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 up to 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

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 \(O(\log N)\) 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 precludes 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:
We present a novel algorithm (section 5) as alternative refor-
mulation 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 com-
plexity 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], dis-
crete 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 meth-
ods 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 compu-
tation 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 advan-
ted for the use of more parallelism through many-core GPUs
and multi-core CPUs [Lauterbach et al. 2010; Tang et al. 2010,
2016]. Since naive 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 meth-
ods, to improve memory access costs and minimise inter-thread
synchronisation. However, work-stacks can lead to work-flow di-
vergence and load-imbalance 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 distinc-
tion 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 num-
ber 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 backtracking 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 backtracking 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 [Lenhart
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 sched-
uling 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 coarsen-
ing parallelism to CUDA warps. Other works have also investigated
languages and frameworks for expressing such large-scale computa-
tions. 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 tra-
versal 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 intersec-
tion 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 sub-
set 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 ex-
 panding 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 mem-
ory. The BVTT is maintained in an array that we call srcFrontier
in a format that aids 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 descendants deeper in the BVH if there is an
intersection between paired BVs (see subsection 5.2). Once the lo-
cal memory cache is full, newly paired descendants 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 repre-
sentations which enable efficient storage and runtime access for
our topologically driven workload expansion scheme described in
subsection 5.2.

**Figure 1:** BVH nodes are stored compactly in one memory
buffer (BVH storage array) with addition set of of small ar-
rays holding metadata about each BVH which we use to in-
fer 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 effi-
cient bandwidth utilisation which can be beneficial since address
accesses of each thread can be combined with single memory trans-
action 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 represen-
tation and indexing scheme for BVH nodes that enables instant
computation of the BVH that a node belongs to, and the descen-
dants 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 im-
plicit 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
(δ ≥ 0) deeper than i can be inferred by

\[ c_j = \left(2^\delta n_i + 2^\delta - 1\right) + j, \quad j \in \{0 \ldots 2^\delta\} \] (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 stati-
cally known formulae and index information (see subsection 5.2).
we only require that BVHs follow our storage representation. In
This section provides the details on how simultaneous BVH traver-
21.47 BVTT-node
BVH Offsets Layout Array
0 15 30 37 44
Layout Arrays. An additional set of small arrays, termed layout-
arrays, is also maintained which hold BVH metadata used for com-
puting positional offsets of nodes relative to the root of their BVH,
and their descendants at runtime. Layout arrays have the same ca-
pacity 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 travers-
sal is implemented on the GPU using the BSP model. We first de-
scribe 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 itera-
tion on the GPU in subsection 5.3.

5.1 Parallel Traversal
The presented method evaluates the intersection of BVHs by iter-
atively expanding the BVTT using the breadth-first search (BFS) as
a the core parallel primitive for traversal. The steps of algo-

Algorithm 1: Iterative Bulk-Synchronous Traversal
// Arguments
GPU gpu_traversal (srcFrontier, dstFrontier, ...)
1 // [input] srcFrontierDef: i^th iteration
2 // [input] srcFrontier
3 // [input] dstFrontier
4 HOST traverse (srcFrontierDef, srcFrontier, dstFrontier,...)  
5 converged ← False   // stores initial BVTT nodes
6 src ← srcFrontierDef // stores initial BVTT nodes
7 dst ← dstFrontier
8 do  
9     (gpu_traversal(src, dst) )   
10     if src == srcFrontierDef then  
11         src ← srcFrontier
12         synchronise()   
13         count ← dstFrontierSzRequest()  
14         if count == (then  
15             converged ← True  
16             else  
17                 dstFrontierSzRequest()  
18             converged ← False
19         done
20     while not converged or True
21 return

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 per-
form intersection tests. Phase 2 defines the main body of computa-
tion performed by a thread since it is where the intersection
test function is applied followed by BVTT expansion. Using the

The output of one iteration becomes the input of the next as the output from one iteration yields approximately $2^{d-1}$ BVTT-nodes, where $E = \frac{N(N-1)}{2}$ 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.

**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:

$$\Delta d = \min(\mu, \Delta l), \quad 0 \leq \Delta d \leq d - 1 \quad (2)$$

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 $\mu$, $1 \leq \mu \leq d - 1$ is the user-specified parameter of expansion, which is used to control the maximum possible depth-step threads are permitted to use. (Note that $\Delta d$ is zero if an entry-node is a leaf). Once $\Delta d$ 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...
Funnel (18.5K Tris)

µ increase. We note that this scheme is in fact similar to Garanzha after step before flushing, which is done by using a single global atomic base address is computed by the first thread of the group as a final dstFrontier. This will then asynchronously write to dstFrontier. During this process, a fixed-size region Q, in local memory, is filled and then flushed iteratively until Q has copied all collective BVTT-nodes to dstFrontier. At each iteration, Q 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:

\[
I = \left\lfloor \frac{C}{\beta \times \kappa} \right\rfloor + 1
\]

where \( \kappa, 1 \leq \kappa \), is the user-specified capacity of Q and \( M = 2^\beta \times |G| \) is the maximum possible output size of a group assuming all threads in the group performed expansion, where |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 \( \kappa \) 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.

**Heuristic for Choosing the Parameter of Expansion.** It may at times prove difficult to choose the parameter of expansion \( \mu \) 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 \( \mu \), a simple formula is proposed in order to estimate a maximum value \( \beta \) subject to size constraints on \( \kappa \). 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 \( \beta = 2^\mu \) BVTT-nodes, \( \beta \) can be computed by

\[
\beta = \left\lfloor \log_2 \left( \frac{\kappa}{\alpha} \right) \right\rfloor, \quad 4 \leq \kappa
\]
Table 1: Our performance results for simultaneous parallelBVH traversal involving inter- and intra-object collisions.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Query time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GeForce GTX 960</td>
</tr>
<tr>
<td>Cloth-Ball</td>
<td>6.43</td>
</tr>
<tr>
<td>Funnel</td>
<td>0.99</td>
</tr>
<tr>
<td>NBody</td>
<td>2.42</td>
</tr>
</tbody>
</table>

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 (6.43ms 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 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 4.4X on the R9 280X and 4.3X on the GTX 960. The highest speedup is on NBody at 7.1X for R9 280X (6.2X 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 \(\mu\) on execution time for each benchmark (with intra-object collision tests for Cloth-Ball and Funnel). We have found that although increasing \(\mu\) 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 \(\mu\) 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...
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 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 \( \kappa \) of the fixed-size region \( Q \) and thread group size \(|G|\) 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 \( \kappa \) and \(|G|\), respectively. Setting either parameter to the highest tested value (e.g. \( \kappa = 2^9 \) and \(|G| = 2^8 \) 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 \( \kappa \) and \(|G|\). 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 (\(|G| \geq 2^5 \)) and allocated local memory size (\( \kappa \geq 2^3 \)) 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 up to 7.1x 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 it is 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 hierarchies.

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


