

Softshell: Dynamic Scheduling on GPUs

Markus Steinberger* Bernhard Kainz* Bernhard Kerbl Stefan Hauswiesner* Michael Kenzel Dieter Schmalstieg*
Graz University of Technology, Austria

Abstract

In this paper we present Softshell, a novel execution model for devices composed of multiple processing cores operating in a single instruction, multiple data fashion, such as graphics processing units (GPUs). The Softshell model is intuitive and more flexible than the kernel-based adaption of the stream processing model, which is currently the dominant model for general purpose GPU computation. Using the Softshell model, algorithms with a relatively low local degree of parallelism can execute efficiently on massively parallel architectures. Softshell has the following distinct advantages: (1) work can be dynamically issued directly on the device, eliminating the need for synchronization with an external source, *i.e.*, the CPU; (2) its three-tier dynamic scheduler supports arbitrary scheduling strategies, including dynamic priorities and real-time scheduling; and (3) the user can influence, pause, and cancel work already submitted for parallel execution. The Softshell processing model thus brings capabilities to GPU architectures that were previously only known from operating-system designs and reserved for CPU programming. As a proof of our claims, we present a publicly available implementation of the Softshell processing model realized on top of CUDA. The benchmarks of this implementation demonstrate that our processing model is easy to use and also performs substantially better than the state-of-the-art kernel-based processing model for problems that have been difficult to parallelize in the past.

CR Categories: I.3.6 [Computer Graphics]: Methodology and Techniques—Languages; I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing;

Keywords: priority scheduling, GPU, priority work queue, real-time scheduling, persistent threads, dynamic parallelism

Links:  DL  PDF  WEB  CODE

1 Introduction

Over the last decade, parallel computing has become increasingly available to a wide audience, largely due to graphics processing units (GPUs), which have evolved into massively-parallel general-purpose co-processors. GPU hardware is evolving rapidly, eliminating the drawbacks of previous designs with the introduction of a multi-level cache or stack [NVIDIA 2009]. On the software side, new programming languages such as CUDA have advanced, but the processing model itself has mostly remained untouched since the beginning of general-purpose GPU programming. This model inherently has several limitations:

*e-mail: {steinberger, kainz, hauswiesner, schmalstieg}@icg.tugraz.at

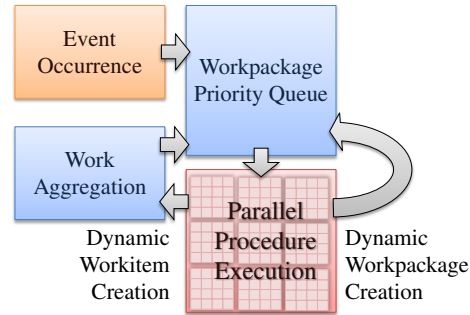


Figure 1: The Softshell processing model: Due to an event, new work becomes available for execution on a GPU. Work items are grouped in workpackages, waiting for their parallel execution in a priority queue. When a SIMD processing unit becomes available, it draws the highest priority workpackage from the queue and runs the procedure associated with it. During execution, new work can dynamically be created on the GPU, either in form of entire workpackages or as single work items to be automatically aggregated. In this way, algorithms with various degrees of parallelism can efficiently be mapped to SIMD architectures.

The first limitation is that sufficient data parallelism is required in every stage of an algorithm. These stages are captured by *kernel* function calls. A fixed number of threads are launched for a single kernel and start executing the same code. There is no way to dynamically adjust the parallelism during a single kernel launch. While from a hardware perspective it would be sufficient if a high number of coherently executing thread groups are available, good performance using *kernel* functions can only be achieved, if kernels are started for thousands of threads [NVIDIA 2011; Khronos 2008]. This rigid requirement often impairs the straight forward mapping of common algorithms for GPU execution. Such a problem is, for example, traversing a tree and executing an operation for every node. One way of parallelizing these classes of algorithms is to subsequently launch a kernel for each tree level. For non-trivial applications, the local tree depth may strongly vary, resulting in underutilization if there are not enough tree nodes available. Additionally, determining the number of nodes per level and mapping them to threads requires synchronization after each level and inefficient parallel reduction methods. The easier and more efficient solution would be to launch new threads for every child node dynamically. Many other problems in graphics show a similar dependency between control flow and parallelism.

The second limitation is due to the fact that kernel launches are entirely controlled by the CPU. Therefore, GPU/CPU synchronization between kernel launches is necessary, if decisions concerning the next kernel launch must be made. The overhead of passing control back and forth between the GPU and CPU can be omitted if the execution of new tasks could be initiated on the GPU itself. Additionally, there is currently no way to interrupt or terminate the execution of a running kernel. Interrupting events can only be considered after a kernel has finished. However, if a user changes input parameters – for example in an interactive visualization – the current kernel launch may become obsolete and any further computations become useless.

The third limitation is, to the best of our knowledge, that currently no system provides GPUs with a functionality similar to *work scheduling*, which is common practice on CPUs and highly exploited by modern operating systems. Scheduling on the GPU is currently based on a simple first-in first-out (FIFO) handling. Thus, work-intensive background tasks can block the entire GPU and delay the execution of high-priority foreground tasks. The absence of a priority-based scheduler makes it impossible to use GPUs for tasks with diverse execution characteristics.

The fourth limitation is the lack of time-awareness during GPU execution. The notion of time is very important for many problems, especially for real-time graphics that must fulfill at least soft real-time constraints.

To tackle these shortcomings, we propose Softshell as outlined in Figure 1. Softshell is a flexible processing model for GPUs that allows a wide range of algorithms to be executed efficiently and autonomously on the GPU. Our main contributions can be summarized as follows:

- We propose the flexible *Softshell* processing model based on the concepts of *events*, *procedures* and *workpackages*, exploiting current GPUs better than previous approaches.
- We design this model in a way that facilitates *arbitrary amounts of work at any point in time* and distributes the work across available processing cores without the need to involve the CPU.
- We describe a novel, three-tier *dynamic scheduler*, which supports dynamic per workpackage priorities that can change at any point in time.
- We introduce a messaging interface that allows bidirectional communication between the CPU and every processing core on the GPU, even during GPU execution.
- We enable *real-time scheduling* by introducing a coherent system-wide time source.
- We provide an implementation of the Softshell model and demonstrate its advantages. It is freely available for download and does not depend on any additional third-party libraries.

2 Related work

Scheduling for massively parallel, SIMD architectures is a young discipline. GPUs have only been available as streaming co-processors for a few years [McCool et al. 2002; Buck et al. 2004]. Thus, few approaches target scheduling strategies for GPUs, and those that exist are rather limited. However, scheduling is an important and mature problem in operating system design [Tanenbaum 2007]. Unfortunately, GPU architectures differ from CPU architectures in several fundamental properties, making a simple adaption of traditional strategies infeasible.

Built-in GPU scheduling is performed on different levels for architectures as shipped today [NVIDIA 2009]. Kernels are split up into blocks of threads, which are executed on different multiprocessors. On each multiprocessor, a thread block scheduler switches between the most suitable threads to be executed [NVIDIA 2011]. This scheduling level could be improved by the use of hardware-based dynamic warp formation and scheduling as proposed by Fung et al. [Fung et al. 2007].

Recently, standard consortia like the Khronos group and hardware vendors strive towards *dynamic parallelism* on the GPU, which will be supported by NVIDIA’s Kepler architecture [NVIDIA 2012]. In the near future, it will be possible to launch kernels directly from the GPU. The main differences to Softshell are the granularity of these operations and the fact that Softshell is independent of the hardware model. According to NVIDIA, kernel launches from the GPU will behave just the same as kernel launches from the CPU

and show the same temporal overheads. Thus, it will also be necessary to launch parallel reduction methods to determine thread to data mappings and launch kernels of sufficient size. In contrast, Softshell provides very fine grained task control, allowing to start individual threads or groups of threads with only a comparatively small overhead. Another recently announced feature of future architectures are parallel command queues, which will allow the concurrent execution of multiple tasks. We expect Softshell to benefit from this feature, as it will allow Softshell to occupy only a part of the GPU, while the remaining cores can execute other tasks, such as OpenGL.

Work queues and mega-kernels allow scheduling on a higher level in software. Work queues allow to dynamically balance workloads between execution units [Cederman and Tsigas 2008]. This technique is most often paired with persistent thread approaches, which have recently been studied and summarized by Gupta et al. [Gupta et al. 2012]. A common application of persistent threads mega-kernels, which maintain control on the GPU while switching arbitrarily between tasks [Aila and Laine 2009]. Refinements to these techniques consider thread groups instead of single threads and allow for work to be added from the host [Chen et al. 2010; Tzeng et al. 2010; Chatterjee et al. 2011]. For previous hardware generations it has been found that a single shared queue performs worse than distributed queues [Chatterjee et al. 2011; Tzeng et al. 2010]. This finding may not hold for current GPUs that provide a globally shared cache [NVIDIA 2009]. In contrast to these approaches, we use a fixed size ring buffer as monolithic queue to enable sorting of this queue for dynamic priority scheduling. Furthermore, we avoid centralized locks and rather use a state variable per element to greatly increase access speed to the queue.

Multi-GPU and heterogeneous systems achieve scheduling on a higher level, as shown in RenderAnts [Zhou et al. 2009]. RenderAnts uses BSGP [Hou et al. 2008], which uses the CPU as a synchronization point between dependent execution steps. Although BSGP improves the process of mapping algorithms to the GPU, it essentially only changes the programming interface for the kernel-based processing model. In a similar manner, Sponge [Hormati et al. 2011] analyzes and optimizes the static graph of StreamIt programs to generate efficient GPU code for various hardware architectures. Besides RenderAnts, many approaches exist that distribute work over multiple GPUs, e.g., [Chen et al. 2010; Rossbach et al. 2011]. It is also possible to combine CPU and GPU execution [Frey and Ertl 2010; Agullo et al. 2010]. Another related approach is GRAMPS [Sugerman et al. 2009]. In this abstract rendering architecture, an algorithm is modeled as a directed graph of worker nodes connected by queues. The GRAMPS architecture is very general, but to date, it has only been tested on a CPU architecture [Sanchez et al. 2011]. While the basis of our processing model is similar to GRAMPS, our scheduler enables arbitrary dynamic priorities on a fine grained level. In this way, the scheduling strategy can be adapted to the target application.

Multi-Level Scheduling is used in NVIDIA OptiX [Parker et al. 2010], a framework for applications based on raytracing. OptiX is probably the most evolved approach in this direction. A fixed priority scheduler is used to lower the number of diverging branches on each multiprocessor. Load balancing between multiple GPUs is dealt with appropriately filling work queues on each GPU. Each execution unit draws elements from this global work queue to fill a local queue. While the OptiX approach is very powerful, it is restricted to applications in which the main workload comes from raytracing. Because it is closed-source, it is difficult to perform a detailed analysis or to build on top of their core results.

Real-time scheduling is necessary for applications that demand certain tasks to be completed by a given deadline. Although time plays an important role in interactive graphics applications, today's GPU programming is focused on throughput rather than time-awareness. If real-time constraints are considered, they are only tracked from the CPU, by choosing which command to send to the GPU next [Kato et al. 2011]. These approaches can work well for light-weight tasks but long running tasks can block the GPU and thus impair the application's real-time behavior. For an extensive overview of real-time scheduling, see the work by Sha *et al.* [Sha et al. 2004].

3 Processing model

The Softshell processing model targets architectures comprised of multiple SIMD units, such as those found on current NVIDIA and ATI GPUs [NVIDIA 2009; Adv 2011] or on Intel's proposed Larrabee architecture [Seiler et al. 2008]. The conventional stream processing model for GPU programming is based on the assumption 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. In contrast, the Softshell processing model 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. In contrast to GRAMPS [Sugerman et al. 2009], 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 1, 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*.

3.1 Softshell entities

Procedures define the execution steps in Softshell. In contrast to massive kernels, which are intended to occupy the entire GPU with thousands of threads, procedures are designed to be executed by small groups of coherent threads. Thus, a procedure is well suited for the execution on *one* SIMD unit. To fully occupy devices with several SIMD units, Softshell schedules multiple (different) procedures in parallel. Procedures are function like constructs that require input parameters. These input parameters are formed by workpackages. Depending on the input workpackage, the active number of threads for a procedure's execution may have to be adjusted by Softshell. To allow for more algorithmic control, the application programmer can demand a fixed number threads to be started for each procedure execution. Softshell expects procedures to run for a short period of time, compared to the execution time of an entire algorithm. Thus, the execution of a procedure will normally not be interrupted until it has been processed and scheduling decisions can be made 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. Using these definitions, Softshell manages work on different granularities. Workpackages can contain an arbitrary number of work items, which can be merged by Softshell to create workpackages of optimal size. It is possible to hide the number of work items within a workpackage from Softshell. This can be necessary if workpackages need to be executed with a fixed number of threads and must not be merged with other workpackages – for example optimized parallel reduction code, which works only with a defined number of work items. In this case the application programmer defines a static number of necessary threads with the definition of the *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. Queues belonging to procedures, which require a static number of executing threads, contain fixed sized workpackages. The queues of all other procedures hold workpackages of arbitrary size. These workpackages might be merged while they are waiting for their execution.

Capturing work on different granularities allows Softshell to automatically adjust a trade-off between efficiency and flexibility. During the execution of a procedure, entire workpackages or single work items can be generated and queued for arbitrary procedures. If a whole workpackage is generated, Softshell only needs to track a single object to provide work for a whole group of threads. If a procedure only produces a few work items, Softshell may combine them to one workpackage providing a sufficient number of work items for an entire group of threads.

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

Former GPU execution models focused on the execution of a single algorithm or tried to achieve the highest throughput for a single pipeline. But real world systems consist of multiple algorithms with severely different execution characteristics. In Softshell, events capture these individual algorithms, which may all fight for the available resources. In comparison to workpackages and procedures, which describe the bits and pieces of an algorithm, an event strings these pieces together and makes the execution of an algorithm traceable.

3.2 Example

To demonstrate the usage of the basic entities of Softshell, we will describe how an octree-based mesh simplification algorithm can be efficiently mapped to the Softshell processing model. Furthermore, we show some code pieces to ease the understanding of the processing model and to demonstrate that code for Softshell can be written in an intuitive manner (Listing 1). One classical mesh simplification approach, which is difficult to optimally map to the GPU, is *hierarchical dynamic simplification (HDS)* [Luebke and Erikson 1997].

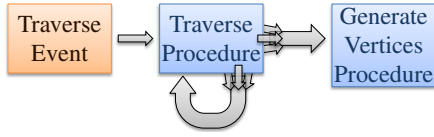


Figure 2: Octree-based mesh simplification [Luebke and Erikson 1997] on the GPU: An event (orange) generates an initial work item. The traverse procedure is executed for every octree node, issuing new work items for the traverse and generate vertices procedure, which are automatically combined by Softshell.

In a preprocessing step, an octree is built and then traversed during rendering. Each node of this octree can either be expanded or collapsed. Expanded nodes increase the local detail by adding triangles to the mesh, while collapsed nodes define vertex positions. The workload for each node may vary strongly, depending on the local complexity of the mesh. This heterogeneous workload can efficiently be handled by the Softshell work aggregation approach, as shown in Figure 2. In our implementation, a *traverse event* creates a single work item for the root node executed by the *traverse procedure*. This procedure creates work items for every child node, which are aggregated to workpackages and again assigned to the *traverse procedure*. Each time the traversal reaches a boundary node, a work item is issued for the *generate vertex* procedure. Triangles are generated directly in the *traverse procedure*.

Using the traditional kernel-based model, a large number of intermediate computations are necessary to assign octree nodes to threads. One would normally traverse the octree in a breadth-first fashion, applying a parallel prefix sum to determine the number of active nodes. After that, CPU and GPU must synchronize, so that the correct number of threads is launched for the next level. This strategy is used in current state-of-the-art parallel octree traversal solutions [Zhou et al. 2011]. Because this approach only parallelizes the execution of nodes on the same level, the GPU utilization may drop significantly for imbalanced octrees. Using Softshell, nodes from all levels can execute concurrently, dynamically balancing between breadth-first and depth-first traversal. Section 6.2.1 provides a quantitative comparison of the example outlined here to a traditional kernel-based implementation and shows that the Softshell approach is easier to write and executes more efficiently.

4 Three-tier scheduling model

Softshell employs a three-tier scheduling model, which is responsible for distributing work submitted to the system to execution units. The first tier is responsible for work distribution between multiple GPUs. The second tier is concerned with workpackage priorities and with assigning workpackages to free execution units. The third tier is active during the execution of a single workpackage, addressing diverging threads, pausing, canceling and restarting the active workpackage. The Softshell model only defines the duties of the different scheduling tiers, not their implementation. Nevertheless, we propose a reference implementation in Section 5.

First-tier scheduling is activated whenever a new event occurs. To influence the execution of concurrently active events, events can be assigned priorities. Depending on the state of all GPUs, the first-tier scheduler sends an event to the most suitable GPU. For this purpose, it considers the event’s priority, the event’s execution history, and the current load on each GPU. The first-tier scheduler tries to initiate the event execution in such a way that (1) high-priority events are completed first, (2) all GPUs are well utilized, and (3) the execution of all events is completed as soon as possible.

Listing 1: The C++ Softshell implementation on top of CUDA is easy to use. For mesh simplification [Luebke and Erikson 1997], only two procedures are required, as shown in Figure 2. Each Procedure is created by implementing the *execute* method. Work items for this example correspond to a node identifier only (line 1). Workpackages consisting of multiple work items are defined using templates (line 2). New work items are handed to Softshell using the *issueWorkItem* command (line 19 and 22) and telling the system for which procedure the work item should be queued (template parameter). The *TraverseEvent* issues a single workpackage for the *TraverseProcedure*.

```

1  typedef uint NodeWorkItem;
2  typedef CombWorkpackage<NodeWorkItem> NodeWp;
3
4  class TraverseProcedure : public Procedure
5  {
6  public:
7      __device__ static void
8      execute(Workpackage* workpackage)
9      {
10         //extract the work item from the package
11         NodeWp* myWp = (NodeWp*)(workpackage);
12         NodeWorkItem myItem = myWp->getMyWorkItem();
13
14         Node* node = getOctreeNode(myItem);
15         for(uint i=0; i < node->numChildren(); ++i)
16         {
17             //check if this node should be expanded
18             if( carryOnTraversal( node->child(i) ) )
19                 issueWorkItem<TraverseProcedure>(
20                     NodeWorkItem( node->child(i) ) );
21             else
22                 issueWorkItem<GenVerticesProcedure>(
23                     NodeWorkItem( node->child(i) ) );
24         }
25         writeTriangleOutput( node );
26     }
27 };
28
29 class TEvent
30 {
31     __device__ static void occurred()
32     {
33         NodeWorkItem root = 0;
34         issueWorkpackage<TraverseProcedure>(
35             ( NodeWp(&root,1) ) );
36     }
37 };
  
```

Second-tier scheduling is the core of Softshell. In contrast to previous approaches that filled up free processing units with the next best entity, Softshell introduces a more sophisticated logic on this layer: Events and workpackage are considered to have individual priorities. Softshell tries to schedule the most important workpackages first, while allowing priorities to change at any point in time. Additionally, every application can define its own per workpackage priority function. This priority function is queried by the second-tier scheduler regularly to schedule the highest priority workpackages first. Obviously, the execution order is independent from the order in which workpackages are issued.

Third-tier scheduling is responsible for longer running procedures. It is active during the execution of a single workpackage and regularly queries the current execution state. The second tier can demand the current execution to stop, if higher priority workpackages became available or the associated event got canceled. If the

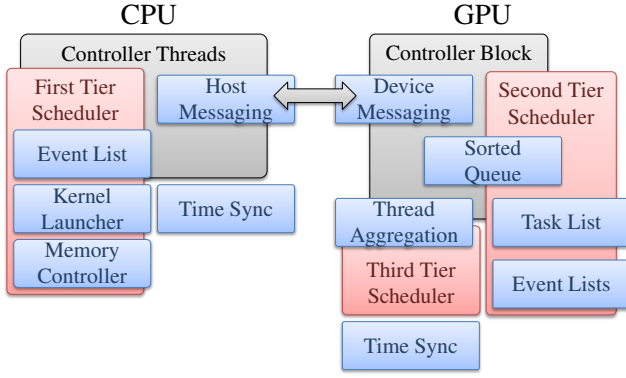


Figure 3: Our implementation of the Softshell processing model is split between CPU and GPU. The CPU part implements the first-tier scheduler and keeps the GPUs active. The second- and third-tier schedulers are realized directly on the GPUs partially relying on recurring execution within the controller block.

workpackage’s execution should continue at a later time, the third tier saves all thread contexts and issues the workpackage together with the saved state for later execution. For complex procedures it can be beneficial to regroup diverging threads according to their execution paths. The third-tier scheduler can achieve this by either locally regrouping threads, or by combining threads on a global level. To achieve global regrouping, the third tier stops the executing workpackage and inserts the threads into a global thread aggregation structure, which works similar to work item combination. If threads within this structure cannot be regrouped for a certain time, they are re-issued regardless of their coherency.

5 Implementation

To demonstrate the utility of Softshell, we describe an implementation built as a mega-kernel [Aila and Laine 2009] on top of NVIDIA CUDA. To implement Softshell, we build all its functionality in software, replacing CUDA’s kernel-based interface by a C++ interface. Note that the described implementation can be made more efficient in the future if implemented as a driver or partially in hardware. In the following section, we focus on the selected key aspects of our implementation. To provide the interested reader with all implementation details, we offer our implementation as open source, to be downloaded at <http://www.icg.tugraz.at/project/mvp>.

Our implementation consists of multiple components interacting with each other, as depicted in Figure 3. The CPU part of the Softshell implementation forwards input from the application to GPUs and keeps the internal states synchronized between multiple GPU devices. Due to the way we synchronize with the kernel launches, one CPU thread per GPU is responsible for launching kernels to keep the GPUs occupied.

The GPU segment of the implementation is concerned with collecting information about the execution and managing work items and workpackages. On the GPU we distinguish between *worker blocks* and one *controller block*. The worker blocks are CUDA thread blocks, which execute the procedures for all workpackages. The controller block is a CUDA thread block whose only responsibility is to carry out recurring maintenance procedures, including the communication with the CPU and different kinds of scheduling procedures. This setup allows us to react on new events and changing priorities, while the GPU is under full load.

Our implementation of the **first-tier scheduler** is straight forward. Whenever an event occurs, the first-tier scheduler determines the GPU with the lowest load and forwards the event to this GPU. To support this decision-making, each GPU regularly informs the CPU about the estimated time t_{exp} until all events assigned to it will be processed.

To track active and previous events, we use an *event list*. This list supports querying the event status and reaction on the event’s completion. Each event launch can be supplied with a set of parameters, which are made available on the GPU executing the event. As events can occur during the execution of the mega-kernel, the parameter transfer must happen in parallel with kernel execution. To achieve this, we use page-locked host memory, which is mapped into the address space of the GPU. Whenever a new set of parameters needs to be passed to the GPU, the *memory controller* searches for a free region of sufficient size within the mapped memory. The first fit is then used for the parameter transfer and marked as used until the event execution has finished.

One of the most important features of our implementation is the **messaging component** that enables a bi-directional communication between the CPU and GPU while the mega-kernel is active. It is needed to start the execution of events, alter event priorities, report information about event execution, and synchronize the time between the GPU and CPU. Again, we use mapped page-locked host memory to establish this communication. We provide two message queues implemented as ring buffers, one for each direction. On the CPU side, we use a mutex to lock the queue when multiple threads want to send messages to the GPU. On the GPU, any running thread can send messages to the CPU, which requires atomic operations to avoid corruption.

As there is no mechanism to automatically react to changes in mapped memory, we poll the queue fill levels to determine changes. For each GPU, there is one CPU thread responsible for polling the message queue, while on the GPU, the controller block checks the queue state. Atomic operations are not supported between GPU and CPU. Therefore, the receiver does not alter the queue state directly when reading messages from a queue. Instead, it returns a message requesting the sender to update the queue state.

Our implementation of the **second-tier scheduler** assigns workpackages to worker blocks. Before the second tier becomes active, the initial workpackages for an event must be created. This action is performed by the controller block, when it receives a message about a new event occurrence. Because workpackages are created using dynamic memory allocation, we use our own memory allocator, ScatterAlloc [Steinberger et al. 2012], to speed up workpackage creation. To handle the assignment of ready workpackages to worker blocks, we establish one monolithic queue. Every entry in this queue holds a reference to the corresponding workpackage and procedure. Worker blocks pull workpackages from the front of the queue, while new workpackages are inserted at the back. The second tier only becomes active after the execution of a procedure has finished, thus we have to rely on procedures not to occupy worker blocks for too long, to be able to react on newly available high priority workpackages.

On GPUs, the high overhead of synchronization primitives prohibits the use of complex data structures like sorted linked lists or heaps. Therefore, we employ a simple ring-buffer-based queue but use the controller block to periodically sort the queue according to the workpackage priorities. To enable concurrent insertion and deletion and to avoid a single mutex for locking the entire queue, we use an atomically operated counter for the front and back of the queue. Every entry has a state flag indicating if it is ready for processing. This flag avoids pulling workpackages from the queue,

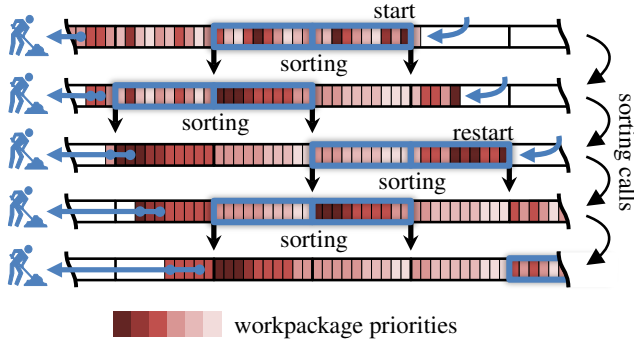


Figure 4: We support insertion and removal of workpackages while the queue is iteratively sorted. Each iteration, two segments (blue frames) are locked and sorted. For the next iteration, the window is moved one segment to the front, taking high-priority workpackages along. If there is not enough time for sorting the front-most segment, the algorithm is restarted at the back.

which are currently being inserted by another thread. To further speed up the insertion process, the queue counters are altered by a single thread only, if multiple threads of a block enqueue data.

Because the queue-sorting algorithm runs periodically, the main objective of the algorithm is to move high-priority workpackages to the front of the queue. The order of workpackages at the back is not relevant as long as these workpackages are not removed from the queue before the sorting algorithm is restarted. Consider the example in Figure 4: To avoid occupying the execution of the controller block for too long, we partition the queue into fixed size segments. Each invocation of the sorting algorithm takes two adjacent segments and sorts their elements, advancing one segment to the front each time; the number of high priority workpackages moved to the front thus equals the segment size. Worker blocks pop elements from the queue while the sorting is run. To avoid data corruption, we mark segments being sorted to prevent their removal (blue frames in Figure 4). Furthermore, we use meta-data collected for every procedure to estimate the time until a segment becomes the front of the queue. If the estimated time frame is too short to execute the sorting algorithm, we refrain from sorting this segment, as stalling the system would be much worse than executing a few lower-priority workpackages. Instead, we restart the sorting process at the back, again moving workpackages of higher priority to the front. To sort the segments in parallel, we use bitonic sort [Batcher 1968], which is well-suited for a small number of elements.

To store meta information of all procedures and call their execute method, we keep a *procedure list*. This meta data includes minimal, maximal, and average execution time and the average number of workpackages issued by the procedure. This setup enables the estimation of procedure and event execution times. Similarly, we store information about the currently active and previously executed events. Additionally to the average execution time and average number of executed workpackage per event, we also keep track of the number of currently queued workpackages for this event. If this active workpackage counter drops to zero, the event has been fully processed and the CPU can be informed.

Because of the lack of GPU hardware support for preemptive multitasking, our current implementation of the **third-tier scheduler** uses a cooperative approach. The application programmer specifies scheduling points, at which the third-tier scheduler is invoked and determines whether execution should continue and whether the thread execution paths are still coherent. Using code analysis for inserting scheduling points would also be possible. However, for

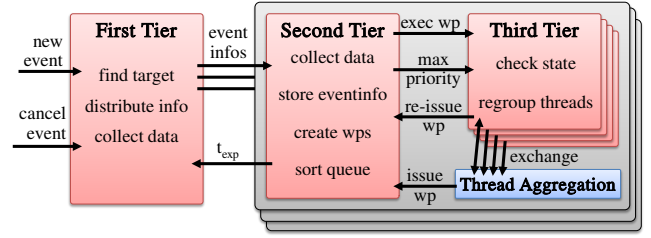


Figure 5: Three-tier scheduler: the first tier is activated when an event occurs and forwards it for execution to the most appropriate device. The second tier creates the initial workpackages (wp) for the event and inserts them into the priority queue. The highest-priority workpackages are forwarded to the third tier for execution by a worker block. The third tier periodically checks the current execution state of all active threads. If the workpackage's priority is too low compared to the front of the queue, it stops execution and re-issues the workpackage. If the threads' execution paths strongly diverge, the third tier regroups the threads.

our experiments we decided to let the developer define scheduling points manually and to leave the full control on the user-side. To determine the variables that need to be stored, we use a similar approach to Optix [Parker et al. 2010] and let the application programmer define the threads payload. In addition to all thread payloads, we also store an identifier for the current execution location. If a block receives a workpackage that has previously been executed, it can restore the execution state and continue the execution at the point where it has been suspended. This is similar to kernel relaunching as described by Hou *et al.* [Hou et al. 2009].

To monitor the thread execution paths, every thread publishes the execution path it will take. If a vote reveals that the thread divergence is above a threshold, the scheduler either stops the workpackage and submits all threads to the global thread aggregation, or exchanges a few threads with threads already in the thread aggregation. The thread aggregation itself keeps a list of stored threads for all execution points of all procedures. The controller block periodically scans through this list. If there are enough threads stored to form an entirely coherent workpackage or if the threads have been in the list for too long, the controller block merges the threads to a workpackage and issues it for execution. In the same way, the thread aggregation also concatenates work items to workpackages, as defined in Section 3.1.

Note that Softshell supports thread synchronization and local shared memory within procedures. The only limitation is that shared memory must not be declared as *extern*. Thread synchronization is implemented using *inline ptx code*, which gives us access to 16 barriers that we use for the different subgroups of threads executing individual procedures.

In a system with real-time properties, **synchronized time** is an important feature. On CUDA devices, a counter, which keeps pace with the device clock rate is available on each multiprocessor. To synchronize these counters during system initialization, we launch a kernel to write each counter's state to an array in global shared memory. With the messaging component we periodically synchronize this array and the time on the CPU using Poincaré-Einstein synchronization [Poincaré 1913]. Unfortunately, the device clock rate is not constant, but varies with the GPU's load. We currently compensate for that by estimating the clock rate between each two synchronization messages. Using the time synchronization component, real-time scheduling strategies, such as earliest deadline first scheduling [Liu and Layland 1973], can be implemented.

The **application programming interface** (API) essentially exposes the following four classes to the application programmer: a *procedure interface*, a *workpackage interface*, an *event class*, and the *scheduler object*, which is available on the CPU and GPU. To implement a new procedure or to provide a new type of workpackage, one may implement the respective interfaces and register their implementation with the scheduler object. Listing 1 shows the simple structures of Softshell for GPU code. Listing 2 shows how the scheduler is controlled from the CPU. The Softshell API provides an efficient control mechanism for multi level parallel execution. With a single command, work for a single thread, an entire group of threads, or multiple groups can be created. Thus, complex algorithms can easily be mapped for GPU execution. Softshell supports a fully customized priority evaluation, whereby arbitrary scheduling strategies can be generated by implementing a single function.

Listing 2: Host side API example: A few function calls are sufficient to control the Softshell processing model. Upon initialization, a custom priority evaluation can be specified via template arguments. In this case, the scheduler is configured to execute the *TraverseProcedure* before all others. Subsequently, a custom event is created and triggered once, before the CPU waits for its completion. The associated GPU code is depicted in Listing 1.

```

1 struct TraversePriority
2 {
3     __device__ static float
4     priority(Procedure* proc, Workpackage* wp)
5     {
6         if (procedureEqual < TraverseProcedure > (*proc))
7             return 1.0f;
8         else
9             return 0.0f;
10    }
11 };
12
13 void meshsimplification(DeviceList devices)
14 {
15     Scheduler scheduler;
16     scheduler.init<TraversePriority>(devices);
17     auto myEvent = scheduler.createEvent<TEvent>();
18
19     myEvent.trigger();
20     myEvent.join();
21 }

```

6 Experimental results

To understand the characteristics of our Softshell implementation, we measured the performance of the key scheduling components, which are presented in the first half of this section. The second half analyzes the advantages and disadvantages of the Softshell processing model for selected computer graphics algorithms. All measurements were run on the same machine featuring an Intel i7 2.80 GHz Quad Core CPU and an NVIDIA Quadro 6000 graphics card.

6.1 Synthetic tests

The performance of our Softshell implementation is strongly related to the performance of our work queue. The overhead associated with issuing a workpackage equals the time it takes to enqueue a workpackage. The time for starting the execution of a workpackage is made up by the dequeue time. How well the execution order matches the workpackage priorities depends on the system’s ability to sort the queue fast enough. Therefore, we focus our synthetic tests on the different facets of our monolithic queue.

The **queue performance** was assessed by measuring the enqueue and dequeue time for increasing thread counts and comparing our method to the methods described by Tzeng *et al.* [Tzeng *et al.* 2010]. The results of this comparison, as shown in Figure 6, demonstrate that our monolithic queue approach is magnitudes faster than the monolithic queue used by Tzeng *et al.*. The performance difference between the two monolithic queuing approaches can be explained by the fact that our queue is lock-free and uses localized state flags only, as outlined in Section 5. This strategy results in a very small linear latency increase for each additional thread accessing the queue concurrently: The enqueue time $t_{enq} \approx 2.74\mu s + n \cdot 0.0025\mu s$, while the dequeue time t_{deq} increases with the number of active thread blocks b according to $t_{deq} \approx 2.13\mu s + b \cdot 0.003\mu s$. Additionally, our queue also outperforms all distributed queuing strategies with the only exception being the localized distributed head, which is managed in shared memory. As shared memory is limited and this part of the queue is not shared between different blocks, its size must be kept small not to stress shared memory and allow the distribution of work between different blocks. Thus, queue accesses will often be forwarded to the queue tail, which resides in global memory. Consequently, our monolithic queue will either perform comparable to these distributed queuing techniques or even faster. As our optimized monolithic queue is shared among all workers, we achieve a very good work distribution [Tzeng *et al.* 2010] and can enable fine grained priority queue management.

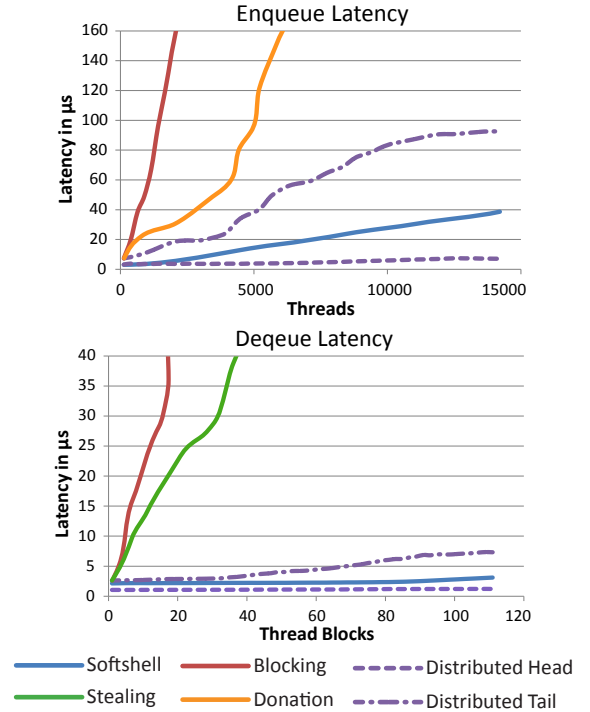


Figure 6: Enqueue and dequeue performance of our monolithic queue and the queuing strategies described by Tzeng *et al.* [Tzeng *et al.* 2010]. The monolithic queue used in our Softshell implementation clearly outperforms other monolithic (blocking) queuing approaches and outperforms most of the techniques used in distributed queuing (distributed tail, donation and stealing). The only access which is faster than our approach is accessing the head of the distributed queue, which is locally kept in shared memory.

An initial event launch involves an additional operation besides queuing workpackages, which is sending a message from the CPU to the GPU. Thus, the overhead for an event launch is increased by the latency of the messaging component. To quantify this latency, we have measured the round trip time of an 8 byte message, which is on average $20.8\mu s \pm 4.6\mu s$. Hence, we estimate the latency of a one-way message to be approximately $10\mu s$. If the mega-kernel is not active, we would introduce this overhead in addition to the overhead for starting the mega-kernel (approximately $20\mu s$). To avoid the additional overhead, we provide a second version of our mega-kernel, which takes an initial message as argument for the kernel launch. If the mega-kernel is not active when an event is triggered, this version is called to initiate the creation of workpackages on the GPU as quickly as possible. In case the kernel is already running, it is most likely that the latency for sending a message will be hidden by the execution already taking place on the device. Nevertheless, this latency can still form a delay for the execution of high priority events if the GPU is executing low priority events only. However, the overhead for sending an execution message to the GPU and receiving a notification about the event being processed ($\approx 21\mu s$) is significantly lower than starting a CUDA kernel and waiting for it to be executed ($\approx 47\mu s$).

The scheduling performance itself was measured by generating a fixed number of workpackages with uniformly distributed random priorities. This test case resembles a setup in which an arbitrary number of events are active and all of the submitted workpackages have arbitrary priorities. To measure the scheduling performance in a single value, we computed the scheduling accuracy, which captures the number of workpackages which were executed according to the requested priorities. For an accuracy of 100%, all workpackages must be executed exactly in the right order. An accuracy of 50% is reached, if in half of the cases a lower priority workpackage is executed before a higher priority workpackage. For our test example with uniformly distributed priorities, this value forms the lower bound, because it is reached if no scheduling is applied. As shown in Figure 7, the more threads are used for scheduling operations, the better the execution order fits the requested priorities. The graphs also show that a great many (more than 20000) workpackages with a short execution time ($100\mu s$) can only be scheduled as desired, if a reasonable large number of threads are designated to the controller block.

6.2 Example applications

In this section, we show that our method accelerates selected computer graphics techniques significantly and opens new possibilities for algorithmic advancement. For all comparisons between a CUDA and a Softshell implementation, we do not alter the algorithms themselves. We only replace the code needed for issuing kernels by the Softshell interface implementations.

6.2.1 View-dependent mesh simplification

For our first comparison, we have implemented HDS [Luebke and Erikson 1997] with a kernel launch behavior similar to state-of-the-art approaches [Zhou et al. 2011] as already described in Section 3.2. Our baseline CUDA implementation manages the following three data structures with the help of the highly optimized CUDPP library: a set of active octree nodes, a set of boundary nodes, and a list of active triangles. For each level, two custom kernels (for analyzing and inserting nodes) and two parallel scans (for active nodes and boundary nodes) are called. An additional scan is run to layout vertices in memory before two kernels update the vertex positions and emit triangles.

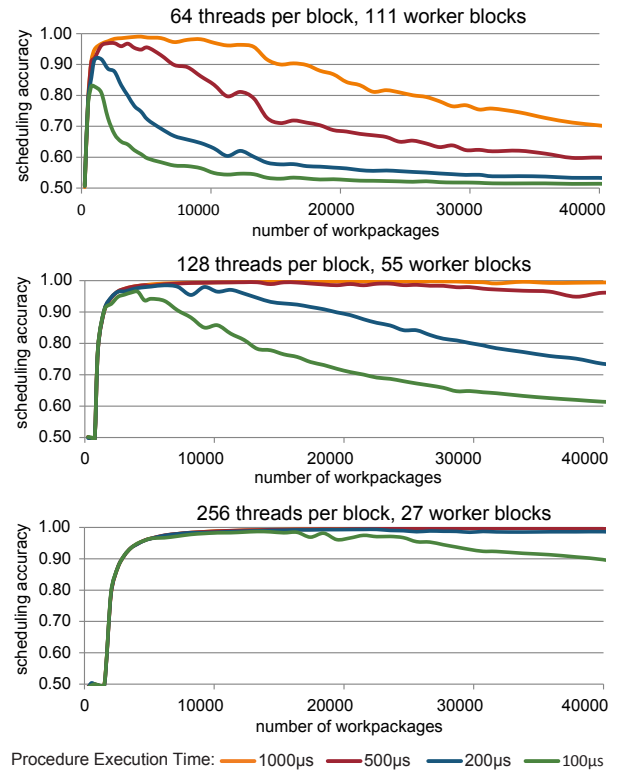


Figure 7: Scheduling accuracy for an increasing number of workpackages with random priorities; worker blocks and controller (sorter) contain an equal number of threads; the queue segment size equals the block size. The more threads are used for sorting the queue, the better the scheduling performance is. For short procedure execution times, the scheduler has problems to stick to the priorities. If there are only a few workpackages in the queue, the scheduler cannot acquire the first segment for sorting and thus the accuracy drops to 50%.

The Softshell implementation manages all data dynamically during execution. Hence, there is no need for CPU interaction until the mesh is completely constructed. Work items are automatically grouped by the work aggregation and the execution order is controlled by Softshell, reducing the necessary programming effort in comparison to the kernel-based implementation. This fact is also captured in the lines of code required for programming the two models: The CUDA implementations has 213 lines of code, while the Softshell implementation consists of 184 lines.

For imbalanced octrees, the Softshell implementation is more than two times faster than the kernel-based implementation, as shown in Table 1. This performance gain is mainly due to the ability to leave the control at the GPU and the ability to draw parallelism from the breadth and depth of the octree. Because the *Traverse procedure* creates new workpackages and the *Generate Triangle procedure* does not, the number of available workpackages (and also the utilization) is increased by prioritizing workpackages for the *Traverse procedure*.

6.2.2 Path tracing

When implementing a path tracer [Kajiya 1986] on the GPU, the greatly varying number of bounces of secondary rays prohibit coherent execution. Using the Softshell processing model, this problem can automatically be addressed by the third-tier scheduler.

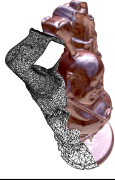

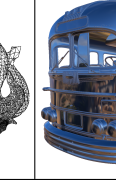
	Happy Buddha			Dragon			Bus		
									
	lev.3	lev.6	lev.9	dist	mid	close	dist	mid	close
vert.	812	56k	796k	3.8k	143k	286k	130k	443k	1.6M
nodes	65	8.3k	362k	501	27k	66k	15k	98k	227k
CUDA	1.77	3.28	9.66	2.75	5.23	7.63	7.19	10.9	36.8
Softshell	0.87	1.66	9.15	0.85	2.48	3.47	3.11	5.42	24.9

Table 1: Octree-based mesh simplification: The Happy Buddha dataset is constructed to a fixed octree level. The other two models are constructed based on the current view (from distant to close), for which the decreasing detail in the back leads to inhomogeneous traversal depths. The more nodes are used in the construction, the more vertices (vert) are created and the higher is the data parallelism per level. The construction times for both techniques are given in milliseconds. For balanced octrees with many nodes (Happy Buddha level 9), the kernel-based CUDA implementation is as fast as the Softshell implementation. In all other cases, Softshell is superior because it leads to a better hardware utilization.

Our sample implementation consists of a single procedure that implements a backward monte-carlo path tracer. For each of the 800×800 pixels, a single workpackage is created. Every thread is mapped to one of the 512 rays randomly shot through each pixel. Paths are traced until their contribution falls below 0.1%. Bounces are modeled in a single loop, with the third-tier scheduler being invoked every 10^{th} iteration. As test scene, we used a Cornell box with a varying number of spheres influencing the number of diverging threads. The results provided in Table 2 show that in this special example the use of dynamic thread regrouping can increase performance to a certain extent. Although the probability of a thread to be stalled during an iteration is 50% – 76% and the execution time for this example is about a minute, the performance gain achieved by thread regrouping is only 4 – 15%. This confirms previous findings that it is hardly possible to gain performance using dynamic thread regrouping for interactive applications [Parker et al. 2010]. Still, this example shows that for long running algorithms with a high rate of thread divergence, dynamic thread regrouping can boost performance.

While the previous example required considerably fewer lines of code for the Softshell implementation, we encounter the reverse situation in this case. The Softshell implementation consists of 95 lines, while the CUDA implementation needs 54 lines. The reason for this change is the fact that this application consists of a single kernel only, which can be expressed very efficiently in CUDA C, while Softshell requires some boilerplate code to be written to individually model all scheduling entities.

This example can be slightly altered to demonstrate the utility of the second tier scheduler. Instead of creating one workpackage for each individual pixel, we create one workpackage for each patch of 4×4 pixels and initially shoot only four rays per pixel into the scene, leading to 64 threads per workpackage. After tracing all 64 paths, we compute the color variance among the paths and resubmit the workpackage, setting its priority proportional to the computed variance. In this way, we prioritize areas for which we are uncertain about the estimated pixel colors. When sending more rays into the scene, we assume that the color converges to its real value and thus reduce the workpackage priority every time the workpackage is resubmitted. In this setup, the second tier scheduler deals with

div.	0.52	0.63	0.76
CUDA	50.4 <i>s</i>	57.4 <i>s</i>	66.8 <i>s</i>
no T3	52.3 <i>s</i>	59.4 <i>s</i>	68.9 <i>s</i>
ours	50.2 <i>s</i>	54.8 <i>s</i>	59.6 <i>s</i>

Table 2: Path tracing with varying number of objects. The more spheres are in the scene, the higher the probability that a thread has to wait for others to finish (div). Using Softshell with no third-tier scheduler (no T3) demonstrates Softshell’s overhead for workpackage management when compared to CUDA. As the third-tier scheduler is activated (ours), diverging threads are regrouped, leading to performance increases of up to 15%.

fully dynamic workpackage priorities. Because uncertain image areas are executed first, the rendered image converges to the ground truth faster than a system, which iteratively shoots the same number of rays through all pixels, as shown in Figure 8.

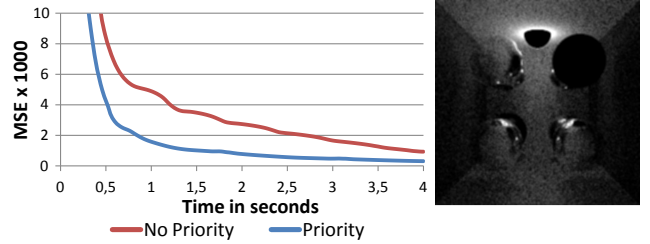


Figure 8: Priority-based path tracing in Softshell for test scene three: Adding rays to uncertain image regions first (blue), creates high quality images with lower mean squared error in comparison to the ground truth more quickly than adding rays to pixels uniformly (red). This behavior is implemented in Softshell by dynamically adjusting the priorities of image patches according to the color-variance within the traced paths. The image on the right visualizes the number of traced rays (cp. result image in Table 2). For white areas, many rays have been shot into the scene, while black areas have been sampled sparsely. Note that uniform regions, like the light source and the black sphere have hardly been sampled.

6.2.3 Image-based visual hull rendering

The image-based visual hull (IBVH) algorithm estimates images at novel viewpoints directly from segmented camera images [Matusik et al. 2000]. It is especially useful in interactive systems that must produce output images with little delay. Without sacrificing image quality, computing the IBVH can be constrained to those image areas in which the underlying geometry has changed [Hauswiesner et al. 2011]. After detecting changes in the scene geometry, only those image areas with a change magnitude above a threshold are recomputed and the remaining areas are warped from the previous image. This threshold controls the output quality directly, but it only indirectly affects the execution time.

For interactive systems, it is desirable to directly constrain runtime and to maximize the image quality within the available time-frame. Using Softshell, a system implementing this behavior can easily be realized. Workpackages are created for the entire image. Their priority is set to the average change magnitude of their associated

region, so that image areas with substantial changes will be processed first. During execution of each workpackage, the remaining time and remaining number of workpackages are queried. Based on the known, constant execution time of image warping, we can decide whether to run the IBVH algorithm or image warping. In this way, quality can dynamically be traded for execution time to guarantee the frame rate, as shown in Figure 9.

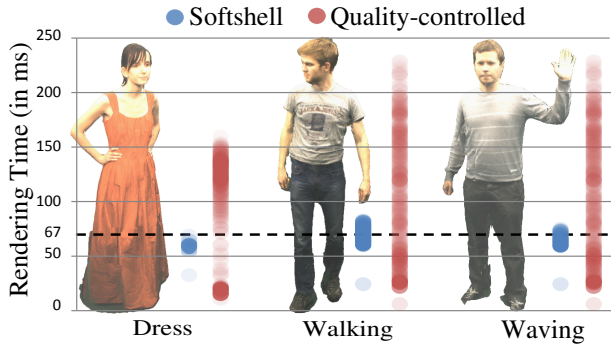


Figure 9: Image-based visual hull (IBVH) rendering with a target time-constraint of 67ms to match the camera frame rate. In this system, we use a low-latency image warping approach for image areas of little change and a full IBVH construction for areas of high change. The tested algorithms differ in the heuristics used to choose between IBVH and image warping. A fixed threshold determined prior to a kernel-launch does not lead to the desired result (red dots). The Softshell implementation makes this decision dynamically, based on the time remaining for the current frame. Work of higher-priority is scheduled first, generating the highest possible quality in the specified time (blue dots).

6.2.4 GPU X3D parser

As the demand for high-quality graphics rises, art assets rapidly grow larger. Parsing a file containing a large model can take a significant amount of time. Using the GPU for parsing model files can speed up this process. However, the X3D file format is composed of many independent constructs, thus, writing efficient parallel GPU code for parsing is no trivial task. Using the BSGP programming model, such a system can be implemented with less programming effort [Hou et al. 2008]. For the execution, still about 80 kernel-functions are automatically created.

For comparison, we provide our own X3D parser in Softshell, which works similar to the BSGP X3D parser. We divide the parsing into two events: At first, we use six procedures (two of them implement sorting and the scan algorithm) to generate a skeleton of the XML-tree. The scan and sort algorithms are called multiple times, for data sizes that individually could not fully occupy the GPU. Softshell schedules these algorithms concurrently whenever data or parts of the data become available. Due to its kernel-based structure, the BSGP parser executes these steps sequentially, losing performance in comparison to our implementation.

In a second step, we parse the XML-tree using one procedure per supported X3D-tag. In this way, the parser can easily be extended to support new X3D-tags. The execution starts at the root node and a new workpackage is created for each encountered node. Thus, groups of threads work on the individual nodes in parallel, while the thread count is dynamically adjusted to the node type. Depending on the node type, new workpackages for child nodes are generated and/or output data is written either to the vertex buffer or a state buffer, which is then used during rendering to adjust the OpenGL

state machine. Softshell schedules workpackages for the different X3D-tag procedures concurrently and therefore increases the GPU utilization. In this way, our X3D parser can launch a higher number of coherently executing thread groups than the BSGP parser. Additionally, the control flow remains entirely on the GPU until the model is ready to be rendered, and a costly back and forth between GPU and CPU is avoided. For a complex scene (Figure 10), parsing takes BSGP 71.25ms, while the Softshell implementation is nearly twice as fast with 36.56ms.



Figure 10: GPU-based X3D parsing: The 13MB test scene contains 2300 nodes of which 1400 are shape defining.

7 Conclusion

We have presented Softshell, a novel processing model for devices composed of multiple SIMD units, such as GPUs. Our processing model enables a more sophisticated control of parallel execution and provides an intuitive API. Thus, it becomes possible to efficiently map algorithms with a relatively low degree of local parallelism for the execution on massively parallel architectures. Softshell’s key features include the ability to generate arbitrary amounts of work directly on the executing device, dynamically adjust priorities for small portions of work, and the ability to control work that has already been submitted to the GPU. Our tests demonstrate that these features can improve the performance of computer graphics applications in particular and potentially open up new possibilities in the field of GPU computing in general.

Acknowledgments

We thank Qiming Hou, Zhejiang University, for his implementation of the BSGP X3D parser and Martin Kenzel for the car models and textures. This research was funded by the Austrian Science Fund (FWF): P23329.

References

- ADVANCED MICRO DEVICES. 2011. *AMD Accelerated Parallel Processing OpenCL - Programming Guide*.
- AGULLO, E., AUGONNET, C., DONGARRA, J., LTAIEF, H., NAMYST, R., THIBAUT, S., AND TOMOV, S. 2010. Faster, cheaper, better – a hybridization methodology to develop linear algebra software for gpus. In *GPU Computing Gems*, vol. 2. Morgan Kaufmann, Sept.
- AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics*, ACM, HPG ’09, 145–149.
- BATCHER, K. E. 1968. Sorting networks and their applications. In *Proc. Spring Joint Computer Conference*, ACM, AFIPS ’68, 307–314.

- BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.* 23, 3 (Aug.), 777–786.
- CEDERMAN, D., AND TSIGAS, P. 2008. On dynamic load balancing on graphics processors. In *Proc. ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, GH '08, 57–64.
- CHATTERJEE, S., GROSSMAN, M., SBIRLEA, A., AND SARKAR, V. 2011. Dynamic task parallelism with a GPU work-stealing runtime system. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, LCPC '11.
- CHEN, L., VILLA, O., KRISHNAMOORTHY, S., AND GAO, G. 2010. Dynamic load balancing on single- and multi-GPU systems. In *Proc. Parallel Distributed Processing*, IEEE, IPDPS, 1–12.
- FREY, S., AND ERTL, T. 2010. PaTraCo: A Framework Enabling the Transparent and Efficient Programming of Heterogeneous Compute Networks. In *Proc. Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV10, 131–140.
- FUNG, W. W. L., SHAM, I., YUAN, G., AND AAMODT, T. M. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. IEEE/ACM International Symposium on Microarchitecture*, IEEE, MICRO 40, 407–420.
- GUPTA, K., STUART, J. A., AND OWENS, J. D. 2012. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, 14.
- HAUSWIESNER, S., STRAKA, M., AND REITMAYR, G. 2011. Coherent image-based rendering of real-world objects. In *Proc. Symposium on Interactive 3D Graphics and Games*, ACM, I3D '11, 183–190.
- HORMATI, A. H., SAMADI, M., WOH, M., MUDGE, T., AND MAHLKE, S. 2011. Sponge: portable stream programming on graphics engines. In *Proc. Architectural support for programming languages and operating systems*, ACM, ASPLOS '11, 381–392.
- HOU, Q., ZHOU, K., AND GUO, B. 2008. BSGP: bulk-synchronous GPU programming. *ACM Trans. Graph.* 27, 3 (Aug.), 19:1–19:12.
- HOU, Q., ZHOU, K., AND GUO, B. 2009. Debugging GPU stream programs through automatic dataflow recording and visualization. *ACM Trans. Graph.* 28, 5 (Dec.), 153:1–153:11.
- KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph.* 20, 4 (Aug.), 143–150.
- KATO, S., LAKSHMANAN, K., RAJKUMAR, R., AND ISHIKAWA, Y. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX annual technical conference*, USENIX Association, USENIXATC '11, 2–2.
- KHRONOS. 2008. *OpenCL The standard for heterogeneous parallel programming*. Khronos OpenCL Working Group.
- LIU, C. L., AND LAYLAND, J. W. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20 (January), 46–61.
- LUEBKE, D., AND ERIKSON, C. 1997. View-dependent simplification of arbitrary polygonal environments. In *Proc. SIGGRAPH '97*, ACM, 199–208.
- MATUSIK, W., BUEHLER, C., RASKAR, R., GORTLER, S. J., AND MCMILLAN, L. 2000. Image-based visual hulls. In *Proc. SIGGRAPH '00*, ACM, SIGGRAPH '00, 369–374.
- MCCOOL, M. D., QIN, Z., AND POPA, T. S. 2002. Shader metaprogramming. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWS '02, 57–68.
- NVIDIA. 2009. NVIDIA's next generation CUDA compute architecture: Fermi. White paper. Available online.
- NVIDIA. 2011. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*. NVIDIA.
- NVIDIA. 2012. *NVIDIAs NextGeneration CUDA Compute Architecture: Kepler TM GK110*, May.
- PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: a general purpose ray tracing engine. *ACM Trans. Graph.* 29, 4 (July), 66:1–66:13.
- POINCARÉ, H. 1913. *The Measure of Time*. New York: Science Press.
- ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. 2011. PTask: operating system abstractions to manage gpus as compute devices. In *Proc. ACM Symposium on Operating Systems Principles*, ACM, SOSP '11, 233–248.
- SANCHEZ, D., LO, D., YOO, R. M., SUGERMAN, J., AND KOZYRAKIS, C. 2011. Dynamic fine-grain scheduling of pipeline parallelism. In *Proc. International Conference on Parallel Architectures and Compilation Techniques*, IEEE, PACT '11, 22–32.
- SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.* 27, 3 (Aug.), 18:1–18:15.
- SHA, L., ABDELZAHER, T., ARZEN, K.-E., CERVIN, A., BAKER, T., BURNS, A., BUTTAZZO, G., CACCAMO, M., LEHOCZKY, J., AND MOK, A. K. 2004. Real time scheduling theory: A historical perspective. *Real-Time Syst.* 28, 2, 101–155.
- STEINBERGER, M., KENZEL, M., KAINZ, B., AND SCHMALSTIEG, D. 2012. ScatterAlloc: Massively Parallel Dynamic Memory Allocation for the GPU. In *Proceedings of Innovative Parallel Computing (InPar12)*.
- SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. GRAMPS: A programming model for graphics pipelines. *ACM Trans. Graph.* 28, 1, 1–11.
- TANENBAUM, A. S. 2007. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA.
- TZENG, S., PATNEY, A., AND OWENS, J. D. 2010. Task management for irregular-parallel workloads on the GPU. In *Proc. High Performance Graphics*, Eurographics Association, HPG '10, 29–37.
- ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. RenderAnts: interactive Reyes rendering on GPUs. *ACM Trans. Graph.* 28, 5 (Dec.), 155:1–155:11.
- ZHOU, K., GONG, M., HUANG, X., AND GUO, B. 2011. Data-parallel octrees for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics* 17, 5 (May), 669–681.