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

Part 14d – integer comparisons redux

In this part, we complete the port of Fabian Giesen's code (version 14b) to C++.

In part 7 we replaced float comparisons with integer comparisons (a very old trick), but dismissed the results because the gains were too small to justify the increase in code complexity. However we made the code significantly faster since then, so the relatively small gains we got at the time might be more interesting today.

Moreover, Fabian's version uses integers for the X's values only. This may also be a more interesting strategy than using them for everything, like in version 7.

Let's try!

Replicating this in the C++ version is trivial. Read part 7 again for the details. The only difference is that Fabian uses a different function to encode the floats:

```
// Munge the float bits to return produce an unsigned order-preserving
// ranking of floating-point numbers.
// (Old trick: http://stereopsis.com/radix.html FloatFlip, with a new
// spin to get rid of -0.0f)

// In /fp:precise, we can just calc "x + 0.0f" and get what we need.
// But fast math optimizes it away. Could use #pragma float_control,
// but that prohibits inlining of MungeFloat. So do this silly thing
// instead.
float g_global_this_always_zero = 0.0f;

static inline udword MungeFloat(float f)
{
    union
    {
        float f;
        udword u;
        sdword s;
    } u;
    u.f = f + g_global_this_always_zero;  // NOT a nop! Canonicalizes -0.0f to +0.0f
    udword toggle = (u.s >> 31) | (1u << 31);
    return u.u ^ toggle;
}
```

While my version from part 7 was simply:

```
static __forceinline udword encodeFloat(udword ir)
{
	if(ir & 0x80000000) //negative?
		return ~ir;//reverse sequence of negative numbers
	else
		return ir | 0x80000000; // flip sign
}
```

So it's pretty much the same: given the same input float, the two functions return the same integer value, except for -0.0f. But that's because Fabian's code transforms -0.0f to +0.0f before doing the conversion to integer, it's not a side-effect of the conversion itself.

This is not *really* needed, since both the sorting code and the pruning code can deal with -0.0f just fine. However it is technically more correct, i.e. more in line with what float comparisons would give, since with floats a positive zero is equal to a negative zero. So it is technically more correct to map both to the same integer value – which is what Fabian's code does.

In practice, it means that my version could produce "incorrect" results when positive and negative zeros are involved in the box coordinates. But it would only happen in edge cases where boxes exactly touch, so it would be as "incorrect" as our "unsafe" versions from past posts, and we already explained why they weren't a big issue. Still, Fabian's version is technically superior, even if the code does look a bit silly indeed - but in a cute kind of way.

Now a perhaps more interesting thing to note is that Fabian's version (well, Michael Herf's version I suppose) is branchless. So could it be measurably faster?

Without further ado, here are the results on my machines – new entries in **bold** letters:

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
Version14b – Ryg/Unsafe	7600	~4100	~35%	~13.00
Version14c - safe	7558	~4600	~38%	~13.07
Version14d - P	**7211**	**~340**	**~4%**	**~13.70**
Version14d - F	**7386**	**~170**	**~2%**	**~13.37**

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
Version14b – Ryg/Unsafe	7641	~2300	~23%	~12.15
Version14c - safe	7255	~2700	~27%	~12.80
Version14d - P	**7036**	**~210**	**~3%**	**~13.20**
Version14d - F	**6961**	**~290**	**~4%**	**~13.34**

Version 14d uses integer comparisons for X's. The P variant uses Pierre's encoding function ("`encodeFloat`"), while the F variant uses Fabian's ("`MungeFloat`"). The deltas are computed against Version 14c this time, to measure the speedup due to integer comparisons (rather than the speedup due to loop unrolling + integer comparisons).

The first thing we see is that indeed, using integer comparisons is measurably faster. This is not a big surprise since we saw the same in Version 7. But the gains are still very small (in particular, smaller than the theoretical 6% predicted by Ryg's analysis) and to be honest I would probably still ignore them at this point. But using integers just for X's is easy and doesn't come with the tedious switch to integer SIMD intrinsics, so it's probably not a big deal to keep them in that case.

On the other hand…

For some reason "`encodeFloat`" is faster on my home PC, while on my office PC it's slower (and "`MungeFloat`" wins). This is unfortunate and slightly annoying. This is the kind of complications that I don't mind dealing with if the gains are important, but for such small gains it starts to be a lot of trouble for minor rewards. I suppose I could simply pick up Ryg's version because it's more correct, and call it a day. That gives a nicely consistent overall X factor (13.37 vs 13.34) on the two machines.

And with this, we complete the port of Fabian's assembly version to C++. Our new goal has been reached: we're faster than version 14b now… at least on these two machines.

What we learnt:

An optimization that didn't provide "significant gains" in the past might be worth revisiting after all the other, larger optimizations have been applied.

Similarly, we are slowly reaching a point where small differences in the setup code become measurable and worth investigating. There might be new optimization opportunities there. For example the question we previously dismissed about what kind of sorting algorithm we should use might soon come back to the table.

In any case, for now we reached our initial goal (make the code an order of magnitude faster), and we reached our secondary goal (make the C++ code faster than Ryg's assembly version).

Surely we're done now!?
How much longer can we keep this going?

Well… Don't panic, but there is still a lot to do.

Stay tuned!