# 3D Vector Normalization Using 256-Bit Intel® Advanced Vector Extensions (Intel® AVX)

**Stan Melax**

## Objective

Show developers how to improve the performance of geometry computations by transposing packed 3D data on-the-fly to take advantage of 8-wide SIMD processing.

## Abstract

This article shows how to exploit 256-bit Intel® Advanced Vector Extensions (Intel® AVX) to normalize an array of 3D vectors. We describe a shuffle approach to convert between AOS and SOA on-the-fly in order to make data ready for up to 8-wide single instruction multiple data (SIMD) processing. The 8x3 to 3x8 transpose can be done in 5 shuffles, and the 3x8 to 8x3 transpose back takes 6 shuffles. Results demonstrate the benefit of wider SIMD on the normalization computation with 2.3 and 2.9 times speedups on 128-bit and 256-bit respectively. The round trip cost, 11 extra instructions, of enabling SOA processing is low enough to justify its usage on this small computation.

## Introduction

Many interactive applications do a lot of geometry processing of one form or another on the CPU. While structure of array (SOA) memory layout is most efficient for processing, in some applications it may not be practical to store data this way. Quite often there is performance-critical code that operates on a regular array of 3D vectors – a very common data structure found in 3D applications. A common example is making an array of 3D normal vectors to all be of unit length. In this article, we begin with this function and optimize the code to exploit the capabilities of the x86 processor, including new 256-bit instructions from the Intel® Advanced Vector Extensions (Intel® AVX), which are part of the new microarchitecture of the second-generation Intel® Core™ processor family, codenamed Sandy Bridge.

The C code for the loop being optimized is shown below:

```
void Normalize(float V[][3],int N)
{
  for(int i=0;i != N;i++)
  {
      float *v=V[i];
      float invmag = 1.0f/sqrtf(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
      v[0] *= invmag;
      v[1] *= invmag;
      v[2] *= invmag;
  }
}
```

In practice, such a routine may be implemented using a C++ 3D vector class with overloaded operators, but the underlying operations and data layout would be the same.

Usage of single instruction multiple data (SIMD) processing with 4-wide 32-bit precision floating point has become widespread since the introduction of the Intel® Streaming SIMD Extensions 2 (SSE2) in 2001. Because the 128-bit register size is a natural fit for 4D data, many 3D interactive applications will use homogeneous vectors, or pad 3D vectors with an extra 32 bits. While an easy way to exploit SIMD processing the gains are rarely a 4x improvement due to other performance bottlenecks. Applying this style of SIMD to our normalization example would only use ¾ of a 128-bit register. While the multiplications can be done in parallel, summing the squares and the inverse square root do not benefit from SIMD. Most importantly, this programming pattern does not scale to wider SIMD such as 256-bit Intel Advanced Vector eXtensions (AVX) that support parallel operations on 8 floating point numbers.

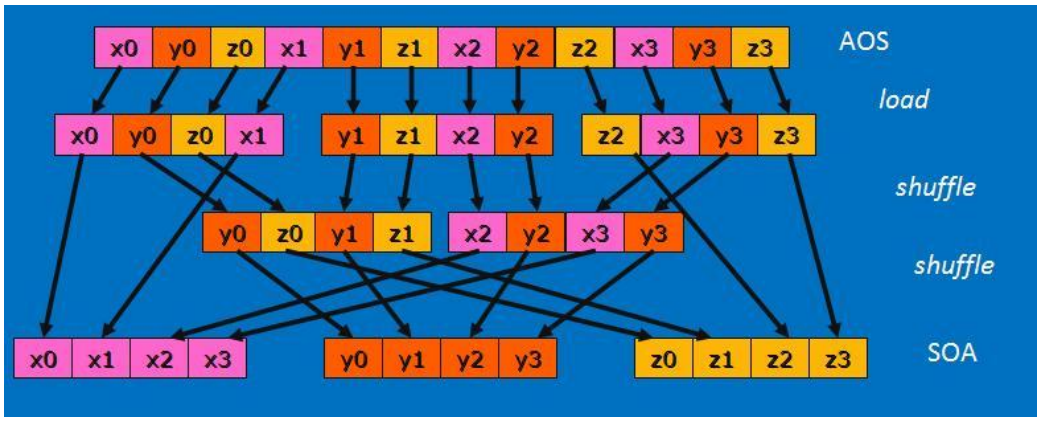
Another approach to exploit SIMD is to process 8 iterations of the loop at a time, i.e., normalize 8 vectors at once. This requires the data to be rearranged so all the X components of the 8 vectors fit into one register, the Y's in the next, and Z's in another. Loading the registers would be easy if our data happened to be stored this way in a structure of arrays (SOA). However, some applications do require the data storage to be a packed sequence of 3D vectors – an array of structures (AOS). Therefore, to utilize 8-wide SIMD processing, we need to transpose the data on-the-fly, do the computation, and transpose back. This article describes how to do the transpose using shuffles on the x86, followed by provides performance results, including both optimized serial and shuffle transpose implementations, that shows the speedup that can be obtained using Intel AVX for normalizing an (AOS) array of 3D vectors.

## The 128-bit and 256-bit AOS to SOA shuffle

The most efficient way to move data from the first level cache into registers is to load 128 bits (or more) at a time. This is a different stride than our packed array of 3D values. However, the pattern of 128-bit alignment repeats for every fourth vector (or 12 floats). Therefore, three aligned 128-bit loads pull the next four 3D vectors into three 128-bit registers.

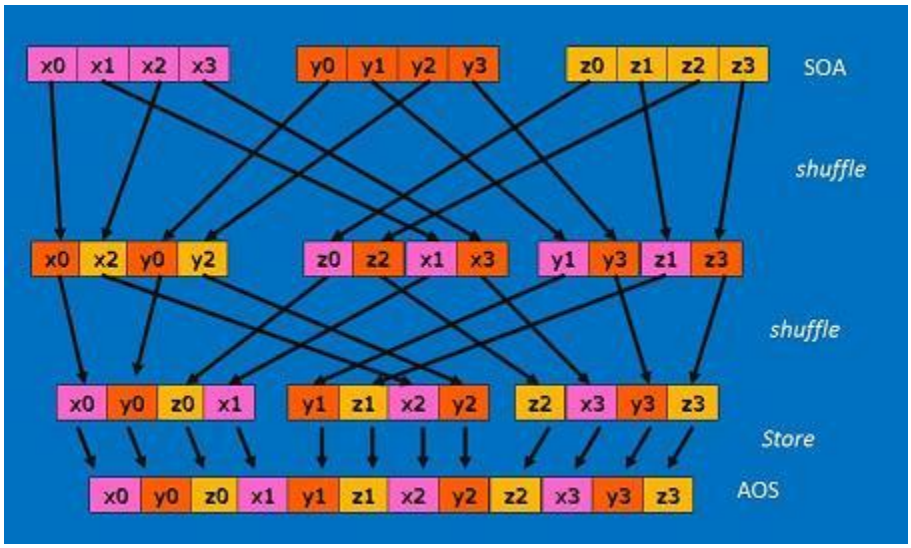
The order of the vector elements is not very useful right after the three loads. Next we need to get the data from four 3D vectors into a usable form by grouping the X's, Y's and Z's into separate registers. The following figure shows the 4x3 to 3x4 transpose using five

Intel AVX 128-bit shuffles.

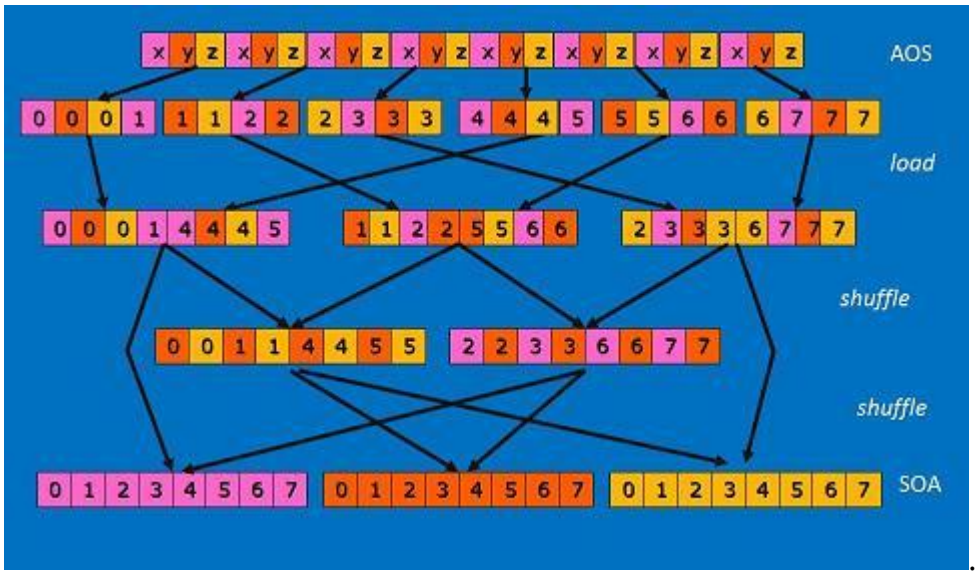



With the data now in SOA form, the computation is done with the same steps as the serial implementation but using the Intel AVX instructions to normalize 4 vectors at a time.

The result of the computation will be in SOA form and needs to be transposed back into AOS. Our conversion operations are not symmetrical. In other words, the transpose back from SOA to AOS cannot be done with the same code. In fact, it takes one more shuffle to convert back. The technique is illustrated in the following figure.




So far we have described 4 float or 128-bit SIMD usage. We can extend this to 8x3 by using the full 256-bit registers and instructions provided by Intel AVX. When shuffles are involved, the way to think about 256-bit Intel AVX is that there are two lanes that are 128-bit each. In other words, AVX provides two lanes of 4 wide SIMD units. As before, we load the first 12 floats into three 128-bit registers which actually are the lower halves of three 256-bit registers. All we have to do now is load the next 12 floats (or next four 3D vectors) into the upper halves of those same three 256-bit registers. Then the shuffle computation is implemented exactly the same way, but using the 256-bit versions of each of the instructions/intrinsics.




The 256 bit conversion back to AOS in 6 shuffles and subsequent stores are similarly an extension of the 128-bit version above. With the ability to transpose data into SOA and back, we are now ready to apply this technique to normalizing an array of 3D vectors.

## Testing and Results

For our testing, we used a single core from an Intel® processor-based system with the microarchitecture codename Sandy Bridge that supports Intel AVX. To focus on instruction costs, L1 caches were primed with the data for 1024 3D vectors. The RDTSC instruction was called before and after each loop to measure the total number of cycles to process the entire dataset. In the tables below, we divide by the size of the array and show throughput results in CPU cycles per 3D vector processed. In addition to the various normalization implementations, we also measure loops with parts removed to gauge the cost of various steps in isolation.

The following table shows the cost per 3D vector of the round trip from AOS to SOA and back to AOS for 3D data on SIMD 128-bit (4 float) and 256-bit (8 float).

| **Transpose Only Round Trip** | **Cycles Per Vector Processed** | **Description** |
|---|---|---|
| shuf_trans128: | 3.0 | 128-bit AOS->SOA->AOS via shuffles |
| shuf_trans256: | 1.5 | 256-bit transpose and back |

The values in the table are cycles per 3D vector. Note that there is more than one vector being transposed per loop iteration. In both cases, the loop iteration takes 12 cycles which is reasonable since there are 11 shuffles per loop. The meaningful result is that the cost is three or fewer cycles per vector for the round trip conversion from our packed array of structures to a structure of arrays and back.

For the next test, we wanted to gauge what the cost would be if there was no transpose required, so we made another data structure where the data was already prearranged in SOA format, i.e., three separate arrays for all the X's, Y's, and Z's. The following table shows the cost of normalizing 3D vectors already in SOA format.

| SIMD Math Only Tests | Cycles Per Vector Normalized | Description |
|---|---|---|
| norm_soa_data4: | 1.8 | 128-bit intrinsics norm on SOA data |
| norm_soa_data8: | 1.1 | 256-bit intrinsics norm on SOA data |

Results are presented for both 128-bit and 256-bit SIMD normalization implementations on preformatted SOA data. Because the cost of shuffling data is not free, it is doubtful that normalizing the AOS data could be as fast as these times. This is meant for reference and provides an indication of the absolute performance limit for just the mathematical computation for our loop.

Note that normalization is using the low level of numerical accuracy. The result of the approximate inverse square root assembly routine is not refined with Newton-Raphson. For some applications, such as generating mesh normals for lighting calculations, low accuracy may be considered sufficient. The goal of these tests is to evaluate the potential benefit for the transpose on-the-fly technique for very small computations.

The cost of normalizing our regular array of 3D vectors (AOS dataset) is shown in the following table:

| 3D Vector Normalization | Cycles Per Vector Normalized | Description |
|---|---|---|
| x87: | 45.0 | serial function without any Intel® SSE or Intel® AVX |
| arch_optim: | 24.3 | serial normalization (no SIMD) with -arch optimization |
| rsqrt_ss: | 8.0 | serial vector normalization using RSQRT assembly |
| mask_trans: | 9.4 | AOS->SOA->AOS uses masking for 4x3 trans |
| shuf_trans4: | 3.5 | 4 float (128 bit) SOA AOS conversion with shuffles |
| shuf_trans8: | 2.7 | 8 float (256 bit) 8x3 transpose with shuffles |

The results are shown for a variety of implementations. To get a fair basis for comparison, we wrote, using hand-picked assembly instructions, an optimal serial implementation, with less accuracy, that averages 8 cycles per loop or 8 cycles per vector normalization.

Also for interest, we compared with another 128-bit AOS->SOA->AOS programming pattern that uses masks and bitwise logical operations to implement the transpose. On current hardware, this method is slower than the serial implementation.

Using Intel AVX's 256 bit SIMD, a regular 3D array of vectors are normalized with a 2.7 cycle throughput. In other words, the SIMD implementation using shuffles performs better than the best serial implementation. Note the cost is not exactly the sum of the computation and the shuffle from the previous two tables. Intel's x86 CPU has independent execution ports for processing shuffles, multiplies, additions, and moves. Overall, the shuffle-based transpose to enable an SOA implementation results in a respectable speedup.

| **SIMD Benefit** | **Best Serial** | **128 bit** | **256 bit** |
|---|---|---|---|
| **Speedup** | 1 (baseline) | 2.3 | 2.9 |

## Conclusion

From a vanilla C/C++ implementation, there are a number of incremental steps that a developer can take to improve code performance, including setting compiler flags to use the available architecture/instructions, picking the faster instructions within the implementation, and exploiting SIMD. Starting from our humble non-tuned Release mode compilation and progressing all the way to our 256-bit Intel® AVX version, the performance improves by over an order of magnitude.

Working with data structures that are already in SOA layout would certainly be the most efficient, but this is not always possible in many applications. Shuffling 3D data between SOA and AOS on the fly is a worthwhile way to get better utilization of the Intel® CPU processor and potentially make 3D applications run faster. The amount of computation per vector in this example is quite small. When there is more work to be done in the innermost loop, the speedup attained by this approach will likely be even greater.

## About The Author

Stan Melax received his Bachelor’s and Master’s degrees in Computing Science from the University of Alberta in Canada in the early 90s. He has spent much of his career in the game industry including BioWare, EA, and Ageia. Stan is now a Graphics Software Engineer at Intel Corporation where he gets to help developers write faster code and make better applications.

## Appendix: Transpose Source Code

### AOS to SOA 128-bit

C source code with 128-bit Intel® Advanced Vector Extensions (Intel® AVX) intrinsics for converting AOS to SOA:

```
float *p;  // address of first vector
__m128 x0y0z0x1 = _mm_load_ps(p+0);
__m128 y1z1x2y2 = _mm_load_ps(p+4);
__m128 z2x3y3z3 = _mm_load_ps(p+8);
__m128 x2y2x3y3 = _mm_shuffle_ps(y1z1x2y2,z2x3y3z3,_MM_SHUFFLE( 2,1,3,2));
__m128 y0z0y1z1 = _mm_shuffle_ps(x0y0z0x1,y1z1x2y2,_MM_SHUFFLE( 1,0,2,1));
__m128 x        = _mm_shuffle_ps(x0y0z0x1,x2y2x3y3,_MM_SHUFFLE( 2,0,3,0)); // x0x1x2x3
__m128 y        = _mm_shuffle_ps(y0z0y1z1,x2y2x3y3,_MM_SHUFFLE( 3,1,2,0)); // y0y1y2y3
__m128 z        = _mm_shuffle_ps(y0z0y1z1,z2x3y3z3,_MM_SHUFFLE( 3,0,3,1)); // z0z1z2z3
```

The output is found in __m128 registers x,y and z.

**SOA to AOS 128-bit**

C source code with 128-bit Intel® Advanced Vector Extensions (Intel® AVX) intrinsics for converting SOA to AOS:

```
__m128 x,y,z;  // Starting SOA data
__m128 x0x2y0y2 = _mm_shuffle_ps(x,y, _MM_SHUFFLE(2,0,2,0));
__m128 y1y3z1z3 = _mm_shuffle_ps(y,z, _MM_SHUFFLE(3,1,3,1));
__m128 z0z2x1x3 = _mm_shuffle_ps(z,x, _MM_SHUFFLE(3,1,2,0));

__m128 rx0y0z0x1= _mm_shuffle_ps(x0x2y0y2,z0z2x1x3, _MM_SHUFFLE(2,0,2,0));
__m128 ry1z1x2y2= _mm_shuffle_ps(y1y3z1z3,x0x2y0y2, _MM_SHUFFLE(3,1,2,0));
__m128 rz2x3y3z3= _mm_shuffle_ps(z0z2x1x3,y1y3z1z3, _MM_SHUFFLE(3,1,3,1));

_mm_store_ps(p+0, rx0y0z0x1 );
_mm_store_ps(p+4, ry1z1x2y2 );
_mm_store_ps(p+8, rz2x3y3z3 );
```

Registers x,y,z containing the data for 4 vectors is shuffled and stored into packed array starting at pointer p.

**AOS to SOA 256-bit**

C source code with 256-bit Intel® Advanced Vector Extensions (Intel® AVX) intrinsics for converting AOS to SOA:

```
 float *p;  // address of first vector
__m128 *m = (__m128*) p;
__m256 m03;
__m256 m14;
__m256 m25;
m03  = _mm256_castps128_ps256(m[0]); // load lower halves
m14  = _mm256_castps128_ps256(m[1]);
m25  = _mm256_castps128_ps256(m[2]);
m03  = _mm256_insertf128_ps(m03 ,m[3],1);  // load upper halves
m14  = _mm256_insertf128_ps(m14 ,m[4],1);
m25  = _mm256_insertf128_ps(m25 ,m[5],1);

__m256 xy = _mm256_shuffle_ps(m14, m25, _MM_SHUFFLE( 2,1,3,2)); // upper x's and y's
__m256 yz = _mm256_shuffle_ps(m03, m14, _MM_SHUFFLE( 1,0,2,1)); // lower y's and z's
__m256 x  = _mm256_shuffle_ps(m03, xy , _MM_SHUFFLE( 2,0,3,0));
__m256 y  = _mm256_shuffle_ps(yz , xy , _MM_SHUFFLE( 3,1,2,0));
__m256 z  = _mm256_shuffle_ps(yz , m25, _MM_SHUFFLE( 3,0,3,1));
```

Eight 3D vectors are loaded from address p and the output is found in __m256 registers x,y and z. Although this may appear intimidating, it is a natural extension of the 128 bit version.

**SOA to AOS 256-bit**

C source code with 256-bit Intel® Advanced Vector Extensions (Intel® AVX) intrinsics for converting SOA to AOS:

```
__m256 x,y,z;  // Starting SOA data
float *p; // output pointer
__m128 *m = (__m128*) p;

__m256 rxy = _mm256_shuffle_ps(x,y, _MM_SHUFFLE(2,0,2,0));
__m256 ryz = _mm256_shuffle_ps(y,z, _MM_SHUFFLE(3,1,3,1));
__m256 rzx = _mm256_shuffle_ps(z,x, _MM_SHUFFLE(3,1,2,0));

__m256 r03 = _mm256_shuffle_ps(rxy, rzx, _MM_SHUFFLE(2,0,2,0));
__m256 r14 = _mm256_shuffle_ps(ryz, rxy, _MM_SHUFFLE(3,1,2,0));
__m256 r25 = _mm256_shuffle_ps(rzx, ryz, _MM_SHUFFLE(3,1,3,1));

m[0] = _mm256_castps256_ps128( r03 );
m[1] = _mm256_castps256_ps128( r14 );
m[2] = _mm256_castps256_ps128( r25 );
m[3] = _mm256_extractf128_ps( r03 ,1);
m[4] = _mm256_extractf128_ps( r14 ,1);
m[5] = _mm256_extractf128_ps( r25 ,1);
```

Registers x,y,z containing the data for eight 3D vectors is shuffled and stored into packed array starting at pointer p.