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

Part 14c – that's how I roll

Our goal today is to look at the details of what Fabian "Ryg" Giesen did in version 14b (an assembly version), and replicate them in our C++ unrolled version (14a) if possible.

First, let's get one thing out of the way: I will not switch back to integer comparisons in this post. I like to do one optimization at a time, as you can probably tell by now, so I will leave this stuff for later. This means we can ignore the `MungeFloat` function and the integer-related changes in Fabian's code.

Then, the first thing you can see is that the code has been separated in two distinct loops: a fast one (starting with the `FastLoop` label), and a safe one (starting with the `CarefulLoop` label).

```
FastLoop:
            cmp         edi, [esi+ecx+24]   // [esi] = BoxListX[Index1].mMinX, compared to MaxLimit - can safely do another 4 iters of this?
            jb          CarefulLoop         // nope!

            ...

            jmp         FastLoop
CarefulLoop:
```

One problem when unrolling the initial loop is that we don't know ahead of time how many iterations we will have to do (it can stop at any time depending on the value of X we read from the buffer). It is much easier to unroll loops that are executed a known number of times when the loop starts.

Sometimes in this situation, one can use what I call the "radix sort strategy": just use two passes. Count how many iterations or items you will have to deal with in a first pass, then do a second pass taking advantage of the knowledge. That's what a radix-sort does, creating counters and histograms in a first pass. But that kind of approach does not work well here (or at least I didn't manage to make it work).

Fabian's approach is to just "look ahead" and check that the buffer still has at least 4 valid entries. If it does, he uses the "fast loop". Otherwise he falls back to the "safe loop", which is actually just our regular non-unrolled loop from version 12. In order to look ahead safely, the sentinel values are replicated as many times as we want to unroll the loop. This is a rather simple change in the non-assembly part of the code. First there:

```
SIMD_AABB_X* BoxListX = new SIMD_AABB_X[nb+5];
```

And then there:

```
BoxListX[nb+1].mMinX = ~0u;
BoxListX[nb+2].mMinX = ~0u;
BoxListX[nb+3].mMinX = ~0u;
BoxListX[nb+4].mMinX = ~0u;
```

That's not assembly so no problem porting this bit to the C++ version.

Now, the "fast loop" is fast for *three* different reasons. First, it is unrolled four times, getting rid of the corresponding branching instructions – same as in our version 14a. Second, because we looked ahead

and we know the four next input values are all valid, the tests against the *MaxLimit* value can also be removed. And finally, the idea we wanted to test at the end of 14a has also been implemented, i.e. we don't need to increase the *Offset* value for each box (we can encode that directly into the address calculation).

At the end of the day, the core loop in Fabian's version is thus:

```
// Unroll 0
movaps      xmm3, xmmword ptr [edx+ecx*2+0]  // Box1YZ
cmpnleps    xmm3, xmm2
movmskps    eax, xmm3
cmp         eax, 0Ch
je          FoundSlot0

// Unroll 1
movaps      xmm3, xmmword ptr [edx+ecx*2+16] // Box1YZ
cmpnleps    xmm3, xmm2
movmskps    eax, xmm3
cmp         eax, 0Ch
je          FoundSlot1

// Unroll 2
movaps      xmm3, xmmword ptr [edx+ecx*2+32]  // Box1YZ
cmpnleps    xmm3, xmm2
movmskps    eax, xmm3
cmp         eax, 0Ch
je          FoundSlot2

// Unroll 3
movaps      xmm3, xmmword ptr [edx+ecx*2+48]  // Box1YZ
add         ecx, 32                           // Advance
cmpnleps    xmm3, xmm2
movmskps    eax, xmm3
cmp         eax, 0Ch
jne         FastLoop
```

That is only **5** instructions per box, compared to the **8** we got in version 14a. Color-coding it reveals what happened: in the same way that we moved the green blocks out of the loop in version 14a, Fabian's version moved the blue blocks out of the (fast) loop. There is only one surviving blue instruction, to increase our offset only once for 4 boxes.

Pretty neat.

In our C++ code it would mean that the two lines **marked in bold letters** would / should vanish from our BLOCK macro:

```
#define BLOCK if(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)              \
              {const float* box = (const float*)(CurrentBoxListYZ + Offset*2);     \
              if(SIMD_OVERLAP_TEST)                                                \
                    goto BeforeLoop;                                               \
              Offset += 8;
```

Now another difference is that since we don't increase the offset each time, we cannot jump to the same address at each stage of the unrolled code. You can see that in Fabian's code, which jumps to different labels (*FoundSlot0*, *FoundSlot1*, *FoundSlot2*, or *FastFoundOne*). This is easy to replicate in C++ using `goto`. If you don't want to use `goto`, well, good luck.

And that's pretty much it. Let's try to replicate this in C++.

As we said, replicating the setup code is trivial (it was already done in C++).

For the safe loop, we are actually going to use our previous unrolled `VERSION3` from part 14a. In that respect this is an improvement over Fabian's code: even our safe loop is unrolled. From an implementation perspective it couldn't be more trivial: we just leave the code from part 14a as-is, and start writing another "fast" unrolled loop just before – the fallback to the safe loop happens naturally.

Now for our fast loop, we transform the `BLOCK` macro as expected from the previous analysis:

```
#define BLOCK4(x, label)                                                    \
{const float* box = (const float*)(CurrentBoxListYZ + Offset*2 + x*2);      \
if(SIMD_OVERLAP_TEST)                                                       \
       goto label;   }
```

As we mentioned, the lines previously marked in bold vanished. Then we added two extra parameters: one ("x") to include the offset directly in the address calculation (as we wanted to do at the end of version 14a, and as is done Fabian's code), and another one ("label") to make the code jump to a different address like in the assembly version.

Now, one small improvement over Fabian's code is that we will put the "overlap found" code *before* the fast loop starts, not after it ends. That's what we did in version 14a already, and it saves one jump.

Another improvement is that we're going to unroll 5 times instead of 4, as we did in version 14a. That's where using `BLOCK` macros pays off: unrolling one more time is easy and doesn't expand the code too much.

After all is said and done, the code becomes:

```
#define VERSION3     // Enable this as our safe loop
#define BLOCK4(x, label)                                                    \
{const float* box = (const float*)(CurrentBoxListYZ + Offset*2 + x*2);      \
if(SIMD_OVERLAP_TEST)       goto label;   }

              goto StartLoop4;
              _asm   align 16
FoundOverlap3:
              Offset += 8;
              _asm   align 16
FoundOverlap2:
              Offset += 8;
              _asm   align 16
FoundOverlap1:
              Offset += 8;
              _asm   align 16
```

```
FoundOverlap0:
              Offset += 8;
              _asm   align 16
FoundOverlap:
              {const udword Index = (CurrentBoxListX + Offset - 8 - (const
char*)BoxListX)>>3;
              pairs.Add(RIndex0).Add(Remap[Index]);
              }
              _asm   align 16
StartLoop4:
              while(*(const float*)(CurrentBoxListX + Offset + 8*5)<=MaxLimit)
              {
                     BLOCK4(0, FoundOverlap0)
                     BLOCK4(8, FoundOverlap1)
                     BLOCK4(16, FoundOverlap2)
                     BLOCK4(24, FoundOverlap3)

                     Offset += 40;
                     BLOCK4(-8, FoundOverlap)
              }

// The following code from Version 14a becomes our safe loop
#ifdef VERSION3
#define BLOCK if(*(const float*)(CurrentBoxListX + Offset)<=MaxLimit)\
{const float* box = (const float*)(CurrentBoxListYZ + Offset*2);     \
if(SIMD_OVERLAP_TEST)                                                \
       goto BeforeLoop;                                              \
Offset += 8;

              goto StartLoop;
              _asm   align 16
BeforeLoop:
              {const udword Index = (CurrentBoxListX + Offset - (const char*)BoxListX)>>3;
              outputPair(RIndex0, Index, pairs, Remap);
              Offset += 8;
              }
              _asm   align 16
StartLoop:
              BLOCK
                     BLOCK
                            BLOCK
                                   BLOCK
                                          BLOCK
                                          }
                                   }
                            }
                     }
                     goto StartLoop;
              }
#endif
```

I know what you're going to say (hell, I know what you *did* say after I posted a preview of part 14): it looks horrible.

Sure, sure. But once again: see through the C++, and check out the disassembly for our fast loop:

```
001E30B0  comiss      xmm2,dword ptr [edi+esi+28h]
001E30B5  jb          StartLoop4+12Fh (01E31D4h)
            {
                  BLOCK4(0, FoundOverlap0)
001E30BB  movaps      xmm0,xmmword ptr [ecx-20h]
001E30BF  cmpnltps    xmm0,xmm1
001E30C3  movmskps    eax,xmm0
001E30C6  cmp         eax,0Fh
001E30C9  je          StartLoop4+9Bh (01E3140h)
                  BLOCK4(8, FoundOverlap1)
001E30CB  movaps      xmm0,xmmword ptr [ecx-10h]
001E30CF  cmpnltps    xmm0,xmm1
001E30D3  movmskps    eax,xmm0
001E30D6  cmp         eax,0Fh
001E30D9  je          StartLoop4+8Bh (01E3130h)
                  BLOCK4(16, FoundOverlap2)
001E30DB  movaps      xmm0,xmmword ptr [ecx]
001E30DE  cmpnltps    xmm0,xmm1
001E30E2  movmskps    eax,xmm0
001E30E5  cmp         eax,0Fh
001E30E8  je          StartLoop4+7Bh (01E3120h)
                  BLOCK4(24, FoundOverlap3)
001E30EA  movaps      xmm0,xmmword ptr [ecx+10h]
001E30EE  cmpnltps    xmm0,xmm1
001E30F2  movmskps    eax,xmm0
001E30F5  cmp         eax,0Fh
001E30F8  je          StartLoop4+6Dh (01E3112h)
//                BLOCK4(32, FoundOverlap4)

                  Offset += 40;
                  BLOCK4(-8, FoundOverlap)
001E30FA  movaps      xmm0,xmmword ptr [ecx+20h]
001E30FE  add         ecx,50h
001E3101  cmpnltps    xmm0,xmm1
001E3105  add         esi,28h
001E3108  movmskps    eax,xmm0
001E310B  cmp         eax,0Fh
001E310E  jne         StartLoop4+0Bh (01E30B0h)
            }
001E3110  jmp         StartLoop4+0ABh (01E3150h)
```

That's pretty much perfect.

We get an initial *comiss* instruction instead of *cmp* because we didn't bother switching X's to integers, and we see the loop has been unrolled 5 times instead of 4, but other than that it's virtually the same as Fabian's code, which is what we wanted.

We get the following results:

Home PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	98822	0	0%	1.0
…	…	…	…	…
(Version12 – 2nd)	(11731)	(~2600)	(~18%)	(~8.42)
Version13 - safe	12236	~2200	~15%	~8.07
Version14a - VERSION3	9012	~3200	~26%	~10.96
Version14b – Ryg/Unsafe	7600	~4100	~35%	~13.00
Version14c - safe	**7558**	**~4600**	**~38%**	**~13.07**

Office PC	Timings (K-Cycles)	Delta (K-Cycles)	Speedup	Overall X factor
Version2 - base	92885	0	0%	1.0
…	…	…	…	…
(Version12 – 2nd)	(10014)	(~2500)	(~20%)	(~9.27)
Version13 - safe	10053	~2500	~20%	~9.23
Version14a - VERSION3	8532	~1500	~15%	~10.88
Version14b – Ryg/Unsafe	7641	~2300	~23%	~12.15
Version14c - safe	**7255**	**~2700**	**~27%**	**~12.80**

The deltas in the results are compared to version 13, similar to what we did for version 14a.

Thanks to our small improvements, this new version is actually faster than version 14b (at least on my machines) – without using integers! As a bonus, this is based on the “safe” version 14a rather than the “unsafe” version 12.

What we learnt:

Once again the assembly version showed us the way. I am not sure I would have “seen” how to do this one without an assembly model I could copy.

Ugly C++ can generate pretty good looking assembly – and vice versa.

Unrolling is like SIMD: tricky. It’s easy to get gains from some basic unrolling but writing the optimal unrolled loop is quite another story.

Stay tuned. In the next post we will complete our port of Fabian’s code to C++, and revisit integer comparisons.