DEPARTMENT: Dissertation Impact

On Dynamic Scheduling for the GPU and its Applications in Computer Graphics and Beyond

Markus Steinberger
Graz University of Technology, Austria

Editor:
Jim Foley
foley@cc.gatech.edu

During the last decade we have witnessed a severe change in computing, as processor clock-rates stopped increasing. Thus, the arguable only way to increase processing power is switching to a parallel computing architecture, like the graphics processing unit (GPU). While a GPU offers tremendous processing power, harnessing this power is often difficult. In our research we tackle this issue, providing various components to allow a wider class of algorithms to execute efficiently on the GPU. These efforts include new processing models for dynamic algorithms with various degrees of parallelism, a versatile task scheduler, based on highly efficient work queues which also support dynamic priority scheduling, and efficient dynamic memory management. Our scheduling strategies advance the state-of-the-art algorithms in the field of rendering, visualization, and geometric modeling. In the field of rendering, we provide algorithms that can significantly speed-up image generation, assigning more processing power to the most important image regions. In the field of geometric modeling we provide the first GPU-based grammar evaluation system that can generate and render cities in real-time which otherwise take hours to generate and could not fit into GPU memory. Finally, we show that mesh processing algorithms can be computed significantly faster on the GPU when parallelizing them with advanced scheduling strategies.

Editor's note: *Markus Steinberger received the 2013 Gesellschaft für Informatik and the 2016 Heinz Zemanek best dissertation awards.*

The field of computing has experienced a significant change in recent years. Up to the early 2000s it was considered a *law* that the clock rate of processors doubles every other year. This increase has come to a full halt due to what is called the *power wall*,[1] as can be seen in Figure 1**.** Thus, researchers and engineers cannot count on their algorithms running faster by simply upgrading to the next generation hardware. Hence, their ability to tackle larger and more important problems in the future—when the technology has further evolved—is not guaranteed any longer. These circumstances lead to unprecedented challenges faced by research and development in many important areas, putting a renewed focus on low-level code efficiency and close-to-hardware algorithm development. At the same time, we can witness a shift in programming paradigms. Although the clock rate is not increasing further, Moore's Law still holds and the transistor count of chips is still growing exponentially. These transistors are spent on parallel execution capabilities, significantly increasing the core count of new processors.

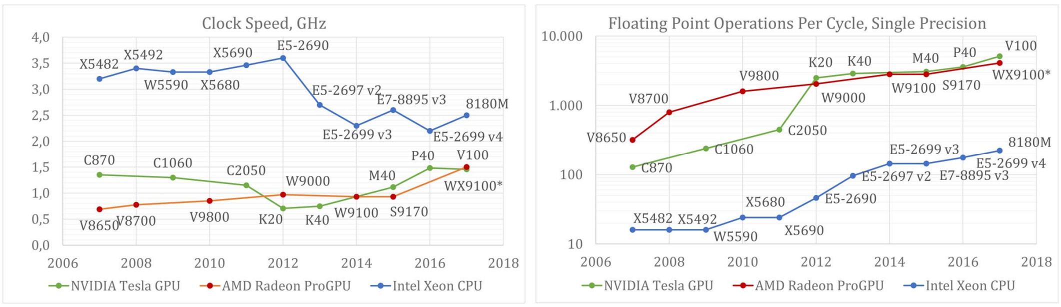

Figure 1. While the clock rate of processors has been stagnant throughout the last decade, the number of operations that can be executed in parallel has been increasing exponentially. The theoretical peak processing power of massively parallel architectures like the GPU is about one order of magnitude higher than the theoretical CPU performance.

While an increase in core count boosts the theoretical compute power, harnessing this power often seems impossible in practice, due to multiple reasons: First, algorithms need to offer sufficient parallel workloads to allow for parallelization across a large number of threads. Second, communication between threads is costly, potentially rendering the performance gains of parallelization negligible. Third, memory access increasingly becomes a bottleneck. Fourth, and maybe foremost, the characteristics of parallel processors are changing rapidly: traditional CPU multi-processing is becoming less important, while single-instruction multiple-data (SIMD) cores on the CPU and massively parallel SIMD compute accelerators like the graphics processing unit (GPU) are gaining increasing relevance for high performance computing. Due to its bare, throughput-oriented design, the GPU offers one order of magnitude higher peak compute power than the CPU, as detailed in Figure 1. These theoretical performance gains, however, are paid for by removing the hardware dynamic execution capabilities and reducing cache sizes. The combination of these facts makes parallel computing, especially on the GPU, not only tedious and inflexible, but virtually impossible to be completed with traditional programming and execution models.

In our work, we address these problems on a software level, working on static and dynamic scheduling models designed for the GPU. Our efforts span low-level data structures, such as work queues for massively parallel devices, communication primitives for different levels of GPU hierarchy, the implementation and support of custom scheduling strategies, new programming models and domain specific languages for the GPU, and auto-tuning and auto-scheduling for complex algorithms. Our tools and strategies have been applied in various fields such as rendering, procedural geometry generation, mesh processing, visualization, image processing, and even low-level sparse linear-algebra. The success of our techniques is rooted in our ability to turn the traditionally inflexible GPU execution model into an interface to a device capable of executing highly dynamic algorithms. Thus, the true execution power of graphics processors becomes available for a wide range of applications, which previously could not be executed on the GPU efficiently.

Limitations of the traditional GPU execution model

To motivate our work—which started around 2010—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

One limitation of the traditional GPU execution model is that large amounts of parallelism must be available in every step of an algorithm to fully utilize the GPU. There should be tens of thousands of threads started for each kernel, with the thread count being specified before the kernel launch. A dynamic adjustment of the executing threads is not possible with the traditional execution model. However, from a hardware perspective it would be sufficient if a large number of coherently executing groups of threads were available at all times. Whether these groups all execute the same function is not important for the hardware. The execution model and its implementation, however, requires 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

Another limitation comes from the fact that execution is controlled externally, traditionally even completely from the CPU. Most often the output of one kernel forms the input to the next. As new kernel launches are externally controlled, all these data need to move from the multiprocessor to global graphics memory, which can be up to two or three orders of magnitude slower than on-chip shared memory. Often, the results of one kernel determine the number of threads to be started in the next kernel. In the traditional model, 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

A third limitation is that it is not possible to influence the execution of a kernel after 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.

Dynamic memory allocation

A 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. Initially there was no dynamic memory allocator provided by GPU vendors. During our work on the topic, NVIDIA had just released a first allocator which was slow and unreliable and did not scale well to large number of threads.

Knowledge of time

A fifth limitation is the fact that time is no explicit construct during GPU execution. The GPU neither provides 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 a 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.

Objectives

Our work during the first years of the project was to provide solutions for all previously mentioned problems of the traditional GPU scheduling model. We initially summarized our goals as follows:

- Dynamic algorithms are enabled to autonomously execute with high performance on current graphics hardware.
- Efficient scheduling of tasks with varying granularities of parallelism becomes possible in a massively parallel environment like graphics processors.
- Efficient dynamic memory management becomes possible on graphics processors, even if tens of thousands of threads allocate memory concurrently.
- Scheduling on graphics processors becomes controllable and should react to dynamic changes.
- 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.
- A scheduling system fulfilling the previous points will allow a wider range of algorithms to be executed on massively parallel hardware and thus harness the true processing power of the GPU.

SCHEDULING MODEL

To be able to implement new scheduling strategies on the GPU, we replaced the prevalent GPU scheduling model—the stream processing model—by a more flexible model. The conventional stream processing model for GPU programming assumes that input data is laid out entirely in memory prior to the call of a kernel. In this way, the entire stream may be executed in parallel.

Softshell

In contrast, our first model, *Softshell*,[2] builds on a more complete adaptation of the original stream processing model. We assume that applications are dynamic and show unpredictable data-dependent execution paths. In this way, any portion of input data can become available at any point in time. Thus, the parallelism is not required to only be spatial; it may also be temporal: Multiple tasks can run in parallel, and data can be added to arbitrary input streams at any time. Furthermore, Softshell also considers cases in which multiple algorithms with different time characteristics run concurrently. Therefore, we allow for arbitrarily changing priorities to enable well-tuned scheduling strategies.

To describe the Softshell processing model as outlined in Figure 2, we introduce the notion of work items, workpackages, procedures and events:

- Work items describe data to be processed by a single thread,
- Workpackages define a collection of work items,
- Procedures describe the functions executed for a work item,
- Events are triggered by the user and initiate the execution by generating work items and workpackages.

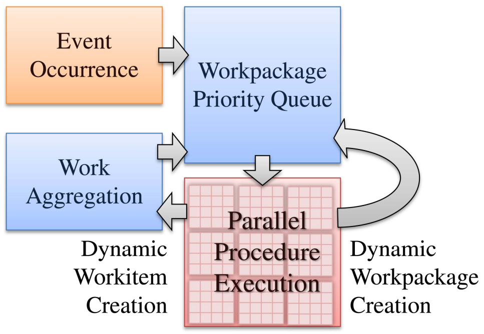
Event Occurrence
Workpackage Priority Queue
Work Aggregation
Parallel Procedure Execution
Dynamic Workitem Creation
Dynamic Workpackage Creation

Figure 2. The Softshell processing model enables dynamic work generation during execution on the GPU and aggregates work items for coherent execution.

Procedures define the execution steps in Softshell. In contrast to kernels, which are intended to fill the entire GPU with thousands of threads, procedures are designed for small groups of coherent threads. Thus, a procedure is well suited for the execution on a single or few SIMD units. To fully occupy a GPU, Softshell executes multiple (different) procedures in parallel. Procedures require input parameters, which are formed by workpackages. Depending on the workpackage, the active number of threads for a procedure's execution is adjusted by Softshell. Softshell will usually not interrupt the execution of a procedure and make scheduling decisions before starting the next procedure.

Work items and Workpackages describe the input data for procedures. While procedures form the description of the execution steps, the combination with work items and workpackages allow Softshell to track what is to be executed. Work items correspond to work that is to be executed by a single thread. These work items can be combined to workpackages, which are intended to be executed in parallel by groups of coherent threads within a procedure.

From a programming model point of view, an algorithm can be constructed like a graph with Softshell. Procedures form the nodes of the graph. Work items and workpackages are added to queues that feed the procedure nodes. Work items and workpackages might be merged while they are waiting for execution.

Events initiate the execution of an algorithm. This initialization can be explicitly triggered by a user supplying specific input, new data becoming available, or a timer. When an event is triggered, it queues initial workpackages for a set of procedures. Work created during the execution of these workpackages is again associated with the original event. The event is considered to be completed when all associated workpackages have been processed. In this way, the application can track the progress or cancel an algorithm represented by an event.

Whippletree

Although Softshell allows dynamic algorithms to run on the GPU, the scheduling model still boils down to the concept of threads (and thread blocks). This abstraction hides important facts from the programmer: On each level of the GPU execution hierarchy, application code can rely on certain assumptions of decreasing strength. At the lowest level, a group of threads executing on the same SIMD unit can rely on implicit synchronization, access to on-chip shared memory, and intra-group communication. One level above, threads running on the same multiprocessor can still use on-chip shared memory and barrier synchronization. At the highest level—across multiprocessors—code may only use operations on global memory. Taking advantage of these guarantees is essential for unlocking the full potential of the GPU.

To this end, we introduced *Whippletree*,[3] a new programming model for the GPU that offers the simplicity and expressiveness of task-based parallelism while enabling application code to take full advantage of the hardware. Whippletree distinguishes three task types that match the GPU execution hierarchy:

Level-0 tasks are multi-threaded tasks that must be executed by threads on the same SIMD unit. Thus, the implementation of such a task can use warp-level features. The number of threads working on a level-0 task is therefore limited by the SIMD width of the device. For efficiency reasons, multiple level-0 tasks of the same kind will be executed on the same SIMD unit if possible. To achieve optimal grouping, a level-0 task's thread count should therefore be an integer factor of the SIMD width. Recently this concept of sub-SIMD-width groups has been picked up by NVIDIA, introducing explicit communication and synchronization features for these kind of thread groups.[4]

Level-1 tasks are multi-threaded tasks with all threads being executed on the same multiprocessor. Block-level features like on-chip shared memory and barrier synchronization can be used. For efficient execution, the thread count should be chosen to be a multiple of the SIMD width. Multiple level-1 tasks might be executed on the same multiprocessor

Level-2 tasks are single-threaded tasks. Only GPU-wide features such as global memory are supported. Thus, level-2 tasks can be arbitrarily assigned to any multiprocessor. Multiple level-2 tasks of the same kind may be grouped to fill up a SIMD unit or multiprocessor, maximizing GPU utilization while trying to keep thread divergence low.

The Whippletree model allows even more control over GPU execution than Softshell. At the same time, it increases the demands on the low level scheduling algorithms, as a combination of tasks on multiple levels of the hierarchy need to be collected and combined.

SCHEDULING ALGORITHMS

To reach the objectives mentioned in the beginning and to realize the Softshell and Whippletree programming models on the GPU, we redesigned multiple algorithms for massively parallel architectures. The most important innovations can be separated in three categories: massively parallel queuing, work priorities, and dynamic memory management.

Massively parallel queuing

At the core of all our scheduling strategies are concurrent queues, which collect and distribute work; usually in a first in, first out (FIFO) manner. The available literature on concurrent queues has a strong focus on lock-freedom, which is often held as key to performance in concurrent systems. Yet the lock-free property is often achieved through methods in the spirit of optimistic concurrency control, i.e., methods built upon the assumption of low resource contention. In our work, we have developed multiple queue designs for the GPU that contradict that assumption and have shown that the additional cost of redundant operations of those queues can outweigh the benefits of true lock-freedom.

Our most efficient parallel queue allows an arbitrary number of threads to access the queue concurrently, while securing element access with per-spot flags.[3] Using these flags, we can assure that blocking only occurs when the queue runs low on elements or is running out of space. With this strategy, our queue outperforms previous lock-free algorithms by multiple order of magnitude, enabling efficient queuing on massively parallel devices and thus the backbone of all our scheduling strategies.

Most recently, we formalized our queue design and showed how the queue can be designed to conform to a fully linearizable FIFO queue.[5] In detail, we show that our queue:

- supports all execution paradigms common on the GPU and thus all three levels of task types;
- ensures that enqueue/dequeue are non-blocking if the queue is full/empty, thus enabling multi-queue setups, which are essential for advanced scheduling;
- can store arbitrarily sized data in a ring-buffer and thus avoid costly memory allocation,
- avoid optimistic concurrency control, assigning threads a unique spot in the queue after a fixed number of operations, as long as the queue is not empty or full; and

- only shows a blocking behavior between threads assigned to the same queue element to avoid read-before write hazards, pushing the blocking condition to edge cases when the queue is nearly full or empty.

Priority scheduling

One of the core features of our work on scheduling is the support for priorities. In our early work on Softshell, we have shown that a monolithic queue approach allows for priorities at the granularity of individual workpackages, and thus a variety of scheduling strategies can be implemented. As it is not feasible to implement priority queues using sorted linked lists or heaps on the GPU, we devised an iterative sorting approach that works in parallel with scheduling from that queue. To this end, we use an adaptive, progressive, non-invasive queue sorting algorithm which avoids synchronization with enqueue and dequeue operations.[2] This sorting algorithm can, therefore, handle dynamically changing priorities with hardly any influence on performance. We have shown that this kind of priority scheduling can be used to speed up path-tracing or adapt algorithms to run closer to a target (input) framerate, as shown in Figure 3.

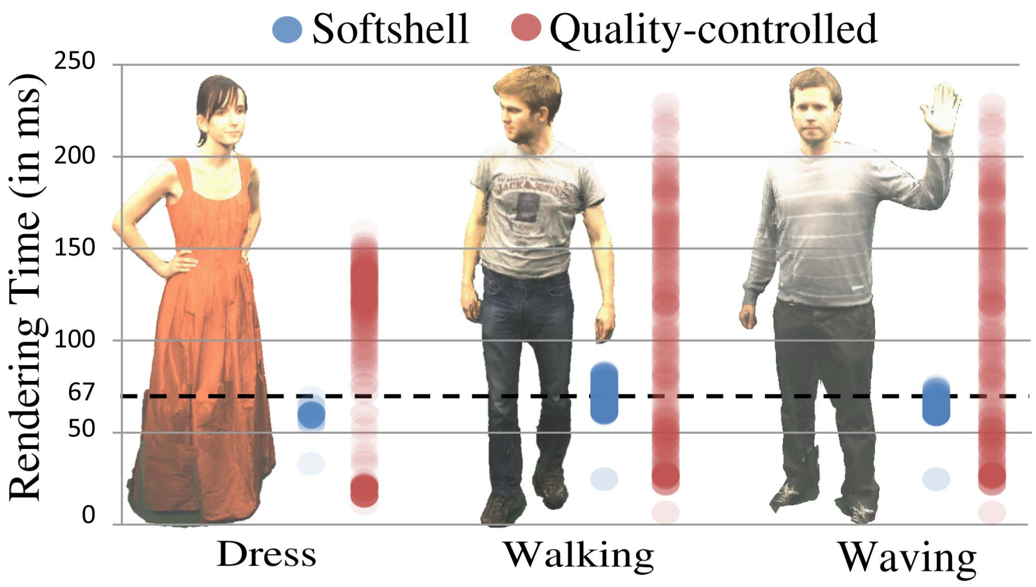

Figure 3. Image-based visual hull rendering (IBVH) with a target time-constraint (dotted line). Prioritizing areas of high change, Softshell uses IBVH to focus on those image areas which will contribute the most to visual quality. As the deadline is being approached, the system dynamically switches to low latency image warping instead, making sure that we stay close to the target framerate (blue dots). Traditional static switching between the techniques varies strongly around the deadline (red dots).

Recently, we have introduced an even more evolved scheduling approach: hierarchical bucket queuing.[6] By building a hierarchy of queues on it the GPU, it is possible to set up a variety of scheduling policies. Traversing the bucket hierarchy during enqueue allows a sorting of tasks into any number of categories, before dequeue, another traversal indicates which category is currently considered the most important. By controlling both traversals, bucket queues can be tailored to the needs of the application, not only for priority scheduling, but also to combine tasks or integrate multiple processes into a single scheduler. They enable simple scheduling strategies, like FIFO or round-robin scheduling, as well as more complex strategies, like earliest-deadline-first or fair scheduling. Implementing the hierarchy traversal with efficiency and parallelization in mind, priority scheduling using bucket queues always outperformed our previous approaches based on sorting. Even in challenging situations our approach only adds a small overhead (between 1% to 5%) compared to non-hierarchical scheduling.

Due to the efficiency of our approach, we have demonstrated, that real-time rendering approaches which are latency sensitive can profit significantly from our approach. For example, latency-controlled, foveated micropolygon rendering is enabled by the ability to direct the processing power to a focus region and stop the subdivision process, as the frame time becomes critical Figure 4. However, while priority-based rendering is certainly interesting for future virtual reality systems, it will require additional work to integrate priorities deeper into the rendering pipeline. To this end, we believe it is necessary to rethink the execution of rendering pipelines on the GPU, moving rendering closer to a dynamic process, which combines software and hardware stages in a more flexible way. Our upcoming work is one step into that direction.[7]

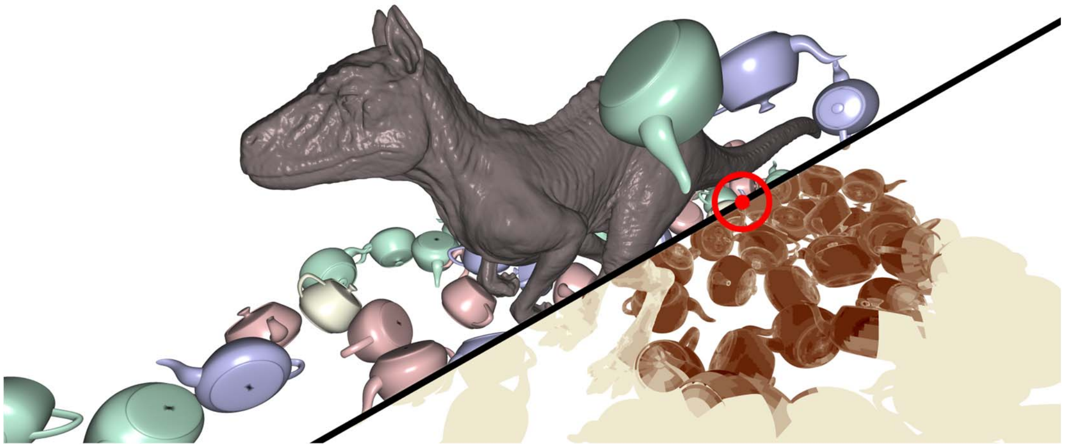
Figure 4. Hierarchical bucket queues enable the implementation of foveated, micropolygon rendering in real-time on the GPU. By dynamically reducing the number recursions in areas distant from the user's current fixation point, rendering quality is adaptively decreased.

Dynamic memory management

In the earlier days of GPU programming, one of the core issues for dynamic algorithms running on the GPU, was the non-existence of a memory manager. While dynamic memory management is well-studied for CPU execution, we quickly found that simply porting CPU strategies to the GPU will not lead to success, as the architecture and demands are too different.[8] While on the CPU, the maximum number of threads that try to allocate memory at the same time is typically in the order of the core count, i.e., between four and twelve, on the GPU tens of thousands of threads might allocate memory in parallel.

Even atomic operations on a single congestion point could not be handled by the hardware fast enough to provide efficient memory management. Thus, to handle tens of thousands of concurrent memory requests efficiently, we proposed a different strategy, completely avoiding single point of congestion: *ScatterAlloc*.[8] ScatterAlloc avoids collisions and thus synchronization by distributing memory requests with the help of a hash function and managing allocated slots with simple bit fields. By carefully choosing the hash function, it is even possible to influence the distance between memory requests of different threads. In this way, we can generate memory access patterns that are efficient on graphics hardware. Thanks to its design, ScatterAlloc was multiple order of magnitude faster than all other memory allocators for the GPU and is, today, still the fastest allocator when a large number of threads allocate small amounts of memory.

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.

Shape Grammars

Both the game and movie industry are increasing their demands for ever larger and yet more detailed environments. Creating these environments is a tedious and time-consuming task for digital artists. Shape grammars have the potential to drastically reduce these manual efforts: Given just a small set of rules, a procedural grammar can generate entire worlds. Unfortunately, the procedural derivation of large cities can take up hours, even on a modern system, greatly limiting the practical usefulness of shape grammars in an iterative design process.

Building on our GPU scheduling efforts, we have derived a novel approach for the parallel evaluation of procedural shape grammars on the GPU: PGA.[9] Unlike previous approaches that were limited in the types of shapes they support, the amount of parallelism they can use, or both, our approach supports state-of-the-art procedural modeling. To increase parallelism, we explicitly express separation between rules, reduce inter-rule dependencies, and introduce intra-rule parallelism. Using our scheduling approaches, we avoid unnecessary back and forth between CPU and GPU and reduce round trips to slow global memory. In this way, our approach is up to 10000 times faster than the standard in CPU-based shape grammar evaluation, while offering equal expressive power. In comparison to previous GPU systems, our approach is nearly 50 times faster, while not sacrificing expressive power of the grammar. Using our parallel shape grammar, the authoring and editing process of procedurally generated architecture can be enriched with instant feedback.

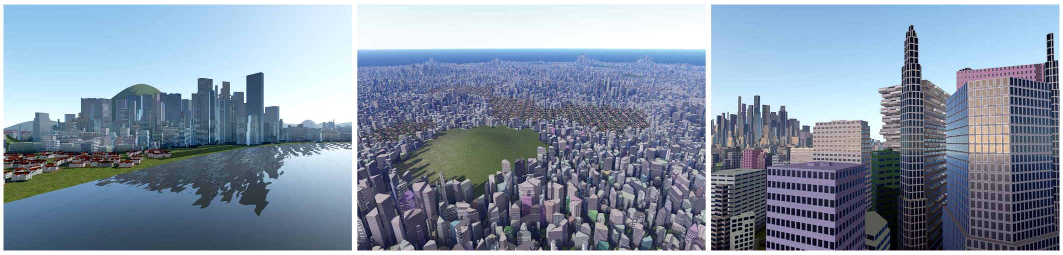
Figure 5. Infinite cities with highly detailed buildings can be generated and rendered on demand in real-time on the GPU using our parallel shape grammar approach interwoven with rendering. the visible 28 km² of the city contain up to 47000 buildings. In full detail, these buildings are generated by 240 million rules, creating 2 billion polygons. Generating an initial view with automatic level-of-detail (7 million polygons) takes 500ms. Exploiting frame-to-frame coherence, we update the geometry in 50ms per frame.

Building on our efficient shape grammar evaluation system, we showed that rendering and shape-grammar-based generation can interwoven to generate and rendering infinity cities in real-time on the GPU.[10] Traditional approaches evaluate a shape grammar once as a preprocessing step and stream the produced geometry during rendering to the GPU. We overcome the problems of streaming by evaluating the shape grammar directly on the GPU for those objects that are visible in the current frame only. This approach can lead to vast improvements in evaluation speed, memory consumption and rendering speed. Only evaluation the shape grammar for objects that are inside the viewing frustum and are not occluded by other objects that are being generated, we skip the evaluation of 75 to 90% if buildings in a typical city scenario. To further reduce the number of processed shapes and allow for highly detailed cities, we proposed an automatic approach for the generation and insertion of what we call surrogate terminals, which are level-of-detail approximations directly derived from the shape grammar and inserted dynamically in the evaluation process. This approach reduces the amount of geometry by another 90%. To reach full real-time performance, we further exploit frame-to-frame coherence, which requires dynamic memory management consistently over hundreds of frames. In this way, we generate and update cities with 47000 visible buildings at 20 fps, even when the viewer is moving at supersonic speed. Example views of such a city are shown in Figure 5.

Inverse Procedural Modeling

While our previous approaches towards procedural modeling on the GPU only considered shape grammars and can be classified as a hand-crafted solution, we continued this line of research towards more systematic scheduling. To this end, we introduced the concept of operator graph scheduling[11] for high performance procedural generation. The operator graph can be seen as an intermediate representation that describes all possible operations and objects that can arise during a procedural generation of any kind. The operator graph is applicable to all procedural generation methods that can be described by a graph, such as L-systems, shape grammars, or stack based generation methods. Building on the operator graph, we show that all its partitions correspond to possible ways of scheduling a procedural generation on the GPU, including the scheduling strategies of previous work. Using Whippletree we can realize any of these strategies, which may show vastly different performance. As the space of possible partitions is exponentially large, we furthermore investigated strategies to reduce the search space, aiding an autotuner to finding the fastest valid schedule for any given operator graph. The best schedule found by our system achieves performance improvements of 8x to 30x over the previous state of the art in GPU shape grammar and L-system generation.

Building on the operator graph representation and highly efficient procedural generation, we investigated machine learning for procedural model that optimizes the output geometry towards various goals.[12] To this end, combine genetic algorithms with our operator graph representation and encode the graph structure into a hierarchical genome representation. In combination with mutation and reproduction operations specifically designed for controlled procedural modeling, our genetic algorithm can evolve a population of individual models close to any high-level goal. Possible scenarios include a volume that should be filled by a procedurally grown tree or a painted silhouette that should be followed by the skyline of a procedurally generated city. These goals are easy to set up for an artist compared to the tens of thousands of variables that describe the generated model and are chosen by the genetic algorithm. Previous approaches for controlled procedural modeling either converge slowly, requiring multiple hours for the optimization of larger models or work under tight time constraints for but get stuck in bad choices quickly. Our approach combines the best of both approaches and converges to a high-quality solution efficiently. An example is shown in Figure 6.

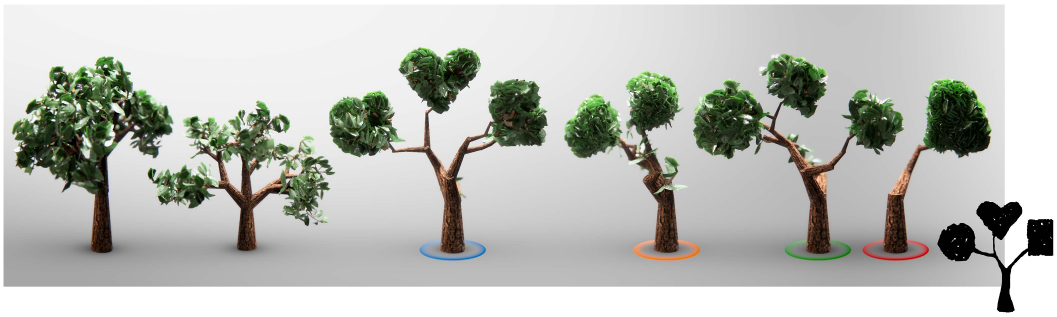
Figure 6. (Left) Procedural generation can generate a large variety of models. Inverse procedural modeling takes a high level target, like the sketch on the right, and evolves the procedural generation towards this goal. Previous approaches are either tuned for speed (red circle and orange circle) and thus often do not reach the goal, or require a long time to achieve a good result (green circle). Our proposed solution converges fast and achieves high quality results (blue circle).

Mesh Processing

A key advantage of working with structured grids, as for example images, is the ability to directly tap into the powerful machinery of linear algebra and large-scale parallelization for many operations. This is not much so for unstructured grids, which used across a variety of disciplines spanning simulation, manufacturing, health care, and entertainment. Thus harnessing their unstructured nature is of paramount importance as it deeply impacts the overall performance of algorithms.

Unstructured grids rely on intermediate data structures for grid traversal, which complicate parallelization of grid algorithms. Especially on the GPU, the conventional ideas behind these intermediate structures is challenged by costly memory access. Introducing a a sparse matrix representation for unstructured grids,[13] we not only reduce memory requirements but also cut down on the bulk of data movement. In combination with our scheduling systems and GPU linear algebra algorithms,[14,15] we show that geometric computations, topological modifications, and the demanding assembly process of standard graph and finite element matrices can be executed efficiently on the GPU, see Figure 7.

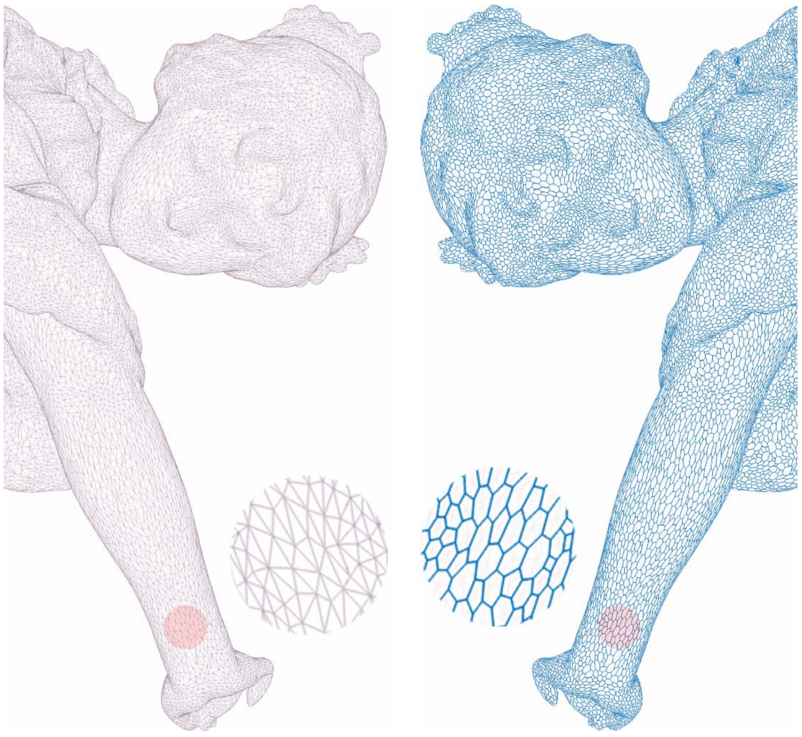

Figure 7. Using our sparse linear algebra formulations and efficient scheduling on the GPU, we can, e.g., take a mesh (right) and create its dual (left) efficiently. Model depicted in the figure is a dancer from 1910, courtesy of the Smithsonian.

SUMMARY

With our research, we did not only provide the fastest queuing mechanisms for the GPU, the fastest dynamic memory allocator for massively parallel architectures, and the only autonomous GPU scheduling framework that can handle different granularities of parallelism efficiently, we also showed the advantages of our model in comparison to state-of-the-art algorithms in multiple fields in computer graphics. In the future, we will focus on alternative rendering pipeline designs, moving real-time rendering towards more compute-based flexible approaches;[7] large scale dynamic graph algorithms;[16] and efficient linear algebra computations on the GPU.[15] We believe that our scheduling efforts will find a deeper integration into upcoming GPU architectures, as can already be witnessed in the most recent NVIDIA architecture, which introduced explicit communication and synchronization mechanism for smaller thread groups.[4] Our scheduling algorithms have been integrated into various other research projects and industrial applications. They are always offered as open source and can be downloaded from https://www.tugraz.at/institute/icg/research/team-steinberger.

REFERENCES

1. H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” *Dr. Dobb’s Journal*, vol. 30, no. 3, 2005, pp. 202–210.
2. m. Steinberger et al., “Softshell: dynamic scheduling on GPUs,” *ACM Transactions on Graphics*, vol. 31, no. 6, 2012, p. 161.
3. M. Steinberger et al., “Whippletree: task-based scheduling of dynamic workloads on the GPU,” *ACM Transactions on Graphics*, vol. 33, no. 6, 2014, p. 228.
4. *NVIDIA Tesla V100 GPU Architecture*, NVIDIA, 2017.
5. B. Kerbl et al., “The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU,” *International Conference on Supercomputing*, 2018.
6. B. Kerbl et al., “Hierarchical Bucket Queuing for Fine-Grained Priority Scheduling on the GPU,” *Computer Graphics Forum*, vol. 36, no. 8, 2016, pp. 232–246.
7. M. Kenzel et al., “A High-Performance Software Graphics Pipeline Architecture for the GPU,” *ACM Transactions on Graphics*, vol. 37, no. 4, 2018, p. 15.
8. M. Steinberger et al., “ScatterAlloc: Massively parallel dynamic memory allocation for the GPU,” *Innovative Parallel Computing*, 2012.
9. M. Steinberger et al., “Parallel Generation of Architecture on the GPU,” *Computer Graphics Forum*, vol. 33, no. 2, 2014, pp. 73–82.
10. M. Steinberger et al., “On-the-fly Generation and Rendering of Infinite Cities on the GPU,” *Computer Graphics Forum*, vol. 33, no. 2, 2014, pp. 105–114.
11. P. Boechat et al., “Representing and Scheduling Procedural Generation using Operator Graphs,” *ACM Transactions on Graphics*, vol. 35, no. 6, 2016, p. 183:1.
12. K. Haubenwallner, H.-P. Seidel, and M. Steinberger, “ShapeGenetics: Using Genetic Algorithms for Procedural Modeling,” *Computer Graphics Forum*, vol. 36, no. 2, 2017.
13. R. Zayer, M. Steinberger, and H.-P. Seidel, “A GPU-adapted Structure for Unstructured Grids,” *Computer Graphics Forum*, vol. 36, no. 2, 2017.
14. A. Derler et al., “Dynamic Scheduling for Efficient Hierarchical Sparse Matrix Operations on the GPU,” *International Conference on Supercomputing*, 2017.
15. M. Steinberger, R. Zayer, and H.-P. Seidel, “Globally Homogeneous, Locally Adaptive Sparse Matrix-vector Multiplication on the GPU,” *International Conference on Supercomputing*, 2017.
16. M. Winter, R. Zayer, and M. Steinberger, “Autonomous, Independent Management of Dynamic Graphs on GPUs,” *High Performance Extreme Computing Conference*, 2017.

ABOUT THE AUTHOR

Markus Steinberger is an assistant professor at Graz University of Technology, Austria, leading the GPU Computing and Visualization Group. His biggest honors include the promotion sub auspiciis prasidentis rei publicae; being the first Austrian to win the GI Dissertation Prize, and winning the Heinz Zemanek Prize. His research interests are reflected by his awards, including ACM CHI, IEEE Infovis, Eurographics, ACM NPAR, EG/ACM HPG, and IEEE HPEC best paper and honorable mention awards. Contact him at steinberger@icg.tugraz.at.

Contact department editor Jim Foley at foley@cc.gatech.edu.