Priority-Based Task Management in a GPGPU Megakernel

Bernhard Kerbl*
Supervised by: Markus Steinberger†

Institute for Computer Graphics and Vision
Graz University of Technology
Graz / Austria

Abstract

In this paper, we examine the challenges of implementing priority-based task management with respect to user-defined preferential attributes on a graphics processing unit (GPU). Previous approaches usually rely on constant synchronization with the CPU to determine the proper chronological sequence for execution. We transfer the responsibility for evaluating and arranging planned tasks to the GPU where they are continuously processed in a persistent kernel. By implementing a dynamic work queue with segments of variable size, we introduce possibilities for gradually improving the overall order and identify necessary meta data required to avoid write-read conflicts in a massively parallel environment. We employ an autonomous controller module to allow queue management to run concurrently with task execution. We revise Batcher's bitonic merge sort and show its eligibility for sorting partitioned work queues. The performance of the algorithm for tasks with constant execution time is evaluated with respect to multiple setups and priority types. An implementation of a Monte Carlo Ray-Tracing engine is presented in which we use appropriate priority functions to direct the available resources of the GPU to areas of interest. The results show increased precision for the desired features at early rendering stages, demonstrating the selective preference established by our management system.

Keywords: GPGPU, megakernel, GPU sorting, priority-based task management, dynamic work queue, Monte Carlo Ray-Tracing

1 Introduction

Parallel computing has become a valuable tool for handling time-consuming procedures that contain a high number of homogeneous, independent operations. Since the rise of the Unified Shader Model which features explicit programmability of GPUs, exploiting data level parallelism on a customary personal computer has become increasingly straightforward. The advent of NVIDIA's GeForce 8 Series introduced the first release of the CUDA architecture and associated compilers for industry standard programming languages. With respect to certain restrictions, the architecture enables programmers to run general purpose computations on compatible devices in parallel. Newer models of GPUs supporting these features are therefore often referred to as general purpose graphics processing units (GPGPU). Several hundred Stream Processors (SP), also referred to as thread blocks, which are grouped in clusters called Streaming Multiprocessors (SM) can be instructed to execute commands at the same time – the programmer simply specifies the number of necessary threads when launching a GPGPU kernel. The combined capacities of these SPs have produced a significant gap in raw speed between high-end GPUs and CPUs [12]. However, using the available resources to their full potential is not an easy task. Memory latencies, insufficient parallelization or unfavorable occupancy at runtime may cause implementations to fall short of their ideal behavior.

One particular cause of poor efficiency in CUDA kernels is unbalanced work load distribution, which can have a limiting effect on performance, especially when considering problems that exhibit irregular behavior, e. g. adaptive ray-tracing techniques [2]. Instead of relying on the built-in CUDA work distribution units, custom task management strategies can be implemented to address these issues [4]. One specific example is given in the OptiX Ray-Tracing Engine and its dynamically load-balanced GPU execution model [14].

In this context, the term *megakernel* refers to a solution where improved work load distribution is achieved using a persistent kernel in order to benefit from uninterrupted computational activity. One or more queues are commonly used to store tasks that are constantly being fetched and executed until the queues are empty. For static work queues, tasks can only be added in between megakernel launches, while dynamic implementations also allow for insertion of new tasks at runtime [4].

When using a megakernel for processing diverse problems and the implied task level parallelism, assigning meaningful priorities to each task can have a positive effect. Especially procedures that involve rendering may experience a considerable boost in usability through proper task classification. Considering applications with guaranteed frame rates, the perceived quality of each frame can

*kerbl@student.tugraz.at
†steinberger@icg.tugraz.at

be improved by focusing on areas that exhibit prominent features [9]. High quality initial views of rendered scenes can be generated by directing the resources of the GPU based on priorities. In an interactive environment with a modifiable view model, early perception of the visible dataset enables efficient adjustment of the extrinsic parameters in order to obtain a desired setup. Similarly, prioritizing user interactions achieves faster response to input commands and postpones time-consuming rendering procedures that are otherwise wasted on a setup that is inadequate.

Previous approaches based on priorities commonly exploit the sophisticated sorting methods on the CPU and establish means for communicating the favored order of execution to the GPU [10, 6]. Moving the responsibility of organizing the runtime agenda to the GPU expectably eliminates the otherwise significant overhead of inter-component communication. Therefore, we aim to provide a functional autonomic task management system for dynamic work queues on the GPU, in order to enable adaptive behavior for CUDA applications without interrupting the execution of tasks.

2 Previous Work

While analyzing the influences of possible limiting factors on ray-tracing kernels, Aila and Laine experimentally bypassed the default CUDA work distribution units, suggesting that this approach might show an improvement for tasks with varying durations [2]. In their implementation, they use a work queue from which tasks are constantly fetched and executed in individual thread blocks, which proved to be very effective. A detailed analysis of possible techniques for using work queues in a GPGPU kernel was authored by Cederman and Tsigas, addressing sophisticated load balancing schemes which are made possible through the atomic hardware primitives supported by the CUDA architecture [4].

In [14], Parker et al. successfully employed a self-provided strategy for balancing work load with the goal of improving efficiency in a ray-tracing engine using a persistent megakernel. Furthermore, they use static prioritization to achieve fine-grained internal scheduling for tasks to avoid penalties resulting from state divergence.

A recent example for a priority-based solution that achieves GPU task management in real-time is TimeGraph [10]. The routine relies on interrupts and substantial communication between the GPU and the CPU which acts as an executive supervisor. A similar system was implemented by Chen et al., in which queues are shared and accessed regularly by the CPU and GPU [6]. In recent developments, Kainz et al. employed prioritization in a rendering application by sorting work queues between kernel launches to achieve improved richness of detail for predetermined frame rates [9].

Efficient sorting algorithms for parallel systems, especially targeting GPGPUs, have become a popular research topic. Following the implementations of Batcher's bitonic merge sort which was developed for parallel sorting networks [3], other algorithms designed for use on massive datasets were adopted for the GPU as well [5, 7, 13, 15].

3 GPU Megakernel System

3.1 Available Resources and Challenges

We target dynamic GPGPU task management by introducing a global, self-organizing work queue in a megakernel system which allows for arbitrary tasks to be added at runtime. Apart from the functions to be invoked upon execution of a task, each queue entry provides additional attributes that are considered during different stages of the management process. Sorting the queue is a time-critical problem, since the megakernel system constantly removes the frontmost entries and processes the associated behavior in the available SPs to achieve proper workload distribution.

Due to the intended autonomy of our management system, we cannot rely on the CPU to rearrange queue entries based on their priorities. Instead, one thread block is reserved and used as a controlling unit which is responsible for sorting the queue at runtime. The remaining thread blocks are henceforth referred to as *working module* to clarify their intended purpose. Since both modules constantly process the shared contents of the queue, leaving them unprotected would cause interference and lead to severe runtime errors. Therefore, we require secure methods for classifying queue entries and deciding whether they are safe to be processed by either module.

3.2 Work Queue Segmentation

Since the contents of a dynamic work queue are potentially volatile, we partition the queue and thereby enable faster detection of segments that are unlikely to change in the near future. Consequently, the controller can quickly select segments that are not yet in use and perform sorting while the working module constantly removes entries from segments in the front of the queue. Treating segments as instances of classes gives us the advantage of storing additional information about the contained queue entries as attributes, such as a reference to the task with the shortest execution time or current availability.

For using segments to restructure the work queue, we identify two additional constraints in order to avoid access conflicts. First, only full segments qualify as appropriate candidates for sorting, which leads to entries at the back being ignored if they are located in a partially populated segment. Second, since we must not tamper with the entries of segments whose contents are currently being executed, we cannot sort the tasks at the very front, which leads to inevitable disarray for the leading segments.

Considering these limitations and their effects on kernel execution, we provide settings to adapt the behavior of the controller as required. The user may select a specific segment size for a particular purpose: while a smaller size generates a finer granularity and reduces the number of neglected entries both in the front and in the back of the queue, it requires high organizational overhead and increases the time needed for establishing a decent order. A bigger segment size may improve the sorting performance for a higher number of queue entries, but at the same time larger chunks of the queue will be ignored due to incomplete segments.

3.3 Mutual Exclusion

Even though we established that unused segments can be quickly identified by the controller using only work queue partitioning, we need to consider overlapping accesses since sorting algorithms are time-consuming as well and may take even longer than task execution. Dealing with these situations requires an unambiguous system-wide regulation for deciding which module is currently allowed access to a segment. We decided to introduce a thread-safe policy for checking and setting the current availability state of a segment, combining two common approaches in a multipurpose attribute variable.

The CUDA instruction set provides means to change the contents of the attribute variable atomically. Each SP in the system tests the state of a particular segment before accessing its contents. For the working module, the state variable behaves like a semaphore, allowing a specified number of accesses before resetting it to the initial zero value. If the controller requires the contents of an available segment, it exchanges the current state with a negative integer. Threads in the working module that are trying to acquire the contents of segments that are being sorted will perform a busy wait until the controller signals completion of the sorting algorithm by assigning a positive value to the variable.

4 Sorting the Queue

4.1 Bitonic Merge Sort

Our controller utilizes an adaptation of the bitonic merge sort to rearrange the contents of the work queue. The algorithm was devised by Ken Batcher for parallel sorting networks as an alternative to the odd-even merge sort and operates by constructing bitonic sequences of increasing lengths and merging them to generate sorted output [3]. A simple setup demonstrating the procedure for applying the algorithm in parallel is illustrated in Figure 1. For an array of length N and T concurrent threads, the algorithm requires $\mathcal{O}(\frac{N}{T} \cdot log^2 N)$ parallel comparison operations. Popular fields of application include collision detection and visibility calculation in particle systems [11]. The most

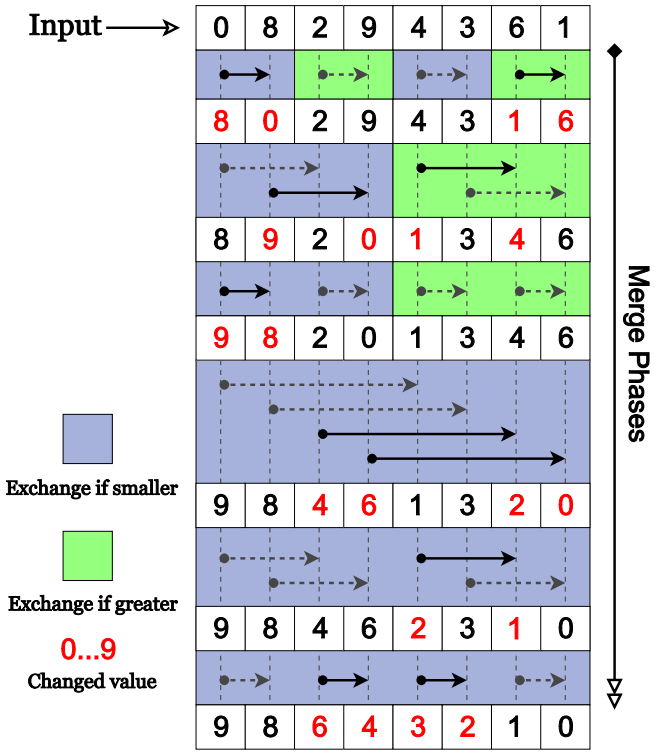

Figure 1: One possible implementation of the bitonic merge sort using one thread per element index. The colored rectangles indicate different comparison operators being used for evaluating whether two values should be exchanged. By forming bitonic sequences in each step, the merge phases are applied consecutively until the input is sorted in descending order.

efficient GPU implementations of the bitonic merge sort were able to outperform the sophisticated std::sort methods on contemporary CPUs [15]. Recently developed algorithms for sorting on the GPU are more commonly based on radix or bucket sort, although some frameworks implement hybrid variants in order to exploit the characteristics of bitonic sequences [13, 7, 16].

4.2 Benefits

We aim to provide the user with the possibility of choosing custom priority values. The bitonic merge sort is comparison based and is therefore applicable for any data type that supports the logical operators $<$ and $>$.

For smaller data sets, the bitonic merge sort is one of the fastest algorithms if the underlying architecture is optimally exploited [8]. Ideally, the thread block size equals the number of elements to be sorted. In such a case, $\frac{N}{T} = 1.0$ and the algorithm requires exactly $log^2 N$ parallel steps to execute.

Concatenating two arrays sorted in distinct order yields a bitonic sequence by definition, which corresponds to the input in the final top-level merge phase of the bitonic

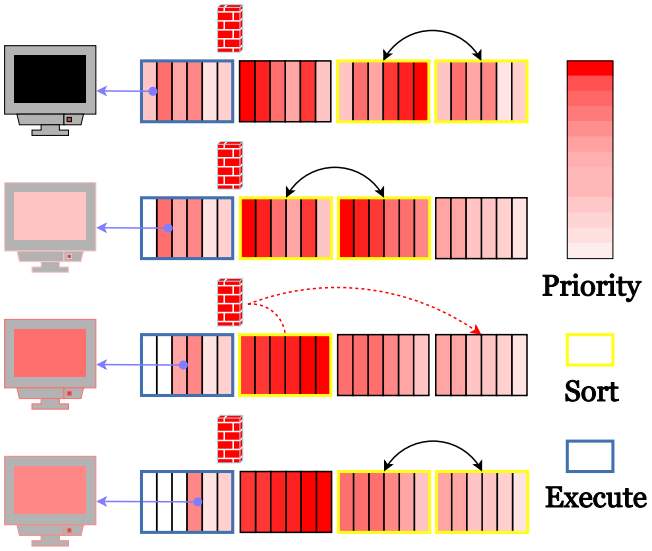

Figure 2: A pair-wise sorting algorithm applied to the segments in a queue. Neighboring segments are combined to achieve gradual restructuring of the contents, starting from the back and advancing towards the front. If a segment is encountered that is currently unavailable, the algorithm restarts from the back.

merge sort. Based on these properties, we can reduce the number of necessary comparison operations significantly when sorting two preprocessed arrays by inverting one array and appending it to the other, thereby creating a bitonic sequence. For fusing two sorted arrays of length $\frac{N}{2}$, the worst-case complexity can thus be expressed as $\mathcal{O}(\frac{N}{T} \cdot log N)$.

4.3 Using Bitonic Merge Sort on Segments

We use an optimized bitonic merge sort to rearrange segments of the work queue sequentially in successive passes. For each pass, we need to acquire available segments and apply the bitonic algorithm to their combined data set. The key-index pairs are retrieved from the work queue using the segment indices as offsets for addressing the associated tasks. We identified two basic qualities that the controller should exhibit when sorting segments:

- Constant refinement: the accuracy of the resulting sequence should increase with the number of passes
- Effectiveness: high-priority work packages should advance towards the front as soon as possible

The basic idea to improve the overall order is to compare each segment with its predecessor iteratively. For a work queue with N segments of size S, we need $N-1$ passes to move the S most important tasks to the leading segment. This approach, though basic, continually improves the order in the queue. If we reach a segment where no predecessor can be acquired, we reinitiate the procedure starting from the back, which is illustrated in Figure 2. The resulting accuracy of the sort is proportional to the number of passes performed. Obtaining a fully sorted list would require a total of $\frac{N^2+N}{2}$ passes, but approximate sorting with an emphasis on prioritizing important tasks over regular ones is sufficient to induce adaptive behavior. Also, as mentioned in Section 4.2, partially sorted input can be processed much faster. This property translates well to the segment-based approach: by monitoring a boolean member variable that is true for segments that are revisited, we can decide whether it is sufficient to apply a top-level bitonic merge. We consider three different combinations of segments and provide an optimized sorting method for each of the following pairings:

- **Unsorted – Unsorted**
- **Unsorted – Sorted** or **Sorted – Unsorted**
- **Sorted – Sorted**

5 Time Management

5.1 Motivation

The system described thus far is capable of managing queued tasks based on their priorities without assistance by external components. However, due to the necessary synchronization of controller and working module by mutual exclusion, a significant delay is added to the total megakernel execution time whenever a SP in the working module transitions into a state of busy waiting, caused by segments being unavailable as they are currently being sorted. We target this issue by implementing a management strategy using time-based regulations for avoiding collisions of the working module and the controller when accessing the work queue. The chosen approach requires collection and maintenance of related meta data for task duration and sorting performance.

5.2 Avoiding Collisions

Usually, the primary objective of a megakernel is to exploit the resources of the GPU. Hence, we do not put any additional constraints on accesses made by the working module. Instead, the controller performs advanced sanity checks before locking two segments for sorting. Following the algorithm described in Section 4.3, we select two qualified segments. The time needed for performing the bitonic merge sort on the selected segments is stored in $time_{sorting}$. We then proceed to probe the anterior sections of the queue: segments in between the chosen candidates for sorting and those that are currently being executed are regarded as time buffers. The required amount of time for the working module to process a buffer segment is estimated by the execution time of the shortest contained

task. The respective variables are read for each segment and accumulated from back to front in a second variable $time_{buffers}$. Once the condition $time_{buffers} \geq time_{sorting}$ is fulfilled, we abort the traversal stage and initiate the bitonic merge sort procedure. If the available time is not sufficient for sorting, the procedure is reinitiated at the back of the queue to prevent possible collisions.

5.3 Collecting Meta Data

5.3.1 Task Execution

For each task, the working module records the time needed for execution and compares it to previous results by default. It is necessary to monitor these values constantly for automatic runtime detection since the execution time of a task may fluctuate considerably and cannot be predicted without error. Whenever a new task is added to the work queue, the corresponding segment updates its estimated buffering effect based on these measurements. In order to minimize the probability of collisions, we choose the pessimistic approach and exclusively store the shortest duration for each task measured so far.

5.3.2 Sorting Methods

For each of the three sorting modes, we measure and update the number of clock cycles needed by the respective method whenever it is invoked. The $time_{sorting}$ variable can thus be estimated more accurately by the controller based on the orderliness of tasks in the segments that are selected for sorting. As opposed to task execution, measurements are discarded if they are lower than previous results, although sorting performance is less likely to change over time. This approach further reduces the likelihood of collisions in the work queue, since always the longest duration is assumed for each sorting method.

5.4 Custom Timing Settings

As an alternative to the default pessimistic behavior, we provide settings for the advanced user to customize the internal time management. A more flexible strategy can be achieved by defining a global multiplier for the time buffers. If a task is known to have a reliable average duration or requires a fixed number of clock cycles defined by its inherent complexity, the automatic runtime detection may be disabled selectively. Instead, a static value can be provided to indicate how many clock cycles should be assumed for execution.

For projects where priority sorting is of utmost importance, meta data acquisition may be disabled for all tasks. The user can then define a constant global variable that is substituted for each buffer segment.

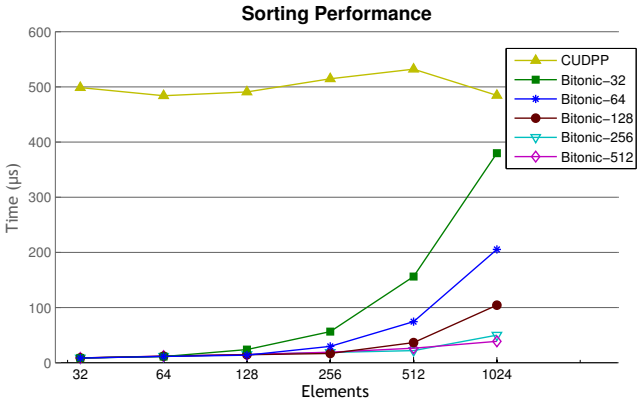

Figure 3: Comparison of performance for the tested sorters with the standard CUDPP radix sort for unsigned integers. Our implementation of the bitonic merge sort yields almost constant results for setups where the number of elements does not exceed the number of threads used.

6 Results

6.1 Bitonic Merge Sort Performance

The performance of the bitonic merge sort in our system is based on two factors, namely the segment size and the number of active threads. The segment size can be modified in order to achieve a desired granularity for the sort. Furthermore, the selected block size for a kernel also defines the dimension of the controller. The algorithm was therefore evaluated using a representative selection of settings. In order to compare its potential efficiency to existing sophisticated routines, the optimized bitonic merge sort was executed using a single thread block for sorting a limited number of keys outside of the megakernel system. This procedure is equivalent to a sorting pass of the controller module and thus models the expected behavior for sorting two segments. We chose thread block dimensions and array lengths as powers of 2. Thread block dimensions range from common CUDA warp size of 32 threads to the largest possible block size of 512 threads. Array lengths start at 32 and end at 1024 elements, which equates to double the maximum segment size in our megakernel system. The tests were conducted using a GeForce 560 Ti. A visualization of the parametrized setups and the resulting runtimes for sorting unsigned integers can be found in Figure 3. We compared our results with the recorded times from the CUDPP test suite [1]. Even for our most unfavorable setting with 32 threads sorting an array of 1024 values, the optimized bitonic merge sort beat the CUDPP radix sort by little over 100 μs ($\sim$ 20%). For more balanced setups, our implementation was up to 56 times faster in comparison. We would like to point out that the CUDPP project targets larger data sets and is indeed very potent for lengthy input [16]. Considering the non-linear growth rates, the emergent trend suggests that with increasing array sizes the bitonic sorter will eventually be bested. How-

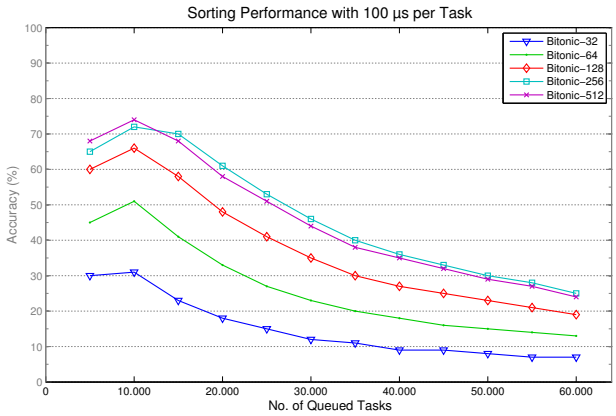

Figure 4: Algorithm performance with a fixed task duration of 100 μs. Starting at 10,000 tasks, we observe a steady decline in accuracy. This suggests that although the negative influence of unreachable segments is constantly reduced, the performance eventually drops due to insufficient time buffers.

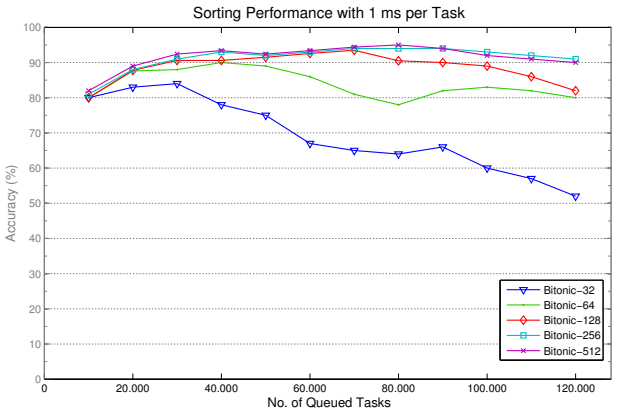

Figure 5: Evaluation of the algorithm performance for complex tasks with runtimes exceeding 1 ms. For smaller thread block sizes, we observe an early decline due to the increased effort for sorting a high number of entries with fewer threads. The highest recorded score of 95% is obtained using 512 threads.

ever, given that we intend to frequently sort pairs of segments containing 512 tasks or less using only one thread block, it appears to be a very suitable solution.

6.2 Sorting the Queue at Runtime

We evaluated the performance of the presented strategy for sorting the queue segment-wise in our megakernel system. Since the duration of a megakernel is implicitly determined by the tasks it executes, we assessed the accuracy of our management system for a given number of tasks instead. We used blocking tasks to occupy thread blocks for a specified number of clock cycles. The segment size for the work queue was set to 256 tasks in order to balance granularity and sorting efficiency. Since both modules in the megakernel start simultaneously, the leading segments are immediately locked by the working module and can never be processed by the controller, which makes it impossible to achieve a fully sorted queue at runtime. Regarding these constraints, we estimated the actual performance by rating each executed task based on its predecessors. We enabled automatic runtime detection and stored the priorities of executed tasks chronologically in a list of entries L which was subsequently evaluated. Ideally, we anticipate a descending sequence of values where an entry at index i (starting from 0) has i previous entries representing tasks with higher priorities. Hence, the score for each tested setup with the specified number of tasks n is defined as follows:

$$S(n) = \frac{\sum_{i=1}^{n-1}\sum_{j=0}^{i-1} f(i,j)}{n-1}, \quad f(a,b) = \begin{cases} \frac{1}{a}, & \text{if } L(b) \geq L(a) \\ 0, & \text{else} \end{cases}$$

Assuming worst case conditions for our test setup, the queue was initiated with task priorities in ascending order, which would yield a total score of 0. The results for a given number of entries where each associated task took at least 100 μs are illustrated in Figure 4. The accuracy of the order in which they were executed peaked at 74% when using a block size of 512 threads for 10,000 tasks. Lower values preceding the apex of each function were caused by the statistical influence of the reserved leading segments which were not sorted before execution.

Due to device-related delays between fetching and processing, tasks may not be executed precisely in the same order as they are ranked in the queue. The resulting effect caused slightly lower scores when using 512 threads compared with a block size of 256 threads for a higher number of tasks. For time-consuming procedures, such as those found in elaborate ray-tracing engines, we considered the results after raising the blocking interval to 1 ms (see Figure 5). With the highest possible number of threads, we achieve a maximum score of 95% for 80,000 entries. Based on these results, we can conclude that the accuracy in the order of execution increases with the complexity of planned tasks, since the prolonged execution time can be utilized to issue additional sorting passes.

6.3 Adaptive Monte Carlo Ray-Tracing

We demonstrate the effects of using custom priorities in a progressive Monte Carlo Ray-Tracing engine and two different scenes. The engine evaluates 512 paths per pixel with a resolution of 800x600 and checks intersections with objects iteratively. Pixels are grouped as patches of 4x4 and each patch is assigned to a designated task instead of using one task for each pixel. The reduced number of work queue entries to be sorted improves the efficiency of the task management system. A task that is executed computes two random traversal paths for each pixel in a patch

and evaluates the new priority based on different strategies, two of which are presented in this section. Tasks repeatedly append themselves to the end of the queue until all rays for the associated pixels have been cast. We provide representative snapshots of both scenes after executing ∼60% of all planned tasks. In order to illustrate the associated progress, we use heat maps to indicate the number of rays cast for each pixel patch.

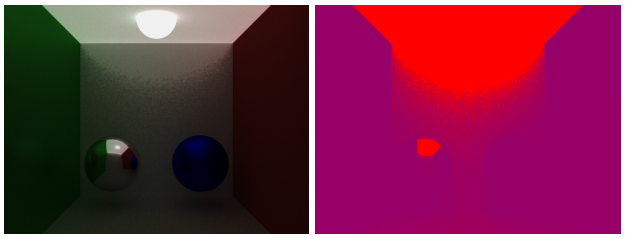

Figure 6: Snapshot of a scene containing objects whose eventual surface colors are influenced by illumination and reflection. Prioritizing patches with high accumulated intensity leads to selective rendering of light sources and white surfaces, which enables a faster perception of details in these areas.

In our first test case, color values returned by rays for each pixel were simply accumulated and the image was then rendered using maximum to white mapping. The examined scene shows the interior of a softly lit room containing reflecting objects and light sources. We used our task management system to focus on high intensity color values. This eventually led to a global preference of brighter areas. Figure 6 shows the scene at an intermediate stage. We can clearly discern the objects that are emitting or reflecting light and observe the increased detail for the left sphere and the ceiling. The priority for each patch was calculated using $priority = \sum_{i=1}^{pixels} color$.

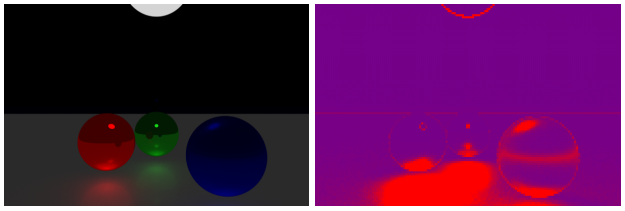

Figure 7: Using difference values as priorities leads to preference of patches with low convergence rates. We observe more rays being cast early on for reflections of spheres on the floor since the surface generates diverse color values depending on the direction of rebounded rays.

For our second example, the output was normalized at each pixel using the heat map data. The selected setup allowed for detection of pixel patches with low convergence rates and prioritization of these areas to achieve enhanced initial views. A large portion of the scene could be neglected at first due to an opaque, unlit wall at the far end of the room (see Figure 7). In Figure 8, we emphasize improved image quality when compared with uniform rendering by magnifying the affected regions of images generated 210 ms after initiation. The applied priority formula can be as expressed as $priority = \sum_{i=1}^{pixels}(color - color_{old})$.

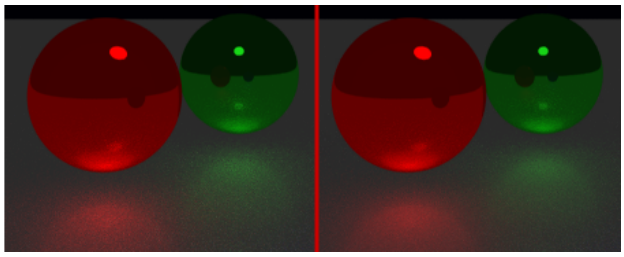

Figure 8: We compare the conventional approach of distributing rays uniformly with the priority-based procedure. The left side shows an early closeup of the scene using the default method and exhibits more noise than the image on the right, which features smooth color reflections as a result of proper task management.

We used the normalized output images from the second test case to assess the mean squared error (MSE) for default and priority-based rendering when compared to the ground truth. Even though evaluating the priority formula required expensive atomic operations, we noticed a clear reduction of the MSE at each point in time (see Figure 9).

7 Conclusion

We presented a priority-based task management system with elaborate timing strategies to enable adaptive behavior for GPGPU programs. We successfully incorporate a controller module in a megakernel system and prevent it from interfering with the continuous execution of queued tasks. We eliminate inter-component communication and the associated overhead by sorting the contents of our dynamic work queue at runtime using a designated thread block. By optimizing the bitonic merge sort algorithm, we establish a basis for iteratively rearranging the segments of the queue. We evaluate the effectiveness of the sorting algorithm by invoking large numbers of tasks and compare performance for tasks with different durations. For more complex tasks, we reach promising scores regarding the order of execution even in unfavorable setups. A Monte Carlo Ray-Tracing engine running in our system shows adaptive behavior and demonstrates how prioritization in a rendering application can speed up the assessment of desired features in a scene. Adaptive rendering offers an extensive field of research for possible priority functions and their impact on image quality. Since the effects of complex formulas do not necessarily outweigh the corresponding overhead, the development of new, efficient priorities requires profound research and sophisticated methodology.

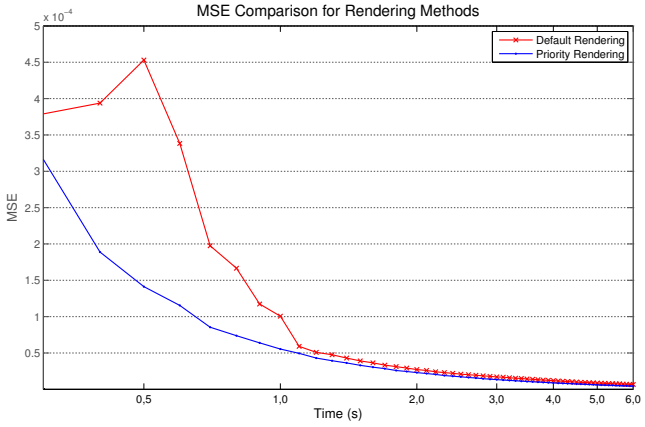

Figure 9: We calculate the mean squared error by subtracting snapshots of the rendered scene generated at regular intervals from the ground truth and evaluating the difference of pixel values. Prioritization of pixel patches that return ambiguous color values leads to improved results for our second test case.

References

[1] Cudpp: Cuda data-parallel primitives library, 2012. `http://gpgpu.org/developer/cudpp`.

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[3] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[4] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[5] Daniel Cederman and Philippas Tsigas. On sorting and load balancing on gpus. *SIGARCH Comput. Archit. News*, 36:11–18, June 2009.

[6] Long Chen, Oreste Villa, Sriram Krishnamoorthy, and Guang R. Gao. Dynamic load balancing on single- and multi-gpu systems. In *IPDPS*, pages 1–12. IEEE, 2010.

[7] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[8] Mihai F. Ionescu. Optimizing parallel bitonic sort. In *Proceedings of the 11th International Symposium on Parallel Processing*, IPPS '97, pages 303–309, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Bernhard Kainz, Markus Steinberger, Stefan Hauswiesner, Rostislav Khlebnikov, and Dieter Schmalstieg. Stylization-based ray prioritization for guaranteed frame rates. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 43–54, New York, NY, USA, 2011. ACM.

[10] Shinpei Kato, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, Berkeley, CA, USA, 2011. USENIX Association.

[11] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[12] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[13] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, first edition, 2007.

[14] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[15] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[16] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, September 2008.