Easy SIMD through Wrappers

By Michael Kopietz

1. Introduction

This article aims to change your thinking on how SIMD programming can be applied in your code. By thinking of SIMD lanes as functioning similarly to CPU threads, you will gain new insights and be able to apply SIMD more often in your code.

Intel has been shipping CPUs with SIMD support for about twice as long as they have been shipping multi core CPUs, yet threading is more established in software development. One reason for this increased support is an abundance of tutorials that introduce threading in a simple “run this entry function n-times” manner, skipping all the possible traps. On the other side, SIMD tutorials tend to focus on achieving the final 10% speed up that requires you to double the size of your code. If these tutorials provide code as an example, you may find it hard to focus on all the new information and at the same time come up with your simple and elegant way of using it. Thus showing a simple, useful way of using SIMD is the topic of this paper.

First the basic principle of SIMD code: alignment. Probably all SIMD hardware either demands or at least prefers some natural alignment, and explaining the basics could fill a paper [1]. But in general, if you're not running out of memory, it is important for you to allocate memory in a cache friendly way. For Intel CPUs that means allocating memory on a 64 byte boundary as shown in ***Code Snippet 1***.

```
inline void* operator new(size_t size)
{
	return _mm_malloc(size, 64);
}

inline void* operator new[](size_t size)
{
	return _mm_malloc(size, 64);
}

inline void operator delete(void *mem)
{
	_mm_free(mem);
}

inline void operator delete[](void *mem)
{
	_mm_free(mem);
}
```

Code Snippet 1: *Allocation functions that respect cache-friendly 64 byte boundaries*

2. The basic idea

The way to begin is simple: assume every lane of a SIMD register executes as a thread. In case of Intel® Streaming SIMD Extensions (Intel® SSE), you have 4 threads/lanes, with Intel® Advanced Ventor Extensions (Intel® AVX) 8 threads/lanes and 16 threads/lanes on Intel® Xeon-p Phi coprocessors .

To have a 'drop in' solution, the first step is to implement classes that behave mostly like primitive data types. Wrap 'int', 'float' etc. and use those wrappers as the starting point for every SIMD implementation. For the Intel

SSE version, replace the float member with __m128, int and unsigned int with __m128i and implement operators using Intel SSE intrinsics or Intel AVX intrinsics as in ***Code Snippet 2***.

```
// SEE 128-bit
inline DRealF operator+(DRealF R)const{return DRealF(_mm_add_ps(m_V, R.m_V));}
inline DRealF operator-(DRealF R)const{return DRealF(_mm_sub_ps(m_V, R.m_V));}
inline DRealF operator*(DRealF R)const{return DRealF(_mm_mul_ps(m_V, R.m_V));}
inline DRealF operator/(DRealF R)const{return DRealF(_mm_div_ps(m_V, R.m_V));}

// AVX 256-bit
inline DRealF operator+(const DRealF& R)const{return DRealF(_mm256_add_ps(m_V, R.m_V));}
inline DRealF operator-(const DRealF& R)const{return DRealF(_mm256_sub_ps(m_V, R.m_V));}
inline DRealF operator*(const DRealF& R)const{return DRealF(_mm256_mul_ps(m_V, R.m_V));}
inline DRealF operator/(const DRealF& R)const{return DRealF(_mm256_div_ps(m_V, R.m_V));}
```

Code Snippet 2: *Overloaded arithmetic operators for SIMD wrappers*

3. Usage Example

Now let's assume you're working on two HDR images, where every pixel is a float and you blend between both images.

```
void CrossFade(float* pOut,const float* pInA,const float* pInB,size_t PixelCount,float
Factor)
{
	const DRealF BlendA(1.f - Factor);
	const DRealF BlendB(Factor);
	for(size_t i = 0; i < PixelCount; i += THREAD_COUNT)
		*(DRealF*)(pOut + i) = *(DRealF*)(pInA + i) * BlendA + *(DRealF*)(pInB + i) +
BlendB;
}
```

Code Snippet 3: *Blending function that works with both primitive data types and SIMD data*

The executable generated from ***Code Snippet 3*** runs natively on normal registers as well as on Intel SSE and Intel AVX. It's not really the vanilla way you'd write it usually, but every C++ programmer should still be able to read and understand it. Let's see whether it's the way you expect. The first and second line of the implementation initialize the blend factors of our linear interpolation by replicating the parameter to whatever width your SIMD register has.

The third line is nearly a normal loop. The only special part is "THREAD_COUNT". It's 1 for normal registers, 4 for Intel SSE and 8 for Intel AVX, representing the count of lanes of the register, which in our case resembles threads.

The fourth line indexes into the arrays and both input pixel are scaled by the blend factors and summed. Depending on your preference of writing it, you might want to use some temporaries, but there is no intrinsic you need to look up, no implementation per platform.

4. Drop in

Now it's time to prove that it actually works. Let's take a vanilla MD5 hash implementation and use all of your available CPU power to find the pre-image. To achieve that, we'll replace the primitive types with our SIMD types. MD5 is running several "rounds" that apply various simple bit operations on unsigned integers as demonstrated in ***Code Snippet 4***.

```
#define LEFTROTATE(x, c) (((x) << (c)) | ((x) >> (32 - (c))))
#define BLEND(a, b, x) SelectBit(a, b, x)

template<int r>
inline DRealU Step1(DRealU a,DRealU b,DRealU c,DRealU d,DRealU k,DRealU w)
{
	const DRealU f = BLEND(d, c, b);
	return b + LEFTROTATE((a + f + k + w), r);
}

template<int r>
inline DRealU Step2(DRealU a,DRealU b,DRealU c,DRealU d,DRealU k,DRealU w)
{
	const DRealU f = BLEND(c, b, d);
	return b + LEFTROTATE((a + f + k + w),r);
}

template<int r>
inline DRealU Step3(DRealU a,DRealU b,DRealU c,DRealU d,DRealU k,DRealU w)
{
	DRealU f = b ^ c ^ d;
	return b + LEFTROTATE((a + f + k + w), r);
}

template<int r>
inline DRealU Step4(DRealU a,DRealU b,DRealU c,DRealU d,DRealU k,DRealU w)
{
	DRealU f = c ^ (b | (~d));
	return b + LEFTROTATE((a + f + k + w), r);
}
```

Code Snippet 4: *MD5 step functions for SIMD wrappers*

Besides the type naming, there is really just one change that could look a little bit like magic — the “SelectBit”. If a bit of x is set, the respective bit of b is returned; otherwise, the respective bit of a; in other words, a blend. The main MD5 hash function is shown in ***Code Snippet 5***.

```
inline void MD5(const uint8_t* pMSG,DRealU& h0,DRealU& h1,DRealU& h2,DRealU& h3,uint32_t
Offset)
{
	const DRealU w0  = Offset(DRealU(*reinterpret_cast<const uint32_t*>(pMSG + 0 * 4) +
Offset));
	const DRealU w1  = *reinterpret_cast<const uint32_t*>(pMSG + 1 * 4);
	const DRealU w2  = *reinterpret_cast<const uint32_t*>(pMSG + 2 * 4);
	const DRealU w3  = *reinterpret_cast<const uint32_t*>(pMSG + 3 * 4);
	const DRealU w4  = *reinterpret_cast<const uint32_t*>(pMSG + 4 * 4);
	const DRealU w5  = *reinterpret_cast<const uint32_t*>(pMSG + 5 * 4);
	const DRealU w6  = *reinterpret_cast<const uint32_t*>(pMSG + 6 * 4);
	const DRealU w7  = *reinterpret_cast<const uint32_t*>(pMSG + 7 * 4);
	const DRealU w8  = *reinterpret_cast<const uint32_t*>(pMSG + 8 * 4);
	const DRealU w9  = *reinterpret_cast<const uint32_t*>(pMSG + 9 * 4);
	const DRealU w10 = *reinterpret_cast<const uint32_t*>(pMSG + 10 * 4);
	const DRealU w11 = *reinterpret_cast<const uint32_t*>(pMSG + 11 * 4);
	const DRealU w12 = *reinterpret_cast<const uint32_t*>(pMSG + 12 * 4);
	const DRealU w13 = *reinterpret_cast<const uint32_t*>(pMSG + 13 * 4);
	const DRealU w14 = *reinterpret_cast<const uint32_t*>(pMSG + 14 * 4);
	const DRealU w15 = *reinterpret_cast<const uint32_t*>(pMSG + 15 * 4);
```

```
        DRealU a = h0;
        DRealU b = h1;
        DRealU c = h2;
        DRealU d = h3;

        a = Step1< 7>(a, b, c, d, k0, w0);
        d = Step1<12>(d, a, b, c, k1, w1);
        .
        .
        .
        d = Step4<10>(d, a, b, c, k61, w11);
        c = Step4<15>(c, d, a, b, k62, w2);
        b = Step4<21>(b, c, d, a, k63, w9);

        h0 += a;
        h1 += b;
        h2 += c;
        h3 += d;
}
```

Code Snippet 5: *The main MD5 function*

The majority of the code is again like a normal C function, except that the first lines prepare the data by replicating our SIMD registers with the parameter passed. In this case we load the SIMD registers with the data we want to hash. One specialty is the “Offset” call, since we don't want every SIMD lane to do exactly the same work, this call offsets the register by the lane index. It's like a thread-id you would add. See ***Code Snippet 6*** for reference.

```
Offset(Register)
{
        for(i = 0; i < THREAD_COUNT; i++)
                Register[i] += i;
}
```

Code Snippet 6: *Offset is a utility function for dealing with different register widths*

That means, our first element that we want to hash is not [0, 0, 0, 0] for Intel SSE or [0, 0, 0, 0, 0, 0, 0, 0] for Intel AVX. Instead the first element is [0, 1, 2, 3] and [0, 1, 2, 3, 4, 5, 6, 7], respectively. This replicates the effect of running the function in parallel by 4 or 8 threads/cores, but in case of SIMD, instruction parallel.

We can see the results for our 10 minutes of hard work to get this function SIMD-ified in *Table 1*.

Table 1*: MD5 performance with primitive and SIMD types*

Type	Time	Speedup
x86 integer	379.389s	1.0x
SSE4	108.108s	3.5x
AVX2	51.490s	7.4x

5. Beyond Simple SIMD-threads

The results are satisfying, not linearly scaling, as there is always some non-threaded part (you can easily identify it in the provided source code). But we're not aiming for the last 10% with twice the work. As a programmer, you'd probably prefer to go for other quick solutions that maximize the gain. Some considerations always arise, like: Would it be worthwhile to unroll the loop?

MD5 hashing seems to be frequently dependent on the result of previous operations, which is not really friendly for CPU pipelines, but you could become register bound if you unroll. Our wrappers can help us to evaluate that easily. Unrolling is the software version of hyper threading, we emulate twice the threads running by repeating the execution of operations on twice the data than SIMD lanes available. Therefore create a duplicate type alike and implement unrolling inside by duplicating every operation for our basic operators as in ***Code Snippet 7***.

```
struct __m1282
{
	__m128		m_V0;
	__m128		m_V1;
	inline		__m1282(){}
	inline		__m1282(__m128 C0, __m128 C1):m_V0(C0), m_V1(C1){}
};

inline DRealF operator+(DRealF R)const
	{return __m1282(_mm_add_ps(m_V.m_V0, R.m_V.m_V0),_mm_add_ps(m_V.m_V1, R.m_V.m_V1));}
inline DRealF operator-(DRealF R)const
	{return __m1282(_mm_sub_ps(m_V.m_V0, R.m_V.m_V0),_mm_sub_ps(m_V.m_V1, R.m_V.m_V1));}
inline DRealF operator*(DRealF R)const
	{return __m1282(_mm_mul_ps(m_V.m_V0, R.m_V.m_V0),_mm_mul_ps(m_V.m_V1, R.m_V.m_V1));}
inline DRealF operator/(DRealF R)const
	{return __m1282(_mm_div_ps(m_V.m_V0, R.m_V.m_V0),_mm_div_ps(m_V.m_V1, R.m_V.m_V1));}
```

Code Snippet 7: *These operators are re-implemented to work with two SSE registers at the same time*

That's it, really, now we can again run the timings of the MD5 hash function.

Table 2: *MD5 performance with loop unrolling SIMD types*

Type	Time	Speedup
x86 integer	379.389s	1.0x
SSE4	108.108s	3.5x
SSE4 x2	75.659s	4.8x
AVX2	51.490s	7.4x
AVX2 x2	36.014s	10.5x

The data in *Table 2* shows that it's clearly worth unrolling. We achieve speed beyond the SIMD lane count scaling, probably because the x86 integer version was already stalling the pipeline with operation dependencies.

6. More complex SIMD-threads

So far our examples were simple in the sense that the code was the usual candidate to be vectorized by hand. There is nothing complex beside a lot of compute demanding operations. But how would we deal with more complex scenarios like branching?

The solution is again quite simple and widely used: speculative calculation and masking. Especially if you've worked with shader or compute languages, you'll likely have encountered this before. Let's take a look at a basic branch of **Code Snippet *8*** and rewrite it to a ?: operator as in **Code Snippet *9***.

```
int a = 0;
if(i % 2 == 1)
	a = 1;
else
```

```
        a = 3;
```

Code Snippet 8: *Calculates the mask using if-else*

```
int a = (i % 2) ? 1 : 3;
```

Code Snippet 9: *Calculates the mask with ternary operator ?:*

If you recall our bit-select operator of ***Code Snippet 4***, we can also use it to achieve the same with only bit operations in **Code Snippet *10***.

```
int Mask = (i % 2) ? ~0 : 0;
int a = SelectBit(3, 1, Mask);
```

Code Snippet 10: *Use of SelectBit prepares for SIMD registers as data*

Now, that might seem pointless, if we still have an ?: operator to create the mask, and the compare does not result in true or false, but in all bits set or cleared. Yet this is not a problem, because all bits set or cleared are actually what the comparison instruction of Intel SSE and Intel AVX return.

Of course, instead of assigning just 3 or 1, you can call functions and select the returned result you want. That might lead to performance improvement even in non-vectorized code, as you avoid branching and the CPU never suffers of branch misprediction, but the more complex the functions you call, the more a misprediction is possible. Even in vectorized code, we'll avoid executing unneeded long branches, by checking for special cases where all elements of our SIMD register have the same comparison result as demonstrated in **Code Snippet *11***.

```
int Mask = (i % 2) ? ~0 : 0;
int a = 0;
if(All(Mask))
        a = Function1();
else
if(None(Mask))
        a = Function3();
else
        a = BitSelect(Function3(), Function1(), Mask);
```

Code Snippet 11: *Shows an optimized branchless selection between two functions*

This detects the special cases where all of the elements are 'true' or where all are 'false'. Those cases run on SIMD the same way as on x86, just the last 'else' case is where the execution flow would diverge, hence we need to use a bit-select.

If Function1 or Function3 modify any data, you'd need to pass the mask down the call and explicitly bit select the modifications just like we've done here. For a drop-in solution, that's a bit of work, but it still results in code that’s readable by most programmers.

7. Complex example

Let's again take some source code and drop in our SIMD types. A particularly interesting case is raytracing of distance fields. For this, we'll use the scene from Iñigo Quilez's demo [2] with his friendly permission, as shown in **Figure *1***.

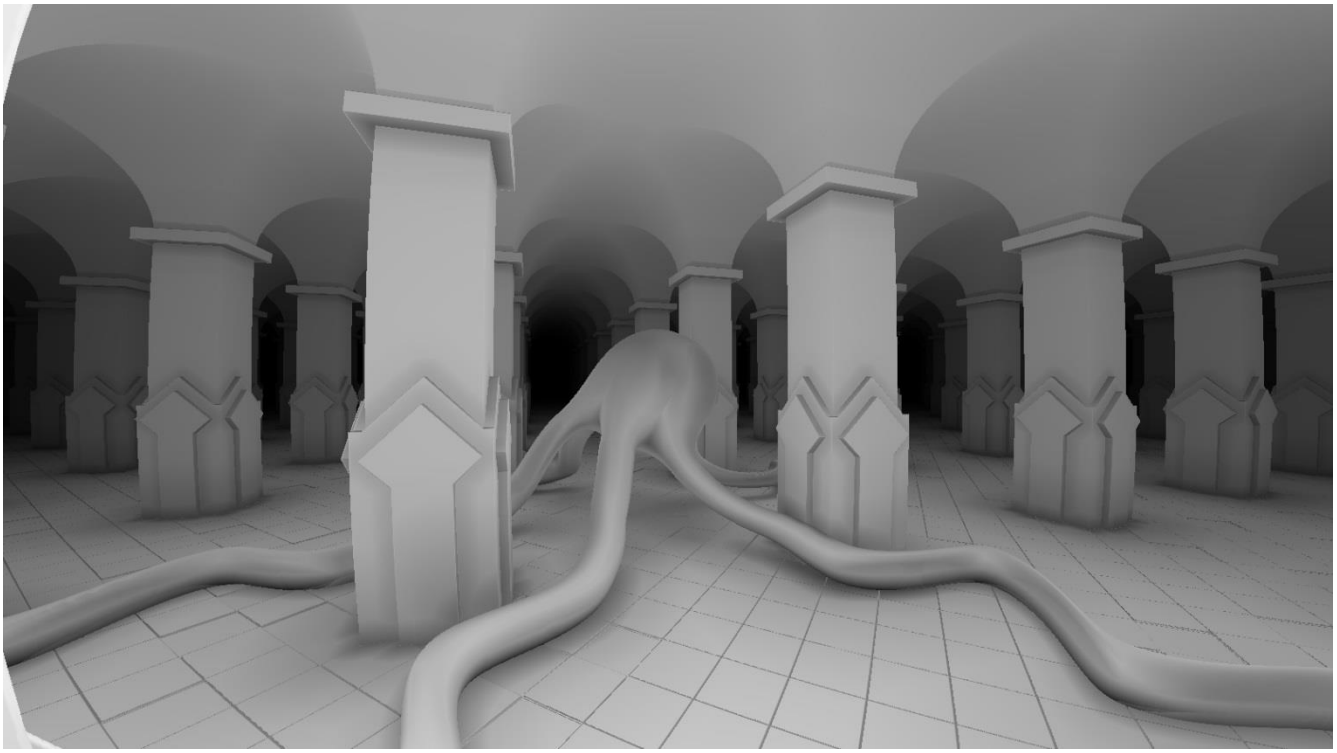

Figure 1: *Test scene from Iñigo Quilez's raycasting demo*

The “SIMD threading” is placed at a spot where you'd add threading usually. Every thread handles a pixel, traversing the world until it hits something, subsequently a little bit of shading is applied and the pixel is converted to RGBA and written to the frame buffer.

The scene traversing is done in an iterative way. Every ray has an unpredictable amount of steps until a hit is recognized. For example, a close up wall is reached after a few steps while some rays might reach the maximum trace distance not hit anything at all. Our main loop in ***Code Snippet 12*** handles both cases using the bit select method we've discussed in the previous section.

```
DRealU LoopMask(RTrue);
for(; a < 128; a++)
{
	DRealF Dist		=	SceneDist(O.x, O.y, O.z, C);
	DRealU DistU		=	*reinterpret_cast<DRealU*>(&Dist) & DMask(LoopMask);
	Dist			=	*reinterpret_cast<DRealF*>(&DistU);
	TotalDist		=	TotalDist + Dist;
	O			+=	D * Dist;
	LoopMask		=	LoopMask && Dist > MinDist && TotalDist < MaxDist;
	if(DNone(LoopMask))
		break;
}
```

Code Snippet 12: *Raycasting with SIMD types*

The LoopMask variable identifies that a ray is active by ~0 or 0 in which case we are done with that ray. At the end of the loop, we test whether no ray is active anymore and in this case we break out of the loop.

In the line above we evaluate our conditions for the rays, whether we're close enough to an object to call it a hit or whether the ray is already beyond the maximum distance we want to trace. We logically AND it with the

previous result, as the ray might be already terminated in one of the previous iterations.

“SceneDist” is the evaluation function for our tracing - It's run for all SIMD-lanes and is the heavy weighted function that returns the current distance to the closest object. The next line sets the distance elements to 0 for rays that are not active anymore and steps this amount further for the next iteration.

The original “SceneDist” had some assembler optimizations and material handling that we don't need for our test, and this function is reduced to the minimum we need to have a complex example. Inside are still some if-cases that are handled exactly the same as before. Overall, the “SceneDist” is quite large and rather complex and would take a while to rewrite it by hand for every SIMD-platform again and again. You might need to convert it all in one go, while typos might generate completely wrong results. Even if it works, you'll have only a few functions that you really understand, and maintenance is much higher. Doing it by hand should be the last resort. Compared to that, our changes are relatively minor. It's easy to modify and you are able to extend the visual appearance, without the need to worry about optimizing it again and being the only maintainer that understands the code, just like it would be by adding real threads.

But we've done that work to see results, so let’s check the timings in *Table 3*.

Table 3: *Raytracing performance with primitive and SIMD types, including loop unrolling types*

Type	FPS	Speedup
x86	0.992FPS	1.0x
SSE4	3.744FPS	3.8x
SSE4 x2	3.282FPS	3.3x
AVX2	6.960FPS	7.0x
AVX2 x2	5.947FPS	6.0x

You can clearly see the speed up is not scaling linearly with the element count, which is mainly because of the divergence. Some rays might need 10 times more iterations than others.

8. Why not let the compiler do it?

Compilers nowadays can vectorize to some degree, but the highest priority for the generated code is to deliver correct results, as you would not use 100 time faster binaries that deliver wrong results even 1% of the time. Some assumptions we make, like the data will be aligned for SIMD, and we allocate enough padding to not overwrite consecutive allocations, are out of scope for the compiler. You can get annotations from the Intel compiler about all opportunities it had to skip because of assumptions it could not guarantee, and you can try to rearrange code and make promises to the compiler so it'll generate the vectorized version. But that's work you have to do every time you modify your code and in more complex cases like branching, you can just guess whether it will result in branchless bit selection or serialized code.

The compiler has also no inside knowledge of what you intend to create. You know whether threads will be diverging or coherent and implement a branched or bit selecting solution. You see the point of attack, the loop that would make most sense to change to SIMD, whereas the compiler can just guess whether it will run 10times or 1 million times.

Relying on the compiler might be a win in one place and pain in another. It's good to have this alternative solution you can rely on, just like your hand placed thread entries.

9. Real threading?

Yes, real threading is useful and SIMD-threads are not a replacement — both are orthogonal. SIMD-threads are

still not as simple to get running as real threading is, but you'll also run into less trouble about synchronization and seldom bugs. The really nice advantage is that every core Intel sells can run your SIMD-thread version with all the 'threads'. A dual core CPU will run 4 or 8 times faster just like your quad socket 15-core Haswell-EP. Some results for our benchmarks in combination with threading are summarized in *Table 4* through *Table 7*.[1]

Table 4: *MD5 Performance on Intel® Core™ i7 4770K with both SIMD and threading*

Threads	Type	Time	Speedup
1T	x86 integer	311.704s	1.00x
8T	x86 integer	47.032s	6.63x
1T	SSE4	90.601s	3.44x
8T	SSE4	14.965s	20.83x
1T	SSE4 x2	62.225s	5.01x
8T	SSE4 x2	12.203s	25.54x
1T	AVX2	42.071s	7.41x
8T	AVX2	6.474s	48.15x
1T	AVX2 x2	29.612s	10.53x
8T	AVX2 x2	5.616s	55.50x

Table 5: *Raytracing Performance on Intel® Core™ i7 4770K with both SIMD and threading*

Threads	Type	FPS	Speedup
1T	x86 integer	1.202FPS	1.00x
8T	x86 integer	6.019FPS	5.01x
1T	SSE4	4.674FPS	3.89x
8T	SSE4	23.298FPS	19.38x
1T	SSE4 x2	4.053FPS	3.37x
8T	SSE4 x2	20.537FPS	17.09x
1T	AVX2	8.646FPS	4.70x
8T	AVX2	42.444FPS	35.31x
1T	AVX2 x2	7.291FPS	6.07x
8T	AVX2 x2	36.776FPS	30.60x

Table 6: *MD5 Performance on Intel® Core™ i7 5960X with both SIMD and threading*

Threads	Type	Time	Speedup
1T	x86 integer	379.389s	1.00x
16T	x86 integer	28.499s	13.34x
1T	SSE4	108.108s	3.51x
16T	SSE4	9.194s	41.26x
1T	SSE4 x2	75.694s	5.01x
16T	SSE4 x2	7.381s	51.40x
1T	AVX2	51.490s	3.37x
16T	AVX2	3.965s	95.68x
1T	AVX2 x2	36.015s	10.53x
16T	AVX2 x2	3.387s	112.01x

Table 7: *Raytracing Performance on Intel® Core™ i7 5960X with both SIMD and threading*

Threads	Type	FPS	Speedup
1T	x86 integer	0.992FPS	1.00x
16T	x86 integer	6.813FPS	6.87x
1T	SSE4	3.744FPS	3.774x
16T	SSE4	37.927FPS	38.23x
1T	SSE4 x2	3.282FPS	3.31x
16T	SSE4 x2	33.770FPS	34.04x

1T	AVX2	6.960FPS	7.02x
16T	AVX2	70.545FPS	71.11x
1T	AVX2 x2	5.947FPS	6.00x
16T	AVX2 x2	59.252FPS	59.76x

[1]Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. **For more information go to http://www.intel.com/performance**.

As you can see, the threading results vary depending on the CPU, the SIMD-thread results scale similar. But it's striking that you can reach speed up factors in the higher two digits if you combine both ideas. It makes sense to go for the 8x speed up on a dual core, but so does it make sense to go for an additional 8x speed up on highly expensive server hardware.

Join me, SIMD-ify your code!

About the Author

Michael Kopietz is Render Architect at Crytek's R&D and leads a team of engineers developing the rendering of CryEngine(R) and also guides students during their thesis. He worked among other things on the cross platform rendering architecture, software rendering and on highly responsive server, always high-performance and reusable code in mind. Prior, he was working on ship-battle and soccer simulation game rendering. Having his root in assembler programming on the earliest home consoles, he still wants to make every cycle count.

Code License

References

[1] Memory Management for Optimal Performance on Intel® Xeon Phi™ Coprocessor: Alignment and Prefetching https://software.intel.com/en-us/articles/memory-management-for-optimal-performance-on-intel-xeon-phi-coprocessor-alignment-and

[2] Rendering Worlds with Two Triangles by Iñigo Quilez
http://www.iquilezles.org/www/material/nvscene2008/nvscene2008.htm

Notices

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Any software source code reprinted in this document is furnished under a software license and may only be used or copied in accordance with the terms of that license.

Intel, the Intel logo, and Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.