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

Part 16 – improved pair reporting

In Part 15 we saw that there were some cycles to save in the part of the code that reports pairs.

By nature, the AVX version had to report multiple results at the same time. Thus in this context it was natural to end up modifying the part that reports overlaps.

But there was no incentive for me to touch this before, because that part of the code is not something that would survive as-is in a production version. In the real world (e.g. in an actual physics engine), the part reporting overlapping pairs tends to be more complicated (and much slower) anyway. In some cases, you need to output results to a hash-map instead of just a dynamic vector. In some cases, you need to do more complex filtering before accepting a pair. Sometimes that means going through a user-defined callback to let customers apply their own logic and decide whether they want the pair or not. All these things are much slower than our simple pair reporting, so basically this is not the place we should bother optimizing.

That being said, the code reporting overlaps does have obvious issues and in the context of this project it is certainly fair game to address them. So, let's revisit this now.

So far we have used a simple container class to report our pairs, which was basically equivalent to an *std::vector<int>*. (If you wonder why I didn't use an actual *std::vector*, please go read Part 11 again).

The code for adding an *int* to the container (equivalent to a simple *push_back*) was:

```
inline_		Container&		Add(udword entry)
		{
			// Resize if needed
			if(mCurNbEntries==mMaxNbEntries)
				Resize();

			// Add new entry
			mEntries[mCurNbEntries++]	= entry;
			return *this;
		}
```

And the code to add a pair to the array was something like:

```
	pairs.Add(id0).Add(id1);
```

So there are two obvious issues here:

- Because we add the two integers making up a pair separately, we do two resize checks (two comparisons) per pair instead of one. Now the way the benchmark is setup, we resize the array during the first run, but never again afterwards. So in all subsequent runs we always take the same branch, and in theory there shouldn't be any misprediction. But still, that's one more comparison & jump than necessary.

- We update *mCurNbEntries* twice per pair instead of one. Since this is a class member, this is exactly the kind of things that would have given a bad *load-hit-store* (LHS) on Xbox 360. I did not investigate to see what kind of penalty (if any) it produced on PC in this case, but regardless: we can do better.

I am aware that none of these issues would have appeared if we would have used a standard *std::vector<Pair>* for example. However, we would have had other issues - as seen in a previous report. I used this non-templated custom array class in the original code simply because this is the only one I was familiar with back in 2002 (and the class itself was older than that, records show it was written around February 2000. Things were different back then).

In any case, we can address these problems in a number of ways. I just picked an easy one, and I now create a small wrapper passed to the collision code:

```
class Pairs
{
	public:
		Pairs(Container& container) : mContainer(container)	{}

	inline_		void	AddPair(udword p0, udword p1)
			{
				// Resize if needed
				const udword CurrentCapacity = mContainer.mMaxNbEntries;
				const udword CurrentSize = mContainer.mCurNbEntries;
				if(CurrentSize+2>CurrentCapacity)
					mContainer.Resize(2);

				// Add new pair
				udword* Dst = mContainer.mEntries + CurrentSize;
				Dst[0] = p0;
				Dst[1] = p1;
				mContainer.mCurNbEntries = CurrentSize+2;
			}

	Container&	mContainer;
};
```

And the pairs are now reported as you'd expect from reading the code:

```
	pairs.AddPair(id0, id1);
```

So it just does one comparison and one class member read-modify-write operation per pair, instead of two. Trivial stuff, there isn't much to it.

But this simple change is enough to produce measurable gains, reported in the following tables. **Version16** there should be compared to *Version 14d* – it's the same code, only the pair reporting function has changed.

New office PC – Intel i7-6850K	Timings (K-Cycles)	Overall X factor
Version2 - base	66245	1.0
…	…	…
Version14d – integer cmp 2	*5452*	*~12.15*
Version15a – SSE2 intrinsics	5676	~11.67
Version15b – SSE2 assembly	3924	~16.88
Version15c – AVX assembly	2413	~27.45
Version16 – revisited pair reporting	**4891**	**~13.54**

Home laptop – Intel i5-3210M	Timings (K-Cycles)	Overall X factor
Version2 - base	62324	1.0
…	…	…
Version14d – integer cmp 2	*5011*	*~12.43*
Version15a – SSE2 intrinsics	5641	~11.04
Version15b – SSE2 assembly	4074	~15.29
Version15c – AVX assembly	2587	~24.09
Version16 – revisited pair reporting	**4743**	**~13.14**

Home desktop PC	Timings (K-Cycles)	Overall X factor
Version2 - base	98822	1.0
…	…	…
Version14d – integer cmp 2	*7386*	*~13.37 (*)*
Version15a – SSE2 intrinsics	16981	~5.81
Version15b – SSE2 assembly	6657	~14.84
Version15c – AVX assembly	Crash (AVX not supported)	0
Version16 – revisited pair reporting	**7231**	**~13.66**

(*) There was an error for this number in Part 15. It said "13.79" instead of "13.37" like in previous reports.

The gains are all over the place: 561 K-Cycles on one machine, 268 K-Cycles on another and a disappointing 155 K-Cycles on my home desktop PC. That last one rings a bell: I have a vague memory of removing the entire pair report mechanism at some point on this PC, to check how much the whole thing was costing me. The gains were so minimal I didn't bother investigating further.

For some reason the new machines give better gains. Due to lack of time (and lack of motivation: this part of the code is not very interesting to me), I did not investigate why. It's faster for "obvious" theoretical reasons (we do less work), we see gains in practice (at least we don't get slower results), that's good enough for now.

Similarly there would be more to do to fully replicate Ryg's changes from the AVX version. By nature the AVX version reports multiple pairs at the same time, but in a way the same can be said about our unrolled loops, and we could try to use the same strategies there. For example the resize check could be done only once before a loop unrolled N times starts, making sure that there is enough space in the array to write N pairs there (without extra checks). But I did not bother investigating further at this point: I am happy with the gains we got from the trivial change, and if the AVX version still has a small additional advantage from its improved pair reporting code, so be it.

What we learnt:

The "C++" code I wrote 15 years ago was not great. That pair reporting part was a bit lame. But then again 15 years ago I had never heard of load-hit-store penalties.

Show your code. Put it online. People will pick it up and point out the issues you forgot or didn't know you had in there.

We closed the gap a bit between our best "C++" SSE2 version and the fastest available SSE2 assembly version.

This was probably the last "easy" optimization available before tackling something much bigger.