2013

Robust Polyhedral Minkowski Sums with GPU Implementation

Min-HO Kyung
Ajou University, kyung@ajou.ac.kr

Elisha P. Sacks
Purdue University, eps@cs.purdue.edu

Victor Milenkovic
University of Miami, vjm@cs.miami.edu

Report Number:
13-001

Robust Polyhedral Minkowski Sums with GPU Implementation

Min-Ho Kyung
Department of Digital Media, Ajou University
kyung@ajou.ac.kr

Elisha Sacks
Computer Science Department, Purdue University
eps@purdue.edu

Victor Milenkovic
Department of Computer Science, University of Miami
vjm@cs.miami.edu

December 22, 2014

Abstract

We present a Minkowski sum algorithm for polyhedra based on convolution. We develop robust CPU and GPU implementations, using our ACP robustness technique to enforce a user-specified backward error bound. We test the programs on 45 inputs with an error bound of 10^{-8}. The CPU program outperforms prior work, including non-robust programs. The GPU program exhibits a median speedup factor of 36, which increases to 68 on the 6 hardest tests. For example, it computes a Minkowski sum with a million features in 20 seconds.

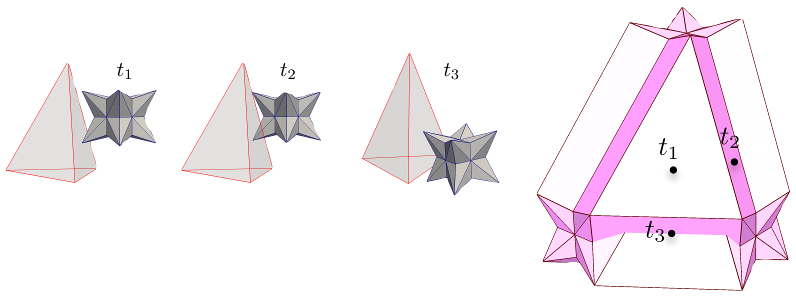

Figure 1: Star and cube snapshots (a) and Minkowski sum (b).

1 Introduction

Minkowski sums are a core computational geometry concept with applications in solid modeling, packing, assembly, and robotics. The Minkowski sum of point sets A and B is

$$A \oplus B = \{a + b | a \in A \wedge b \in B\}.$$

If A and B are polyhedra, $A \oplus B$ is a polyhedron. Fig. 1 shows the Minkowski sum of a star A and a tetrahedron B. Minkowski sums are intimately related to contact analysis. Let $-A + t$ denote A reflected around the origin and translated by t. The polyhedra $-A+t$ and B overlap if t is in the interior of $A \oplus B$ and touch if t is on its boundary. In Fig. 1, the snapshots show $-A + t_i$ and B at three t_i on the Minkowski sum boundary. A vertex of A lies on a facet of B at t_1, two edges are tangent at t_2, and both cases occur at t_3.

The most efficient approach to Minkowski sum computation uses Kaul and Rossignac's [11] subset of the kinetic convolution [3]. Our prior algorithm [18] using that approach is the first robust implementation of any convolution-based method for general polyhedra (Sec. 2). We present an improved algorithm that is faster and that uses less memory (Sec. 3). The computational bottleneck in convolution algorithms is finding which facets intersect. We present a novel kd-tree algorithm that finds all intersecting pairs in a set of geometric objects without splitting objects and without using a hash table to avoid duplicate tests. The memory bottleneck is the arrangement of the convolution. We present a novel technique for discarding the portion that cannot contribute to the Minkowski sum boundary.

We implement our algorithm robustly, using our ACP robustness technique [17] to enforce a user-specified backward error bound (Sec. 4). An appendix describes a GPU implementation. We tested both programs on 45 pairs of polyhedra with an error bound of 10^{-8} (Sec. 5). The CPU program outperforms prior work, including non-robust programs. The GPU program exhibits a median speedup factor of 36, which increases to 68 on the 6 hardest tests. For example, it computes a Minkowski sum with a million features in 20 seconds.

2 Prior work

The main approaches to computing Minkowski sums of polyhedra are convex decomposition and convolution.

The first approach decomposes the polyhedra into convex components, computes the Minkowski sums of the components with a specialized algorithm [6], and returns their union. This approach is inefficient because a polyhedron with r reflex edges can have $\Omega(r^2)$ convex pieces, so an input with n edges can entail a union of $\Omega(n^4)$ component Minkowski sums. Hachenberger [9] provides an exact implementation of this algorithm. The program is very slow [4, 2, 18]. Varadhan and Manocha [20] reduce the constant factor by computing the union approximately yet with the correct topology, using a volumetric grid. They observe that the approach remains slow.

The second approach computes a set of facets, called a convolution, that is a superset of the boundary of $A \oplus B$. Each (open 3D) cell c of the arrangement of these facets satisfies $c \subseteq A \oplus B$ if $-A + t \cap B \neq \emptyset$ for an arbitrary $t \in c$ (otherwise $c \cap A \oplus B = \emptyset$), and hence $A \oplus B$ is the closure of the union of these cells.

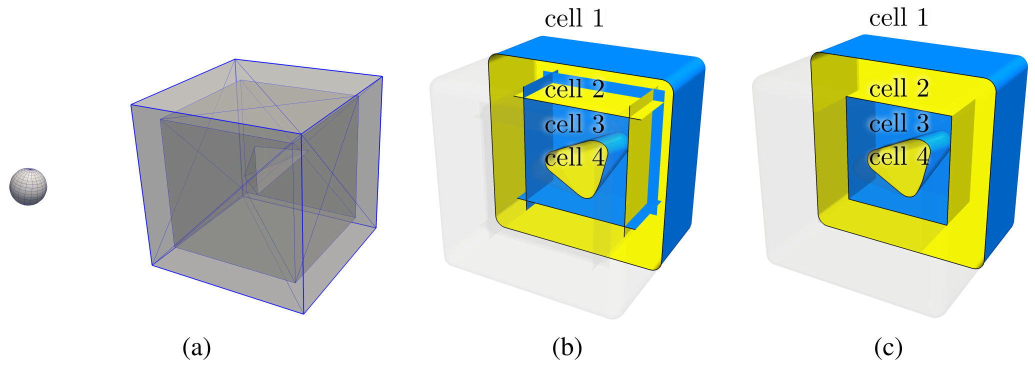

Figure 2: Sphere and hollow box containing tetrahedron (a), convolution (b), and Minkowski sum (c).

The kinetic convolution [3] is defined using an assignment of a set of outward normals to each boundary point of a polyhedron, called a tracing. A facet is assigned its normal n. An edge with incident facet normals n_1 and n_2 is assigned the arc n_1n_2 on the Gaussian sphere. A vertex is assigned the interior of the spherical polygon defined by the arcs of the incident edges. Features from A and B are compatible if they have a common normal. The kinetic convolution is the set of Minkowski sums of compatible pairs. The 2D version of the convolution has a winding number property that determines if cell $c \in A \oplus B$. This property does not hold for general polyhedra [3].

We illustrate these concepts with A a sphere and with B a hollow box that contains a solid tetrahedron (Fig. 2). The arrangement of the convolution has four cells. The sphere $-A + t$ is outside the box for t in cell 1, overlaps the box for t in cell 2, is inside the box and outside the tetrahedron for t in cell 3, and overlaps the tetrahedron for t in cell 4. The Minkowski sum is the closure of the union of cells 2 and 4.

Lien [14] provides a non-robust implementation of a kinetic convolution algorithm. He computes the 2D arrangement of each facet of the convolution defined by the other facets. He groups the faces of these arrangements into polygonal surfaces. He identifies those surfaces that bound the Minkowski sum with intersection tests. There is no error bound and the putative Minkowski sum can self-intersect.

Fogel and Halperin [6] compute the Minkowski sum of two convex polyhedra. The kinetic convolution is the Minkowski sum boundary, so arrangement is trivial. Barki, Dennis, and Dupont [2] compute the Minkowski sum of a general polyhedron and a convex polyhedron. They compute a subset of the kinetic convolution that works for this special case. The rest of the algorithm is similar to Lien's [14]. Both Fogel and Halperin and Barki et al provide exact implementations.

Campen and Kobbelt [4] use the kinetic convolution to compute the outer boundary of the Minkowski sum. The algorithm has limited applicability because inner boundaries are common, e.g. when the polyhedra have inner cavities, in part layout, and in mechanical design. They provide an exact implementation via a prior technique [19] that requires them to use planes as geometric primitives and to define vertices as intersection points of three planes. The vertex accuracy is 10^{-5} even though the plane accuracy is double float. Converting the Minkowski sum to a boundary representation can cause self-intersection.

Kaul and Rossignac [11] define a subset of the kinetic convolution, which we call the convex convolution, that is still a superset of the boundary of $A \oplus B$. The set of normals at a boundary point $p \in A$ is replaced by the subset corresponding to the convex closure of A in a neighborhood of p. Hence, no normals are assigned to concave edges and to some vertices with concave incident edges, and fewer normals are assigned to the other vertices with concave incident edges.

We [18] developed a robust implementation of a convex convolution algorithm. The data structures are incompatible with distributed computation. The memory footprint is larger than the output size. The robustness technique requires custom logic for every type of degenerate input, and the error is not under user control or even bounded.

Li and McMains [13] approximate the outer boundary of the Minkowski sum using the convex convolution, voxelization, and the GPU. The accuracy is limited by the volumetric resolution: the reported results have a resolution of 1024^3, which yields a 10^{-3} error. Increasing the resolution incurs a cubic running time penalty and is limited by the GPU memory size.

Input: polyhedra A and B.

1. Construct convex convolution.
2. Intersect facets.
3. Arrange facets.
4. Compute Minkowski sum boundary.
5. Triangulate Minkowski sum boundary.

Output: triangulated boundary of $A \oplus B$.

Figure 3: Minkowski sum algorithm.

3 Algorithm

Fig. 3 summarizes our Minkowski sum algorithm. The inputs are polyhedra with triangular facets with outward oriented normals. Step 1 constructs the convex convolution. Step 2 intersects its facets. Step 3 computes the arrangements defined by the intersection edges. Step 4 identifies the faces of the arrangements that comprise the Minkowski sum boundary. Step 5 triangulates the boundary facets via monotone decomposition.

Although our algorithm has the same structure as our prior algorithm [18], we employ novel techniques that reduce memory usage and that enable distributed computation. We describe the algorithm briefly and discuss the innovations in detail. We can assume that the inputs are in general position because of our robustness technique (Sec. 4). This assumption simplifies the algorithm and the presentation.

3.1 Step 1

Step 1 of the algorithm constructs the convex convolution of A and B. Its facets are triangles and parallelograms because the facets of A and B are triangles. We identify the pairs of features with shared normals using a binary spatial partition of the Gauss sphere [16]. Although the kinetic convolution can be computed in an output sensitive manner [3], we do not attempt output sensitive computation of the convex convolution because the running time of our algorithm is negligible.

3.2 Step 2

Step 2 of the algorithm constructs the intersection edges of the convolution, which we call FF-edges. One endpoint of each FF-edge is the intersection point of a convolution edge and a facet, which we call an EF-vertex. The other endpoint is also an EF-vertex or is a convolution vertex. In Fig. 4, facet $s = v_1v_2v_3v_4$ intersects facets $\{r, u, v, w\}$ and forms EF-vertices $\{v_5, v_6, v_7, v_8, v_9\}$ and FF-edges $\{v_4v_5, v_5v_6, v_6v_7, v_8v_9\}$.

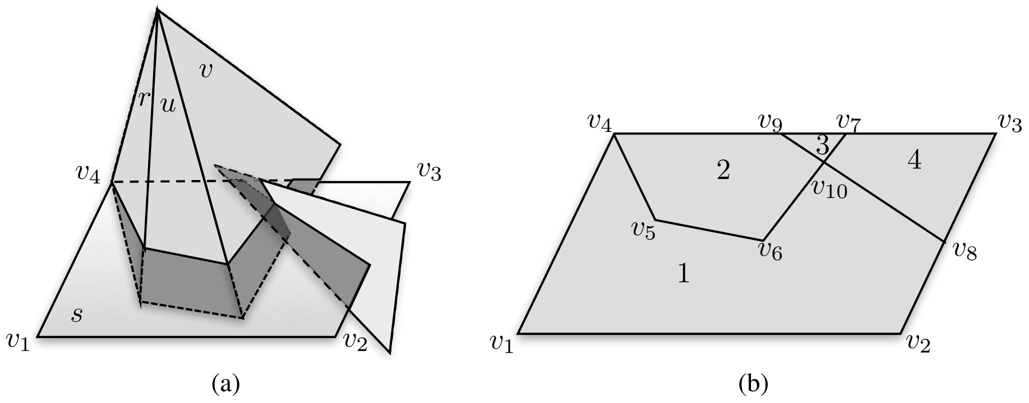

Figure 4: Intersecting facets (a) and arrangement of s (b).

Testing a pair of facets for intersection and if so constructing their FF-edge are constant-time operations. The key to an efficient algorithm is to test as few non-intersecting pairs as possible. Since an output sensitive algorithm is not available, we construct a kd-tree for the facets and test every pair of facets that share a leaf. The drawback of kd-trees is that two facets can share multiple leafs if some splitting planes intersect them both. Prior work uses hash tables to detect duplicate pairs. Hash table construction is slow on the GPU. For example, Alcantara et al [1] construct a hash table with 5 million entries in 35.7ms using 1.42x table storage. We have developed an alternate data structure that can be constructed efficiently on the GPU.

Each facet p is labeled with a bit vector $l(p) = 0$. We construct a kd-tree for the labeled facets. If the splitting plane at depth d intersects p, p is assigned to the left subtree and the right subtree is assigned a copy of p the dth bit of whose label is set to one. If not, p is assigned to one subtree. A leaf is generated if a maximum depth or a minimum number of facets is reached. Fig. 5 shows an example in which facets p and q are inserted into a kd-tree with root n_0, depth-1 nodes n_1 and n_2, and leafs $n_3, \ldots, n_6$. When p is split at $d = 1$ by the dashed vertical line, it is assigned to n_5 and n_6 with labels 10 and 11. In a leaf, facets p and q are tested for intersection if $l(p) \wedge l(q) = 0$ (the bit-wise conjunction). In our example, p and q appear in leafs n_3, n_5, and n_6, but are only tested in n_3 because $l(p) \wedge l(q)$ is 10 in n_5 and is 11 in n_6.

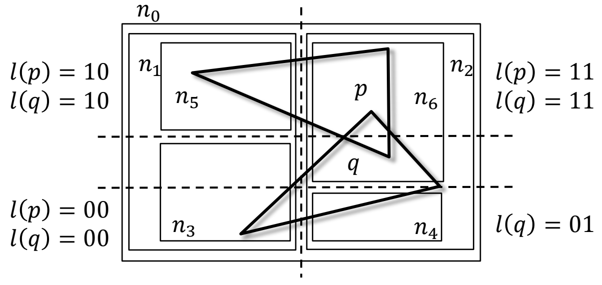

Figure 5: Labeled facets p and q in depth-2 kd-tree.

No pair is tested twice because $l(p) \wedge l(q)$ can equal zero only in the leftmost leaf that contains p and q. Any other leaf that contains p and q has a common ancestor with the leftmost leaf at some depth d. The dth bits of $l(p)$ and $l(q)$ equal one in the other leaf because p and q were assigned to both subtrees at depth d and the other leaf is in the right subtree.

Every intersecting pair is tested because $l(p) \wedge l(q) = 0$ in their leftmost leaf. If not, the leftmost leaf would be in the right subtree of some ancestor and p and q would be assigned to both of its subtrees. Since p and q intersect, no splitting plane separates them, so they would share a leaf in the left subtree. This leaf would be to the left of the leftmost leaf.

3.3 Step 3

Step 3 of the algorithm computes the arrangement of each facet defined by its FF-edges. The arrangement consists of faces bounded by polygonal loops. We split the edges of the facet at their EF-vertices and split its FF-edges at their intersection points, which we call FFF-vertices. We form the loops by traversing the sub-edges. We compute their nesting order, which defines the faces, via ray casting in the plane of the facet. In Fig. 4, FF-edges v_6v_7 and v_8v_9 intersect at FFF-vertex v_{10} and the faces are numbered.

3.4 Step 4

Step 4 of the algorithm identifies the faces of the facet arrangements that bound the Minkowski sum. Only some faces are candidates. Of the four faces that share a sub-edge of an FF-edge, the two faces on the outward normal sides of both facets are candidates. If two or more faces share a sub-edge of a convolution edge, two of them are candidates based on a geometric test [11]. The candidates define polyhedral surfaces. A surface contributes to the Minkowski sum boundary if it is closed and $-A + t$ does not overlap B for an arbitrary vertex t of the surface.

The CPU program assigns the candidate faces to surfaces by breadth-first traversal of the face adjacency graph. This algorithm is suboptimal for the GPU because the running time is proportional to the graph diameter independently of the number of processors. Instead, we compute the surfaces in parallel with the union-find algorithm whose time complexity is nearly proportional to e/p for e edges and p processors. Each face is initialized to a singleton set. In each cycle, each face is assigned a process that merges its set with those of its graph neighbors.

To classify a surface, we find the vertex with maximum z. The surface is an outer boundary if its incident faces wind counterclockwise about the positive z direction; otherwise, it is an inner boundary. We assign an inner boundary to its cell by intersecting a ray through one of its vertices with the outer boundaries and selecting the closest one that is intersected an odd number of times.

3.5 Discarding blocked vertices using data groups

The Minkowski sum boundary is typically a small subset of the arrangement of the convolution. The rest of the arrangement, called the blocked portion, is in the Minkowski sum interior. Storing the blocked portion makes the memory footprint of the program far exceed the output size, which inhibits distributed computation. We discard most blocked vertices and never compute most blocked sub-edges and faces.

The facets of the convolution are oriented so that the polyhedra $-A + t$ and B intersect for t in a neighborhood of a facet on the negative side of its normal. Since this neighborhood is blocked, we can discard the part of the convolution that it contains without losing any part of the Minkowski sum boundary.

We use three tests for being blocked. 1) Let $e = v_0v_k$ have tangent $u = v_k - v_0$ and contain vertices $v_1, \ldots, v_{k-1}$ sorted along e (Fig. 6). The v_i are EF-vertices for a convolution edge and are FFF-vertices for an FF-edge. Each v_i is the intersection point of e with a facet s_i with normal n_i. If $n_i \cdot u > 0$, the intersection of v_0v_i with a neighborhood of v_i is in the interior of $A \oplus B$ due to s_i, so the sub-edge $v_{i-1}v_i$ is blocked. Likewise, v_iv_{i+1} is blocked if $n_i \cdot u < 0$. The only non-blocked sub-edge in our example is v_2v_3. 2) If both incident sub-edges of v_i are blocked, v_i is blocked. 3) If v_i is a blocked EF-vertex, the sub-edge v_iv_f of the incident FF-edge f is blocked.

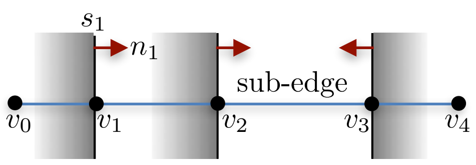

Figure 6: Blocked sub-edges of $e = v_0v_4$.

We reduce the memory footprint by dividing the facets into groups prior to edge intersection and merging the groups after discarding blocked vertices. We form groups of approximately equal size by constructing a kd-tree for the facet centroids. The facets whose centroids share a leaf are the primary members of a group. We ensure that a group contains all the facets that intersect its primary members by including the facets that intersect the bounding box of its primary members. For each group, we construct FF-edges for the primary members that intersect other members, construct FFF-vertices for the intersecting pairs of FF-edges, and discard the blocked vertices. After merging the group data, we form the sub-edges and the faces.

4 Robustness

The goal of robustness is to implement an algorithm in a manner that guarantees an accurate output for every input. The output of a computational geometry program can have combinatorial and geometric components. In our case, these are the combinatorial structure of the Minkowski sum and the coordinates of its vertices. One error metric, called the forward error, is the distance between the output and the exact answer. Euclidean distance is a fine metric for geometric error, but we are unaware of a useful metric for combinatorial error. Moreover, the forward error conflates the quality of the algorithm, which is what we wish to measure, with the condition of the problem, which is beyond our control. Consequently, numerical analyis eschews forward error in favor of backward error: the minimum distance from the input to an alternative input for which the output is the exact answer. The backward error models computational error identically to measurement error in the input. It models combinatorial and geometric error in the same manner.

The mainstream approach to robustness in computational geometry [5] is to compute combinatorial structure exactly. Since the control logic is expressed in terms of predicates, polynomials whose signs are interpreted as truth values, it suffices to evaluate predicates exactly. Exact evaluation is accelerated by first attempting to resolve the sign with error-bounded floating point arithmetic. This approach works poorly for degenerate predicates, that is predicates whose value is zero. The first problem is that exact evaluation of degenerate predicates is slow because floating point evaluation necessarily fails. The impact on overall running time is large because degenerate predicates are common in real-world inputs, due to symmetry, to design constraints, and to convention. The second problem is that degeneracy creates a third branch at every point in the control logic. These branches are typically ignored in theoretical analysis because they contribute nothing fundamental to the algorithm. Yet they must be handled by a robust implementation.

Halperin [10] addresses these problems with a robustness technique, called controlled perturbation (CP), in which a random perturbation in $[-\delta, \delta]$ is added to each input parameter. The algorithm is executed with error-bounded floating point predicate evaluation. If all the predicates are resolved, the combinatorial output is correct for the perturbed input, so the backward error is bounded by δ. Otherwise, the algorithm is rerun, perhaps with a larger δ. The problem with CP is that a large δ is required to resolve a singular predicate (zero value and zero gradient), e.g. 10^{-5} in Minkowski sums [18].

We [17] solve this problem with an extension of CP called adaptive precision controlled perturbation (ACP), that implements the algorithm exactly on the perturbed input, so there are no predicate failures. We evaluate predicates in floating point interval arithmetic. If the interval does not contain zero, the sign is resolved. Otherwise, we re-evaluate in extended precision interval arithmetic, starting with quad-double and doubling the precision until the sign is resolved. We set $\delta = 10^{-8}$ because this error is negligible in applications, yet δ is large enough that there are no degenerate predicates with high probability and that floating point interval arithmetic resolves almost all the predicates. We handle predicates that are degenerate on the perturbed input (a situation that has yet to occur) by aborting evaluation if 848 bit precision is insufficient, doubling this precision threshold, and restarting the Minkowski sum program with a different random seed.

The CPU implementation of the ACP strategy is straightforward. The coordinates of the vertices of A and B are the input parameters, hence are perturbed. The coordinates of the other vertices are defined parameters. These vertices store their coordinates as floating point intervals and also store pointers to their defining vertices. A convolution vertex $v = a \oplus b$ points to a and b. An EF-vertex points to its convolution edge and facet. An FFF-vertex points to its defining facets. These pointers enable us to increase the precision of a vertex by recursively increasing the precision of its defining vertices then recomputing the intervals of its coordinates. Precision is increased solely when a predicate cannot be resolved in floating point and is decreased immediately to avoid storage of extended precision data. The GPU version of ACP is described in the appendix.

5 Results

We tested our programs on nine polyhedra with 30 to 37,000 triangular facets (Fig. 7). We scaled each input to the unit box. We computed the Minkowski sums of all 45 pairs of polyhedra. The 9 duplicated pairs would have many degenerate predicates without input perturbation, since the convolution facets are duplicated, so many predicates are nearly degenerate after perturbation. The other 36 pairs have few nearly degenerate predicates. Fig. 8 shows ten representative Minkowski sums.

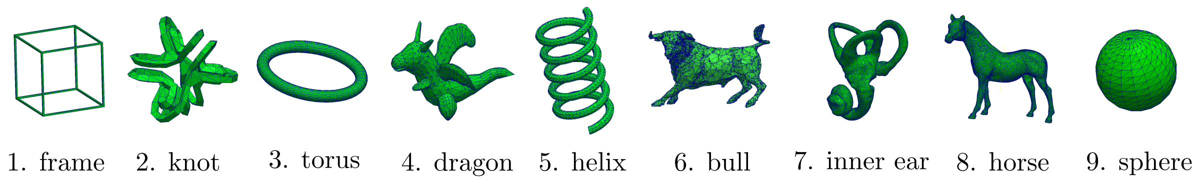

Figure 7: Test polyhedra.

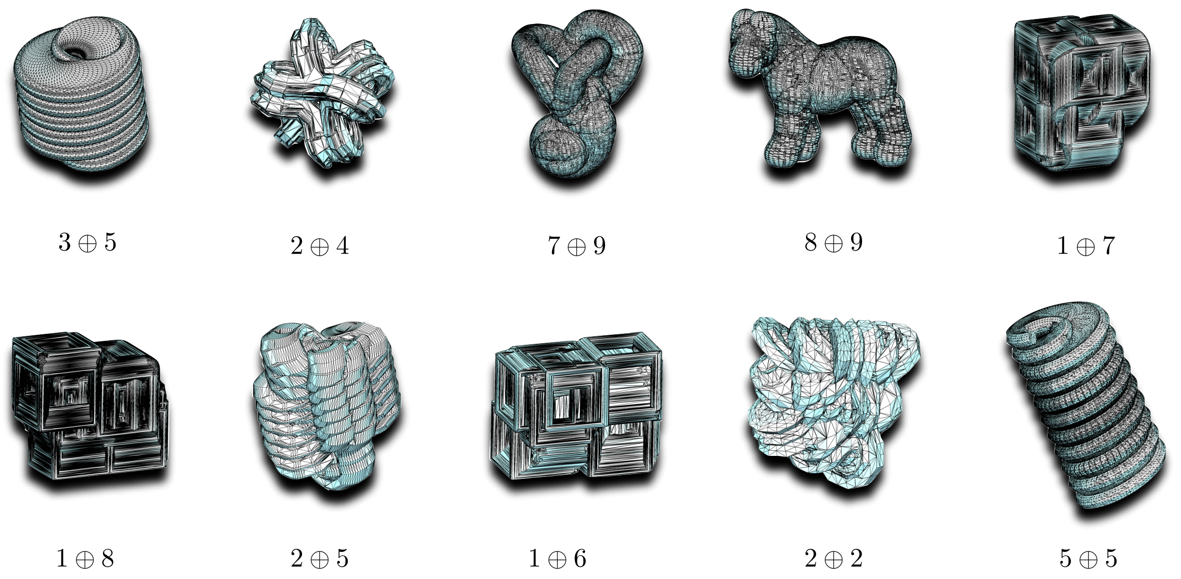

Figure 8: Minkowski sums.

Table 1 summarizes the results. The complexity of a polyhedron, convolution, or arrangement is its total number of vertices, edges, and faces. The complexity of a Minkowski sum also includes the number of cell boundaries and cells. The CPU tests are on one core of an Intel Core 2 Duo. The GPU tests are on a 3.5GHz i7 CPU and a GeForce Titan graphics card with 6GB onboard memory.

The complexity of the convolution is linear in the sum of the complexities of the polyhedra, as $c/(a+b)$ is between 1 and 143 with a median value of 7. The arrangements have the same complexity as the convolution, as s/c is between 1 and 6 with a median value of 2. Discarding blocked elements sharply reduces the complexity of the arrangements: the median reduction is 74% overall and is 93% for the six largest Minkowski sums. There is little room for further reduction the arrangements, as m/c is between 1 and 15 with a median value of 3.

The kinetic convolution is much larger than the convex convolution. The facet ratio is 5–24 on 40 of our 45 pairs. We omitted the 5 largest pairs because we only implemented an n^2 algorithm for generating the kinetic convolution.

The GPU program exhibits a median speedup of 36 relative to the CPU program. The median speedup increases to 68 on the six pairs with the largest Minkowski sums. The running times and memory footprints of these pairs are one to two orders of magnitude larger than those of the other 39 pairs.

Table 2 shows the memory usage and running time for the six largest Minkowski sums with 2, 4, and 8 data groups. A blank entry indicates that the memory footprint exceeds the GPU memory size. The memory savings is sublinear because of the non-primary members, which comprise 10%–20% of each group. Also, some blocked elements cannot be detected until the groups are merged because the relevant tests cross group boundaries. We could reduce the memory footprint by discarding blocked elements after each group is merged into the step 4 input. The running time increases with the number of groups because FF-edges and FFF-vertices involving non-primary members of groups are computed redundantly. We could remove the redundancy by caching the data in a hash map.

5.1 Comparison with prior work

We compare our CPU program to the published running times for prior programs adjusted for processor speed, except that we timed Hachenberger's [9] exact convex decomposition [18]. Hachenberger's program took 2,000–4,000 times longer than our program on small tests, 10,000–100,000 times longer on medium tests, and aborted or did not terminate after several hours on the other tests. Our program is 30 times faster than Varadhan and Manocha's [20] approximate convex decomposition on their hardest test, grate1/grate2.

Table 1: Results: a and b complexities of polyhedra A and B, c complexity of convex convolution, s complexity of arrangement after discarding blocked elements, p percentage discarded, m complexity of Minkowski sum, t running time on GPU in seconds, f speedup factor.

	a	b	c	s	p	m	t	f
$1 \oplus 1$	280	280	2,327	1,852	72	664	0.16	3
$1 \oplus 2$	280	2,960	12,333	48,942	79	36,468	0.22	7
$1 \oplus 3$	280	6,204	9,159	26,897	35	19,918	0.14	4
$1 \oplus 4$	280	6,986	12,618	26,305	38	18,510	0.08	8
$1 \oplus 5$	280	12,038	24,478	149,932	66	97,218	0.18	14
$1 \oplus 6$	280	37,190	93,602	194,593	93	77,380	0.42	23
$1 \oplus 7$	280	96,696	109,090	230,341	89	101,452	0.26	31
$1 \oplus 8$	280	119,084	142,270	302,407	90	126,752	0.31	36
$1 \oplus 9$	280	2,282	2,368	5,255	9	4,904	0.04	2
$2 \oplus 2$	2,960	2,960	110,820	152,354	92	54,498	0.83	33
$2 \oplus 3$	2,960	6,204	64,587	106,173	30	41,846	0.14	18
$2 \oplus 4$	2,960	6,986	119,407	164,963	37	64,184	0.22	36
$2 \oplus 5$	2,960	12,038	230,598	405,937	82	128,324	0.43	54
$2 \oplus 6$	2,960	37,190	845,761	1,527,126	94	296,286	1.63	107
$2 \oplus 7$	2,960	96,696	1,060,917	1,844,852	90	266,340	1.23	106
$2 \oplus 8$	2,960	119,084	1,186,562	2,186,929	87	391,008	1.25	110
$2 \oplus 9$	2,960	2,282	25,252	52,965	17	32,116	0.07	13
$3 \oplus 3$	6,204	6,204	36,676	69,853	83	47,056	0.35	11
$3 \oplus 4$	6,204	6,986	59,510	109,221	21	34,954	0.12	16
$3 \oplus 5$	6,204	12,038	97,992	172,993	41	93,598	0.14	29
$3 \oplus 6$	6,204	37,190	358,663	710,078	77	184,324	0.44	44
$3 \oplus 7$	6,204	96,696	321,871	639,309	70	204,878	0.46	49
$3 \oplus 8$	6,204	119,084	363,718	746,377	68	286,672	0.54	55
$3 \oplus 9$	6,204	2,282	8,982	19,448	0	18,764	0.04	11
$4 \oplus 4$	6,986	6,986	136,535	298,006	74	106,222	0.33	32
$4 \oplus 5$	6,986	12,038	220,356	375,690	24	89,114	0.25	33
$4 \oplus 6$	6,986	37,190	819,442	1,534,669	78	282,494	0.65	55
$4 \oplus 7$	6,986	96,696	956,038	1,874,686	70	315,968	0.80	53
$4 \oplus 8$	6,986	119,084	1,096,359	2,222,506	66	493,322	0.88	58
$4 \oplus 9$	6,986	2,282	24,735	48,980	21	17,422	0.08	12
$5 \oplus 5$	12,038	12,038	480,627	1,571,112	97	1,064,996	3.67	87
$5 \oplus 6$	12,038	37,190	1,394,215	2,502,129	86	311,152	1.13	64
$5 \oplus 7$	12,038	96,696	1,525,204	2,785,053	80	333,962	1.30	79
$5 \oplus 8$	12,038	119,084	1,664,459	3,188,945	77	449,584	1.35	75
$5 \oplus 9$	12,038	2,282	48,320	103,355	14	56,134	0.08	23
$6 \oplus 6$	55,784	55,784	15,947,962	12,172,855	98	1,082,338	19.24	63
$6 \oplus 7$	55,784	145,050	16,734,559	12,904,378	95	887,584	6.42	73
$6 \oplus 8$	55,784	178,625	19,361,581	14,912,546	93	1,547,752	6.63	70
$6 \oplus 9$	37,190	2,282	150,939	250,485	35	124,890	0.20	31
$7 \oplus 7$	145,050	145,050	13,954,664	11,101,744	93	1,454,272	12.01	58
$7 \oplus 8$	145,050	178,625	15,276,998	12,012,517	89	1,259,276	7.75	68
$7 \oplus 9$	96,696	2,282	154,394	292,433	33	190,872	0.23	42
$8 \oplus 8$	178,625	178,625	21,990,951	17,676,700	93	2,306,742	17.87	68
$8 \oplus 9$	119,084	2,282	169,628	320,331	32	207,802	0.27	41
$9 \oplus 9$	2,282	2,282	5,488	11,502	8	9,796	0.06	5

Table 2: Impact of data groups: c_m convolution size in GB, p_k and t_k peak memory usage in GB and running time in seconds for k groups.

	c_m	p_2	p_4	p_8	t_2	t_4	t_8
$6 \oplus 6$	1.07	–	3.42	2.57	–	19.24	23.20
$6 \oplus 7$	1.12	–	2.87	1.73	–	6.42	7.53
$6 \oplus 8$	1.27	–	3.42	2.11	–	6.63	7.40
$7 \oplus 7$	0.97	4.2	2.49	1.55	11.03	12.01	13.64
$7 \oplus 8$	1.05	4.0	2.23	1.23	7.42	7.75	8.45
$8 \oplus 8$	1.42	–	3.98	2.33	–	17.87	19.31

Our program is several times faster than Lien's [14] non-robust convolution algorithm. His running times on his two largest examples, knot/bull and knot/inner ear, are 755 seconds and 921 seconds, versus 174 seconds and 130 seconds for our program. He does not test any duplicated pairs. Our program is over 100 times faster than Barki et al's [2] exact Minkowski sums of a convex polyhedron and a general polyhedron. For example, their running times for knot/sphere and grate2/sphere are 172 seconds and 280 seconds, versus 1 second for our program. We accurately compute the full Minkowski sum six times faster than Campen and Kobbelt [4] compute the outer boundary with low precision. Moreover, their web server[1] returns a "memory or time exceeded" message on our six largest examples and on some others. Finally, our program is twice as fast as our prior program [18].

Our GPU program computes $1 \oplus 6$ and $1 \oplus 7$ about 10 times faster than Li and McMain's GPU program [13], after adjusting for GPU speed. Our error is 10^{-8} and is under our control, while theirs is 10^{-3} and is governed by the GPU memory capacity.

6 Discussion

We have presented a convolution algorithm for Minkowski sums of polyhedra with robust CPU and GPU implementations. The algorithm contains several innovations that support distributed computation. The computational bottleneck is finding the intersecting pairs of facets. We enabled a distributed algorithm by creating a novel type of kd-tree that eliminates duplicate entries without using global memory. The memory bottleneck is the arrangements of the facets of the convolution. We removed this bottleneck by processing the facets in groups and by removing most of the blocked geometry. We solved the robustness problem, which is the primary implementation challenge for computational geometry algorithms, using our ACP strategy. We conclude with plans for future work.

One research direction is to develop a multi-core CPU implementation. The obvious approach is to assign a core to each group in steps 2 and 3, which dominate the running time, but this approach is memory bound for four or more cores. A second research direction is to distribute steps 2 and 3 over multiple GPU's. We can easily assign the groups from our current algorithm to multiple GPU's, but we need to control the cost of data communication.

A third challenge is to compute the swept volume of a polyhedron over a spatial path. The swept volume can be approximated by constructing facets, computing their arrangements, and extracting cell boundaries [4]. The inputs are large because many facets are required for accurate approximation. The algorithms and GPU programs that we developed for Minkowski sums should solve these problems.

Acknowledgments

Kyung is supported by the Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education, Science, and Technology (2012-0001704). Sacks is supported by NSF grant CCF-0904832. Milenkovic is supported by NSF grant CCF-0904707.

[1]http://www.graphics.rwth-aachen.de/webbsp/

References

[1] Daniel Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John Owens, and Nima Amenta. Real-time parallel hashing on the GPU. *ACM Transactions on Graphics*, 28(5):154:1–154:9, 2009.

[2] Hichem Barki, Florence Denis, and Florent Dupont. Contributing vertices-based Minkowski sum of a nonconvex-convex pair of polyhedra. *ACM Transactions on Graphics*, 30(1):3:1–3:16, 2011.

[3] J. Basch, L. J. Guibas, G. D. Ramkumar, and L. Ramshaw. Polyhedral tracings and their convolution. In Jean-Paul Laumond and Mark Overmars, editors, *Algorithms for Robotic Motion and Manipulation*, pages 171–184. A K Peters, 1997.

[4] Marcel Campen and Leif Kobbelt. Polygonal boundary evaluation of Minkowski sums and swept volumes. *Eurographics Symposium on Geometry Processing*, 29(5):1613–1622, 2010.

[5] Exact computational geometry. http://cs.nyu.edu/exact.

[6] Efi Fogel and Dan Halperin. Exact and efficient construction of Minkowski sums of convex polyhedra with applications. *Computer-Aided Design*, 39(11):929–940, 2007.

[7] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple precision binary floating point library with correct rounding. *ACM Transactions on Mathematical Software*, 33:13, 2007.

[8] S. Gottschalk, M. C. Lin, and D. Manocha. Obbtree: A hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH '96*, pages 171–180. ACM, 1996.

[9] Peter Hachenberger. Exact minkowski sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces. *Algorithmica*, 55:329–345, 2009.

[10] Dan Halperin. Controlled perturbation for certified geometric computing with fixed-precision arithmetic. In *ICMS*, pages 92–95, 2010.

[11] Anil Kaul and Jarek Rossignac. Solid-interpolating deformations: Construction and animation of PIPS. *Computers and Graphics*, 16(1):107–115, 1992.

[12] Min-Ho Kyung, Elisha Sacks, and Victor Milenkovic. Robust minkowski sum computation on the GPU. Technical Report 13-001, Purdue University, 20013.

[13] Wei Li and Sara McMains. Voxelized Minkowski sum computation on the GPU with robust culling. *Computer-Aided Design*, 43(10):12701283, 2011.

[14] J.M. Lien. A simple method for computing Minkowski sum boundary in 3d using collision detection. *Algorithmic Foundation of Robotics VIII*, pages 401–415, 2009.

[15] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[16] Victor Milenkovic, Elisha Sacks, and Min-Ho Kyung. Robust Minkowski sums of polyhedra via controlled linear perturbation. In *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pages 23–30. ACM, 2010.

[17] Victor Milenkovic, Elisha Sacks, and Steven Trac. Robust free space computation for curved planar bodies. *IEEE Transactions on Automation Science and Engineering*, 10(4):875–883, 2013.

[18] Elisha Sacks, Victor Milenkovic, and Min-Ho Kyung. Controlled linear perturbation. *Computer-Aided Design*, 43(10):1250–1257, 2011.

[19] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18:305–363, 1997.

[20] Gokul Varadhan and Dinesh Manocha. Accurate Minkowski sum approximation of polyhedral models. *Graphical Models*, 68(4):343–355, 2006.

[21] Kun Zhou, Quiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):126:1–126:11, 2008.

A GPU implementation

We describe the GPU implementation at the requisite level of detail for an experienced GPU programmer to reproduce it.

Data structures A vertex has single float coordinates and is represented by an index into an array V. An edge consists of tail and head vertices, an incident facet, and *emin* and *ccw* values (explained below), and is represented by an index into arrays E_i. A facet consists of vertices, edges, and an outward normal, and is represented by an index into arrays F_i. The data structures are initialized with the polyhedra A and B.

Convolution We use one thread per vertex of A to find the facets of B with which it forms convolution facets, and vice versa. We use one thread per convex edge of A to find the convex edges of B with which it forms facets. We store the feature pairs contiguously in an array *FP* using atomic operations for synchronization. Since these operations are slow, we form groups of $m = 256$ threads, store the output of each group in local shared memory, and flush the output to *FP* using an atomic operation to find the start index. We use one thread per *FP* entry to create an array *VV* of the pairs of indices (a, b) of the vertices $V_a \oplus V_b$ of the facets. We sort *VV* lexicographically, remove the duplicates, and add vertices to V for the remaining pairs. We use one thread per *FP* entry

We create an array with an entry (e, a, b) for every convolution edge e with a and b the smaller and larger of its vertex indices. We sort the array lexicographically by (a, b). Each segment with identical (a, b) entries defines a set of equivalent edges in distinct facets. We set the *emin* fields of the edges to the minimum edge index of the set. We sort the edges in counterclockwise order around $b - a$ and set the *ccw* field of each edge to the index of its successor in this order if the edge is from a to b, and to the index of its predecessor otherwise.

Groups When the set of facets is too large to be processed on the GPU, it is divided into groups, the groups are processed, and the results are merged after removing blocked entities. We place the indices of the facets in an array P, compute their centroids, and place their coordinates in arrays X, Y, and Z. We compute the group ids in an array G. Initially, $G = 0$. We sort the five arrays lexicographically by (g, x) and then compute the x-medians of the groups by enumerating G and X. We split the current groups using thread i to set $g_i \leftarrow 2g_i$ if x_i is less than the median of g_i and $g_i \leftarrow 2g_i + 1$ otherwise. We cycle through the coordinates until we obtain the requested number of groups. We create arrays P_i of the facet indices of the primary members of the groups by scanning P and G. We use one thread per facet not in P_i to test if it intersects the bounding box of group i and if so to add it to P_i.

The next three paragraphs discuss FF-edge creation, FFF-vertex creation, and blocked geometry within a group. The following paragraph explains how the group data is merged.

FF-edges We construct a kd-tree of facets with a prior GPU algorithm [21], but we do not split facets that intersect cutting planes. Leaf node data are stored in an array of leaf facets *LP*, an array of leaf labels *LL*, and an array of bit vectors *LB*. We generate a duplicate-free array *PP* of all *P* pairs. Let N_{st} be an array containing the indices of start entries in *LP* for groups and N_n be an array containing the leaf sizes. We assign $\frac{1}{2}N_n[l](N_n[l] - 1)$ threads for each leaf l to enumerate pairs. Each thread i maps its index i to (a, b), an entry of the upper triangular matrix of size $N_n[l]$, which is then added by $N_{st}[l]$. Then, we insert the facet pair $(LP[a], LP[b])$ to *PP*, if (a, b) passes the bit vector test $LB[a] \wedge LB[b] = 0$ and the oriented bounding box test.

We use one thread per pair to generate an array of all (e, p) pairs with p one facet of a *PP* pair and e an edge of the other facet. We set e to the *emin* of the edge. We sort the array, remove duplicate entries, and assign each entry (e, p) to a thread that checks if e intersects p. If so, an EF-vertex is created by adding (e, p) to an array EF_g and adding the intersection point to V. For each pair in *PP* with two EF_g entries (or one entry and a shared vertex), a thread creates an FF-edge by adding the pair to an array FF_g and updating the E_i.

FFF-vertices We sort FF_g in lexicographic order. For each segment with first index s, we use m threads with a shared buffer to generate the triples (s, a, b) with $a < b$ for all second indices a and b. We use one thread per triple to test if the three facets intersect. If so, an FFF-vertex is created by adding the triple to an array FFF_g and adding the intersection point to V.

Blocked geometry We form an array L of pairs (e, v) with e an edge and v a vertex. An EF-vertex forms one pair with its convolution edge. An FFF-vertex forms three pairs with its three FF-edges. We use one thread per pair to compute the projection of v along e, $d = v \cdot (h - t)$ where t and h are the end points of e. We sort L in d order. We use one thread per pair to compute an array S of the signs of $n \cdot (h - t)$ where n is the normal of the facet that e intersects at v.

We compute an array B in which $b_i = 1$ if v_i is blocked. Initially, $B = 0$. One thread per L pair sets $b_i = 1$ if $e_i = e_{i+1}$ and $s_i = s_{i+1} = 1$ or if $e_i = e_{i-1}$ and $s_i = s_{i-1} = -1$. Next, one thread per L pair sets $b_i = 1$ if $e_i = e_{i+1}$, $s_i = 1$, and $b_{i+1} = 1$ or if $e_i = e_{i-1}$, $s_i = -1$, and $b_{i-1} = 1$. A blocked FFF-vertex is deleted. An FF-edge with no FFF-vertices is deleted if one of its vertices is blocked. A blocked EF-vertex is deleted unless it is incident to an FF-edge, as it may be needed to recompute a FFF-vertex, as explained in the ACP paragraph.

Merging Let FF_g, EF_g, and FFF_g be the output of group g. We append each EF_g to the global *EF* and add its start index to the EF-vertex indices in FF_g. We append each FF_g to the global *FF* and add its start index to the FF-edge indices in FFF_g. We append each FFF_g to the global *FFF*. We sort *FFF* and remove duplicates, sort *FF*, remove duplicates, and update *FFF*. We sort *EF*, remove duplicates, and update *FF*. We update the indices with GPU prefix sum and sorting operations.

Sub-edges We use one thread per vertex to create the sub-edges of the *emin* edges. For $e = v_0 v_k$ with remaining vertices $v_1, \ldots, v_{k-1}$, $v_i v_{i+1}$ is a sub-edge if $s_i > 0$ and $s_{i+1} < 0$. We use one thread per *emin* sub-edge to create equivalent sub-edges for the rest of its edge set, and to set the *emin* and *ccw* fields.

We set the next field of each sub-edge $v_i v_j$ to the next sub-edge $v_j v_k$ in counterclockwise order around the normal of the facet of $v_i v_j$. If this sub-edge is blocked (so it was not created), next is set to null. We form an array T of triples (p, f, e) for every sub-edge $e = th$ incident to facet f and for $p = t, h$. We sort T by (p, f), which segments it into groups consisting of the sub-edges of a face that are incident to a vertex. We use one thread per group to set the next fields. The computation depends on whether p is a vertex of f, a boundary EF-vertex, an interior EF-vertex, or an FFF-vertex (Fig. 4b). The groups for each of these four cases need to be adjacent in T to prevent execution flow divergence among adjacent threads. We negate f if p is an EF-vertex on its boundary. Sorting T places these vertices first, the facet vertices next, the interior EF-vertices next, and the FFF-vertices last.

Faces We use a thread to generate the sub-edge loop of a given sub-edge by following next fields until it returns to that sub-edge or encounters a null next field. In the first case, the loop id of every sub-edge in the loop is set to its smallest sub-edge index and this id is recorded in an array H. Using one thread per sub-edge would generate every loop once for each of its sub-edges. We reduce the redundancy by repeatedly running one thread for every fifth sub-edge whose loop id is null until none is found. In practice, the first iteration finds most of the loops. Even if multiple threads traces the same loop, synchronization is unnecessary, because they set the same loop id for the edges in the loop.

We sort H, remove duplicates, and use one thread per loop id to classify the loops as outer or inner boundaries. For v_j the vertex with the largest z component, a loop is an outer boundary if the sub-edges $v_i v_j$ and $v_j v_k$ form a convex angle. We generate a face for each outer boundary. We assign each inner boundary to its face by intersecting a ray through one of its vertices with the outer boundaries and selecting the closest one. We choose a pair of two vertices, each from inner- and outer-boundaries not obscured by any edges, and create new edges connecting them to make a single boundary of the face.

Cell boundaries We implement the union-find algorithm for generating the cell boundaries using arrays P and M with one entry per face including unsubdivided facets. The P entry of a face is its parent face in the tree that represents its set. The M entry is true if this set might be a closed surface. Initially, every $P[i]$ is set to i so that every face is in a singleton set, and M is true. We use one thread per face to merge its set with the sets of the adjacent faces along its edges. The adjacent face along an edge e is the incident face of $e.ccw$. If no face is adjacent along an edge, the M entry of the root of the f set is set to false. Merging is done in the standard manner and the M entry of the union is set to the conjunction of the M entries of the merged sets. We iterate this process until every pair of adjacent faces is in the same set. A synchronization error can occur if two threads try to set the same entry, but it is corrected by the next iteration. To lower the chance of synchronization errors, we merge the set with a larger root id to the other set with a smaller root id. In practice, the first iteration finds all the boundaries with no errors.

Minkowski sum As described in step 4 of Sec. 3, we classify the boundaries as outer and inner, form cells, assign inner boundaries to cells, and identify the free cells using our GPU version of a prior collision detection algorithm [8].

ACP In the CPU version of ACP, a predicate throws an exception if it is ambiguous at the current precision. A controller catches the exception, increases the precision, and retries the predicate. This approach is impractical on the GPU. Moreover, the ACP library uses an initial precision of double float, whereas single float is much faster on the GPU. The GPU version of ACP evaluates a predicate on a sequence of arguments, using stages instead of exceptions to increase precision. Stage 1 evaluates in single float, records the signs for the unambiguous elements in the sequence, and forms a second sequence from the ambiguous elements. Stage 2 increases the precision of the second sequence to double float and reevaluates the predicate in double float. Stage 3 evaluates the third sequence in quad-double [15]. In the extremely rare cases where ambiguous predicates remain, the GPU passes them to the CPU for evaluation using MPFR [7].

Pseudo-code We conclude the description of the GPU algorithm with selected pseudo-code. Please email the first author, Kyung, for further details.

```
procedure MinkowskiSum($A$, $B$, $MS$)
Input
    $A$, $B$: input models in triangular meshes
Output
    $MS$: Minkowski sum boundaries
begin
    1. Construct convolution of $A$ and $B$. The convolution result is stored in
            $F_i$: a global array of facets in 4D vectors containing vertex IDs
            $E_i$: a global array of edges in $(t, h, \mathit{next}, \mathit{ccw})$
            $VV$: a global array of vertex positions
    2. Call DoGroupFacets($P_0$,..,$P_{m-1}$)
    3. for group $g = 0, .., m-1$ do
        (a) Construct a $kd$-tree of facets $P_g$ and output the leaf nodes in an array of leaf facets LP, an
            array of leaf labels LL, and an array of bit vectors LB.
        (b) Call EnumerateFacePairsWithFiltering(LP, LL, LB, PP)
        (c) Call PairwiseFacetIntersection(PP, $EF_g$, $FF_g$)
        (d) Call CreateFFFVertices($FF_g$, $FFF_g$)
        (e) Call RemoveBlockedVertices($EF_g$, $FF_g$, $FFF_g$)
       end
    4. Merge $EF_g$'s, $FF_g$'s, and $FFF_g$'s to EF, FF, and FFF
    5. Construct sub-edges by subdividing edges in $E_i$ and $FF_g$ with vertices in $EF_g$ and $FFF_g$
    6. Construct faces by tracing sub-edges, whose start vertex ids are stored in $H$
    7. Append first edge ids of unsubdivided facets of $F_i$ to $H$
    8. Call ConstructCellBoundaries($H$, $P$)
    9. For all $k$, delete $H[k]$ if boundary $P[k]$ is determined to be blocked by collision test
   10. Construct MS by tracing edges from the remaining edges in $H$
end
```

```
procedure DoGroupFacets($P_0$,..,$P_{m-1}$)
Output
    $P_0$.., $P_{m-1}$: facet groups
begin
    Initialize $P$ with a sequence $0, .., |F_i| - 1$, and $G$ with 0
    Place the coordinates of facet centroids in arrays $X$, $Y$, and $Z$.
    for $k = 0, .., \lfloor \log m \rfloor$ do
        Let $C$ be one of $X$, $Y$, $Z$ in cyclic order
        Sort $P$, $G$, $X$, $Y$, and $Z$ lexicographically by $G$ and $C$.
        Construct $M$ containing the median of $C$ for each facet group distinguished by $G$
        for $i = 0, .., |P| - 1$ in parallel
            if $C[i] < M[G[i]]$ then
                $G[i] \leftarrow 2 * G[i]$
            else
                $G[i] \leftarrow 2 * G[i] + 1$
            end
        end
    end
    Sort $P$ by $G$
    Copy each facet group in $P$ to $P_0, .., P_{m-1}$
end
```

```
procedure EnumerateFacePairsWithFiltering(LP, LL, LB, PP)
Input
    LP: an array of leaf facet ids
    LL: an array of leaf labels
    LB: an array of bit vectors
Output
    PP: an array of facet id pairs
begin
    Construct $N_{st}$ and $N_n$ from $LL$, arrays of start entries and numbers of leaf facets, respectively.
    Generate a work item array $W$ in which $N_n[l](N_n[l] - 1)/2$ entries assigned for leaf $l$
       are set to the number $l$.
    for $i = 0..(|W| - 1)$ in parallel
        $g \leftarrow W[i]$
        $k \leftarrow i -$ the first entry of work items for leaf $l$
        Find the largest integer $r$ satisfying $r(r + 1)/2 \leq k$.
        $(a, b) \leftarrow (k - r(r + 1)/2,\ r + 1) + N_{st}[l]$.
        if LB$[a] \wedge$ LB$[b] = 0$, then
            if OBB(LP$[a]$) $\cap$ OBB(LP$[b]$), then
                Insert (LP$[a]$, LP$[b]$) to PP.
    end
end
```

```
procedure PairwiseFacetIntersection(PP, EF, FF)
Input
    PP: an array of facet id pairs
Output
    EF: an EF-vertex array
    FF: an FF-edge array
begin
   Generate EF, an array of all unique edge/facet id pairs made from facet pairs in PP.
   Initialize Done with 0.
   Repeat the for-loop in the single-, double-, quad double- (GPU),
           and higher-precision interval arithmetic(CPU), respectively.
       for $i = 0..(|EF| - 1)$ in parallel
           if $Done[i] = 0$ then
               if $EF[i].e$ intersects $EF[i].f$ then
                   Insert $EF[i]$ to EF
               else if uncertain then return
               $Done[i] \leftarrow 1$.
       end
   for $i = 0..(|PP| - 1)$ in parallel
       if two facets in $PP[i]$ share two EF-vertices (or a EF- and a facet vertices) then
           Insert $PP[i]$ to FF
           Create a new edge of $PP[i]$ to $E_i$
end
```

```
procedure CreateFFFVertices($FF_g$, $FFF_g$)
Input
    $FF_g$: an array of FF-edges in two facet ids
Output
    $FFF_g$: an array of FFF-vertices in three facet ids
begin
   Sort $FF_g$ in lexicographic order
   Enumerate FF-edge pairs with the same first facet id in $FF_g$ to an array $P$.
   Initialize Done with 0.
   Repeat the for-loop in the single-, double-, quad double- (GPU),
           and higher-precision interval arithmetic(CPU), respectively.
       for $i = 0..|FF_g| - 1$ in parallel
           if $Done[i] = 0$ then
               if edge $P[i].a$ intersects edge $P[i].b$ then
                   Form a facet triple from $P[i]$ and insert it to $FFF_g$
                   Insert the intersection point to $V$
               else if uncertain then return
               $Done[i] \leftarrow 1$.
           end
end
```

```
procedure RemoveBlockedVertices(EF$_g$, FF$_g$, FFF$_g$)
begin
    Form $L$, an array of $(e, v)$ with $e$ and edge and $v$ an EF- or FFF-vertex on $e$
    Create $D$, $S$, and $B$, arrays of size $L$
    for $i = 0, .., |L| - 1$ do
        $D[i] = L[i].v \cdot (L[i].e.h - L[i].e.t)$      Sort $L$ lexicographically by $(L[i].e, D[i])$
    for $i = 0, .., |L| - 1$ do
        $n \leftarrow$ the normal of the facet that $L[i].e$ intersects at $L[i].v$
        $S[i] = \text{sign}(n \cdot (L[i].e.h - L[i].e.t))$
    end
    The entries of $B$ are set to false
    for $i = 1, .., |L| - 2$ do
        if $(L[i].e = L[i+1].e) \wedge (S[i] = S[i+1] = 1)$ then
            $B[i] = 1$
        if $(L[i].e = L[i-1].e) \wedge (S[i] = S[i-1] = -1)$ then
            $B[i] = 1$
    end
    for $i = 1, .., |L| - 2$ do
        if $(L[i].e = L[i+1].e) \wedge (S[i] = 1) \wedge (B[i+1] = 1)$ then
            $B[i] = 1$
        if $(L[i].e = L[i-1].e) \wedge (S[i] = -1) \wedge (B[i-1] = 1)$ then
            $B[i] = 1$
    end
    Remove FFF-vertex with id $L[i].v$ from FFF$_g$ if $B[i] = 1$
    Remove FF-edges with no FFF-vertices from FF$_g$, if they have a blocked end vertex
    Remove EF-vertices with no $FF$-edges from EF$_g$
end
```

```
procedure ConstructCellBoundaries($H$, $P$)
Input
     $H$: an array of start edge ids of face loops
Output
     $P$: an array of cell ids
begin
    Initialize $P$ with a sequence $0, .., |H| - 1$
    Repeat the for-loop until $P$ is not updated any more
        for $i = 0..(|H| - 1)$ in parallel
            $e_0 \leftarrow H[i]$, $e \leftarrow e_0$
            do
                $ne \leftarrow e.ccw$
                if $ne = 0$ or $ne$ is not opposite to $e$ then
                    $P[i] = -1$
                else
                    $r \leftarrow$FindRootWithPathCompression($P$, $i$)
                    $r' \leftarrow$FindRootWithPathCompression($P$, $ne.inface$)
                    if $r < r'$ then $P[r'] \leftarrow r$
                    else if $r > r'$ then $P[r] \leftarrow r'$
                end
                $e \leftarrow e.next$
            while $(e \neq e_0)$
        end
    For all $i$, delete $H[i]$ and $P[i]$ if $P[i] = -1$ or $P[P[i]] = -1$
end
```