Bulk-Synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs

Citation for published version:
Chitalu, F, Dubach, C & Komura, T 2018, Bulk-Synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs. in *I3D '18 Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.*, 4, ACM, Montreal, Quebec, Canada, pp. 4:1-4:9, ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games
, Montreal, Canada, 15/05/18. https://doi.org/10.1145/3190834.3190848

Digital Object Identifier (DOI):
10.1145/3190834.3190848

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
I3D '18 Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games

Bulk-Synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs

Floyd M. Chitalu
University of Edinburgh
floyd.m.chitalu@ed.ac.uk

Christophe Dubach
University of Edinburgh
c.dubach@ed.ac.uk

Taku Komura
University of Edinburgh
tkomura@ed.ac.uk

ABSTRACT

Simultaneous BVH traversal, as a dynamic task of pair-wise proximity tests, poses several challenges in terms of parallelization using GPUs. It is a highly dynamic and data-dependent problem which can induce control-flow divergence and inefficient data-access patterns. We present a simple solution using the bulk-synchronous parallel model to ensure a uniform mode of execution, and balanced workloads across GPU threads. The method is easy to implement, fast and operates entirely on the GPU by relying on a topology-centred work expansion scheme to ensure large concurrent workloads. We demonstrate speedups of upto 7.1× over the widely used "streams" model for GPU based parallel collision detection.

CCS CONCEPTS

• **Computing methodologies → Collision detection**; **Massively parallel algorithms**; Shared memory algorithms;

KEYWORDS

collision detection, parallel computing, GPU, BVH, BSP

ACM Reference Format:
Floyd M. Chitalu, Christophe Dubach, and Taku Komura. 2018. Bulk-Synchronous Parallel Simultaneous BVH Traversal for Collision Detection on GPUs. In *I3D '18: I3D '18: Symposium on Interactive 3D Graphics and Games, May 4–6, 2018, Montreal, QC, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3190834.3190848

1 INTRODUCTION

Collision detection (CD) has a wide-spectrum of practical applications including physics based simulations, robotic motion planning, virtual disassembly, haptic rendering and ray-tracing. It is a well known and long studied problem of finding a number of interactions at low computational cost. As a result, collision detection is at the core of many applications in computer science and engineering today.

However, CD can be computationally expensive due to it potential for having vast workloads. A simple approach exhaustively testing for pair-wise intersection between geometry will not scale optimally due to the inherent $O(N^2)$ complexity. This particular constraint has lead to the common solution of using acceleration data structures such as bounding volume hierarchies (BVHs) [Ericson 2005]. BVHs attempt to reach an optimal case of $O(logN)$ by quickly culling the search-space of potential collisions.

In spite of this potential benefit, geometry may reach scales of tens- to hundreds-of-thousands of triangles or more which makes the prospect of employing BVHs alone insufficient. Furthermore, BVHs can also degenerate if the enclosed geometry is relatively small, such that traversing entire BVHs becomes a layer of overhead.

Previous methods tackling the problem of optimizing BVH based CD on the GPU [Du et al. 2015; Lauterbach et al. 2010; Tang et al. 2011, 2016] offer in-part successful but also complex solutions which can suffer from GPU under-utilisation. They emulate the logic of conventional single-threaded CPU traversal by relying on thread-level private work-stacks and temporal coherence [Li and Chen 1998]. Such heuristics-based optimisations can serve to complicate traversal logic and thus may constrain GPU performance. Work-stacks serve to reduce memory access and synchronisation costs. However, they are a source of control-flow divergence, and load-imbalance which is managed by a separate GPU task in the execution pipeline. Moreover, pre-existing solutions have used work-stacks to effectively mimic recursion on the GPU because threads perform traversal *in-place*: Evaluation of pair-wise tests in BVH sub-trees is computed independently as threads push and pop intermediate BVH node-pairs to-and-from work-stack memory which creates the divergence in control-flow as a side-effect. Temporal coherence on the other hand, has a high memory footprint since it is based on explicitly storing the BVH node-pairs where traversal terminates. It is also a potential source of work-flow divergence because simply checking when and how to store such node-pairs contributes to the overhead of branching on GPUs.

We present a simple approach to simultaneously traverse a large number of BVHs for CD in parallel and entirely on the GPU. The method is based on the Bulk-Synchronous Parallel (BSP) model [Valiant 1990] were traversal is reformulated as an iterative "fork and join" scheme to: (1) mitigate explicit load-balancing that requires using separate work-rebalancing tasks on the GPU, (2) minimise control-flow divergence by reducing the amount of work mapped to each thread and performing full-restarts from a user-specified entry level such as the root-level, and (3) allow for efficient memory access patterns that may be coalesced while seamlessly unifying synchronisation, communication and storage by relying on the BSP model. Our tests, which are performed on three UNC dynamic scene benchmarks (see Figure 6), also reveal up to 7.1× speedup over the "streams" model for GPU based CD from Tang *et al.* [2011], which is currently the standard model employed by others [Du et al. 2017, 2015; Tang et al. 2013, 2016].

Contributions. The contributions form a simple solution, from using the topological structure of BVHs and simplified thread-level operations for reducing control-flow divergence, to efficiently traverse multiple BVHs in parallel on the GPU:

I3D '18, May 4–6, 2018, Montreal, QC, Canada

- We present a novel algorithm (section 5) as alternative reformulation of simultaneous and parallel traversal of multiple BVHs for pair-wise CD on the GPU.
- Parametric workload expansion (subsection 5.2): *Adaptive depth-stepping* and *static workload expansion* are introduced as key features for ensuring large concurrent workloads and controlling the rate of traversal, using the topological structure inherent in the traversed BVHs.
- A lock-free scheme to write intermediate BVH node-pairs to global memory using iterative buffered-writes (subsection 5.3), which can be controlled based on the topological properties of BVHs and available hardware resources.

2 RELATED WORK

Collision Detection in Physics-based Animation. Collision detection lends itself well to physics-based simulation problems for real-time and off-line use-cases [Ericson 2005]. It has been particularly useful for large scale problems involving complex non-rigid objects such as cloth [Bridson et al. 2002; Brochu et al. 2012] were the complexity of interactions (including self-collisions) places emphasis on the need for efficient culling of triangle intersection tests which have a high computational cost. BVHs are a common data structure in many such works with their ability to quickly cull of the search space of potential interactions [Teschner et al. 2005]. Numerous approaches including axis-aligned bounding boxes (AABB) [Bergen 1997], oriented bounding boxes (OBB) [Gottschalk et al. 1996], discrete oriented polytopes (k-DOP) [Klosowski et al. 1998] have been introduced for this purpose, which function as approximations to the underlying geometric primitives that they enclose in the form of coarse bounding volumes.

Parallel Collision Detection. Methods to accelerate CD through parallelism on GPUs have been investigated for over a decade now. Early pioneering works such as that of Knott and Pai [2003] made use of the parallel rasterization capabilities of GPUs. Recent methods including Tang *et al.* [2016] and others [Tang et al. 2011, 2013; Weller et al. 2017; Wong et al. 2014] utilize the general purpose computational capabilities of modern GPUs to accelerate computation following the advent of parallel programming frameworks such as CUDA and OpenCL. Wong *et al.* [2014] present a parallel adaptive scheme combining octrees and hierarchical grid structures for broad-phase CD with deformable objects. Weller *et al.* [2017] recently introduce a CUDA based scheme, kDet, which is based on a hierarchical grid structures to find the set of potentially colliding pairs using polygon sizes.

In general, mapping BVH traversal to GPUs is recognised as a challenging task as demonstrated by prior efforts that have advocated for the use of more parallelism through many-core GPUs and multi-core CPUs [Lauterbach et al. 2010; Tang et al. 2010, 2016]. Since naïve approaches can easily result in hardware under-utilisation due to low workloads, the most influential methods such as Lauterbach *et al.* [2010] and other variants [Du et al. 2015; Tang et al. 2011, 2013, 2016] have relied on *front tracking* [Tang et al. 2010] for sustaining high workloads which is ideal for GPUs. In this approach, the *bounding volume test tree* (BVTT) [Gottschalk 2000] of BVH node-pairs where traversal terminates is explicitly cached and then used as input for next time. In addition, thread-level private work-stacks are another common feature in these methods, to improve memory access costs and minimise inter-thread synchronisation. However, work-stacks can lead to work-flow divergence and load-imbalances that require a separate GPU task to perform work redistribution between threads (see Lauterbach *et al.* [2009] and [2010] for details). Similar approaches have also been used in robotic motion planning [Pan et al. 2010; Pan and Manocha 2011]. The related work of Hermann *et al.* [2013] performs CD for motion planning using voxel maps maintained in GPU global memory.

Our method shares some similarities with these approaches but does not rely on work-stacks nor front-tracking. We adopt the BVTT as the primary input but distinctively express traversal as an iterative one-to-one mapping between threads and evaluated node-pairs. Further, we focus on the specific problem of pair-wise CD between BVH-nodes, whereas many of these approaches are focused on parallelizing the entire CD pipeline. Another distinction is that these previous methods have not considered a case for the ability to use the topological information of BVHs to increase workloads at faster rates, since the maximum number of BVH-node pairs created when two nodes intersect is constrained by the number of children per-node. In order to increase workloads at faster rates, these methods are required to change their BVH construction scheme and thus, traversal logic, in order to incorporate having a larger set of children per-node to speed up traversal rates.

Stackless Traversal. We also note that the presented method is not the first to adopt stack-less traversal since we share a similar design premise to Hapala *et al.* [2013]. In contrast, Hapala et al. have presented an iterative method for ray-tracing on CPUs and GPUs with backtracing and a state-machine to infer which nodes to process next. Barringer and Akenine-Möller [2013] present a similar stack-less approach with full restarts, while Laine [2010] encodes the traversal trial using bit information. In this paper, we instead propose an approach that is entirely GPU based and strictly forward stepping with no notion of backtracing nor state-machines that are used during traversal. We use the topological information of our BVHs, which is encoded as memory locations and offsets, to infer the BVH nodes to traverse next and additionally use this information to increase workloads at faster rates.

Data Parallel Models for Graph Processing. Parallel traversal shares many challenges with large scale graph problems on GPUs, where issues of load-imbalance (irregularity), control-flow divergence, non-coalesced memory access patterns are most common [Lenharth et al. 2016; Merrill et al. 2012]. Harish *et al.* [2007] present one of the earliest solutions to solve breadth first search (BFS), single source shortest path, and all-pairs shortest path, while later works such as Cederman *et al.* [2008] and Tzeng *et al.* [2010] also address issues of load imbalance at the thread-level. Aila *et al.* [2010] investigated the related difficulties of divergence on GPUs in context in ray-tracing.

Recent work focuses on the design of general frameworks for different kinds of large graph structures on GPUs such as the scheduling model for irregular inhomogeneous workloads proposed by Steinberger *et al.* [2014]. Khorasani *et al.* [2014] present a CUDA based model focusing on minimising warp divergence by coarsening parallelism to CUDA warps. Other works have also investigated languages and frameworks for expressing such large-scale computations. Hou *et al.* [Hou et al. 2008] previously present a programming language for expressing the BSP model [Valiant 1990] on GPUs by addressing the challenge of producing efficient stream code and barrier synchronization. The recent Enterprise [Liu et al. 2016] and Gunrock [Wang et al. 2016] frameworks define an iterative BFS traversal of large graphs using the BSP model similar to the influential work of Merrill *et al.* [2012]. In contrast, our inspired work focuses on the specific problem domain of simultaneous BVH traversal for pair-wise CD but also borrows key ideas such as GPU based parallel BFS as a building block. Further, these methods are optimised in large part for massive load imbalance across vertices (as seen in scale-free graphs), but BVHs/BVTTs do not have that kind of imbalance. So different optimisation decisions may be appropriate.

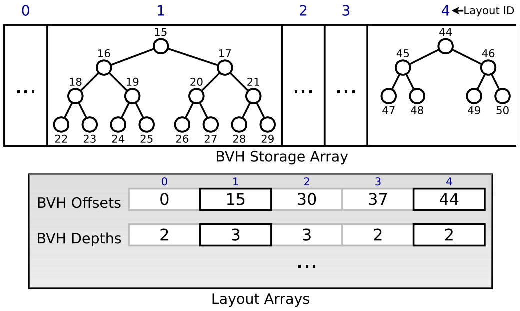
3 METHOD OUTLINE

In what follows, we refer to a pair of BVH-nodes tested for intersection as a *BVTT-node* and additionally refer to each such BVH-node as an *entry-node*: During traversal, a BVTT-node is discarded after a bounding volume (BV) intersection test, such that if the result is true, the BVTT-node is *expanded* by replacing it with a new subset of BVTT-nodes. This new subset is constructed by pairing the descendants of one respective entry-node with those of the other. Alternatively, pairings may be produced between either entry-node and the descendants of the other if the entry-node is a leaf. If the tested entry-nodes do not intersect, no further intersection tests are performed with their descendants. The partial search for geometry that is in close proximity is complete if the BVTT-node is a leaf-pair. In general, this process is recursively repeated until *completion*, i.e. the state of reaching BVTT-nodes where no further intersection tests can be performed.

In practice, we accelerate simultaneous BVH traversal by expanding the BVTT in a bulk-synchronous parallel manner (see subsection 5.1), where the threads evaluate the BVTT at the same level simultaneously. The BVHs and BVTT are stored in global memory. The BVTT is maintained in an array that we call `srcFrontier` in a format that aides the parallel access (see section 4). At every iteration of expanding a BVTT, threads fetch BVTT nodes from `srcFrontier` and test for intersection between respective entry-nodes. If there is an intersection, the descendant nodes are paired and cached as the BVTT nodes for the next iteration in local shared memory (see subsection 5.1). In order to increase the parallelism of this process, we start the expansion at a level deeper from the BVTT root, and pair decendants deeper in the BVH if there is an intersection between paired BVs (see subsection 5.2). Once the local memory cache is full, newly paired descendents are flushed to another global memory array that we call `dstFrontier` in a lock-free manner (see subsection 5.3). Finally, `dstFrontier` and `srcFrontier` are swapped and the iteration for the next BVTT level is repeated until there are no more BVTT nodes in `srcFrontier`.

4 DATA STORAGE AND REPRESENTATION

This section describes the employed BVTT and BVH node representations which enable efficient storage and runtime access for our topologically driven workload expansion scheme described in subsection 5.2.



	0	1	2	3	4
BVH Offsets	0	15	30	37	44
BVH Depths	2	3	3	2	2

Figure 1: BVH nodes are stored compactly in one memory buffer (BVH storage array) with addition set of of small arrays holding metadata about each BVH which we use to infer node descendants at runtime.

BVTT Storage and Representation. A BVTT-node is represented as a simple index-pair where each index is a location of a BVH-node in the global memory. We refer to each index as an *entry*. The BVTT is stored as a large contiguous array in order for threads to access GPU global memory in contiguous and aligned memory blocks [Fauzia et al. 2015] (see Figure 3). The availability of vector load/store instructions on certain GPU architectures allows for efficient bandwidth utilisation which can be beneficial since address accesses of each thread can be combined with single memory transaction issued due to the one-to-one sequential and aligned access to memory [Cook 2013; Luitjens 2013].

BVH Storage and Representation. We propose a novel representation and indexing scheme for BVH nodes that enables instant computation of the BVH that a node belongs to, and the descendants of this node. All BVHs are assumed to be stored compactly in a contiguous array at known offsets with the first at the zeroth offset, whereby the employed hierarchy representation is an implicit binary-tree that is full and complete with nodes stored in a Pre-order Traversal manner as shown in Figure 1. We pad each BVH by rounding the number of leaf-nodes to the nearest power-of-two to enable *implicit indexing* of the descendants of any node. Though padding can potentially result in a higher memory footprint, the additional storage cost is relatively low compared to, for example, the BVTT memory itself, since only bounding volume information is stored per-node (its "payload"). Information referencing geometry that is associated with each leaf-node can be stored separately. Given an arbitrary entry-node i, its j-th descendant that is δ levels ($\delta \geq 0$) deeper than i can be inferred by

$$c_j = \left(2^{\delta} n_i + 2^{\delta} - 1\right) + j, \quad j \in \left(0 \ldots 2^{\delta}\right] \tag{1}$$

where n_i, $0 \leq n_i \leq N-1$ is the position of node i relative to the root node of a BVH with N nodes, and c_j is the relative positions of the descendants with respect to node i. This representation is strictly forward-stepping and infers the descendants of a node using statically known formulae and index information (see subsection 5.2).

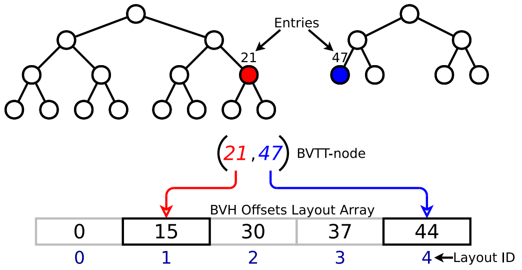


Figure 2: Computing the *layout ID* of an entry-node’s BVH given an entry’s value e.g. 21 or 47: We search for the lowest insertion index of an entry in the layout array of BVH offsets using a lower-bound binary search and subtract one from this insertion index.

Layout Arrays. An additional set of small arrays, termed *layout-arrays*, is also maintained which hold BVH metadata used for computing positional offsets of nodes relative to the root of their BVH, and their descendants at runtime. Layout arrays have the same capacity as the number of BVHs being evaluated, and store low-cost information such as offsets and depths (see Figure 1). Layout arrays can be pre-computed once on the host, during initialization, and then uploaded to the GPU since all information about each BVH may be known at this time.

Inferring BVH Information at Runtime. Since an entry of a BVTT-node does not encode information about the BVH containing its respective entry-node, a unique ID corresponding to each BVH is required to compute the descendants of the entry-node. We refer to this ID of each BVH as the *layout ID*. The layout ID is used to access layout arrays for the information belonging to the BVH containing a given entry-node. We compute the layout ID of a given entry-node by performing a modified lower-bound binary search [Cormen et al. 2009] over the layout array of BVH offsets, using the entry’s value (memory index) as the search target (see Figure 2). The layout ID gotten from this binary search is then used to read layout arrays for information (e.g. the depth) corresponding to the BVH containing the respective entry-node. Note that the overhead of performing this search operation is negligible since it is done in fast GPU local memory with $O(\log_2 N)$ complexity. Further, our method can handle both cases of static and dynamically changing BVHs since we only require that BVHs follow our storage representation. In addition, the implicit representation of BVHs greatly simplifies the construction process which is ideal and would lend itself well to parallel construction methods on GPUs [Lauterbach et al. 2009].

5 ALGORITHM

This section provides the details on how simultaneous BVH traversal is implemented on the GPU using the BSP model. We first describe the general steps to perform GPU traversal in subsection 5.1 and then describe our topologically-driven workload expansion scheme in subsection 5.2. Finally, we describe how intermediate BVTT-nodes are written to global memory at the end of each iteration on the GPU in subsection 5.3.

Algorithm 1: Iterative Bulk-Synchronous Traversal

```
// @arguments
// [input]  srcFrontierDef: 1st iteration
// [input]  srcFrontier
// [output] dstFrontier
HOST traverse (srcFrontierDef, srcFrontier, dstFrontier, ...)
    converged ← False
    src ← srcFrontierDef // stores initial BVTT nodes

    dst ← dstFrontier
    do
        gpu_traversal(src, dst,...)
        if src == srcFrontierDef then
            src ← srcFrontier
        synchronise()
        count ← dstFrontierSzRequest()
        if count == 0 then
            converged ← True
        else
            dstFrontierSzReset()
            swap(src, dst)
    while converged ≠ True
    return

GPU gpu_traversal (srcFrontier, dstFrontier, ...)
    ▷ Phase 1: read
    data ← read(global_id, srcFrontier, ...)
    ▷ Phase 2: traverse
    if intersection(...) then
        expandBVTT(...)
    ▷ Phase 3: write
    write(dstFrontier, ...)
    return
```

5.1 Parallel Traversal

The presented method evaluates the intersection of BVHs by iteratively expanding the BVTT using the breadth-first search (BFS) as a the core parallel primitive for traversal. The steps of algorithm 1 outline the pseudo-code of our method. The host (e.g. CPU thread) will invoke the GPU by calling `gpu_traversal()` in an iterative loop that will terminate when the traversal operation is complete. After invoking the GPU, the host must wait for the current iteration to complete which is represented by a call to `synchronise()`. Once the GPU has finished, the host will then read the new number of BVTT-nodes from the GPU using the function `dstFrontierSzRequest()` which is a GPU-to-Host memory copy command for a single integer value. The value read by the host determines the workload size for the next iteration and will be used to check if the traversal operation has completed.

The contents of `srcFrontier` and `dstFrontier` in algorithm 1 are distinguished to be read- and write-only, respectively, in order to implement *double buffering*, which is used to alias the output of one iteration as input for the next (see Figure 3). Swapping will occur at the end of each iteration on the host with all data remaining on the GPU.

GPU Thread Operations. To start traversal on the GPU, the host will launch approximately as many GPU threads as there are BVTT-nodes in `srcFrontier` (see Figure 3). This will be either the starting amount of default BVTT nodes if it is the first iteration, or resulting amount of the last iteration returned by `dstFrontierSzRequest()`. In phase 1 of algorithm 1, each thread will read a BVTT-node from `srcFrontier` into private register memory and then subsequently read the bounding volume information of each entry-node to perform intersection tests. Phase 2 defines the main body of computation performed by a thread since it is where the intersection test function is applied followed by BVTT expansion. Using the

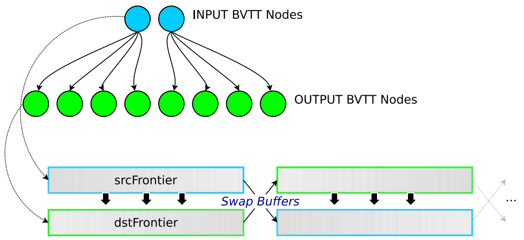


Figure 3: We use two buffers as the main storage for BVTT nodes, the first buffer `srcFrontier` holds the input nodes that are evaluated for intersection while the second buffer `dstFrontier` stores the output nodes from BVTT expansion. The output of one iteration becomes the input of the next as we maintain all traversal data on the GPU.

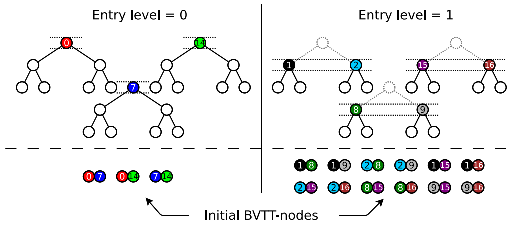


Figure 4: An example of *static workload expansion* excluding self-collisions where we defer the de-facto entry-level (e.g. root level) to descendants at deeper levels in the BVHs to create larger input size for the first iteration(s).

BVTT-node information that is now in private register memory, a thread will then proceed to evaluate it for intersection followed by expansion of the BVTT with new BVTT-nodes if the entry-nodes are found to intersect. Finally, in phase 3, threads collectively copy the new BVTT-nodes to `dstFrontier` for the next iteration.

5.2 Work Expansion

In this section, we describe our topologically driven workload expansion scheme. We introduce the concepts of *static workload expansion* and *adaptive depth-stepping* which are used to overcome GPU under-utilisation resulting from the small workloads of testing higher levels of BVHs, and to control the rate of traversal.

Static Workload Expansion. Evaluating the levels closest to the root nodes can yield small workloads compared to what is expected by GPUs to reach high throughputs. Therefore, in order to increase workloads for the initial iteration(s), evaluation of BVTT-nodes that are constructed from the root nodes is deferred to those constructed from their descendants at lower levels. Figure 4 provides an illustrative example of deferring the *entry-level* of three implicit heirarchies.

Given an entry-level l_e, $0 \le l_e \le d-1$ of a BVH with depth d, we compute its nodes using Equation 1 with $\delta = l_e$ and $n_i = 0$ and account for the actual memory locations of each such node (i.e. c_j) by adding the storage offset of the BVH. Once the entry-level of each BVH is computed, the set of BVTT-nodes evaluated in the first iteration of traversal is then obtained by pairing every node in the entry-level of one BVH to those of another. Deferring the entry-level yields approximately $2^{2l_e}E$ BVTT-nodes, where $E = \frac{N(N-1)}{2} + S$ is the number of collision checks between N BVHs, with S representing the number of self-collision checks. Such an increment can average-out the workloads over multiple iterations while also reducing total number of iterations since entry-level BVTT-nodes can be pre-computed on the host and uploaded once to the GPU as `srcFrontierDef` in algorithm 1.

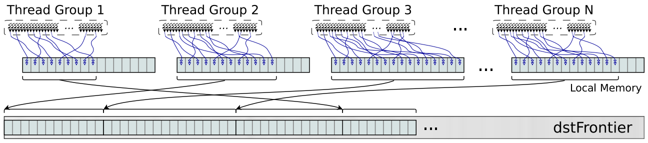


Figure 5: Updating `dstFrontier` using lock-free asynchronous writes that are iteratively buffered to- and copied from local to global memory.

Adaptive Depth-Stepping. Recall that expanding the BVTT is the process of creating new BVTT-nodes from the descendants of every pair of entry-nodes that are found to intersect; We introduce the concept of adaptive depth-stepping to infer the by-level distance to such descendants while accounting for any differences between the depths of tested BVHs. In what follows, the term *depth-step* is used to denote the by-level (jumping) distance that is computed at runtime, from an entry-node to its descendants: This allows us to (1) continue sprouting the descendants in one BVH while reaching the leaves of another, (2) further increase workloads at faster rates while reducing the number of iterations to complete traversal, and (3) tune for performance when writing to global memory.

We compute the depth-step by:

$$\Delta d = min(\mu, \Delta l), \quad 0 \le \Delta d \le d - 1 \tag{2}$$

where $\Delta l = (d-1) - \lfloor log_2(n_i + 1) \rfloor$ is the by-level distance to the leaf-level of the BVH containing an entry-node n_i, and d assumes the depth of the same BVH. The variable μ, $1 \le \mu \le d-1$ is the user-specified parameter of expansion, which is used to control the maximum possible depth-step threads are permitted to use. (Note that Δd is zero if an entry-node is a leaf). Once Δd is known, the descendants of an entry-node are then determined by using Equation 1 with $\delta = \Delta d$, which is then followed by BVTT-expansion.

5.3 Writing Traversal Output

We now describe our lock-free scheme for writing BVTT-nodes to `dstFrontier`, which is designed upon the BSP philosophy for fully utilizing the massive parallelism of modern GPUs.

All new BVTT nodes written to `dstFrontier` will be first accumulated in local shared memory and then flushed in coarse-grained chunks to global memory to prevent individual thread access to global memory as shown in Figure 5. Traversal can potentially induce non-coalesced access to `dstFrontier` as a result of control-flow divergence which may be an additional source latency overhead. Thread-groups are used to achieve this by using an iterative *write-wait-flush* memory update scheme. algorithm 2 outlines the steps of how the threads `T` that performed expansion as part of a group `G` copy their collective subset of BVTT-nodes to

Algorithm 2: Lock-free synchronised write-access

```
   global dst_offset                              // size of dstFrontier
1  local base_offset
2  $i \leftarrow 0$
3  do
4      if $num\_data > 0$ then
5          $c \leftarrow$ atomic_add(C, num_data) - checkpoint
6          if $c < \kappa$ then
7              $r \leftarrow \kappa - c$                      // remaining space
8              $w \leftarrow$ min(num_data, $r$)            // amount written
9              write(Q, $c$, data, $w$)
10             num_data $\leftarrow$ num_data - $w$
11     synchronise_group()
12     $s \leftarrow$ min(C - checkpoint, $\kappa$)           // queue size
13     if $s > 0$ then
14         checkpoint $\leftarrow$ C                     // Q $\equiv \emptyset$
15         if local_id == 0 then
16             base_offset $\leftarrow$ atomic_add(dst_offset, $s$)
17         synchronise_group()
18         async_copy(dstFrontier, base_offset, Q, $s$)
19     else
20         break
21     $i \leftarrow i + 1$
22 while $(i \times \kappa) < M$
```

dstFrontier. During this process, a fixed-size region Q, in local memory, is filled and then flushed iteratively until G has copied all collective BVTT-nodes to dstFrontier. At each iteration, G write to Q with flushing done to asynchronously copy the accumulated BVTT-nodes from local to global memory. (we used the OpenCL async_work_group_copy built-in).

The chosen thread group size and allocated size of Q have a direct effect on the number of iterations taken to copy all BVTT-nodes to global memory, which is also dependent on the maximum possible output. For a given traversal iteration, the total number of iterations I to copy all BVTT-nodes of a group G to dstFrontier is determined by:

$$I = \begin{cases} 1 & : \text{if } \mathtt{M} \leq \kappa \\ \lceil \frac{\mathtt{M}}{\kappa} \rceil & : \text{if } \mathtt{M} > \kappa \end{cases} \tag{3}$$

where κ , $1 \leq \kappa$, is the user-specified capacity of Q and $\mathtt{M} = 2^{2\mu} \times |\mathtt{G}|$ is the maximum possible output size of a group assuming all threads in the group performed expansion, where $|\mathtt{G}|$ is the user-specified thread group size.

Lock-free Synchronisation During Shared Write-Access. Writing to the local memory region Q can have serious impact on performance since it is a shared resource. For this reason, a shared variable C is used, which is a counter allocated per thread-group and is used to atomically compute a writing offset to the shared fixed-size region Q for each thread. At each iteration, group threads T compete for write-access to Q by atomically adding to the counter C (line 5). Each successful thread reserves a region to write its BVTT nodes such that those obtaining a valid offset that is within the bounds of κ will then asynchronously write to Q (lines 6-10). In essence, the threads that have data to write in the current iteration simultaneously contribute toward computing the offset of their collective output relative to a common base address in dstFrontier. This base address is computed by the first thread of the group as a final step before flushing, which is done by using a single *global atomic add* after Q is filled (lines 12-20). Since Q is the sole interface to global memory, lone-thread accesses to global memory is reduced significantly, which can be more expensive to synchronise as workloads increase. We note that this scheme is in fact similar to Garanzha at el. [2011], however they use the first thread in a batch (CUDA warp) to compute the base offset into a global memory region whereas we use the first thread in a group.

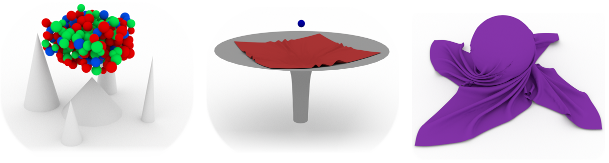

(a) NBody (146K Tris) (b) Funnel (18.5K Tris) (c) Cloth-Ball (92K Tris)

Figure 6: UNC Dynamic Scene Benchmarks used for evaluation purposes[Curtis et al. 2017].

Heuristic for Choosing the Parameter of Expansion. It may at times prove difficult to choose the parameter of expansion μ given the other parameters and hardware constraints that must be considered. The parameter has a direct effect on a number of features in the presented method by effectively providing a fine level of control over the rate of traversal. To facilitate the choice of μ, a simple formula is proposed in order to estimate a maximum value $\overline{\mu}$ subject to size constraints on κ. The purpose is to at-least guarantee a minimum number of threads that will write *all* their BVTT-nodes in a single iteration to Q. Assuming the worst-case, where every group thread writes $\beta = 2^{2\mu}$ BVTT-nodes, $\overline{\mu}$ can be computed by

$$\overline{\mu} = \left\lfloor \log_4 \left(\frac{\kappa}{\alpha} \right) \right\rfloor, \quad 4 \leq \kappa \tag{4}$$

where α, $1 \leq \alpha \leq \left\lfloor \sqrt[4]{\kappa} \right\rfloor$ is a user-specified value for the minimum number of group threads $\in$ T guaranteed to write all their BVTT-nodes in single iteration. Thus, the guaranteed threads will collectively write $\alpha \times \beta$ BVTT-nodes to Q in the current iteration, such that the n^{th} thread to atomically offset C, where $n = \lceil \frac{\kappa+1}{\beta} \rceil$, will write at-most $\kappa \pmod{\beta}$ BVTT-nodes to Q and the rest will be written in the next iteration.

6 RESULTS

We evaluate our method using OpenCL 1.2 on the AMD Radeon R9 280X (3GB VRAM, 32KB Local memory) and Nvidia Geforce GTX 960 (4GB VRAM, 49KB Local memory) GPUs. Three benchmarks (Figure 6) from the UNC Dynamic Scene Benchmarks dataset [Curtis et al. 2017] are used for evaluation purposes: *NBody* (6a) has the largest number of objects at 305 with a total of 146K triangles, it is a rigid-body simulation involving many interacting objects without self collisions. *Funnel* (6b) is a soft+rigid body simulation and is the smallest benchmark made up of four low-resolution meshes (total of 18.5K triangles). In this benchmark, the primary interactions occur between the cloth and funnel. *Cloth-Ball* (6c) is another soft+rigid simulation with two objects that have 92K triangles in total. We use simple axis-aligned bounding boxes (AABB) storing one triangle per leaf-node and our BVHs constructed in bottom-up fashion.

Table 1: Our performance results for simultaneous parallel BVH traversal involving inter- and intra-object collisions.

Benchmark	Query time (ms)	
	Geforce GTX 960	Radeon R9 280
Cloth-Ball	6.43	3.08
Funnel	0.99	0.57
NBody	2.42	1.16

6.1 Performance

Table 1 summarises the performance of our algorithm when we consider both inter- and intra-object tests for the case of Cloth-Ball and Funnel. The presented results are based on the heaviest workloads (colliding leaf-nodes pair) experienced at the most demanding time-step in each benchmark (Cloth-Ball 3.1mil, Funnel 314K, Nbody 117.6K). During experiments, our GPU kernels are executed at-least eight times to reduce potential noise in time measurements because of system *warm-up* overhead. However, no significant differences were observed between test runs.

The presented method is able to perform parallel simultaneous queries in real-time. Execution time is fastest on *Funnel* with query time under 1ms. *Cloth-Ball* takes the longest time (6.43*ms* on the GTX 960). This benchmark has the largest workloads with over 3.1*million* overlapping leaf-node pairs due to the self-collisions induced by the cloth's motion. For this benchmark, our method is able to complete traversal within 6.5*ms*. *NBody* has the lowest number of leaf-node overlaps because is a rigid body simulation. Its BVHs are approximately twice as much slower to evaluate than Funnel due to the larger number of objects (305), and hence, the resulting BVTT. The results reveal our method's strong ability to exploit large scale parallelism on GPUs to quickly evaluate a large number of BVTT nodes for pair-wise CD.

Speedup. We compared the performance against our implementation of the "streams" model by Tang *et al.* [2011]. Comparisons are made using the time-step/frame with the heaviest workloads on each benchmark. We did not include intra-object collisions for Cloth-Ball and Funnel to ensure that workloads fit in our global memory buffers for the streams model. A reduced implementation was used with only pair-wise collision queries to ensure a fair comparison. The implementation also used explicit (not *deferred*) front-tracking with *stream registration* based on segmented locking mechanism (see Tang *et al.* for details). According to Tang *et al.*, deferred front tracking simply trades memory overhead for additional runtime computations.

To emulate the BVTT-node cache (moving front) used by the streams model, we setup the benchmarks as follows: For each benchmark, we extract a pair of keyframes $(k_t, k_{t+\Delta t})$ which are consecutive in time, with each keyframe k representing the geometry of a particular time-step t. Next, we then build the BVH of each mesh in the benchmark for k_t and $k_{t+\Delta t}$. The BVTT-node cache is created by traversing the BVHs of the meshes of k_t until completion and saving the BVTT nodes at which traversal terminates as described by Tang *et al.* [2011]. Our traversal tests and comparisons are performed using BVHs constructed from $k_{t+\Delta t}$ since we can use the BVTT-node cache built from k_t as input for traversal at $t + \Delta t$, thereby allowing the streams model to have a valid cached input set from a "previous" time-step. We have not included the cost of work redistribution for the streams model in our evaluation.

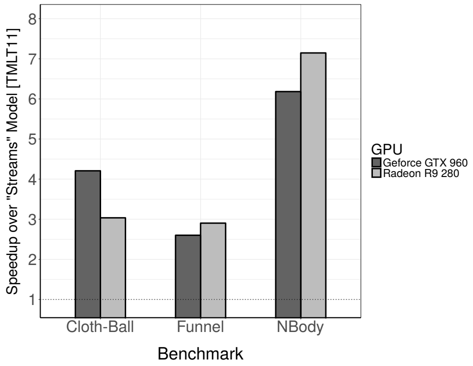


Figure 7: Speedup over the "streams" model

Figure 7 shows speedup were comparisons are based on BVH traversal times to find the set of potentially colliding triangles pairs. Performance of our method on all benchmarks is faster with an average speedup of 4.4× on the R9 280X and 4.3× on the GTX 960. The highest speedup is on NBody at 7.1× for R9 280X (6.2× on the GTX 960) which has the largest workloads in our comparison setup. In general, we found that adapting the streams model on arbitrary GPU architectures is non-trivial due to its dependence on the available amount of local memory for the work-stacks and exploiting L1 caches. Our method is an efficient and a more simpler option for mapping traversal to GPUs

6.2 Parameter Effects and Trade-Offs

The explorable nature of our exposed parameters can make finding correlations between their configurations and the resulting performance unintuitive with no obvious settings. Figure 8 shows the results illustrating the effects of the *entry level* l_e and the *parameter of expansion* μ on execution time for each benchmark (with intra-object collision tests for Cloth-Ball and Funnel). We have found that although increasing μ reduces the number of iterations in our method, care must be taken when making the choice of value. For our evaluated range $(1 - 4)$, making further increments beyond $\mu = 3$ produces a drastic slow-down where the execution-time is on average 3 to 5 times slower than choosing a value between 1 and 3. The observed spikes in the data of Figure 8 , which are observed for $\mu = 3$ and $\mu = 4$, are due to our BVH padding scheme. Some levels in our hierarchies contain more inactive BVH nodes than others, hence the periodic spikes subject to l_e. (Note that in our implementation, we filtered out the any BVTT-nodes containing padded BVH-nodes during entry-level construction). Generally, a choice of smaller values of μ e.g. 2, is a suitable for the case of reducing execution time, even though this choice is at the behest of more iterations to complete traversal. We found l_e to serve our method well for statically reducing the number of iterations to complete traversal while providing the large workloads that we need to

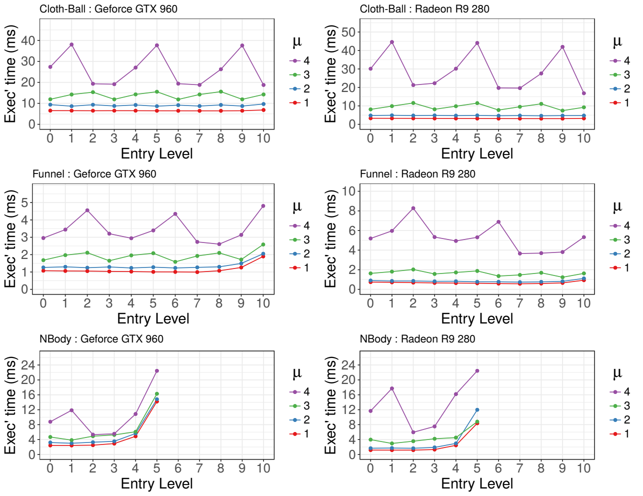


Figure 8: The effects of the entry level l_e and the parameter of expansion μ on the execution time.

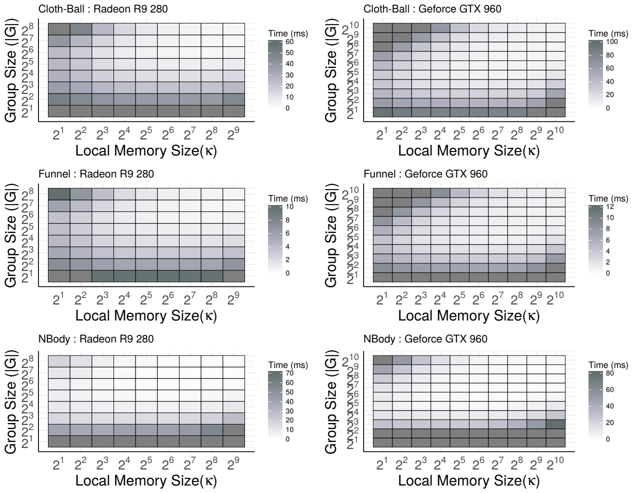


Figure 9: The effects of the allocated local memory size κ and the chosen thread group size $|\mathtt{G}|$ on the execution time.

utilise the GPU. There are some limitations on the exploitation of l_e however, since its chosen value must correlate with the number of BVHs tested to control the resulting input size. On the Nbody simulation, we see a more rapid (exponential) performance drop with l_e compared to the other benchmarks due to the faster rate of increase in the initial input size. For example, we observe that setting $\mu = 2$ and making increments on l_e from 1 to 5 results in a sharp change in execution time from $3ms$ to $16ms$ respectively on the GTX 960.

Local memory and thread-group sizes. The allocated local memory size κ of the fixed-size region Q and thread group size $|\mathtt{G}|$ also have an effect on performance and its scaling properties due to their influence on scheduling. Figure 9 provides our findings regarding the change of execution time relative to κ and $|\mathtt{G}|$, respectively. Setting either parameter to the highest tested value (e.g. $\kappa = 2^9$ and $|\mathtt{G}| = 2^8$ on the R9 280X) while maintaining the other at a minimum (e.g. 2^1) showed slower performance in most cases with the exception of the NBody simulation on the R9 280X. More generally, we observe similar behavioural patterns on both GPUs with the GTX 960 appearing a little more constrained in terms of the optimal choices of κ and $|\mathtt{G}|$. In our results we have found that configurations that use mid-range values are sufficient to obtain good performance relative to the worst case for each benchmark. We observe that our method favours medium-to-large thread groups ($|\mathtt{G}| \geq 2^5$) and allocated local memory size ($\kappa \geq 2^6$) for good performance. The results of Figure 9 are a demonstration of the importance of the trade-offs to be made through our parameters which is crucial for portability.

7 CONCLUSION

We have presented a simple alternative solution for simultaneously traversing a large number of BVHs for CD on GPUs. Our method utilizes the BSP model to overcome the irregular and data-dependent nature traversal. The simplicity of our approach stems from the use of topological properties inherent within an implicit hierarchical representation to harness the parallelism of GPUs. From this, we have presented our topologically-driven workload expansion scheme which provides fine control over the rate of traversal while also increasing workloads for the first iteration(s). In addition, we have described a simple lock-free global memory updating method that can be controlled to adapt algorithm performance based on the available hardware resources. This can likewise be extended with more complex lock-free synchronisation mechanisms using scan primitives such as prefix-sum [Sengupta et al. 2007]. Our method can evaluate complex hierarchies in real-time, and with a speedup of upto 7.1× over the widely used "streams" model.

Limitations and Future Work. The presented algorithm faces a number of limitations which affect performance. Our solution minimizes the compute workload per thread while increasing the DRAM traffic as a side-effect. This is because threads perform just one intersection test, such that in order to perform it, they need to stream data from global memory. Also, our BVH node array is sparsely populated due to padding, which can easily cause excessive L2 and global memory traffic. Such padding can, in the worst-case, also double the storage requirements per BVH subject to the number of leaf nodes. We also note that is it a possibility that our approach of using a one-to-one mapping between threads and BVTT-nodes may not utilise the benefits of GPU caches because there is no opportunity for the reusing BVTT-nodes from `srcFrontier`: The initial read operation of phase 1 (see algorithm 1) is effectively a *cold start* with no opportunity for explicit data reuse since little temporal locality exists when reading BVTT-nodes and BVH node data.

In future work, we plan to extend support for BVH compression by eliminating inactive BVH nodes. This would serve as a solution to the highlighted limitation that the currently employed padding scheme is likely to have limited exploitation of GPU caches leading to excessive global memory traffic alongside the higher memory footprint. Support for moving fronts without complex memory management would also benefit our method well since it is strictly forward stepping with no notion of “node-collapse” to backtrack up heirarchies.

ACKNOWLEDGMENTS

This work was supported by grant for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (http://pervasiveparallelism.inf.ed.ac.uk/) from the UK Engineering and Physical Sciences Research Council (EPSRC). We also thank the reviewers for their useful feedback.

REFERENCES

Timo Aila and Tero Karras. 2010. Architecture Considerations for Tracing Incoherent Rays. In *Proc. of the Conf. on High Performance Graphics (HPG ’10)*. Aire-la-Ville, Switzerland, Switzerland, 113–122.

Rasmus Barringer and Tomas Akenine-MÃűller. 2013. Dynamic stackless binary tree traversal. 2, 1 (2013), 38–49.

Gino van den Bergen. 1997. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools* 2, 4 (1997), 1–13.

Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. In *Proc. of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’02)*. New York, NY, USA, 594–603.

Tyson Brochu, Essex Edwards, and Robert Bridson. 2012. Efficient Geometrically Exact Continuous Collision Detection. *ACM Trans. Graph.* 31, 4, Article 96 (July 2012), 7 pages. https://doi.org/10.1145/2185520.2185592

Daniel Cederman and Philippas Tsigas. 2008. On Dynamic Load Balancing on Graphics Processors. In *Proc. of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH ’08)*. Aire-la-Ville, Switzerland, Switzerland, 57–64.

Shane Cook. 2013. *CUDA Programming: A Developer’s Guide to Parallel Computing with GPUs* (1st ed.). San Francisco, CA, USA.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.).

Sean Curtis, Stephane Redon, and Simon Pabst. 2017. UNC Dynamic Scene Benchmarks. (2017).

Peng Du, Elvis S. Liu, and Toyotaro Suzumura. 2017. Parallel Continuous Collision Detection for High-performance GPU Cluster. In *Proc. of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2017) *(I3D ’17)*. 4:1–4:7.

Peng Du, Jie-Yi Zhao, Wan-Bin Pan, and Yi-Gang Wang. 2015. GPU Accelerated Real-Time Collision Handling in Virtual Disassembly. *Journal of Computer Science and Technology* 30, 3 (2015), 511–518.

Christer Ericson. 2005. *Real-time collision detection.* Amsterdam ; Boston.

Naznin Fauzia, Louis-Noël Pouchet, and P. Sadayappan. 2015. Characterizing and Enhancing Global Memory Data Coalescing on GPUs. In *Proc. of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization.* 12–22.

Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011. Simpler and Faster HLBVH with Work Queues. In *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*, Carsten Dachsbacher, William Mark, and Jacopo Pantaleoni (Eds.). ACM.

Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. 1996. OBBTree: A hierarchical structure for rapid interference detection. In *Proc. of the 23rd annual conf. on Computer graphics and interactive techniques.* 171–180.

Stefan Aric Gottschalk. 2000. *Collision Queries Using Oriented Bounding Boxes.* Ph.D. Dissertation. AAI9993311.

Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. 2013. Efficient Stack-less BVH Traversal for Ray Tracing. In *Proc. of the 27th Spring Conf. on Computer Graphics.* 7–12.

Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proc. of the 14th International Conf. on High Performance Computing (HiPC’07)*. Berlin, Heidelberg, 197–208.

Andreas Hermann, Sebastian Klemm, Zhixing Xue, Arne Roennau, and RÃijdiger Dillmann. 2013. GPU-based Real-Time Collision Detection for Motion Execution in Mobile Manipulation Planning. (11 2013).

Qiming Hou, Kun Zhou, and Baining Guo. 2008. BSGP: Bulk-synchronous GPU Programming. In *ACM SIGGRAPH 2008 Papers (SIGGRAPH ’08)*. ACM, New York, NY, USA, Article 19, 12 pages.

Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proc. of the 23rd International Symposium on High-performance Parallel and Distributed Computing.* 239–252.

James T Klosowski, Martin Held, Joseph SB Mitchell, Henry Sowizral, and Karel Zikan. 1998. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE transactions on Visualization and Computer Graphics* 4, 1 (1998), 21–36.

Dave Knott and Dinesh K. Pai. 2003. CInDeR: Collision and Interference Detection in Real-time Using graphics hardware. In *Proc. of the Graphics Interface 2003 Conference, June 11-13, 2003, Halifax, Nova Scotia, Canada.* CIPS, Canadian Human-Computer Communication Society, 73–80.

Samuli Laine. 2010. Restart Trail for Stackless BVH Traversal. In *Proc. of the Conf. on High Performance Graphics (HPG ’10)*. 107–111.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. 2009. Fast BVH Construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384.

C. Lauterbach, Q. Mo, and D. Manocha. 2010. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Computer Graphics Forum* 29, 2 (2010), 419–428.

Christian Lauterback, Qi Mo, and Dinesh Manocha. 2009. Work distribution methods on GPUs. (2009). https://www.researchgate.net/profile/Dinesh_Manocha/publication/267257965_Work_distribution_methods_on_GPUs/links/54ecdbfa0cf27fbfd771af9c.pdf

Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2016. Parallel Graph Analytics. *Commun. ACM* 59, 5 (April 2016), 78–87.

Tsai-Yen Li and Jin-Shin Chen. 1998. Incremental 3D Collision Detection with Hierarchical Data Structures. In *Proc. of the ACM Symposium on Virtual Reality Software and Technology (VRST ’98)*. New York, NY, USA, 139–144.

Hang Liu, H. Howie Huang, and Yang Hu. 2016. iBFS: Concurrent Breadth-First Search on GPUs. In *Proc. of the 2016 International Conf. on Management of Data.* 403–416.

Justin Luitjens. 2013. CUDA Pro Tip: Increase Performance with Vectorized Memory Access. (2013). https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-increase-performance-with-vectorized-memory-access/

Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. In *Proc. of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2012) *(PPoPP ’12)*. 117–128.

Jia Pan, Christian Lauterbach, and Dinesh Manocha. 2010. g-Planner: Real-time Motion Planning and Global Navigation Using GPUs. In *Proc. of the Twenty-Fourth AAAI Conf. on Artificial Intelligence (AAAI’10)*. 1245–1251.

Jia Pan and Dinesh Manocha. 2011. *GPU-Based Parallel Collision Detection for Real-Time Motion Planning.* Springer Berlin Heidelberg, Berlin, Heidelberg, 211–228.

Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan Primitives for GPU Computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH ’07)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 97–106.

Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (Nov. 2014), 11 pages.

Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. 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.* 63–70.

Min Tang, Dinesh Manocha, and Ruofeng Tong. 2010. MCCD: Multi-Core collision detection between deformable models using front-based decomposition. *Graphical Models* 72, 2 (2010), 7–23.

Min Tang, Ruofeng Tong, Rahul Narain, Chang Meng, and Dinesh Manocha. 2013. A GPU-based Streaming Algorithm for High-Resolution Cloth Simulation. In *Computer Graphics Forum*, Vol. 32. 21–30.

Min Tang, Huamin Wang, Le Tang, Ruofeng Tong, and Dinesh Manocha. 2016. CAMA: Contact-Aware Matrix Assembly with Unified Collision Handling for GPU-based Cloth Simulation. *Computer Graphics Forum (Proceedings of Eurographics 2016)* 35, 2 (2016), 511–521.

M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. 2005. Collision Detection for Deformable Objects. *Computer Graphics Forum* 24, 1 (2005), 61–81.

Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task Management for Irregular-parallel Workloads on the GPU. In *Proc. of the Conf. on High Performance Graphics (HPG ’10)*. Aire-la-Ville, Switzerland, Switzerland, 29–37.

Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. 33, 8 (1990), 103–111.

Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU. In *Proc. of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’16)*. 11:1–11:12.

RenÃľ Weller, Nicole Debowski, and Gabriel Zachmann. 2017. kDet: Parallel Constant Time Collision Detection for Polygonal Objects. *Computer Graphics Forum* 36, 2 (2017), 131–141. https://doi.org/10.1111/cgf.13113

Tsz Ho Wong, Geoff Leach, and Fabio Zambetta. 2014. An Adaptive Octree Grid for GPU-based Collision Detection of Deformable Objects. *Vis. Comput.* 30, 6-8 (June 2014), 729–738. https://doi.org/10.1007/s00371-014-0954-1