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

Part 14a – loop unrolling

The main codepath from version 13 had only 9 assembly instructions, and there wasn't much room left for improvements in terms of reducing the instructions count. Nonetheless we saw that doing so had a measurable impact on performance. At the same time, we identified a problem with the code alignment of our main loop, which was apparently not properly handled by the compiler.

In that situation, a typical thing to do is unroll the loop. It reduces the amount of branch instructions needed to loop. And of course, misaligned loops are less of a problem if we loop less.

Loop unrolling is pretty much the oldest trick in the book, but it still has its place today. On the *Atari ST* for example, things were pretty simple: any removed instruction gave an immediate performance gain. There were no cache effects, the code always took a very predictable amount of time, and the more you unrolled, the faster it became. No surprises. This led to certain abuses where loops got unrolled thousands of times, creating huge blocks of code that quickly consumed all the available memory. In the old days of *TASM* on PC, the *REPT* / *ENDM* directives could be used to unroll the code and achieve similar speedups.

These days however, processors are a little bit more complicated, subtle, unpredictable. And loop unrolling is not always a win. In particular, if you unroll too much, the code can become *slower*. The previously mentioned Intel optimization manual has some details about this, in chapter 3.4.1.7. It became more difficult to foresee and explain, but the basic strategy remains as simple as in the past, as we will see just now.

Our code at the end of version 13 looked like this:

```
        udword Offset = 0;
        const char* const CurrentBoxListYZ = (const char*)&BoxListYZ[RunningAddress];
        const char* const CurrentBoxListX = (const char*)&BoxListX[RunningAddress];

        _asm    align   16

        while(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)
        {
            const float* box = (const float*)(CurrentBoxListYZ + Offset*2);
#ifdef SAFE_VERSION
            if(_mm_movemask_ps(_mm_cmpngt_ps(b, _mm_load_ps(box)))==15)
#else
            if(_mm_movemask_ps(_mm_cmpnle_ps(_mm_load_ps(box), b))==12)
#endif
            {
                const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
                outputPair(RIndex0, Index, pairs, Remap);
            }

            Offset += 8;
        }
```

If we hide the safe/unsafe version business behind a trivial SIMD_OVERLAP_TEST macro, unrolling that loop twice can be as easy as simply copy-pasting the inner block, like this (VERSION1_UNROLL2 in the code):

```
_asm    align   16

while(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)
{
    const float* box = (const float*)(CurrentBoxListYZ + Offset*2);
    if(SIMD_OVERLAP_TEST)
    {
        const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
        outputPair(RIndex0, Index, pairs, Remap);
    }
    Offset += 8;

    // START OF ADDED BLOCK
    if(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)
    {
        const float* box = (const float*)(CurrentBoxListYZ + Offset*2);
        if(SIMD_OVERLAP_TEST)
        {
            const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
            outputPair(RIndex0, Index, pairs, Remap);
        }
        Offset += 8;
    }
    // END OF ADDED BLOCK
}
```

This is as easy as it gets, and in this specific case it already provides performance gains! *However*, unroll it exactly the same way but three times (VERSION1_UNROLL3 in the code) and it becomes slower again. That's a neat illustration of what we said just before: unrolling the loop too much can hurt performance. Here is a quick summary of timings for the office PC, safe version (new entries in **bold** letters):

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	92885	0	0%	1.0
…	…	…	…	…
Version13 - safe	10053	~2500	~20%	~9.23
Version14a - VERSION1_UNROLL2	**9368**	**~685**	**~6%**	**~9.91**
(Version14a - VERSION1_UNROLL3)	**(10263)**	-	-	-

So what is wrong with unrolling the loop three times here? Looking at the disassembly reveals a potential problem: the part that outputs the pair is replicated three times inside the loop, making the code quite large. The *Intel optimization manual* warns about this, for example stating that "unrolling can be harmful if the unrolled loop no longer fits in the trace cache". Whether this is what actually happens here or not is not terribly relevant: the important point is that it gives us a new theory to test, and a new

direction for our next attempt. Refactoring the code so that the block that outputs the pair moves out of the loop gives birth to version 2, here unrolled twice (VERSION2_UNROLL2):

```
        goto LoopStart;
OverlapFound:
        const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
        outputPair(RIndex0, Index, pairs, Remap);
        Offset += 8;

        _asm    align   16
LoopStart:
        while(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)
        {
            const float* box = (const float*)(CurrentBoxListYZ + Offset*2);
            if(SIMD_OVERLAP_TEST)
                goto OverlapFound;

            Offset += 8;

            if(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)
            {
                const float* box = (const float*)(CurrentBoxListYZ + Offset*2);
                if(SIMD_OVERLAP_TEST)
                    goto OverlapFound;

                Offset += 8;
            }
        }
```

As you can see it is very similar to the previous version: the only difference is that the part where an overlap is found has been clearly moved out of the way, before the loop even starts. When an overlap is found, we jump there and fallback to the start of the loop.

The presence of `goto` should not be frowned upon: as I've said before, you must see through the C++, and consider the generated assembly. Generally speaking the assembly is shock-full of jump instructions (i.e. `goto`) anyway. And after version 13 where we tried to mimic the desired assembly in C++, the appearance of `goto` here should not be a surprise.

In any case: it's 2017, I don't have the time or energy to discuss once again whether `goto` should be considered harmful. It's a tool. It serves a purpose. If you can avoid it, do so, but if you cannot, don't be ashamed. People for whom this is a deal-breaker can stick to version 13. Or maybe look out for a future version in a future post that would perhaps drop them – I don't know yet.

Now, as we did with version 1, we can try to unroll the loop twice (VERSION2_UNROLL2) and three times (VERSION2_UNROLL3). The results for the office PC / safe version are:

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	92885	0	0%	1.0
…	…	…	…	…
Version13 - safe	10053	~2500	~20%	~9.23
Version14a - VERSION1_UNROLL2	9368	~680	~6%	~9.91
(Version14a - VERSION1_UNROLL3)	(10147)	-	-	-
Version14a - VERSION2_UNROLL2	**9051**	**~1000**	**~10%**	**~10.26**
Version14a - VERSION2_UNROLL3	**8802**	**~1200**	**~12%**	**~10.55**
(Version14a - VERSION2_UNROLL4)	**(9638)**	**-**	**-**	**-**

All "Version 14a" entries are compared to Version 13 here, not to each-other.

What we see seems to confirm the theory. Looking at the disassembly, the output-pair-related code did move out of the loop, making it tighter overall. As a result, unrolling version 2 three times still gives performance gains. But if we try to unroll it one more time (VERSION2_UNROLL4), we see the performance drop again. It's the same pattern as with version 1: at some point the unrolled code apparently becomes too large and starts to be slower.

In any case, the assembly for VERSION2_UNROLL3 looks pretty good:

```
01103002  comiss      xmm1,dword ptr [edi+esi]
01103006  jb          LoopStart+96h (01103098h)
0110300C  movaps      xmm0,xmmword ptr [ecx+esi*2]
01103010  cmpnltps    xmm0,xmm2
01103014  movmskps    eax,xmm0
01103017  cmp         eax,0Fh
0110301A  je          LoopStart+51h (01103053h)
0110301C  add         esi,8

0110301F  comiss      xmm1,dword ptr [edi+esi]
01103023  jb          LoopStart+96h (01103098h)
01103025  movaps      xmm0,xmmword ptr [ecx+esi*2]
01103029  cmpnltps    xmm0,xmm2
0110302D  movmskps    eax,xmm0
01103030  cmp         eax,0Fh
01103033  je          LoopStart+51h (01103053h)
01103035  add         esi,8

01103038  comiss      xmm1,dword ptr [edi+esi]
0110303C  jb          LoopStart+96h (01103098h)
0110303E  movaps      xmm0,xmmword ptr [ecx+esi*2]
01103042  cmpnltps    xmm0,xmm2
01103046  movmskps    eax,xmm0
01103049  cmp         eax,0Fh
0110304C  je          LoopStart+51h (01103053h)
0110304E  add         esi,8

01103051  jmp         LoopStart (01103002h)
```

It is color-coded the same way as in previous posts, to show where all the bits and pieces moved. The compiler did exactly what we wanted, unrolling the code three times in the obvious way. As expected, the green block from version 13 (the part that outputs pairs) moved out of the loop. Perhaps as a result, the compiler *finally* stopped spilling our registers to the stack, and we now use **8** instructions per test (as opposed to **9** in version 13).

So the good news is that we found an easy way to make the code faster. And in fact, this version is a significant milestone: *we just reached our 10X speedup target!*

The bad news is that this code size limit might vary from one platform to the next, so without testing on each of them there is no guarantee that what we are doing here actually helps. It might very well be the case that `VERSION2_UNROLL3` is faster on PC for example, but slower on some console – or on some other PC with another processor. In that respect, loop unrolling is a perhaps more fragile and debatable optimization than things like avoiding cache misses, etc, which are more universally beneficial. Still, we started the questionable optimizations with version 13 already, and this isn't worse.

To test various amount of unrolling more easily on each platform, it is perhaps useful to capture the common bits in a macro. This is what `VERSION2_UNROLL` does, and it admittedly starts to look slightly evil:

```
#define BLOCK   if(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)          \
                {const float* box = (const float*)(CurrentBoxListYZ + Offset*2);\
                    if(SIMD_OVERLAP_TEST)                                       \
                        goto OverlapFound;                                      \
                    Offset += 8;
        goto LoopStart;
OverlapFound:
        const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
        outputPair(RIndex0, Index, pairs, Remap);
        Offset += 8;

        _asm    align   16
LoopStart:
        BLOCK
            BLOCK
                BLOCK
                }
            }
            goto LoopStart;
        }
```

The last `goto` has been added because the initial `while` directive disappeared, replaced with the `if` within the `BLOCK` macro. Amazingly, this "innocent change" is enough to significantly change the generated assembly. Fortunately it doesn't change the timings - this form is perhaps even a tiny bit faster:

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	92885	0	0%	1.0
…	…	…	…	…
Version13 - safe	10053	~2500	~20%	~9.23
Version14a - VERSION1_UNROLL2	9368	~680	~6%	~9.91
(Version14a - VERSION1_UNROLL3)	(10147)	-	-	-
Version14a - VERSION2_UNROLL2	9051	~1000	~10%	~10.26
Version14a - VERSION2_UNROLL3	8802	~1200	~12%	~10.55
(Version14a - VERSION2_UNROLL4)	(9638)	-	-	-
Version14a - VERSION2_UNROLL	**8738**	**~1300**	**~13%**	**~10.63**

It looks quite fine.

However… running the same tests on the home PC exposes the main issue with loop unrolling:

Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	98822	0	0%	1.0
…	…	…	…	…
Version13 - safe	12236	~2200	~15%	~8.07
Version14a - VERSION1_UNROLL2	10573	~1600	~13%	~9.34
Version14a - VERSION1_UNROLL3	11388	~840	~7%	~8.67
Version14a - VERSION2_UNROLL2	11290	~940	~7%	~8.75
Version14a - VERSION2_UNROLL3	10425	~1800	~14%	~9.47
Version14a - VERSION2_UNROLL4	9598	~2600	~21%	~10.29
Version14a - VERSION2_UNROLL	9691	~2500	~20%	~10.19

The results are similar…. yet so subtly different. In particular:

- Unrolling version 2 four times (VERSION2_UNROLL4) still provides clear performance gains, while it made the code slower on the office PC.

- VERSION2_UNROLL4 and VERSION2_UNROLL are almost the same speed… even though one is unrolled one more time! In theory VERSION2_UNROLL should be equivalent to VERSION2_UNROLL3 (as on the office PC). But for some unknown reason it is clearly faster here.

These observations gave birth to a new version, VERSION3, which copies VERSION2_UNROLL but unrolls the loop *five* times – the unrolling limit on the home PC.

And lo and behold…

Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	98822	0	0%	1.0
…	…	…	…	…
Version13 - safe	12236	~2200	~15%	~8.07
Version14a - VERSION1_UNROLL2	10573	~1600	~13%	~9.34
Version14a - VERSION1_UNROLL3	11388	~840	~7%	~8.67
Version14a - VERSION2_UNROLL2	11290	~940	~7%	~8.75
Version14a - VERSION2_UNROLL3	10425	~1800	~14%	~9.47
Version14a - VERSION2_UNROLL4	9598	~2600	~21%	~10.29
Version14a - VERSION2_UNROLL	9691	~2500	~20%	~10.19
Version14a - VERSION3	**9012**	**~3200**	**~26%**	**~10.96**

That's quite a speedup! I cannot explain it but I'll take it. Running this new version on the office PC gives:

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	92885	0	0%	1.0
…	…	…	…	…
Version13 - safe	10053	~2500	~20%	~9.23
Version14a - VERSION1_UNROLL2	9368	~680	~6%	~9.91
(Version14a - VERSION1_UNROLL3)	(10147)	-	-	-
Version14a - VERSION2_UNROLL2	9051	~1000	~10%	~10.26
Version14a - VERSION2_UNROLL3	8802	~1200	~12%	~10.55
(Version14a - VERSION2_UNROLL4)	(9638)	-	-	-
Version14a - VERSION2_UNROLL	8738	~1300	~13%	~10.63
Version14a - VERSION3	**8532**	**~1500**	**~15%**	**~10.88**

So the gains for **VERSION3** are minimal on this machine, but we got lucky: at least this version isn't slower. Sometimes you end up with different "best" versions on different machines. That is probably the biggest problem with loop unrolling: until you test "everywhere", selecting a "winner" is a bit like a shot in the dark.

What one can do in difficult cases is to look at the numbers: sometimes the best version on one machine is still slower (in absolute number of cycles) than sub-optimal versions on another machine. So one

strategy here is to select the fastest version on the slowest machine, and accept whatever results it gives on the faster machine. That way you make sure that your choices don't penalize the machines that need the optimizations the most.

In our case here, we got lucky anyway: `VERSION3` wins, and this is the version we will continue to work with.

But what is there left to do anyway?

Well, looking at the previous disassembly, another obvious thing to try pops up: increasing *esi* for each box seems useless. We could use the relevant offsets in the address calculations, increase *esi* only once in the end, thus saving some extra instructions and making the code tighter.

That's the plan for the next post. However, before tackling this in the C++ version, we will revisit the assembly version thanks to a special guest.

What we learnt:

Loop unrolling still has its uses in 2017.

But it is a lot trickier than in the past. The optimal amount of unrolling varies from one machine to the next, and unrolling too much can actually make the code run slower.

We reached our 10X speedup target!