Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU

Markus Steinberger* Michael Kenzel* Pedro Boechat* Bernhard Kerbl* Mark Dokter* Dieter Schmalstieg*

Graz University of Technology, Austria

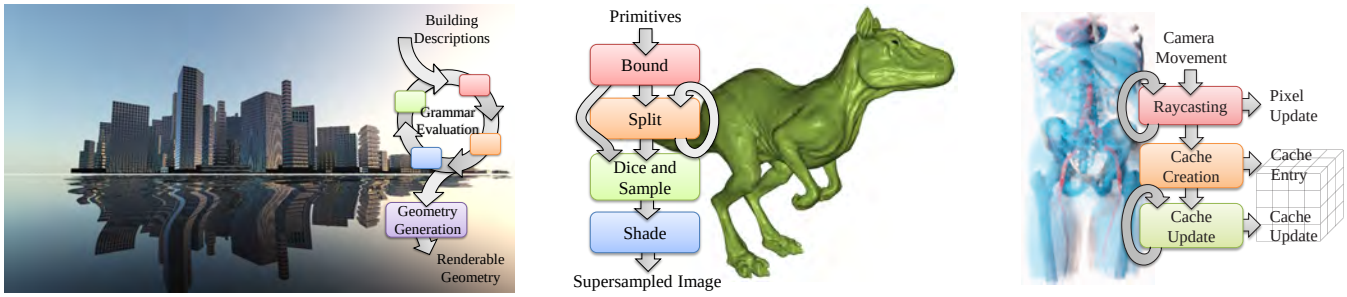


Figure 1: *Complex rendering pipelines, such as procedural city generation (left), Reyes rendering (middle) and volume rendering with irradiance caching (right) can be implemented effectively based on Whippletree and its task-based programming model. In all three examples, we were able to significantly increase performance.*

Abstract

In this paper, we present Whippletree, a novel approach to scheduling dynamic, irregular workloads on the GPU. We introduce a new programming model which offers the simplicity and expressiveness of task-based parallelism while retaining all aspects of the multi-level execution hierarchy essential to unlocking the full potential of a modern GPU. At the same time, our programming model lends itself to efficient implementation on the SIMD-based architecture typical of a current GPU. We demonstrate the practical utility of our model by providing a reference implementation on top of current CUDA hardware. Furthermore, we show that our model compares favorably to traditional approaches in terms of both performance as well as the range of applications that can be covered. We demonstrate the benefits of our model for recursive Reyes rendering, procedural geometry generation and volume rendering with concurrent irradiance caching.

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

Keywords: GPU, scheduling, parallel computing, megakernel, persistent threads, Dynamic Parallelism

Links: DL PDF WEB CODE

*e-mail: steinberger@icg.tugraz.at, kenzel@icg.tugraz.at, boechat@icg.tugraz.at, kerbl@icg.tugraz.at, dokter@icg.tugraz.at, schmalstieg@icg.tugraz.at

1 Introduction

Today's graphics processing unit (GPU) has evolved from special-purpose, fixed-function hardware into a fully programmable, massively parallel processor with thousands of processing cores. Harnessing this processing power is key in a wide range of applications. While applications generating static, parallel workloads naturally lend themselves to execution on the GPU, irregular workloads can usually not be effectively mapped for efficient GPU execution. Thus, we see an imminent need for a programming model that lends itself to high performance GPU execution of dynamic, irregular workloads. To achieve high performance, such a programming model must take into account the special characteristics of parallel execution on the GPU.

Parallel execution on the GPU is organized into a three-level hierarchy. At the lowest level, small groups of cores operate in a single instruction, multiple data (SIMD) fashion. While it is possible for control flow within the same SIMD group to take different branches, execution of these branches has to be serialized, leading to a condition known as *thread divergence*. At the intermediate level, multiple SIMD groups are organized into a *streaming multiprocessor* (SM). Each SM contains a small amount of fast local shared memory accessible to all SIMD groups on the SM. At the top level, the GPU itself consists of multiple SM units and is connected to global graphics memory. Two programming models are prevalent on this hardware: shading languages and compute languages.

Using shading languages such as GLSL or HLSL, parts of the otherwise fixed graphics pipeline can be programmed. Scheduling of the graphics pipeline on the parallel execution cores of the GPU is taken care of by opaque, special-purpose hardware. The simplicity and efficiency of this programming model, however, comes at the cost of being confined to the rigid pipeline structure. A substantial part of the implementation effort associated with advanced rendering techniques goes into circumventing the limitations of the graphics pipeline, *e. g.*, by making clever use of multipass rendering.

Compute languages such as CUDA or OpenCL depart from the notion of a graphics pipeline and present a data-parallel programming model: A *kernel function* is *launched* on the GPU where is it executed by a large number of threads. Threads are grouped into so-called *warps*, which are scheduled for execution on the SIMD groups of an SM following a time and space multiplexing scheme.

Blocks are groups of warps running on the same SM. Therefore, all threads in a block can communicate through the shared memory on their SM.

Much research has recently been devoted to using compute languages to implement software rendering pipelines that cannot effectively be expressed using shading languages. Examples include raytracing [Parker et al. 2010], Reyes rendering [Zhou et al. 2009], and multi-fragment effects [Liu et al. 2010]. However, both shading languages and compute languages are arguably far from ideal for these purposes. A programmable pipeline should be allowed to follow arbitrary, potentially cyclic, execution graphs [Sugerman et al. 2009]. We identify the following features required of a programming model to effectively deal with dynamic irregular workloads:

Multiprogramming, *i. e.*, a multiple instruction, multiple data (MIMD) model, to allow simultaneous execution of multiple pipeline stages. A MIMD model is essential for exposing parallelism which is needed to fully utilize current GPU hardware. Compute languages force every dynamic pipeline to be broken into individual kernels instead of multiplexing multiple stages, in spite of the hardware offering fully independent SM units.

Dynamic work generation. To model recursive pipelines, one needs to be able to dynamically launch new work from GPU code. Efficient scheduling of such dynamic and potentially irregular workloads calls for automatic load balancing.

Data locality. Data should be forwarded from one pipeline stage to the next through shared memory whenever possible. When pipeline stages are implemented in separate kernels, such forwarding is not possible, since shared memory is not persistent across kernel launches.

Support for different kinds of parallelism. The work to be done in different pipeline stages may offer different opportunities for parallel processing. One stage may process a large number of independent data elements in parallel, while in another stage, multiple threads could work on the same element.

While many researchers have gone into addressing the problem of multiprogramming and dynamic work generation, data locality received only little attention. No work to date offers efficient support for different kinds of parallelism. In this paper, we introduce *Whippletree*, a new approach to task-based parallelism on the GPU, which is the first to address all of the above issues in a single solution. The main contribution of Whippletree is its fine-grained resource scheduling and its ability to exploit sparse, scattered parallelism. Complex branching or recursive graphics pipelines as well as algorithms processing hierarchical data structures can be scheduled such that SM units are efficiently filled with coherent workloads. Moreover, the Whippletree model allows full MIMD task-parallelism without problematic interleaving of multiple kernels. Tasks can be created dynamically and scheduling considers data locality.

We demonstrate that Whippletree is very efficient by studying the performance of recursive, tree, and pipeline algorithms. We compare our implementation to successive kernel launches as well as a scheduling approach built on top of NVIDIA's Dynamic Parallelism. These comparisons demonstrate that our implementation is the most efficient approach when handling fine-grained parallelism. Finally, we discuss three application examples: Reyes rendering, irradiance caching for volume rendering, and procedural geometry generation on the GPU, which would be hard to implement on currently existing programming models (Figure 1).

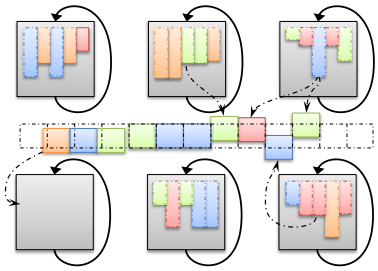


Figure 2: *A persistent megakernel model combines scheduling from a queue implemented in software with persistent worker threads running in a loop until the queue is empty. During execution, new work can be placed in the queue.*

2 Related work

We first review traditional GPU compute languages. When launching a kernel, an execution configuration has to be specified, defining the number of threads per block as well as the number of blocks. This configuration remains fixed for the duration of the kernel. A multi-stage algorithm or pipeline must be split into individual kernels. The output of one kernel usually serves as input for the next.

This model is inefficient for irregular workloads: First, a dependent kernel can only be launched after the previous kernel has finished. Thus, a small number of long-running threads can cause significant delay. Second, all data forwarded from one kernel to the next must be passed through slow global memory. Third, a change of execution configuration requires a new kernel launch. Instruction to launch a kernel is given by application code running on the CPU. If the execution configuration for one kernel depends on results of a previous kernel launch, CPU-GPU synchronization and a memory transfer from graphics memory to system memory becomes necessary.

Persistent threads Many limitations of current compute languages are inherent to the programming model rather than actual limitations of the hardware. Consequently, improving the programming model has received much attention. One important solution to support dynamic work generation is the *persistent threads* approach [Aila and Laine 2009]. A single kernel keeps each SM continuously occupied. All threads spin in a loop and draw new work items from a software queue in each iteration. New work can dynamically be generated by simply pushing items into the queue.

The key to efficient scheduling in software is a fast queue implementation. A system can only handle dynamic workloads, if the queue supports simultaneous enqueue and dequeue operations. Unfortunately, contention from many threads can quickly make the work queue become a major bottleneck. Multiple distributed queues, *e. g.*, one per SM, can be used to mitigate contention. To achieve load balancing in such a setup, work stealing [Cederman and Tsigas 2008; Chen et al. 2010; Chatterjee et al. 2011] or work donation [Tzeng et al. 2010] can be used.

Further improvements include exploiting producer-consumer locality via local shared memory [Satish et al. 2009; Breitbart 2011; Yan et al. 2013]. The persistent threads model can further be extended to support GPU-wide synchronization via message passing [Stuart and Owens 2009; Luo et al. 2010; Xiao and Feng 2010] and dynamic priorities [Steinberger et al. 2012].

Megakernels Ubershaders [Hargreaves 2005] work around the limitation that only a single graphics pipeline configuration can active at any time. All potentially required functionality is put

into one big shader program that branches to different subroutines based on the input data. The equivalent concept in the context of compute languages is called a *megakernel*. Thread divergence can become a major issue in such an approach. Stream compaction techniques [Hoberock et al. 2009] try to avoid this problem by sorting before megakernel execution. In a *persistent megakernel*, all threads run in a loop, draw some work from a queue, and branch to the corresponding subroutine (Figure 2).

The choice of the number of threads processing each element drawn from the queue has a defining influence on the properties of the resulting system. Distributing items to individual threads can lead to severe thread divergence and prevents features like shared memory from being used. Scheduling a whole warp or even block to work on an item reduces thread divergence and lowers queue contention. Optix [Parker et al. 2010], *e. g.*, schedules warp-sized ray bundles and uses heuristics to reduce divergence. Other approaches also use warp-sized [Tzeng et al. 2010] or even block-sized [Chatterjee et al. 2011] work items.

Softshell [Steinberger et al. 2012] supports both, work items being processed by individual threads as well as work items being processed by a whole block simultaneously, making it the approach most closely related to ours. While offering more flexibility than previous systems, it is still quite restrictive. Good performance is only achieved if a single block size is used, warp-level features are not considered, and tasks are mixed up arbitrarily. Furthermore, scheduling involves a high amount of slow dynamic memory allocation and intermediate data structures, resulting in relatively slow performance. Additionally, if different block sizes are used, idle threads cause a significant performance drop.

Time-sliced kernels Laine et al. [2013] argue that thread divergence poses a severe problem to megakernels and propose to use *time-sliced kernels* (TSK) with dedicated queues per kernel instead. TSK is only applicable to problems which can be efficiently separated into individual stages. It also prevents taking advantage of data locality as data needs to be forwarded from one kernel to the next through global memory.

Dynamic Parallelism *Dynamic Parallelism* [NVIDIA 2012] allows launching new kernels directly from code executing on the GPU. However, in the current implementation the book keeping necessary to track the kernel launch hierarchy seems to be a performance limiting factor. Furthermore, in order to avoid blocking the GPU, preemptive scheduling of blocks must be supported, which in turn requires saving and restoring block-level execution contexts. Finally, data communication between kernels has to go through slow global memory.

As a reference implementation of our programming model, we present the Whippletree Megakernel (WMK), an evolution of the traditional megakernel concept. We compare WMK to TSK and another scheduler based on Dynamic Parallelism. We demonstrate that WMK outperforms both contenders on scenarios with heterogeneous workloads, despite being a pure software implementation.

3 Programming model

The GPU is designed for maximizing throughput by latency hiding. Each SM keeps many warps *in flight*. Whenever a warp has to wait for a memory access, the SM switches to another warp. Fast context switching is critical for this strategy to succeed, thus, the state of all warps is kept directly on the SM. The number of warps in flight on an SM is thus limited by the hardware resources available for storing warp states, *i. e.*, the size of the register file and shared memory.

Table 1: *Overview of the three task types supported by our programming model in decreasing order of the associated feature set. M denotes the warp size of the device and B the maximum block size.*

task type	thread count	feature set
level-0	$2..M$	warp-level
level-1	$M..B$	block-level
level-2	1	global

On each level of the GPU execution hierarchy, application code can rely on a certain set of assumptions of decreasing strength. At the lowest level, code executed in the same SIMD group can rely on implicit synchronization due to lock-step execution, access to shared memory, and special instructions for intra-warp communication. One level above, code running on the same SM can still use shared memory and barrier synchronization. At the highest level—across multiprocessors—code may only use operations on global memory.

While previous approaches hide these special properties of the hardware architecture from the user, we recognize the fact that taking advantage of the guarantees described above is essential for unlocking the full potential of the GPU. To this end, we propose *Whippletree*, a new programming model designed to bring the simplicity and expressiveness of task-based parallelism to the GPU in a way that does not limit performance.

To provide a powerful programming abstraction while still enabling application code to take full advantage of the hardware, Whippletree distinguishes three task types (Table 1):

level-0 tasks are multi-threaded tasks that must be executed by threads of the same SIMD group. Thus, the implementation of such a task can use warp-level features such as lock-step execution, warp vote functions, register shuffle, as well as shared memory. The number of threads (thread count) working on a level-0 task is therefore limited by the warp size of the device. The thread count is specified by the user when defining the task. For efficiency reasons, multiple level-0 tasks of the same kind may be executed in the same warp if possible. To achieve optimal grouping, a level-0 task's thread count should therefore be an integer factor of the warp size.

level-1 tasks are multi-threaded tasks where all threads are to be executed on the same SM. Block-level features like shared memory and barrier synchronization can be used. For efficient execution, the thread count should be chosen to be a multiple of the warp size.

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 SM. Multiple level-2 tasks of the same kind may be grouped to fill up a warp, maximizing GPU utilization while trying to keep thread divergence low.

The Whippletree programming model provides four basic building blocks:

Work items define data elements to be processed during the execution of an algorithm.

Procedures provide the code that implements the processing of work items. Each procedure declares the number of threads it requires to process a single work item. Furthermore, it must specify its task type, defining the set of features the code may be able to use. Procedures are expected to terminate in a finite amount of time.

Tasks are created by specifying a work item and the procedure the work item should be handed to. Whippletree takes care of scheduling available tasks for execution in a way that respects the constraints associated with the respective task type. During execution, a task can dynamically spawn new tasks.

Programs are closed sets of procedures. A program must be self-contained in the sense that any procedure in a program may only spawn tasks for procedures also part of the program.

3.1 Programming interface

We created a reference implementation of Whippletree as a C++ template library in CUDA. Listing 1 shows a procedure implementing one stage of a simple Reyes pipeline using our interface and Listing 2 details on how this pipeline is put together and launched on the host side.

Listing 1: *Example demonstrating a Whippletree procedure implementing the recursive split stage in a Reyes rendering pipeline similar to the one presented in Figure 1 and Section 6.1. Note that this is a simplified example where only two threads work together on a small problem.*

```
struct Quad {
  Vertex v[4];
};

struct Split : public Procedure
{
  typedef Quad WorkItem;
  static const int NumThreads = 2;
  static const int TaskType = WARP;
  static const int SharedMemory = 0;

  template<class Context>
  __device__ static void execute(
     int threadId, const Quad* quad, void* shared)
  {
    float3 mDist = quad.v[threadId+1] - quad.v[0];
    float approxS = length(mDist);
    float sizeX = Context::shfl(approxS,0);
    float sizeY = Context::shfl(approxS,1);
    Quad out;
    if(sizeX > sizeY)
    {
      float3 v01 = quad.v[1] - quad.v[0];
      float3 v23 = quad.v[3] - quad.v[2];
      out.v[0] = quad.v[0] + 0.5f*threadId*v01;
      out.v[1] = quad.v[0] + 0.5f*(threadId+1)*v01;
      out.v[2] = quad.v[2] + 0.5f*threadId*v23;
      out.v[3] = quad.v[2] + 0.5f*(threadId+1)*v23;
    }
    else
    { ... }
    if(out.size() > Splitthreshold)
      spawn<Split>(out);
    else
      spawn<DiceAndSample>(out);
  }
};
```

A user defines procedures by deriving a new class from `Procedure` and declaring the task type and resources needed for execution such as thread count and shared memory. The procedure also needs to declare the type of its input work item, which can be an arbitrary type—in this example a `Quad`. The code for the procedure is provided by implementing the `execute()` method template, which is called from each thread executing the given task. The work item, thread index, and a pointer to the shared memory region allocated for the task's execution are passed in as parameters. In this example, two threads are being used to split an input quad along its longer axis into two new quads—each generated by one thread. Functions of the task-specific feature set, like barrier synchronization, are made accessible through the type passed in the `Context` template argument. In this example, the `shfl` instruction is used to efficiently exchange the computed quad dimensions. Finally, new tasks are generated for the `Split` or `DiceAndSample` procedures depending on the quad dimensions.

Listing 2: *Program specification and host code to initialize and run a simple Reyes pipeline, which is partially defined in Listing 1.*

```
struct SimpleReyesProgram : public
   Program<Bound, Split, DiceAndSample, Shade> {};

typedef Whippletree<SimpleReyesProgram,
  GlobalPerProcedureQueuing> ReyesPipeline;

struct QuadInit
{
  typedef Quad* InputData;
  __device__ static void init(
    int threadId, InputData quads)
  {
    spawn<Bound>(quads[threadId]);
  }
};

void initAndRunPipeline(Model& model)
{
  ReyesPipeline reyes;
  reyes.init<QuadInit>(model.num(), model.quads());
  reyes.run();
}
```

On the host side, a program is put together by instantiating the `Program` template with all necessary procedures. A runnable `ReyesPipeline` is formed by combining the Reyes `Program` with a scheduler implementation using the `Whippletree` template. Before the pipeline can be executed by calling the `run()` method, input work items must be created by calling the `init()` method template of the `Whippletree` class. Similar to the concept of a `Procedure`, the type specified as the template argument to `init()` has to provide a method that is called to spawn the input work items in parallel on the GPU. In this example, we simply spawn a task for each quad of the model. Overall, the template-based design not only allows us to enforce the constraints of the Whippletree task model at compile time, but also enables potentially huge benefits from static optimizations such as function inlining.

4 Implementation

In conjunction with the Whippletree programming model, we present a scheduling approach that provides an efficient mapping of the Whippletree programming model to current SIMD-based GPU architectures: the *Whippletree Megakernel* (WMK). WMK supports multi-programming of different tasks on top of CUDA. It respects the constraints of different task types, while ensuring high performance. Furthermore, we also propose easy-to-use mechanisms for load balancing, data-locality-aware queuing, as well as configurable scheduling strategies.

WMK inherits some traits from persistent megakernels. Blocks of threads (worker-blocks) are launched to exactly fill up all multiprocessors. These worker-blocks execute a loop drawing tasks from work queues. Procedures are implemented as branches in the main loop. When new tasks are generated, they are inserted into the queues. In this way, concurrent execution of multiple tasks, *e. g.*, multiple stages of a pipeline, is supported. Load balancing between worker-blocks is achieved through the work queues.

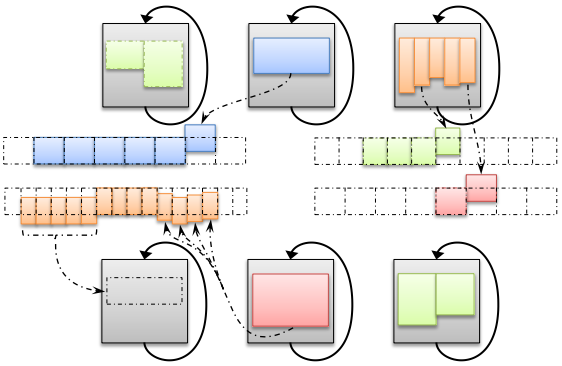


Figure 3: *WMK consists of worker-blocks, continuously drawing tasks from queues. We use one queue per procedure, which is essential for a divergence-free execution of different task types. During execution, new tasks of any kind can be created.*

4.1 Dynamic worker-blocks

Unlike persistent megakernels, WMK dynamically assigns available threads to work on the incoming tasks. To support this feature, we associate an individual queue with each procedure (Figure 3).

To have a sufficient number of idle threads available to start the execution of any procedures, we synchronize all threads inside a block at the beginning of each loop iteration. According to the current scheduling strategy, a queue is chosen to retrieve tasks for all threads. If this queue holds level-2 tasks, WMK tries to draw one task for each thread. For level-0 tasks, we compute how many tasks fit in a warp and try to fill up all warps in the worker-block. For level-1 tasks, the easiest approach would be to draw a single task and use just as many threads as needed for execution, while the remaining threads are idle. This strategy would be very inefficient for programs containing level-1 tasks of varying thread count. To increase utilization, we execute multiple level-1 tasks concurrently in the same worker-block. Similar to the execution of multiple level-0 tasks, we draw as many level-1 tasks from the queue as can be executed concurrently with the available number of threads.

Since multiple level-1 tasks are executed concurrently, the default CUDA barriers, which synchronize the entire block, cannot be used. Instead, WMK implements custom barrier synchronization on top of the synchronization primitives found in NVIDIA's PTX instruction set. PTX allows the use of up to 16 named barriers to synchronize an arbitrary number of warps. During execution, we use the barriers 1-15 and dynamically assign them to individual tasks. In this way, a worker-block can dynamically assign threads to different tasks, as shown in Figure 4. Application code can access this synchronization function though the `Context` template argument. Further functions accessible via the `Context` argument are warp voting and register shuffle. These instructions cannot be directly mapped to the underlying CUDA instructions either, as they have to work on subsets of the threads in a block/warp. For example, voting for level-0 tasks, uses warp votes, but selects only those parts of the voting result that correspond to threads in the same level-0 task.

To maximize occupancy, the worker-block size should be chosen to be the smallest common multiple of the thread counts of all level-1 tasks. However, a first restriction to this choice comes from the maximum block size supported by the hardware. Additionally, tasks may use shared memory, which may further limit the block size. As procedures are fused into one kernel, the static shared memory requirements of all procedures would accumulate, which may severely reduce occupancy. To prevent unnecessarily large

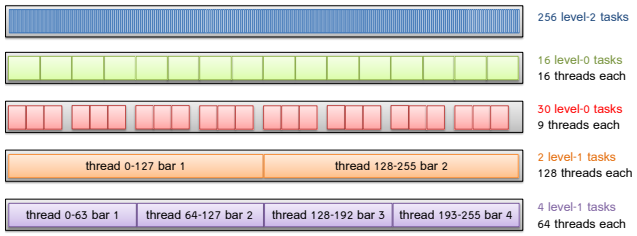


Figure 4: *A worker-block of 256 threads dynamically assigns its threads to the incoming tasks and allots individual barriers (bar) to level-1 tasks while maximizing utilization.*

shared memory requirements, we can exploit the fact that all threads of a worker-block will always execute the same procedure. We dynamically assign shared memory to tasks and thus enable reuse of shared memory by different procedures. Still, the procedure with the largest shared memory demands limits the usable block size. Considering these constraints, we evaluate all possible worker-block sizes and choose the one yielding the maximum utilization. These steps are implemented using metaprogramming techniques to make the decision at compile time.

4.2 Queue management and data locality

Our implementation builds on a configurable library of high-performance queues, which are organized in a two level hierarchy. In order to exploit data locality, a small amount of shared memory is used to implement the top level of the hierarchy. Requests are always answered by the top level queues if possible. To support load balancing among worker-blocks and SM units, the lower-level hierarchy implements queues in global memory. For these global queues, we use a queue similar to the one proposed by Steinberger et al. [2012]. These queues support concurrent insertion and removal.

Global queuing The global queues use a fixed-size ring buffer with front and back pointers and overflow/underflow protection counters. When a thread wants to add a task to the queue, it increases the back pointer of the queue using atomic addition, allocating a slot to this thread. Every slot is protected with an individual lock flag to avoid read-before-write and write-before-read hazards. If the flag indicates an empty slot, a thread can safely insert a task into that slot and set the flag to 'filled' afterwards. When elements are to be removed from the queue, the front pointer is atomically increased to retrieve a filled slot. If the flag indicates that no task has been added to the queue for this slot yet, the thread keeps polling the flag until a task becomes available. Then it removes the task from the queue setting the flag to 'free'.

To reduce contention, we use warp votes to determine how many threads in the warp want to enqueue a task. Then, a single thread requests slots for all threads, before all threads write the newly created tasks to the queue in parallel. This strategy automatically leads to fast coalesced memory access.

Local queuing As round trips to global memory are costly, shared memory queues are essential for high performance. Shared memory is limited and might also be used during task execution, so these queues must be rather small. After deciding the worker-block size, we use the remaining shared memory for local queuing.

By default, we evenly distribute the memory among all queues, but the queue sizes can also be configured manually. To achieve full occupancy when drawing tasks from local queues, we require each

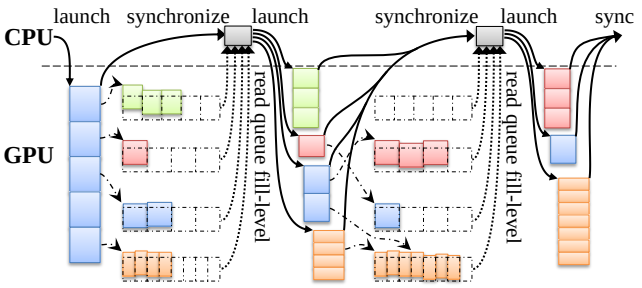


Figure 5: *TSK maps different procedures to separate kernels. Queues do not need to support concurrent enqueue/dequeue. Data between kernel launches has to be passed through global memory. The CPU must read queue fill rates, before launching new kernels.*

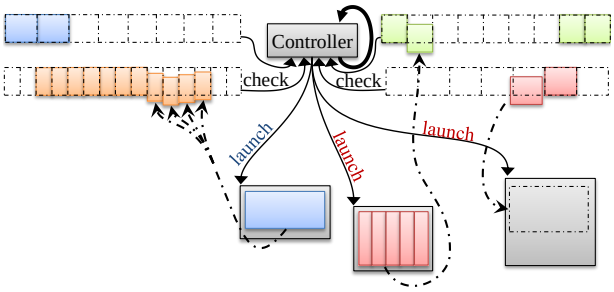


Figure 6: *Hybrid dynamic parallelism uses queues to collect tasks for each procedure. A controller block periodically checks the queues and launches new workers directly from the GPU.*

queue to be able to hold enough tasks to fill a worker-block. If there is too little memory to achieve this goal, a heuristic can be used to select which tasks should be backed by a local queue: starting with the task that requires the least memory, we allocate queues with minimal size, until there is no more space available.

4.3 Scheduling policies

Different scheduling policies can be employed to decide which procedure to execute next. As the optimal choice of policy is highly application-specific, we provide multiple predefined policies and means for customization. Predefined policies include *random selection*, *round-robin scheduling*, *fixed priority scheduling*, and *execution-time oriented scheduling*.

With random selection, each procedure can be assigned a probability. New tasks are drawn from the work queues according to the in this way defined distribution. If the number of tasks available in the chosen queue is too low to fill the worker-block, we choose another procedure based on the remaining probabilities. In case of round-robin scheduling, every worker-block stores information about which procedure has been executed last and chooses the queue associated with the next procedure during the following dequeue.

In fixed-priority scheduling, procedures with higher priority are always chosen before lower priority procedures as long as they fill up the worker block. Such a strategy aids in, *e. g.*, keeping the queue lengths short by draining elements from the back of a pipeline by prioritizing procedures at the end of the pipeline. In execution-time oriented scheduling, each procedure is assigned a time quota. The overall time spent executing each procedure is monitored. The procedure which is furthest from the desired quota is then chosen during dequeue.

4.4 Alternative implementations

The Whippletree programming model is not restricted to the WMK reference implementation. We provide two alternative scheduler implementations that fulfill the Whippletree programming model: *time-sliced kernels* (TSK) and *hybrid dynamic parallelism* (HDP), both making use of efficient global queuing.

Time-sliced kernels (TSK) A common approach for supporting multiple tasks is calling separate kernels in turn. Similar to Laine et al. [2013], we extend this approach by allowing every procedure to have a dedicated input queue (Figure 5). Because the queues do not need to support concurrent enqueue and dequeue operations, queuing can be implemented even more efficiently without the additional lock per queue slot. As each procedure is executed in a separate kernel, the execution configuration can be individually chosen to achieve the best possible occupancy. Furthermore, divergence is also expected to be low. To achieve a good GPU utilization, even if the queue fill-levels are low, all kernels can be issued into different streams and thus be executed concurrently. Because new tasks can be generated during execution, pipelines with cycles may require multiple iterations of kernel launches, until the entire pipeline is executed.

Hybrid dynamic parallelism (HDP) Dynamic Parallelism allows launching new kernels from inside a kernel. However, the dimension of a new kernel launch must be at least one block. Thus, we take an approach similar to TSK by combining dynamically launched kernels with individual queues for each procedure. These launches are handled by a controller kernel which essentially takes on the role of the CPU in TSK and launches new kernels until all queues are empty. To this aim, it continuously checks the number of tasks available in each queue. As soon as there are enough tasks to start at least one warp, the controller forms as many blocks as possible and launches them in a new worker kernel (Figure 6). Furthermore, it keeps track of the number of blocks which have not started their execution yet. If this number becomes low, the controller will also launch partially filled blocks to make sure available processing cores will be used and the program will eventually run to completion. As tasks may concurrently be enqueued and dequeued, we use the same queues for HDP as for WMK.

All three approaches have their distinct advantages and disadvantages (Table 2). All techniques can collect level-2 and level-0 tasks to fill entire thread blocks. While TSK relies on the CPU to control execution, HDP and WMK operate in autonomy. A downside of Dynamic Parallelism is that it must keep track of all dynamically launched kernels, which, at least in current implementations, entails

Table 2: *Comparison of our three different scheduler implementations: time-sliced kernels (TSK), hybrid dynamic parallelism (HDP), and the Whippletree megakernel (WMK).*

	TSK	HDP	WMK
collect tasks per procedure	•	•	•
GPU autonomy		•	•
no book keeping	•		•
optimal occupancy	•	•	
inter-task synchronization			•
adaptive scheduling			•
fast local queuing			•

significant book-keeping overhead. The separation of procedures into different kernels as implemented by TSK and HDP has the advantage that each kernel can be optimized to achieve perfect register and shared memory usage, which might lead to a better utilization.

Of all approaches, WMK has the richest feature set. It supports inter-task communication and synchronization via message passing. Only WMK can fully adjust its scheduling policy to the current needs: Since it dequeues tasks directly before execution, it may select or reorder tasks allowing arbitrary scheduling policies. HDP and TSK can only slightly influence which task is being executed next, as the hardware scheduler determines which thread block is chosen for execution. WMK can use fast shared queues, while HDP and TSK require all work items to be transmitted via global memory.

5 Performance analysis

We compare the performance of WMK to our two alternative schedulers TSK and HDP. To assess the impact of local queues we also compare a version of WMK that uses global queues only (WMK$_G$). We tested three synthetic benchmarks: a simple recursion, a tree traversal and a multi-stage pipeline with strongly varying characteristics. As simulated load, we executed a fixed number of *fused-multiply-add* (FMA) operations or a fixed number of global memory reads and writes, to simulate compute-bound and memory-bound applications respectively. Furthermore, we included Softshell [Steinberger et al. 2012] (SFT) into our comparison, as it is closely related to Whippletree. As SFT does not support level-0 tasks, we modeled them with Softshell's workpackage primitive. In the following, we give an overview of the benchmark results. The complete results can be found in the supplemental material. Our test system was running Windows 8 on an Intel i7 CPU with 3.4GHz. All tests were compiled using CUDA 5.5 and run on NVIDIA GTX 580, 680, and TITAN cards, representing three different GPU generations.

Recursive algorithm To evaluate each technique's task creation and scheduling overhead, we simulate a simple recursive application with level-2 and level-1 tasks. 38 000 initial tasks are processed and re-spawned with linearly decreasing probability, which reaches 0% at iteration 40. For every task, 512 FMA instructions or 64 memory transactions are executed as simulated load.

The WMK variants delivered the best performance in all 12 tests, achieving speedups up to $40\times$ over SFT and $4\times$ over TSK (Table 3). As this test case only contains a single procedure, WMK does not suffer from a suboptimal execution configuration. However, the queues used in WMK (and HDP) are more complex, requiring four additional registers and causing kernel occupancy to drop from 100% to 75% when compared to TSK. Table 3 also lists the theoretical peak performance. Memory access loads are closer to the theoretical peak than FMA loads. While this might be due to the lack of instruction-level parallelism in our simulated load, it also indicates that the scheduling strategies do not demand excessive memory bandwidth.

WMK achieves the highest performance despite running at comparatively low occupancy, which suggests that the read-back and launch overhead in TSK is significant. Queuing in shared memory boosts performance significantly, especially on the GTX 680. While TSK, HDP, and WMK perform reasonably well in all test cases, SFT is between 3 to 40 times slower than the other approaches. The major problem of SFT is the overhead of dynamic memory allocation and virtual function calls, which significantly slows down scheduling.

Expanding tree In this example, we generate a rapidly expanding tree structure. Each node has two children, which can be of any of four node types. Each node type is associated with a different procedure. For each child, a new level-2 task is spawned. The tree is expanded up to 2^{24} nodes. We set the simulated load for each node to 120 FMA instructions using 24 registers. Figure 7 shows the average number of nodes processed per second for increasing tree depths. Across all GPU types and scenarios, the WMK variants either achieved the highest performance or were only marginally behind the best technique. We observed that WMK with global queues works well for low tree depths, while local queues boost performance for larger tree depths up to a factor of $2\times$, especially on the GTX 580 and 680. At low tree depths, load balancing between worker-blocks is important. For larger tree depths, a sufficient number of tasks is available to all worker-blocks, and the reduced memory latency of shared queues increases performance.

Table 3: *Recursive algorithm using level-2 and level-1 tasks with FMA and memory access simulated load for SFT, TSK, HDP, WMK, and WMK$_G$, on the GTX 580, 680, and TITAN. HDP is only supported on the GTX TITAN. Performance provided in GFLOPS (for FMA) and GB/s (for Mem).*

		580		680		TITAN	
		FMA	Mem	FMA	Mem	FMA	Mem
theoretical perf.		1581	192.4	3090	192.2	4500	288.4
level-2	SFT	12.3	15.1	19.6	33.3	24.6	47.5
	TSK	342.9	79.6	82.3	82.3	874.1	120.9
	HDP	-	-	-	-	636.8	106.7
	WMK$_G$	404.0	88.5	43.0	48.4	**1005.0**	147.9
	WMK	**559.6**	**114.6**	**147.9**	**131.6**	934.0	**150.5**
level-1	SFT	273.7	43.5	66.9	14.9	295.4	54.5
	TSK	878.1	117.9	268.6	60.0	919.8	139.9
	HDP	-	-	-	-	881.0	123.6
	WMK$_G$	884.5	126.4	270.5	71.6	1071.2	145.4
	WMK	**987.3**	**126.4**	**1172.3**	**154.0**	**1145.7**	**161.1**

TSK becomes more efficient with increasing tree depth as the relative overhead of kernel launches diminishes. HDP slightly outperforms TSK for small trees, likely because it avoids CPU-GPU round-trips. For larger trees, the effect is reversed. SFT performs poorly and even slightly drops in performance for larger tree depths. SFT again suffers from memory management and work aggregation overheads, which get more severe as the allocator's memory pool is filling up. Surprisingly, the performance on the GTX 680 is overall very low in comparison to the other two GPU architectures. We can only guess that the 680's design, focused on high graphics performance, is detrimental to algorithms that involve a lot of control and scheduling and less processing.

Feed-Forward pipeline To evaluate each technique's ability to cope with procedures of different character, we simulate eight variants of a seven-stage feed-forward pipeline. We compare the use of procedures of equal thread count (256) versus different thread counts (1–512), and equal number of registers used (32 per thread) versus different number of registers (8–63 per thread). In addition, we randomize the simulated load of each task (20–1800 FMA instructions). Each pipeline stage shows expanding behavior, generating tasks for more threads than were used in the previous stage. We start each pipeline with 3000 initial tasks. To keep queue lengths as short as possible, we set the scheduling policy to preferably draw elements from stages at the back of the pipeline. As this strategy keeps queue lengths short, it allows much larger problem sets to be handled and can have a positive impact on performance as elements might stay in cache between enqueue and dequeue. As TSK and HDP do not offer the ability to influence the scheduling policy, there is no control over queue lengths. As a consequence TSK and HDP reach up to $2M$ tasks in the queues, while WMK only uses up to $100k$.

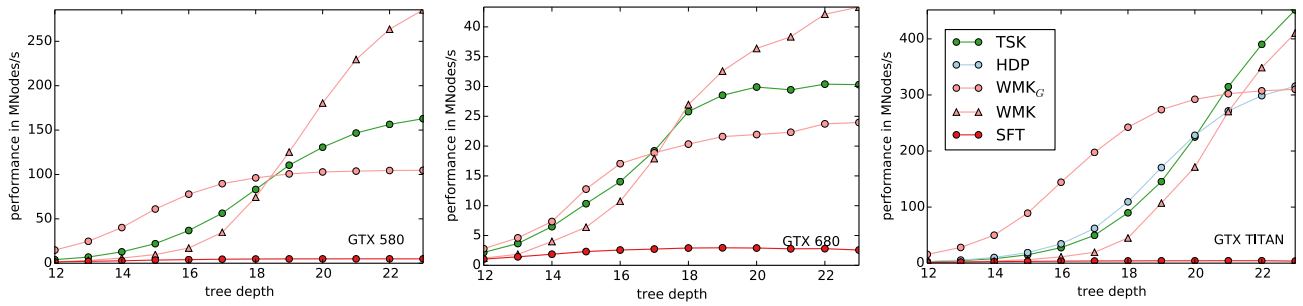


Figure 7: *Expanding tree example on a GTX 580, 680, and TITAN. Performance is given in million nodes per second. Each tree node is associated with either of four procedures. Thanks to fast load balancing via global memory, WMK$_G$ is very efficient for a low number of nodes. For wide trees, WMK achieves the best performance in most cases. On the GTX TITAN, TSK is overall slightly faster, gaining an advantage already at low tree depths. SFT is very slow due to its dynamic memory and work aggregation overhead.*

Table 4: *Relative execution time for a 7-stage pipeline compared to TSK as the baseline, using FMA instructions as simulated load on a GTX 580, 680, and TITAN. WMK$_G$ uses queues in global memory. HDP failed to complete the tests with different thread counts.*

		580		680		TITAN	
		ER	DR	ER	DR	ER	DR
equal thread counts	SFT	4.81	4.77	4.68	4.58	4.72	4.69
	TSK	1.00	1.00	1.00	1.00	**1.00**	1.00
	HDP	-	-	-	-	1.85	0.97
	WMK$_G$	0.94	0.86	0.91	0.83	1.42	1.16
	WMK	**0.58**	**0.55**	**0.38**	**0.45**	1.04	**0.82**
different thread counts	SFT	12.52	12.31	12.93	12.79	12.12	12.66
	TSK	1.00	1.00	1.00	1.00	**1.00**	1.00
	WMK$_G$	1.52	1.18	1.00	1.17	1.78	1.29
	WMK	**0.99**	**0.77**	**0.72**	**0.71**	1.27	**0.95**

ER = equal number of registers, DR = different number of registers

Relative execution times compared to TSK are shown in Table 4. TSK is very competitive, as it requires only seven kernel launches. WMK with queues in shared memory shows the best performance in 10 out of 12 cases. HDP did not complete the tests for different thread counts, which we suspect to be due to issues in the implementation of the still relatively new Dynamic Parallelism. WMK only shows a slight regress in performance with different thread counts compared to equal thread counts. The relative execution time of WMK/WMK$_G$ decreases in 5 of 6 cases when moving from equal to different number of registers. It seems that the concurrent execution of kernels with different register usage affects the hardware scheduler more severely than a lower occupancy affects WMK. Although this acyclic pipeline with strongly differing procedure characteristics is theoretically very well suited for kernel-based approaches, WMK overall seems the best choice due to its ability to keep data in shared memory. Again, SFT is $4.5\times$ slower than TSK in the equal thread counts setup. In the different thread counts test, the performance of SFT further reduces to a factor of 12, which can be attributed to Softshell's inability to handle tasks with different thread counts.

6 Applications

To show the utility of Whippletree, we build three graphics applications with different task types: a real-time Reyes pipeline, procedural geometry generation, and a volume renderer with concurrent irradiance caching.

6.1 Real-time Reyes pipeline

The Reyes rendering pipeline [Cook et al. 1987] is primarily used in cinematic productions for high quality image synthesis. A GPU implementation is challenging, as the pipeline is recursive, irregular, and has unbounded memory requirements. A Reyes pipeline consists of the following stages: *Bound*, *Split*, *Dice*, *Shade*, *Sample*, and *Composite*. Bound performs culling. Split and Dice are responsible for splitting input patches into micropolygons of subpixel size. Shade computes vertex colors, which are interpolated in Sample. The subpixel samples are then combined in Composite.

Previous attempts at bringing Reyes to the GPU have always split the pipeline into multiple kernels, using four [Patney and Owens 2008], five [Tzeng et al. 2010], or eight kernels [Zhou et al. 2009]. Tzeng et al. [2010] use persistent threads, but only for their first stage. We model the entire pipeline as a single Whippletree program. Furthermore, we swap the shade and sample stages, and perform per-sample shading. Input data is given in the form of cubic Beźier patches with a 4×4 control mesh. The different task types available in the Whippletree programming model allow each stage to be expressed in a natural yet efficient manner, as detailed in Table 5.

The *Bound* stage is modeled as a level-0 task of 16 threads. Each thread transforms one vertex to screen space and writes the vertex position to shared memory. Then, the orientation of each face is computed, and, if all face away from the camera, the patch is culled. For the *Split* stage, we use a level-0 task of 4 threads to split a patch along its longer axis. Each thread operates on one row or column of the input patch. *Dice + Sample* uses level-1 tasks of 256 threads to slice each patch 15 times along each axis, generating 16×16 vertices. At first, we use one thread per vertex, then we use one thread per micropolygon for sampling. *Shade* uses independent level-2 tasks to perform shading. *Composite* is realized in OpenGL during display.

Table 5: *Implementing a Reyes pipeline, we take advantage of all three task types available in the Whippletree programming model. Bound and Split rely on lock-step execution and communicate via shared memory, Dice + Sample uses barrier synchronization, and Shade runs as independent threads.*

Stage	Task type	Threads	Register	Shared Memory
Bound	level-0 task	16	38	128 bytes
Split	level-0 task	4	63	192 bytes
Dice + Sample	level-1 task	256	40	3072 bytes
Shade	level-2 task	1	30	0

Table 6: *Reyes rendering times (ms) on a GTX 680 and TITAN with a screen resolution of* 1920×1080 *pixels. An implementation using all three task types outperforms using level-2 tasks only. WMK achieves the best performance by exploiting data locality with queues in shared memory.*

		Sphere		Teapot		Killeroo	
		680	TITAN	680	TITAN	680	TITAN
level-2 only	TSK	7325	737.3	5804	509.3	439.3	37.5
	HDP	-	699.1	-	472.5	-	36.7
	WMK_G	4812	825.2	3809	670.0	297.3	57.0
	WMK	2410	672.7	1522	442.3	121.1	39.2
multi level	TSK	75.3	7.2	90.7	8.0	87.8	7.1
	HDP	-	6.7	-	7.0	-	6.9
	WMK_G	52.6	8.3	61.3	10.2	59.7	10.1
	WMK	**16.4**	**5.3**	**20.3**	**6.7**	**21.7**	**6.6**

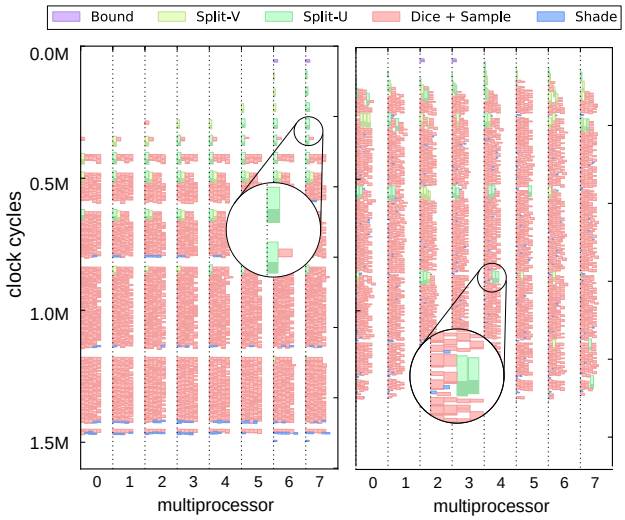
We evaluate the performance of the Reyes implementation using three scenes: *Sphere* (8 input patches), *Teapot* (32 patches) and *Killeroo* (11500 patches, shown in Figure 1, middle). Furthermore, to evaluate whether the ability to use different task types impacts execution speed, we build a version using only level-2 tasks. In this version, every task is executed by a single thread only. Therefore, there is no need for inter-thread communication. However, the level-2-only version needs to cope with the same per-task load and, thus, needs more registers to store intermediate values.

The results for all Reyes tests can be found in Table 6. The level-2-only version of the pipeline is significantly slower than the full version. For the Killeroo scene, the difference is approximately $5\times$, for Sphere and Teapot $150\times$. The increased register and shared memory usage explains the slowdown for Killeroo. Sphere and Teapot start out with very few input patches, so there is very little parallelism available during the first split operations. Using different task types to have multiple threads work on each item cooperatively offers more parallelism in the initial phase, leading to better GPU utilization and reduced task latency.

In the full version, WMK achieves the best performance in all cases. For WMK the usage of shared queues leads to a substantial performance improvement of up to $3\times$ on the GTX 680 and $1.5\times$ on the GTX TITAN. HDP achieved the second best performance, outperforming TSK by about 10%. In this cyclic pipeline setup, the repeated kernel launch overhead of TSK induces a large performance penalty. Especially in the initial phase with only few available tasks the launch overhead is apparent, as brief kernel executions are interrupted by long waiting periods (see Figure 8).

6.2 Procedural geometry generation

Procedural generation of geometry plays an important role in content creation for movies and games. An example of such a procedural approach are shape grammars. Shape grammars are mainly used for urban environments as building structures and façades often follow a repetitive pattern. Such patterns can be generated using production rules. A production rule consists of a sequence of shape operators such as, *e. g.*, geometric transformations, that are to be applied to a given input shape to produce a set of output shapes. Starting from a few simple input shapes, evaluation of these rules leads to increasingly elaborate shape compositions, eventually producing highly detailed output geometry. The generation process itself can be viewed as a tree: rules generally create multiple output shapes, which are again processed by other rules. So-called terminal shapes define the final output geometry, making up the leaves of the tree.



(a) *TSK* **(b)** *WMK*

Figure 8: *Execution trace for a subset of the multiprocessors on a GTX TITAN while rendering a single Reyes frame of the Teapot scene. Every rectangle corresponds to an executed task. Darker areas correspond to enqueue times. TSK achieves a higher occupancy. However, the clearly visible read-back and kernel launch overhead (empty regions) leads to an overall longer execution time compared to WMK. This is especially harmful in the beginning, as little data is available. WMK is able to distribute work more efficiently and shows good load balancing.*

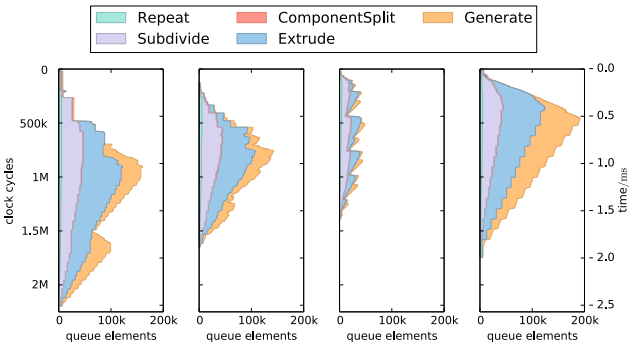
The Whippletree programming model allows us to easily re-implement a state of the art GPU shape grammar [Steinberger et al. 2014] (Figure 1, left). Every shape operator can be modeled as a procedure. Using the different task types, we are able to exploit intra-operator parallelism. For example, the operator that splits a cube into six faces can be modeled as a level-0 task using six threads, a twelve-thread task is used to generate the vertices, indices and normals of a box terminal, and an n-thread task is used to evenly distribute n shapes inside a bounding shape.

We compare the performance of our implementation using the TSK, HDP, WMK_G, and WMK schedulers and evaluate the benefits of using multi-threaded tasks. Furthermore, we investigate the influence of different scheduling policies on queue lengths.

The derivation times for a scene with a varying number of skyscrapers (Figure 1, left) is shown in Table 7. Using multi-threaded level-0 task for six out of twenty operators boosts performance by up to $3\times$. WMK achieves good results for all scene sizes. In the scene with only a single house, the performance-enabling factor is load balanc-

Table 7: *Evaluation times (in ms) for the procedural generation of different numbers of skyscrapers on a GTX TITAN. Exploiting intra-operator parallelism using level-0 tasks significantly increases performance.*

buildings	1		256		1024	
parallel operators	on	off	on	off	on	off
TSK	1.99	2.98	2.94	5.77	6.14	17.30
HDP	0.37	1.03	1.58	4.65	**5.30**	15.84
WMK_G	**0.14**	0.26	1.49	2.85	5.73	10.85
WMK	0.15	0.26	**1.41**	2.49	5.33	9.17



(a) *TSK* **(b)** *HDP* **(c)** *WMK* **(d)** *WMK reverse*

Figure 9: *Queue fill levels for a simplified skyscraper test scene. TSK and HDP start the execution of tasks whenever possible to keep the GPU busy, leading to excessive memory requirements for queuing (a, b). WMK can be configured to prioritize procedures associated with tasks closer to the leaves of the derivation tree, keeping queue lengths short (c). Inverting this scheduling policy increases queue fill levels to a maximum (d).*

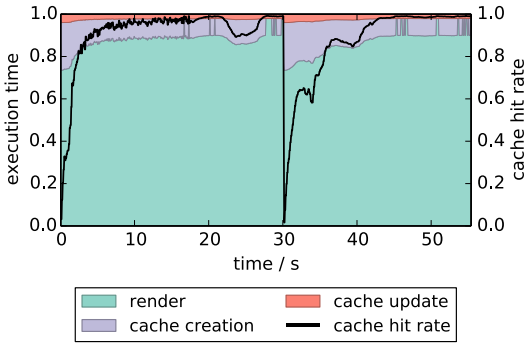


Figure 10: *Relative execution time allocated to the three different procedures during Monte Carlo volume rendering with irradiance caching. Using WMK, we dynamically adapt the time spent on cache creation and update based on the cache hit rate (black). Note how the cache creation time follows the cache hit rate. A movement of the light source at $t = 30s$ triggers a cache reset.*

ing, thus WMK_G performs slightly better than WMK. In the large scene, HDP achieves the best performance. This is not surprising, as WMK suffers from suboptimal occupancy due to the fusion of mostly simple procedures with a few very complex procedures into a single kernel. Overall, our Whippletree implementation is $5 - 10\times$ faster than the original implementation [Steinberger et al. 2014].

Figure 9 visualizes the influence of the scheduling policy and technique on queue length for a simplified rule set. While TSK and HDP launch kernels for all available tasks, WMK can dynamically make fine-grained scheduling decisions. In this example, we prioritize procedures associated with tasks close to the leaves of the derivation tree. Thus, data is drained out of the system more quickly, keeping queue lengths short, which is important for handling large problems.

6.3 Volume rendering with irradiance caching

As a final example, we apply Whippletree to volume rendering (Figure 1, right). Starting with the CUDA implementation by Kroes et al. [2012], we develop a Monte Carlo volume rendering (MCVR) solution with irradiance caching for accelerated global illumination. The main challenge in this application lies in balancing three competing procedures on the GPU: *rendering* to generate an output image using raycasting, *cache creation* to compute new irradiance cache entries and *cache update* to improve the quality of the cache by eliminating discontinuities in the estimated irradiance field. Each operation can be modeled as a procedure. Rendering executes as level-1 task with 16×16 threads, each casting a ray through a certain pixel into the scene. Cache creation works in level-1 tasks with 128 threads, each tracing rays from a common starting point into random directions to compute an irradiance estimate. Cache update uses level-0 tasks of 32 threads to check whether cache entries need to be updated.

We follow a progressive rendering approach, starting rendering tasks covering the entire screen. During rendering, each ray uses the irradiance cache to query the incoming radiance at random positions along the ray. If no cache entry is present for a given location, a cache creation task is spawned, and a rough light estimate is used instead. After raycasting, each ray adds its color estimate to the output buffer and the task re-enqueues itself to be scheduled at a later point in time. In this way, rays are continuously generated for all pixels, and the Monte Carlo estimate is consistently improved. When the camera is moved or the scene changes, the color buffer is cleared and Monte Carlo integration starts over. Note that the irradiance cache only needs to be flushed if the light sources or volume transfer function change. Cache update behaves similar to rendering: a fixed number of cache update tasks are kept in the system, which continuously re-enqueue themselves for scheduling.

To be able to quickly respond to camera movements while at the same time ensuring fast updates of the irradiance cache, scheduling must dynamically adapt to the situation at hand. Out of the presented scheduling approaches, only WMK supports the necessary scheduling policies. If the cache is well-filled, the majority of irradiance lookups can be answered, and it is not necessary to create additional cache entries. If the cache is nearly empty, it is desirable to allocate more execution time to cache creation. Nevertheless, there must still be enough time for rendering to guarantee interactive feedback to camera motion. To achieve this goal, we dynamically adapt the time spent on cache creation and update proportional to the cache hit rate (Figure 10 and supplemental video).

Additionally, we prioritize the rendering tasks individually. Due to the complex interaction between the inhomogeneous volume data set and scene light sources, the convergence rate generally varies strongly across the image. To achieve more uniform convergence, we estimate the expected gain estimated from previous Monte Carlo iterations to prioritize the rendering tasks. Using irradiance caching without prioritization of screen regions, we already could improve convergence rates by $4\times$ over Kroes et al. [2012]. Adapting the scheduling based on the expected gain yields another $2.95\times$ improvement.

7 Conclusions and future work

By incorporating the essentials of the GPU execution hierarchy into a task-based programming model, Whippletree provides a powerful abstraction without compromising generality or performance. Complex software pipelines, recursive algorithms and many other applications, which were previously very hard to map to the GPU, can now be expressed in a simple and natural way. Independence of an underlying scheduling approach allows the same Whippletree program to be run on top of different GPU schedulers. This kind of flexibility enables easy experimentation and selecting the most efficient scheduling approach on a per-application basis.

With WMK, we introduced a new approach to high-performance scheduling of dynamic, inhomogeneous workloads on the GPU. Benchmarks show that it is always efficient, particularly so in cases where data locality can be exploited. Conventional approaches based on launching multiple, separate kernels cannot take advantage of data locality across kernel launches. Thus, they only reach the performance of WMK in scenarios where only few kernel launches are needed and a high level of parallelism is available at all times.

Implementing Reyes rendering, procedural generation of geometry, and Monte Carlo volume rendering as real-world examples, we could confirm the applicability and high performance of our approach. We demonstrated how the ability to access all levels of the GPU execution hierarchy leads to a significant performance advantage. WMK even enables adaptive scheduling policies, such as time-quota-based scheduling of multiple processes, or fixed-priority scheduling to keep queue lengths short in pipeline setups.

In the future, we would like to see hardware vendors shift their focus from the unnecessarily restrictive, kernel-based model towards a new kind of programming interface that allows the GPU to more directly be programmed as the hierarchical MIMD machine that it really is. We envision features such as hardware-accelerated, freely-configurable queues, and the ability to specify the execution configuration on a per SM level to become available. To offer a maximum of control to software schedulers, it would be important to provide the functionality to launch individual warps while managing the allocation of shared memory and registers dynamically. In this way, WMK could overcome its biggest weakness, the fact that GPU occupancy is limited by the heaviest procedure.

Dynamic Parallelism could be evolved to avoid book keeping in cases where no synchronization on kernel completion is necessary. Another enhancement would be the option to bind a dynamically launched kernel to the SM of the parent block, allowing communication with the new kernel through shared memory. Even without these features, a more efficient implementation of Dynamic Parallelism could enable a combination of HDP and WMK to capture the best of both worlds. Subsets of procedures with similar resource requirements could be merged into individual WMK kernels, enabling local queuing between those procedures. At the same time Dynamic Parallelism could be used to start those kernels to provide appropriate load balancing. Whippletree is open source and can be downloaded at `http://www.icg.tugraz.at/project/parallel`.

Acknowledgements

This research was funded by the Austrian Science Fund (FWF): P23329.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. HPG*, 145–149.

BREITBART, J. 2011. Static GPU threads and an improved scan algorithm. In *Proc. Euro-Par 2010*, 373–380.

CEDERMAN, D., AND TSIGAS, P. 2008. On dynamic load balancing on graphics processors. In *Proc. Graphics Hardware*, 57–64.

CHATTERJEE, S., GROSSMAN, M., SBIRLEA, A., AND SARKAR, V. 2011. Dynamic task parallelism with a GPU work-stealing runtime system. In *Proc. Languages and Compilers for Parallel Computing*.

CHEN, L., VILLA, O., KRISHNAMOORTHY, S., AND GAO, G. 2010. Dynamic load balancing on single- and multi-gpu systems. In *IEEE Parallel Distributed Processing*.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4 (Aug.), 95–102.

HARGREAVES, S. 2005. Generating shaders from HLSL fragments. *ShaderX3: Advanced rendering with DirectX and OpenGL*.

HOBEROCK, J., LU, V., JIA, Y., AND HART, J. C. 2009. Stream compaction for deferred shading. In *Proc. HPG*, 173–180.

KROES, T., POST, F. H., AND BOTHA, C. P. 2012. Exposure render: An interactive photo-realistic volume rendering framework. *PLoS ONE 7*, 7 (07).

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proc. HPG*.

LIU, F., HUANG, M.-C., LIU, X.-H., AND WU, E.-H. 2010. Freepipe: A programmable parallel rendering architecture for efficient multi-fragment effects. In *Proc. I3D*, 75–82.

LUO, L., WONG, M., AND HWU, W.-M. 2010. An effective GPU implementation of breadth-first search. In *Proc. Design Automation Conference*, ACM, 52–55.

NVIDIA. 2012. *CUDA Dynamic Parallelism Programming Guide*.

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 TOG 29*, 4(66).

PATNEY, A., AND OWENS, J. D. 2008. Real-time Reyes-style adaptive surface subdivision. *ACM TOG 27*, 5(143).

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore GPUs. In *Proc. IEEE Parallel&Distributed Processing*.

STEINBERGER, M., KAINZ, B., KERBL, B., HAUSWIESNER, S., KENZEL, M., AND SCHMALSTIEG, D. 2012. Softshell: Dynamic scheduling on GPUs. *ACM TOG 31*, 6(161).

STEINBERGER, M., KENZEL, M., KAINZ, B., MÜLLER, J., WONKA, P., AND SCHMALSTIEG, D. 2014. Parallel generation of architecture on the GPU. In *Computer Graphics Forum*, vol. 33, 73–82.

STUART, J. A., AND OWENS, J. D. 2009. Message passing on data-parallel architectures. In *Proc. Parallel&Distributed Processing*, IEEE.

SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. GRAMPS: A programming model for graphics pipelines. *ACM TOG 28*, 1, 4:1–4:11.

TZENG, S., PATNEY, A., AND OWENS, J. D. 2010. Task management for irregular-parallel workloads on the GPU. In *Proc. HPG*, 29–37.

XIAO, S., AND FENG, W. 2010. Inter-block GPU communication via fast barrier synchronization. In *IEEE Parallel Distributed Processing*.

YAN, S., LONG, G., AND ZHANG, Y. 2013. Streamscan: fast scan algorithms for GPUs without global barrier synchronization. In *ACM Principles and Practice of Parallel Programming*, 229–238.

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. RenderAnts: interactive Reyes rendering on GPUs. *ACM TOG 28*, 5 (Dec.), 155:1–155:11.