# OBJECT COLLISION DETECTION VIA THE GJK AND BVH ALGORITHMS

*Mihai Zorca, Berke Egeli, Chris Müller, Liam van der Poel*

Department of Computer Science
ETH Zurich, Switzerland

## ABSTRACT

We present optimized implementations of two 3D collision detection algorithms: The Gilbert-Johnson-Keerthi Algorithm (GJK) and Bounding Volume Hierarchies (BVH). Previous papers consider only the speedup from simplifying GJK to perform intersection tests on convex shapes. Our contribution is to use SIMD vectorizations and other optimizations to achieve a speedup of 13.5×. BVH extends collision detection to general shapes. Our choice of k-DOP Bounding Volume (BV) quad trees allows efficient SIMD parallel intersection tests. Combined with half-precision floats we increased performance by up to 8.4× compared to baseline.

## 1. INTRODUCTION

**Motivation.** In computer graphics, video game design and related fields it is an important problem to quickly know if two objects intersect. For example, almost all video games require knowing when the player collides with an obstacle in their environment. This occurs very frequently and realistic interactions with obstacles are critical to immersive gameplay.

**Contribution.** In this paper we present optimized implementations of two collision detection algorithms: Gilbert-Johnson-Keerthi (GJK) and Bounding volume hierarchy (BVH).

**Related work.** The original GJK algorithm [1] computes the minimum distance between two convex polytopes. A simplified version described by Muratori [2] only tests if two objects intersect or not. This version is known as Boolean GJK. The major contribution is the *DoSimplex* method which simplifies the algorithm given we do not need to explicitly compute the separation distance. Linahan [3] provides an overview of the evolution of GJK, a geometric interpretation of the problem and proves that the optimizations proposed by Muratori are valid. However none of the previously mentioned papers consider the use of SIMD vectorisations and other optimizations to further increase performance.

Bounding Volume Hierarchies have been used in many fields of computer science to speed up collision detection [4]. Choosing a certain Bounding Volume means trading off tightness of fit and computation time for the intersection test. Many different types of BVs have been tried: spheres [5], AABBs [6], kDOPs [7], OBBs [8] and even convex hull trees [9].

The type of BV chosen for the tree and the number of children of each inner node also influence SIMD optimization potential. In particular, the choice from [4] of k-DOP BV quad trees allows efficient SIMD parallel intersection tests [4].

## 2. BACKGROUND INFORMATION

This section presents the theoretical background on the two algorithms we implemented during this course: The Gilbert-Johnson-Keerthi Algorithm (GJK) and Bounding Volume Hierarchies (BVH).

### 2.1. The Gilbert-Johnson-Keerthi (GJK) Algorithm

GJK determines if two convex object $A, B$ intersect. First a few mathematical definitions are needed: An Object is a non-empty set of points in 3d Euclidean space. The Minkowski difference $A - B$ of two objects is the object given by subtracting all points in $B$ from all points in $A$: $A - B := \{a - b : a \in A, b \in B\}$. An object $A$ is convex if and only if $(1 - \lambda)x + \lambda y \in A$ for all $x, y \in A$ and $\lambda \in [0, 1]$. A *simplex* is generalization of the notion of a triangle to $n$ dimensions.

The algorithm uses the Theorem that $A, B$ intersect if and only if their Minkowski difference contains the origin [10]. The pseudocode is given in Algorithm 1.

This algorithm refers to the following functions: Support(A, D) which returns the furthest point on $A$ in direction $D$, and DoSimplex(S, D) which updates the simplex $S$, search direction $D$ and returns whether or not the updated simplex can contain the origin. The simplex $S$ is contained within our Minkowski sum $A - B$, so $S$ contains the origin only if $A - B$ does.

**Algorithm 1:** GJK Pseudocode

```
Input: Object A, Object B, vector D
vector P = Support(A, D) - Support(B, -D)
simplex s = P
vector D = -P
while True do
    vector P = Support(A, D) - Support(B, -D)
    if dot(A, D) < 0: then
        return False        // No intersection
    s += P
    if DoSimplex(S, D): then
        return True         // Intersection
```

### 2.2. Bounding Volume Hierarchies (BVH)

A limitation of GJK is that it can only be applied when both objects are convex. More general 3d objects are often represented by a mesh of geometric primitives, such as triangles, and are not necessarily convex. Bounding Volume Hierarchies (BVHs) are a data structure that allows efficient intersection tests in this more general case [4]. Instead of doing intersection tests betwen all primitives of one object and all primitives of another, we recursively wrap the primitives with simple Bounding Volumes (BVs) such as rectangles, spheres or Discrete Oriented Polytopes (k-DOP). This results in a tree structure with bounding volumes as inner nodes, and the geometric primitives as the leaves.

Our Bounding Volume is the k-DOP, and is defined using a fix set $\{d_1, \ldots, d_{k/2}\}$ of $k/2$ *direction vectors* in $\mathbb{R}^3$. Any k-DOP is then represented using a vector $a \in \mathbb{R}^k$. Concretely, it is the set $\{x \in \mathbb{R}^3 \mid \forall_{i \in [k/2]} : a_i \leq d_i^\top x \leq a_{2i}\}$.

Two k-DOPs (represented by the vectors $u, v \in \mathbb{R}^k$) **intersect** if and only if there does **not** exist an index $i$ such that $u_i < v_{2i}$ or $v_i < u_{2i}$.

To compute intersection of two BV trees, we traverse both trees and check if any volumes intersect. During traversal, if some parent volumes do not collide, their child volumes do not have to be tested. See Algorithm 2 for the high-level traversal pseudocode [4].

**Algorithm 2:** BVHintersect(BVHNode a, BVHNode b)

```
if a and b are both leaves then
    testTriangles(a, b)
else if a is leaf then
    for each child b_i of b do
        if a and b_i intersect then
            BVHintersect(a, b_i)
else if b is leaf then
    for each child a_i of a do
        if a_i and b intersect then
            BVHintersect(a_i, b)
else
    for each child a_i of a and child b_j of b do
        if a_i and b_j intersect then
            BVHintersect(a_i, b_j)
```

### 2.3. Cost Analysis

Our cost measure is the number of additions, multiplications, and comparisons (and float conversions in the quantized BVH). We assume the compiler converts additions and multiplications into FMAs wherever it can. For both GJK and BVH the flop count is highly dependent on the topology of the two shapes we are comparing. We therefore have a flop counter that is dynamically incremented with the number of operations in each branch of the code. The cycle count is measured using the code from the assignments.

We defined the cost measure of GJK as the following:

$$\begin{aligned} C_{GJK}(n, m) &= (C_{mul}, C_{fma}, C_{cmp})(n, m) \\ &= n(m+1)(C_{mul} + 2C_{fma} + C_{cmp}) \end{aligned}$$

where $n$ is the total number of points of both of the input objects and $m$ is the total number of iterations the GJK algorithm performs.

In order to ensure the convergence of the algorithm, we have limited the number of iterations to 100 iterations. In each iteration of the algorithm, we traverse over the vertices of the input objects. Thus, GJK has the worst case complexity of $O(n)$.

We defined the cost measure of BVH as the following:

$$\begin{aligned} & C_{BVH}(n_{ii}, n_{li}, t) \\ &= (C_{mul}, C_{fma}, C_{cmp}, C_{cvt})(n_{ii}, n_{li}, t) \\ &\leq 512 n_{ii} C_{cmp} + 128 n_{li} C_{cmp} + 50t(aC_{mul} + bC_{fma} + cC_{cmp}) + 16(n_{ii} + n_{li}) C_{cvt} \end{aligned}$$

where $n_{ii}$ is the no. of inner-inner tests, $n_{li}$ is the no. of leaf-inner tests, and $t$ is the no. of triangle-triangle tests the BVH algorithm performs. Finally, $a, b, c$ are constants s.t. $a+b+c = 1$ and $C_{cvt} = 0$ for non-quantized code. In practice, however, we measured the actual number of flops using counters in the code instead of relying on this model.

Asymptotically, the BVH trees are shallow, with a height expected in $O(\log n)$ (w.r.t. $n$, the number of triangles in the original mesh). However, in the worst possible case, no triangles intersect but all BVs overlap, so all nodes of the tree must be tested. As each tree has less than $2n$ nodes and each node is tested against at most 4 other nodes, the worst case traversal complexity is in $O(n)$.

## 3. GJK & BVH IMPLEMENTATION

This section explains the baseline implementation and optimization details for our two algorithms.

### 3.1. GJK Baseline Implementation

Baseline GJK algorithm takes two 3D objects as inputs. An object is represented by a struct that stores an $N \times 3$ matrix (where $N$ is the number of vertices of the object). We refer to this as the Array of Structures (AoS) format. This is the most intuitive and common way that shapes represent their data.

The baseline implementation of GJK mainly consists of multiple Support functions and the DoSimplex function.

**Support Function.** A Support function takes an object and a search direction as input. It returns the vertex on the object that lies farthest away in the search direction. To do this, we iterate over every vertex of the object and calculate the dot product between each vertex and the search direction. We then take the argmax of these values.

**DoSimplex Function.** The DoSimplex function takes a simplex and a search direction as input. Based on the number of points in the simplex we apply the corresponding plane tests for a line, triangle or tetrahedron. These tests update the search direction and the simplex. We detect a collision when we have a tetrahedron simplex and it contains the origin. Otherwise we either continue the search, or return that we will never contain the origin.

**Profiling.** We used DTrace tool to profile the baseline implementation. Profiling showed that GJK spends between 65% and 98% of the total execution inside Support functions. Execution time of the Support functions scales with input size, for the DoSimplex function it remains constant.

**Locality.** The GJK algorithm benefits from spatial locality. GJK algorithm also exhibits temporal locality because we access the same set of points in each iteration. However, depending on the size of the input objects, the algorithm can not properly exploit this locality. If the objects do not fit in the cache, they start evicting each other. Thus, they need to be brought back from main memory to caches. Unfortunately, due to the iterative nature of the algorithm, it is not possible to reorder or parallelize the support calls in a way that avoids capacity misses.

**Bandwidth/Data Transfer Analysis.** The bandwidth/data trasfer analysis showed that baseline GJK is compute bound for all of the input sizes used in the experiments.

The analysis showed us that the main bottleneck of GJK are the Support functions, so this was where we focused the optimization effort. Cache optimizations are not feasible due to iterative nature of the algorithm. Finally, GJK can benefit greatly from SIMD intrinsics because it is compute bound and the Support functions have a greatly parallelizable nature.

### 3.2. GJK Standard C Optimizations

**Block Optimizations.** Inside the support function, the computations in the main loop are independent from previous iterations. This allows basic block optimizations. We applied scalar replacement, loop unrolling and accumulators to exploit instruction level parallelism (ILP) in the loop. We tested the algorithm with two different loop unrolls, 2 and 8.

**Method Inlining and Joint Loop Optimization.** Inside the main loop of the GJK algorithm, we sequentially call a support function on each input object. Let the inputs have $N$ and $M$ vertices respectively. This gives $N + M$ iterations in the support functions. We can reduce the total number of iterations by manually inlining the support calls inside GJK and joining the two loops from the support functions. This trick splits the loops into three parts. First we iterate over both objects for $min\{N, M\}$ iterations. Second we complete the iterations for the first object. Third we complete the iterations for the second object. The first loop fully iterates over the smaller object, so we only enter the second or the third loop. This inlining reduces the total number of iterations to $max\{N, M\}$.

### 3.3. GJK SIMD Optimizations

**Vectorization with AVX2 Intrinsics.** Vectorization allows us to do the dot product and argmax operations in the support functions in parallel. First load the object vertices into 128-bit vectors. Then append two 128-bit vectors into a 256-bit vector. Calculate two dot products in parallel using the *_mm256_dp_ps* intrinsic. Store the results of 8 dot products in a single 256-bit vector. Finally, apply one argmax operation to get the maximum dot product. The $i$'th slot of the result vector contains the maximum dot product of all dot products in the $i$'th slot of the input vector. Likewise we vectorize the branching in the argmin operation. Finally, outside of the loop we apply an argmax over the maximum dot product vector.

AoS format and 3D points limit our ability to vectorize. We did not use AVX/AVX512 intrinsics for the main computations as while they would let us vectorize the multiplication in the dot product, we would still have to extract the values inside the vector to do the

scalar addition of three floats. This overhead negated any speedup from the vectorization. Due to AoS format, we also had to load each vertex to 128-bit vectors because we could only separately calculate the dot product of 4 consecutive floating point numbers in the 256-bit vectors.

**Conversion to Structure of Arrays (SoA) Format and Vectorization with AVX512 Intrinsics.** Vectorization with AoS format did not provide significant speedup because of the limitations outlined in the previous section. So we changed the representation of our vertices to SoA format which is more suitable for vectorization. We could now directly use AVX512 intrinsics without having to shuffle or combine data. In SoA format, each coordinate of a vertex is stored in three different arrays. We load each coordinate into three 512-bit vectors separately. We calculate the dot product as shown in Algorithm 3. Then we calculate the argmax as in the AVX2 version. This implementation provides us significant speedup because we are able to calculate 16 dot products in parallel without shuffling the data.

**Strength Reduction in Index Calculation.** We achieved substantial speedup by applying strength reduction to the index calculation. In our initial AVX512 implementation, we were using _mm512_set_epi32 to calculate the indices for argmax operation. Taking this operation outside of the main loop and replacing it with _mm512_add_epi32 inside the loop allowed us to achieve our maximum speedup.

**Algorithm 3:** AVX512 dot product

**Input:** vector X, vector Y, vector Z, vector D[0], vector D[1], vector D[2]

```
1 tmp_0 = _mm512_mul_ps(X, D[0])
2 tmp_1 = _mm512_fmadd_ps(D[1], Y, tmp_0)
3 dp = _mm512_fmadd_ps(D[2], Z, tmp_1)
```

### 3.4. BVH Baseline Implementation

Baseline BVH algorithm takes two 3D objects as input. These objects are represented as an AoS triangle mesh: a struct that stores an array of triangles. Triangles in turn are just arrays of 3D points. AoS format is chosen for the same reasons as mentioned in 3.1.

For both input objects, first the BVH tree has to be constructed. For this tree construction we follow the SIMDop paper [4] and use the *Batch Neural Gas* algorithm described in [11]. Tree construction can be regarded as a "pre-processing" step, as it only has to be performed once, independent of the number of scene objects / performed collision tests. For this reason, we focus our optimization efforts not on this tree construction, but rather on the subsequent tree traversal.

Following Algorithm 2, our intersection test is a recursive procedure that takes two BVH tree nodes as input. Each BVH node is either an inner node, in which case it stores the k-DOP bounding volumes of its 4 children. Or it is a leaf, in which case it is simply a tagged pointer to a triangle array (of at least 1 and at most 4 triangles).

Its three main components are *triangle-triangle* tests, *leaf node - inner node* tests and *inner node - inner node* tests.

**Inner-Inner Tests.** This is the core case, when an inner node $n_a$ from tree $a$ overlaps with an inner node $n_b$ from tree $b$. In this case, we have to test for each child of $n_a$, if its k-DOP bounding volume overlaps with any k-DOP bounding a child of $n_b$. As each inner node has 4 children, this results in 16 k-DOP overlap tests. Recalling section 2.2, testing k-DOP overlap is a series of at most $k$ floating point compares. However, if we find an index $i$ then we can stop early and conclude disjointness. This becomes important later, as vectorized implementations lack this "early-stopping" ability and always perform all $k$ comparisons.

**Leaf-Inner Tests.** This happens when we reach a leaf in one tree, but still consider inner nodes in the other tree. The only difference to the *inner-inner* case is that we test one k-DOP (bounding the leaf) vs. 4 k-DOPs of the children of the inner node.

**Leaf-Leaf Tests.** Here, we have reached leaves in both trees, so we can directly test the triangles both leaves point to using the standard Möller triangle-triangle test [12]. As leaves point to between 1 and 4 triangles, up to 16 such tests have to be performed.

**Note:** Our attempt to vectorize the Triangle-Triangle case was not successful. Möller's algorithm is very unsuitable to vectorization, as it is mostly comprised of case distinctions. This, together with the profiling results, made us focus our optimization efforts on traversal and the Leaf-Inner and Inner-Inner Tests.

**Profiling.** We implemented various counters in the BVH code, which allowed us to profile the implementation. Profiling showed that BVH spends between 70% and 95% on tree traversal. The remaining time is spent in the leaf triangle-triangle tests.

**Locality.** As can be expected for a tree traversal algorithm (read: chasing pointers), locality is poor. After a node has been fetched and one of the three test cases is applied, this node will never be tested again. Therefore, almost no temporal locality is available. The only exception to this is the leaf-inner case: the leaf node of one tree can be tested against multiple nodes of the other tree. Still, at most $O(1)$ memory - that of that

| $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $b_1$ | $b_1$ | $b_1$ | $b_1$ | $b_2$ | $b_2$ | $b_2$ | $b_2$ | $b_3$ | $b_3$ | $b_3$ | $b_3$ | $b_4$ | $b_4$ | $b_4$ | $b_4$ |

**Table 1**: Permutation of the values for the four DOPs $a_1, ..., a_4$ of a node $A$ and the four DOPs $b_1, ..., b_4$ of a node $B$ to produce a single 512 Bit AVX register for comparing all $4 \times 4$ possible combinations [4].

one leaf-parent node - will have temporal locality, independent of input sizes. A node stores the kDOPs of all 4 children in contiguous memory and leaves point to a contiguous triangle-array, therefore some spacial locality exists. However, the location of different nodes is arbitrary and they have to be fetched via pointer access. So again, at most $O(1)$ memory (of 2 nodes) has spacial locality at any given time.

**Bandwidth/Data Transfer Analysis.** Like all aspects of this algorithm, operational intensity depends on the underlying scene geometry. Specifically, the ratio of triangle-triangle tests to inner-node tree traversal tests is relevant here. Over all, the baseline operational intensity is between $\approx 0.15$ and $\approx 0.45$. Further, while operational intensity depends on the scene geometry, it does **not** depend on input size: almost all cache misses we incur are compulsory misses.

### 3.5. BVH SIMD Optimizations

We vectorize both inner-inner and leaf-inner tests. The main idea for vectorizing comes from [4].

**Inner-Inner Tests.** Here we aim to perform all 16 possible tests at once. The 4 kDOP vectors in an inner node are stored contiguously in a $k \times 4$ float $2D$-array. This means, a single `_mm_512_load_ps()` can load the kDOP values for 4 DOP directions for all four children into one vector $v$. After this, for each of the $k/2$ DOP directions, we extract a suitable permutation for comparing the intervals using a `_mm512_permutexvar_ps()` instruction. Table 1 shows the permutations inside the registers. The two registers can then be compared via `_mm512_cmp_ps_mask()` and each of the resulting 16 bits indicates (non-)overlap along the current direction of one of the 16 pairs of children.

**Leaf-Inner Tests.** The approach is similar to the Inner-Inner vectorization, except that now we test one kDOP against 4 kDOPs. To fully exploit the AVX-512 registers, we now test all four kDOP pairs **along four DOP directions** at the same time. An non-obvious consequence is that during loading, the new permutations no longer cross AVX lanes. Thus, the faster `_mm512_permute_ps()` instruction can be used.

### 3.6. BVH Quantization via Half-Precision Floats

The `x86` extension set called `F16C` provides hardware support for converting to and from 16 bit half-precision floats. As our tree traversal is memory-bound, we aim to reduce the memory bandwidth by storing the DOP values using 16 bit half floats. Each register now fits 32 such half floats. However, no half-precision arithmetic operations are supported, so the values have to be converted back to 32 bit floats. The conversion happens via two steps:

- An `_mm512_extracti64x4_epi64()` instruction and the `_mm512_cvtph_ps()` conversion for the high 16 half floats and
- A no-op cast `_mm512_castsi512_si256()` followed by `_mm512_cvtph_ps()` for the low 16 half floats

Aside from this additional conversion overhead, all other vectorization ideas translate very well. As each load takes 32 half floats, we also unroll the loop once, even though registers and ports are already filled in a single iteration.

Due to less precise BV fit, there is a slight increase in false positives for triangle-triangle tests (non-overlapping triangles tested due to overlapping BVs). However, the faster traversal most often makes up for this and we still see a speedup when using the quantized implementation.

### 3.7. BVH Specialized Recursion

Observe that as we traverse the trees, once we reach a leaf in either tree we will not visit an inner node in that tree again until we back-track from our current recursion level. We exploit this and build two recursive functions: the general case seen in Algorithm 2 and the specialized case, where the inner-inner case is not present. The specialized function has fewer parameters (only 2 pointers and one int vs. 4 pointers and 2 ints in the general case). Further, we don't have to run an always negative test for the inner-inner case. We apply this to both the vectorized and quantized versions.

## 4. EXPERIMENTAL RESULTS

This section gives the hardware and software specifications for the machines used to run GJK and BVH respectively, as well as the experimental results of the optimizations described in the Method section.

**Experimental setup for GJK.**

We conducted our experiments for GJK using an Intel® Core™ i7-1068NG7 processor which uses the Icelake architecture. It has a base frequency of 2.3 GHz. The cache sizes for L1, L2 and L3 caches

are 80KiB (32 KiB **I** + 48 KiB **D**), 512KiB and 8MiB (shared) respectively. L1 and L2 caches are 8 way associative, and L3 cache is 16 way associative. To compile our code, we used GCC version 10.2.0. The optimization flags we used are: `-std=c11 -O3 -march=icelake-client -ffast-math`. The experiments were conducted using **cold** caches.

As inputs we generated two spheres of equal sizes. The sizes of the each sphere ranged from 16 vertices to $2^{22}$. GJK algorithm only needs the convex hull of input objects to operate. Therefore, all of the points of the spheres were on the surface.

**Results (GJK).** Fig. 1 shows the speedup each optimization achieved compared to the baseline GJK implementation. It appears that standard C optimizations failed to improve GJK algorithm significantly. The block optimizations provide 25% speedup and combining block optimizations with the method inlining/joint loop trick brings us up to 45-50% total speedup. The mediocre speedup is caused by the dependencies in our bottleneck. When we check Fig. 3 we can see that our potential peak performance is actually not that high. In the optimized versions, inside support function loop, we do 1 multiplication, 2 FMAs and 1 comparison operation. All of these instructions create a Read After Write (RAW) dependency. Just by considering instruction mix, the peak performance drops from 4 flops/cycle to 3 flops/cycle. When we further consider the instruction dependencies, we observed that we issue 48 flops in 19 cycles per iteration of the loop. This further brings our potential peak performance to 2.52 flops/cycle. Unfortunately, our standard C optimized versions failed to reach this peak. This suggests that there are still possible avenues of optimization in these versions.

Vectorization with AVX2 intrinsics failed to achieve any meaningful speedup. The vectorization limitations from AoS format and the subsequent shuffling/combining of smaller vectors hurt the speed of the algorithm. When we look at Fig. 4, AVX2 implementation seems to have a slightly better performance compared to standard C versions, however this is misleading since we simply do 15% more computations in AVX2 version to do the same work as in standard C versions.

Our initial AVX512 implementation without strength reduction achieved more than 2x speedup compared to AVX2 implementation with strength reduction. Conversion to SoA format greatly benefited the vectorized implementations because it enabled us to use larger vectors. We further gained 3x speedup when we applied strength reduction to the index calculations in the AVX512 implementations. This

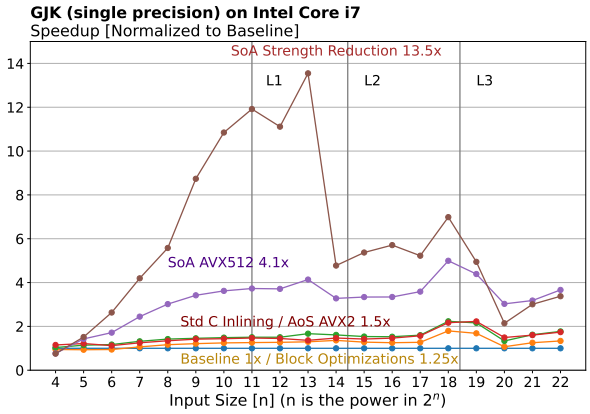


**Fig. 1**: Speedup plot of all versions of GJK algorithm. Execution time is measured in cycles. The execution times are normalized to the baseline runtime.

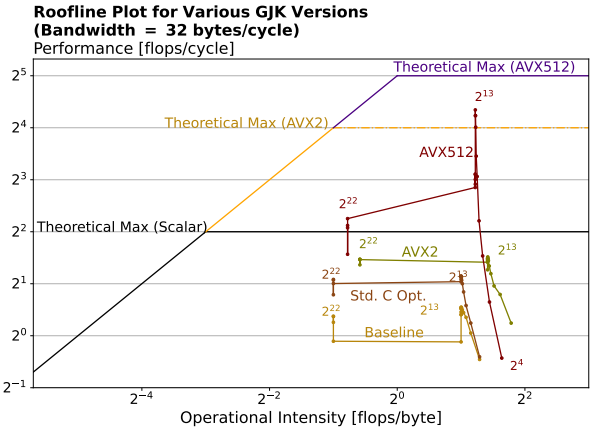


**Fig. 2**: Roofline plots for GJK. AVX2 version has around 15% and AVX512 has around 30% more flops.

brought us to our peak performance of 13.5x speedup compared to the baseline implementation. As can be seen from Fig. 4, our maximum performance is close to the peak performance of 22.4 flops/cycle we get when we consider the instruction mix. We again use 1 multiplication, 2 FMAs and 1 comparison operation in our bottleneck loop. However, in the vectorized implementations we also use an additional maximum operation. This means that when we consider only the instruction mix, we issue 112 flops in 5 cycles. We are likely to be even closer to real peak performance when we consider the instruction dependencies as well.

In our bandwidth/data transfer analysis, we mentioned that the baseline GJK implementation was compute bound. From Fig. 2 it can be seen that, this turned out to be true for almost all of our implementations. Only when the data does not fit in cache the AVX512 implementations become memory bound.

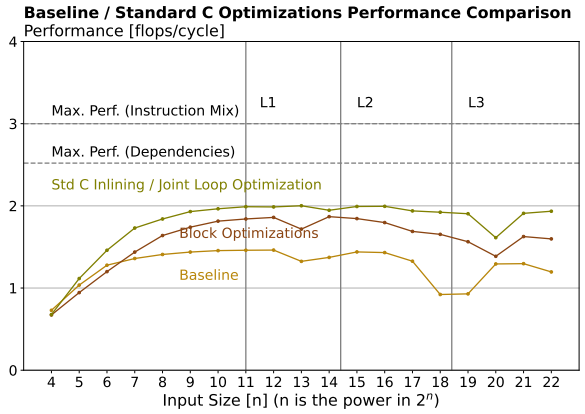


**Fig. 3**: Performance plots for scalar implementations of GJK. The operation counts are roughly the same.

This is caused by the shift in operational intensity when the data does not fit in cache. Our operational intensity decreases because we iterate over the same two objects over and over again. When the objects do not fit in cache we have to repeatedly bring them into caches, hence, more data transfers between L3 cache and main memory.

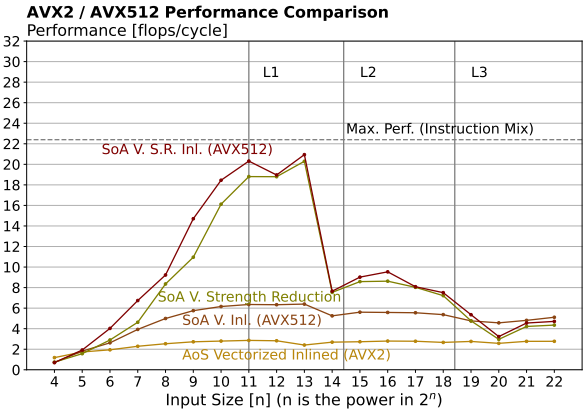


**Fig. 4**: Performance plots for vectorized implementations of GJK. The operation count of AVX512 versions is roughly 10% more than operation of AVX2 version.

**Experimental Setup for BVH.** We conducted our experiments for BVH using a server with an Intel Xeon `D-2141I` CPU (Skylake Server architecture). It has a nominal base frequency of 2.20 GHz. The cache sizes for L1, L2 and L3 caches are 64 KiB (32 KiB **I** + 32 KiB **D**) per core, 1 MiB per core and 11 MiB (shared) respectively. L1 cache is 8-way set associative, L2 cache is 16-way set associative and L3 is 11-way associative. To compile our code we used GCC version 11.1. The optimization flags we used are `-std=c11 -march=skylake-avx512 -mprefer-vector-width=512 -ffast-math`. The experiments were conducted using **cold** cache. Warm cache isn't representative (one wouldn't repeat the exact same collision test more than once).

As input, we imported two high-resolution 3D models (in .OBJ format) and set up two scenes: where the objects intersect and where they don't intersect. To try and somewhat control for the algorithm's dependence on scene geometry, we decided to generate the both scenes each at multiple resolutions, using Blender's "Decimate Geometry" modifier [13]. The full-resolution scenes have $\approx$ 5.4M triangles total, and we generated scenes in resolutions from $10\%, 20\%, ..., 90\%, 100\%$.

**Results (BVH).** Many of the results obtained cannot be immediately explained by a typical performance analysis. Rather, it is helpful to look at the number of triangle-triangle tests performed, shown in Fig. 5. In the following, we will keep referring back to the triangle-triangle test number when discussing the results.

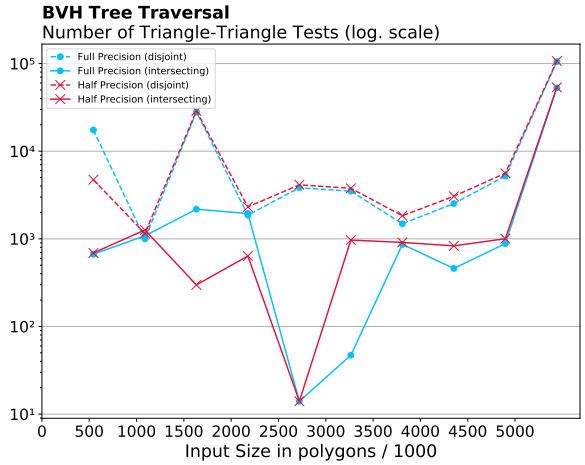


**Fig. 5**: Number of triangle-triangle tests performed during BVH traversal.

Fig. 6 shows the runtime in cycles for each BVH version, as a function of # Triangles in the Scenes. We see fairly uniform runtimes, even as the input sizes vary by millions of triangles. The one exception is a 4× (disjoint) or 10× spike (intersecting) in runtime for the 100% scene. The main reason is that we also perform 20× (disjoint) or 60× (intersecting) more triangle tests. We suspect that Blender's "decimate geometry" is very beneficial to the convergence of Neural Gas during tree construction. The intersecting case generally takes an order of magnitude less cycles, with a similar decrease in triangle-triangle tests.

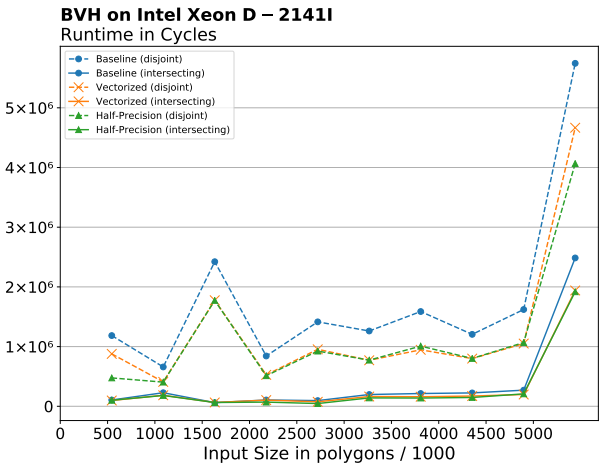


**Fig. 6**: Runtime of BVH in cycles for all optimizations and both intersecting and disjoint scenes.

Fig. 8 shows the performance in `flops/cycles` of BVH. Within each optimization level and scene-type (intersecting or not), performance is fairly consistent. As a greater proportion of time is spent traversing trees when the objects are apart, we also see the largest performance increase. Vectorized code achieved almost a 3× performance increase at ≈ 1.4 `flops/cycle`. An even larger increase was reached via quantization: over 4×. However, those performance increases are **misleading**, as the runtime speedup is only around 1.6×. To explain this, recall that baseline implementation does not have to test all kDOP directions in case it finds non-overlap along one of them. Both vectorized and quantized code do not have this "early stopping" so they perform more flops for the same result.

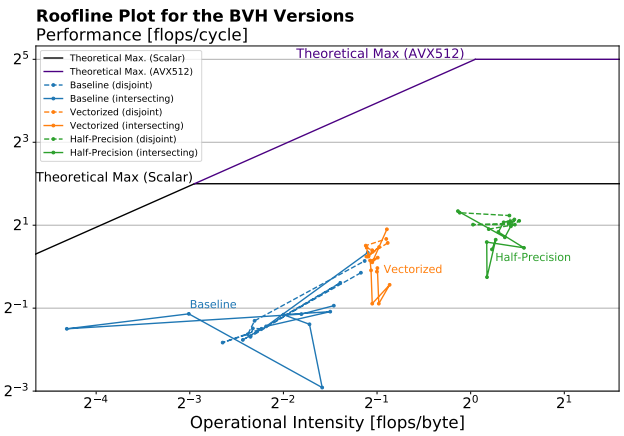


**Fig. 7**: Roofline plot for the BVH optimizations in both scenes. Vectorized code has ≈ 2× and quantized code ≈ 3× the flops of baseline.

Looking at the roofline plot in Fig. 7, we see that indeed most BVH runs were memory bound. Only half-precision quantized code - and only for half the input scenes - is barely compute bound. The around 2× operational intensity of half-precision vs. vectorized fp32 is a clear consequence of moving ≈ half the data per node. Interestingly, operational intensity remains consistent for each optimization level, across sizes and even intersecting and disjoint scenes. This is in contrast to the runtime (and to a lesser extent performance) differences between the intersecting and disjoint scenes.

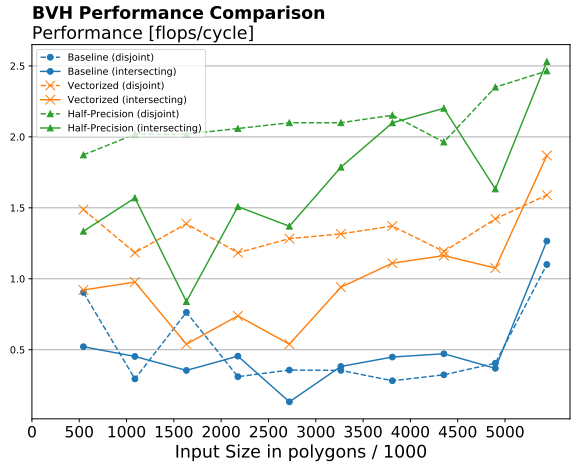


**Fig. 8**: Performance of the BVH optimizations, for both intersecting and disjoint scenes.

## 5. CONCLUSIONS

We have presented optimized implementations of two collision detection algorithms: The Gilbert-Johnson-Keerthi Algorithm (GJK) and Bounding Volume Hierarchies (BVH).

The best performing implementation of GJK gave a 13.5× speedup compared to the baseline implementation, performing close to the theoretical peak of 22.4 flops/cycle. SoA format and the use of AVX-512 gave the biggest improvement in performance.

For BVH, performance and cycle count depend more on underlying geometry. We achieve a maximal speedup of 2.45× (with an up-to 8× increase in performance). Performance was below peak, due to the poor locality and "pointer-chasing" nature of tree traversal.

Future work could further parallelize intersection testing (GJK and BVH traversal) by utilizing multiple cores (or even GPUs). And with ever-wider vector registers, the tree arity on BVH trees could further be increased for even better performance.

## 6. CONTRIBUTIONS OF TEAM MEMBERS

### 6.1. Mihai

Created the Baselines for both GJK and BVH. Implemented all BVH optimizations (vectorization, quantization, recursion). Instrumented BVH code for profiling and analysis. Collected cycle, flop count and data movement information for BVH. Plotted the speedup, performance and roofline models for the BVH implementations.

### 6.2. Berke

Implemented the initial baseline for GJK. Focused on GJK optimizations and analysis. Did the profiling, bandwidth/data trasfer and locality analysis for baseline GJK implementation. Implemented all of the standard C optimizations and vectorized versions of GJK. Collected the cycle and flop count information for the plots by instrumenting the code. Plotted the speedup, performance and roofline models for the GJK implementations.

### 6.3. Chris

Analyzed BVH Triangle-Triangle collision for optimization potential. Contributed to runtime measurement framework BVH: E.g. mesh loading/conversion, added cube and sphere collision test cases. Multiresolution runtime tests with Mihai. Investigated BVH runtime instabilities. Created interactive 3d visualizations GJK and BVH to aid analysis and verification of performance bottlenecks, correctness and runtime stability.

### 6.4. Liam

Implemented random GJK test cases for profiling optimizations. Collected cycle and flop count information for BVH performance plots. Worked on triangle-triangle optimization for BVH.

## 7. REFERENCES


[1] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *IEEE Journal on Robotics and Automation*, vol. 4, no. 2, pp. 193–203, 1988.

[2] Casey Muratori, "Implementing gjk," 2006.

[3] Jeff Linahan, "A gilbert-johnson-keerthi algorithmgeometric interpretation of the boolean gilbert-johnson-keerthi algorithm," 2015.

[4] Toni Tan, René Weller, and Gabriel Zachmann, "Simdop: Simd optimized bounding volume hierarchies for collision detection," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 7256–7263.

[5] Philip M. Hubbard, "Approximating polyhedra with spheres for time-critical collision detection," *ACM Trans. Graph.*, vol. 15, no. 3, pp. 179–210, July 1996.

[6] Gino van den Bergen, "Efficient collision detection of complex deformable models using aabb trees," *J. Graph. Tools*, vol. 2, no. 4, pp. 1–13, Jan. 1998.

[7] J.T. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-dops," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.

[8] S. Gottschalk, M. C. Lin, and D. Manocha, "Obbtree: A hierarchical structure for rapid interference detection," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1996, SIGGRAPH '96, p. 171–180, Association for Computing Machinery.

[9] S. A. Ehmann and M. C. Lin, "Accurate and fast proximity queries between polyhedra using convex surface decomposition," *Computer Graphics Forum (Proc. of EUROGRAPHICS 2001)*, vol. 20, no. 3, pp. 500–510, 2001.

[10] Paul R. Halmos, "Review: Nelson Dunford and Jacob T. Schwartz, Linear operators. Part I: General theory," *Bulletin of the American Mathematical Society*, vol. 65, no. 3, pp. 154 – 156, 1959.

[11] René Weller, David Mainzer, Abhishek Srinivas, Matthias Teschner, and Gabriel Zachmann, "Massively Parallel Batch Neural Gas for Bounding Volume Hierarchy Construction," in *Workshop on Virtual Reality Interaction and Physical Simulation*, Jan Bender, Christian Duriez, Fabrice Jaillet, and Gabriel Zachmann, Eds. 2014, The Eurographics Association.

[12] Tomas Möller, "A fast triangle-triangle intersection test," *Journal of Graphics Tools*, vol. 2, no. 2, pp. 25–30, 1997.

[13] Blender Online Community, *Blender - a 3D modelling and rendering package*, Blender Foundation, Stichting Blender Foundation, Amsterdam, 2021.