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

Part 9c – data alignment

Last time we were left with a bit of a mystery. Our optimizations produced consistent results so far, on both machines the code was benchmarked on.

However things changed when switching to SIMD.

We have a naive SIMD implementation (9a) producing these results:

Home PC:	34486 K-cycles.
Office PC:	31864 K-cycles.

And an optimized SIMD implementation (9b) producing these results (for the "safe" version, but timings are similar for the unsafe one):

Home PC:	32565 K-cycles.
Office PC:	15116 K-cycles.

In other words, the change from naive to optimized SIMD had almost no effect on one machine, but it made things run twice faster on the other machine.

By now you should know what to do next: check the disassembly. Recall that the naive version looked like 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);
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)
```

The optimized version (9b) however looks much better:

```
                SIMD_OVERLAP_TEST(BoxList[Index1])
01382FD8  movups      xmm0,xmmword ptr [eax+8]
01382FDC  cmpleps     xmm1,xmm0
01382FE0  movmskps    eax,xmm1
01382FE3  cmp         eax,0Fh
01382FE6  jne         CompleteBoxPruning+2C1h (01383041h)
```

As expected the scalar loads (*movss*) and unpacking instructions (*unpcklps*) all vanished, replaced with a single "proper" SIMD load (*movups*). In both cases the last 4 instructions are similar, and implement the comparison / test / branch that didn't change between the naive & optimized functions.

In other words, since the timings are similar, it looks like on the home PC the single *movups* load is as costly as the large block of 14 *movss/unpcklps* from the naive version.

I suppose that's a lesson in itself: all instructions don't have the same cost. The number of instructions is not a reliable way to evaluate performance.

At this point you could fire up a profiler like V-Tune which would probably reveal what's going on (I didn't try), or you could just observe, formulate a theory, and design an experiment to test the theory. You know, the scientific method.

Observe:

```
movups      xmm0,xmmword ptr [eax+8]
```

There's only one line to observe really so it doesn't take a rocket scientist to spot the only potential issue here: that ***u*** is for unaligned, and it should ideally be an ***a***, for aligned.

That is, we should have a *movaps* instead of a *movups* here. In intrinsics-speak, we've been using *_mm_loadu_ps* while we should ideally have used *_mm_load_ps*.

Could unaligned loads have such an impact? Well that's our theory now. Let's test it.

We used unaligned loads because we had no choice: the size of our AABB classes was 24 bytes (6 floats). To be able to use aligned loads, we'd need a multiple of 16 bytes.

So we could add some padding bytes to reach sizeof(AABB)==32, but that would consume more memory, which means also more cache misses.

Or we could just split our AABB in two separate classes, like this:

```
struct SIMD_AABB_X
{
    float mMinX;
    float mMaxX;
};

struct SIMD_AABB_YZ
{
    float mMinY;
    float mMinZ;
    float mMaxY;
    float mMaxZ;
};
```

We would then have to allocate two internal arrays instead of one to store the bounds:

```
SIMD_AABB_X* BoxListX = new SIMD_AABB_X[nb+1];  // MODIFIED
SIMD_AABB_YZ* BoxListYZ = (SIMD_AABB_YZ*)_aligned_malloc(sizeof(SIMD_AABB_YZ)*(nb+1), 16);  // NEW
```

On one hand we would then have to compute two addresses within our inner loop instead of one, to load data from two different memory addresses instead of one before:

```
const SIMD_AABB_X& Box0X = BoxListX[Index0];    // MODIFIED

const SIMD_AABB_YZ& Box0YZ = BoxListYZ[Index0];     // MODIFIED
```

That's likely to be slower. On the other hand we would then be able to use aligned loads for the YZ part, and that should be a win.

In some cases the quickest way to know is simply to try. The disassembly becomes:

```
                SIMD_OVERLAP_TEST(BoxListYZ[Index1])    // MODIFIED
00B730D9  cmpleps     xmm1,xmmword ptr [ecx]
            {
                SIMD_OVERLAP_TEST(BoxListYZ[Index1])    // MODIFIED
00B730DD  movmskps    eax,xmm1
00B730E0  cmp         eax,0Fh
00B730E3  jne         CompleteBoxPruning+2FEh (0B7313Eh)
```

No *movaps*!

But the compiler was able to merge the move directly in the cmp instruction, which is similar - it's an aligned load.

And the results apparently confirm our theory:

Home PC:

Unsafe version:

Complete test (brute force): found 11811 intersections in 781500 K-cycles.
17426 K-cycles.
16272 K-cycles.
16223 K-cycles.
16239 K-cycles.
16224 K-cycles.
16226 K-cycles.
16289 K-cycles.
16248 K-cycles.
16448 K-cycles.
16240 K-cycles.
16224 K-cycles.
16398 K-cycles.
16251 K-cycles.
16485 K-cycles.
16239 K-cycles.
16225 K-cycles.
Complete test (box pruning): found 11715 intersections in 16223 K-cycles.

Safe version:

Complete test (brute force): found 11811 intersections in 781569 K-cycles.
15629 K-cycles.
14813 K-cycles.
14841 K-cycles.
14827 K-cycles.
14803 K-cycles.
14933 K-cycles.
14821 K-cycles.
14803 K-cycles.
15048 K-cycles.
14833 K-cycles.
14803 K-cycles.
14817 K-cycles.
14802 K-cycles.
14802 K-cycles.
14802 K-cycles.
14998 K-cycles.
Complete test (box pruning): found 11811 intersections in 14802 K-cycles.

Office PC:

Unsafe version:

Complete test (brute force): found 11811 intersections in 826531 K-cycles.
13634 K-cycles.
12709 K-cycles.
12707 K-cycles.
13063 K-cycles.
12858 K-cycles.
13824 K-cycles.
13357 K-cycles.
13502 K-cycles.
12863 K-cycles.
13041 K-cycles.
12711 K-cycles.
12932 K-cycles.
12711 K-cycles.
12709 K-cycles.
12928 K-cycles.
12851 K-cycles.
Complete test (box pruning): found 11715 intersections in 12707 K-cycles.

Safe version:

Complete test (brute force): found 11811 intersections in 813914 K-cycles.
13582 K-cycles.
12562 K-cycles.
12825 K-cycles.
13048 K-cycles.
12806 K-cycles.
12562 K-cycles.
12915 K-cycles.
12627 K-cycles.
13216 K-cycles.
12899 K-cycles.
13546 K-cycles.
13163 K-cycles.
12572 K-cycles.
12769 K-cycles.
12662 K-cycles.
12938 K-cycles.
Complete test (box pruning): found 11811 intersections in 12562 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
Version9b - unsafe	32477	~2000	~5%	~3.04
Version9b - safe	32565	~1900	~5%	~3.03
Version9c - unsafe	16223	~16000	~50%	~6.09
Version9c - safe	14802	~17000	~54%	~6.67

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
Version9b - unsafe	15097	~16000	~52%	~6.15
Version9b - safe	15116	~16000	~52%	~6.14
Version9c - unsafe	12707	~2300	~15%	~7.30
Version9c - safe	12562	~2500	~16%	~7.39

So the code on the home PC became about twice as fast as before, which makes it roughly as fast as what we got on the office PC for the previous version.

On the office PC, we only saved about 2500 K-Cycles, which means the aligned loads are indeed faster but not that much faster.

In any case, the important point is that our SIMD version is *finally* always faster than our best scalar version so far (version 8). Pheeeew!

Mystery solved?

Maybe, maybe not.

One question that pops up now is: why is the "safe" version measurably faster than the "unsafe" version on the home PC (14802 vs 16223), even though the safe version does a little bit more work? What's going on here?

I was ready to move on but that stuff caught my eyes. I couldn't explain it so I stayed a bit longer. Observed. And designed a new test to confirm the theory that the mystery had indeed been solved.

If the unaligned loads were indeed responsible for making the code 2X faster, then switching back to *movups* **in version 9c** (while keeping the separate classes) should also bring back the performance of version 9b – on the home PC in particular.

But. It. Did. Not.

On the home PC, changing the *_mm_load_ps* to *_mm_loadu_ps* (while keeping the separate classes, i.e. while keeping the data buffers 16-byte aligned) gave these results:

Safe version:

> Complete test (box pruning): found 11811 intersections in 16513 K-cycles.

Unsafe version:

> Complete test (box pruning): found 11715 intersections in 19180 K-cycles.

i.e. the cost of unaligned loads for the safe version was 16513 - 14802 = 1711 K-Cycles.

And the cost of unaligned loads for the unsafe version was 19180 - 16223 = 2957 K-Cycles.

That last number is pretty similar to the cost of unaligned loads on the office PC (as we mentioned, ~2500 K-Cycles). So that number is pretty consistent, and seems to indicate the real cost of *movups* vs *movaps*: roughly 2000 to 3000 K-Cycles, on both machines.

But then...

why did we go from ~32000 to ~16000 K-Cycles between versions 9b and 9c on the home PC? Where is that coming from, if not from *movups*?

The mystery was not solved, after all.

New questions, new theories, new tests. The only other change we did was the switch from "*new*" to "*_aligned_malloc*". The SIMD_AABB_YZ has a force-inlined empty ctor, so this couldn't be a case of calling a slow compiler-generated ctor a large number of times (this things happen). Just to test the theory that it had something to do with this change though, I switched from "new" to "_aligned_malloc" in version 9b (*without* splitting the classes).

And lo and behold, this change alone made version 9b go from ~32000 K-Cycles to ~18000 K-Cycles at home (for both safe & unsafe versions).

So that's where the big performance drop was coming from!

After this line was identified, figuring out the "why" was a walk in the park. In short: the performance of *movups* depends on the alignment of the returned address.

In version 9b, on the home PC:

- If the bounds array is aligned on a 4-bytes boundary, the function takes ~33000 K-Cycles.
- If the bounds array is aligned on an 8-bytes boundary, the function takes ~19000 K-Cycles.
- If the bounds array is aligned on a 16-bytes boundary, the function takes ~18000 K-Cycles.

So there is a huge hit for the 4-bytes boundary case, and we were previously unknowingly hitting it. Switching to *_aligned_malloc*, or using *__declspec(align(N))* on the SIMD_AABB class, fixes the issue in version 9b.

Let's recap for the home PC:

Instruction	Alignment	Version	Timings
movups	data 4-bytes aligned	single class (version9b)	33000
movups	data 8-bytes aligned	single class (version9b)	19000
movups	data 16-bytes aligned	single class (version9b)	18000
movups	data 16-bytes aligned	separate classes (version9c safe)	16000
movups	data 16-bytes aligned	separate classes (version9c unsafe)	19000
movaps	data 16-bytes aligned	separate classes (version9c safe)	14000
movaps	data 16-bytes aligned	separate classes (version9c unsafe)	16000

What this all means is that the initial theory was actually correct (the performance cost was indeed because of unaligned loads), but there was a subtlety in there: there's the cost of the *movups* instruction itself compared to *movaps* (which is constant at ~2000 / 3000 K-Cycles) ***and*** the cost of the data-loading with *movups* (which is variable and sometimes considerably more expensive).

We also see that version 9b 16-bytes aligned can be faster than version 9c using *movups* (18000 vs 19000). That is probably because as we mentioned, version 9c needs to compute two addresses now.

And then finally, with *movaps*, the "safe" version 9c is still faster than the "unsafe" version 9c.

Why?

I still don't know.

I would have to try V-Tune at that point. For now, I will be happy to admit that I don't have an answer, and I don't see what test I could add to find one.

In any case the performance difference is not large, and since the fastest version is also the safest, there is little motivation to go to the bottom of it.

But the main reason for not bothering is that, ***spoiler***: that difference will vanish in the next post anyway.

What we learnt:

SIMD is hard. It took 3 versions for it to become faster than our scalar code.

The cost of unaligned loads can vary greatly depending on data alignment.

Data alignment has an impact on performance *even with unaligned loads*.