SIMDop: SIMD Optimized Bounding Volume Hierarchies for Collision Detection

Toni Tan[1], René Weller[1] and Gabriel Zachmann[1]

***Abstract*— We present a novel data structure for SIMD optimized simultaneous bounding volume hierarchy (BVH) traversals like they appear for instance in collision detection tasks. In contrast to all previous approaches, we consider both the *traversal* algorithm and the *construction* of the BVH. The main idea is to increase the branching factor of the BVH according to the available SIMD registers and parallelize the simultaneous BVH traversal using SIMD operations. This requires a novel BVH construction method because traditional BVHs for collision detection usually are simple binary trees. To do that, we present a new BVH construction method based on a clustering algorithm, Batch Neural Gas, that is able to build efficient n-ary tree structures along with SIMD optimized simultaneous BVH traversal. Our results show that our new data structure outperforms binary trees significantly.**

I. Introduction

Collision detection (CD) algorithms are essential for sampling-based motion planning algorithms. They are used to test whether a sampled configuration is in collision with the workspace obstacles. In most sampling-based motion planning algorithms, the collision computation is the computational bottleneck that requires up to 90% of computation time [1].

For most CD algorithms that work with polyhedral models, *Bounding Volume Hierarchies (BVHs)* are the common technique used to accelerate the intersection queries. The basic idea is simple: instead of calculating slow and complex geometric intersection tests between all geometric primitives, we wrap them recursively into simple *bounding volumes (BVs)* such as spheres, *axis-aligned bounding boxes (AABB)*, *oriented bounding boxes (OBB)* or *discrete oriented polytopes (k-DOP)*, that allow faster intersection tests. This generates a tree data structure with a single large BV at the root position that encloses all geometric primitives. Obviously, the geometric primitives are the leaves of such a BVH.

As for the traversal, we usually have *two* BVHs that we want to check for intersection, one for each object. We start with the root nodes and simultaneously traverse recursively the children in case of intersection of the BVs (see Algorithm 1 and Figure 1).

Following the trend of acceleration by parallelization it is obvious to apply this idea also to BVH traversals. Unfortunately, the parallelization of the simultaneous traversal for collision detection is not obvious. Actually, due to their recursive nature, BVHs are not very well suited for massively parallel acceleration on the GPU. Moreover, especially for online planning, robots are often not equipped with powerful GPUs.

[1]University of Bremen, Germany

Algorithm 1: BVHtraversal(BV a, BV b)

```
if $a$ and $b$ are both leaves then
    checkPrimitives($a$, $b$)
else if $a$ is leaf then
    forall children $b_i$ of $b$ do
        if $a$ and $b_i$ intersect then
            BVHtraversal($a$, $b_i$)
else if $b$ is leaf then
    forall children $a_i$ of $a$ do
        if $a_i$ and $b$ intersect then
            BVHtraversal($a_i$, $b$)
else
    forall children $a_i$ of $a$ and $b_i$ of $b$ do
        if $a_i$ and $b_i$ intersect then
            BVHtraversal($a_i$, $b_i$)
```

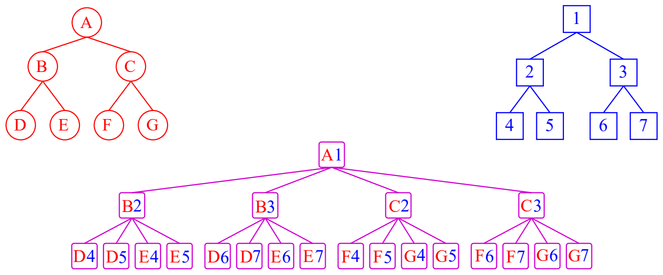


Fig. 1: The simultaneous recursive traversal of two binary BVHs during the collision check results in a bounding volume test tree.

However, the simultaneous traversal required in BVH-based collision detection can still benefit from the SIMD instruction sets of modern CPUs.

To take advantage from this parallel operations, we can:

1) simply switch on a compiler option and hope that the compiler will do the optimization,
2) optimize the *traversal* function manually, depending on the chosen BVH,
3) or adapt the complete BVH structure which additionally requires a redesign of the BVH *construction*.

In this paper we have implemented and tested all of these three methods. There is only one suitable function in Algorithm 1 to optimize the traversal without changing the tree structure: the test for intersection of a pair of BVs. Hence, the benefit of SIMD optimization relies heavily on the

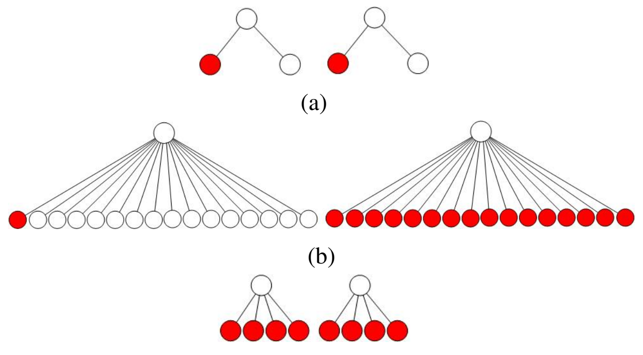


Fig. 2: Different strategy to compute intersections of the child nodes in the simultaneous traversal algorithm: (a) classic collision query tests only one pair of nodes. With SIMD, assuming 16 registers, we can (b) test one node of the left BVH against 16 nodes of the right BVH simultaneously in the case of a branching factor of 16 or (c), in case of a branching factor of 4, test all nodes from same level at one time.

type of the BV. For two spheres, we simply have to compute the distance of two points and compare it to the sum of the spheres' radii. This is not very well suited for the for SIMD parallelization because of the length of current AVX512 registers that are able to store 16 floating point values. As a consequence, the intersection test for two spheres can be hardly optimized for SIMD. Similarly, the intersection test for AABBs requires four comparisons. Modern AVX registers compare 16 float value in a single instruction and this number will increase with upcoming CPU generations. Hence, these BVs could benefit only from the third method, an optimized BVH, but hardly from a simple optimization of the traversal. Consequently, we decided to use a BV that naturally supports all three methods: the k-DOP. Basically, k-DOPs are an extension of AABBs to arbitrary orientations [2]. They offer a natural trade-off between tightness of the BV and computation time for the intersection test. They show comparable performance to other kinds of bounding volumes [3]. By choosing the number of orientations k according to the SIMD instruction set, it is straightforward to adapt this BV-type to further SIMD developments.

However, this simple SIMD-parallelization still tests only two BVs in one instruction (see Figure 2a). Hence, it can be applied to almost all existing k-DOP-based BVHs that typically rely on a binary tree. However, we can also parallelize it in a way that one BV of the first BVH is tested simultaneously against *all* children of the other BVH (see Figure 2a). This is exactly the idea of our new data structure that we call *SIMDop*. In order to take full advantage of SIMD in this case we additionally have to change the branching factor of the tree. This is non trivial because traditional BVH construction methods, like the *surface area heuristic (SAH)*, median-, or mid point-split, that assign the primitives into the sub-trees are not suitable for higher branching factors. Consequently, we have developed new BVH construction methods, this includes simple heuristics but also a new method that is based on Batch Neural Gas clustering. The advantage of such n-ary trees is not only the SIMD accelerated traversal. Additionally, we get less children than with binary trees and the children are also smaller. We have implemented our novel SIMDop BVH and the results show that it outperforms traditional binary trees by an order of magnitude.

II. Previous Work

In many fields of computer science, BVHs has been used widely to accelerate intersection computation. Usual BVs for the BVHs are spheres [4], AABBs [5] and their memory optimized derivative called BoxTree [6] that is closely related to kd-Trees, k-DOPs [7], [2], a generalization of AABBs, OBBs [8] or convex hull trees [9]. Additionally, a wide variety of special BVs for special applications has been developed. For instance spherical shells [10], swept spheres [11], spheres that are cut by two parallel planes called slab cut balls [12], quantised orientation slabs with primary orientations (QuOSPO) trees [13] that combine OBBs with k-DOPs, or combinations of spherical shells with OBBs that was proposed by [14] for objects that are modelled by Bezier patches.

In sampling-based motion planning, AABB-based BVHs are widely used to calculate collision between candidate trajectories in workspace [15]. The *Flexible Collision Library (FCL)* [16] also supports several BVs for its BVH such as AABB, OBB, rectangle swept spheres (RSS) and k-DOPs. Another approach using a hierarchical three-stage sequence of BVs namely AABB, Sphere, and OBBs [17].

Usually, a BVH is constructed in a pre-processing step that can be computationally more or less expensive. Basically, there exist two major strategies to build BVHs: bottom-up and top-down. The bottom-up approach starts with elementary BVs of leaf nodes and merges them recursively together until the root BV is reached. A very simple merging heuristic is to visit all nearest neighbours and minimize the size of the combined parent nodes in the same level [18]. Less greedy strategies combine BVs by using tilings [19].

However, the most popular method is the top-down approach. The general idea is to start with the complete set of elementary BVs, then split that into some parts and create a BVH for each part recursively. The main problem is to choose a good splitting criterion. A classical splitting criterion is to simply pick the longest axis and split it in the middle of this axis. Another simple heuristic is to split along the median of the elementary bounding boxes along the longest axis. However, it is easy to construct worst case scenarios for these simple heuristics. SAH tries to avoid these worst cases by optimizing the surface area and the number of geometric primitives over all possible split plane candidates [20]. Originally developed for ray tracing, it is today also used for collision detection. The computational costs can be reduced to $O(n \log n)$ [21], [22] and there exists parallel algorithms for the fast construction on the GPU [23]. Many other splitting criteria were compared in [24].

The influence of the trees' branching factor is widely neglected in the literature. Usually, most authors simply use binary trees for collision detection. According to Zachmann and Langetepe [25], the optimum branching factor can be larger. Mezger et al. [26] stated that, especially for deformable objects, 4-ary trees or 8-ary could improve the performance. This is mainly due to fewer BV updates. To our knowledge, there does not exist any work that investigates the influence of the branching factor of the BVH for simultaneous traversal tasks.

III. SIMD Recap

Originally, *SIMD instruction sets* had been introduced to support integer computation for intensive multimedia applications, but later they have been extended to support floating point computation which extends the usefulness also for scientific computations. The idea is that a single instruction operates on different input data values (e.g. 8 or 16 floating point values) simultaneously. Several slightly different SIMD instruction sets are available for various CPUs; e.g. NEON for Arm based CPUs and SSE/AVX for both Intel and AMD CPUs (see Table I for a list of available SIMD instruction sets and the supported data types). The most current AVX512 instruction set supports computation of 16 single precision-float in parallel. In this paper, we focus on mainly AVX512, however the idea can be easily implemented on other SIMD instruction sets such as SSE, AVX, and NEON. Moreover, we included measurements for AVX in our results and we are confident, that more powerful AVX registers will be available for the other platforms soon.

Name	Width	Types	supported CPUs
NEON	128 bits	4x single 2x double*	Armv7-A/R and above *only available for Armv8-A
SSE	128 bits	4x single	Intel Pentium III and above AMD Athlon XP and above
SSE2 SSE3 SSE4	128 bits	4x single 2x double	Intel Pentium 4 and above AMD Athlon XP and above
AVX AVX2	256 bits	8x single 4x double	Intel Sandy Bridge and above AMD Bulldozer and above
AVX512	512 bits	16x single 8x double	Intel Skylake-X and above

TABLE I: Floating point support for various SIMD Instruction Sets

IV. Our SIMDop Data Structure

The main idea of our SIMDop data structure is to construct BVHs with higher branching factor that can be later used during run-time in a SIMD optimized traversal algorithm. Hence, the core is the *construction* that is typically done in a pre-processing step. We propose different methods to construct such n-ary BVHs.

A. BVH Construction

We decided to use a top-down approach for the hierarchy construction. The general idea is to start with the complete set of elementary BVs, then split that into some parts and create a BVH for each part recursively. Moreover, we use a *wrapped hierarchy* according to the notion of Agarwal et al. [27], where inner nodes are tight BVs for all their leaves, but they do not necessarily bound their direct children. Compared to layered hierarchies, the big advantage is that the inner BVs are tighter. The main challenge is to choose a good splitting criterion especially, because traditional splitting criteria like SAH do not work for n-ary trees. We propose several splitting criteria for higher branching factors that we will shortly sketch in the following sections.

1) Longest Axis Split: A classical splitting criterion for binary trees is to sort the primitives along all coord axis and simply pick the longest axis and split this sorted list in the middle of this axis. Obviously, we can easily extend this two n-ary trees by not splitting in the middle, but split the number of BVs into n equal parts. However, this leads to fairly well balanced trees (see Figure 3).

2) Extended Longest Axis Split: This is an extension to the longest axis split for n-ary trees where n is preferably in the power of two. We do not simply split along one axis but perform in the first stage a binary longest axis split and than recursively split the primitive sets again until we reach n. In other words, we perform a traditional binary tree split but remove the not needed nodes: instead, we can directly mount all children to the parent node.

3) Batch Neural Gas Clustering: Clustering algorithms, especially BNG, have shown to be very efficient for BVH constructions of 4-ary trees [28]. A nice property of BNG is that it exhibits very robust behavior with respect to the initial cluster center position in contrast to other clustering algorithms like k-means. However, in the original work, the authors used spheres as basic primitives instead of more usual polygonal representations. We simply used the centers of the polygons instead of the spheres' centers reported in the original work in our polygonal implementation. We did not use magnification control, which additionally considers the size of the spheres to produce better clustering results. However, this can be easily added in the future to our polygon-based BNG.

Figure 3 shows the first hierarchy level for all our splitting criteria and different branching factors.

B. BVH Traversal

The key part to optimize the traversal in Algorithm 1 is the test for intersection of the child bounding volumes. For binary trees, the four possible combinations of child pairs are usually traversed sequentially. SIMD enables us to accelerate this intersection test in several ways:

- We can use a SIMD instructions to replace a single test of a pair of BVs (see Figure 2a). This would leave the for-loop untouched and just replace the intersection method.
- We can also remove the first part of the for-loop and test one BV of the first BVH simultaneously against *all* children of the other BVH (see Figure 2b). For AVX512 this results in a 1 vs 16 check. Accordingly, we call our BVH using this approach 1vs16-SIMDop.

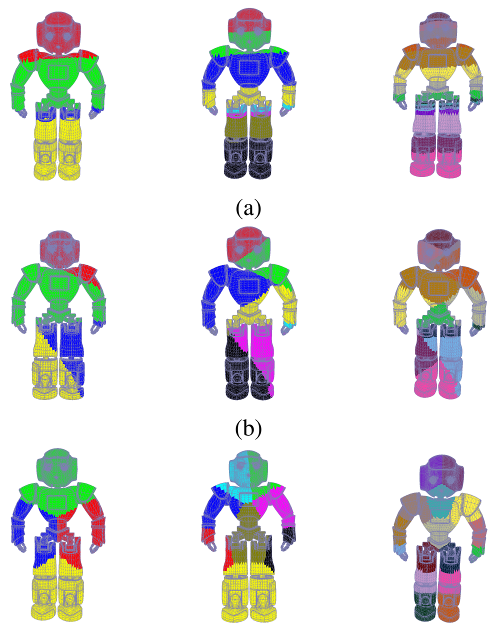


Fig. 3: The results of our hierarchy construction algorithms showing the color coded first level of the hierarchy: (a) longest axis, (b) extended longest axis, and (c) Batch Neural Gas. The trees have degree four on the left side, degree eight in the center, and sixteen on the right side.

- Finally, we can remove all for-loops and test all nodes from the same level at one time (see Figure 2c). With AVX512 this results in a 4 vs 4 test and we call the respective BVH 4vs4-SIMDop.

An implementation of the first idea is straight forward, it requires a simple replacement of the comparison inside the intersection function.

Algorithm 2 shows the naive implementation for the 1vs16-SIMDop, i.e. the removal of the inner for-loop by testing one BV of object A against n BVs of object B, using the current AVX512 instruction set looks as follows, assuming that we are using DOPs with k orientations for the BVs[1].

Removing both loops for the 4vs4-SIMDop, i.e. testing all n child BVs of object A against all n child BVs of object B for a pair of nodes requires just a slightly different ordering of the Dop values which results in the AVX512 code that is shown in Algorithm 3.

There are some drawbacks of these SIMD implementations: first, we have to copy all the values of the DOPs to AVX registers. Second, we have to combine the temporal results using *or*-instructions to compile the end result. A non-parallel version to check two DOPs for overlap would simply

Algorithm 2: _m512 intersect(DOP a, DOP b1,...,b16)

```
_mm512 endResult
for i=0; i¡k/2; i++ do
    _mm512 oriAL = _mm512_set1_ps(a[i])
    _mm512 oriBL = _mm512_set_ps(b1[i],...,b16[i])
    _mm512 resL = _mm512_cmp_ps(oriAL, oriBL,
     _CMP_LT_OS)
    _mm512 oriAH = _mm512_set1_ps(a[k/2+i])
    _mm512 oriBH = _mm512_set_ps(b1[k/2+i],...,b16[k/2+i])
    _mm512 resH = _mm512_cmp_ps(oriAH, oriBH,
     _CMP_GT_OS)
    _mm512 tempRes = _mm512_kor(resL,resH)
    endResult = _mm512_kor(endResult, tempRes)
    if endRes == 65535 then
        break
return endResult
```

Algorithm 3: _m512 intersect(DOP a1,..,a4, DOP b1,..,b4)

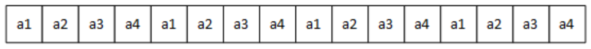
```
_mm512 endResult
/ for i=0; i¡k/2; i++ do
    _mm512 oriAL = _mm512_set_ps(a1[k/2+i],...,a4[k/2+i])
    _mm512 oriBL = _mm512_set_ps(b1[i],...,b4[i])
    _mm512 resL = _mm512_cmp_ps(oriAL, oriBL,
     _CMP_LT_OS)
    _mm512 oriAH = _mm512_set_ps(a1[i],...,a4[i])
    _mm512 oriBH = _mm512_set_ps(b1[k/2+i],...,b4[k/2+i])
    _mm512 resH = _mm512_cmp_ps(oriAH, oriBH,
     _CMP_GT_OS)
    _mm512 tempRes = _mm512_kor(resL,resH)
    endResult = _mm512_kor(endResult, tempRes)
    if endResult == 65535 then
        break
return endResult
```

compare two values and use one boolean operation. Hence, in these naive implementations we would need 9 AVX instructions vs. 3 instructions in the non-AVX implementation to compare one orientation of the DOP. Moreover, the non-AVX version could escape the loop earlier for some of the 16 children whereas in the SIMD cases we have to iterate the loop $k/2$-times if only one of the 16 children overlaps the other DOP. Hence, we could assume an acceleration of at most $\frac{3\times16}{9}$, because we test 16 children simultaneously, not considering the faster loop escapes and the smaller BVs of the SIMDop structure.

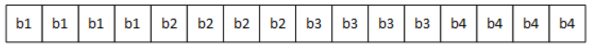
C. Optimization

Our benchmarks have shown that actually, our naive implementations for Algorithms 2 and 3 perform worse than the non-AVX version. The main reason is that the *_mm512_set_ps* and *_mm512_set1_ps* instructions that load the data into the AVX registers require more time than the other instructions. For the 1vs16-SIMDop we can easily solve this by directly storing the values into a proper AVX format. This would lead to a theoretical benefit of $\frac{3\times16}{5}$ because we only need 5 instructions per orientation, not considering the smaller BVs.

[1]Variables of the type *_mm512* are AVX512 variables with a length of 512 Bit. Intrinsic AVX512 instructions usually start with the prefix *_mm512_* followed by the particular operation and end with a suffix that indicates the data type: e.g. *_mm512_cmp_ps* compares the two input variables of 512 Bit width of the type single precision floating point (*_ps*), following the rule defined in the third parameter and returns the result as a 512 Bit vector. *_mm512_kor* defines a bitwise logical OR comparison using masks.

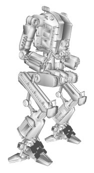


(a)

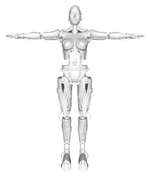


(b)

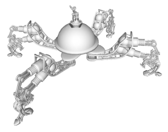
Fig. 4: Permutation of the values for the for DOPs $a1, ..., a4$ of an object A and the four DOPs $b1, ..., b4$ of an object B to produce a single 512 Bit AVX register for comparing all 4×4 possible combinations in Algorithm 3.



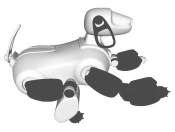
(a)



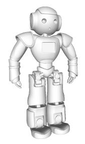
(b)



(c)



(d)



(e)

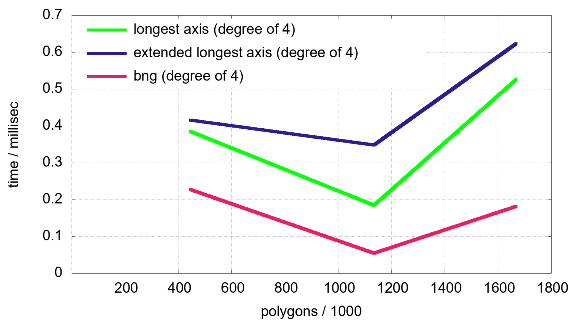
Fig. 5: The objects we used in our timings: (a) ATST walker robot, (b) female robot, (c) quadripod robot, (d) dog robot, and (e) Nao.

Moreover, we tested to use prefetching to improve the cache performance. Unfortunately, the size of a cache line in CPUs supporting AVX512 is exactly 512 Bit, so we did not see any acceleration. However, we were able to reduce the memory bandwidth by storing the DOP values as half floats. Since the 3rd generation of Intel® Core™ processors, the conversion of 16-bit half floats back to 32-bit float values is supported by the *vcvtps2ph* instruction without computational overhead. The resulting increase of false positives for the BVH traversal was neglectable.

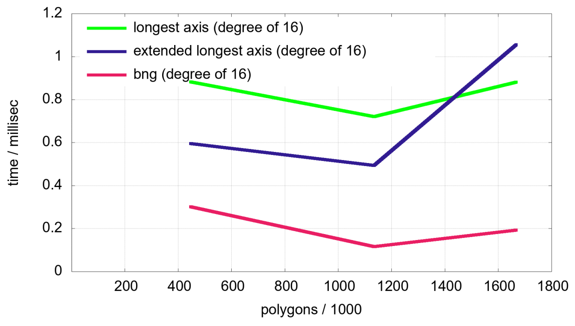
Obviously, we could also store the data for the 4vs4-SIMDop into an appropriate AVX512 variable. However, this would increase the memory footprint by a factor of 4 because we would have to copy each DOP value four times. In order to avoid this waste of memory and to further improve cache performance, we decided to use a different strategy for the 4vs4-SIMDop: AVX512 supports the function *_mm512_castps128_ps512* that casts 128 Bit SSE data to AVX512 data with zero latency. Hence, we store 4 floating point values per DOP orientation and use the permutation function *_mm512_permutexvar_ps* to shuffle the values to their correct positions (see Figure 4). This requires one more AVX instruction, leading to a theoretical benefit of $\frac{3 \times 16}{6}$ compared to the sequential binary tree, but it significantly improves cache performance.

V. Results

We have implemented our algorithms using C++ and *Intel Intrinsics functions* using Visual Studio 2017. We focused our implementation on the most recent AVX512 instruction



(a)



(b)

Fig. 6: Average collision query time for the different splitting criteria for the quadripod robot using (a) the 4vs4-SIMDop and (b) the 1vs16-SIMDop with respect to the polygon count. The results are very similar for all objects.

sets. All tests were performed on a system with an Intel I7 7800X CPU, 64GB of main memory and a NVIDIA Geforce GTX 980 GPU with 4GB of memory. We used the standardized benchmarking suite proposed by Trenkel et al. [3]. Figure 5 shows some of the used models with different shapes and resolutions in our timings: in particular, a ATST walker robot, a female android, a quadripod and a robotic dog. According to Trenkel et al. [3], we present all results in this section for the most time consuming distance preset, i.e. a distance of zero. For the best performance of the hand optimized traversal function of the binary tree, k should be divisible by 16. We set the number of orientations of the DOPs to $k = 32$ where not other mentioned because it performs better than $k = 16$ for all methods.

First, we evaluated the influence of the splitting criterion described in Section IV-A. The BNG clustering outperforms the other heuristics significantly in all our test cases, independent of the objects' shapes and polygon count (see Figure 6). The benefit of the clustering increases with and increasing number of branches in the tree. In term of BVH construction time, the BNG clustering-based SIMDop for both degree of 4 and 16 can be constructed almost as fast BVH constructed using V-COLLIDE and binary DOP tree (see Figure 7).

Moreover, we compared the performance of our two SIMDop variants to the other methods 1 and 2, i.e. the binary tree-based data structures with the compiler flag SIMD optimization and the manually SIMD-optimized traversal algorithm. The compiler flag optimized binary DOP tree

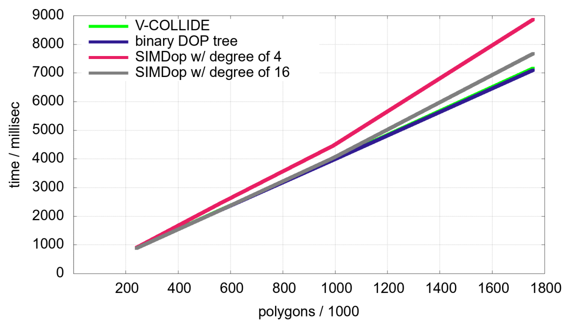
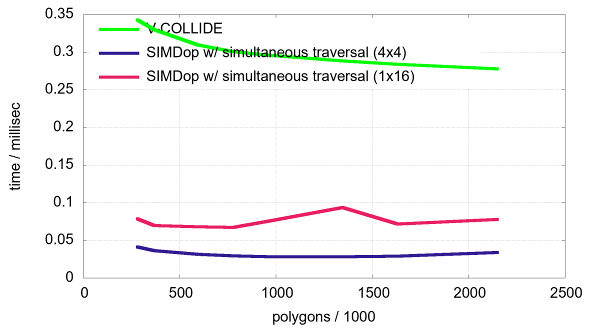
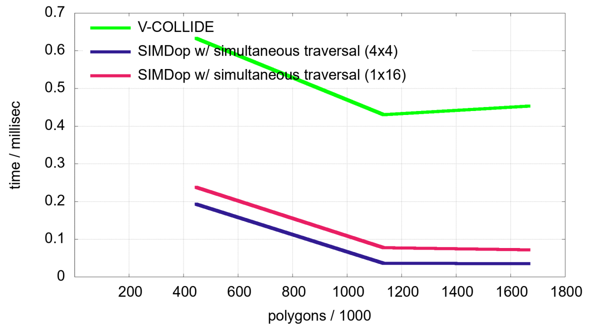


Fig. 7: A comparison of BVH construction time of our SIMDop based on BNG clustering algorithm compared with V-COLLIDE and binary DOP tree for the ATST walker robot.



(a)

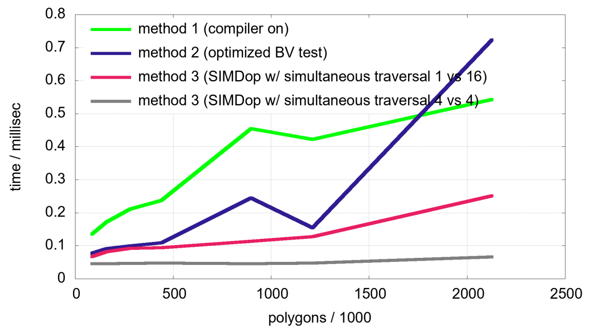


(b)

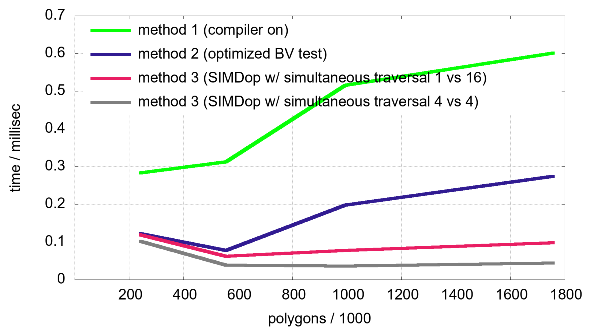
Fig. 8: A comparison of our SIMDop with V-COLLIDE library for (a) female robot, and (b) quadripod. The results show that our 4vs4-SIMDop performs best and faster than V-COLLIDE by up to eight times for (a) female robot and thirteen times for (b) quadripod.

and the manually AVX optimized DOP tree traversal have very similar running times. This gives a hint that compiler optimization seems to work very well. However, our two SIMD optimized SIMDop versions, the 4vs4-SIMDop and 1vs16-SIMDop both outperform both binary DOP trees by at least factor of 8 for the Nao (see Figure 9a), a factor of 13 for the ATST walker robot (see Figure 9b). In all cases this factor increases with an increasing polygon count. This is slightly higher than the theoretical factor of $\frac{3\times16}{5}$ we expected from the number of instructions for the intersection function. This indicates that the decreased size of the BVs due to the higher branching factor and the reduced number of overall BVs in a tree with higher branching factor compensate the increasing number of iterations required for the SIMD loop.

We also compared our SIMDop to the V-COLLIDE library that is often used for sample-based path planning tasks. An experimental comparative analysis has shown that V-COLLIDE outperforms other CD libraries like PQP [29]. Figure 8 shows that our 4vs4-SIMDop is able to outperform V-COLLIDE by a factor of up to 13 for the quadropod.



(a)



(b)

Fig. 9: Average collision query times for the compiler optimized binary DOP tree (method 1), the manually optimized binary DOP tree (method 2) and the two SIMD optimized Dop tree versions, the 1vs16-SIMDop and the 4vs4-SIMDop (method 3) with respect to the number of polygons in the (a) Nao and (b) ATST walker robot. The results show that our SIMDop are up to eight and thirteen times faster than both binary DOP trees and the 4vs4-SIMDop performs best.

We also investigated the influence of the actual SIMD version on the performance of our SIMDop. Figure 10 shows the results measured for an AVX version and an AVX512 implementation. The AVX512 implementation is twice as fast as the AVX due to the width of the AVX512 registers.

And finally, we evaluated the performance gain using SIMD optimized version of simultaneous BVH traversal compared with non-optimized version (see Figure 11). We roughly get a speedup around 2 for both 4vs4-SIMDop and 1vs16-SIMDop version, which is below our expectation since the AVX512 register can process 16 data at one time.

Hence, we investigated further by using a profiling tool Intel® VTune$^{\mathrm{TM}}$ to profile the actual collision query timing. Table II shows profiling result for object Nao using 4vs4-

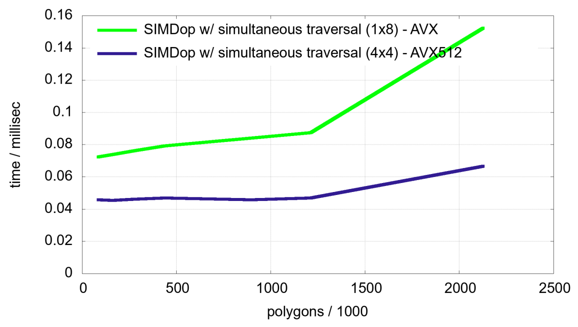


Fig. 10: Average collision query times for AVX implementation of our SIMDop in comparison with AVX512.

SIMDop. According to VTune, our 4vs4-SIMDop is able to vectorize 72.70% floating point instructions with the full vector capacity, which should theoretically give us a speedup of $\frac{72.7\times16}{100}$, however the gain is bound by memory, which took around 40.3% of computing time (whereas 33.40% of the time is stalled by main memory access). And also, our 4vs4-SIMDop has to test more orientations as much as three times more on average compared with non-optimized version (see Figure 12). In the end, we get roughly a speedup of 2 for the SIMD optimized version compared with the non-optimized version.

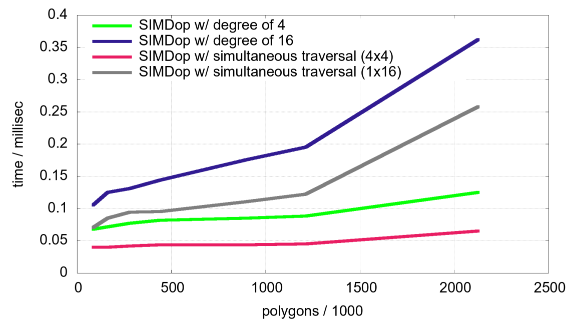


Fig. 11: Average collision query times using object Nao for our SIMDop with and without SIMD optimized simultaneous traversal.

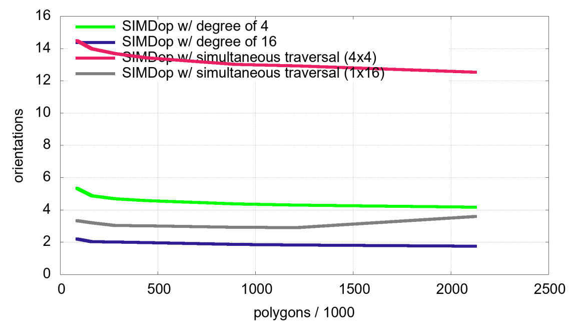


Fig. 12: Average orientations tested for a single bounding volume test of our SIMDop with and without SIMD optimited simultaneous traversal.

	SIMDop w/ degree of 4	SIMDop 4vs4
L1 Bound	3.00%	2.00%
L2 Bound	1.20%	1.60%
L3 Bound	2.70%	3.30%
DRAM Bound	21.10%	33.40%
Floating Point Vectorization	0%	72.70%

TABLE II: A performance analysis of our 4vs4-SIMDop using Intel® VTune™ for Nao.

VI. Conclusions And Future Work

We have presented two versions for a SIMD optimized bounding volume hierarchy for simultaneous BVH traversal. The main idea is to use higher n-ary trees instead of classical binary trees. We have presented several new heuristics for the top-down construction of such tree data structures with higher branching factor.

The BNG-based method performs best. Even if we tested only up to 16-ary trees, the clustering-based construction is already prepared to support higher branching factors following future SIMD developments. Our results show that, depending on the object, our SIMDop BVHs outperform traditional BVHs by more than an order of magnitude.

Our approach also opens up several directions for future work. For instance, we would like to include magnification control to the BNG construction algorithm. Moreover, other clustering algorithms than BNG could be considered. In this work, we relied on DOPs as BVs because of a fair comparison with the manual optimized traversal scheme. However, we would like to investigate also other BV types that do not have the problem of the later escape of the for-loop. Also the influence of the number of orientations for the DOPs requires further investigations. Finally, probably other applications using BVHs like ray tracing or occlusion computations could benefit from our SIMDop BVHs, too.

Acknowledgment

The research reported in this paper has been (partially) supported by the German Research Foundation DFG, as part of Collaborative Research Center (Sonderforschungsbereich) 1320 “EASE - Everyday Activity Science and Engineering”, University of Bremen (http://www.ease-crc.org/). The research was conducted in subproject R03 <Embodied simulation-enabled reasoning>.

References

[1] M. Reggiani, M. Mazzoli, and S. Caselli, “An experimental evaluation of collision detection packages for robot motion planning,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, Sept 2002, pp. 2329–2334 vol.3.

[2] G. Zachmann, “Rapid collision detection by dynamically aligned dop-trees,” in *Proceedings of the Virtual Reality Annual International Symposium*, ser. VRAIS ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 90–. [Online]. Available: http://dl.acm.org/citation.cfm?id=522258.836122

[3] S. Trenkel, R. Weller, and G. Zachmann, “A benchmarking suite for static collision detection algorithms,” in *International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, V. Skala, Ed. Plzen, Czech Republic: Union Agency, 29 January–1 February 2007. [Online]. Available: http://cg.in.tu-clausthal.de/research/colldet_benchmark

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

[5] G. 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=763345.763346

[6] G. Zachmann, "Minimal hierarchical collision detection," in *Proceedings of the ACM symposium on Virtual reality software and technology*, ser. VRST '02. New York, NY, USA: ACM, 2002, pp. 121–128. [Online]. Available: http://doi.acm.org/10.1145/585740.585761

[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, Jan. 1998. [Online]. Available: http://dx.doi.org/10.1109/2945.675649

[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*, ser. SIGGRAPH '96. New York, NY, USA: ACM, 1996, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/237170.237244

[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] S. Krishnan, A. Pattekar, M. C. Lin, and D. Manocha, "Spherical shell: a higher order bounding volume for fast proximity queries," in *Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics : the algorithmic perspective: the algorithmic perspective*, ser. WAFR '98. Natick, MA, USA: A. K. Peters, Ltd., 1998, pp. 177–190. [Online]. Available: http://dl.acm.org/citation.cfm?id=298960.299006

[11] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast proximity queries with swept sphere volumes," Nov. 14 1999. [Online]. Available: http://citeseer.ist.psu.edu/408975.html;ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/ssv.ps

[12] T. Larsson and T. Akenine-Möller, "Bounding volume hierarchies of slab cut balls." *Comput. Graph. Forum*, vol. 28, no. 8, pp. 2379–2395, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/cgf/cgf28.html#LarssonA09

[13] T. He, "Fast collision detection using quospo trees," in *Proceedings of the 1999 symposium on Interactive 3D graphics*, ser. I3D '99. New York, NY, USA: ACM, 1999, pp. 55–62. [Online]. Available: http://doi.acm.org/10.1145/300523.300529

[14] S. Krishnan, M. Gopi, M. Lin, D. Manocha, and A. Pattekar, "Rapid and accurate contact determination between spline models using shelltrees," 1998.

[15] U. Schwesinger, R. Siegwart, and P. Furgale, "Fast collision detection through bounding volume hierarchies in workspace-time space for sampling-based motion planners," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 63–68.

[16] J. Pan, S. Chitta, and D. Manocha, "Fcl: A general purpose library for collision and proximity queries," in *2012 IEEE International Conference on Robotics and Automation*, May 2012, pp. 3859–3866.

[17] D. Ferguson, M. Darms, C. Urmson, and S. Kolski, "Detection, prediction, and avoidance of dynamic obstacles in urban environments," in *2008 IEEE Intelligent Vehicles Symposium*, June 2008, pp. 1149–1154.

[18] N. Roussopoulos and D. Leifker, "Direct spatial search on pictorial databases using packed r-trees," in *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '85. New York, NY, USA: ACM, 1985, pp. 17–31. [Online]. Available: http://doi.acm.org/10.1145/318898.318900

[19] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez, "Str: A simple and efficient algorithm for r-tree packing," Institute for Computer Applications in Science and Engineering (ICASE), Tech. Rep., 1997.

[20] J. Goldsmith and J. Salmon, "Automatic creation of object hierarchies for ray tracing," *IEEE Comput. Graph. Appl.*, vol. 7, no. 5, pp. 14–20, May 1987. [Online]. Available: http://dx.doi.org/10.1109/MCG.1987.276983

[21] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," *Symposium on Interactive Ray Tracing*, vol. 0, pp. 61–69, 2006.

[22] I. Wald, "On fast construction of sah-based bounding volume hierarchies," in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, ser. RT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 33–40. [Online]. Available: http://dx.doi.org/10.1109/RT.2007.4342588

[23] C. Lauterbach, M. Garland, S. Sengupta, D. P. Luebke, and D. Manocha, "Fast bvh construction on gpus." *Computer Graphics Forum*, vol. 28, no. 2, pp. 375–384, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/cgf/cgf28.html#LauterbachGSLM09

[24] G. Zachmann, "Virtual reality in assembly simulation – collision detection, simulation algorithms, and interaction techniques," Dissertation, Darmstadt University of Technology, Germany, May 2000.

[25] G. Zachmann and E. Langetepe, "Geometric data structures for computer graphics," in *Proc. of ACM SIGGRAPH*. ACM Transactions of Graphics, 27–31July 2003. [Online]. Available: http://www.gabrielzachmann.org/

[26] J. Mezger, S. Kimmerle, and O. Etzmuß, "Hierarchical Techniques in Collision Detection for Cloth Animation," *Journal of WSCG*, vol. 11, no. 2, pp. 322–329, 2003.

[27] P. Agarwal, L. Guibas, A. Nguyen, D. Russel, and L. Zhang, "Collision detection for deforming necklaces," *Computational Geometry: Theory and Applications*, vol. 28, pp. 137–163, 2004.

[28] R. Weller, D. Mainzer, A. Srinivas, M. Teschner, and G. Zachmann, "Massively parallel batch neural gas for bounding volume hierarchy construction," in *Virtual Reality Interactions and Physical Simulations (VRIPhys)*. Bremen, Germany: Eurographics Association, Sept. 2014.

[29] M. Reggiani, M. Mazzoli, and S. Caselli, "An experimental evaluation of collision detection packages for robot motion planning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, Sep. 2002, pp. 2329–2334 vol.3.