Parallel Construction of Bounding Volumes

Mattias Karlsson, Olov Winberg and Thomas Larsson

Mälardalen University, Sweden

Abstract

This paper presents techniques for speeding up commonly used algorithms for bounding volume construction using Intel's SIMD SSE instructions. A case study is presented, which shows that speed-ups between 7–9 can be reached in the computation of k-DOPs. For the computation of tight fitting spheres, a speed-up factor of approximately 4 is obtained. In addition, it is shown how multi-core CPUs can be used to speed up the algorithms further.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and techniques—Graphics data structures and data types

1. Introduction

A bounding volume (BV) is a shape that encloses a set of geometric primitives. Usually, simple convex shapes are used as BVs, such as spheres and boxes. Ideally, the computation of the BV dimensions results in a minimum volume (or area) shape. The purpose of the BV is to provide a simple approximation of a more complicated shape, which can be used to speed up geometric queries. In computer graphics, BVs are used extensively to accelerate, e.g., view frustum culling, occlusion queries, selection (picking), ray tracing, path planning, and collision detection.

To speed up the computations, several attractive architectural features of modern processors can be exploited [GBST06]. In particular, single instruction mutliple data (SIMD) computation units offer an efficent mechanism to exploit fine-grained parallelism [Bik04, HOM08]. SIMD computations are well-suited to speed-up various types of multimedia and geometry processing operations [YSL98, MY02, LAML07]. In ray tracing, ray packets and/or groups of BVs can be processed simultaneously using SIMD [WBS07, DHK07].

Here we present our results from turning various sequential BV construction algorithms into vectorized methods by using Intel's Streaming SIMD Extension (SSE). We focus on three of the most widely used BV types: the axis-aligned bounding box (AABB), the discrete orientation polytope (k-DOP), and the sphere. In particular, these types of volumes are suitable for dynamic scenes, since they are fast to recompute [MKE03, LAM06]. As Sections 2–4 show, the SIMD vectorization of the algorithms leads to generous speed-ups, despite that the SSE registers are only four floats wide. In addition, the algorithms can be parallelized further by exploiting multi-core processors, as shown in Section 5.

2. Fast SIMD computation of k-DOPs

A k-DOP is a convex polytope enclosing another object such as a complex polygon mesh [KHM*98]. The polytope is constructed by finding $k/2$ pairs of parallel planes (slabs) that tightly surrounds the object. Implicitly, these planes define a bounding polytope covering the object. If needed, the enclosing polytope can be extracted by computing the intersection of the half-spaces bounded by the planes.

To construct a k-DOP, the minimum and maximum projection values of the n input points are computed for each slab direction, which are given by a pre-determined and fixed normal set N with $k/2$ normals. An efficient choice is to derive the normals from the unit cube. The three face normals are used to define an AABB (6-DOP). Then four "corner" normals can be added to define a 14-DOP, and six more "edge" normals to define a 26-DOP. Since all these normals have components in the set $\{0, \pm 1\}$, it is possible to reduce the cost of the projection operation.

The SSE instructions operate on registers containing four 32-bit precision floating point numbers. To utilize the full width of these registers, the 3D input points are first rearranged, or swizzled, into a set G of $m \approx n/4$ vertex groups

G_0	G_1	...	G_{m-1}
$X_0 = \{x_0, x_1, x_2, x_3\}$	$X_1 = \{x_4, x_5, x_6, x_7\}$	...	$X_{m-1} = \{x_{n-4}, x_{n-3}, x_{n-2}, x_{n-1}\}$
$Y_0 = \{y_0, y_1, y_2, y_3\}$	$Y_1 = \{y_4, y_5, y_6, y_7\}$	...	$Y_{m-1} = \{y_{n-4}, y_{n-3}, y_{n-2}, y_{n-1}\}$
$Z_0 = \{z_0, z_1, z_2, z_3\}$	$Z_1 = \{z_4, z_5, z_6, z_7\}$	...	$Z_{m-1} = \{z_{n-4}, z_{n-3}, z_{n-2}, z_{n-1}\}$

Figure 1: *SoA vertex layout for efficient SIMD processing. The n input vertices are stored in $m = n/4$ vertex groups. Each group G_i holds four vertices stored as 3 arrays X_i, Y_i, and Z_i.*

FIND-k-DOP_SSE
input: $G = \{G_0, G_1, \ldots, G_{m-1}\}$, $N = \{\mathbf{n}_0, \mathbf{n}_1, \ldots, \mathbf{n}_{k/2-1}\}$
output: $D = \{S, L\} = \{\{s_0, s_1, \ldots, s_{k/2-1}\}, \{l_0, l_1, \ldots, l_{k/2-1}\}\}$

1. **for each** $\mathbf{n}_j \in N$
2. $\quad P \leftarrow$ PROJECT$(G_0, \mathbf{n}_j)$
3. $\quad S_j \leftarrow P$
4. $\quad L_j \leftarrow P$
5. **for** $i = 1$ **to** $m-1$
6. $\quad$ **for each** $\mathbf{n}_j \in N$
7. $\quad\quad P \leftarrow$ PROJECT$(G_i, \mathbf{n}_j)$
8. $\quad\quad S_j \leftarrow \text{minps}(S_j, P)$
9. $\quad\quad L_j \leftarrow \text{maxps}(L_j, P)$
10. **for** $j = 0$ **to** $k/2-1$
11. $\quad s_j \leftarrow \min(S_j)$
12. $\quad l_j \leftarrow \max(L_j)$
13. $\quad s_j \leftarrow s_j / \|\mathbf{n}_j\|$
14. $\quad l_j \leftarrow l_j / \|\mathbf{n}_j\|$

Figure 2: *Data-parallel computation of a k-DOP. G is the set of SoA-arranged vertex groups and N is the set of normals or slab directions. The output $D = \{S, L\}$ is the set of intervals defining the size of the k-DOP along the slab directions.*

stored in an array with 16-byte alignment. A single vertex group $G_i = \{X_i, Y_i, Z_i\}$ holds four vertices stored in three arrays, one for each coordinate axis. In Figure 1, this vertex data structure is shown. Note that this representation is often referred to as Structure of Arrays (SoA) [GBST06]. Arrays of four floats are hereafter called 4-tuples.

Pseudocode for the computation of a general k-DOP is given in Figure 2. First all 4-tuples S_j and L_j holding the potential extremal projection values are initialized (Lines 1–4). Then the remaining vertex groups are iterated. For each group and normal, the four projection values P along the normals are computed (Lines 5–7). By using minps and maxps instructions, the current extremal values are updated without introducing branches (Lines 8–9). Finally, the extremal scalar values s_j and l_j are extracted from the resulting 4-tuples S_j and L_j, and they are also scaled to compensate for the use of non-unit length normals in N (Lines 10-14).

Note that when implementing this algorithm, the inner loop (Lines 6–9) is unrolled and each projection calculation is optimized. As an example, given the normal $\mathbf{n} = (1, 0, -1)$, and a vertex group G_i, the computation of the projections reduces to $P = X_i - Z_i$, i.e., four vertices are projected using only one parallel subtraction.

3. Fast SIMD computation of spheres

To compute tight fitting bounding spheres efficiently, the EPOS algorithm has been proposed [Lar08]. EPOS is a two-pass algorithm where, in the first pass, an initial tentative sphere is computed from a few selected extremal points of a model. The selection of these points is made by computing a k-DOP, while also storing the actual points that give rise to the minimum and maximum projection values for each slab direction. A simple extension of the algorithm in Figure 2 accomplishes this. For example, to get the appropriate indices A and B of the points with current minimum and maximum projection values, the following code replaces Lines 8 and 9:

8.1 $M = \text{cmpltps}(P, S_j)$
8.2 **if** $\text{movmsk}(M) > 0$ **then**
8.3 $\quad I = \text{andps}(C_i, M)$
8.4 $\quad A_j = \text{maxps}(I, A_j)$
8.5 $\quad S_j \leftarrow \text{minps}(S_j, P)$
9.1 $M = \text{cmpgtps}(P, S_j)$
9.2 **if** $\text{movmsk}(M) > 0$ **then**
9.3 $\quad I = \text{andps}(C_i, M)$
9.4 $\quad B_j = \text{maxps}(I, B_j)$
9.5 $\quad L_j \leftarrow \text{maxps}(L_j, P)$

With the instruction cmpltps a bit-mask M indicating the smaller projection values in P is created (Line 8.1). C_i contains indices of the vertices in group G_i. A bitwise *and* operation on the mask M and C_i, and the following maxps operation will give the new indices A_j for the current extremal points. In fact, this method eliminates the need for branching completely, since the if-statement on Line 8.2 can be removed without altering the results. However, experimental tests indicated that keeping the if-statement controlling the mask for changes gave faster execution times. The indices of the maximum points are found similarily (Lines 9.1–9-5).

After the retrieval of the k extremal points, the initial tentative sphere is computed using an exact solver which gives the smallest enclosing sphere of the selected points. Then in the second pass of the algorithm, all points of the model are considered again and the sphere is incrementally grown for each point that compromise the current bounding

EXPANDSPHERE_SSE
input: $G = \{G_0, G_1, \ldots, G_{m-1}\}, \mathbf{c}, r$
output: $\mathbf{c}, r$

1. **for each** $G_i \in G$
2. $\quad D \leftarrow$ GETSQUAREDDISTANCES$(G_i, \mathbf{c})$
3. $\quad M \leftarrow \text{cmpgtps}(D, r^2)$
4. $\quad$ **if** $\text{movmsk}(M) > 0$ **then**
5. $\quad\quad \mathbf{c}, r \leftarrow$ UPDATESPHERE$(M, D, G_i, \mathbf{c}, r)$

Figure 3: *Data-parallel algorithm that expands the initially computed sphere when necessary.*

sphere, as shown by the pseudocode in Figure 3. In Line 2, the squared distances D from the current centre to the four current points G_i are calculated. Line 3 returns the mask M indicating points outside the current sphere. The branch in Line 4 handles the case when the sphere has to be updated. The actual updating has some data dependencies, since four points are checked against the current sphere in parallel, and each point in the vertex group that is found to be outside the current sphere leads to an update of both the centre and the radius of the current sphere. This is handled by the function in Line 5.

Triangle Mesh	n_v	n_t
Frog	4010	7964
Chair	7260	14372
Tiger	30892	61766
Bunny	32875	65536
Horse	48485	96966
Golfball	100722	201440
Hand	327323	654666

Table 1: *The number of vertices n_v and triangles n_t for the polygonal meshes used for benchmarking.*

4. Results

A PC with an Intel Core 2 Quad Q8200 CPU, 2.33 GHz with 4GB of RAM running Windows 7, 32-bit version (x86), was used to measure the execution times of the presented methods. The algorithms were implemented in C++ and compiled under Microsoft Visual Studio 2008 using the default release mode. All algorithms discussed in this section were run single-threaded.

For benchmark purposes a test suit executing all algorithms were used. Each algorithm was executed for seven different polygon meshes. These meshes are visualized inside the different types of computed BVs in Figure 4. The number of points for each model is listed in Table 1. Time was measured by calculating the average execution time over 20 repetitions, executed after an initial "warm-up run" to avoid possible start-up artefacts. In this way, each algorithm has the same potential to benefit from the CPU cache and models with fewer points are likely to benefit the most. A real world application will probably not benefit from the cache to the same extent, but as the cache hierarchies in modern computers are complex, it seems not trivial to clear the cache in a comprehensive manner before each algorithm run.

As seen in Table 2, data-parallel computations of k-DOPs at the instruction level improves the computation time of the corresponding sequential algorithms significantly by a factor of 7–9. Clearly, the obtained speedups are better than the ideal speedup of 4 for 4 floating point wide SIMD registers. One probable reason is the branch elimination provided by the minps and maxps instructions in the vectorized solution.

A peculiar exception to the obtained high performance, however, is the AABB computation of the hand model, where a speed-up factor of "only" 4.4 is reached. The degraded speed-up in this case appears to have a connection to L2 cache misses on the used test machine.

The computation times of the spheres given in Table 3 show less improvement and reaches a quite consistent speedup factor of approximately 4. The reasons are due to data dependencies and the extra computation needed to retrieve the index of each extremal point spanning the k-DOP planes. Besides the EPOS algorithm, Ritter's algorithm [Rit90] and a developed SIMD version of it are also included for comparison. Ritter's method is very fast, but it consistently computes larger spheres. Also, note that the EPOS-6 method using SSE seems to reach very close to the high performance of the SSE version of Ritter's method.

The quality of the computed spheres is given in Table 4. The radius differences between the corresponding sequential and SSE methods derive from the fact that several vertices can have identical projections and, therefore, the vertices selected as extremal may vary across different implementations of the algorithms.

5. Multi-core parallelization

By utilizing multi-core processors in addition to SIMD computation another level of parallelism can be exploited. This is straightforward when computing AABBs and k-DOPs, since each thread can compute an optimal sub-volume of a subset of the input points, and then after all points have been processed all the sub-volumes can be merged sequentially into the final optimal BV. The results of using this strategy for computing 26-DOPs are given in Table 5.

Multi-threading was implemented by using OpenMP and the previous mentioned quad-core machine was used. As can be seen, for the five last input models with more than 30000 points, this gives a further speed-up in the range 1.0-1.8 using 2 cores and between 2.2–3.5 using 4 cores. However, since there is a start-up cost for multi-threading, the computation of the 26-DOP becomes slower for the two simplest

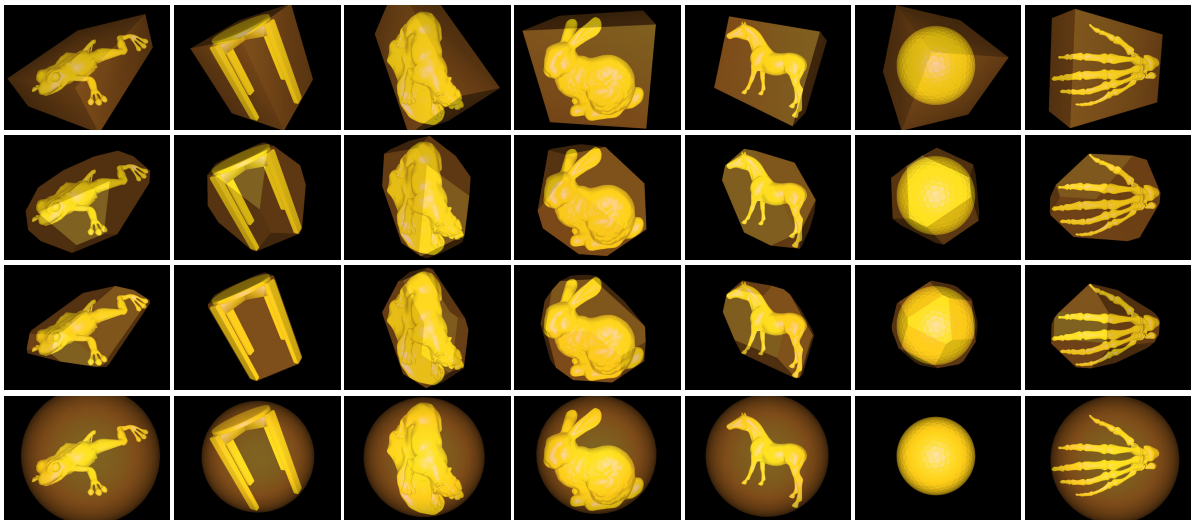

Figure 4: *Visualizations of the polygon meshes used as data sets and the resulting AABBs (row 1), 14-DOPs (row 2), 26-DOPs (row 3), and spheres (row 4). The spheres were computed using EPOS-26.*

	FindAABB			**Find-14-DOP**			**Find-26-DOP**		
Point set	Seq	SSE	*S*	Seq	SSE	*S*	Seq	SSE	*S*
Frog	0.028	0.003	9.11	0.088	0.010	9.23	0.160	0.018	8.78
Chair	0.048	0.006	8.68	0.142	0.017	8.25	0.257	0.032	7.97
Tiger	0.205	0.024	8.64	0.595	0.073	8.20	1.079	0.138	7.84
Bunny	0.217	0.025	8.62	0.634	0.077	8.27	1.152	0.145	7.95
Horse	0.330	0.038	8.72	0.965	0.114	8.49	1.755	0.214	8.20
Golfball	0.670	0.080	8.34	1.956	0.240	8.17	3.548	0.448	7.93
Hand	2.260	0.514	4.40	6.482	0.910	7.12	11.721	1.588	7.38

Table 2: *AABB and k-DOP execution times in ms and speed-ups S.*

	EPOS-6			**EPOS-14**			**EPOS-26**			**Ritter**		
Point set	Seq	SSE	*S*	Seq	SSE	*S*	Seq	SSE	*S*	Seq	SSE	*S*
Frog	0.071	0.021	3.32	0.132	0.036	3.68	0.207	0.056	3.69	0.053	0.019	2.78
Chair	0.120	0.033	3.67	0.223	0.056	4.00	0.338	0.091	3.73	0.088	0.029	3.02
Tiger	0.491	0.114	4.31	0.889	0.190	4.67	1.384	0.299	4.63	0.365	0.111	3.30
Bunny	0.524	0.123	4.25	0.949	0.203	4.67	1.474	0.317	4.65	0.389	0.119	3.27
Horse	0.786	0.190	4.13	1.428	0.324	4.40	2.227	0.509	4.37	0.595	0.198	3.00
Golfball	1.607	0.367	4.38	2.915	0.616	4.73	4.538	0.958	4.74	1.208	0.365	3.30
Hand	5.383	1.444	3.73	9.668	2.245	4.31	15.047	3.339	4.51	4.055	1.442	2.81

Table 3: *Sphere execution times in ms and speed-ups S.*

		EPOS-6		**EPOS-14**		**EPOS-26**		**Ritter**	
Point set	Optimal	Seq	SSE	Seq	SSE	Seq	SSE	Seq	SSE
Frog	0.59903	0.61349	0.61349	0.60019	0.60019	0.59903	0.59903	0.65965	0.65040
Chair	0.63776	0.69474	0.68974	0.64359	0.64359	0.63792	0.63793	0.73014	0.72789
Tiger	0.51397	0.52531	0.52531	0.51507	0.51507	0.51507	0.51507	0.53835	0.53835
Bunny	0.64321	0.65017	0.65017	0.64423	0.64423	0.64415	0.64415	0.67694	0.67694
Horse	0.62897	0.63023	0.63023	0.62899	0.62899	0.62897	0.62897	0.63476	0.63476
Golfball	0.50110	0.50155	0.50154	0.50145	0.50145	0.50114	0.50114	0.51531	0.50350
Hand	0.52948	0.52949	0.52951	0.52949	0.52951	0.52949	0.52950	0.52949	0.52951

Table 4: *The optimal radius and the radii computed by the tested bounding sphere algorithms for each point set.*

Point set	1 core	2 cores	4 cores
Frog	0.018	0.079	0.030
Chair	0.032	0.086	0.037
Tiger	0.138	0.129	0.060
Bunny	0.145	0.143	0.065
Horse	0.214	0.178	0.081
Golfball	0.448	0.297	0.140
Hand	1.588	0.897	0.452

Table 5: *Parallel computation of 26-DOPs using both multiple cores and SSE. Execution times are in ms.*

models. Similar results were obtained when multi-threading was used in the computation of AABBs and 14-DOPs, but since these algorithms involves less work per iteration in the main loop, the start-up cost affected the algorithms slightly more. Therefore, for simple points sets, a scheduling mechanism is needed that can divide the work of computing several BVs at a time among the available cores.

Creating a multi-threaded version of the EPOS-algorithm requires some additional thought. The vertices spanning the slabs of a k-DOP are easily found by a simple extension of the multi-threaded k-DOP computation algorithm. But it is not clear how to best grow the sphere in the last phase of the algorithm efficiently, since it involves updating both the center point and radius of the sphere which introduces data dependecies among the threads. The simplest approach seems to be to keep the initially determined centre point of the sphere stationary, thereby changing the quality of the computed spheres slightly. In this way, the multi-threaded computation of the required radius in the last pass of the algorithm becomes efficient and straightforward. Another simple solution would be to search for all points violating the initially computed sphere using multi-threaded loop parallelization, but instead of updating the center and radius of the sphere repeatedly during the search, the points are only stored in a list. Then the necessary updates of the sphere can proceed afterwards by processing this list sequentially.

6. Conclusions and future work

Clearly, the presented algorithms speed up the BV computation substantially. Since our BV computation scheme generalizes trivially to wider SIMD registers, further significant speed-ups can be expected for future CPUs. For example, the Intel Advanced Vector Extensions (AVX) is supported in forthcoming Intel 64 processors. AVX provides support for 256-bit wide SIMD registers, which means eight 32-bit precision floating point numbers can be processed in parallel. Similarly, the number of cores per CPU will continue to increase and this will also benefit the presented algorithms.

There are several areas worth investigating further. For example, it is expected that parallel BV computation methods can be utilized to make bounding volume hierarchy (BVH) construction faster. Many top-down BVH construction methods repeatedly call a BV computation algorithm for various subsets of the input points. Interestingly, Wald et al presents a parallel BVH building method targeting modern multi-core architectures [WIP08]. As an alternative to CPU-based approaches, it would also be interesting to develop highly parallel GPU-based algorithms for BV and BVH computation, since GPUs offer more concurrency and higher bandwith [LGS*09].

References

[Bik04] BIK A. J. C.: *The Software Vectorization Handbook*. Intel Press, 2004. 1

[DHK07] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (2007), 1225–1233. 1

[GBST06] GERBER R., BIK A. J. C., SMITH K. B., TIAN X.: *The Software Optimization Cookbook, 2nd Ed.* Intel Press, 2006. 1, 2

[HOM08] HASSABALLAH M., OMRAN S., MAHDY Y. B.: A review of SIMD multimedia extensions and their usage in scientific and engineering applications. *The Computer Journal 51*, 6 (2008), 630–649. 1

[KHM*98] KLOSOWSKI J. T., HELD M., MITCHELL J. S. B., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 21–36. 1

[LAM06] LARSSON T., AKENINE-MÖLLER T.: A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics 30*, 3 (2006), 450–459. 1

[LAML07] LARSSON T., AKENINE-MÖLLER T., LENGYEL E.: On faster sphere-box overlap testing. *journal of graphics tools 12*, 1 (2007), 3–8. 1

[Lar08] LARSSON T.: Fast and tight fitting bounding spheres. In *Proceedings of The Annual SIGRAD Conference* (November 2008), Linköping University Electronic Press, pp. 27–30. 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009). 5

[MKE03] MEZGER J., KIMMERLE S., ETZMUSS O.: Hierarchical techniques in collision detection for cloth animation. *Journal of WSCG 11* (2003), 322–329. 1

[MY02] MA W.-C., YANG C.-L.: Using intel streaming SIMD extensions for 3D geometry processing. In *PCM '02: Proceedings of the Third IEEE Pacific Rim Conference on Multimedia* (2002), Springer-Verlag, pp. 1080–1087. 1

[Rit90] RITTER J.: An efficient bounding sphere. In *Graphics Gems*, Glassner A., (Ed.). Academic Press, 1990, pp. 301–303. 3

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007). 1

[WIP08] WALD I., IZE T., PARKER S. G.: Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics 32*, 1 (2008), 3–13. 5

[YSL98] YANG C.-L., SANO B., LEBECK A. R.: Exploiting instruction level parallelism in geometry processing for three dimensional graphics applications. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (1998), pp. 14–24. 1