"Box pruning revisited" – an optimization project by Pierre Terdiman – 2017

Part 9a - SIMD

For this post, we start again from version 6b, ignoring our experiments with integer comparisons and branches in versions 7 and 8. That is, we go back to this version:

```
static __forceinline int intersects2D(const AABB& a, const AABB& b)
{
	if(	b.mMax.y < a.mMin.y || a.mMax.y < b.mMin.y
	||	b.mMax.z < a.mMin.z || a.mMax.z < b.mMin.z)
		return 0;
	return 1;
}
```

The reason for doing so is simply that the above function is easier to work with: these 4 comparisons are a perfect fit for SIMD. So we will try to replace them with a single SIMD comparison, and thus, a single branch. We can skip the work we did in version 8 since SIMD will replicate this naturally.

After the modification we did in part 5, we now read y's and z's directly there, and the code is ripe for SIMDifying. It would have been much more difficult to see or consider doing with the initial implementation, where the axes came from a user-defined parameter. So this is the moment where we harvest the "great things coming from small beginnings".

Doing so is not entirely trivial though. There is SIMD and SIMD. Converting something to SIMD does not guarantee performance gains. A naive conversion can in fact easily be slower than the scalar code you started from.

But let's give it a try. We want to compute:

```
if(	b.mMax.y < a.mMin.y
	||	a.mMax.y < b.mMin.y
	||	b.mMax.z < a.mMin.z
	||	a.mMax.z < b.mMin.z)
```

So we could just load these elements in two SSE registers, do a single SSE comparison, use *movemask* to get the results, and… done? The modifications are very simple, just replace this:

```
if(intersects2D(BoxList[Index0], BoxList[Index1]))
```

With this:

```
const AABB& a = BoxList[Index0];
const AABB& b = BoxList[Index1];
const __m128 b0 = _mm_set_ps(b.mMax.y, a.mMax.y, b.mMax.z, a.mMax.z);
const __m128 b1 = _mm_set_ps(a.mMin.y, b.mMin.y, a.mMin.z, b.mMin.z);
const __m128 d = _mm_cmplt_ps(b0, b1);
if(!_mm_movemask_ps(d))
```

It is straightforward and it works:

Home PC:

Complete test (brute force): found 11811 intersections in 781594 K-cycles.
38231 K-cycles.
34532 K-cycles.
34598 K-cycles.
34996 K-cycles.
34546 K-cycles.
34486 K-cycles.
34649 K-cycles.
34510 K-cycles.
34709 K-cycles.
34533 K-cycles.
34496 K-cycles.
34621 K-cycles.
34556 K-cycles.
34867 K-cycles.
35392 K-cycles.
34666 K-cycles.
Complete test (box pruning): found 11811 intersections in 34486 K-cycles.

Office PC:

Complete test (brute force): found 11811 intersections in 810655 K-cycles.
32847 K-cycles.
32021 K-cycles.
32046 K-cycles.
31932 K-cycles.
32283 K-cycles.
32139 K-cycles.
31864 K-cycles.
32325 K-cycles.
33420 K-cycles.
32087 K-cycles.
31969 K-cycles.
32018 K-cycles.
32879 K-cycles.
32588 K-cycles.
32214 K-cycles.
31875 K-cycles.
Complete test (box pruning): found 11811 intersections in 31864 K-cycles.

The gains are summarized here:

Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(101662)			
Version2 - base	98822	0	0%	1.0
Version3	93138	~5600	~5%	~1.06
Version4	81834	~11000	~12%	~1.20
Version5	78140	~3600	~4%	~1.26
Version6a	60579	~17000	~22%	~1.63
Version6b	41605	~18000	~31%	~2.37
(Version7)	(40906)	-	-	-
(Version8)	(31383)	(~10000)	(~24%)	(~3.14)
Version9a	34486	~7100	~17%	~2.86

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
(Version1)	(96203)			
Version2 - base	92885	0	0%	1.0
Version3	88352	~4500	~5%	~1.05
Version4	77156	~11000	~12%	~1.20
Version5	73778	~3300	~4%	~1.25
Version6a	58451	~15000	~20%	~1.58
Version6b	45634	~12000	~21%	~2.03
(Version7)	(43987)	-	-	-
(Version8)	(29083)	(~16000)	(~36%)	(~3.19)
Version9a	31864	~13000	~30%	~2.91

The gains for version 9a are relative to version 6b (we ignored versions 7 and 8). And we see something interesting here: our SIMD version “worked”, it did provide clear gains compared to version 6b, but… it remains slower than our branchless scalar version, version8.

Oops.

This is the source of never-ending myths and confusion about SIMD on PC: if you started from version 6b, you are likely to conclude that adding SIMD is easy and provides easy gains. If you started from version 8, you are likely to conclude that SIMD is slower than “optimized” scalar code, and not worth the trouble. And if you started from version 8, “optimized” the code with SIMD, and concluded without benchmarking that Version9a was faster, then you are a bloody fool.

In any case, you’d all be wrong. As previously mentioned, there is SIMD and SIMD. Our initial attempt here is just a little bit rough and, indeed, suboptimal.

How do we see it?

As usual: always check the disassembly!

If you do, you will discover that the above SIMD code, which is what I would call the naive version, gave birth to this monstrosity:

```
                const AABB& a = BoxList[Index0];
                const AABB& b = BoxList[Index1];
                const __m128 b0 = _mm_set_ps(b.mMax.y, a.mMax.y, b.mMax.z, a.mMax.z);
                const __m128 b1 = _mm_set_ps(a.mMin.y, b.mMin.y, a.mMin.z, b.mMin.z);
011D2F08  movss       xmm0,dword ptr [eax+4]
011D2F0D  movss       xmm2,dword ptr [edx+4]
011D2F12  movss       xmm4,dword ptr [eax+8]
011D2F17  movss       xmm1,dword ptr [edx+8]
011D2F1C  movss       xmm3,dword ptr [edx+14h]
011D2F21  unpcklps    xmm1,xmm2
011D2F24  movss       xmm2,dword ptr [eax+10h]
011D2F29  unpcklps    xmm4,xmm0
011D2F2C  movss       xmm0,dword ptr [edx+10h]
011D2F31  unpcklps    xmm4,xmm1
011D2F34  movss       xmm1,dword ptr [eax+14h]
011D2F39  unpcklps    xmm3,xmm0
011D2F3C  unpcklps    xmm1,xmm2
011D2F3F  unpcklps    xmm3,xmm1
                const __m128 d = _mm_cmplt_ps(b0, b1);
011D2F42  cmpltps     xmm3,xmm4
                if(!_mm_movemask_ps(d))
011D2F46  movmskps    eax,xmm3
011D2F49  test        eax,eax
011D2F4B  jne         CompleteBoxPruning+2B6h (011D2FA6h)
```

Soooo... errrr... fair enough: that's only one comparison (*cmpltps*) and one branch (*test/jne*) indeed. But that's still quite a bit more than what we asked for.

In particular, one may wonder where the unpack instructions (*unpcklps*) are coming from - we certainly didn't use *_mm_unpacklo_ps* in our code!

So what's going on?

Well, what's going on is that we fell into the intrinsics trap. There is usually a one-to-one mapping between intrinsics and assembly instructions, but not always. Some of them are "composite" intrinsics that can generate an arbitrary number of assembly instructions - whatever it takes to implement the desired action. This is the case for *_mm_set_ps*, and this is where all the *movss* and *unpcklps* are coming from. The two *_mm_set_ps* intrinsics generated ***14*** assembly instructions here. A lot of people assume that using intrinsics is "the same" as writing in assembly directly: it is, until it isn't. Sometimes it hides a lot of important details behind syntactic sugar, and that can bite you hard if you are not aware of it.

So here's our lesson: if your SIMD code uses *_mm_set_ps*, it is probably not optimal.

While we're at it, there is another lesson to learn here: the compiler won't do the SIMD job for you. We did not have to reorder the data here, we did not have to align structures on 16-byte boundaries, we did not have to do complicated things, the change was as straightforward as it gets.

And yet, even in this simple case, with the /SSE2 flag enabled, the compiler still didn't generate the naive SIMD version on its own. So once again: people claiming that using the /SSE2 flag will automatically make the code "4x faster" (as I read online a few times) are tragically, comically wrong. For that kind of speedup, you need to work hard.

And work hard, we will.

Back to the drawing board.

What we learnt:

Writing something in SIMD does not guarantee better performance than a scalar version.

There's no 1:1 mapping between intrinsics and assembly instructions when you use "composite" intrinsics.

Don't use *_mm_set_ps*.

The compiler will not even write naive SIMD for you.