Volume 0 (1981), Number 0 pp. 1–13 COMPUTER GRAPHICS *forum*

Hierarchical Bucket Queuing for Fine-grained Priority Scheduling on the GPU

Bernhard Kerbl[1], Michael Kenzel[1], Dieter Schmalstieg[1], Hans-Peter Seidel[2] and Markus Steinberger[2]

[1]Graz University of Technology, Austria
{kerbl|kenzel|schmalstieg}@icg.tugraz.at

[2]Max Planck Institute for Informatics, Saarbrücken, Germany
{hpseidel|msteinbe}@mpi-inf.mpg.de

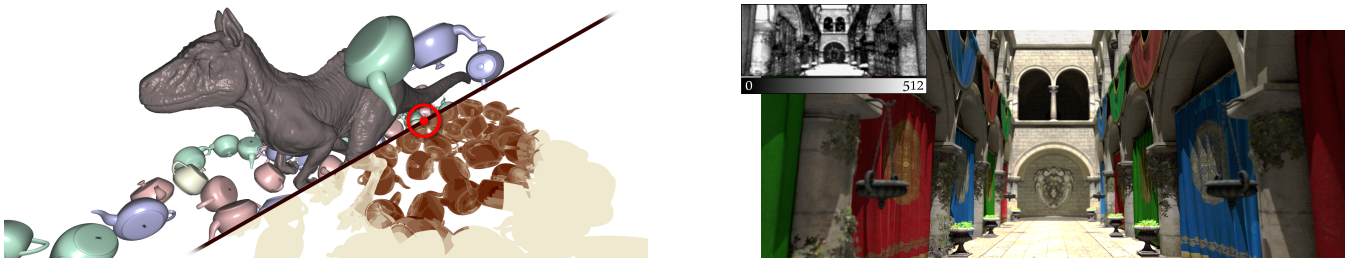


Figure 1: (left) Hierarchical bucket queues enable the implementation of foveated, Reyes-style micropolygon rendering in real-time on the GPU. By dynamically reducing the number of split recursions in areas distant from the user's current fixation point (red), rendering quality can adaptively be decreased where not needed, leading to speedups of more than 100%. (right) Using prioritization in a GPU path tracer, compute power can be directed primarily to areas of high variance to achieve faster convergence. The overlay visualizes the number of traced paths.

Abstract

While the modern graphics processing unit (GPU) offers massive parallel compute power, the ability to influence the scheduling of these immense resources is severely limited. Therefore, the GPU is widely considered to be only suitable as an externally controlled co-processor for homogeneous workloads which greatly restricts the potential applications of GPU computing. To address this issue, we present a new method to achieve fine-grained priority scheduling on the GPU: hierarchical bucket queuing. By carefully distributing the workload among multiple queues and efficiently deciding which queue to draw work from next, we enable a variety of scheduling strategies. These strategies include fair-scheduling, earliest-deadline-first scheduling, and user-defined dynamic priority scheduling. In a comparison with a sorting-based approach, we reveal the advantages of hierarchical bucket queuing over previous work. Finally, we demonstrate the benefits of using priority scheduling in real-world applications by example of path tracing and foveated micropolygon rendering.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing

1. Introduction

While advances in semiconductor fabrication continue to follow Moore's Law, increases in clock speed have been stagnant due to physical limitations. Instead, hardware and software have turned towards parallelization as an answer to the ever growing demand for more compute power. The graphics processing unit (GPU) has evolved into a hardware architecture which offers an extreme level of parallelism. Execution on the GPU is, however, dictated by a simple, hardwired first-in, first-out (FIFO) scheduler which cannot be influenced dynamically. Thus, the GPU is commonly used as a co-processor for large, homogeneous workloads controlled by the central processing unit (CPU). For many applications which depend on sophisticated scheduling to, *e. g.*, dynamically adapt power usage, enable fair resource sharing, or meet real-time constraints, the GPU is not considered a viable option.

The static nature of GPU execution is a severe limitation even in its core application of interactive graphics. For example, virtual reality applications demand low-latency, real-time rendering for high-resolution, head-mounted displays (HMD). Long latencies or spikes in rendering time can lead to severe discomfort. According to current beliefs [Hun15], foveated rendering—adaptively rendering regions around the user's fixation point in higher resolution—is one way to address these issues. However, using a traditional rendering API, one can only rely on predictions about how long rendering might take to decide upfront on an allocation of compute power that might meet a given deadline, without guarantee. One solution could be a progressive renderer with dynamic priorities. It could adaptively focus compute power around the current fixation point. Rendering can stop, as the deadline of acceptable latency is reached, maximizing the quality achieved within a given time frame.

Using a current rendering API such as OpenGL or Direct3D, commands are simply streamed to the GPU and executed more or less in order. Similarly, compute jobs (*kernels*) dispatched to the GPU using a compute API, such as OpenCL [SGS10] or CUDA [Nvi08], are inserted into opaque queues, from which they are launched on the GPU. Once a job has been dispatched, it cannot be modified and must run to completion. This regime only permits very coarse-grained control of execution on the GPU. The lack of fine-grained control led researchers to start looking for ways to work around the rigid hardware scheduling. By launching a large kernel in what is often called a *persistent threads* approach, all compute units can be occupied with a single program, which can manually draw work from a software queue [AL09]. However, the focus of virtually all work in this area so far is on dynamic load balancing, while the idea of priority scheduling has received little to no attention.

Although scheduling for the CPU is well researched, strategies that work on the CPU hardly translate to the GPU, due to the vast architectural differences. Computation on the GPU is based on single instruction, multiple data (SIMD) execution of small groups of threads called *warps*. A kernel is launched as a grid of *blocks*, with each block consisting of the same number of warps that all fit on one of the GPU's multiprocessors. Scheduling a task on the GPU means mapping it to blocks or warps. A large number of tasks is typically required to provide enough work to make efficient use of the massively parallel GPU. At the same time, the execution time of each task is kept short to avoid branch divergence. Memory access is particularly costly on the GPU. Moreover, there is no support for task preemption and thread context restoration in software.

The sum of these facts makes priority scheduling a difficult problem on the GPU: There is a large number of short tasks to be managed, thus, scheduling decisions must be made very often. Making scheduling decisions involves reading information about the execution state from memory, which is very costly. Due to the massive parallelism and short task durations, tasks enter and leave the scheduling system in parallel at a high rate. Constant reorganization of shared data structures such as sorted queues or heaps is problematic, as locking must be avoided. Due to the nature of execution on the GPU, a priority queue in the original sense is not even a theoretical possibility as there is no absolute order that could be established on the large number of concurrent operations carried out at any given point in time. Thus, priority scheduling can only aim to execute tasks with higher priority before tasks with lower priority *on average*.

However, even a relaxed priority scheduling approach can enable a variety of applications and needs to overcome the aforementioned issues. We tackle them with our approach based on a hierarchical organization of buckets and make the following contributions:

- We present a flexible, hierarchical queuing structure for the GPU that can be configured to implement a variety of scheduling policies and is efficient for massively parallel access.
- We expose our bucket queues through a scheduling control model consisting of three simple entry points that allows for an easy and efficient implementation of different scheduling policies. Our model can be plugged into any persistent threads solution and would also lend itself to implementation in future hardware.
- We investigate different methods to implement fair-scheduling, earliest-deadline-first scheduling, and user-defined scheduling policies on top of our approach and compare the performance of each policy in a set of synthetic tests.
- We show how priority scheduling can be used for guaranteed-latency, foveated micropolygon rendering as well as adaptive path tracing.

2. Related Work

External GPU Priority Scheduling Recent GPU architectures can execute multiple kernels concurrently [Nvi12] given that sufficient resources are available. This feature relies on multiple work queues situated on the GPU, each feeding all available multiprocessors. While priorities are not supported for these queues, future architectures might add them, as outlined in the provisional OpenCL 2.1 standard [KG15]. These priorities will likely be static and only applicable to entire kernels, unlike our approach, which supports fine-grained priorities.

For current architectures, a variety of external priority schedulers exist. They can be viewed as an extension to the driver, controlling which kernels are forwarded to the GPU at which point in time. Timegraph [KLRI11], for example, delays and reorders kernels sent to the GPU according to deadlines. Further, the use of a single global priority queue [EA12] or a directed acyclic graph [MLH*12] has been explored, as has been the possibility to include multiple CPU and GPU nodes [WWO14]. The major issue with these approaches is that they consider jobs as large, opaque entities. Memory transfers [KLK*11] as well as kernels [BK12] can be split into smaller jobs to reduce the time until the GPU is responsive again. By scheduling blocks instead of entire kernels, using knowledge about how many multiprocessors are available, a better real-time scheduling performance can be achieved [LAF14]. Yet, all such approaches influence the scheduling only indirectly, which leads to several disadvantages: a long delay between making a scheduling decision and its effect on the GPU, the need to synchronize CPU and GPU, and the inherent inefficiency of submitting kernels too small to fully occupy the GPU. Additionally, dynamic priorities and work generation on the GPU are not supported.

Dynamic Scheduling on the GPU Due to the limited possibilities for influencing GPU execution from the CPU, software scheduling on the GPU itself received attention. Cederman et al. [CT08] implemented a first software scheduler based on work queues on the GPU. At about the same time, the term "persistent threads" was coined [AL09], which describes the idea of performing software scheduling on the GPU by occupying all available multiprocessors of with worker threads. These worker threads draw new work from custom work queues and finish only once all work has been completed. Work queues can be made to allow work to be inserted from the CPU [CVKG10], and support work stealing [CGSS11] and work donation [TPO10]. Persistent threads approaches in combination with work queues have been used in a variety of application areas: sparse matrix-vector multiplication [BG09], sorting algorithms [SHG09], scan algorithms [Bre10], and construction of kd-trees [VHBS16].

While all the approaches above show that a variety of problems can be tackled using software solutions, none of them address the

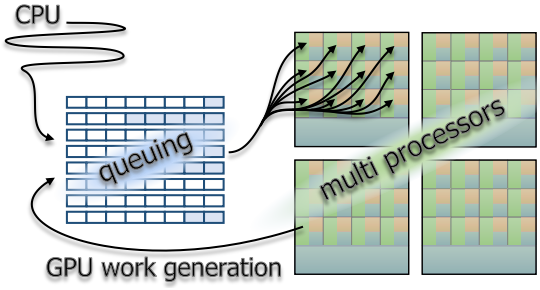


Figure 2: To keep the multiprocessors of the GPU busy, work is usually managed in queues located on the GPU. These queues can be filled by the CPU as well as the GPU.

problem of priority scheduling. Scheduling is mostly interesting in combination with multi-tasking. To add multi-tasking to a persistent threads approach, all tasks have to be compiled into one large *megakernel* [Har04]. While there are certain downsides to megakernels [LKA13], they are often outweighed by better scheduling and the potential to exploit data locality [SKB*14]. Persistent threads megakernels have been used in the Optix raytracing framework [PBD*10] as well as the dynamic task scheduling frameworks of Softshell [SKK*12] and Whippletree [SKB*14]. These schedulers are probably the approaches closest to ours. While not their main focus, they provide some ways of prioritizing workloads on the GPU: Softshell uses a monolithic queue, which can be progressively sorted, slowly moving high priority tasks to the front of the queue. In our evaluation, we demonstrate that the reaction time to high priority tasks is rather long, when only sorting progressively. Whippletree uses multiple queues to concurrently schedule warp and block level tasks as well as collect tasks of the same type. At the same time, Whippletree considers priorization of these queues, which can be used to, *e. g.*, keep queue lengths short during pipeline execution. Our hierarchical bucket queuing can be seen as a superset to the Whippletree and Softshell scheduler and supports significantly more sophisticated scheduling behaviors.

The main limitation of current approaches for dynamic scheduling on the GPU is the lack of facilities for preemption. While preemption is to be considered very costly on the GPU, there have been proposals to add it to the hardware by either saving the entire thread state or waiting for the current block to finish [TGC*14]. To reduce the cost of preemption, already computed results can be partially or entirely discarded, and blocks simply be restarted later [PPM15]. Our work is complementary to these extensions. We do not consider preemption and instead rely on tasks completing in a sufficiently short amount of time. However, our scheduling strategies could be extended to take advantage of preemption.

3. GPU Priority Scheduling

Work queues are widely accepted as the standard tool for task management on the GPU. They are used by software schedulers and the GPU hardware scheduler, as shown in Figure 2. They can be filled by the CPU and also directly by kernels executing on the GPU [Jon12]. Work queues are further useful as they allow for work aggregation [SKB*14]. The goal of our work is to design a priority scheduling scheme that can integrate with and enhance these existing practices. We identify the following set of requirements a queueing system must fulfill to achieve this goal:

R1 While one application might require a single queue and permit mixing different tasks, another application might need multiple queues to aggregate tasks of different types.
Thus, priority scheduling mechanisms and the queuing back-end must be customizable.

R2 When a multiprocessor requests work, the GPU-wide scheduling must not block, as this would halt other multiprocessors concurrently requesting work.
Consequently, a separate, serial priority scheduler between queues and multiprocessors cannot be used.

R3 Current strategies for GPU scheduling allow tasks to be generated and inserted into the queues at any point by any thread on the GPU. To avoid stalls, producers must be able to add work in parallel without interference.
Hence, complex data structures that require locking whenever a new task is generated are not an option.

R4 Peak performance can only be achieved if a multiprocessor does not stop executing tasks unless all tasks have been processed.
Thus, execution must not be halted for priority scheduling, i. e.*, it is not possible to use periodic rebuilds of data structures or very complex scheduling algorithms.*

3.1. Hierarchical Buckets

To fulfill these requirements, we propose the use of multiple instances of an efficient queue, organized into a hierarchical structure of buckets. By making the hierarchy configurable by the application, it is possible to cover a variety of scenarios. Every node of the constructed hierarchy can have an arbitrary number of children, *i. e.*, it is possible to place any number of buckets within another bucket. Leaf nodes in the resulting tree, *i. e.*, the final buckets, correspond to queues holding tasks. Buckets can be limited to certain types of tasks, allowing each queue to be optimized for the subset of data types it has to hold. Adding an element to a bucket queue hierarchy corresponds to walking down the tree until a leaf node is found and pushing into the corresponding queue.

To demonstrate the merit of such a configurable system, we have provided several examples in Figure 3: (a) The simplest configuration corresponds to a single bucket containing all types of tasks, which is analogous to previous-generation GPU command queues and simple persistent threads approaches, as well as Softshell [SKK*12]. (b) A one-level hierarchy with multiple child buckets all taking the same task type corresponds to current-generation GPU work queues. (c) A one-level hierarchy where every bucket only accepts tasks of a single type corresponds to the behavior of Whippletree [SKB*14]. (d) By using a two-level hierarchy, the first level can implement a discrete set of priorities, while the second level can collect tasks of different type. This way, it is possible to combine priority scheduling and work aggregation. (e) In a more complex setup, tasks for two different processes can be stored (P0 and P1). While P0 enqueues all tasks indiscriminately, P1 supports organizing tasks with different priorities. Note that in this example, certain priorities are only assumed by particular tasks (high: red and

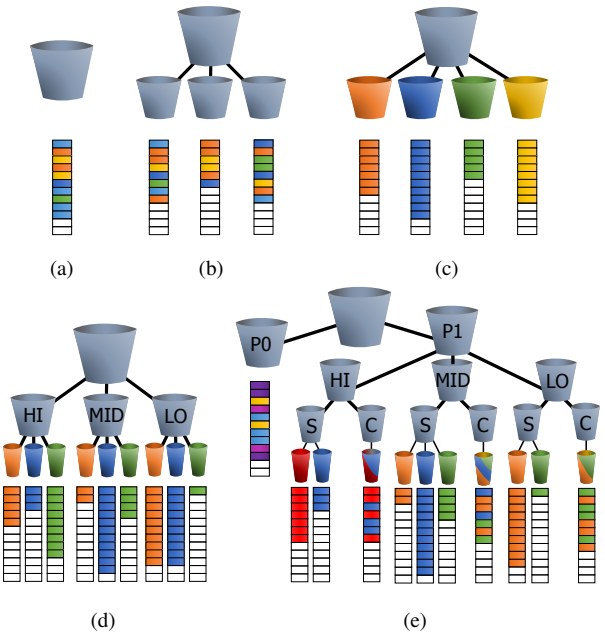


Figure 3: Hierarchical bucket queues can (a,b) capture simple setups similar to the work-dispatching mechanisms found on the GPU, (c) collect tasks according to their type, (d) combine prioritization with work aggregation, and (e) capture complex setups. Note that colored buckets can be specifically optimized for the subset of datatypes they are supposed to hold.

blue, mid: orange, blue, green, low: orange, green). At the lowest level, the queues can distinguish between small tasks (S) that need to be aggregated to fully utilize a multiprocessor and larger tasks that can be stored in a combined queue (C) with other task types.

3.2. Customizable Priorities

While the fundamental concept of a queuing hierarchy enables a variety of applications in itself, we need to define an efficient and easy-to-use scheme for establishing priority scheduling on top of it. Taking into account **R2**-**R4**, we propose a configurable, lightweight priority scheduling model based on the observation that a persistent threads megakernel fed from a queueing hierarchy enables the following three forms of scheduling:

1. When a new task is generated, the bucket hierarchy is traversed and the appropriate queue to store this task in can be selected.
2. When a multiprocessor finishes a task (or a set of tasks), the bucket to retrieve the next task from can be selected.
3. During execution, a small number of maintainer threads can update the queues by reorganizing elements in the background.

Given these possibilities, a variety of scheduling strategies can be realized. Being able to choose one of several queues during enqueue allows for (discrete) sorting according to arbitrary criteria. Being able to choose which queue to dequeue from allows the system to make decisions based on the current execution state, which might have changed since the tasks were enqueued. For example, tasks could be sorted into queues according to which process they belong to when they are generated. As a working block dequeues the next task, it could choose according to how much processing power each process has consumed in the meanwhile to, *e. g.*, achieve fair scheduling. In addition to these fundamental control mechanisms, dedicated maintainer threads can be deployed to continuously adjust the order of tasks within queues according to a constantly changing metric.

With our approach, we imagine all three forms of scheduling to be freely programmable, similar to the way shaders bring programmability to a graphics pipeline: before execution, task types are defined, the bucket queue hierarchy is set up and user-defined callback functions that implement each type of scheduling decision are registered. These functions are called whenever (1) a new task is generated, (2) a multiprocessor requests new data, or (3) during queue maintenance. Our current implementation employs a persistent threads megakernel approach similar to Whippletree [SKB*14], using their basic megakernel with our own scheduling mechanism added in. We also use their basic queue implementation in the leaves of the bucket hierarchy. Note that any other persistent threads approach and queue could be used instead.

3.3. Enqueue

Whenever a new task is generated, we want to enqueue it efficiently, while still allowing for control by the application. Given that the bucket hierarchy is known at compile-time, enqueue can be completed with a single traversal of the hierarchy. Callbacks registered with every bucket in the hierarchy are called during traversal of the tree to decide which sub-bucket to choose. When a leaf node is reached, we enqueue the task into the corresponding queue.

Consider the example in Figure 3e: The callback for the first bucket returns which process the task belongs to. In case the task comes from P1, the next callback determines if it is of high, medium, or low priority, before deciding if the task is large enough to be stored in a combined queue or if it should be merged with others of the same kind. Note that we could also only provide a single callback determining the final queue. However, enforcing a hierarchy makes it easy to reuse and extend an existing scheduling policy. For example, if another process with a different set of queues is added, it will be represented by a new branch under the first bucket, and no changes to the remaining parts of the hierarchy are needed.

As long as the underlying queue implementation supports concurrent enqueue operations, our priority scheme fulfills **R3**. In case that there is no space available in the queue, there are different possibilities to recover. We can walk back up the hierarchy while executing the callbacks with a limited set of possibilities, return that the enqueue failed due to lack of free storage, or keep retrying the enqueue operation until space becomes available. Since the first option is difficult to implement for the user and the third option bears the risk of deadlocking, we opt for the second option and consider the case of a failed enqueue an exception that the application has to handle.

Tasks can also be generated from the CPU, *e. g.*, via a traditional kernel launch, or an initial set of tasks that initiate a more complex dynamic algorithm. In this case, we can either traverse the bucket hierarchy on the CPU and transfer the tasks to the appropriate queues

on the GPU, or launch a kernel where each thread is responsible for generating and enqueuing one or more tasks. While the first option can generate tasks even while the persistent megakernel is running [CVKG10], the second option allows to handle multiple tasks in parallel. As the first option usually involves a performance overhead and the Whippletree programming model only supports the second, we also limit our implementation to that case.

To optimize the traversal process, we skip the evaluation of any callbacks if there is only a single viable choice, *e. g.*, a multi-bucket setup where each bucket is constrained to a specific task type (compare Figure 3c). Even if callbacks are evaluated, their execution is usually very efficient, since the task's payload will normally have already been loaded into local memory and no additional memory transactions are required. If the underlying system were to be implemented in a hardware scheduler, we would still expect the callbacks to be evaluated on the compute cores of the GPU. As task generation (or kernel launches via Dynamic Parallelism) happens during kernel execution, the thread generating the task can immediately perform the traversal.

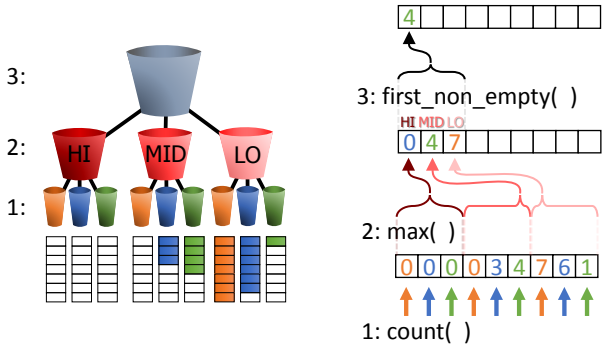


Figure 4: Dequeue example: (1) the callback for the leaf buckets returns the number of elements in the queue; (2) the callback for the inner buckets selects the queue with the highest element count; (3) the root bucket selects the child with the highest priority reporting a non-empty queue. In this example, dequeue would choose the green bucket with MID priority.

3.4. Dequeue

Similar to enqueue, we expect callbacks for dequeue to be registered with every bucket. When a multiprocessor finishes its previous work and requires new tasks to be dequeued, a possible solution would be to traverse the hierarchy top-down. However, empty queues are to be expected a common case during dequeue, especially with a large number of buckets, prolonging the search for available tasks. Furthermore, due to the SIMD nature of execution on the GPU, there is always at least a full warp of threads available when fetching new tasks. Hence, we can make use of these otherwise idle threads to implement a parallel bottom-up traversal for dequeue: every thread starts at a different leaf and walks up the hierarchy. At each bucket, the information coming from each of its children is combined by the respective callback until the final scheduling decision is computed at the root. This whole process basically corresponds to a parallel reduction.

Once the queue to dequeue from has been chosen, we compute the number of tasks to be fetched such that the currently available threads on the multiprocessor all receive sufficient work. Next, we try to dequeue the respective number of tasks from the queue. However, other worker blocks may have consumed all tasks from that queue during the time spent in traversal. In this case, we restart the dequeue process and mark the now empty queue. We found this trial-and-error approach to be much more efficient than locking queues during traversal (**R2**).

The bottom-up approach is not only more efficient when dealing with a large number of queues, but turns out to be a good approach for implementing many more complex scheduling strategies in general. Consider two single-threaded task types, each having its own bucket—one high priority, the other low. Assume that there is one task in the high priority queue and several in the low priority queue. A sophisticated scheduler might want to choose to execute multiple low priority tasks instead of a single high priority task, as it can make better use of the available processors. If we used a naïve top-down traversal, it would be difficult to implement such a behavior across multiple hierarchy levels, as this would require descending into every leaf and keeping track of all the results computed along the way. With the bottom-up approach, however, such behavior can be achieved in a very simple and natural way.

If this bottom-up scheme were to be implemented in hardware, we would also consider performing the traversal on the compute units of the GPU, since that would make the full instruction set available to the callbacks. Considering that some warps usually finish before others, there is potential to hide the added latency from the evaluation of the callbacks, by having the first warp to complete the previous task immediately start evaluating the dequeue callbacks. This is likely to lead to new tasks already being available as soon as the remaining warps complete.

3.5. Maintain

While a large number of important scheduling strategies can be implemented by enqueuing and dequeuing logic alone, there are cases that require changing the order of elements already in queues, *i. e.*, sorting the queues. While fully sorting the queues is not a viable option (*cf.* **R4**), Steinberger et al. [SKK*12] showed that progressive sorting can be used to gradually rearrange queue contents in a non-blocking fashion. We adopt this approach, allowing each queue to be flagged for automatic maintenance by registering a callback for it. Given a task that is currently stored in the queue, the callback must return a numerical priority value that can be used for sorting.

If any queue is marked for maintenance, we dedicate a configurable number of GPU threads to continuous sorting. Instead of sorting the entire queue at once, only tasks within a limited sorting window are considered at each point. A sorting pass progressively advances the sorting window from the back of the queue to the front, as proposed in the original approach [SKK*12] and outlined in Figure 5. Once a sorting pass is finished, sorting is restarted at the back of the queue. To avoid stalling dequeue operations, we uphold a safety margin to the very front of the queue. Multiple queues are handled by simply cycling through them. Some queues,

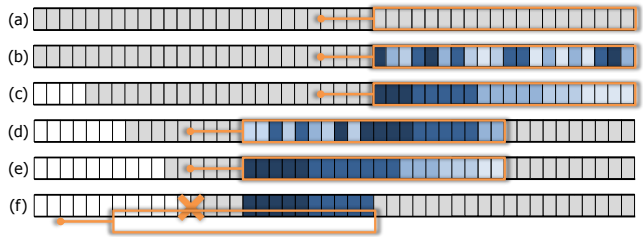


Figure 5: Progressive sorting of a queue using a sorting window (orange) with safety margin to the front, while tasks are concurrently removed from the queue; time steps (a)-(f): (a) for all tasks in the sorting window, the callback delivers priorities (dark blue to white); (b) the returned values are sorted locally; (c) tasks are exchanged according to the local sorting; (d) the sorting window is advanced to the front; (e) the next segment is sorted; (f) the safety margin reaches the front of the queue, sorting cannot continue without stalling the execution and must be canceled and restarted at the back.

however, might require more attention than others. Therefore, we record the actual number of exchange operations carried out during one sorting pass and the number of new elements received since last restarting the sort. Using these values, we prioritize the sorting of those queues which we expect to contain the highest number of unsorted elements.

In practice, we reserve a single GPU worker block to be used as a dedicated maintainer. For an integration with a hardware scheduler, a programmable unit would be required if the callback needs to be reevaluated during each sort, *i. e.*, if priorities of queued tasks are allowed to change. In this case, we would suggest assigning a small block of threads to maintenance, similar to the current software approach. However, if priorities are static, dedicated maintainer threads can be avoided. Instead, the priority of each task can be computed when it is enqueued and stored alongside the task. The priorities can then be read from memory and sorting can be implemented by a hardwired unit instead.

3.6. Application Programming Interface

Since we use Whippletree's execution model, our API builds upon its template-based CUDA/C++ interface. In Whippletree, a task can be defined as follows (slightly simplified):

```
struct Task {
  static const int NumThreads;
  typedef int Payload;
  __device__ static
  void execute(int tid, Payload& data);
};
```

When chosen for execution, the task's `execute` method is called by the requested number of threads, all receiving the dequeued payload as input.

Following the spirit of a C++ interface, the bucket hierarchy and callback functions are also set up using templates in the API we provide. Both buckets (`Bucket`) and queues (`Queue`) expect a template class that specifies the callback functions:

```
template<class LeafCB, class... Tasks>
struct Queue;

template<class BucketCB, class.. Children>
struct Bucket;
```

For leaf nodes, two callbacks can be provided:

```
struct LeafCallback {
  template<class Queue>
  __device__ static
  CustomType checkLeaf(const Queue& q);

  template<class Task>
  __device__ static
  Comparable maintain(Task::Payload& data);
};
```

The `checkLeaf` callback is called during dequeue and provides the information that is propagated up the hierarchy. The `maintain` callback is optional. Only if it is present will the maintainer attempt to sort the queue. Note that the return value of this method can be chosen freely, the only requirement is that a suitable comparison operator exists.

The bucket's `traverse` callback for enqueuing and its `propagate` callback for dequeuing have the following signature:

```
struct BucketCallback {
  template<class Task>
  int traverse(Task::Payload& data);

  int propagate(CustomType* infos);
};
```

Note that `traverse` itself is a template and can be specialized for different tasks. Also note that `CustomType` can be of any type as long as it is compatible with the return type of `checkLeaf`.

Once a user has set up the callback functions, the queuing hierarchy can be established. The following example shows how Figure 3d can be defined with just a few lines of code, using the callback definitions `LeafHasData`, `RoundRobin`, and `Discrete`, which we discuss in the upcoming section:

```
typedef Queue<LeafHasData, Task1> Queue1;
typedef Queue<LeafHasData, Task2> Queue2;
typedef Queue<LeafHasData, Task3> Queue3;
typedef Bucket<RoundRobin<3, AlwaysFirst>,
  Queue1, Queue2, Queue3> B3;
Bucket<Discrete<3>, B3, B3, B3> Root;
```

Note that each queue only stores the payload for a single task type. `B3` defines a bucket that has three queues as children. By adding `B3` three times to the root node, three instances of `B3` are created as immediate sub-buckets of the root.

Given the bucket hierarchy root node, our implementation generates the scheduling logic from the user-provided callbacks and combines it with the task `execute` functions into a megakernel. If at least one maintainer callback is provided, additional routines are added to the megakernel for turning the block with id '0' into the maintainer upon kernel launch.

4. Scheduling Policies

We now show how the discussed queuing framework can be brought to use in practice. First, we demonstrate how simple scheduling mechanisms, *e. g.*, discretized prioritization and round-robin can be set up with bucket queues. After that, we focus on more advanced examples such as fair scheduling, earliest-deadline-first, and user-defined policies, and compare our results to previous work. For our evaluation, we used an Intel i7-4771 with 16GB RAM running Windows 10 and an NVIDIA Geforce GTX 980Ti.

4.1. Discretized Priorities

A simple way to build priority scheduling is using a fixed number of buckets, each for a different priority. Whenever a new task is created, its priority value is computed and it is inserted into the appropriate bucket. During dequeuing, available buckets are probed in descending order of priority. Assuming priorities, *e. g.*, in the range $[0,1)$, we discretize them linearly according to the overall number of buckets (see Figure 6). The corresponding callbacks are straightforward to set up:

```
template<int NumChildren>
struct Discrete {
  template<class Task>
  __device__ static
  int traverse(Task::Payload& payLoad){
    return payLoad.priority * NumChildren;
  }
  __device__ static
  int propagate(bool* infos) {
    for(int i = NumChildren-1; i >= 0; --i)
      if(infos[i])
        return i;
    return 0;
  }
}
```

```
struct LeafHasData {
  template<class Queue>
  __device__ static
  bool checkLeaf(const Queue& q){
    return q.count() > 0;
  }
};
```

`traverse` discretizes the priority and returns the id of the bucket with appropriate priority. `checkLeaf` determines if data is available in the queue, and `propagate` runs through the buckets in descending order of priority, choosing the first non-empty bucket.

4.2. Round-Robin

Another common strategy that is straightforward to implement is per-multiprocessor round-robin. Consider a setup with one root bucket and an arbitrary number of sub-buckets that should be executed in a round-robin fashion. Every multiprocessor stores the id of the last sub-bucket chosen for dequeue in shared memory. During each invocation of the propagate method, the id is incremented to identify the next bucket for dequeuing.

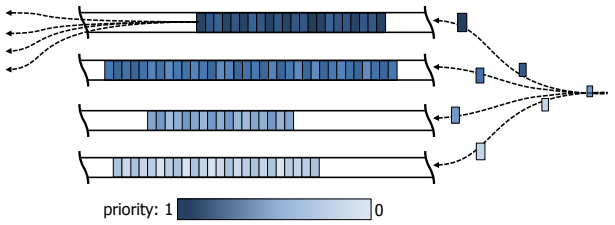


Figure 6: Discretized priorities can be configured with very simple callback functions. During enqueue, the appropriate bucket is chosen, while dequeue takes tasks from the bucket with the highest priority that is not empty.

```
template<int NumChildren, class Traverser>
struct RoundRobin : public Traverser {
  __device__ static
  int& getLast(){
    __shared__ int last;
    return last;
  }
  __device__ static
  int propagate(bool* infos){
    int &next = getLast();
    next = (next + 1) % NumChildren;
    for(int i = 0; i < NumChildren; ++i){
      if(infos[next])
        return next;
      next = (next + 1) % NumChildren;
    }
    return 0;
  }
};
```

Note that the Round-Robin class requires another class as template argument that is supposed to provide the `traverse` method. Deriving the scheduling policy from a custom template facilitates the reuse of existing Round-Robin mechanics for dequeue and mixing them with any kind of enqueue policy. The initial queue is chosen randomly through additional operations that have been omitted here for the sake of brevity. For better control, we support an optional initializer method that is called right after kernel launch.

4.3. Fair Scheduling

In a system that supports the execution of multiple processes, one common goal is to provide a fair scheduler to assign an equal amount of processor time to each process. A process can either be an individual task that is respawned multiple times, or an entire group of different tasks that are capable of instancing each other. We consider two solutions for fair scheduling with our framework, by either using multiple separate buckets or a single sorted bucket.

Separate Buckets A first way to implement fair scheduling is by using separate sub-buckets that are pooled by a fair-scheduling root bucket. To implement this concept, we associate a counter with each sub-bucket and record the total time that processes from this bucket have consumed so far. During dequeue, the fair scheduling bucket

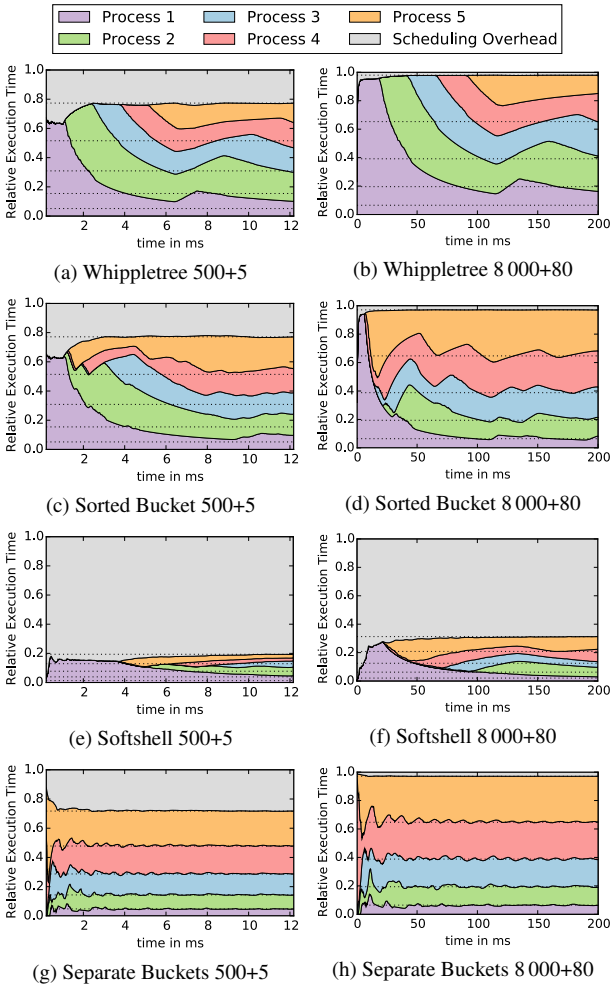
selects the child whose counter is currently lowest. Keeping track of the total time consumed allows us to either assign equal compute times to all buckets or enforce predefined target quotas for individual processes.

Sorted Bucket Alternatively, quota-driven fair scheduling can also be set up by utilizing the queue maintainer. Mixing all tasks in the same queue, we can simply use the deviation in runtime from their desired target quota as sorting criterion. However, as sorting the queues takes time and the priorities are constantly changing, scheduling might significantly lag behind.

To measure the time spent on each task, we queried the built-in cycle counter present on each multiprocessor before and after executing a task. Note that this measure is not guaranteed to capture the exact time a task was actually executing instructions, as the hardware warp scheduler switches between all warps present on a multiprocessor based on their ready state. Thus, all tasks being executed on the same multiprocessor can influence the execution time measure of each other. If a more accurate time measurement is needed, the megakernel can be configured to use a single, large thread block per multiprocessor. Whippletree can then make uniform scheduling decisions among the entire multiprocessor, and all tasks executed concurrently belong to the same bucket. In this case, the consumed clock cycles capture the time an entire multiprocessor was assigned to a certain process and the measurements can be considered fair.

Evaluation To evaluate both fair scheduling implementations and compare our approach with previous work, we set up five processes and launched 1 000 initial tasks for each. Each task was primed to execute a random number of fused-multiply-add (FMA) instructions and memory (MEM) operations to simulate diverse workload characteristics. In order to evaluate how behavior is influenced by the overall rate at which scheduling decisions need to be made, we set up two scenarios to generate different per-task loads: 500 FMA + 5 MEM and 8 000 FMA + 80 MEM on average. After executing, each task immediately enqueued a copy of itself to ensure that the system remained fully occupied. We recorded 10 000 scheduling decisions, capturing the time spent on each process. To measure the overhead of scheduling, we executed the same tasks with conventional CUDA kernel launches and recorded the difference to the average task throughput. We compared bucket queuing with Whippletree [SKB*14] without scheduling and Softshell [SKK*12] with priority sorting as references. The results are shown in Figure 7.

Results obtained from Whippletree indicate that handling lightweight tasks with 500 FMA and only a few memory transactions provide a challenging scenario for dynamic scheduling approaches. The overhead of storing and fetching task payloads in a queue led to a slowdown of about 20% in comparison with CUDA. However, in the 8 000+80 scenario, the overhead became negligible. Whippletree follows a simple FIFO approach and thus did not consider the desired quotas (dotted lines), assigning processing resources to one task after the other. Using the maintainer (Sorted Bucket) shows that progressive sorting caused hardly any overhead compared to Whippletree (less than 2%), which is not surprising, considering that only one out of 88 employed thread blocks was used for sorting on the Geforce GTX 980Ti. In the 500+5 scenario, sorting did not



(a) Whippletree 500+5 (b) Whippletree 8 000+80

(c) Sorted Bucket 500+5 (d) Sorted Bucket 8 000+80

(e) Softshell 500+5 (f) Softshell 8 000+80

(g) Separate Buckets 500+5 (h) Separate Buckets 8 000+80

Figure 7: Quota driven scheduling with target time quotas (dashed lines) of 7%, 13%, 20%, 27% and 33%. Efficacy and overhead of our framework are compared against Softshell and Whippletree for reference. While separate buckets can quickly adjust the scheduling to match the desired quota, sorting takes significantly longer and oscillates around the targets.

meet the quotas within 10 milliseconds, since tasks were consumed too quickly from the front of the queue to enable thorough sorting in the back. In the 8 000+80 scenario however, scheduling converged after approximately 100 milliseconds with noticeable oscillations around the target quotas. Softshell uses dynamic memory allocation for the payload and a hash map to combine payloads. Thus, its overhead is immense in comparison to a simple CUDA kernel (only 20% and 25% achieved throughput). However, as expected, its scheduling strategy (although slower) behaves similar to the sorting implemented by the maintainer. Using separate buckets clearly achieved the best scheduling behavior in the examined scenarios. In comparison with Whippletree's FIFO execution, a slight increase in scheduling overhead could be observed in both scenarios, which we ascribe to the additional effort of checking multiple queues during dequeue.

4.4. Earliest-Deadline-First

Earliest-deadline-first is a common strategy in hard real-time scenarios, where all processing power is dedicated to the job with the closest deadline. Using our bucket queuing framework, we find multiple ways to implement earliest-deadline-first scheduling. As our first approach, we can set up a maintainer that sorts tasks according to the deadline of their associated job (Sorted). Second, we simply use separate buckets for each job and always choose the job with the earliest deadline (PerJob). Third, given an application that takes a known, finite amount of time to run, we can discretize the entire runtime into buckets (Discretized). Hence, each bucket corresponds to a specific time frame of the program's execution and tasks can be enqueued accordingly, while dequeue will always draw from the bucket with the closest upcoming deadlines. All tasks within a bucket must be executed before its associated time frame passes to ensure that no task deadline is missed.

In many applications, tasks that failed to meet their deadline can generally be skipped. Thus, as an optimization of our third approach, we can still work in discretized time but instead keep a ringbuffer of buckets that only holds tasks up to a certain interval into the future (WrapAroundBuckets). As time progresses, the oldest buckets can then be reused for upcoming deadlines.

Evaluation To test earliest-deadline-first scheduling, we ran a controller block separate from the megakernel to periodically create tasks for six recurring task types at intervals between 1*ms* and 8*ms*. For each task type, between one and four tasks were recurrently created, each running between 0.1*ms* and 4*ms* to execute a mixture of FMA and MEM instructions. Each scheduling algorithm was evaluated by gradually increasing the number of tasks by up to a factor of 128 of the initial load and measuring how many tasks still completed within their deadline. For WrapAroundBuckets, we used a future window of 10*ms* and 256 buckets. Again, we compare overall behavior and performance to both Whippletree and Softshell. The results for these tests are shown in Figure 8.

Except for Softshell, all approaches managed to respect deadlines for low workloads. Unfortunately, Softshell's overhead was simply too high to cope with a load multiplier above 1. Tasks were issued periodically one interval's time before the deadline. Thus, their occurrence, to a certain extent, followed the deadline and simple FIFO scheduling (employed in Whippletree) with its lower overhead could keep up with the other approaches. However, at higher loads (starting at a load multiplier of about 30), differences emerged. Whippletree and Discretized did not (or could not) identify the job with the closest deadline and were the first approaches to miss deadlines. Soon after, PerJob and Sorted also dropped in performance. PerJob's drop is comparatively steep, which can be explained by the fact that PerJob still tries to schedule tasks that already missed their deadline. Also, PerJob showed a relatively high overhead, which we ascribe to the necessity of finding the queue with the earliest deadline to dequeue items. However, the average lateness increased similarly for Discretized, Whippletree, PerJob and Sorted. WrapAroundBuckets performed better than all other approaches, with comparatively high ratio of tasks on time and exhibiting a smaller increase in average lateness with increasing load. It is the only approach that clearly outperformed the simple FIFO scheduling implemented by Whippletree.

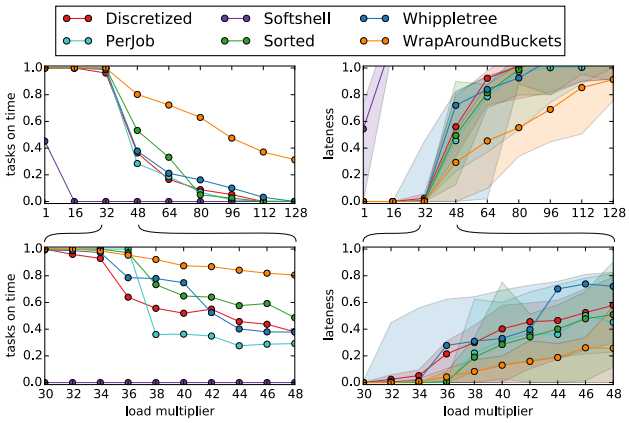


Figure 8: Earliest-deadline-first results for different scheduling implementations with details on the bottom. (left) The fraction of tasks executed on time by each method as the workload increases. (right) The average lateness shows by how much time on average tasks miss their deadline. The minimum and maximum lateness for each method is overlayed as well.

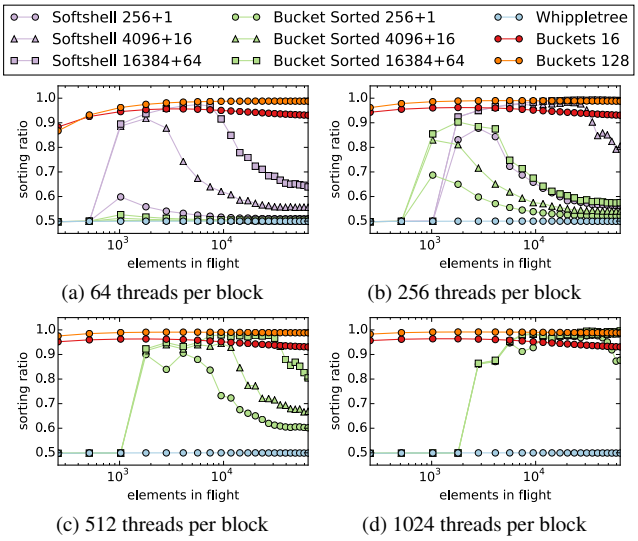
4.5. Application Defined Priorities

Finally, we evaluated the capabilities of bucket queues in applications that involve tasks with arbitrary priorities. A possible example for such a scenario can be given by any adaptive algorithm where the importance of a particular operation is difficult or impossible to predict in advance, *e. g.*, Reyes-style subdivision or adaptive image sampling. We compared the performance of queue sorting to discretized-priority buckets. In our test setup, we launched W initial tasks and assigned uniformly distributed, random priorities to them. Every task executed a variable workload of FMA and MEM operations and spawned another task with a random priority. This way, an average of W tasks were contained within the queuing structure at all times during the evaluation. We recorded the order in which tasks were executed, as well as their associated priority. The test was stopped as soon as N tasks had finished. We computed the achieved scheduling accuracy as

$$S = \frac{1}{N(W-1)} \cdot \sum_{i=0}^{N} \sum_{j=0}^{W} s_{i,j},$$

where i and j may represent any two tasks that were simultaneously queued for execution. $s_{i,j} = 1$ for tasks i and j, iff the task with the higher priority was chosen first; otherwise, $s_{i,j} = 0$. Note that the restriction to simultaneously enqueued tasks is necessary for a meaningful assessment, since a high-priority task n that was only generated after a low-priority task m finished could not possibly be processed before m. For random priorities, no scheduling at all leads to an expected accuracy of ~50%. If all elements are executed in the correct order, scheduling accuracy equals 100%. As multiprocessors execute in parallel, 100% accuracy may never be reached in practice.

Evaluation The measured scheduling accuracy is shown in Figure 9. As reference for comparison, we again include Softshell and Whippletree. Softshell and Bucket Sorted exhibited similar behavior. Queue sorting turned out to be ineffective under low and high



(a) 64 threads per block (b) 256 threads per block

(c) 512 threads per block (d) 1024 threads per block

Figure 9: Scheduling accuracy with varying task execution times (FMA+MEM Softshell and Bucket Sorted) and threads per block. Bucket queues either use 16 or 128 buckets.

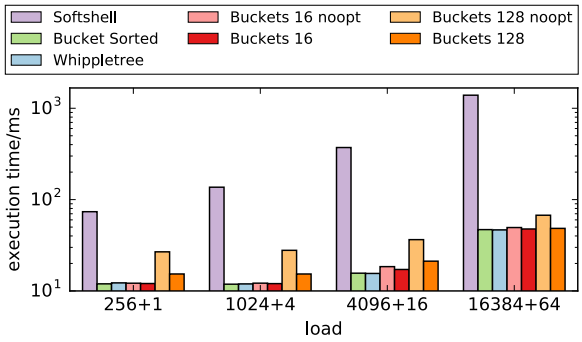


Figure 10: Softshell shows up to 10× the execution time of the other approaches. Bucket Sorted and Buckets 16 show only a small execution time overhead. Using a high number of buckets without our upward propagation optimization significantly increases execution time (up to 2×).

loads. With only a few elements in the queue, sorting was not able to start as there was no sufficient safety margin and thus the achieved accuracy was about 50%. On the other hand, with a high number of elements in the queue, progressive sorting was not fast enough to move high priority elements to the front. Consequently, sorting accuracy quickly deteriorated with rising number of elements in flight W. The comparably high scores recorded for Softshell are misleading; due to its immense scheduling overhead, a lot more time could be spent on sorting relative to time spent on task execution and thus higher accuracies were achieved for the same W. The performance of these two techniques is also dependent on the task duration, since long-running tasks (*e. g.* 4096+16 or 16394+64) are dequeued less frequently, and, thus, remain longer inside the queue and undergo additional sorting passes. In contrast, the accuracy of Buckets was not affected significantly by task runtime, nor by W.

We observed a slight drop in performance for very large W. Since discretized prioritization leads to fast consumption of high-priority tasks, a larger number of tasks accumulated in low-priority queues and could not be accurately distinguished. Overall, the error of bucket queues approximately equaled the expected discretization error. With 16 buckets, we achieved an accuracy of up to 94%; with 128 buckets, this figure rose to 99%. Bucket Sorted only achieved such high accuracies with more than 512 threads being used for sorting, and about 10 000 elements in the queue. Since Whippletree does not consider priorities in its scheduling at all, it yielded an accuracy of 50%. Softshell failed to run to completion for 512 and 1024 threads per block.

The impact on execution time is shown in Figure 10. The log-scale plot demonstrates that Softshell took roughly ten times longer than all other techniques to finish any of the test cases. Bucket Sorted again only added a small overhead to the execution time compared to Whippletree. As Buckets did not run a sorting algorithm, it conserved more bandwidth and processing power for task execution. However, with an increasing number of buckets, more time was spent on traversing the bucket hierarchy. Processing the hierarchy top-down with only a single thread (Buckets noopt) instead of bottom-up (see Section 3.4) more than doubled the execution time (256+1 and 4096+16). Using our bottom-up approach (Buckets), the overhead was significantly reduced, underlining the usefulness of this design choice in the first place. In this case, the 16 bucket version reached execution times on par with Whippletree and achieved a scheduling accuracy of up to 94%.

5. Use Cases

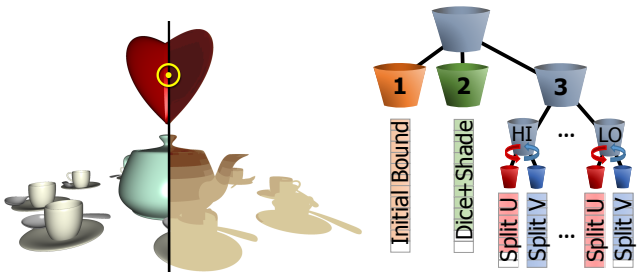
To show that hierarchical bucket queuing is applicable to rendering, we present two use case scenarios: micropolygon rendering and path tracing. For both applications, we assume soft real-time constraints.

5.1. Micropolygon Rendering

Virtual reality applications demand low-latency, high-resolution image synthesis, especially when using an HMD output device. Frame latency is a particularly sensitive issue in such applications and must be kept below a certain threshold. One potential way to address this problem would be to use a rendering method that allows gradual refinement of the image until the deadline for maximum tolerable latency expires, thus maximizing the image quality within a given timeframe.

One rendering technique that produces such gradual refinement is micropolygon rendering, as used in the Reyes pipeline [CCC87]. While Reyes-style micropolygon rendering has been implemented on the GPU before [TPO10, SKB*14], using our priority scheduling, we can now take advantage of foveated rendering [CCL02] to significantly improve image quality by prioritizing refinement around the user's fixation point.

Ideally, we want a smooth transition from fine to coarse patches and reasonable visual quality throughout the entire image to avoid popping artifacts during motion. For each patch, we evaluate its split priority based on its projected patch size and distance to the current fixation point, thus prioritizing large patches that are close to the fixation point. Dynamic priority scheduling is essential in this example,



(a) Foveated rendering of Tea scene (b) Micropolygon render setup

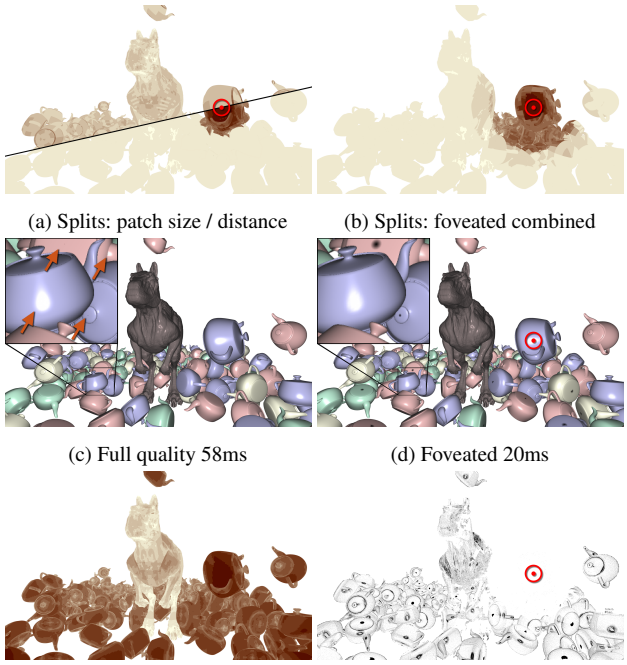
Figure 11: Foveated micropolygon rendering. (a) The color intensity on the right side indicates the degree of subdivision applied, which is highest at the fixation point (yellow ring mark) and gradually falls off with increasing distance. (b) Schematic visualization of the bucket queue setup to achieve the desired prioritization.

since we cannot predict the projected extents of all patches resulting from recursive splits in advance. Building on the Whippletree Reyes implementation [SKB*14], we set up a three-level bucket hierarchy, as shown in Figure 11b. The first bucket distinguishes between split tasks and all other task types. It prioritizes the execution of the initial bound tasks, shading and dicing over patch splits. This leads to an early creation of all initial split tasks and drains the pipeline of all intermediate data that is ready to be rendered. On the second level, split tasks are inserted into discretized priority buckets. Below each priority bucket, a round-robin scheduler switches between splits along the u and v direction, which are implemented as separate tasks. Before we enqueue patches to be split, we check if the procedure is likely to execute before the target latency is reached. If this is not the case, we stop the recursion and directly forward the patch to the dicing and rendering stages. For time measurement we again rely on the per-multiprocessor cycle counts, which we synchronize before each frame by writing their current state to a global buffer.

We test our approach with two animated scenes: Killeroo and Tea, shown in Figures 1 and 11, as well as our supplemental video. We chose a viewport with a resolution of 3840×2160. Conventional Reyes rendering of the Killeroo scene at full image quality takes between 30ms and 60ms, the Tea scene takes between 15ms and 22ms. With foveated rendering, we can limit the Killeroo scene to a guaranteed 20ms and the Tea scene to 10ms by adaptively generating full image quality only around the fixation point and lower quality in the remaining image. Figure 12 demonstrates the details of our approach for the Killeroo scene.

5.2. Pathtracing

While Monte Carlo pathtracing on the GPU is becoming increasingly popular, generating high-quality images quickly remains a challenging task due to the notorious noise, which is most prominent in the early stages of image synthesis. For implementations on the CPU, adaptive, priority-based solutions have been proposed to reduce noise and enhance image quality with a low sample budget, *e. g.*, by distributing more samples to image regions with a high estimated local error [HJW*08, ODR09]. Since the error estimate and the rate of convergence in an image region may change with



(a) Splits: patch size / distance (b) Splits: foveated combined

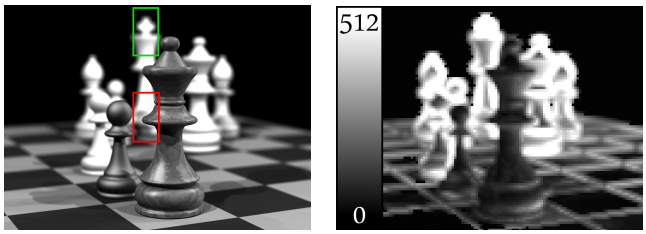
(c) Full quality 58ms (d) Foveated 20ms

(e) Splits: full quality (f) difference enhanced 200×

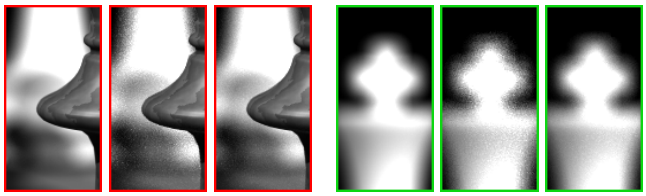
Figure 12: (a, top) Using the projected patch size as priority, the processing power is distributed evenly; (a, bottom) using the distance to the focus point instead leads to very localized refinement. (b) A combination of both creates a reasonable falloff. (c and e) A full-quality render pass produces high geometric detail, but consumes a considerable amount of time. Using foveated rendering, only the focus area (red ring mark) is rendered at full quality but image generation is sped up significantly. (f) Visualization of the difference between full quality and foveated rendering.

each new sample, adaptive sampling usually relies heavily on a continually maintained priority queue to always identify the region with the highest error estimate before casting new samples.

Porting the adaptive sampling approach to the GPU is non-trivial, since massively parallel execution generally provides little support for efficient and adequately sorted queues. However, with bucket queue scheduling, different prioritization schemes can efficiently be incorporated. As a demonstration, we implemented a path tracer for static scenes, with support for low-discrepancy sequence sampling, depth-of-field (DOF) and an arbitrary number of light sources. Our implementation uses a single task type, which casts four rays with a random number of maximum bounces for each pixel upon execution. To avoid write conflicts, each task is assigned to a dedicated 8×8 pixel region of the output image. On completion, each task enqueues a copy of itself, thus allowing the persistent thread blocks to constantly fetch new work for execution. This approach has been shown to achieve high occupancy on GPUs [AL09]. We force the persistent threads execution to pause every 50ms to display the current image via OpenGL. Using this one task in Whippletree establishes our base-line implementation, which, in practice, performs uniform sampling of the image domain.



(a) Chessboard scene with DOF (b) Sample distribution



(c) Mixed-focus region (d) Out-of-focus region

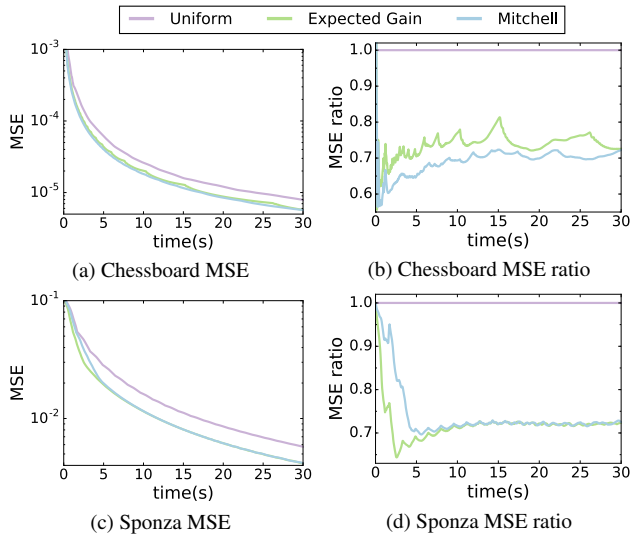
Figure 13: Comparison of uniform and adaptive sampling using the expected gain error estimate as priority. (a) Pathtraced chessboard scene with DOF and 9 light sources rendered at resolution 1024×750. (b) Sample distribution with prioritization after $4s$, brighter means more. (c,d) Comparison of the result of uniform and adaptive sampling after $4s$ to ground truth (2048 samples/pixel) in marked regions: left: ground truth; center: uniform; right: adaptive.

In order to enable adaptive sampling through prioritization, we set up a single-level hierarchy with 128 individual buckets. The priority of each task is computed based on an error metric and discretized into these 128 buckets. To evaluate our approach, we test two different per-pixel error metrics. First, we use the expected error reduction constructed around the Monte-Carlo error estimate $E \propto \sigma/\sqrt{N}$:

$$\Delta E \approx \frac{\sigma}{\sqrt{N}} - \frac{\sigma}{\sqrt{N+4}},$$

where σ is the current variance estimate over all N samples for a pixel so far. We call this error metric 'expected gain'. Second, we use the error metric devised by Mitchell [Mit87] for obtaining antialiased images. Per-pixel estimates are added up to obtain the local error estimate for a 8×8 region. Both metrics require at least two samples to compute an initial variance estimate. Thus, we set the priority of the initial tasks to the maximum, forcing four samples to be computed for each pixel before relying on priorities for further sampling.

Figure 1 and 13 outline the adaptive behavior when using the expected gain metric for sampling the scenes. At equal run times, our prioritized rendering clearly reduces noise in comparison with uniform sampling. In Figure 14, we show the development of the mean squared error (MSE) over time for the two prioritization schemes after the initial four samples have been computed. Adaptive sampling quickly reduces the MSE by up to 45% for the Chessboard scene and 35% for the Sponza scene when compared with uniform sampling. While both metrics perform equally well for low sample counts, the expected gain metric seems to perform slightly better at early frames, but loses its advantage after the initial noise has been cleared up.



(a) Chessboard MSE (b) Chessboard MSE ratio

(c) Sponza MSE (d) Sponza MSE ratio

Figure 14: Progression of the MSE during path tracing. (a, c) With priority scheduling, the MSE reduces more rapidly. (b, d) The ratio of the MSE relative to baseline uniform sampling shows that prioritized sampling reduces the MSE by up to 45% in early frames.

6. Conclusion and Future Work

Using hierarchical bucket queues on the GPU, it is, for the first time, possible to set up a variety of scheduling policies directly on the GPU. Bucket queues can be tailored to the needs of the application, not only for priority scheduling, but also to combine tasks or integrate multiple processes into a single scheduler. Bucket queues are easily configured by defining a small number of callback functions. They enable simple scheduling strategies, like FIFO or round-robin scheduling, as well as more complex strategies, like earliest-deadline-first or fair scheduling. According to our tests, priority scheduling using bucket queues always outperformed previous approaches based on sorting, making it the preferred choice on the GPU. We integrated our bucket queues into the Whippletree [SKB*14] framework and were able to show that even in challenging situations our approach only adds a small overhead (between 1% to 5%) compared to the original Whippletree. While being directly applicable to any persistent threads approach, our design also lends itself well to a potential implementation in hardware.

Rendering can also profit from priority scheduling as we have demonstrated for latency controlled, foveated micropolygon rendering and priority-driven pathtracing. Foveated rendering is enabled by the ability to direct the processing power to a focus region. With path tracing, prioritizing the image areas that are expected to reduce the image error the most can reduce the difference to the ground truth by up to 45% in comparison to uniform sampling. While priority-based rendering is certainly interesting for future virtual reality systems, it will require additional work to integrate priorities deeper into the rendering pipeline. Actually, we believe it is necessary to rethink the execution of rendering pipelines on the GPU, including dynamic sample distributions and adaptive geometry refinement. To this aim, we believe an experimentation framework for designing rendering pipelines is needed, including a priority scheduler similar to ours.

Acknowledgements

This work was partially funded by FFG (contract 849901) and EU FP7 (contract 611526), and supported by the Max Planck Center for Visual Computing and Communication. The chess scene in Figure 13 was first presented in the work of Hachisuka et al. [HJW*08] and was modeled by Wojciech Jarosz.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. High Performance Graphics* (New York, NY, USA, 2009), HPG '09, ACM, pp. 145–149. 2, 11

[BG09] BELL N., GARLAND M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 18:1–18:11. 2

[BK12] BASARAN C., KANG K.: Supporting preemptive task executions and memory copies in GPGPUs. In *Proc. Euromicro Conference on Real-Time Systems* (Washington, DC, USA, 2012), ECRTS '12, IEEE Computer Society, pp. 287–296. 2

[Bre10] BREITBART J.: Static GPU threads and an improved scan algorithm. In *Proc. Conference on Parallel Processing* (Berlin, Heidelberg, 2010), Euro-Par '10, Springer Berlin Heidelberg, pp. 373–380. 2

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 95–102. 10

[CCL02] CATER K., CHALMERS A., LEDDA P.: Selective quality rendering by exploiting human inattentional blindness: Looking but not seeing. In *Proc. ACM Symposium on Virtual Reality Software and Technology* (New York, NY, USA, 2002), VRST '02, ACM, pp. 17–24. 10

[CGSS11] CHATTERJEE S., GROSSMAN M., SBÎRLEA A. S., SARKAR V.: Dynamic task parallelism with a GPU work-stealing runtime system. In *Proc. Languages and Compilers for Parallel Computing* (Berlin, Heidelberg, 2011), LCPC '11, Springer Berlin Heidelberg, pp. 203–217. 2

[CT08] CEDERMAN D., TSIGAS P.: On dynamic load balancing on graphics processors. In *Proc. Symposium on Graphics Hardware* (Aire-la-Ville, Switzerland, 2008), GH '08, Eurographics Association, pp. 57–64. 2

[CVKG10] CHEN L., VILLA O., KRISHNAMOORTHY S., GAO G.: Dynamic load balancing on single- and multi-GPU systems. In *Proc. Parallel Distributed Processing* (2010), IPDPS '10, IEEE, pp. 1–12. 2, 5

[EA12] ELLIOTT G., ANDERSON J.: Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems 48*, 1 (2012), 34–74. 2

[Har04] HARGREAVES S.: Generating shaders from HLSL fragments. In *ShaderX3: Advanced rendering with DirectX and OpenGL*. Charles River Media, Inc., Rockland, MA, USA, 2004. 3

[HJW*08] HACHISUKA T., JAROSZ W., WEISTROFFER R. P., DALE K., HUMPHREYS G., ZWICKER M., JENSEN H. W.: Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 33:1–33:10. 11, 13

[Hun15] HUNT W.: Virtual reality: The next great graphics revolution. High Performance Graphics, Keynote, 2015. 1

[Jon12] JONES S.: Introduction to dynamic parallelism. Nvidia GPU Technology Conference, May 2012. 3

[KG15] KHRONOS-GROUP: The OpenCL specification 2.1, 2015. 2

[KLK*11] KATO S., LAKSHMANAN K., KUMAR A., KELKAR M., ISHIKAWA Y., RAJKUMAR R.: RGEM: A responsive GPGPU execution model for runtime engines. In *Proc. Real-Time Systems Symposium* (Washington, DC, USA, 2011), RTSS '11, IEEE Computer Society, pp. 57–66. 2

[KLRI11] KATO S., LAKSHMANAN K., RAJKUMAR R., ISHIKAWA Y.: Timegraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX Annual Technical Conference* (Berkeley, CA, USA, 2011), USENIXATC '11, USENIX Association, pp. 2–2. 2

[LAF14] LEE H., AL FARUQUE M.: GPU-EvR: Run-time event based real-time scheduling framework on GPGPU platform. In *Proc. Design, Automation and Test in Europe Conference and Exhibition* (2014), DATE '14, IEEE, pp. 1–6. 2

[LKA13] LAINE S., KARRAS T., AILA T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proc. High-Performance Graphics* (New York, NY, USA, 2013), HPG '13, ACM, pp. 137–143. 3

[Mit87] MITCHELL D. P.: Generating antialiased images at low sampling densities. *SIGGRAPH Comput. Graph. 21*, 4 (Aug. 1987), 65–72. 12

[MLH*12] MEMBARTH R., LUPP J.-H., HANNIG F., TEICH J., KÖRNER M., ECKERT W.: Dynamic task-scheduling and resource management for GPU accelerators in medical imaging. In *Architecture of Computing Systems*, vol. 7179 of *ARCS '12*. 2012, pp. 147–159. 2

[Nvi08] NVIDIA C.: Programming guide, 2008. 2

[Nvi12] NVIDIA: Next generation CUDA computer architecture Kepler GK110, 2012. 2

[ODR09] OVERBECK R. S., DONNER C., RAMAMOORTHI R.: Adaptive wavelet rendering. *ACM Trans. Graph. 28*, 5 (Dec. 2009), 140:1–140:12. 11

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July 2010), 66:1–66:13. 3

[PPM15] PARK J. J. K., PARK Y., MAHLKE S.: Chimera: Collaborative preemption for multitasking on a shared GPU. *SIGARCH Comput. Archit. News 43*, 1 (Mar. 2015), 593–606. 3

[SGS10] STONE J. E., GOHARA D., SHI G.: OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering 12*, 1-3 (2010), 66–73. 2

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore GPUs. In *Proc. International Symposium on Parallel & Distributed Processing* (Washington, DC, USA, 2009), IPDPS '09, IEEE Computer Society, pp. 1–10. 2

[SKB*14] STEINBERGER M., KENZEL M., BOECHAT P., KERBL B., DOKTER M., SCHMALSTIEG D.: Whippletree: Task-based scheduling of dynamic workloads on the GPU. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 228:1–228:11. 3, 4, 8, 10, 11, 12

[SKK*12] STEINBERGER M., KAINZ B., KERBL B., HAUSWIESNER S., KENZEL M., SCHMALSTIEG D.: Softshell: dynamic scheduling on GPUs. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 161:1–161:11. 3, 5, 8

[TGC*14] TANASIC I., GELADO I., CABEZAS J., RAMIREZ A., NAVARRO N., VALERO M.: Enabling preemptive multiprogramming on GPUs. *SIGARCH Comput. Archit. News 42*, 3 (June 2014), 193–204. 3

[TPO10] TZENG S., PATNEY A., OWENS J. D.: Task management for irregular-parallel workloads on the GPU. In *Proc. High Performance Graphics* (Aire-la-Ville, Switzerland, Switzerland, 2010), HPG '10, Eurographics Association, pp. 29–37. 2, 10

[VHBS16] VINKLER M., HAVRAN V., BITTNER J., SOCHOR J.: Parallel on-demand hierarchy construction on contemporary GPUs. *IEEE Transactions on Visualization and Computer Graphics 22*, 7 (July 2016), 1886–1898. 2

[WWO14] WEN Y., WANG Z., O'BOYLE M.: Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *Proc. High Performance Computing* (2014), HiPC '14, IEEE, pp. 1–10. 2