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

Part 12 – ASM FTW

Fasten your seatbelts; this one is going to be a bit bumpy. Last time we were left with unanswered questions and doubts about the quality of the compiler-generated code.

Those who know me probably saw it coming from a mile away: this time we'll rewrite the loop in assembly. Old-school, baby.

It is not always useful, but it is often informative and eye-opening.

To make things easier, we will limit ourselves to a small part of the loop. We are not going to rewrite the whole function. Just this snippet:

```
const SIMD_AABB_YZ& Box0YZ = BoxListYZ[Index0];
SIMD_OVERLAP_INIT(Box0YZ)

udword Index1 = RunningAddress;

while(BoxListX[Index1].mMinX<=MaxLimit)
{
    SIMD_OVERLAP_TEST(BoxListYZ[Index1])
        pairs.Add(RIndex0).Add(Remap[Index1]);
    Index1++;
}
```

And we're only going to do the "unsafe" version, since that's (slightly) less work.

The easiest way to do the conversion is to use inline assembly in *Visual Studio* (you know now why this whole experiment is using a *Win32* project - inline assembly is not supported anymore in *Win64* builds).

You just comment out lines and start replacing them with blocks of `_asm` code. The assembly code there can directly access some of the C++ variables, it's all rather simple. Sort of.

For example this:

```
const SIMD_AABB_YZ& Box0YZ = BoxListYZ[Index0];
SIMD_OVERLAP_INIT(Box0YZ)
```

Becomes:

```
_asm
{
    mov         edx, BoxListYZ              // edx = BoxListYZ
    mov         edi, Index0                 // edi = Index0
    add         edi, edi                    // edi = Index0 * 2
    lea         esi, dword ptr [edx+edi*8]  // esi = BoxListYZ + Index0*16 = &BoxListYZ[Index0] (*16 because sizeof(SIMD_AABB_YZ)==16)
    movaps      xmm2, xmmword ptr [esi]     // xmm2 = BoxListYZ[Index0] = Box0YZ <= that's the _mm_load_ps in SIMD_OVERLAP_INIT
    shufps      xmm2, xmm2, 4Eh             // that's our _mm_shuffle_ps in SIMD_OVERLAP_INIT
}
```

Then this:

```
while(BoxListX[Index1].mMinX<=MaxLimit)
{
    Index1++;
}
```

Becomes for example:

```
    const float MaxLimit = Box0X.mMaxX;
    _asm
    {
        movss       xmm1, MaxLimit              // xmm1 = MaxLimit

        mov         edi, RunningAddress         // edi = Index1

        mov         eax, BoxListX               // eax = BoxListX
        lea         esi, dword ptr [eax+edi*8]  // esi = BoxListX + Index1*8 = &BoxListX[Index1] (*8 because sizeof(SIMD_AABB_X)==8)
        comiss      xmm1, xmmword ptr [esi]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
        jb          ExitLoop

        align       16                          // Align start of loop on 16-byte boundary for perf
EnterLoop:

        inc         edi                         // Index1++
        add         esi, 8
        comiss      xmm1, xmmword ptr [esi]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
        jae         EnterLoop
ExitLoop:;
    }
```

And once the loop is done and you double-checked it iterates the proper number of times (it's easy to compare against the C code when both are right there in the same file), you just add the missing bits.

The SIMD overlap test:

```
SIMD_OVERLAP_TEST(BoxListYZ[Index1])
```

…will need some code before the loop to compute the initial address:

```
  mov         edi, RunningAddress     // edi = Index1
  mov         eax, edi                // eax = Index1
  shl         eax, 4                  // eax = Index1 * 16
  mov         edx, BoxListYZ          // edx = BoxListYZ
  add         edx, eax                // edx = &BoxListYZ[Index1]
```

And then the actual test inside the loop will be something like:

```
    movaps      xmm3, xmm2                  // xmm2 = pre-shuffled box, constant for the duration of the loop
    cmpltps     xmm3, xmmword ptr [edx]     // that's _mm_cmpgt_ps in SIMD_OVERLAP_TEST
    movmskps    eax, xmm3                   // that's _mm_movemask_ps in SIMD_OVERLAP_TEST
    cmp         eax, 0Ch
    jne         NoOverlap

    <some code here to output the pair>

    align       16
NoOverlap:;
```

At this point you don't immediately write the code to output the pair. First you just increase a counter (using a free register like maybe *ebx*) and you count the number of overlaps you get. Then you *printf* the results and compare it to the C code. Is the number correct? Good, then you can now complete the function and actually output the pair. Writing a *push_back* in assembly is tedious and pointless, so you don't actually need to do it. Just move the corresponding code to a C function like this:

```
static void __cdecl outputPair(udword id0, udword id1, Container& pairs, const udword* remap)
{
    pairs.Add(id0).Add(remap[id1]);
}
```

And then call it from the assembly block, like this:

```
  movaps      SavedXMM1, xmm1
  movaps      SavedXMM2, xmm2
  pushad

      push        Remap
      push        pairs
      push        edi
      push        RIndex0
      call        outputPair;
      add         esp, 16

  popad
  movaps      xmm1, SavedXMM1
  movaps      xmm2, SavedXMM2
```

So there are a couple of subtleties in there.

The called function must have a *__cdecl* signature, because we sent parameters to it via the stack. Alternatively, compile the whole project with the */Gd* compile option (it's the default anyway).

If you omit *__cdecl* in the function signature and compile the whole program with the *__fastcall* convention (*/Gr* compile option), then it will crash when trying to call "*outputPair*".

The parameters are passed in reverse order, that's the convention. Note how you can directly use the *Remap*, *pairs* and *RIndex0* variables from the assembly code. They're all C++ variables, the compiler

knows what to do. It's very convenient. After the call the "*add esp, 16*" fixes the stack pointer - it's 16 because we pushed four 4-bytes elements to the stack before calling the function.

Now the function call is wrapped by a *pushad*/*popad* couple. It's the lazy man's way to save and restore all registers to/from the stack. This makes sure that whatever happens in *outputPair*, it's not going to modify the values kept in our CPU registers.

And then the *movaps* on *xmm1* and *xmm2* do the same save & restore for the two *xmm* registers we care about. These ones are easy to forget because most of the time *outputPair* does not change the *xmm* registers.... but it does when there is a re-allocation within the container. If you are not aware of that, this can be a hard-to-find bug.

Right.

If you put all this together you get this first, "naive" version:

```
        const float MaxLimit = Box0X.mMaxX;
        const udword RIndex0 = Remap[Index0];

        __m128  SavedXMM1;
        __m128  SavedXMM2;

        _asm
        {
            movss       xmm1, MaxLimit              // xmm1 = MaxLimit
            mov         edx, BoxListYZ              // edx = BoxListYZ

            mov         edi, Index0                 // edi = Index0
            add         edi, edi                    // edi = Index0 * 2
            lea         esi, dword ptr [edx+edi*8]  // esi = BoxListYZ + Index0*16 = &BoxListYZ[Index0] (*16 because sizeof(SIMD_AABB_YZ)
            movaps      xmm2, xmmword ptr [esi]     // xmm2 = BoxListYZ[Index0] = Box0YZ <= that's the _mm_load_ps in SIMD_OVERLAP_INIT
            shufps      xmm2, xmm2, 4Eh             // that's our _mm_shuffle_ps in SIMD_OVERLAP_INIT

            mov         edi, RunningAddress         // edi = Index1
            mov         eax, edi                    // eax = Index1
            shl         eax, 4                      // eax = Index1 * 16
            add         edx, eax                    // edx = &BoxListYZ[Index1]

            mov         eax, BoxListX               // eax = BoxListX
            lea         esi, dword ptr [eax+edi*8]  // esi = BoxListX + Index1*8 = &BoxListX[Index1] (*8 because sizeof(SIMD_AABB_X)==8)
            comiss      xmm1, xmmword ptr [esi]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
            jb          ExitLoop

            align       16                          // Align start of loop on 16-byte boundary for perf
EnterLoop:
            movaps      xmm3, xmm2                  // xmm2 = pre-shuffled box, constant for the duration of the loop
            cmpltps     xmm3, xmmword ptr [edx]     // that's _mm_cmpgt_ps in SIMD_OVERLAP_TEST
            movmskps    eax, xmm3                   // that's _mm_movemask_ps in SIMD_OVERLAP_TEST

            cmp         eax, 0Ch
            jne         NoOverlap

            movaps      SavedXMM1, xmm1
            movaps      SavedXMM2, xmm2
            pushad
                push        Remap
                push        pairs
                push        edi
                push        RIndex0
                call        outputPair2;
                add         esp, 16
            popad
            movaps      xmm1, SavedXMM1
            movaps      xmm2, SavedXMM2

            align       16
NoOverlap:;
            inc         edi                         // Index1++
            add         esi, 8
            add         edx, 16
            comiss      xmm1, xmmword ptr [esi]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
            jae         EnterLoop
ExitLoop:;
        }
```

Let's see how it fares against the questions raised in part 11.

Question 1 was: *why write the code in such a way that it destroys the constant box?*

Answer: because we don't have a choice. We do the same in our assembly version: *xmm3* is also the constant box, and it's also destroyed by the *cmpltps* instruction, exactly like in the compiler-generated version. It turns out that there is no alternative. We used this intrinsic:

_mm_cmpgt_ps

But what we got in the assembly was:

cmpltps

Do you spot the difference? It's "greater than" in the intrinsic, and "lower than" in the actual assembly. It's the intrinsic trap again. You can see it here for example. The intrinsic is marked as CMPLTPSr, which means there is no actual *cmpgtps* instruction: it does not exist. So the compiler uses the "reverse" instruction instead (CMPLTPS) and swaps the arguments. That gives birth to the code we saw.

Beyond that, even if *cmpgtps* would exist, one way or another the comparison instruction would overwrite the contents of one register. It can be either our constant box, or the freshly loaded data for the new box. And both alternatives come down to the same amount of instructions: one load, and one cmp. We cannot do less.

So, it's a wash. The compiler was not to blame here, our question was simply naïve.

Question 2 was: *why reload the constant box from the stack? Why not keeping it in a register? The whole loop only uses two xmm registers (!) so it's not like there's a shortage of them.*

- ⇨ We fixed this. The constant box is now kept in *xmm2*, which is only saved and restored when an overlap occurs. We win.

Question 3 was: *why do you reload it from the stack each time? It's a constant for the whole loop. There are free available xmm registers. Just put it in xmm2, keep it there, and the "movss" instruction above just vanishes.*

- ⇨ We fixed this. This was about *MaxLimit*, which is now kept in *xmm1*. The "*movss*" disappeared indeed. We win.

Question 4 was about the *BoxListX* array being constantly written to the stack (and sometimes reloaded from it, with *VC10*).

- ⇨ We fixed this. It's now kept in the *esi* register at all time. We win.

Question 5 was: *why do you save edx to the stack all the time (address 003E3028)? Why don't you just save and restore it only when an overlap actually occurs?*

Question 6 was: *same question for ecx.*

- ⇨ We fixed this. We don't write any CPU registers to the stack, unless an overlap occurs. We win.

Sounds good!

The most often used codepath is now 10 instructions, compared to 13 before. Exciting!

But then you run the benchmark and....

Home PC:

```
Complete test (brute force): found 11811 intersections in 781774 K-cycles.
 15193 K-cycles.
 14574 K-cycles.
 14399 K-cycles.
 14310 K-cycles.
 14619 K-cycles.
 14366 K-cycles.
 14310 K-cycles.
 14322 K-cycles.
 14310 K-cycles.
 14310 K-cycles.
 14309 K-cycles.
 14740 K-cycles.
 14346 K-cycles.
 14311 K-cycles.
 14375 K-cycles.
 14374 K-cycles.
Complete test (box pruning): found 11715 intersections in 14309 K-cycles.
```

Office PC:

Complete test (brute force): found 11811 intersections in 812640 K-cycles.
13814 K-cycles.
13636 K-cycles.
13544 K-cycles.
13745 K-cycles.
13547 K-cycles.
13610 K-cycles.
13549 K-cycles.
13423 K-cycles.
12929 K-cycles.
13498 K-cycles.
13052 K-cycles.
13213 K-cycles.
12917 K-cycles.
13166 K-cycles.
13510 K-cycles.
13817 K-cycles.
Complete test (box pruning): found 11715 intersections in 12917 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
(Version10)	(16667)	-	-	-
Version11 - unsafe	14372	~1800	~11%	~6.87
Version11 - safe	14512	~200	~2%	~6.80
Version12	14309	-	-	~6.90

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
(Version10)	(15648)	-	-	-
Version11 - unsafe	12570	~100	~1%	~7.38
Version11 - safe	12611	-	-	~7.36
Version12	12917	-	-	~7.19

...it's pretty much exactly the same speed as before, or even marginally slower.

You didn't think it would be that easy, did you?

It's not.

But it was important to go through this process: it gives a better understanding of why the compiler made certain choices, and it shows why conventional wisdom says that it is hard to beat the compiler these days. Most of the time, the hand-written version is indeed slower than the compiler-generated version.

That would be the conclusion you would stop at, if this would be the conclusion you wanted to reach.

Well…

Old habits die hard and ex-demomakers cannot stop here I'm afraid.

The reality is that you cannot do worse than the compiler: just start from the compiler-generated version. How can you do worse if you copy what it did and improve from there?

The real benefit of the assembly version is that it opens the door for more tinkering. It's pretty hard (if not impossible) to make the compiler generate exactly what you want, to create various experiments. So

for example you cannot easily remove a specific instruction in the code flow to check its impact on performance. But the assembly version lets you do whatever you want, without interference.

And we're certainly going to exploit this.

What we did see so far, fair enough, is that our naive ideas about this code were wrong: keeping things in registers and avoiding spilling to the stack is apparently not such a big deal. This much is very true: these days it's very difficult to foresee how a change will affect performance. But that is exactly why the assembly version helps: it allows you to do very precise experiments, and see the impact of each line.

For example you can comment out the part that outputs the pairs, without the optimizing compiler removing the whole SIMD test when it discovers that it now can.

Do that and you discover that the timings barely change. These writes to output overlapping pairs are virtually free.

Continue doing this meticulously, and you will soon reach that one:

```
inc     edi     // Index1++
```

Comment it out, and…

```
Complete test (brute force): found 11811 intersections in 822828 K-cycles.
 11727 K-cycles.
 10996 K-cycles.
 11431 K-cycles.
 10930 K-cycles.
 10984 K-cycles.
 11271 K-cycles.
 11632 K-cycles.
 11412 K-cycles.
 11621 K-cycles.
 12076 K-cycles.
 11480 K-cycles.
 11624 K-cycles.
 11244 K-cycles.
 10963 K-cycles.
 11492 K-cycles.
 11074 K-cycles.
Complete test (box pruning): found 11715 intersections in 10930 K-cycles.
```

Wo-ah.

WO-AH.

This is now significantly faster, even though *edi* is only used when we output pairs. So it could be that not increasing *edi* removes some cache misses, because we use it as an index in the remap table, and if it's

constant we remap the same index all the time, so, less cache misses. Except we just saw that removing the whole block of code writing out the pairs barely had an impact on performance. Hmmm.

So, scratching head, observing, formulating a theory: is that because we use "*inc*" instead of "*add*" ? Testing: nope, replacing "*inc edi*" with "*add edi, 1*" doesn't change anything.

More theories led to more tests that didn't reveal anything. For some reason, increasing *edi* there is costly. I admit I still don't know why.

But at the end of the day, the "why" doesn't matter much. At this level it's a bit like quantum mechanics: there are a lot of weird rules at play, weird things happen, and trying to explain "why" in an intuitive way doesn't always work. What matters more is that the assembly version allowed us to discover a weakness in the code, and we would never have seen that in the C++ version, because it was only:

```
Index1++;
```

Which gave birth to three different adds:

```
003E301C  add       ecx,8
003E301F  add       edx,4
003E3022  add       edi,10h
```

We don't have any control over these assembly adds in the C++ code, because there's only one line there, and we cannot just remove it without breaking the whole loop. The assembly version however gives us control over the atomic instructions, for more fine-grained experiments. Experimenting at C++ level is like playing the piano with boxing gloves: you end up creating noise rather than music.

Note also how that one was found by accident: random changes. Random mutations, as if we'd be running a genetic algorithm on the code sequence. Random jumps to get out of a local minimum. I remember a competition that once took place in the office between me and a co-worker. At some point I came up with a counter-intuitive piece of code, full of branches, that turned out to be faster than all the branchless versions we had tried before. That co-worker told me something I'd never forget: "*I would never have tried that!*". It was said with a tone clearly expressing his disgust at what looked like a terrible idea and a terrible piece of code. And that's the lesson here: assumptions are the mother of all fuckups, you know that quote. When you're that close to the metal, sometimes it's good to forget what you think you know, and just try things. Even stupid things. *Especially* stupid things.

Sometimes, like here, it works. Increasing *edi* is slow? Fine, I don't need to know why: *I just won't do it*.

With this clear goal in mind, writing a new version doesn't take long.

The new strategy is to use a single offset (*ecx*) to address both the *BoxListX* and *BoxListYZ* arrays. Then, when an overlap occurs, we recompute the box index from that single offset, instead of increasing a counter each time. As a result, we can replace the inc / add / add from the previous version with just one

add. The most used codepath is now only 8 instructions (compared to the 13 from the compiler-generated version), and most importantly we don't touch *edi* anymore.

```
        const float MaxLimit = Box0X.mMaxX;
        const udword RIndex0 = Remap[Index0];
        __m128  SavedXMM1, SavedXMM2;

        _asm
        {
            movss       xmm1, MaxLimit              // xmm1 = MaxLimit
            mov         edx, BoxListYZ              // edx = BoxListYZ

            mov         edi, Index0                 // edi = Index0
            add         edi, edi                    // edi = Index0 * 2
            lea         esi, dword ptr [edx+edi*8]  // esi = BoxListYZ + Index0*16 = &BoxListYZ[Index0] (*16 because sizeof(SIMD_AABB_YZ)==16)
            movaps      xmm2, xmmword ptr [esi]     // xmm2 = BoxListYZ[Index0] = Box0YZ <= that's the _mm_load_ps in SIMD_OVERLAP_INIT
            shufps      xmm2, xmm2, 4Eh             // that's our _mm_shuffle_ps in SIMD_OVERLAP_INIT

            mov         edi, RunningAddress         // edi = Index1
            mov         eax, edi                    // eax = Index1
            shl         eax, 4                      // eax = Index1 * 16
            add         edx, eax                    // edx = &BoxListYZ[Index1]

            mov         eax, BoxListX               // eax = BoxListX
            lea         esi, dword ptr [eax+edi*8]  // esi = BoxListX + Index1*8 = &BoxListX[Index1] (*8 because sizeof(SIMD_AABB_X)==8)
            comiss      xmm1, xmmword ptr [esi]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
            jb          ExitLoop

            xor         ecx, ecx
            align       16                          // Align start of loop on 16-byte boundary for perf
    EnterLoop:
            movaps      xmm3, xmm2
            cmpltps     xmm3, xmmword ptr [edx+ecx*2]
            movmskps    eax, xmm3
            cmp         eax, 0Ch
            jne         NoOverlap

            movaps      SavedXMM1, xmm1
            movaps      SavedXMM2, xmm2
            pushad
                // Recompute Index1
                add         ecx, esi
                sub         ecx, BoxListX
                shr         ecx, 3

                push        Remap
                push        pairs
                push        ecx
                push        RIndex0
                call        outputPair;
                add         esp, 16
            popad
            movaps      xmm1, SavedXMM1
            movaps      xmm2, SavedXMM2

            align       16
    NoOverlap:;
            add         ecx, 8
            comiss      xmm1, xmmword ptr [esi+ecx]     // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit
            jae         EnterLoop
    ExitLoop:;
        }
```

So, did that work this time?

Yep:

Home PC:

Complete test (brute force): found 11811 intersections in 781991 K-cycles.
15514 K-cycles.
11767 K-cycles.
11733 K-cycles.
11734 K-cycles.
12045 K-cycles.
11758 K-cycles.
11737 K-cycles.
11736 K-cycles.
11748 K-cycles.
11744 K-cycles.
11733 K-cycles.
11736 K-cycles.
11755 K-cycles.
11758 K-cycles.
11736 K-cycles.
11731 K-cycles.
Complete test (box pruning): found 11725 intersections in 11731 K-cycles.

Office PC:

Complete test (brute force): found 11811 intersections in 820108 K-cycles.
10815 K-cycles.
10923 K-cycles.
10528 K-cycles.
10509 K-cycles.
10804 K-cycles.
10524 K-cycles.
10921 K-cycles.
10027 K-cycles.
10815 K-cycles.
10792 K-cycles.
10019 K-cycles.
10016 K-cycles.
10983 K-cycles.
10016 K-cycles.
10495 K-cycles.
10014 K-cycles.
Complete test (box pruning): found 11725 intersections in 10014 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
(Version10)	(16667)	-	-	-
Version11 - unsafe	14372	~1800	~11%	~6.87
Version11 - safe	14512	~200	~2%	~6.80
Version12 - first	14309	-	-	~6.90
Version12 - second	11731	~2600	~18%	~8.42

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
(Version10)	(15648)	-	-	-
Version11 - unsafe	12570	~100	~1%	~7.38
Version11 - safe	12611	-	-	~7.36
Version12 - first	12917	-	-	~7.19
Version12 - second	10014	~2500	~20%	~9.27

There you go! We're getting close to a 10X speedup...

Who said hand-written assembly cannot beat the compiler?

Now it would be tempting to be smug about it, conclude that "assembly rules" or something, that compilers are "lame", or any of the many things ex demo-coders are fond of saying.

But things in the real world are not that black-or-white, as I will demonstrate in the next post.

Stay tuned. We are not done.

What we learnt:

An assembly version is often useful to let us play with the code, experiment, and discover potential performance gains that wouldn't have been easy to spot with the C++ version.

Hand-written assembly can still be faster than the compiler-generated version. Or so it seems.