Faster parallel collision detection at high resolution for CNC milling applications

Xin Chen
Georgia Institute of Technology
xchen384@gatech.edu

Dmytro Konobrytskyi
Uber Advanced Technologies Group
dkonobr.cv@gmail.com

Thomas M. Tucker
Tucker Innovations Inc.
tommy@tuckerinnovations.com

Thomas R. Kurfess
Georgia Institute of Technology
kurfess@gatech.edu

Richard W. Vuduc
Georgia Institute of Technology
richie@cc.gatech.edu

ABSTRACT

This paper presents a new and more work-efficient parallel method to speed up a class of three-dimensional collision detection (CD) problems, which arise, for instance, in computer numerical control (CNC) milling. Given two objects, one enclosed by a bounding volume and the other represented by a voxel model, we wish to determine all possible orientations of the bounded object around a given point that do not cause collisions. Underlying most CD methods are 3 types of geometrical operations that are bottlenecks: decompositions, rotations, and projections. Our proposed approach, which we call the aggressive inaccessible cone angle (AICA) method, simplifies these operations and, empirically, can prune as much as 99% of the intersection tests that would otherwise be required and improve load balance. We validate our techniques by implementing a parallel version of AICA in SculptPrint, a state-of-the-art computer-aided manufacturing (CAM) application used CNC milling, for GPU platforms. Experimental results using 4 CAM benchmarks show that AICA can be over 23× faster than a baseline method that does not prune projections, and can check collisions for 4096 angle orientations in an object represented by 27 million voxels in less than 18 milliseconds on a GPU.

KEYWORDS

collision detection, massively parallel collision detection, GPU

1 INTRODUCTION

We consider a collision detection (CD) problem that arises in the area of *computer numerical control (CNC) milling* [15], an application in advanced manufacturing. An example appears in Figure 1. There is a shape one wishes to cut starting from a block of material, such as the head from an initial cube of plastic (the left of Figure 1). The computational task is to construct a path that a cutting tool can make that eventually ends with the target object (e.g., the head), starting from the input (e.g., the block of plastic).

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337838

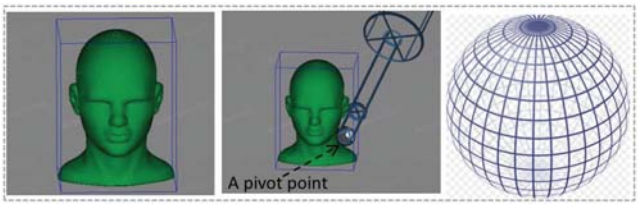


Figure 1: Inputs to the collision detection (CD) problem: a head object from CAD benchmark, a tool composed of bounding cylinders and a pivot point at the end of the tool. The orientation of the tool at the pivot is represented by a pair of angles in polar coordinates (φ, γ), where $\varphi \in (0, \pi)$ and $\gamma \in (0, 2\pi)$.

Figure 2: The output of one CD test is an accessibility map (AM). A map at (m, n)-resolution is discretized uniformly into $m \cdot n$ points, with each point denoting some (φ, γ) orientation. Here, the map is shown as a grid whose vertical axis corresponds to φ and whose horizontal axis corresponds to γ. Schematically, a black point indicates a collision between the voxel and the tool when oriented at (φ, γ), and a white point means no collision.

There are several possible CD methods, which are widely used in other settings, like CAD/CAM [1, 3, 16, 29]; computer animation, games, and physical simulations [14]; motion planning [23, 24]; and virtual assembly [5]. To improve the speed of CD, prior approaches have combined computer graphics analysis techniques with efficient parallelization. Such techniques include culling to prune redundant computation [26, 28], as well as algorithms that can exploit GPU features like visibility queries in the depth buffer, and frame buffers and fragment shaders [7, 14, 24]. But there are also efficient parallel CD methods for both general-purpose CPUs and GPUs [4, 18, 23, 33].

Underlying most of these approaches are three types of fundamental, computationally intensive operations: decompositions, rotations, and projections. We illustrate them in the bottom of Figure 4, in the case where one wishes to check whether a cylinder (i.e., one model of a tool) intersects with a box (i.e., part of the object being milled). Briefly, these operations are a sequence of geometric calculations that transform the input object into other

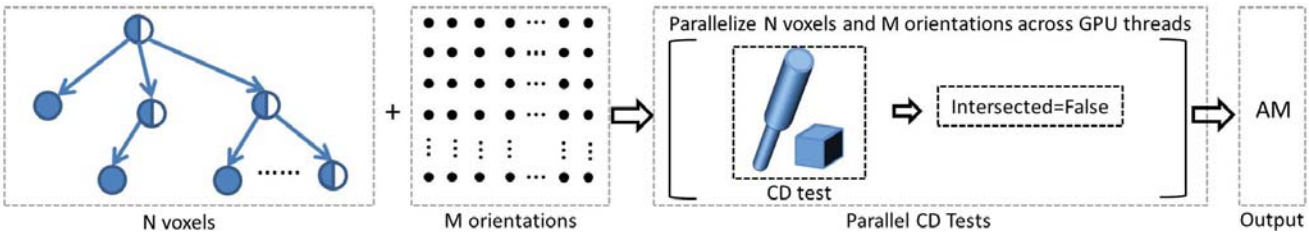


Figure 3: Schematic of the baseline parallel scheme to calculate the AM. It begins with the target (left), stored as a adaptive (non-uniform) volumetric octree with N voxels, and the $M = m \cdot n$ discrete orientations of the tool to check for collisions. Each (GPU) thread considers one orientation and executes Algorithm 2, which traverses the octree to see if that orientation causes an intersection with any voxels. (Algorithm 2 does not need to visit all voxels if it detects early in the traversal that no intersection is possible.)

representations, as explained in Section 2. These three operations also appear commonly in other types of basic geometric intersection tests, such as sphere-box intersection, box-box intersection, and cylinder-sphere intersection. Such tests are the basis of discrete collision detection (DCD) and continuous collision detection (CCD) algorithms in computer graphics [3, 4, 10, 17, 18, 33].

However, for CD, cylinder-box intersection tests dominate and may be sped up considerably. We do so using a novel abstraction, called the *inaccessible cone angle (ICA)*, that eliminates a high percentage of the usual cost of CD tests for decompositions, rotations, and projections. We further accelerate this method via a parallel algorithm that we call the *aggressive inaccessible cone angle (AICA)* method. Prior methods to test for the intersection between two objects relied on a general *bounding volume hierarchy (BVH)* and fine-grained volumetric representation of the cutting tool. However, in our application we can replace this representation by a simpler collection of bounding volumes, like the bounding cylinders suggested in Figure 1, thereby making the three fundamental operations cheaper. This simplification also suggests a new way to express the computation, yielding a method that has smaller "constant factors" and is easier to load-balance. Our ICA abstraction could be extended into a new primitive and, thereby, applied in other CD contexts that involve rotational operations (Section 6).

In brief, the main claimed contributions of this paper are the ICA abstraction, the AICA parallel algorithm, and an empirical validation thereof. [1] The basic ICA abstraction allows us to reason about the object over all orientations in a computationally compact way, thereby reducing the number of operations and checks than prior art. When using an adaptive volumetric octree to store the target object (e.g., the head of Figure 1), we observe that as many as 99% of the CD tests can be eliminated on a variety of complex input geometries. We have prototyped our approach in a version of the commercial SculptPrint software package. Our AICA method can be over 23× faster than a baseline approach that uses 3D projection, and nearly 4.8× faster than another novel method we present. In absolute performance, AICA enables the checking of 4096 orientations for an object represented by 27 million voxels in just 18 milliseconds on a recent GPU.

[1] An initial sketch of the ICA appeared earlier in an unrefereed work [15]. However, the previously proposed calculation of ICA is incorrect. In this paper, we first give a correct description and then present a parallel scheme, neither of which appeared before.

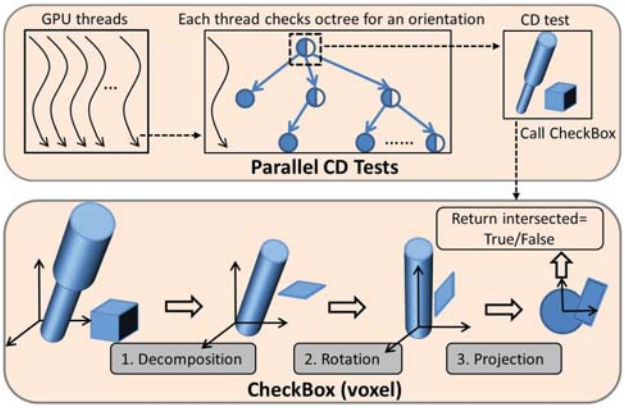


Figure 4: Our baseline algorithm that performs the parallel CD tests by calling CheckBox, which involves the three steps that cost 216 operations.

2 BACKGROUND AND MOTIVATION

Problem Statement. The inputs of our CD problem are (a) a 3D object, which is the *target* (e.g., the head); (b) another 3D object, which is the *tool*; (c) a pivot point upon which one end of the tool is fixed; and (d) the set of (discretized) orientations of the tool to consider. The output is an accessibility map (AM) that indicates whether or not each orientation leads to a collision between the two objects. Figures 1 and 2 illustrate these inputs and outputs.

To efficiently detect collisions, we will assume the setup of Figure 3. The target object is represented by a high-resolution volumetric (voxel-based) adaptive octree and the tool object is enclosed within a collection of simple bounding cylinders. Both octree and bounding volumes (BVs) are spatial data structures widely used in many applications [3, 16, 18–20, 28]. We denote the total number of voxels (root + interior + leaves) in the target object by N, and use M for the number of discrete tool orientations to check. We will consider single GPU-parallel algorithms, where the basic building unit of computation is a CD test, which checks if a given orientation intersects with a given voxel; the adaptive octree will allow both the baseline algorithm and our improved schemes to dynamically prune CD tests when no collisions are possible.

Baseline Algorithm. Figure 4 illustrates our baseline algorithm, which is GPU-parallel. Each GPU thread considers an orientation,

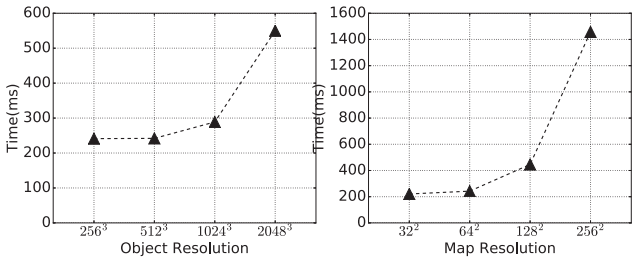


Figure 5: Execution time of varying the object resolution in the head model (map size is 64^2) and varying the map resolution (object resolution is 1024^3), with Algorithm 2.

traversing the octree to determine whether that orientation yields any intersections with the target. (The code that each thread runs is CheckOctree, shown in Algorithm 2.) During its traversal, the thread performs a CD test at each voxel it traverses, assessing whether the tool at the given orientation intersects the voxel. The intersection calculation is performed by a subroutine referred to as CheckBox, which consists mainly of three computationally intensive geometrical operations: *decomposition*, *rotation*, and *projection*. The decomposition step decomposes the tool into one or more cylinders, the voxel into 6 faces, and each face into 4 line segments. Secondly, rotation changes the coordinate frame so that the cylinder becomes axis-aligned, which greatly simplifies some subsequent calculations and requires 9 elementary operations (e.g., scalar arithmetic). Thirdly, the projection step projects the geometries from 3D space to 2D. In total the algorithm CheckBox executes at most $N_c \cdot 6 \cdot 4 \cdot 9 = 216 \cdot N_c$ elementary operations.

Initial experimental study. To gain some intuition for how this baseline performs, consider the following experiment on a NVIDIA GTX 1080Ti GPU (see Section 5.1 for hardware details).

Suppose we generate an AM with the baseline algorithm. Figure 5 shows the execution time as we vary either the object resolution ($N = k^3$ voxels on the x-axis of the left subplot) or accessibility map resolution ($M = l^2$ along the x-axis of the right subplot). Even though increasing the object resolution sharply increases the number of voxels in the octree representation, the largest observed increase in execution time is a factor of two (2) when increasing the number of voxels by eight (1024^3 to 2048^3 grid). This scaling behavior is sublinear in resolution because the octree induces a pruning of possible checks. By contrast, when the map resolution increases from 128^2 to 256^2, a 4× increase in cells, the corresponding execution time also increases by the same factor. This observation is also not surprising as the total amount of work is proportional to the number of orientations being tested. For relatively low-resolution accessibility maps (e.g., 32^2 or 64^2), the execution time appears flat; that behavior is due to the number of threads of work being less than or comparable to the number of physical execution cores (3,548 CUDA cores on this particular system). However, that absolute time is high enough to prohibit real-time CD. (Real-time is not required in CNC milling but can be in other graphics problems.)

Our paper focuses on the performance improvement of the CD test between cylinders and voxels. It contains two parts: ICA abstraction in Section 3 and our parallel algorithm AICA in Section 4. ICA aims to reduce the cost of a single CD test. AICA aims to improve parallelism and load balance.

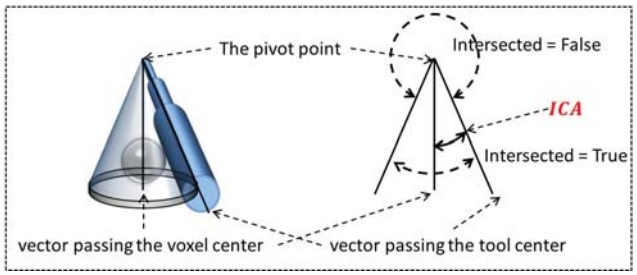


Figure 6: (*Left*) How a cone is formed with the tool cylinders exactly touching the surface of the sphere in 3D. (*Right*) How ICA is formed to check the intersection in 2D.

3 ICA ABSTRACTION

We improve the baseline by first making it more work-efficient, namely, by reducing the high cost of the three basic geometrical operations of decompositions, rotations, and projections. Our approach is an abstraction, the *inaccessible cone angle* (ICA), which simplifies the 3D operations into 2D equivalents.

3.1 Spherical approximation

First, consider the following approximation, designed to reduce the complexity of a CD test: replacing a voxel by a sphere.

A general strategy to make CD tests cheaper is to *axis-align* the objects, that is, perform the calculations in a coordinate frame where one or more axes align "naturally" to one of the objects. Since it is rare that the two objects of an intersection test are simultaneously axis-aligned, we need to axis-align one object and rotate the other. In our method, we choose to axis-align the cylindrical tool because projecting the side surface of a cylinder is more complex than projecting the face of a voxel.

However, calculating the new coordinates of the voxel's geometric elements (e.g., faces, edges) in this new coordinate frame can still be high. In 2D, an axis-aligned line segment becomes skewed after a rotation, and so does a square. In 3D, this problem worsens because a voxel has multiple (six) faces.

By contrast, a sphere would be naturally neutral to axis-alignment regardless of how it is rotated. Consequently, the complexity of an elementary intersection test involving the sphere would be invariant to its rotation. We could, therefore, approximate the voxel by, say, a circumscribed sphere. Doing so truncates the corners, but it is possible to resolve the inaccuracy introduced by this approximation. We explain how in Section 3.3.

3.2 Inaccessible Cone Angle (ICA)

Suppose we are given (i) a spherical object approximated by a voxel, (ii) a tool composed of several cylinders, and (iii) a position where one end of the tool is fixed. Our goal is to calculate all the *inaccessible* orientations, that is, orientation angles at which the tool collides with the spherical object. Any remaining orientations are *accessible*. With this setup, we propose the concept of an *inaccessible cone angle*, or ICA, which represents the possible region of intersection between the tool and the voxel.

Figure 6 gives a general example on how a cone is formed in considering a potential collision. On the left, there is the tool, a

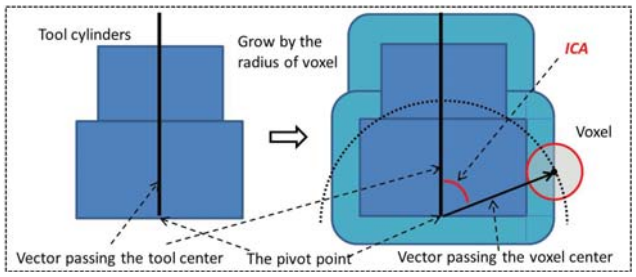


Figure 7: The tool cylinders and the voxel are simplified into rectangles and a circle. The voxel's ICA value is calculated as the maximum angle that the circle touches the surface of rectangles.

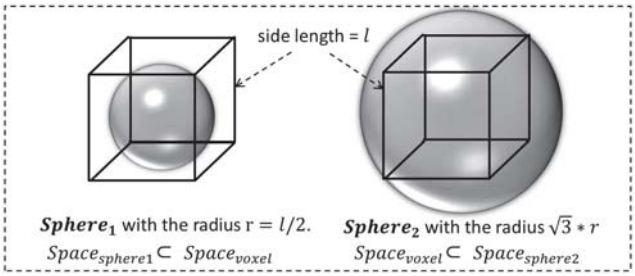


Figure 8: Two spheres are constructed for each voxel. For sphere_1**, its surface is tangent to the 6 sides of the voxel, and for** sphere_2**, the voxel's 8 corners are on its surface.**

target sphere, and a pivot point, with vectors from the pivot through the centers of the sphere and the tool. Observe that the tool may touch the sphere at many points, but that the angle between the tool vector and the target vector is constant in all cases. The set of all orientations for which the tool surfaces touch the sphere forms a cone, which we refer to as the *inaccessible cone*: all tool directions within the cone will yield a collision (intersected=**True**), while directions outside are collision-free (intersected=**False**).

The inaccessible cone is associated with an angle between two vectors, one passing through the center line of the tool and the other passing through the center of the voxel. This angle is the ICA, calculated as a 2D value; see Figure 6 (*right*). The ICA is the largest angle at which the tool collides with the sphere, or, conversely, the smallest angle at which the tool does not do so.

Algorithm 1 CheckICA

```
Input: tool as cylinders, orientation S_i, pivot point p, voxel
1:  procedure CheckICA(cylinders, S_i, p, voxel)
2:      r ← radius of sphere within the voxel
3:      v ← the center of the voxel
4:      ica_1 ← GetToolICA(cylinders, p, v, r)
5:      ica_2 ← GetToolICA(cylinders, p, v, √3 r)
6:      vector_1 ← the center line of tool at orientation S_i
7:      vector_2 ← the vector passing p and c
8:      angle ← the angle between vector_1 and vector_2
9:      if angle ≤ ica_1 then
10:         return intersected = True
11:     else if angle ≥ ica_2 then
12:         return intersected = False
13:     else                                   ▷ Corner case
14:         return CheckBox(cylinders, S_i, p, voxel)
15:     end if
16: end procedure
```

To calculate an ICA, one must determine at which points a circle might just touch a given rectangle (tool). Figure 7 illustrates how to do so. A cross-section of a cylinder that passes its center line is a constant rectangle, and a cross-section of a sphere that passes its center is a constant circle, so these 2D geometries are used to represent the original 3D objects. Our idea starts by logically expanding the size of the rectangle by the radius of the voxel. Then, fixing the distance between the pivot point and the center of the voxel, it determines all points along an arc, centered at the pivot, that intersect the expanded rectangle. These points are *crossed points*. A crossed point might be located at any point on the border of the expanded rectangle, whether it be on the side, the bottom, or the corner. Crossed points correspond to centers of voxels whose circumscribed sphere just touches the original rectangle.

3.3 CheckICA algorithm to preserve accuracy

The preceding procedure approximates a voxel by a sphere. The resulting CD test may, therefore, yield false-positive reports of accessibility. For instance, if the tool intersects with a corner of the voxel—a literal "corner case"—the approximation will report "accessible" because that corner is outside the sphere. This case is detrimental in CNC milling, where any collision could damage the target part or tool.

The algorithm CheckICA in Algorithm 1 covers such cases. It considers *two* spheres at each voxel, one inscribed within the voxel (Sphere_1) and one circumscribed about the voxel (Sphere_2), as shown in Figure 8. Each of these spheres yields an ICA value. Then, it verifies the following two conditions by comparing the two ICA values as in line 9 and 11. Lastly, if the angle lies between the two ICA values, which is a ***corner case*** in line 13, we cannot verify the intersection using only the definition of ICA; therefore, we must fallback to the original CheckBox described in Section 2. If in the *absence* of a corner case, the cost of invoking CheckICA is $N_c \cdot 2 \cdot 5 + 3 = 10 \cdot N_c + 3$ operations, where $10 \cdot N_c$ is the cost of calculating ICA and 3 is the cost of verifying the intersection with the ICA values. Here 2 means the 2 spheres; N_c denotes the number of cylinders in the tool; 5 means there are 5 components to check for each rectangle (cylinder).

One question is how often the CheckICA algorithm might need to invoke the baseline CheckBox, or what is the probability that we encounter a corner case. We define the ***ICA efficiency*** as the fraction of intersection tests that do *not* resort to calling CheckBox. That is, an efficiency of zero means we always call CheckBox, and a value of 1 means we never need to call CheckBox.

Figure 9 derives a theoretical estimate of ICA efficiency, in a simple setting where the cylinders are approximated by a straight line and there are an infinite number of orientations to check. ICA efficiency is inversely proportional to (r/dist), where r is the radius of a voxel, and dist is the distance from the pivot point to the center

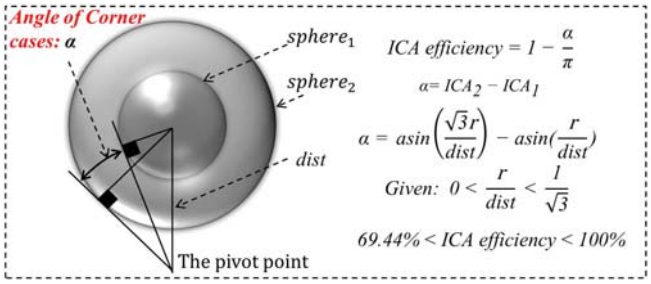


Figure 9: Theoretical ICA efficiency analysis. We assume that the sizes of the cylinders do not influence the ICA value and the tool becomes a straight line.

of the voxel. In practice, for most voxels the distance should be much larger than the radius, resulting in a higher ICA efficiency than the minimal value: the pivot point must be outside the 3D object, and a point inside the object must result in a collision. The relation between the distance and the radius are crucial to ICA efficiency. As the resolution of the target object increases, the voxel will have a smaller r, thereby yielding a higher ICA efficiency. Thus, ICA efficiency benefits naturally from high-resolution representations.

The concept of an ICA confers several benefits. First, using the ICA in elementary tests does not require any decomposition, since it represents the entire tool. Secondly, the ICA value is independent of the given orientation of the tool. Thus, regardless of the number of orientations a test needs to check, we need only compute the ICA once. Thirdly, the ICA does not require any expensive rotations or projections thanks to the spherical approximation. Compared with CheckBox, CheckICA reduces the overall cost from $216N_c$ operations to $10N_c + 3$ operations, a roughly 20-fold decrease.

4 DESIGN OF PARALLEL AICA

Our approach, AICA, consists of two stages: parallel ICA calculation and parallel CD tests. These are illustrated in Figure 10. The inputs to the first stage are the target octree and tool geometry, and the output is a memoized table storing that holding some precomputed ICA values for the upper-levels of the tree. The number of levels is a tunable parameter, described below. In the second stage, each logical GPU thread again checks one orientation; however, when it performs a CD test and calls CheckICA, the CheckICA algorithm can now use the memoized table to look up the precomputed ICA values instead of computing them on-the-fly. Any yet-to-be-computed ICA values are computed on-demand. The rest of this section describes our approach to GPU thread mapping, parallelization of the CD tests, and reduction of costly corner cases.

4.1 GPU threads mapping

For the parallel ICA calculation, we compute the values of the voxels on the top S levels in octree. Each thread corresponds to a voxel, yielding a highly efficient SIMDization.

For the parallel CD test, given the workload with the N voxels and the M orientations, there are two natural parallelization strategies. One is to partition the octree among threads, and then each thread processes M orientations. The other is to map each orientation to a thread and then each thread will traverse the octree, as with the baseline algorithm. We use the latter, for two reasons. One

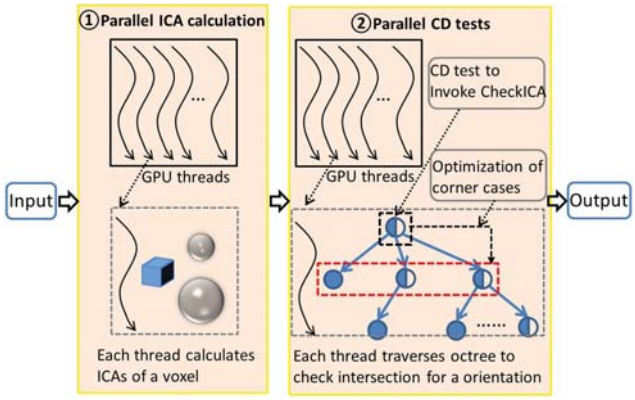


Figure 10: Overview of aggressive inaccessible cone angle (AICA) with two stages: parallel ICA calculation and parallel CD tests.

Algorithm 2 CheckOctree

```
     Input: tool, orientation S_i, pivot point p, octree
1:   procedure CheckOctree(tool, S_i, p, octree)
2:       stack ← {voxels at the top level of octree}
3:       while !stack.IsEmpty() do
4:           voxel = stack.pop()
5:           intersected = CheckBox(tool, S_i, p, voxel)
6:           if !intersected then
7:               continue
8:           else if intersected & voxel.IsLeafNode() then
9:               return True
10:          else
11:              stack.push_back(voxel.children())
12:          end if
13:      end while
14:      return False
15:  end procedure
```

is that it enables more aggressive exploitation of the adaptive octree for pruning. Finding an interior voxel node that does not intersect with the tool can avoid any calculations on all of its descendants; similarly, a solid voxel that intersects with the tool means that we can directly return that a collision will occur. Another reason is simplicity: assigning threads to orientations is an owner-computes strategy that avoids communication and synchronization. The overall algorithm, which each thread executes, appears in Algorithm 2.

4.2 Mitigating load imbalance

The choice of thread mapping affects load imbalance in the baseline. The execution time of a thread is determined by the number of checks at the line 3 of Algorithm 2, which varies with each orientation. To mitigate this load imbalance, we calculate ICAs of the voxels on upper S levels of octree in parallel as a precomputation stage before the parallel CD tests, rather than calculating ICA at runtime as shown in the left side of Figure 11. In practice, each thread's calculation of an ICA and comparisons alternate with checks, and the time spent in the two phases appears in Figure 11. We create,

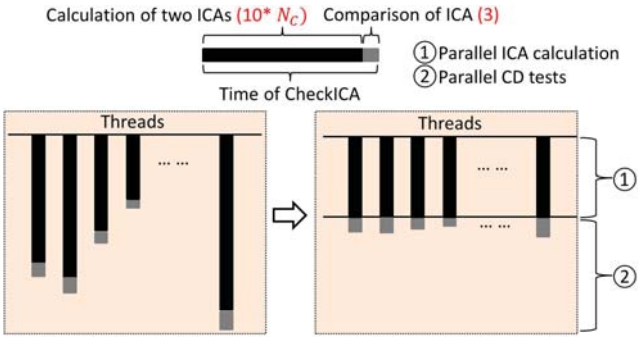


Figure 11: The parallel ICA calculation mitigates load imbalance and improves the performance, by saving the cost of redundant ICA calculation and efficient parallelization.

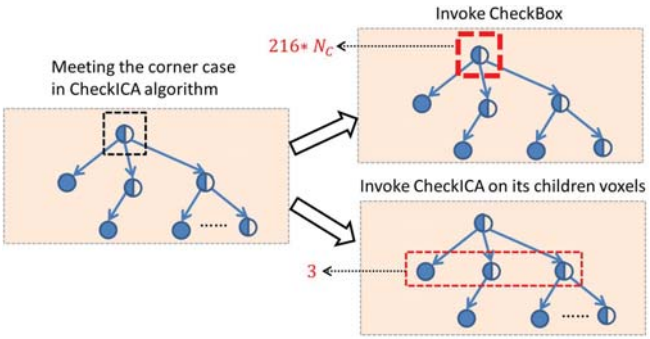


Figure 12: Optimization on the corner cases. When meeting a corner case on a voxel, AICA algorithm expands it into its children voxels and calls CheckICA recursively.

for each voxel in the target, a memoized table of ICA values. These are the values labeled ica_1 and ica_2 for the two spheres, meaning one pair per voxel. This increased storage is a modest fraction of the total voxel storage and this precomputation is pleasingly parallel since there are no inter-voxel dependencies. Precomputation is feasible because ICA is independent of the tool's orientation.

This approach confers three overall benefits. First, it avoids redundant calculations as the ICA values in the table are calculated once but reused by all threads. Secondly, it mitigates load imbalance as calculating ICA in the precomputation stage is easily parallelizable, at the granularity of voxels. Lastly, it reduces the execution time of all threads and thus improves overall performance because of efficient SIMDization on GPU.

As S increases, the total cost of all CheckICA tends to increase, whereas the amortized cost of CheckBox tends to decrease. Thus, there is a tradeoff. A heuristic is that S can be set to a relatively higher value on recent, more powerful GPUs (Section 5.4).

4.3 Optimization on the corner cases

For corner cases, we may still need to invoke the baseline CheckBox to verify the intersection. However, we can reduce the corner case cost by utilizing the hierarchical spatial structure as an optimization.

Suppose that the algorithm stops at a voxel facing the corner case as shown in Figure 12 (left). We have two choices. One is to directly invoke the baseline algorithm. The other is to expand the voxel into its children voxels, and then apply our CheckICA algorithm recursively on each voxel; the recursion stops when no further expansion is allowed, in which case CheckBox is still used as the fallback. Our optimization approach is to choose the latter.

The cost of this approach is an increase of the number of checks resulting from the expansion. Nevertheless, the benefit is the reduction in cost of CD test by invoking CheckICA. We believe that the benefit largely outweighs the cost. Note that the cost of a single CheckICA is 3 here, rather than $N_c * 10 + 3$, since most voxels' ICA values have already been calculated in the precomputation stage. The corner case is also an important factor that causes the load imbalance, so we will benefit from optimizing it, too. This tradeoff will be evaluated in detail in Section 5.2.

5 EVALUATION

We evaluated three aspects of our approach: (1) examining the impact of the parallelism method and verifying the efficacy of ICA efficiency; (2) assessing absolute performance with various object resolutions and various AM resolutions; (3) analyzing the cost of the parallel ICA calculation under various configurations.

Our comparison includes AICA and four other schemes:

- A parallel box (***PBox***), which is the baseline algorithm with parallel CD tests using CheckBox.
- ***An optimized* PBox** is still on the baseline algorithm but using axis-aligned bounding boxes (AABBs). The optimization is to apply AABBs on the voxel after each rotation. If no intersection exists on the bounding box, we can directly return False.
- A parallel ICA (***PICA***), which is the algorithm with the parallel CD tests using CheckICA.
- A memoized ICA (***MICA***), which is the algorithm that has the parallel CD tests using CheckICA and has the parallel ICA calculation but without the optimization of corner cases.
- Our approach, ***AICA***, which has both.

The first two—*PBox* and *optimized PBox*—represent the state-of-the-art, and are both implemented in SculptPrint.

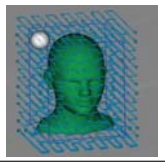
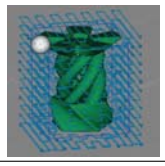
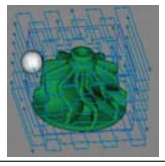
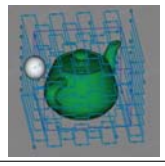
Table 2: Experimental test platforms.

Two Setups	GTX 1080 Ti	GTX 1080
CPU	Intel i7-2600K 3.40GHZ	Intel i7-7820HK 2.90GHZ
DRAM	16GB	32GB
OS	Windows 7	windows 10
CUDA runtime	9.1	10.0
GPU card	11GB, 1.68GHZ 3548 CUDA cores	8GB, 1.77GHZ 2560 CUDA cores

5.1 Experimental Setup

CAD Benchmarks. Our experiments use 4 CAD benchmarks for evaluation. A summary of the input meshes and their detailed

Table 1: Geometric statistics of sample CAD Benchmarks.

	Head				Candle Holder				Turbine				Teapot			
																
Number of Triangles	23028				38000				57792				57600			
Dimension XYZ(mm)	48.6*46.0*64.4				48.4*48.9*57.7				48.9*48.9*31.1				46*46*31			
Bounding Volume	51331				21275				7823				25619			
Effective Resolution	256^3	512^3	1024^3	2048^3	256^3	512^3	1024^3	2048^3	256^3	512^3	1024^3	2048^3	256^3	512^3	1024^3	2048^3
#layers	6	7	8	9	7	7	8	9	6	7	8	9	6	7	8	9
#voxels in octree(10^6)	0.44	1.06	4.26	17.56	0.57	1.59	5.92	26.94	0.62	1.37	6.44	26.06	0.74	1.53	6.14	23.89
#points on path(10^3)	61.14	101.3	203.7	409.3	58.32	97.32	196.9	360.6	29.43	41.46	83.48	168.2	30.60	44.57	89.37	179.1

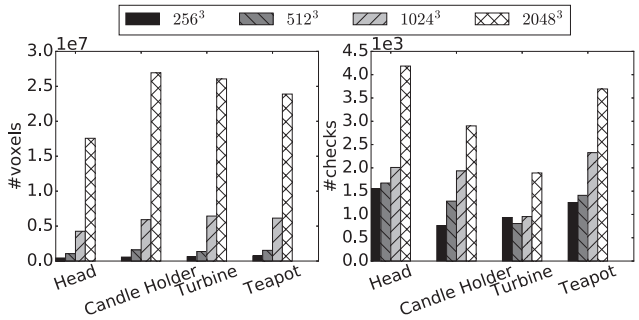


Figure 13: Comparison between the number of voxels in octree and the number of checks under various object resolutions. The actual number of checks on the critical thread is much smaller than the total number of voxels.

geometrical characteristics are listed in Table 1. For each benchmark, we evaluate 4 target resolutions on the construction of octree, from 256^3 to 2048^3. The tool geometry has 4 cylinders, with varying radii (31.5, 20, 6.225, 6.35)mm and heights (22.1, 78, 76.2, 25.4)mm. The AM resolution starts from 32^2 to 256^2. To choose representative pivot points, we generate a path surrounding the CAD models, with each point on the path having a 1 mm distance from the surface of the model.

Configuration. We implement our algorithms in SculptPrint, a computer-aided manufacturing (CAM) application for producing CNC tool paths [9]. During the process of generating an AM, we assume that all of the information about the 3D object model has already been loaded onto the GPU, since this information is read-only and only need only be loaded once. Thus, the cost of the transferring is excluded in our experimental results.

In our experiments, 2000 random points are chosen from the path as the pivot points. The last row of Table 1 shows the total number of points on the path. Every experimental result in this section is the average value of the 2000 samples. We directly expand the top 5 levels of octree into one level, and report the final number of levels under various resolutions in Table 1. This expansion aims at reducing the height of octree (see the load imbalance part in Section 5.2). The parameter S is set to 8, which means that the parallel ICA calculation stage computes the ICA values for the voxels on the upper 8 levels (~7 billion voxels).

5.2 Analysis of Parallelism Exploitation

Threads mapping. Figure 13 shows the total number of voxels under various resolutions for the 4 models. However, the total number of voxels does not reflect the actual workload on each thread. The right-hand plot presents the actual number of checks on the critical thread. Because of the spatial hierarchy of octree in Algorithm 2, each thread unnecessarily traverses all voxels. It is obvious that the number of the actual checks is much smaller than the number of voxel in octree, indicating that our approach of the threads mapping is very efficient.

Mitigating load imbalance. Figure 14 shows how the parallel ICA calculation stage mitigates load imbalance and, thereby, improves performance. The leftmost plot shows the actual number of checks executed by each thread. The leftmost and rightmost threads run the same number of checks, because we expanded the top 5 levels into 1 level as mentioned before, and these threads have to check all voxels on the top level before returning. In the two plots of the second column, we can see that the execution time is proportional to the number of checks, where CheckICA is used for CD tests. Comparing the two GPU cards, we can see that the time on GTX 1080 is a little shorter than the other, because GTX 1080 has a higher clock rate 1.77 GHz than 1.68 GHz of GTX 1080 Ti. In the two plots of the third column, the bottom area represents the time of the parallel ICA calculation, which mitigates the load imbalance by reducing the execution time of calculating ICA values. Note that the CD tests in the corner cases are not influenced by the parallel ICA calculation, which still takes a relatively long period of time. Comparing the two GPU cards, we can see that the time of the parallel ICA calculation on GTX 1080 (~ 3.8 ms) is longer than the other (~ 3.1 ms), because GTX 1080 has 2560 CUDA cores while GTX 1080 Ti has 3548 CUDA cores.

The last column of Figure 14 shows the cost of the corner cases and the effect of our optimization technique, which effectively improves performance further from the previous column.

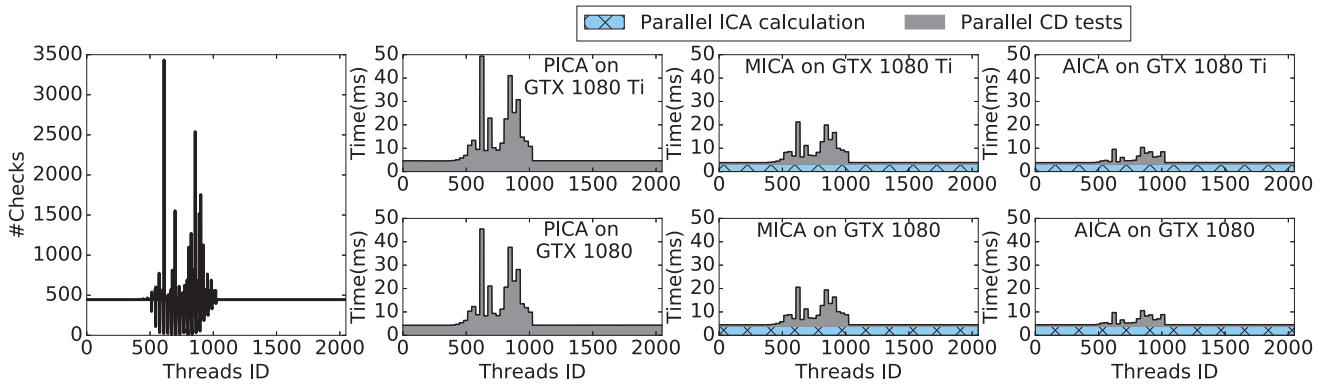


Figure 14: The parallel ICA calculation mitigates the load imbalance and improves the performance, with the head model with 1024^3 resolution and 2048 orientations. The first column plot shows the actual number of checks on all threads.

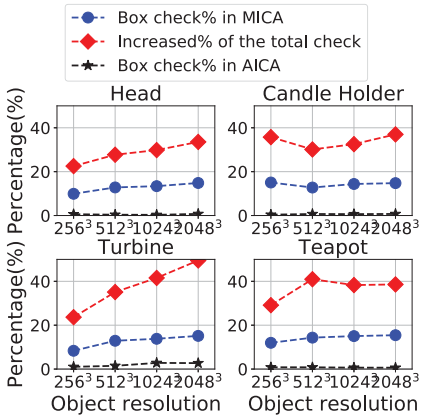


Figure 15: Optimization of corner cases. An intentional increase of the total checks is made to reduce the number of Box checks from MICA to AICA, where ICA efficiency (=1-Box checks%)

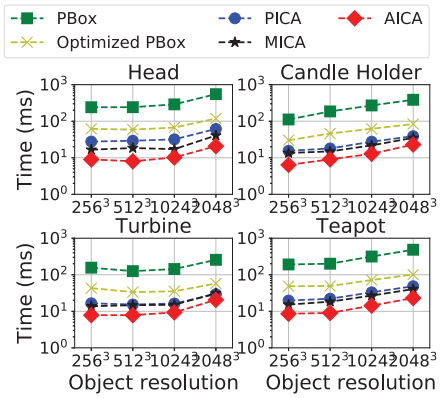


Figure 16: Averaged execution time of 5 approaches with various object resolutions. Our approach AICA performs 23.9× faster than the approach of CheckBox, and 4.8× faster than our best optimized version of CheckBox.

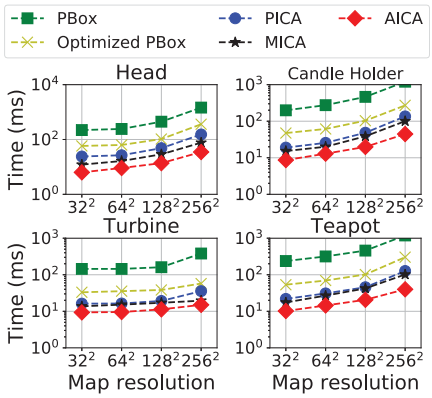


Figure 17: Averaged execution time of 5 approaches with various AM resolutions. Our approach AICA performs 20.2× faster than the approach of CheckBox, and 4.1× faster than our best optimized version of CheckBox.

Optimization of corner cases. Since the performance of generating the AM is determined by the thread that has the longest execution time, namely the critical thread, we only report the execution on the critical thread.

Figure 15 reports three types of percentages: box checks in MICA, box checks in AICA, and the increased percentage of the total check from MICA to AICA. Recall that our optimization of corner cases reduces box checks at the expense of increasing the total checks, where the cost of a single ICA check is much smaller than the cost of a single box check. We can see that the percentage decreases from 14.4% to 0.9% on average, comparing AICA with MICA, resulting in a 34.1% increase on the total number of checks. Suppose that we need to reduce S number of box checks; then, the number of required ICA checks should be larger than the number S as the cost, because one Box check demands a substitute of multiple ICA checks on the expanded children voxels.

However, increasing the number of ICA checks is worthwhile because doing so reduces the number of box checks and improves performance, given the inherent difference in costs between the two types of checks. The remaining percentage of box checks is the actual ICA efficiency, which is 99% on average, indicating that 99% of CD tests benefit from the ICA abstraction.

5.3 Overall Performance Results

Varying Object resolution. Figure 16 shows the average execution time for all 4 models. For PICA, it is 23.9× faster than PBox, and 4.8× faster than the optimized version on average of the 4 models. That is because CheckICA only needs 2D computations with the ICA and avoids most of the three computation-intensive operations that exist in CheckBox. MICA improves the speed by 28.3% on average compared to PICA. The simple parallelization of the ICA precomputation is faster than the on-the-fly ICA computation, even though MICA memoizes ICA values for all voxels, while PICA only applies the calculations on the voxels in the current test. Note that the cost of the precomputation increases as the number of the voxels grows. On the 256^3 resolution, the improvement is 32.5% while for size 2048^3, the improvement becomes only 19.3%. A detailed discussion of the cost on varying the number of voxels

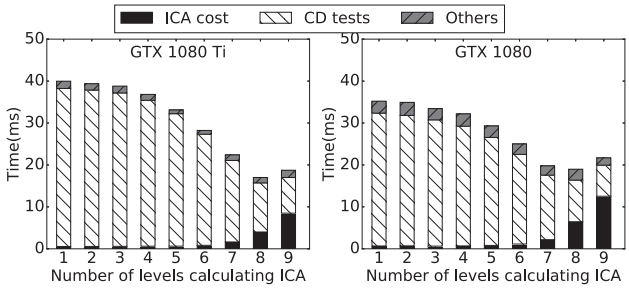


Figure 18: Time breakdowns under various numbers of layers in octree, using the head input model in 2048^3 resolution with AICA approach. Though the ICA cost increases as the growth of the layers, the overall performance is improving.

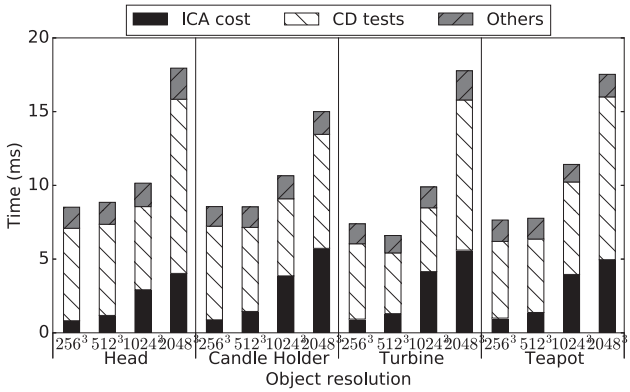


Figure 19: Time breakdowns under various resolutions of object models with AICA. As the object resolution rises, most of the increasing portion comes form the ICA cost.

in octree is given in Section 5.4. For AICA, it is 81.1% faster than MICA on average, indicating that increasing the total number of the checks can still yield a higher ICA efficiency. A detailed analysis of this tradeoff is given in Section 5.2.

Varying the resolution of the AM. Increasing the resolution of the AM leads to a large number of orientations to check. The object model resolution is fixed to 1024^3. The 32^2 resolution requires 1024 threads and 64^2 needs 4096 threads. The experiments run on GTX 1080 Ti that has 3548 cores. This number of cores explains why the increasing ratio of the execution time from 32^2 to 64^2 is smaller than the others. Figure 17 presents the average execution time of all 4 models. For PICA, it is 20.2× faster than PBox on the average of the 4 models, and 4.1× better than our optimized CheckBox. For MICA, it is improved by 39.5% compared to PICA. AICA achieves 84.8% improvement than MICA due to a high ICA efficiency, which it includes the cost of creating the memoized table.

5.4 Cost Analysis

Calculating ICA affects execution time in the manner shown in Figure 18. There, the x-axis represents the number of the upper levels in octree that we calculate the ICA values. If the ICA value is not precomputed, it will be calculated in runtime. In both subplots, for the upper 3-4 levels in octree, there is not much time reduction of CD tests. This is because the execution time is always bounded by the critical thread. Starting from the 5 upper level, an obvious reduction of the time appears, and meanwhile, the cost of calculating ICA increases as the number of voxels grows exponentially. Taking the cost into account, it is still worth calculating ICA for the voxels on the uppper 8 levels to speed up the CD tests.

We also explore variations in the object resolution from 256^3 to 2048^3. Figure 19 presents the performance results. The execution time increases gradually, as the cost of ICA calculation increases with an exponential growth of the number of voxels. Therefore, we believe that using a more powerful GPU card can reduce the cost further, and thus achieve better performance. If we do not have a new GPU card, we still can gain a decent performance by tuning the parameter S to control the cost of the parallel ICA calculation.

6 APPLY ICA TO BOUNDING BOX

ICA abstraction is also applicable to bounding box. Bounding box is a general data structure used to simplify computation by enclosing a series of arbitrary objects. Similar to the way that a voxel is approximated by 2 spheres described in Figure 8, a bounding box can also be approximated into 2 cylinders. Both cylinders are able to use ICA for the checking. There should be certain corner cases that are not covered by ICA, but the percentage should be very small, similar to our ICA efficiency analysis. Therefore, ICA is a general geometric abstraction that substantially simplifies CD tests. We believe that our ICA abstraction will be implemented into a new standard elementary procedure in many other tests, such as cylinder-sphere, sphere-box, and box-box CD tests.

7 RELATED WORK

The problem of efficient CD has been well studied and excellent surveys are available [6, 13, 31]. In this section, the approaches of improving the performance of CD are classified into three categories: acceleration using graphics representations, acceleration using novel parallelization schemes, and lastly others.

Many graphics techniques including spatial data structure [16, 18–20], culling [7], ray casting [12], visibility query and collision map, are commonly used for approximation of CD [1, 30, 32]. By using depth maps store distance values to represent outer shape of objects, Kolb [14] handles collision detection and reaction of particles with objects for arbitrary shape in massively parallel simulations. Sucan [24] describes a collision map data structure, which uses axis aligned cubes to model the point cloud and to perform collisions. Bounding sphere is commonly used for CD acceleration as an encapsulation of the target object [2, 8, 11, 21] with an approximation. Our proposed approach targets on the performance improvement of elementary CD tests and preserves accuracy.

Many algorithms [17, 18, 23, 25, 27, 33] have been proposed to exploit computational capabilities of a multi-core platform. Lauterbach [17] presents novel GPU-based algorithms to efficiently perform collision and separation distance queries using tight-fitting bounding volumes. With the goal to compute collision-free paths for robots in complex environments, Pan [23] presents a novel GPU-based parallel algorithm to perform collision queries for sample-based motion planning. A novel hybrid parallel continuous collision detection is proposed by Kim [10], which utilizes both CPUs and GPUs to achieve the interactive performance of CD. Instead of focusing a shared-memory test bed, Du [4] targets on high-performance

cluster with a parallel continuous collision detection(CCD) algorithm aiming to accelerate CCD culling by efficiently distributing workload. Our parallel algorithm can be integrated with these various parallel schemes to explore efficient parallelisms.

Pan [22] formulates collision checking as a two-class classification problem, applying machine learning to compute the collision probability for acceleration. Ding [3] conducts the interference detection between the tool oriented bounding boxes and the gray octants of the surface octree in order to simplify the computation process of updating tool positions and orientations in 5-axis machining. Zhiwei [34] proposes an efficient algorithm of CD to generate tool posture collision-free area for the whole free-form surface by sampling and cubic B-surface interpolation. Since these techniques still need to process the elementary CD tests, our proposed abstraction can be embedded to further improve the performance.

8 CONCLUSIONS AND FUTURE WORK

The key ideas of our proposed methods are the ICA concept, which is a new geometric abstraction for the CD problem, and its parallel algorithm AICA, including the mitigation of load-imbalance and the optimization on corner cases. We have prototyped our AICA algorithm within a real CNC milling tool, SculptPrint [9]. Experimental results on 4 CAD benchmarks demonstrate that AICA is up to 23× faster than the approach of the traditional approach.

While our results show ICA can be effective, our experimental analysis also identifies several new opportunities. For instance, neighboring pivot points, which were outside the scope of this paper, are likely to have AM with overlapping values. Therefore, future work should develop methods to reuse the AM values among nearby pivots. Another idea is to construct an algorithm that can intelligently tune the parameter S to adjust the cost of the parallel ICA calculation. Lastly, to broaden its use in computer graphics, our AICA should be extended and tested against other spatial volume structures common in that domain, such as BVH (Section 1) and kd-trees, among others.

REFERENCES

[1] Jeremy A Carter, Thomas M Tucker, and Thomas R Kurfess. 2008. 3-Axis CNC path planning using depth buffer and fragment shader. *Computer-Aided Design and Applications* 5, 5 (2008), 612–621.

[2] Jung-Woo Chang, Wenping Wang, and Myung-Soo Kim. 2010. Efficient collision detection using a dual OBB-sphere bounding volume hierarchy. *Computer-Aided Design* 42, 1 (2010), 50–57.

[3] S Ding, MA Mannan, and Aun Neow Poo. 2004. Oriented bounding box and octree based global interference detection in 5-axis machining of free-form surfaces. *Computer-Aided Design* 36, 13 (2004), 1281–1294.

[4] Peng Du, Elvis S Liu, and Toyotaro Suzumura. 2017. Parallel continuous collision detection for high-performance GPU cluster. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, 4.

[5] 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.

[6] Christer Ericson. 2004. *Real-time collision detection*. CRC Press.

[7] Naga K Govindaraju, Stephane Redon, Ming C Lin, and Dinesh Manocha. 2003. CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association, 25–32.

[8] Philip M Hubbard. 1996. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics (TOG)* 15, 3 (1996), 179–210.

[9] Tucker Innovations Inc. [n. d.]. SculptPrint. http://www.sculptprint.com. Accessed: 2018-09-30.

[10] Duksu Kim, Jae-Pil Heo, Jaehyuk Huh, John Kim, and Sung-eui Yoon. 2009. HPCCD: Hybrid parallel continuous collision detection using CPUs and GPUs. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 1791–1800.

[11] Dong-Jin Kim, Leonidas J Guibas, and Sung-Yong Shin. 1998. Fast collision detection among multiple moving spheres. *IEEE Transactions on Visualization and Computer Graphics* 4, 3 (1998), 230–242.

[12] David Knott. 2003. *CInDeR: collision and interference detection in real time using graphics hardware*. Ph.D. Dissertation. University of British Columbia.

[13] Sinan Kockara, Tansel Halic, K Iqbal, Coskun Bayrak, and Richard Rowe. 2007. Collision detection: A survey. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*. IEEE, 4046–4051.

[14] Andreas Kolb, Lutz Latta, and Christof Rezk-Salama. 2004. Hardware-based simulation and collision detection for large particle systems. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, 123–131.

[15] Dmytro Konobrytskyi. 2013. *Automated CNC tool path planning and machining simulation on highly parallel computing architectures*. Ph.D. Dissertation. Clemson University.

[16] Dmytro Konobrytskyi, Mohammad M Hossain, Thomas M Tucker, Joshua A Tarbutton, and Thomas R Kurfess. 2018. 5-Axis tool path planning based on highly parallel discrete volumetric geometry representation: Part I contact point generation. *Computer-Aided Design and Applications* 15, 1 (2018), 76–89.

[17] Christian Lauterbach, Qi Mo, and Dinesh Manocha. 2010. gProximity: hierarchical GPU-based operations for collision and distance queries. In *Computer Graphics Forum*, Vol. 29. Wiley Online Library, 419–428.

[18] Orion Sky Lawlor and Laxmikant V Kalée. 2002. A voxel-based parallel collision detection algorithm. In *Proceedings of the 16th international conference on Supercomputing*. ACM, 285–293.

[19] Roby Lynn, Didier Contis, Mohammad Hossain, Nuodi Huang, Tommy Tucker, and Thomas Kurfess. 2017. Voxel model surface offsetting for computer-aided manufacturing using virtualized high-performance computing. *Journal of Manufacturing Systems* 43 (2017), 296–304.

[20] Roby Lynn, Mahmoud Dinar, Nuodi Huang, James Collins, Jing Yu, Clayton Greer, Tommy Tucker, and Thomas Kurfess. 2018. Direct Digital Subtractive Manufacturing of a Functional Assembly Using Voxel-Based Models. *Journal of Manufacturing Science and Engineering* 140, 2 (2018), 021006.

[21] Ian J. Palmer and Richard L. Grimsdale. 1995. Collision detection for animation using sphere-trees. In *Computer Graphics Forum*, Vol. 14. Wiley Online Library, 105–116.

[22] Jia Pan, Sachin Chitta, and Dinesh Manocha. 2017. Probabilistic collision detection between noisy point clouds using robust classification. In *Robotics Research*. Springer, 77–94.

[23] Jia Pan and Dinesh Manocha. 2012. GPU-based parallel collision detection for fast motion planning. *The International Journal of Robotics Research* 31, 2 (2012), 187–200.

[24] Ioan A Şucan, Mrinal Kalakrishnan, and Sachin Chitta. 2010. Combining planning techniques for manipulation using realtime perception. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2895–2901.

[25] Avneesh Sud, Naga Govindaraju, Russell Gayle, Ilknur Kabul, and Dinesh Manocha. 2006. Fast proximity computation among deformable models using discrete voronoi diagrams. *ACM Transactions on Graphics (TOG)* 25, 3 (2006), 1144–1153.

[26] Min Tang, Dinesh Manocha, Jiang Lin, and Ruofeng Tong. 2011. Collision-streams: Fast GPU-based collision detection for deformable models. In *Symposium on interactive 3D graphics and games*. ACM, 63–70.

[27] Min Tang, Dinesh Manocha, and Ruofeng Tong. 2009. Multi-core collision detection between deformable models. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*. ACM, 355–360.

[28] Min Tang, Dinesh Manocha, Sung-Eui Yoon, Peng Du, Jae-Pil Heo, and Ruo-Feng Tong. 2011. VolCCD: Fast continuous collision culling between deforming volume meshes. *ACM Transactions on Graphics (TOG)* 30, 5 (2011), 111.

[29] Joshua Tarbutton, Thomas R Kurfess, Tommy Tucker, and Dmytryi Konobrytskyi. 2013. Gouge-free voxel-based machining for parallel processors. *The International Journal of Advanced Manufacturing Technology* 69, 9-12 (2013), 1941–1953.

[30] Joshua A Tarbutton, Thomas R Kurfess, and Tommy M Tucker. 2010. Graphics based path planning for multi-axis machine tools. *Computer-Aided Design and Applications* 7, 6 (2010), 835–845.

[31] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. 2017. *Handbook of discrete and computational geometry*. Chapman and Hall/CRC.

[32] Sai-Keung Wong, Wen-Chieh Lin, Chun-Hung Hung, Yi-Jheng Huang, and Shing-Yeu Lii. 2013. Radial view based culling for continuous self-collision detection of skeletal models. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 114.

[33] Xinyu Zhang and Young J Kim. 2014. Scalable collision detection using p-partition fronts on many-core processors. *IEEE transactions on visualization and computer graphics* 20, 3 (2014), 447–456.

[34] Lin Zhiwei, Shen Hongyao, Gan Wenfeng, and Fu Jianzhong. 2012. Approximate tool posture collision-free area generation for five-axis CNC finishing process using admissible area interpolation. *The International Journal of Advanced Manufacturing Technology* 62, 9-12 (2012), 1191–1203.