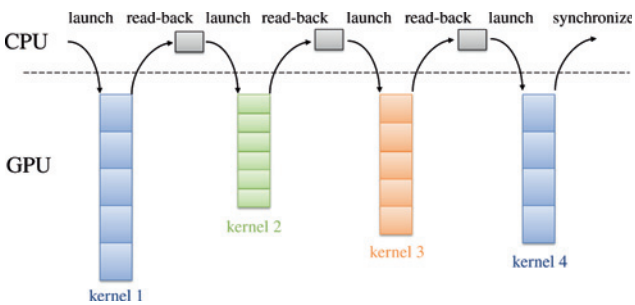

Figure 1: Using the traditional kernel-based programming model, every algorithm is separated into kernels. These kernels are controlled by the CPU which leads to constant back and forth between CPU and GPU.

Dynamic memory allocation
The fourth limitation is the absence of an efficient dynamic memory allocator. Dynamic memory allocation is an essential component for virtually every computer program to dynamically react to variable input. The sole existing dynamic memory allocator for the GPU is part of the NVIDIA CUDA toolkit [4]. This allocator is slow, unreliable, and does not scale well to large numbers of threads.

Knowledge of time
The fifth limitation is the unavailability of time during GPU execution. The GPU does neither provide a common time basis between threads nor between CPU and GPU. Thus, it is, e. g., impossible for an algorithm to react to the time passed since kernel launch. Overall, the knowledge of time is essential for a large number of problems, especially for real-time applications, which need to stick to deadlines.

1.2 Objectives

Our work offers solutions for all previously mentioned problems of the current GPU scheduling model. The objectives of this work can be summarized in six points:

O1 Dynamic algorithms are enabled to autonomously execute with high performance on current graphics hardware.
O2 Efficient scheduling of tasks with varying granularities of parallelism becomes possible in a massively parallel environment like graphics processors.
O3 Efficient dynamic memory management becomes possible on graphics processors, even if tens of thousands of threads allocate memory concurrently.
O4 Scheduling on graphics processors becomes controllable and should react to dynamic changes.
O5 An autonomous scheduler on graphics processors can detect execution characteristics which are suboptimal for the hardware and regenerate healthy execution configuration leading to an overall speedup.
O6 A scheduling system fulfilling **O1**–**O5** will allow a wider range of algorithms to be executed on massively parallel hardware and thus harness the true processing power of the GPU.

We consider the objects to be fulfilled only, if artificial tests as well as real world applications show significant improvements using our algorithms compared to the traditional GPU execution model.

2 Scheduling model

To be able to implement new scheduling strategies on the GPU, we replace the currently prevalent GPU scheduling model – the stream processing model – by a more flexible model. Our model, *Softshell*, does not require all data to be available before execution. Softshell views algorithms as dynamic, irregular, data dependent execution paths. To realize our model, we define five scheduling entities:

- **Work items** describe data to be processed by a single thread,
- **Item sets** describe data to be processed by a small group of threads,
- **Workpackages** define work for a large group of threads that require synchronization primitives to cooperatively work on the input data,
- **Procedures** describe the functions executed for an item, item set, or package.
- **Events** are triggered by the user and initiate the execution by generating items, item sets and packages.

Using these entities, a large class of parallel algorithms can be described, includes all algorithms that can be written in the stream processing model. Additionally, using Softshell, every small concentration of parallelism can be expressed and revealed to the underlying scheduler.

3 Scheduling algorithms

To reach the aforementioned objectives, we adapt and redesign multiple algorithms for massively parallel architectures. The most important innovations can be separated in four categories: massively parallel queuing, work priorities, dynamic memory management, and execution of heterogeneous procedures.

Massively parallel queuing

At the foundation of most scheduling systems are queuing algorithms. A queue provides the means to collect, combine, and distribute work items, item sets, and workpackages. Previous queues for parallel architectures always assumed that so-called *non-blocking* algorithms achieve the best performance. In our work, we show that well-designed queues which fall into the category of blocking algorithms are superior to their non-blocking counterparts. Our most efficient parallel queue allows an arbitrary number of threads to access the queue concurrently, while securing element access with per-spot flags. Using these flags, we can assure that blocking only occurs when the queue runs low on elements. With this strategy, our queue significantly outperforms the previous state-of-the-art, making it the most efficient queue for massively parallel environments.

Work priorities

One of the core features of Softshell are work priorities. By enabling dynamic priorities at the granularity of individual workpackages, a variety of scheduling strategies can be implemented. As it is not feasible to implement priority queues using sorted lists or heaps on the GPU, we devised a different approach. We use an adaptive, progressive, non-invasive queue sorting algorithm which avoids synchronization with enqueue and dequeue operations. This sorting algorithm can, therefore, handle dynamically changing work priorities with hardly any influence on performance.

Dynamic memory management

Dynamic memory management is well-studied for CPU execution. However, this research cannot be directly transferred to GPU execution, as the architecture difference is too severe. To handle tens of thousands of concurrent memory requests efficiently, we propose a new dynamic memory allocator for massively parallel environments: *ScatterAlloc*. ScatterAlloc avoids collisions and thus synchronization by distributing memory requests with the help of a hash function. By carefully choosing the hash function, it is possible to influence the distance between memory requests of different threads. In this way, we can generate memory access patterns that are very efficient on graphics hardware. Thanks to its design, ScatterAlloc forms the currently most efficient memory allocator for the GPU, outperforming commercial memory allocators by multiple orders of magnitude.

Execution of heterogeneous procedures

To offer the execution power of the GPU to dynamic algorithms with distributed parallelism, it is essential to execute all parts of an algorithm with highest efficiency. To achieve this goal, we present an execution and scheduling model which is able to dynamically distribute the available resources between work items, item sets and workpackages. At the same time, our approach enables executing procedures to make use of different feature levels and guaranties available on the GPU. Due to its adaptability and load balancing features, we call our approach *Whippletree*. With its special design, Whippletree takes over the control of the entire GPU, communicates with the under-

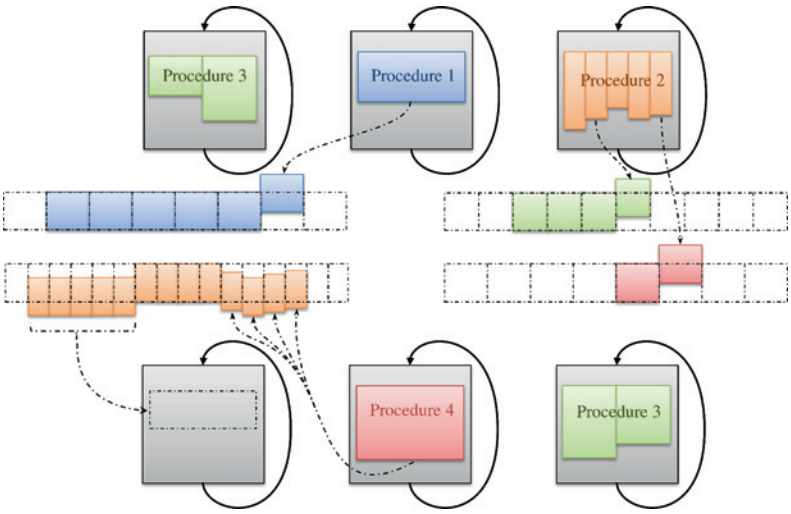

Figure 2: Our Whippletree Megakernel maintains a queue for each procedure. Workers continuously pull these queues. If there are enough tasks in a queue, the workers remove them and dynamically assign threads to the incoming work. Hence we are able to guarantee a high number of active threads independent of the thread configuration required by the individual tasks.

lying work queues and dynamically assigns the available resources to the incoming work (see Figure 2).

4 Application scenarios

Our scheduling strategies not only show outstanding performance in synthetic tests, but also in a variety of application scenarios. In the following, we present a brief excerpt of algorithms that either profit from our scheduling techniques or are – for the first time – able to execute on the GPU using our approach.

Real-time Reyes pipeline

Reyes-style rendering [1] is mainly used for cinematic production to generate high quality imagery. A Reyes implementation on the GPU is difficult, because the pipeline is recursive, irregular, and strongly expanding. The basic idea behind Reyes is the subdivision of scene geometry into a multitude of so-called micropolygons. These micropolygons are so small that they can be approximated with simple quads to still generate high quality renderings. The pipeline is usually separated into four to six states. While previous approaches tried to implement these stages as separate kernels [5, 12, 13], a complete description is possible using our model (see Figure 3). Our model executes the entire pipeline as one entity, enabling load balancing between the different pipeline stages. For the example shown in Figure 3, 11532 input primitives are subdivided using more than one million intermediate work descriptor. Overall, 99 million micropolygons are generated. On a current GPU (NVIDIA GTX Titan), the entire process takes less than 25 ms using our scheduler.

Image-based rendering with camera synchronization

Using the so-called image-based visual hull (IBVH) algorithm, approximated depth images for a virtual camera can be generated from a set of segmented input images [2]. Most often, IBVH is too time consuming to generate images of a complete scene in real-time. However, if parts of a scene are (nearly) static, they can be reconstructed with a lower frequency to save execution time. Usually, the reconstruction of a scene is controlled via a motion threshold to determine which parts should be reconstructed with IBVH. While this threshold influences the output quality directly, the execution time is influenced only indirectly. Running the IBVH algorithm for real-time camera input, it

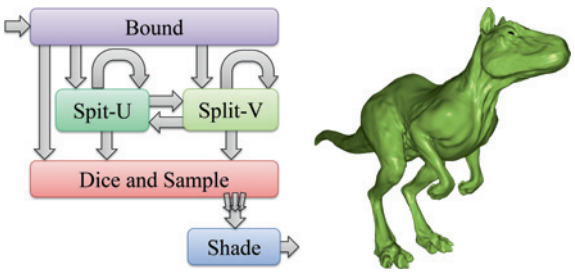

Figure 3: (left) The Reyes pipeline can be modelled with five procedures. The procedure characteristics vary strongly, with 1 to 256 threads and 16 to 63 used registers. (right) This pipeline can be used to render the Killeroo dataset which is subdivided into 99 million micropolygons in real-time.

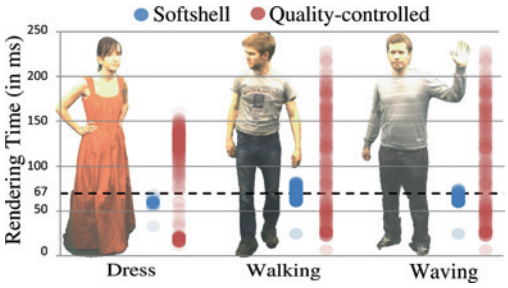

Figure 4: Image-based visual hull rendering trying to match the camera input rate of 67 ms. While a quality controlled algorithm (red) is not able to match the camera rate (dashed line), our scheduling algorithm (blue) is able to follow the input rate. At the same time, our algorithm guarantees that the best possible quality is being generated.

would be desirable to directly control the execution time to synchronize the rendering with the cameras. Using our algorithms, such an approach can be realized easily. The motion estimates can directly be used as execution priorities. In this way, the scheduler reconstructs areas with strong motion first. Having a notion of time, we can replace the IBVH algorithm by simple data reuse, as soon as the next synchronization point approaches. In this way, it is possible to dynamically trade reconstruction quality for execution time while at the same time generating the highest quality image (see Figure 4).

Procedural modeling

Using procedural modeling, entire worlds can be built with little effort. Based on simple rulesets, complex objects, such as buildings, can be generated by the computer. However, the computation time required to build large cities may reach multiple hours. As designers have to adjust the parameters that control the generation process, long computation times lead to a very cumbersome process. To speed up the generation process, researches tried to bring procedural modeling to the GPU. However, up to now, it was not possible to run feature-rich grammars on the GPU, as their derivation process was assumed to be too complex for GPU execution. All previous GPU procedural modeling attempts strongly simplified the supported grammars to achieve parallel execution. Additionally, the observed speedups compared to a CPU execution were marginal.

Using our scheduling strategies, we were able to map the currently most complex grammar (CGA [3]) for GPU execution. As the entire grammar derivation is run on the GPU, it is possible to combine derivation with rendering. Thus, we can limit the generation to those parts of the world which are actually visible. Due to this combination, we can generate detailed metropolis in real-time, while a viewer can move through the city with high speed. An example of such a generated city is shown in Figure 5. Using traditional techniques, the generation of this city would take hours, while we regenerate it 20 times in each second.

5 Conclusion

The GPU processing model previewed in this article is designed to make use of distributed time-varying parallelism. The foundation of our model is formed by the currently fastest queuing algorithm for graphics processors. Our queue assigns individual slots to all thread seeking access. Every slot is protected by its own lock, which enables full parallel access to the queue. Hence, our queue answers requests in constant time – independently of the number of threads accessing the queue concurrently. Combining our queues with the Whippletree Megakernel, it

Figure 5: Thanks to our scheduling strategies, highly detailed cities with complex buildings can be generated in real-time on the GPU. The visible 28 km^2 of this city contain 47 000 buildings, which are generated by 240 million rules. Updating the geometry 20 times per second, we generate fluent renderings even if a viewer moves at supersonic speed through the city.

is possible to execute algorithms with diverse execution characteristics and varying amounts of parallelism efficiently. Due to the adaptive nature of the Whippletree Megakernel, GPU utilization is always kept high. To enable algorithms to adapt to different input parameters, we designed the currently fastest dynamic memory allocator for massively parallel architectures. Our memory allocator reduces the number of accesses to the same memory location by scattering request of individual threads in memory. To be able to control the scattering process, we came up with an intelligent hash function, which creates memory access patterns which are well suited for GPU execution. Finally, we introduced dynamic, fine granular work priorities which provide means to influence the execution order. In this way, it is possible to direct the execution power to the most important parts of an algorithm or prioritize one algorithm over another. Using this priority feature, it is for the first time possible to use advanced scheduling strategies on the GPU, such as fair scheduling, time quota-based scheduling, or earliest deadline first scheduling.

In the future, we expect the degree of parallelism in computing to rise further. Thus, it is of essential importance to provide a high amount of parallelism within applications and algorithms. Basically, there are three options to offer sufficient amounts of parallelism for future GPU architectures. First, an algorithm can be altered to expose a higher amount of parallelism; second, multiple algorithms can be executed concurrently; and third, the scheduling can be advanced so that the hardware can exploit the entire parallelism provided by an algorithm. In all three cases, the techniques and models described in our work can be of significant value. Thus, we expect that the results of our work will influence future GPU programming languages, compilers and schedulers.

References

1. Robert L. Cook, Loren Carpenter, and Edwin Catmull. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, August 1987.
2. Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven J. Gortler, and Leonard McMillan. Image-based visual hulls. In *Proc. SIGGRAPH '00*, SIGGRAPH '00, pages 369–374. ACM, 2000.
3. Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.
4. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. NVIDIA, 2013.
5. Anjul Patney and John D. Owens. Real-time Reyes-style adaptive surface subdivision. *ACM Trans. Graph.*, 27(5):143:1–143:8, December 2008.
6. Markus Steinberger. *Dynamic Resource Scheduling on Graphic Processors*. PhD thesis, Graz University of Technology, 2013.
7. Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Softshell: Dynamic scheduling on GPUs. *ACM Transactions on Graphics (TOG)*, 31(6):161, 2012.
8. Markus Steinberger, Bernhard Kainz, Bernhard Kerbl, Stefan Hauswiesner, Michael Kenzel, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Transactions on Graphics (Proceedings of the ACM SIGGRAPH Asia 2014)*, 36(6), 2014.
9. Markus Steinberger, Michael Kenzel, Bernhard Kainz, Jörg Müller, Peter Wonka, and Dieter Schmalstieg. Parallel Generation of Architecture on the GPU. *Computer Graphics Forum*, 33(2):73–82, 2014.
10. Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, 2012.
11. Markus Steinberger, Michael Kenzel, Bernhard Kainz, Peter Wonka, and Dieter Schmalstieg. On-the-fly Generation and Rendering of Infinite Cities on the GPU. *Computer Graphics Forum*, 33(2):105–114, 2014.
12. Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the GPU. In *Proc. High Performance Graphics*, HPG '10, pages 29–37, 2010.
13. Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. RenderAnts: interactive Reyes rendering on GPUs. *ACM Trans. Graph.*, 28(5):155:1–155:11, December 2009.

Bionotes

MSc PhD Markus Steinberger
Graz University of Technology, Graz, Austria
steinberger@icg.tugraz.at

Markus Steinberger graduated with an MSc in Computer Science (Telematics) from Graz University of Technology, Austria in 2010 with highest distinction. He finished his PhD within the field of GPU scheduling, visualization and geometric modeling in 2013 under the supervision of Prof. Dieter Schmalstieg at Graz University of Technology. He received the highest possible honor for achievement in Austria, the *promotio sub auspiciis prasidentis rei publicae*. In 2013 and 2014 he was with the mobile computer vision research group of Kari Pulli at NVIDIA, California. In 2014 he became the first Austrian to win the GI Dissertation Prize. His wide research interests are reflected by the numerous awards won by his papers, including ACM CHI, IEEE Infovis, Eurographics, ACM NPAR, EG/ACM HPG best paper and honorable mention awards. Since 2014 he is a senior researcher at the Institute for Computer Graphics and Vision at Graz University of Technology, leading the GPU and parallel computing group.