Scalable and Conservative Continuous Collision Detection for GPU

David Belgrod
New York University

Bolun Wang
Beihang University

Zachary Ferguson
New York University

Marco Attene
IMATI - CNR

Daniele Panozzo
New York University

Teseo Schneider
University of Victoria

Figure 1: We utilize our scalable continuous collision detection (CCD) algorithm within the simulation of elastic material to detect and prevent collisions [Li et al. 2020]. Here we show one such simulation where we twist a tetrahedralized mat (9K tetrahedra) several times. Our method accelerates both the CCD and the distance computations used throughout the simulation. This leads to a $3 \times$ speed-up in the CCD and a $1.1 \times$ speed-up overall.

ABSTRACT

We introduce an algorithm for continuous collision detection (CCD) for linearized trajectories designed to be scalable on modern parallel architectures and provably correct when implemented using floating point arithmetic.

We employ a classical two-phase approach, with a broad-phase CCD to quickly filter out primitives having disjoint bounding boxes, and a narrow-phase CCD that establishes whether the remaining primitive pairs actually collide. Our broad-phase algorithm is particularly simple, yet efficient and scalable, thanks to the experimental observation that sweeping along a coordinate axis performs surprisingly well on modern GPUs. For narrow-phase CCD, we re-design the recently proposed interval-based algorithm of Wang et al. [2021] to work on massively parallel hardware.

To evaluate the correctness, scalability, and overall performance of our approach, we introduce a large scale benchmark for broad- and narrow-phase CCD with exact times of impact, and compare our approach with five state-of-the-art methods. We integrate our algorithm with the IPC contact solver, and evaluate its impact on challenging simulation scenarios.

We release the dataset with analytic ground truth, which required over 5 CPU years to be generated, the implementation of all the algorithms tested, our testing framework, and a reference CPU and GPU implementation of our algorithm as an open source project to foster adoption and development of linear CCD algorithms.

1 INTRODUCTION

Continuous collision detection (CCD) is used extensively in graphics, engineering, and scientific computing for the simulation of rigid and deformable objects, and in geometry processing to ensure that self-intersections are not introduced in parameterization or deformation applications. Objects are typically represented by triangle meshes. In this work we focus on the common case where mesh vertices move along linear trajectories. Hence, collision can occur either when an edge hits another edge, or when a vertex hits a triangle [Wang et al. 2021].

CCD is usually divided into two steps: (1) broad-phase, which is a conservative filter that identifies candidate colliding pairs, and (2) narrow-phase, which validates each pair with an accurate and more computationally intensive algorithm. While the narrow-phase is local and involves only a pair of primitives, the broad-phase usually relies on acceleration data structures to prune unnecessary pairs and avoid the quadratic complexity of a brute-force evaluation on all possible pairs. Both problems have been extensively studied in graphics, engineering, and scientific computing in the last three decades (Section 2).

Approximate CCD. CCD algorithms are often used as a building block of contact solvers, which usually introduce contact forces to remove collision between primitives present in a scene. Since the forces are introduced after a collision happens to remove the intersections, these methods do not require a conservative CCD algorithm: small numerical errors in the CCD queries are unlikely
to affect the overall simulation and it is thus common to sacrifice nu-
merical guarantees in the CCD algorithm, favoring approximations
that lead to a lower computation cost.

Conservative CCD. Conservative CCD is necessary when nu-
merical errors cannot be tolerated, such as in a new trend of contact
models and corresponding simulators [Ferguson et al. 2021; Li et al.
2020, 2021] which guarantee (and also assume), by construction,
that no interpenetraions are present in the scene at any moment
in time. These algorithms provide a robust and accurate model-
ing of contact, but are unable to tolerate CCD imprecisions: if a
CCD query misses a collision, a self-intersection will appear break-
ing the assumption of a self-intersection free state, and thus pre-
vents the simulation from terminating. These imprecisions include
floating-point rounding errors, which need to be accounted for in a
conservative CCD algorithm.

Broad-Phase. The broad-phase of a CCD algorithm usually aims
at detecting collision between (axis-aligned) boxes around primi-
tives. Several existing methods simplify the problem by either
either checking for collisions only at the end of the time interval (i.e., dis-
crete collision detection), or by assuming that only a small fraction
of the scene moves. These simplifications allow for faster algo-
rithms; however, they are not realistic in presence of elastic bodies.
Many acceleration data structures exist (e.g., hash grids, spatial
trees, bounding volumes hierarchies) to avoid checking “far away”
boxes, each providing different advantages in different situations.
However, most of these structures are complex to parallelize in par-
ticular on GPUs where dynamic memory allocation is not an option.
Additionally, these structures’ complexity make it hard to verify
and ensure correctness when using floating-point computations. In
our work, we discovered that the simplest strategy, sweep along
the most varying axis, is the most effective on GPUs. The algorithm
requires only a parallel sort for the sweep, then every GPU core
will compare pairs of boxes. This strategy is not only simple and
massively parallel, but it is also trivial to ensure correctness: the
only computation performed on the boxes is comparison between
floating-point numbers, which is exact.

Narrow-Phase. To the best of our knowledge, the only method
providing a conservative exact narrow-phase CCD check for linear
trajectories is the Tight-Inclusion (TI) CCD of Wang et al. [2021].
Their main idea is to use an interval based root find with floating-
point filters to ensure correctness on the result. Unfortunately, TI
cannot be directly translated to GPU as it contains several branches,
dynamic allocation, and registers. In our work, we redesigned TI to
be GPU friendly.

Two-Pronged Evaluation Approach. Validating correctness of al-
gorithms implemented in floating-point is a major challenge, es-
specially when multiple operating systems and architectures are
considered, since floating-point computations slightly differ due to
either hardware specifics or modern compilers trying to reorder
operations to improve performance. This makes it challenging to
have a correct code and especially hard to keep it up to date, as
compilers and architectures evolve. To provide a way to validate
our implementation on a specific system, we provide a large dataset
of CCD queries evaluated using exact computations (Section 4)
and we provide a large statistical experimental validation of our
method.

Evaluation. We compare our algorithm with five state-of-the-art
methods on five different scenes; our overall CCD algorithm (broad
and narrow-phase) is up to 20 times faster than the best competing
method (Section 5). To show the effectiveness of our acceleration,
we integrate our method into IPC [Li et al. 2020] and run several
challenging examples showing that our method provides a 3.1 to
11.5× speed-up in CCD and 1.1 to 4.5× total speedup.

Contributions. In our work we introduce a novel CCD pipeline
for linearized trajectories. Given two meshes for the start and end
of a step, our method returns the time at which the impact occurs.
Our pipeline includes the novel parallel GPU and CPU broad-phase
algorithm we call Sweep and Tiniest Queue as well as a redesign of
the TI algorithm to account for GPU-specific limitations. Addition-
ally, we introduce a dataset of five scenes obtained from different
simulator containing between 50 thousands to half a million primi-
tives (i.e., vertex-face and edge-edge). For every successive pair of
frames we compute the ground truth Boolean result and time of
impact using a symbolic solver [Wolfram Research Inc. 2020]. We
use this dataset to validate the output of our system.

2 RELATED WORK

We present an overview of the broad and narrow phase collision
detection algorithms that we benchmark in our study, and refer
to [Serpa and Rodrigues 2020] for a more complete one for the
broad-phase and to [Wang et al. 2021] for narrow-phase collision
detection algorithms.

2.1 Datasets

The UNC Dynamic Scene Benchmarks [Curtis et al. 2012] features
keyframes from simulation data and is commonly used through-
out collision detection works as a source of benchmark data. This
dataset covers a variety of simulation methods, materials (e.g., de-
formable and rigid), and physics (e.g., cloth and fracturing solids).
We borrow three scenes (cloth-funnel, cloth-ball, and n-body sim-
ulation) from this dataset to test and benchmark our algorithm.
Additionally, we provide the ground truth Boolean results and sym-

colic time of impacts which was not included in the original dataset.
Serpa and Rodrigues [2020] benchmark several classic broad-
phase collision detection algorithms. In doing so they provide not
only reference implementations of these algorithms but also a
benchmarking framework and procedurally generated scenarios.
These benchmark scenes focus on simple primitives (e.g., cubes
and spheres) in free-fall or undergoing random motion. In contrast
with our work, we focus exclusively on triangle meshes. Serpa and
Rodrigues [2020] also focus entirely on static collision detection
where as our work evaluates broad-phase methods on continuous
collision detection scenarios.

Wang et al. [2021] introduced a large scale dataset for narrow-
phase CCD algorithms. The dataset is designed to cover common
cases extracted from simulation scenarios and challenging degener-
ate cases. Wang et al. [2021] use the dataset to evaluate the accuracy
(the number of false positives), correctness (the number of false
We discuss the 12 methods benchmarked in [Serpa and Rodrigues 2019] comparisons. Unlike Grid, a large memory usage, but, unlike Grid, the number of boxes per voxel is assigned to the cells intersecting it. We detect intersections between boxes by recursively checking its intersection with the root is a box containing the whole space-time scene. Every query box traverses the tree by recursively checking its intersection with the box at the tree’s node until it reaches the leaf. The BVH can be updated “bottom-up” (i.e., if a leaf box grows, it can update its parent until the root), dramatically reducing the update cost in dynamic scenes. We use a deferred BVH (DBVT-D) which performs a single tree-query.

GpuGrid. An OpenCL GPU parallel implementation of the Grid algorithm based on Bullet [Coumans and Bai 2019]. It requires the same delicate choice of \( v \) that is even exacerbated as GPUs have less memory than CPUs.

SH. A parallel CPU implementation of Grid, that encodes the grid implicitly using a spatial hashing function. Each candidate box is rasterized in the grid, and for each voxel, a hash value is computed. These hash values are used to store the elements IDs in a hash map (mapping from voxel indices to a vector of element IDs contained inside the voxel). The candidate collisions can then be found by rasterizing the query element and looking up the voxel indices. Our implementation is based on the code from [Li et al. 2020], which has been modified to produce all collision candidates in one parallel loop and include axis-aligned bounding box checks of elements, to make it comparable with the other approaches. This algorithm has the same shortcomings as Grid: a wrong choice of \( v \) might lead to either slow performances or excessive memory usage. We use a heuristic for the voxel size equal to two times the maximum of the average edge lengths and the average displacement length. This ensures the average element fits within a single voxel.

GpuLBVH. An OpenCL GPU implementation of the Linear BVH in Bullet [Coumans and Bai 2019]. Similar to the BVH, the tree is organized using Morton encoding. By default, Serpa and Rodrigues [2019] assume a maximum of 18n possible intersections, with n input bounding boxes, and discards any successive one. Changing the default size for all our scenes required a trade off between performance for smaller scenes in exchange for more collisions in larger scenes. To avoid this unnecessary shortcoming, which introduces false negatives, we changed the algorithm to process all intersections in an approximate amount of time: If the list of candidates reaches the maximum, we stop storing them and just count their number. Once we know the total number of candidates, we re-execute the algorithm with the correct preallocated size. We note that this change affects only a few cases where the number of candidates is larger than 18n.

KDT. An optimized KD-Tree designed to handle static scenes based on the efficient implementation in [Serpa and Rodrigues 2019]. The spatial subdivision is designed to adaptively partition the space and have a small number of boxes attached to each cell. We note that when using the automatic box-inflation present in the implementation [Serpa and Rodrigues 2019], the algorithm does not report any collision (i.e. it fails to detect true positives). We thus alter the code to disable this feature and use our own default 1%
inflation, which is reasonable, suggesting that there is a bug in the auto inflation code.

Tracy. The parallel method of Tracy et al. [2009] who builds off the incremental SAP of Baraff [1992] and Cohen et al. [1995] by including the ability to insert AABBs without the need to perform a full sort of the axes.

CGAL. The CGAL implementation of an interval-tree SAP algorithm designed to hand d-dimensional axis-aligned boxes (in our experiments, we use d = 3) [Kettner et al. 2016; Zomorodian and Edelsbrunner 2000]. This method works by first performing a SAP on the first axis and then uses range- and interval-trees on the subsequent axes.

2.3 Narrow-Phase

Narrow-phase CCD can be represented as a root finding problem of a function of time. Roots of this function correspond to the point of impact. CCD of linear trajectories without minimal separation equates to finding the roots of a cubic polynomial. Many methods focus on solving these cubic polynomials using numerical methods [Provot 1997]. Provot [1997] introduce the most common strategy of finding a time of coplanarity and then performing an inside check. This idea has since been expanded to solve both rigid [Kim and Rossignac 2003; Redon et al. 2002] and deformable collisions [Hutter and Fuhrmann 2007; Tang et al. 2011].

The down fall of these methods is that they assume infinite precision. When implemented using floating-point numbers, these methods can both miss collisions (false negatives) and report non-existent collisions (false positives).

Alternatively to numerical root-finding algorithms, some propose using inclusion-based root-finding algorithms to determine if a root exists in the co-domain of our function with some tolerance [Redon et al. 2002; Snyder 1992; Snyder et al. 1993; Von Herzen et al. 1990; Wang et al. 2021]. This can either be done using interval arithmetic [Snyder 1992] or by designing custom inclusion functions [Wang et al. 2021]. The latter has the benefit of producing tighter inclusion functions than general interval arithmetic and can be performed in floating-point with specially crafted error bounds. These methods are able to avoid false negatives, but can produce false positives which add extra numerical padding to simulated objects and can result in worse convergence when used in line-search based implicit solvers.

Both Brochu et al. [2012] and Tang et al. [2014] propose exact CCD methods, but despite their claims these methods have been shown to produce both false positives and negatives [Wang et al. 2021]. The only existing method for determining the exact solution is to perform expensive symbolic root finding [Wolfram Research Inc. 2020]. Unfortunately, this is computationally expensive and impractical to use in a simulation pipeline. We use the ground truth generated by this method to verify the results of all queries used in this work.

3 ALGORITHM

Our algorithm, summarized in Algorithm 1, takes as input two triangle meshes M₀, M₁ at time t = 0 and t = 1 moving along linear trajectories and returns the earliest time of impact t* (t* = ∞ if there are no collision between M₀ and M₁). The two triangle meshes are represented as a collection of triangles Tᵢ indices and vertex coordinates Vᵢ represented using floating-point numbers. Our parallel algorithm is implement both on GPU and CPU.

**Overview.** We first build a set B = {bᵢ}, i = 1, ..., k of k boxes \( bᵢ = (bᵢ₀, bᵢ₁) \) \( (bᵢ₀, bᵢ₁) \) are the min and max corner of the box respectively) around every primitive (triangles, edges, and vertices) on M₀ and M₁. On GPU we represent them in single precision while ensuring that every input (in double precision) is exactly contained in its box \( bᵢ \) (Section 3.1). This phase is fast and takes around 10% of the runtime (Figure 2).

Then we pass B to our broad phase algorithm to discard any far away candidate (Section 3.2). The broad phase produces a set of n candidates intersection \( C = \{(i, rᵢ)\} | 1 \leq i \leq n \), where every pair \((i, rᵢ)\) indicates that the primitives in the boxes \( bᵢ \) and \( bᵢ \) potentially intersect. This stage takes about 50% of the runtime (Figure 2).

Finally, to obtain the time of impact \( t* \), we run a parallel version of TI using the collision candidates \( C \) and the input meshes M₀ and M₁ (Section 3.3). The core idea is the same as in [Wang et al. 2021], but we redesigned the algorithm avoiding recursion and using a worker queue paradigm to make it GPU parallelizable. This stage accounts for 30% of the runtime on average (Figure 2). Note that, on GPU, we execute the narrow-phase using the same precision as the input (e.g., double precision for all our experiments).

**Algorithm 1 Overview of our CCD algorithm.**

```text
1: function CCD(M₀, M₁)
2: \( B \leftarrow \text{BUILD_BOXES}(M₀, M₁) \) \n3: \( C \leftarrow \text{BROAD_PHASE}(B) \) \n4: \( t* \leftarrow \text{NARROW_PHASE}(C, M₀, M₁) \) \n5: return t*
6: end function
```

3.1 Construction of the Boxes

To construct a tight single precision box \( b = (bₓ, bᵧ, bᵣ) \) around a primitive (i.e., a triangle, edge, or vertex) we first compute the extend of the box \( bₓ, bᵧ, bᵣ \) in double precision using the min and max of the coordinates of the primitive. For instance, for a triangle, \( bₓ \) is the minimum of the \( x \), \( y \), \( z \)-coordinates of the three vertices. To conservatively convert \( bₓ \) in single precision, for each coordinate (e.g. \( x \)) we first round \( bₓ \) to its nearest single precision value and check if \( bₓ ≤ bₓ \). In case it is not, we decrease \( bₓ \) to its previous representable single precision value using the function \( \text{nextafter}(bₓ) \). The procedure for \( bᵧ \) is similar. When running our algorithm on GPU, we need to copy the boxes on the device; in our experiments this time, B2G, is negligible (Figure 2).

3.2 Broad-Phase

Our Sweep and Tiniest Queue (STQ) algorithm is based on the (surprising) observation that brute-force checking all possible pairs is...
not only easy to parallelize, but very fast on modern architectures due to their memory layout and large number of ALUs, which favors heavier computation with a structured memory access. Unfortunately, this simple approach has a runtime quadratic with respect to the number of pairs and cannot be applied to large scenes. To overcome this limitation we borrow ideas from the Sweep and Prune algorithm [Baraff 1992; Cohen et al. 1995] to limit the average complexity of our algorithm (Figure 3).

Algorithm. Our STQ starts by computing the variance $\sigma$ of the boxes’ centers $B_C$ and finding the most varying axis $a$ (line 4) [Liu et al. 2010]. We then sort the boxes $B$ along the axis $a$ and initialize a queue $Q_1$ that will hold pairs of boxes that overlap on the $a$-axis. Since $a$ is the axis of maximum variance this will lead to the smallest possible queue among the three axes if the data is uniformly distributed.

In the first step, for every box $b_i$ we check if it intersects its next box $b_{i+1}$ along the $a$-axis; if it does we append the pair $(b_i, b_{i+1})$ to $Q_1$ (line 8).

Then STQ extracts a pair $(b_i, b_j)$ from $Q_1$ and check if it intersects in the remaining two axes $a'$ (line 17). If they do, we append the pair to the output $C$ (global). Finally we add the pair $(b_i, b_{j+1})$ to an output queue $Q_o$ if $(b_i, b_{j+1})$ intersects along the $a$-axis (line 21, Figure 4).

Implementation Remarks. We remark that the STQ algorithm creates queues with at most $k - 1$ elements, as each pass on $Q_i$ pushes at most one pair. Additionally, as not all pairs are always added, the sizes are monotonically decreasing. On GPU, we exploit this observation to pre-allocate the correct queue-size and guarantee that we will always have the necessary space. To efficiently parallelize STQ on GPU we exploit shared memory per thread block.

In our experiments, the most efficient strategy consists of splitting the sorted boxes $B'$ into $m$ blocks containing 32 queries each (i.e., $m = k/32$). Since a warp consists of 32 threads, choosing a smaller number will introduce unnecessary branching. Using larger thread blocks can lead to more imbalanced workloads and longer runtime of branching in the code. The GPU thread block-scheduler performs well even with a larger grid size of thread blocks.

When running our algorithm on GPU, we need to prepare the data for the narrow-phase; this requires:

1. splitting the set $C$ into edge-edge and vertex-face cases (SO),
2. transforming the pairs in $C$ into narrow-phase data (CD),
3. coping the vertex coordinates to the device (V2G).

Similar to copying the boxes to GPU these intermediate stages are negligible (Figure 2).

3.3 Narrow-Phase

Since our algorithm builds upon [Wang et al. 2021], we first provide a self-contained overview of the original algorithm.

Summary of [Wang et al. 2021]. The algorithm uses interval arithmetic to detect collisions. It starts by constructing the interval $I = I_t \times I_{uv} \subseteq [0, 1]^3$, where $I_t = [0, 1]$ is the time interval and $I_{uv}$ is the parameterization of the space. For a point-face query
We can thus parallelize over interval splits instead than on queries. This observation leads to performing the same operations on every thread independently from the candidate\(^1\).

We start by constructing, for every collision candidate \(C\), the initial interval \(I = [0, 1]^3\) (line 2) and append them to the input queue \(Q_\epsilon\). We then process in parallel all intervals in \(Q_\epsilon\) and produce and output intervals’ queue \(Q_o\) (line 6). At the end, we swap the roles of the two queues (line 15) and continue alternating until \(Q_o\) is empty.

\textbf{Time of Impact.} We modified how we process a single interval (line 20) with respect to [Wang et al. 2021] to account for the time of impact. We first check if \(I^I\) (i.e., left hand-side of the time interval of \(I\)) is larger than the current time of impact \(t^*\). In this case \(I\) can safely be skipped (line 22). Then we proceed as in [Wang et al. 2021] and, if the box \(B\) constructed from \(I\) does not intersect \(C_e\), \(I\) can be discarded as it does not contain a root (line 26). Finally, if the width \(w(I)\) of \(I\) is smaller than a user provided tolerance \(\delta\) (or if \(B\) is contained in \(C_e\), we report collision and return \(I^F\) (29). If it is not the case, we split \(I\) into a left \(I^l\) and right \(I^r\) interval and return the current time-of-impact as optimal.

\textbf{Discussion.} The algorithm has only two necessary synchronizations: 1) the update of \(t^*\) and 2) appending intervals to \(Q_o\). We update \(t^*\) using a mutex as \texttt{atomicMin} does not support floating-point numbers. To efficiently append intervals we keep track of the size of \(Q_o\) and use \texttt{atomicAdd} to increase the size counter when appending new elements.

## 4 DATASET GENERATION

Our dataset is composed of five simulated scenes (Figure 5 top row) and the corresponding ground truth data for continuous collisions between frames. From the UNC Dynamic Scene Benchmark [Curtis et al. 2012], we include two co-dimensional cloth simulations with a large number of self collisions (Cloth-Ball and Cloth-Funnel) and a simulation of a large number of rigid and deformable spherical bodies (N-Body). We also include two elasto-dynamic scenes featuring large compression and nonlinear buckling simulated using the method of Li et al. [2020] (Armadillo-Rollers and Rod-Twist).

\textbf{Ground Truth.} Differently from all other datasets for CCD, we generate ground truth for each successive pair of frames from each scene, using a combination of symbolic computation and conservative filtering. This is done by first enumerating all possible collision pairs (collision candidates) through a brute-force approach. We only consider point-triangle and edge-edge pairs as these pairs capture the first collisions between triangles [Provoz 1997]. Furthermore, we ignore points that are vertices of the triangles and edges that share a common endpoint as these are trivially colliding. For each collision candidate, we determine if the pair collides using the provably conservative CCD of Wang et al. [2021]. While this CCD algorithm is guaranteed to not have false negatives, it may produce false positives. To eliminate false positives, we find exact solutions of the CCD query using the symbolic solver in Mathematica [Wolfram Research Inc. 2020]. While we could skip the middle step and reduce the number of registers used, which is a common performance bottleneck on GPUs, where each streaming multiprocessor (SM) has a very small pool of registers available.

\(^1\)An additional subtle benefit of this algorithm is that it makes the GPU kernel shorter,
directly use the symbolic CCD this would be prohibitively slow, as the symbolic solvers take several seconds per query. Instead, the method of Wang et al. [2021] quickly filters the majority of the collision pairs, leaving a smaller number of candidates to validate with the symbolic solver.

Time of Impact Expressions. In addition to the Boolean ground truth we also include the symbolic expressions for the valid roots (as Wolfram Language MX files). It is necessary to save the symbolic expressions (instead of a real or rational number) as they may include operators and functions that cannot be evaluated exactly (e.g., square roots). These expressions are later used to validate our results with the ground truth data. We report the average running time for correct implementations (the additional material reports the timings of all methods). We note that all methods detect similar number of candidates. The LBVH method provides a function to update the data structure with new positions instead of rebuilding it from scratch at every frame, which we use in our experiments. For all other methods, the acceleration data structure is rebuilt at each frame. The results are summarized in Figure 5. For the narrow-phase we compare only with TI [Wang et al. 2021] as it is the only correct algorithm. For each narrow-phase algorithm, we compare the list of candidates times more.

5.1 Comparison
For each broad-phase algorithm, we compare the list of candidates with the ground truth data. We report the average running time for correct implementations (the additional material reports the timings of all methods). We note that all methods detect similar number of candidates. The LBVH method provides a function to update the data structure with new positions instead of rebuilding it from scratch at every frame, which we use in our experiments. For all other methods, the acceleration data structure is rebuilt at each frame. The results are summarized in Figure 5. For the narrow-phase we compare only with TI [Wang et al. 2021] as it is the only correct algorithm. For each narrow-phase algorithm, we only record the time of impact for the whole time-step.

The performance of the broad phase methods are mostly independent from the scene: ours, LBVH, and SH are among the fastest

---

Figure 5: Log (top) and normal (bottom) bar charts of the timings for every scene. The green colors shows the performances for the consumer architectures (CPU1 and GPU1), while orange is for the professional ones (CPU2 and GPU2).

5 RESULTS
We run all CPU methods on two CPUs: CPU1 a consumer architecture (Intel®Core(TM) i7-5930K CPU @ 3.5 GHz) and CPU2 a professional CPU (AMD Ryzen Threadripper PRO 3995WX 64-Cores @ 2.7 GHz) and two GPU: GPU1 a gaming card (NVidia 3080 Ti) and GPU2 a professional card (NVidia v100). For every CPU run we limit the number of threads to 12. We note that for many computation GPU2 behaves like a GPU but costs around ten times more.

4https://opendata.blender.org/
Algorithm 3 Overview of the narrow-phase.

```
function NARROW_PHASE(C, M₀, M₁)
    Q₀ := BUILD_INTERVALS(C, M₀, M₁)
    t* := 1
    while Q₀ ≠ ∅ do
        for all l ∈ Q₀ do
            if l ≠ 0 then
                Q₀ := Q₀ ∪ l
            end if
        end for
        Q₀ := { }  // In parallel
        for all l ∈ Q₀ do
            t*, l′, l″ := PROCESS_INTERVAL(I, t*)
            if l′ ≠ 0 then
                Q₀ := Q₀ ∪ l′
            end if
            if l″ ≠ 0 then
                Q₀ := Q₀ ∪ l″
            end if
        end if
        Q₁ := Q₀
    end while
    return t*
end function
```

5.2 Scaling

To assess the scalability of our method we run the last ten frames of Rod-Twist on CPU2 varying the number of threads from 1 to 32 (Figure 6). Our algorithm scales well with respect to the number of threads: with 8 threads the BP is 7.5 times faster and it gets to 22 times faster with 32 threads. Constructing the boxes scales worst: is it just 3.7 times faster with 8 threads and 6.5 for 32.

5.3 Time of Impact Validation and Accuracy

Using the time of impact expressions discussed in Section 4, we both validate our computed time of impacts occur before the ground truth and compute an error for the roots. The results of this study are presented in Figure 7. The average error for over all queries is ~0.0023 with a standard deviation of ~0.018 and a median error of ~1.03 × 10⁻⁶. We note that this distribution varies between scenes (e.g., cloth-ball has an average error of ~8.33 × 10⁻⁶ compared to...
Table 1: Performance of our new CCD and broad phase for the unit tests of Erleben [2018] in [Li et al. 2020, Figure 11], the five-cube stack, mat-twist, and mat-knives simulations. For tiny scenes the overhead of our method worsens performance, but in general leads to a performance increase in both the CCD and simulation as a whole. For larger scenes, the bottleneck shifts to the linear solves and Hessian matrix assembly leading to a smaller overall improvement of running time.

Figure 8: Several frames of the five-cube stack example. The whole scene has only 30 tetrahedra and is therefore CCD bound.

Figure 9: Several frames of a codimensional simulation. The mat has 9K tetrahedra, while the codimensional triangles are not deformable.

We run the five-cube stack example (Figure 8) that contains several resting contacts. Similarly to the unit tests, as the meshes are extremely coarse, when using our method the simulation is 4.5× faster. When using denser meshes (Figure 1 has 9K tetrahedra) and the elastic deformations become more challenging, the non-linear elastic solver dominates the IPC runtime and the speedup provided when using our method becomes less prominent; only 8% times faster overall. Our method naturally support CCD between codimensional object (Figure 9); for this scene we again see similar speedups.

6.1 Avoiding a Time of Impacts Equal to Zero
An important caveat of using a conservative CCD method inside the IPC algorithm is that is should not produce a time of impact \( t^* \) of zero. A \( t^* \) of zero would stop the non-linear solver because IPC uses the \( t^* \) to determine the maximum step-size allowable inside the implicit time-stepping optimization. Additionally, IPC guarantees every step results in an intersection-free state, so \( t^* \) cannot be zero (i.e., not initially intersecting).

To avoid a zero \( t^* \), we make slight modifications to Algorithm 1 and 3 (Appendix A). As part of the strategy to avoid \( t^* = 0 \), the IPC algorithm uses a minimum separation in the CCD to prevent taking a step that results in parts exactly touching. We choose the minimum separation relative to the initial distance between the query’s primitives (0.8× in all our experiments). To implement minimum separation CCD, we use the same strategy as TI [Wang et al. 2021]: we enlarge the box \( C_e \) by the minimum separation distance.

7 CONCLUSION
We introduce a novel, scalable CCD algorithm combining broad and narrow phase collision detection. Our algorithm is provably conservative and our implementation has been tested on multiple combinations of recent operating systems and hardware architectures. Our algorithmic contribution specifically targets parallel architectures with high memory bandwidth (and high latency), which have very different requirements than traditional serial architectures. Our algorithm scales well to GPU hardware: a Nvidia 3080 Ti GPU (MSRP ~1.2K USD) achieves a speed comparable to a CPU server chip with 64 cores/128 threads (MSRP ~10K USD). We believe that our GPU algorithm could be extended to run on multi-GPU. Our preliminaries experiments show that the broad phase becomes 2.3 times faster when using 4 GPUs for the N-Bodies scene.

While the correctness of our algorithm is provable assuming infinite precision [Baraff 1992; Cohen et al. 1995], certifying that our
implementation is correct, with computation performed with floating point numbers, is very challenging. The correctness will heavily depend on the hardware used and the compiler, which evolve over time. Our dataset provides a way to check for correctness of CCD codes on different settings, and while passing the benchmark is not a formal proof of correctness, we believe it is a realistic and practical approach to evaluate the conservativeness of our implementation. It helped us design our algorithm, and using it we found counter-examples for other CCD codes (Additional material). The benchmark is easy to extend, and we plan to keep it up to date and add to it more scenes and challenging queries in the next years.

When integrated with the state of the art solver IPC, our approach reduces the overall simulation time, which we believe is of practical relevance to the graphics and simulation community. Our implementation will be released on GitHub with the MIT license, to foster its adoption in academia and in industry.

REFERENCES

David Baraff. 1992. Dynamic simulation of non-penetrating rigid bodies. Technical Report. Cornell University.

Tyrone Brochu, Essex Edwards, and Robert Bridson. 2012. Efficient geometrically exact continuous collision detection. ACM Transactions on Graphics 31, 4, Article 96 (July 2012), 7 pages.

Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. 1995. COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. In Proceedings of the 1995 Symposium on Interactive 3D Graphics (Monterey, California, USA) (I3D ’95). Association for Computing Machinery, New York, NY, USA, 189–194.

Erwin Coumans and Yunfei Bai. 2016–2019. PyBullet, A Python module for physics simulation for games, robotics and machine learning. http://pybullet.org.

Sean Curtis, Russ Gayle, Naga Govindaraju, Ikhun Kabul, Ming Lin, Simon Pabst, Stephane Redon, Aneesud Sud, Min Tang, Sung-eui Yoon, Jueyi Zhao, and Dinesh Manocha. 2012. UNC Dynamic Scene Benchmarks. http://gamma.cs.unc.edu/DYANMBC.

Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE 5, 1 (1998), 46–55.

Kenny Erleben. 2018. Methodology for Assessing Mesh-Based Contact Point Methods. ACM Transactions on Graphics 37, 3, Article 39 (July 2018), 30 pages.

Zachary Ferguson, Minchen Li, Teseo Schneider, Francesca Gil-Ureta, Timothy Langlois, Chenfanfu Jiang, Denis Zorin, Danny M. Kaufman, and Daniele Panozzo. 2021. Intersection-free Rigid Body Dynamics. ACM Transactions on Graphics (Proceedings of SIGGRAPH) 40, 4, Article 183 (2021).

Marco Hutter and Arminf Sulmann. 2007. Optimized continuous collision detection for deformable triangle meshes. (July 2007).

Lutz Kettner, Andreas Meyer, and Afra Zomorodian. 2016. Intersecting Sequences of dD iso-oriented Boxes. https://doc.cgal.org/latest/Box_intersection_d/index.html.

Byunghak Kim and Jared Rossignac. 2003. Collision prediction for polyhedra under screw motions. Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications, 4–10.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. 2020. Incremental Potential Contact: Intersection-and Inversion-Free, Large-Deformation Dynamics. ACM Transactions on Graphics (Proceedings of SIGGRAPH) 39, 4, Article 49 (July 2020), 20 pages.

Minchen Li, Danny M. Kaufman, and Chenfanfu Jiang. 2021. Codimensional Incremental Potential Contact. ACM Transactions on Graphics (Proceedings of SIGGRAPH) 40, 4, Article 170 (2021).

Fuchang Liu, Takahiro Harada, Youngjun Lee, and Young J Kim. 2010. Real-time Collision Culling of a Million Bodies on Graphics Processing Units. ACM Transactions on Graphics 29, 6 (Dec. 2010), 1–8.

Xavier Provot. 1997. Collisions and self-collision handling in cloth model dedicated to design garments. In Computer Animation and Simulation. Springer, 177–189.

Stephanie Redon, Abderrahmane Kheddar, and Sabine Coquillart. 2002. Fast Continuous Collision Detection between Rigid Bodies. Computer Graphics Forum 21 (May 2002).

Teseo Schneider, Jérémie Dumais, XiFeng Gao, Denis Zorin, and Daniele Panozzo. 2019. Polyfem. https://polyfem.github.io.

Ygor Rebouças Serpa and Maria Andriéa Formico Rodrigues. 2019. Broadmark: A Testing Framework for Broad-Phase Collision Detection Algorithms. Computer Graphics Forum 38, 1 (2020), 436–449.

John M. Snyder. 1992. Interval Analysis for Computer Graphics. Computer Graphics (Proceedings of SIGGRAPH) 26, 2 (July 1992), 121–130.

John M. Snyder, Adam R. Woodbury, Kurt Fleischer, Bena Currin, and Alan H. Barr. 1995. Interval Methods for Multi-Point Collisions between Time-Dependent Curved Surfaces. In Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (Anaheim, CA) (SIGGRAPH ’95). Association for Computing Machinery, New York, NY, USA, 321–334.

Min Tang, Dinesh Manocha, Sung-eui Yoon, Peng du, Jia-Pil Heo, and Ruofeng Tong. 2011. VoCCD: Fast continuous collision culling between deforming volume meshes. ACM Transactions on Graphics 30 (Jan. 2011), 111.

Min Tang, Ruofeng Tong, Zhendong Wang, and Dinesh Manocha. 2014. Fast and Exact Continuous Collision Detection with Bernstein Sign Classification. ACM Transactions on Graphics 33 (Nov. 2014), 186:1–186:8. Issue 6.

Daniel J Tracy, Samuel R Buss, and Bryan M Woods. 2009. Efficient large-scale sweep and prune methods with AABB insertion and removal. In 2009 IEEE Virtual Reality Conference. IEEE, 191–198.

Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. 1990. Geometric Collisions for Time-Dependent Parametric Surfaces. In Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (Dallas, TX, USA) (SIGGRAPH ’90). Association for Computing Machinery, New York, NY, USA, 39–48.

Bolun Wang, Zachary Ferguson, Teseo Schneider, Xin Jiang, Marco Attene, and Daniele Panozzo. 2021. A Large Scale Benchmark and an Inclusion-Based Algorithm for Continuous Collision Detection. ACM Transactions on Graphics (2021).

Wolfram Research Inc. 2020. Mathematica 12.0. http://www.wolfram.com.

Afra Zomorodian and Herbert Edelsbrunner. 2000. Fast Software for Box Intersections. In Proceedings of the Sixteenth Annual Symposium on Computational Geometry (Clear Water Bay, Kowloon, Hong Kong) (SCG ’00). Association for Computing Machinery, New York, NY, USA, 129–138.

A ZERO TIME OF IMPACT AND MINIMUM SEPARATION

To avoid $t^* = 0$, we make slight modifications to Algorithm 1 and 3. In IPC, if Algorithm 1 return $t^* = 0$ we perform the narrow-phase again but set the minimum separation to 0 and enable a no zero Tol strategy.

This no zero Tol strategy dictates that if $t^2 = 0$, I should always be split (ignoring user tolerances and maximum number of splits). We note that under floating-point division this split might not be possible, but this has not been encountered in practice and would most likely require a degenerate case involving tiny distances (which IPC does a good job of preventing thanks to its barrier potential method of handling contacts). In the end, because minimum separation was disabled, the resulting $t^*$ is multiplied by a scaling factor less than 1 to avoid exactly touching after the step (we use 0.8 in our examples).