Work distribution methods on GPUs

Christian Lauterback* Qi Mo† Dinesh Manocha‡

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599

Technical Report TR009-16

Abstract

Due to their high thread and data parallelism, commodity GPU architectures currently provide very high performance and general programmability. Many algorithms have been successfully ported to GPUs, but several limitations have prevented scalable implementations of many less easily parallelizable recursive and hierarchical algorithms. In this paper, we investigate general approaches for dynamic work distribution and balancing on GPUs to allow recursive algorithms such as hierarchy algorithms. We propose a new and simple method that instead employs only minimal synchronization between cores and explicit balancing, but is more suited to the properties of the architecture. We show an implementation of several applications on a current GPU and our results show that for applications with fine-grained parallelism it outperforms other currently used work distribution methods since it avoids limitations of GPU architectures and provides competitive performance on coarse-grained applications.

1 Introduction

Commodity graphics processing units (GPUs) are increasingly used for general purpose computations. They offer a high level of data and thread parallelism along with higher memory bandwidth, as compared to CPUs. These features have been successfully exploited to achieve higher performance for many geometric, scientific and database applications.

Current GPUs are many-core processors with a high number of cores. In this paper, we address the problem of designing efficient recursive algorithms on current GPU processors that can balance the load among various processing units. Examples of such algorithms include methods that use hierarchies or trees, adaptive refinement schemes, sorting, recursive search, etc. Moreover, many of these techniques are frequently used as sub-algorithms in high-performance computing applications. In many cases, the overall cost of the task is not known apriori and the resulting application may generate additional sub-tasks at runtime. For example, in a tree search several branches will be pruned relatively quickly whereas others are traversed very deeply. In order to utilize all the hardware resources, the work may therefore need to be frequently re-distributed. This is different from problems or algorithms with more regular structures or where the work distribution is known (e.g. fast fourier transform).

GPUs have been used to implement recursive algorithms such as ray tracing using spatial or bounding volume hierarchies [Foley and Sugerman 2005; Horn et al. 2007]. In practice, the problem is different since for these applications many parallel queries are evaluated and no work for an individual query is distributed among the multiple units, therefore allowing each query to be handled independently and there is no need intercommunication between the cores. As a result, it is relatively simple to design good parallel algorithms for such cases. However, many other problems such as the construction of spatial hierarchies by divide-and-conquer algorithms can pose more challenges. Current solutions for performing these computations on GPUs [Zhou et al. 2008; Lauterbach et al. 2009] used work queues. However, these algorithms can spend considerable time on explicit management of these queues, and thereby adding significant overhead to the overall computation. In particular, the hierarchy construction algorithm is one that performs a lot of work for each task before creating new work units, and such can amortize expensive communication. For many problems, this is not the case and much less computations is performed before communication is necessary. In these cases, the overhead of work distribution will become the bottleneck.

In this paper, our goal is to exploit the parallel capabilities of current many-core GPUs to efficiently perform recursive algorithms. Even though current GPU architectures provide the capabilities to implement intercommunication and synchronization between cores, we show that the particular hardware architecture of GPUs poses problems for traditional work balancing approaches as frequently used in previous work. We look at several examples of common applications as well as different solutions to work distribution including work queues, work stealing as well as a novel minimal work balancing approach presented in this paper and compare them. Our results show that for applications with fine-grained parallelism, methods that need significant communication do perform significantly worse than our minimal scheme due to latency bottlenecks.

2 Background

We first briefly summarize previous work in the areas of GPU-based algorithms and parallel computation. Later, we give a brief overview of GPU architectures.

2.1 Previous work

There is considerable literature in parallel computing on the use of work queues for load balancing, including locking and non-locking shared queues such as work stealing approaches [Arora et al. 1998; Hendler and Shavit 2002; Chase and Lev 2005; Hendler et al. 2006] or dynamic load balancing of irregular algorithms [**?**]. These techniques map very well to hierarchical and recursive operations and have been employed extensively in parallel systems and parallel programming languages such as Cilk [Frigo et al. 1998]. Other techniques are based on stream computing that can handle unstructured computations more efficiently [**?**]. However, most of these techniques have not been used on GPUs as the overhead of performing communication between the cores through main memory is high. Instead, previous techniques for GPU work queues in the context of load balancing have used explicit compaction methods between kernel calls. The overhead of these methods makes them efficient only for applications with relatively high computational intensity and coarse-grained parallelism [Zhou et al. 2008; Lauterbach et al. 2009]. Recently, some initial research has been performed into using work queues and work stealing for octree construction [Cederman and Tsigas 2008a] and sorting [Cederman and

*cl@cs.unc.edu
†qmo@cs.unce.edu
‡dm@cs.unc.edu

Tsigas 2008b]. In addition, other synchronization primitives have been investigated [Ha et al. 2008].

There is extensive literature on hierarchical data structures and algorithms as well as GPU computing. Many GPU-based algorithms exploit hierarchies or irregular data structures for fast rendering or simulations. These include multigrid solvers [**?**], photon mapping [Purcell et al. 2003], fluid simulation [Kyle Hegeman and Miller 2006], random access data structures [Lefohn et al. 2006], spatial hashing [Lefebvre and Hoppe 2006], random access trees [Lefebvre and Hoppe 2007], visualization of adaptive mesh refinement data [**?**], etc. Most of these applications perform specialized hierarchy operations or computations.

2.2 GPU architectures

In this section, we give a brief overview of current many-core GPU architectures that we exploit in our algorithm. We refer the reader to [Volkov and Demmel 2008] for a more in-depth analysis. In general, the GPUs have a relatively large number of independent processing cores, each of which is optimized to perform vector operations but runs at comparatively low frequencies compared to current CPU architectures. The high vector width – between 8 and 64 for current generation of GPUs – also implies that any efficient algorithm needs to utilize data parallelism to achieve high performance. Another main issue is the GPU memory system that typically provide more bandwidth as compared to CPU memory systems, but has a higher latency. Moreover, the caches in GPUs are much smaller than CPU caches and current GPUs only provide read-only access, which limits their use for general purpose computing. The main goal of these caches is not to reduce latency, but rather to reduce the amount of memory bandwidth used.

In order to improve latency, GPUs use two main techniques: first, each core has local scratch memory that can be used in programs as an explicitly managed store and provides very low latency. Thus, if algorithms are designed with sufficient locality, most of their memory accesses should go there. Second, the high main memory latency is circumvented by use of hardware multi-threading while waiting for the results of memory accesses. This can be achieved by running several data-parallel tasks on each core such that the processor can switch between them as they are blocked waiting for memory accesses. Finally, the vector size used in programs can be extended beyond the size of the actual hardware vector units such that the computations can be strip mined (i.e. processed in chunks of real vector size). This means that GPUs can efficiently process these large vectors for latency hiding as well. Overall, this means that algorithm design should aim at providing both as much task and as much vector parallelism as possible since it will improve the efficiency of the processors.

3 Work distribution on GPU architectures

In this section, we discuss work distribution algorithms and introduce our minimal synchronization approach that is particularly suitable for hierarchies and other recursive algorithms where fine-grained load balancing is necessary. Our approach is general and applicable to different GPU architectures. We first introduce the notation used in the rest of the paper.

3.1 Notation

In general, we refer to each independent processing unit on the GPU as a core, each of which is a vector processor that executes the same instruction on a data array elements in parallel. A running program is a task that can be executed in parallel on each core of the GPU, and we refer to the specific code run by the task as a kernel. This kernel is identical on all the cores. We assume that our kernels take work units as input that can be processed independently and can lead to generation of new work units. The exact definition of a work unit depends on the kernels: for example, when testing two hierarchy nodes for overlap, the result can be that similar intersection tests needs to be performed recursively on the children of these nodes. In this case, the intersection test is the kernel and the work unit is the pair of hierarchy nodes. More general, a recursive algorithm gets a work unit as an input that defines all parameters to the recursive function call, and then may spawn additional work units representing recursive calls, as well as other work units that may combine results from those recursive calls. Overall, we look at the complete set of active work units at any step in the algorithm as the *front*, which may or may not be available in an explicit form. If the order in which recursive calls are generated and executed can be represented as a tree, then the front is simply a cut through the tree. The size of the front governs the available parallelism and thus should be as large as possible since elements in the front can be executed independently. In practice, the size depends on the order of execution of work units, the configuration of input objects and the size of the hierarchy.

3.2 Processing hierarchical workloads

The main challenge when performing parallel operations on hierarchies is the dynamic nature of work distribution. Since the workload is not known a priori, assigning work units to different cores and vector lanes in advance is impossible. A front of sufficient size to occupy all cores may not even exist until after some steps in the computation. For example, in hierarchical collision detection the initial front just has one element, i.e. the pair of root nodes; as the traversal generates more pairs of nodes the size of the front typically increases in geometric progression. Work is typically also not evenly distributed over the hierarchy since some sub-trees of the hierarchy may be skipped early whereas others need to be processed much deeper. Therefore, some mechanism for distributing work between cores and load balancing during the hierarchy operation is necessary. We identify multiple existing solutions for performing this balancing on multi-threaded CPU or GPU architectures.

Compaction: Prior approaches used explicitly managed queues on GPUs due to limitations that made core synchronization impossible. In this case, each kernel has an input and an output work queue and is bounded in how many work units it can spawn per step. The work kernel is executed once, then the output queue is processed by a compaction kernel that eliminates empty positions in the queue. The roles of output/input queue are then swapped and the work kernel executed again, until no more output units exist. However, this approach has several disadvantages: it makes use of the GPU's static task scheduler to distribute the work units in the input queue, which does not perform any load balancing. In addition, there may be significant overhead for repeatedly running the compaction kernel, both due to call overhead and limited available parallism of the compaction. Finally, bounding the number of work units created per step may lead to limitations for several algorithms. In the context of hierarchy construction [Zhou et al. 2008; Lauterbach et al. 2009], this approach works due to two reasons: a) the work kernel performs a relatively large amount of work per step and b) it could only write out a very small number (two) of new work units per step, thus making list maintenance relatively cheap.

Work queues: Due to the introduction of atomic operations such as CAS and memory fence operations, it is also possible to use global shared memory structures on GPUs. The simplest solution is to use a shared work queue in global memory and control access to it via a synchronization primitive. However, this approach has significant contention for access to the work queue and thus high

overhead as cores are busy waiting for work access.

Work stealing: Algorithms such as non-blocking work stealing [Arora et al. 1998] have been very successful in multi-core applications due to a reduction in contention. In this case, every core has a separate work queue which is still accessible to other processors. However, cores can steal work units from others' queues whenever their own queue is empty. By using an implementation with an array-based deque such as in [Arora et al. 1998], blocking in access to the queues can be reduced. In addition, ideally each core will try to steal from different work queues first to further reduce contention. Previous work [Cederman and Tsigas 2008a; Cederman and Tsigas 2008b] has found this to perform better than global work queues.

However, all these techniques do not currently work well on GPUs for multiple reasons. Primarily, they are based on the assumption that low-latency communication between cores is possible in order to manage concurrent access to shared structures. Unfortunately, this is only possible in a very restricted sense on current GPUs. The main barrier to communication is the latency and lack of a memory consistency model in the global GPU memory shared by the cores, i.e. different cores are not guaranteed to see memory writes from other cores or may not even see them in the same order they were written. Even though newer GPU architectures provide atomic operations such as compare-and-swap (CAS) that could be used for locking operations, the remaining problem is that previous writes to the memory protected by the lock may not have been executed yet, thus preventing implementation of work queues or other structures shared by all cores. Using memory fence operations, consistency can be enforced, but with relatively high overhead. Even if memory consistency were not a problem, busy waiting such as by spinning on a lock variable is relatively inefficient on an architecture with high memory latency and hardware multi-threaded execution can also lead to priority inversion and prevent other threads on the same core from performing useful work. In addition, actual task scheduling to the cores is still handled by fixed-function units on hardware that cannot be affected by the programming interface. The number of actual tasks is almost always higher than the number of cores to allow hardware multi-threading, and since the scheduler does not make any guarantees as to fairness in core allocation, it is not guaranteed that any task is actually executed in parallel to any other. Thus, global communication between all tasks cannot be guaranteed as some may not even be started until others end.

3.3 Lightweight work balancing

Our approach is motivated by this challenge and tries to circumvent the lack of a memory consistency model and still provide some limited coordination between cores to avoid maintenance overhead. While some explicit queue maintenance work outside of the work kernel is necessary, we drastically reduce the amount of time spent on it by only performing it as a load balancing step between cores when we detect that enough cores are idle. In our work organization approach each task maintains a private work queue either in shared or global memory (depending on the size of work units) that can be either be read in a data parallel manner to provide a work unit for all vector element, or just one unit for the whole task, depending on the application. Putting new elements in the local work queue can be implemented easily by either using a parallel prefix sum based on the results of each work kernel, or atomic increments to the local work queue index to avoid write conflicts. There are two cases where work processing must end: first, the queue is empty and no more work is available and second, the allocated space for the queue is full and the work kernel cannot be executed because any further work elements could not be stored. In that case, we consider the task inactive and atomically increment a global idle

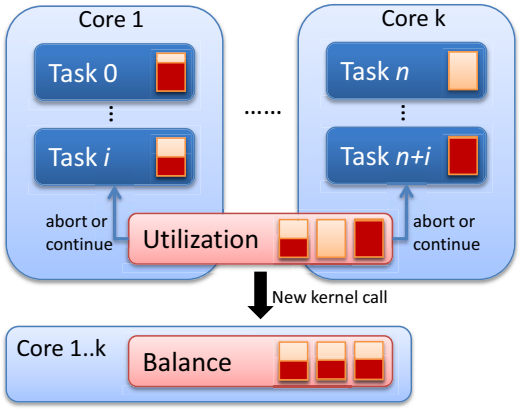

Figure 1: Our approach: *In our approach, each task keeps its own local work queue in local memory and can generate new work units without coordinating with others. After processing a work unit, each task is either able to run further or has an empty or completely full work queue and wants to abort. By checking how many tasks are marked idle after each step, tasks will abort whenever a certain threshold of idle tasks is reached. At that point, an explicit work balancing kernel is launched that rearranges the work queues and distributes work units such that all cores have roughly the same workload.*

$executeWithBalancing(globalWorkQueue, taskNr, threshold, nIdle)$
begin
  $localQueue := globalWorkQueue(taskNr)$;
  comment: copy work queue to local memory;
  while $!wq.empty() \wedge !wq.full() \wedge nIdle \leq threshold$ **do**
    $item := wq.pop()$;
    $kernel(wq, item)$;
  od
  $atomicInc(idle)$;
  $globalWorkQueue(taskNr) := localQueue$;
end

Figure 2: Lightweight work balacing. nIdle is initialized to 0 before execution and then incremented whenever a task exits for some reason. threshold is a user-specified parameter setting the number of idle tasks allowed until balancing.

counter. To make sure that a sufficient amount of tasks is active to ensure parallel utilization of the whole GPU, each task reads the counter after processing a work unit and compares against an idle threshold to determine whether re-balancing work is needed. If that is the case, then the task aborts (see Fig. 1 for illustration.) Note that even without a memory consistency model, every task will see increments to the global counter eventually.

In case balancing is needed, a we exectute a global work distribution kernel that steals work from some of the queues and distributes it to those that are not full. Work redistribution is performed in parallel by first counting the total number of work items and computing a roughly equal number of work units for each queue. Resorting the work units to the new queues for all cores is performed in parallel by computing offsets through a data parallel prefix sum algorithm and then copying to the new positions by all cores in parallel. Overall, there are a couple of factors influencing the performance of this approach, most of which are dependent on which actual work kernel is used. Foremost, the idle threshold should be set relatively high to avoid having to balance work too often, which incurs extra overhead for multiple kernel launches and synchronization overhead. In

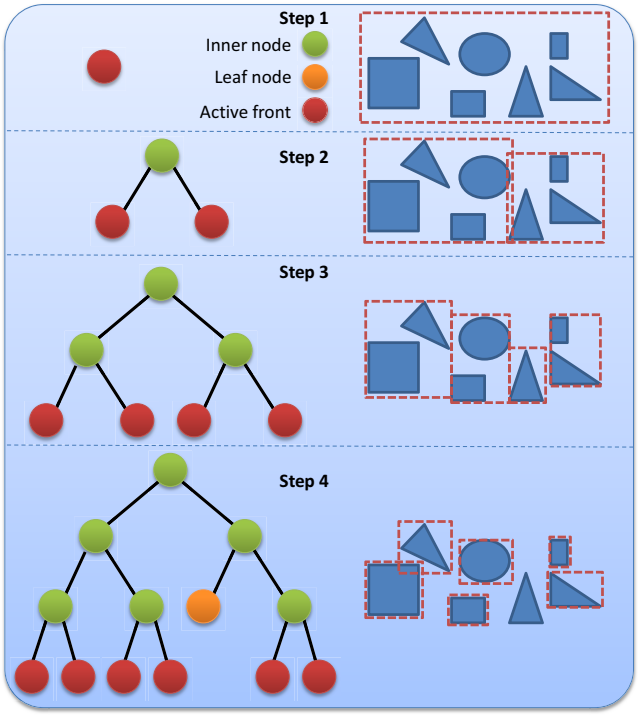

Figure 3: Hierarchy construction: *The front of active work units during several steps in the construction of an object hierarchy. Each entry in the front can be processed independently by parallel tasks, but subsequent steps depend on output from the previous one and thus require some degree of coordination.*

addition, the number of tasks launched in parallel should exceed the number of cores on the GPU, taking into account that each can run multiple tasks thanks to hardware multi-threading. Therefore, even if a part of the tasks is idle, others running on the same core may still be active. The optimal number of tasks is the one that assures that all GPU cores are scheduled enough tasks as they can handle using hardware multithreading.

4 Applications

We use several benchmark applications with different properties to evaluate the different GPU work distribution techniques.

4.1 Hierarchy construction

Object hierarchies are widely used in many applications such as computer graphics and collision detection. An object hierarchy is a tree of bounding volumes (such as boxes or spheres) with the main property that each node's bounding box contains all its childrens'. At the leaf nodes, nodes then refer to individual primitive geometric objects, such as triangles or points. Construction of these hierarchies usually progresses by recursively splitting groups of objects into two or more smaller groups until only single objects are left. Since the splits usually do not result in the same amount of primitives in all the subgroups, the height of the tree is not known in advance and the tree in general can be highly unbalanced. Thus, load balancing is critical to good performance in many applications.

This divide-and-conquer approach maps well to our algorithm since all split operations are fully independent and can thus be handled in parallel. We employ the same approach as in [Lauterbach et al. 2009] using axis-aligned bounding boxes (AABBs) where the main work kernel takes in a set of triangles, computes a split of the set into two new subsets according to a heuristic, resorts the primitives based on the split, generates the new bounding box and then may generate two new splits (or none, if it created a leaf node). The construction algorithm starts with only one split (the root node) and then increases parallelism geometrically until leaf nodes are constructed.

4.2 Sorting

An obvious candidate for recursive algorithms is Quicksort, which was also investigated in [Cederman and Tsigas 2008b]. While it is similar in structure to hierarchy construction since it starts with the whole list, then recursively subdivides, the main difference is that the main pivoting operation has much less computational intensity. Therefore, we expect the overhead in the work balancing method to be much more critical to overall performance. Unlike the hierarchy construction, we only recurse until the list of keys to be sorted is small enough to fit into local memory and then revert to a simpler sort for higher performance.

4.3 Collision detection

An application for object hierarchies is to check for collisions between two disjoint objects (inter-object collisions) as well as self collisions for deformable objects (intra-object collisions). The main operation in collision detection is, given several objects, to find whether and at which points they overlap. The objects consist of several geometric primitives (e.g. triangles), which reduces the problem to finding which of these triangles intersect. As a special case, self-intersection tests whether any of the primitives of the object intersect each other, which is of use for applications such as cloth simulation. The naïve solution to collision detection is to test every combination of primitives, which is not practical due to quadratic run-time. However, given a hierarchy built on top of each object's primitives, it is possible to intersect the hierarchies instead to find potential collisions and then only perform primitive-primitive intersection for those candidates.

Simultaneous hierarchy traversal: In general, both the geometric primitives as well as the bounding volumes used in the hierarchy can vary depending on the application and other criteria. However, the algorithm for intersecting hierarchies is the same. Starting with the two roots, two hierarchy nodes are intersected by analytically testing their respective bounding volumes for overlap. If they overlap, then all possible pairings of their children are recursively tested for intersection as well (see Fig. **??**.) If both of the nodes are leafs, then the two corresponding primitives are put on the list of potential intersections. If only one of the two is a leaf, then it is tested against the children of the other node.

From the standpoint of our approach, the main work units are pairs of hierarchy nodes and for a binary tree each intersection can generate up to four pairs per step. All intersection tests between the node of the hierarchy can be performed independently. We use the object hierarchy generated by the construction algorithm described above with AABB as the bounding volume as the input for our collision detection algorithm, then perform self-collision by intersecting the hierarchy with itself. As such, the mapping to our work model are relatively simple. Unlike the previous applications that use data parallelism in the actual work, the intersection kernel can instead be run using the vector units to process several intersections in parallel and push the resulting new intersection pairs on the work queue or in a separate result queue for actual primitive pairs. This way, we can handle many more parallel node intersections. After the hierarchy traversal, the overall list of primitive pairs is then used as input into a intersection test kernel that tests for actual intersection, using

the GPU's static work scheduler.

4.4 Generic recursion kernel

In order to evaluate performance of a generic recursive algorithm on different methods more directly, we also tested a generic work kernel that spawns up to t new work units and performs w work at each step by executing mathematical operations in w iterations of a loop. Unlike the other approaches, this kernel handles a separate work unit for each vector lane and thus for a vector size of v can actually create up to vt new work units, thus putting a much larger load on work organization. As a termination criterion, a random number in 0..1 is generated for each evaluation and compared to a user-specified termination probability p. If larger, then new work units are spawned, otherwise no new work is created.

This kernel allows to vary the computational load of a recursive algorithm, as well as the rate at which new work is generated and thus new work units may need to be distributed. This allows us to investigate a broader spectrum than presented by the two real-world applications and also test for architecture-specific performance characteristics. In order to have a fair comparison of different work distribution approaches, the random number generator is seeded in a way that the resulting recursion tree structure only depends on the initial seed and is thus reproducible.

5 Results and Analysis

5.1 Results

We now describe results from an implementation of the methods and applications described above on a real GPU architecture.

5.2 Implementation

We have implemented our approach using a Intel Core2 Duo system at 2.83 GHz on 4 cores. We are using CUDA on a NVIDIA GTX 285 GPU that has a total of 30 processing cores and 1 GB of memory. For all approaches and work kernels, we start more independent GPU tasks than actual cores to allow for hardware multithreading. The optimal number of tasks to fully occupy the GPU can be calculated by examining the resource utilization of the work kernels. Using our work balancing approach, we set the criterion for performing a balancing step to half the tasks being idle. Note that since we launch more tasks than cores this does not result in half the processing cores being idle.

We now demonstrate results from our implementation of the algorithms described in the previous section. We use several common benchmark scenes from previous work to allow easier comparison with other techniques (see Fig. 4). These scenes range from 40k to 146k triangles each and have relatively complex structure. For example, the Flamenco model has several cloth layers very close together, representing a hard case for culling in collision detection.

Fig. 5 shows our results for construction of an object hierarchy on several geometric triangle models ranging from 40k to 146k triangles. We use both axis-aligned bounding boxes (AABBs) as well as more complex and tighter-fitting oriented bounding boxes (OBBs) for both approaches. For OBB construction, similar to previous approaches we perform splitting with axis-aligned planes first and then fit the OBB around the triangles in a post-processing step similar to refitting using the PCA approach from [Gottschalk et al. 1996]. We compare the impact of our approach for hierarchy construction compared to previous compaction-based implementations [Zhou et al. 2008; Lauterbach et al. 2009], but improvements are relatively minor at about 10-13% faster timings.

Sorting peformance using Quicksort is shown in Fig. 6. We use the implementation on several randomly generated datasets with increasing number of values to sort. The results show that our minimal work balancing approach performs much better here due to less overhead and contention in list access. To compare absolute performance, we also compare against a state-of-the-art sorting algorithm presented in [Satish et al. 2009] based on optimized radix sort. Similar to results in [Cederman and Tsigas 2008b], we observe that Quicksort is still somewhat slower in practice. Similar to construction, the main problem is that the initial steps of the Quicksort algorithms cannot use full parallelism since only one or very few partitions are available. In addition, these partitions are most likely contain the largest number of values to sort and thus take the most time to process.

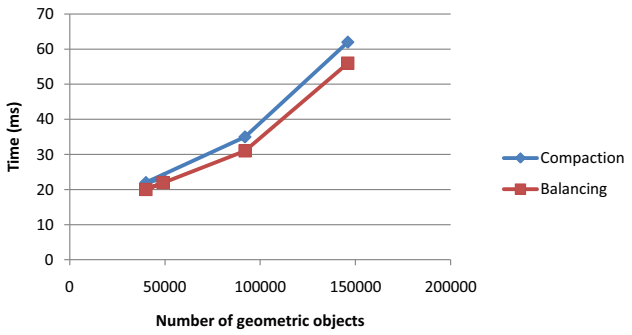

Figure 5: Hierarchy construction: *Timings of our approach (in ms) for parallel hierarchy construction, using AABBs or OBBs as bounding volumes. Results are relatively close since the overall time is dominated by the computationally intensive work kernels, but our approach performs about 5-10% faster than the compaction-based implementation in previous work.*

Figure 6: Quicksort: *Timings of our sorting a list of n values (in ms) for different work balancing algorithms, as well as for a fast radix sort algorithm.*

5.3 Analysis

Finally, we look at the performance of our generic recursion kernel for different parameters for branching factor t and termination probability p controlling how much total work units are generated, as well as work per kernel w which directly influences the computational load versus time spent generating and distributing work. The results are summarized in Fig. **??**. In general, we see that, as expected, an increase in w decreases the impact of work distribution method as more of the total time is spent doing actual work. In the opposite case, we observe that for smaller work units our work balancing method performs better since it requires less synchronization. Interestingly, we also see that the absolute performance does not change much up to some value of w, which suggests that up to a point additional computational intensity is almost free, which has interesting applications for algorithm design choices.

When changing t and p to increase the amount of work (see Fig. **??** b)), we see that all implementations except compaction scale well. The graphs also show some sub-linear scalability for work stealing and our approach. We ascribe that to higher performance in the late stages of processing when all tasks have sufficient work to perform without needing to perform work on synchronization at all, whereas both global work queues and compaction always have the same overhead.

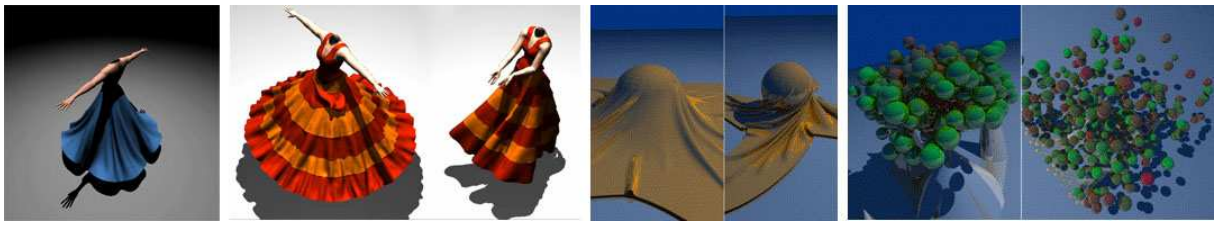

Figure 4: Benchmarks: *The benchmark scenes for construction used in this paper. Top left: Flamenco (49K triangles), top right: Princess (40K), bottom left: Cloth dropping on sphere (92K) and bottom right: n-body simulation (146K). Our algorithm can perform interactive self-collision using continuous triangle intersection on all of these models.*

Our approach has some limitations worth noting. For one, aborting the work kernel occasionally incurs both overhead from additional kernel calls (about on the order of 3-7 μs on current architectures [Volkov and Demmel 2008]), time spent idling when waiting for the last active tasks to complete as well as the actual time spent in the balancing kernel which is directly proportional to the number of work units in all queues. In our applications, individual work units are very small (e.g. 8-16 bytes) and reordering them during balancing is relatively cheap. For applications with larger units, it would be advisable to reorder indices instead.

6 Conclusion and future work

6.1 Conclusion and future work

We have presented a lightweight work balancing technique for use on GPU architectures and compared it against several previously used methods on a variety of applications. Our results show that traditional methods work for applications with very computationally intense kernels, but are slow when the overall runtime is not dominated by the computation and requires frequent synchronization operations. In contrast, our method – while incurring some overhead – provides faster overall performance during work execution due to its lightweight nature.

We are interested in investigating other applications that have a recursive nature. One candidate is collision detection where objects represented by hierarchies are tested for overlap using a recursive hierarchical traversal. Due to the computational intensity of intersection tests between primitives especially when using complex bounding volumes and continuous intersection, GPUs should be a good candidate for performing the intersection tests. Another application would be other recursive search problems such as chess and other planning algorithms that are implemented recursively.

References

ARORA, N. S., BLUMOFE, R. D., AND PLAXTON, C. G. 1998. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, ACM, New York, NY, USA, 119–129.

CEDERMAN, D., AND TSIGAS, P. 2008. On dynamic load balancing on graphics processors. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 57–64.

CEDERMAN, D., AND TSIGAS, P. 2008. On sorting and load balancing on gpus. *SIGARCH Comput. Archit. News 36*, 5, 11–18.

CHASE, D., AND LEV, D. 2005. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, ACM, New York, NY, USA, 21–28.

FOLEY, T., AND SUGERMAN, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proc. ACM SIGGRAPH/EG Conf. on Graphics Hardware*, 15–22.

FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. 1998. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not. 33*, 5, 212–223.

GOTTSCHALK, S., LIN, M., AND MANOCHA, D. 1996. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. of ACM Siggraph'96*, 171–180.

HA, P. H., TSIGAS, P., AND ANSHUS, O. J. 2008. Wait-free programming for general purpose computations on graphics processors. In *IPDPS*, IEEE, 1–12.

HENDLER, D., AND SHAVIT, N. 2002. Non-blocking steal-half work queues. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, ACM, New York, NY, USA, 280–289.

HENDLER, D., LEV, Y., MOIR, M., AND SHAVIT, N. 2006. A dynamic-sized nonblocking work stealing deque. *Distrib. Comput. 18*, 3, 189–207.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree GPU raytracing. In *Proc. I3D '07*, 167–174.

KYLE HEGEMAN, N. A. C., AND MILLER, G. S. 2006. Particle-based fluid simulation on the gpu. 228–235.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. In *Proc. Eurographics '09*.

LEFEBVRE, S., AND HOPPE, H. 2006. Perfect spatial hashing. In *SIGGRAPH '06*, ACM, New York, NY, USA, 579–588.

LEFEBVRE, S., AND HOPPE, H. 2007. Compressed random-access trees for spatially coherent data. In *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)*, Eurographics.

LEFOHN, A., KNISS, J. M., STRZODKA, R., SENGUPTA, S., AND OWENS, J. D. 2006. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics 25*, 1 (Jan.), 60–99.

PURCELL, T., DONNER, C., CAMMARANO, M., JENSEN, H., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, 41–50.

SATISH, N., HARRIS, M., AND GARLAND, M. 2009. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 1–10.

VOLKOV, V., AND DEMMEL, J. W. 2008. Benchmarking gpus to tune dense linear algebra. 1–11.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. In *Proc. SIGGRAPH Asia*.