Lazy Work Stealing for Continuous Hierarchy Traversal on Deformable Bodies

Vinícius da Silva, Claudio Esperança and Ricardo Guerra Marroquim
COPPE, Federal University of Rio de Janeiro, Horácio Macedo Avenue 2030, Rio de Janeiro, Brazil

Keywords: Dynamic Load Balance, Continuous Collision Detection, Deformable Bodies, GPGPU.

Abstract: This study presents the results of research in dynamic load balancing for Continuous Collision Detection (CCD) using Bounding Volumes Hierarchies (BVHs) on Graphics Processing Units (GPUs). Hierarchy traversal is a challenging problem for GPU computing, since the work load of traversal has a very dynamic nature. Current research resulted in methods to dynamically balance load as the traversal is evaluated. Unfortunately, current grid-based GPU computing interfaces are not well suited for this type of computing and load balancing code can generate excessive overhead. This work presents a novel algorithm to address some of the most glaring problems. The algorithm uses the new concept of lazy work stealing, which tries to get the most out of the parallel capabilities of GPUs by greedy work stealing and lazy work evaluation. Also, the algorithm is designed to augment shared memory usage per block and diminish CPU-GPU context exchange penalties.

1 INTRODUCTION

Collision Detection (CD) is the research area that studies the problem of intersecting bodies. This problem arises in several fields, such as graphics and physics simulations and robot motion. Several approaches to the problem have been proposed and they can be classified as either Discrete Collision Detection (DCD) or Continuous Collision Detection (CCD).

CCD considers the entire time interval between frames while evaluating collision and is widely used in recent works. DCD considers just a moment of the interval and, thus, can miss collisions. CCD is more costly than DCD (Provot, 1997), however, unlike DCD, it cannot miss collisions inside the interval and, thus, is more precise.

CCD is becoming more affordable by applications because of the increasing of parallel capabilities of processors, allied with the rise in popularity of General Purpose Graphics Processing Units (GPGPU) (Nickolls and Dally, 2010). There are several development interfaces for GPGPU architectures available, but this work uses the definitions of (NVIDIA, 2012).

Bounding Volume Hierarchies (BVHs) are commonly used to accelerate CD. BVHs can be classified according to the type of Bounding Volumes (BVs) used. A common choice is the OBB-tree, that is, a binary tree where each node contains an Oriented Bounding Box (OBB) enclosing ever finer parts of the model or scene.

Contributions. The contribution of this work is the presentation of a novel load balancing algorithm for hierarchy traversal, based on the task stealing load balancing approach discussed by Cederman and Tsigas (Cederman and Tsigas, 2009). The algorithm's performance is evaluated on two systems employing an NVIDIA GeForce GT520 and an NVIDIA GeForce GTS450 GPU. It is also compared with the load balancing scheme used in the gProximity approach (Lauterbach et al., 2010).

1.1 Related Work

CD is a widely studied field and great overviews can be found in (Teschner et al., 2004) and (Ericson, 2004). The research on BVH based CCD in recent years is mainly divided between two paths of optimization ideas: parallelism and culling.

Different algorithms have been proposed to parallelize traversal work among threads in CPU, between CPU and GPU in hybrid systems, and among threads purely in the GPU. (Kim et al., 2009c) parallelized traversal work among threads in CPU by doing a breath-first serial traversal until a hierarchy level with enough nodes to feed all threads is found. The work of (Tang et al., 2010b) distributes the nodes of

Silva V., Esperança C. and Guerra Marroquim R..
Lazy Work Stealing for Continuous Hierarchy Traversal on Deformable Bodies.
DOI: 10.5220/0004656904300437
In *Proceedings of the 9th International Conference on Computer Graphics Theory and Applications* (GRAPP-2014), pages 430-437
ISBN: 978-989-758-002-4

the Bounding Volume Test Tree (BVTT) front among CPU threads to achieve better parallelism.

Another approach in (Kim et al., 2009a) explores GPUs to perform the elementary tests in a hybrid CPU-GPU environment. Lauterbach et al. (Lauterbach et al., 2010) uses one GPU kernel to traverse the tree and another to balance the workload. In recent work, (Tang et al., 2011) extended these ideas by providing an algorithm for stream registration of the data generated during traversal, and using a deferred front tracking approach to lower the BVTT front memory overhead.

Another related area intensely investigated in the last years is the design of culling methods for BVH-based CCD. The main motivation is to diminish the overhead generated by the huge number of false positives and redundant computations that pure BVH traversal can generate. The efforts are divided in high-level and low-level culling algorithms. High-level algorithms are heuristics designed to discard portions of the hierarchy early while traversing. Several works can be found in the subject. In (Heo et al., 2010), surface normals and binormal cones are used to discard parts of the geometry while traversing the hierarchy, while (Tang et al., 2008) creates the concept of Continuous Normal Cones by extending the Normal Cones to CCD.

On the other hand, low-level culling algorithms focus on discarding redundant and false-positive primitive tests.(Tang et al., 2010a) uses Bernstein polynomials to formulate a non-penetration filter which detects if a feature pair cannot be coplanar on the entire time interval and thus cannot intersect. In another work (Tang et al., 2008), a set of all the meaninful elementary tests generated by adjacent triangles is precomputed, obviating the evaluation of their collision on traversal.

The work in (Curtis et al., 2008) diminishes the number of redundant primitive tests by extending the triangle representation to carry feature assignment info.

2 BOUNDING VOLUME HIERARCHIES

The main idea behind BVHs is to approximate geometry by the coarse Bounding Volumes (BVs) representation, for which collision can be evaluated easily. Oriented Bounding Boxes (OBBs (Gottschalk et al., 1996)) are reported as a good choice for use in GPUs because they provide a better cost-benefit in respect with culling efficiency and computational overhead (Lauterbach et al., 2010).

The hierarchy traversal is the operation that actually evaluates collision and generates a potentially colliding primitive pair list. To finish the collision detection, classic edge-edge and vertex-face elemental tests, are evaluated for each triangle pair in the potentially colliding primitive pair list. For this, linear motion of each vertex between frames is assumed. The cubic equations are described in (Provot, 1997).

To achieve better performance, the prototypes associated with this work implement two culling methods. First, the Non-penetration Filter algorithm (Tang et al., 2010a) is used. It eliminates the need for solving the cubic equation of a vertex-face pair (p,t) when point p remains on the same side of triangle t during all time interval. Similarly, an edge-edge pair can be culled if they cannot be coplanar during the time interval. Second, the concept of Orphan Sets defined in (Tang et al., 2008) is used to ignore adjacent triangle intersections in traversal.

2.1 Front-based Decomposition

The prototypes associated with this work implement the concept of front-based decomposition (Tang et al., 2010b). This concept allows the usage of time coherence in BVH traversal by saving the front of the traversal's Bounding Volume Test Tree (BVTT). This front can be used as the starting point for the traversal in the next frame, saving computational time and providing more parallelism. The front can be acquired while in traversal by saving the pairs of BVH nodes in which the traversal ends. These pairs are those whose collision is evaluated as false or leaf pairs. The concepts are shown in Figure 1.

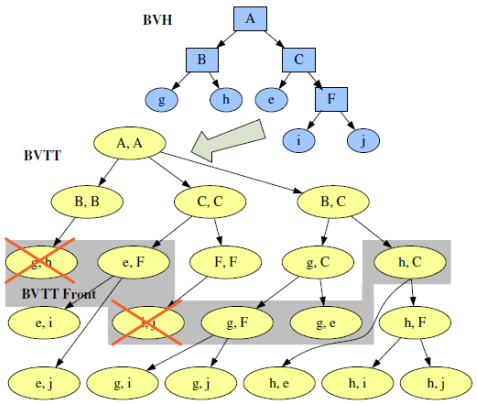

Figure 1: BVH and associated BVTT for an intra-collision traversal. The marked nodes in the BVTT are the front, i.e., the nodes where the collision traversal ended. As shown in (Tang et al., 2010b).

The front saved from previous frame can be used as the starting point for the traversal in the current

frame. However, while the simulation proceeds, the adition of BVTT front nodes can turn it outdated. Better heuristics to detect an outdated front are model dependent and empirical. Rebuilding the front means traversing the hierarchy again from the root.

3 LOAD BALANCING ON BOUNDING VOLUME HIERARCHY TRAVERSAL

The main difficulty that arises when designing CCD BVH traversal algorithms for GPUs is the fact that current grid-based GPU computing interfaces are not well suited for applications where the workload is not known a priori. BVH traversal falls in this category since it is not possible to predict the traversal before evaluating it and the process generates work on-the-fly, requiring dynamic load balancing.

Four different methods for load balancing on GPUs are discussed in (Cederman and Tsigas, 2009): Static Task List, Blocking Dynamic Task Queue, Lock-free Dynamic Task Queue and Task Stealing. Static Task List is the simplest load balancing scheme. Balancing occurs before issuing all work, thus it is inherently inflexible. The algorithm proposed in (Lauterbach et al., 2010) tries to address the inflexibility of the Static Task List by redistributing nodes if the number of unused cores is higher than a predefined threshold.

The Task Stealing algorithm (Arora et al., 1998) was designed to balance load among processes in a multiprocessor system. It uses double ended queues (deques) to fulfill this task. Each process has its own deque from which it can pop or push threads. This way, the process can acquire work (pop) or generate work dynamically (push). These operations are done on the bottom side of the deque. When a process finishes its job, it tries to steal from the deques of other processes. This operation is done on the top side of the deque, for the sake of parallelism. Figure 2 shows the concept. (Cederman and Tsigas, 2009) produced good results when adapting the Task Stealing algorithm (Arora et al., 1998) to balance the load generated by the parallel quicksort algorithm on GPUs.

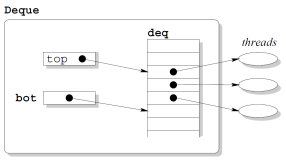

Figure 2: Deque scheme of (Arora et al., 1998). As shown in the paper.

4 LAZY WORK STEALING

In this section the novel Lazy Work Stealing algorithm is presented. It is based on the approach described in (Cederman and Tsigas, 2009) for Task Stealing. The main goal of the algorithm is to be more flexible than other methods by diminishing the overhead of load balancing and thus freeing GPU to do the actual work, i.e. the traversal itself. Some optimization ideas guide our novel algorithm.

1. Ideally, a block should be able to acquire work with a minimal performance hit on other working blocks.
2. We want to minimize balancing calls and CPU-GPU context change overhead.
3. Maximum available shared memory per block. The front update pass (Tang et al., 2010b), can benefit greatly from using shared memory, since the method is memory bound.

Since Task Stealing is on-demand in essence, the problem of performance hits on working blocks is addressed. Also, the CPU-GPU context change is diminished, since the task stealing code must be incorporated into the traversal kernel. The problems related with shared memory are also solved by setting the number of launched blocks for the traversal kernel as a number of blocks that can actually run in parallel on the device, and thus promote optimal device occupancy. This way, we achieve maximum shared memory per block and avoid resource waste, since all multiprocessors will be busy. It is important to note that even if a kernel can be launched with a huge number of blocks, just a few reside at the same time on the device, and this information can be queried on all current CUDA enabled devices.

Unfortunately not all problems are solved using this approach. The original Task Stealing approach (Arora et al., 1998) requires that all blocks with empty deques keep pooling another block deques in a round robin fashion, attempting to steal nodes. In this case, even if the multiprocessor is busy, it may not actually be doing useful work. Thus, we propose an improved method by performing modifications to better suit the stealing part of the algorithm to GPUs:

1. A three-pass approach is used for node management since the amount of shared memory per block can be augmented. In each traversal loop iteration, each thread pops one node, evaluates it and all generated work nodes or front nodes are saved in local thread memory (pass 1). Continuing in the same loop iteration, all nodes generated in a block are saved in a shared memory stack. This is achieved by using a prefix-sum (Sengupta

et al., 2007) approach (pass 2). Ending the iteration, part of the shared stack is pushed to global memory if the shared stack is nearly full (pass 3). The algorithm is described in depth in Algorithm 1.

2. In the pop pass (pass 1), a three-level pop approach is also used. The first pop attempt is on the shared stack (level 1). If unsuccessful, an attempt to pop from the block's deque is done (level 2). If it fails, work is stolen from the deques of other blocks (level 3) and the block resumes traversal. If no work can be stolen, the traversal ends. The overall scheme is presented in Figure 3. The algorithm is described in depth in Algorithm 2. Level 2 and 3 pop functions are explained in more detail in Algorithms 3 and 4.
3. The operations in all deques are done in batch to avoid excessive increments on global memory deque pointers. Just one thread in a block is responsible for doing the actual operation on deque pointers. The other threads just help to push or pop the affected nodes. In the algorithm listings, the functions *pop*, *popTop* and *popBot* are only changing pointers, i.e., they don't actually transfer the nodes.
4. A lazy approach is used to acquire stolen nodes (pop level 3). Since a block has a reasonable number of threads running in parallel, the stealing part of the algorithm pops all deques at the same time, each thread changing the top pointer of one global deque. However, just the nodes from the first stolen deque are actually transferred to the block stack at this time. The transfer is done in little batches to ensure saving shared stack size for traversal. To achieve this lazy transfer, it is necessary to save the deque indices and sizes of all successful pop operations of a task steal, and forbid deque pointers to be reset when a deque is empty. It is important to note that these changes aim at maximizing stolen nodes in a stealing attempt and maximizing stack size available for future traversal. Specifically, if a block b_i successfully pops on the top side of blocks b_j and b_k, saving stealing info (indices and sizes) to p_{ij} and p_{ik} and $j < k$, then nodes from global deque d_k are transferred to local stack s_i only after all deque d_j nodes are transfered, processed and s_i as well as deque d_i become empty again. The algorithm is described in depth in Algorithm 4.

In the algorithm listings, the function *assignShared()* is used to denote one assignment for the entire block, i.e., only one thread of the block actually alters the variable. Also, when a variable is declared as *volatile*, it must bypass incoherent cache memory.

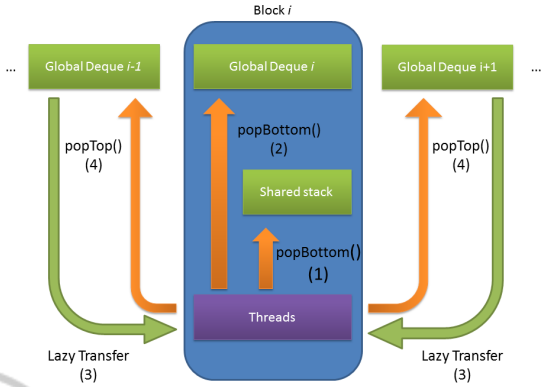

Figure 3: Lazy work acquisition scheme. First, threads try to pop from their block's shared stack (1). If not successful, they try to pop from its global deque (2). If not successful, enough nodes (if any) are transferred from lazily stolen nodes (3). If there are no more nodes from lazy steal, all other block deques are stolen (4).

5 EXPERIMENTS

The algorithm was tested in two systems. The first uses a GeForce GTS 450 card, which has 1 GB memory, 57.7 GB/s memory bandwidth and 4 SMs (streaming multiprocessors), each one with 48 CUDA cores, giving a total of 192 CUDA cores. The second uses a GeForce GT 520 card, which has 1 GB memory, 14.4 GB/s memory bandwidth and 1 SM, giving a total of 48 CUDA cores. Several commonly benchmarks were used, namely the BART, Funnel and Cloth/Ball models. The BART benchmark evaluates inter-collisions, and we have used four different resolutions (64, 256 , 1024 and 4096 triangles). The Funnel benchmark has 18.5K triangles and tests self-collision in deformable motion. Cloth/Ball benchmark evaluates the same type of collision, but has a heavier workload with 92K triangles. Linear motion between frames is assumed for all vertices. Figure 4 shows the benchmark scenes.

Table 1 shows the performance results for each benchmark. The numbers include time for hierarchy refit, front update, front-based traversal, load balancing and triangle pair intersection. The implementation uses the non-penetration filters (Tang et al., 2010a) and orphan sets (Tang et al., 2008) culling methods. Table 2 shows the speedup of the GTS 450 in relation with the GT520 and demonstrates how the algorithm scales. The calculation of speedup is just the ratio between the columns GT 520 and GTS 450 in table 1.

Table 3 shows the average number of processed

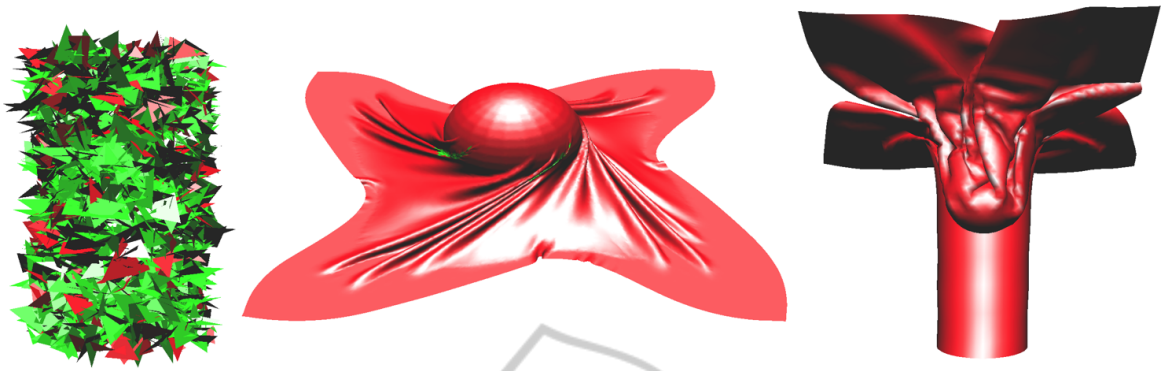

Figure 4: Continuous Collision Detection results using the Lazy Work Stealing algorithm. The collided faces are shown in green. From left to right, the benchmarks are: BART, funnel, cloth/ball.

Algorithm 1: Novel algorithm traversal and front acquisition. 1) Node acquisition (popWork). 2) Work compaction (compactStacks). 3) Push from shared stack to global deque if necessary (pushWork).

```
1:  function TRAVERSE(gBvtt , gFront)
2:      Parameters (global memory):
3:          gBvtt                ▷ BVTT work deques
4:          gFront               ▷ BVTT front deques
5:      Block shared memory:
6:          sBvtt                ▷ BVTT work shared data
7:          sFront               ▷ BVTT front shared data
8:      Thread local memory variables:
9:          tBvtt[3]             ▷ generated work nodes
10:         tFront               ▷ generated front node
11:         t   ▷ Thread index. Used in all algorithms
12:         b   ▷ Block index. Used in all algorithms
13:     loop
14:         if NOT(POPWORK(gBvtt , sBvtt , tBvtt))
    then
15:             PUSHREMAININGFRONT
16:             return TRAVERSAL_END
17:         if NOT(tBvtt.empty) then
18:             EVALCOLLISION(tBvtt , tFront)
19:         COMPACT(sBvtt,tBvtt,sFront,tFront)
20:         if NOT(PUSHWORK(gBvtt , sBvtt) then
21:             return OVERFLOW
22:         if NOT(PUSHWORK(gFront , sFront)
    then
23:             return OVERFLOW
24:     end loop
25: end function
```

front nodes per frame. This information depicts well the resources necessary to compute CCD for high density models and the differences in timings among benchmarks.

Algorithm 2: Novel lazy steal popWork device function. Each thread tries to pop a node in a 3 level approach until successful or all deques are inactive.

```
1:  function POPWORK(gBvtt , sBvtt , tBvtt)
2:      Block shared memory:
3:          pFlag[3] ▷ Pop results for each level pass
4:          pStr[NDEQUES]        ▷ Saved pop starts
5:          pSz[NDEQUES]         ▷ Saved pop sizes
6:          v          ▷ Current work stealing victim
7:          nDeques              ▷ Number of deques
8:      if t = firstThread then                ▷ Level 1
9:          stack ← sBvtt.stack
10:         pFlag[L1] ← stack.POP(pStr[b],pSz[b])
11:         pFlag[L3] ← bPopFlag[L1]
12:     SYNC
13:     if pFlag[L1] then
14:         if t < pSizes[b] then
15:             tBvtt ← sBvtt.stack[pStr[b]+t]
16:     else
17:         LEVEL2POP(gBvtt,pFlag,pStr,pSz,v)
18:         SYNC
19:         if NOT(pFlag[L2]) then             ▷ Steal
20:             LEVEL3POP(gBvtt,pFlag,pStr,pSz,v)
21:         SYNC
22:         if pFlag[L3] AND t < pSz[b] then
23:             tBvtt ← gBvtt.deque[v][pStr[b]+t]
24:     return pFlag[L3]
25: end function
```

6 COMPARISON AND ANALYSIS

The Lazy Work Stealing algorithm is compared with gProximity (Lauterbach et al., 2010) in an implementation done by the author of this paper. Table 4 shows the comparision of performance timings of the Lazy Work Stealing algorithm and gProximity for

Algorithm 3: Level 2 pop function. Pop from the deque owned by the block.

```
1: function LEVEL2POP(gBvtt,pFlag,pStr,pSz,v)
2:     Global memory:
3:         gBvtt.active[] is volatile
4:     if t = firstThread then
5:         d ← gBvtt.deque[b]
6:         pFlag[L2] ← d.POPBOT(pStr[b],pSz[b])
7:         if pFlag[L2] then
8:             v ← b
9:         else
10:            gBvtt.active[b] ← INVALID
11:            pStr[b] ← ∞
12:            v ← ∞
13:        pFlag[L3] ← pFlag[L2]
14: end function
```

Algorithm 4: Level 3 pop function. Lazy transfer if there are available nodes from last steal. Otherwise steal from all active blocks. Signs for traversal end if all blocks are inactive.

```
1: function LEVEL3POP(gBvtt , pFlag , pStr , pSz , v)
2:     Global memory:
3:         gBvtt.active[] is volatile
4:     Block shared memory:
5:         activeDequesFlag      ▷ Mark if there are active blocks.
6:     if t < nDeques then
7:         if pStr[t] ≠ INVALID then
8:             v ← ATOMICMIN(v , t)
9:             pFlag[L3] ← TRUE
10:    SYNC
11:    if t = v then
12:        pStr[b] ← pStr[t]
13:        pSz[b] ← pSz[t]
14:        pStr[t] ← INVALID
15:    while NOT(pFlag[L3]) do          ▷ No transfers
16:        SYNC
17:        ASSIGNSHARED(activeDequesFlag,FALSE)
18:        SYNC
19:        if t < nDeques then
20:            if gBvtt.active[t] = TRUE then
21:                activeDequesFlag ← TRUE
22:                d ← gBvtt.deque[t]
23:                if d.POPTOP(pStr[t] , pSz[t]) then
24:                    gBvtt.active[b] ← TRUE
25:                    v ← ATOMICMIN(v , t)
26:                    pFlag[L3] ← TRUE
27:        SYNC
28:        if activeDequesFlag = FALSE then
29:            break
30:        if t = v then
31:            pStr[b] ← pStr[t]
32:            pSz[b] ← pSz[t]
33:            pStr[t] ← INVALID
34:        SYNC
35: end function
```

Table 1: Lazy Work Stealing performance results. Times in ms.

Model	Triangles	GTS 450	GT 520
BART64	64	2.6	1.6
BART256	256	3.4	3.5
BART1024	1024	8.4	19.7
BART4096	4096	51.3	169.9
Funnel	18.5K	17.4	48.2
Cloth/Ball	92K	77.3	225.7

Table 2: Lazy Work Stealing speedup on GTS 450 in relation with the GT 520. Calculation is the ratio of GT 520 and GTS 450 columns of Table 1.

Model	Triangles	Speedup
BART64	64	0.61
BART256	256	1.03
BART1024	1024	2.34
BART4096	4096	3.31
Funnel	18.5K	2.77
Cloth/Ball	92K	2.92

the GT520 systems for all benchmarks. The numbers include time for hierarchy refit, front update, front-based traversal, load balancing and triangle pair intersection. Table 5 shows the same comparison for the GTS450 system. Finally, 6 compares the speedup of the algorithms. The speedup in this case is the ratio of the GTS 450 timings and the GT 520 timings for each benchmark. This demonstrates how the algorithms scale.

It is important to note that both algorithms have similar code for refit and triangle intersection. The only noticeable changes are in the load balancing, front update and traversal code. Also, both implementations use the same culling methods.

Based on the collected data, the algorithms seem to have similar performance for the analyzed benchmarks. Which algorithm performs better depends on the benchmark and system. Both algorithms seem to scale well. The GTS 450 system has 4 times more resources than the GT520 system and both systems have a near 3 times speedup, with gProximity having a peak of 3.47 speedup for the Cloth/Ball benchmark. The BART64 and BART256 examples showed poor scalability because of the lack of work available. Also it was observed that most of the time for BVH management is spent with the front update. This is mainly due to the number of nodes that the front may have. For instance, the peak in the number of processed front nodes in a frame pass 10 million for the Cloth/Ball benchmark.

Table 3: Lazy Work Stealing average number of processed front nodes per frame. Numbers in K nodes per frame.

Model	Front nodes
BART64	0.807
BART256	9.159
BART1024	127.394
BART4096	2,156.260
Funnel	493.391
Cloth/Ball	2,482.040

Table 4: Comparison of the Lazy Work Stealing (LWS) algorithm and gProximity timings on GT 520. Times in ms.

Model	Tris	LWS	gProximity
BART64	64	1.6	1.6
BART256	256	3.5	3.1
BART1024	1024	19.7	14.7
BART4096	4096	169.9	173.4
Funnel	18.5K	48.2	49.0
Cloth/Ball	92K	225.7	238.3

Table 5: Comparison of the Lazy Work Stealing (LWS) algorithm and gProximity timings on GTS 450. Times in ms.

Model	Tris	LWS	gProximity
BART64	64	2.6	3.1
BART256	256	3.4	3.6
BART1024	1024	8.4	7.5
BART4096	4096	51.3	47.6
Funnel	18.5K	17.4	16.3
Cloth/Ball	92K	77.3	68.6

Table 6: Comparison of the Lazy Work Stealing algorithm and gProximity speedup on GTS 450 in relation with the GT 520.

Model	Tris	LWS	gProximity
BART64	64	0.61	0.51
BART256	256	1.03	1.16
BART1024	1024	2.34	1.96
BART4096	4096	3.31	3.64
Funnel	18.5K	2.77	3.00
Cloth/Ball	92K	2.92	3.47

6.1 Limitations

The proposed algorithm has some limitations. First, it is highly memory bound. Thus, the performance is very dependent of the device memory bandwidth, mainly because of the front update pass. Second, the best size of the shared stacks depends on the model. If the model has a lighter workload, setting a higher stack size can forbid nodes to get to the global deques and in consequence forbid blocks to steal work. Analogously, if the model has a heavier workload, setting a lesser stack size can forbid blocks to benefit from the performance of the shared memory latency. Finally, the requirements of global memory size are higher when compared with methods that use deques that can be reset.

7 CONCLUSIONS AND FUTURE WORK

In this work, the Lazy Work Stealing algorithm for load balance of continuous collision detection on GPUs is presented. The algorithm relies on heavy usage of device shared memory to diminish the overhead of node management on traversal. Also, it tries to diminish work acquisition overhead using a greedy steal, lazy transfer approach.

The immediate plans for future work include the implementation of more culling methods, such as Representative Triangles (Curtis et al., 2008) and Continuous Normal Cones (Tang et al., 2008), to achieve better performance. Another path of development is to test the algorithm with more modern GPUs, such as NVidia's Kepler compute architecture.

In addition, a deeper comparison of Lazy Work Stealing with other load balancing approaches could be an interesting topic as well as the usage of this load balancing algorithm on other problems where work loads depend on the geometry, such as Ray Tracing.

ACKNOWLEDGEMENTS

The authors would like to thank the GAMMA research group at the University of North Carolina at Chapel Hill and Jonas Lext, Ulf Assarsson, and Tomas Möller of the Chalmers University of Technology for making available the benchmarks used in this work.

REFERENCES

Arora, N. S., Blumofe, R. D., and Plaxton, C. G. (1998). Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129, New York, NY, USA. ACM.

Cederman, D. and Tsigas, P. (2009). On sorting and load balancing on GPUs. *SIGARCH Comput. Archit. News*, 36(5):11–18.

Curtis, S., Tamstorf, R., and Manocha, D. (2008). Fast collision detection for deformable models using representative-triangles. In Haines, E. and McGuire,

M., editors, *Proceedings of the 2008 Symposium on Interactive 3D Graphics, SI3D 2008, February 15-17, 2008, Redwood City, CA, USA*, pages 61–69. ACM.

Ericson, C. (2004). *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Gottschalk, S., Lin, M. C., and Manocha, D. (1996). Obbtree: a hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 171–180, New York, NY, USA. ACM.

Group, K. (2012). The opencl specification version: 1.2 document revision: 19.

Heo, J.-P., Seong, J.-K., Kim, D., Otaduy, M. A., Hong, J.-M., Tang, M., and Yoon, S.-E. (2010). Fastcd: Fracturing-aware stable collision detection. In Popovic, Z. and Otaduy, M. A., editors, *Proceedings of the 2010 Eurographics/ACM SIGGRAPH Symposium on Computer Animation, SCA 2010, Madrid, Spain, 2010*, pages 149–158. Eurographics Association.

Kim, D., Heo, J.-P., Huh, J., Kim, J., and Yoon, S.-E. (2009a). HPCCD: Hybrid parallel continuous collision detection using cpus and gpus. *Computer Graphics Forum (Pacific Graphics)*.

Kim, D., Heo, J.-P., Huh, J., Kim, J., and Yoon, S.-E. (2009b). Hpccd: Hybrid parallel continuous collision detection using cpus and gpus. Computer Graphics Forum (Pacific Graphics).

Kim, D., Heo, J.-P., and Yoon, S.-e. (2009c). Pccd: parallel continuous collision detection. In *SIGGRAPH '09: Posters*, SIGGRAPH '09, pages 50:1–50:1, New York, NY, USA. ACM.

Lauterbach, C., Mo, Q., and Manocha, D. (2010). gproximity: Hierarchical gpu-based operations for collision and distance queries. In *Proceedings of Eurographics 2010*.

Ni, T. (2009). Directcompute.

Nickolls, J. and Dally, W. J. (2010). The gpu computing era. *IEEE Micro*, 30(2):56–69.

NVIDIA (2012). *NVIDIA CUDA Programming Guide*.

Provot, X. (1997). Collision and self-collision handling in cloth model dedicated to design garments. In *Graphics Interface 97*, pages 177–179.

Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. (2007). Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, GH '07, pages 97–106, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Tang, M., Curtis, S., Yoon, S.-E., and Manocha, D. (2008). Interactive continuous collision detection between deformable models using connectivity-based culling. In *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 25–36, New York, NY, USA. ACM.

Tang, M., Manocha, D., Lin, J., and Tong, R. (2011). Collision-streams: Fast GPU-based collision detection for deformable models. In *I3D '11: Proceedings of the 2011 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 63–70.

Tang, M., Manocha, D., and Tong, R. (2010a). Fast continuous collision detection using deforming nonpenetration filters. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, pages 7–13, New York, NY, USA. ACM.

Tang, M., Manocha, D., and Tong, R. (2010b). Mccd: Multi-core collision detection between deformable models using front-based decomposition. *Graphical Models*, 72(2):7–23.

Teschner, M., Kimmerle, S., Zachmann, G., Heidelberger, B., Raghupathi, L., Fuhrmann, A., Cani, M.-P., Faure, F., Magnetat-Thalmann, N., and Strasser, W. (2004). Collision detection for deformable objects.