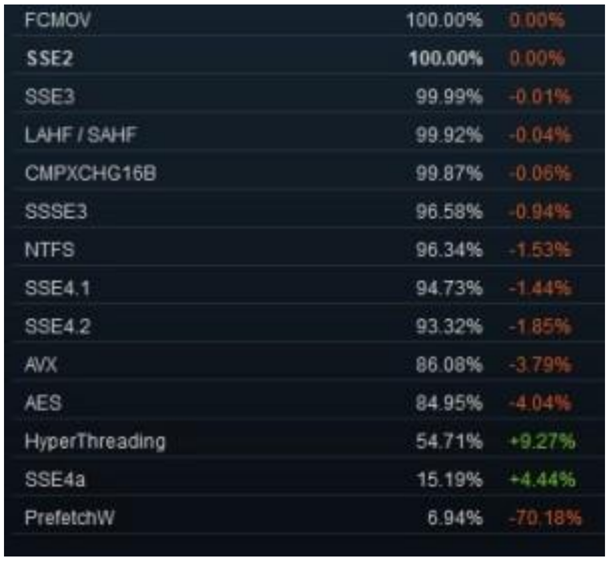
“Box pruning revisited” – an optimization project by Pierre Terdiman – 2017

Part 15 – AVX

In version 14b we looked at Ryg's initial experiments with this project. If you followed his progress on GitHub, you probably already know that he went ahead and did a lot more than what we covered so far. In particular, his latest version uses AVX instructions. I couldn’t try this myself before, since my PC did not support them. But things are different now (see previous post) so it's time to look at Ryg's most recent efforts (which are already one year old at this point).

According to Steam's hardware survey from April 2018, AVX is widely available (86%) but still not as ubiquitous as SSE2 (100%!):



FCMOV	100.00%	0.00%
SSE2	100.00%	0.00%
SSE3	99.99%	-0.01%
LAHF / SAHF	99.92%	-0.04%
CMPXCHG16B	99.87%	-0.06%
SSSE3	96.58%	-0.94%
NTFS	96.34%	-1.53%
SSE4.1	94.73%	-1.44%
SSE4.2	93.32%	-1.85%
AVX	86.08%	-3.79%
AES	84.95%	-4.04%
HyperThreading	54.71%	+9.27%
SSE4a	15.19%	+4.44%
PrefetchW	6.94%	-70.18%

That's why I never really looked at it seriously so far. At the end of the day, and in the back of my mind, I would like to use these optimizations on PC but also other platforms like consoles. Hopefully we will go back to this later in the series - I did run all these tests on consoles as well, and the results are not always the same as for the PC. In PhysX we are already suffering from the power difference between a modern PC and current-gen consoles: the consoles are sometimes struggling to handle something that runs just fine on PC. Using AVX would only tip the scales even more in favor of PCs. That being said, we had the same discussion about SSE2 back in the days, and now SSE2 is everywhere... including on consoles. So it is reasonable to expect AVX to be available in all next-gen consoles as well, and the time spent learning it now is hopefully just a good investment for the future. (I would totally take that time and play with AVX just for fun if I wouldn't have a child. But these days my free time is severely limited, as you noticed with the one year gap between 14d and 14e, so I have to choose my targets wisely).

Alright.

Like last time, Fabian was kind enough to include detailed notes about what he did. So, I will just copy-paste here what you probably already read a year ago anyway. In his own words:

Here's what I did to the code to arrive at the current version:

I already wrote a note on the earlier changes that were just cleaning up the ASM code and unrolling it a bit. That text file is a gist and available here:

https://gist.github.com/rygorous/fdd41f45b24472649aaeb5b55bbe6e26

...and then someone on Twitter asked "what if you used AVX instead of SSE2?". My initial response boiled down to "that's not gonna work with how the loop is currently set up". The problem is that the original idea of having one box=4 floats, doing all 4 compares with a single SSE compare, doing a movemask, and then checking whether we get the number that means "box intersects" (12 in the original case) fundamentally doesn't translate well to 8-wide: now instead of one box per compare, we're testing two (let's call them box1[0] and box1[1]), and instead of two possible outcomes, we now have four, where ? stands for any hex digit but 'c':

1. *Both box1[0] and box1[1] intersect box0. (=movemask gives 0xcc)*
2. *box1[0] intersects box0, box1[1] doesn't. (=movemask gives 0x?c)*
3. *box1[1] intersects box0, box1[0] doesn't. (=movemask gives 0xc?)*
4. *neither intersects box0. (=movemask gives 0x??)*

Instead of the previous solution where we had exactly one value to compare against, now we need to do a more expensive test like "(mask & 0xf) == 12 || (mask & 0xf0) == 0xc0". Not the end of the world, but it means something like

```
mov tmp, eax
and tmp, 15
cmp tmp, 12
je  FoundOne
and eax, 15
cmp eax, 12
je  FoundOne
```

which is one more temp register required and several extra uops, and as we saw in the gist, this loop was pretty tight before. Whenever something like this happens, it's a sign that you're working against the grain of the hardware and you should maybe take a step back and consider something different.

*That "something different" in this case was converting the loop to use a SoA layout (structure of arrays, google it, enough has been written about it elsewhere) and doing some back of the envelope math (also in this repository, "notes_avx.txt") to figure out how much work that would be per loop iteration. End result: 12 fused uops per iter to process *8* boxes (instead of 20 fused uops/iter to process 4 using SSE2!), still with a fairly decent load balance across the ports. This could be a win, and not just for AVX, but with SSE2 as well.*

Addressing

The first problem was that this code is 32-bit x86, which is register-starved, and going to SoA means we (in theory) need to keep around five pointers in the main loop, instead of just one.

*Possible (barely) but very inconvenient, and there's no way you want to be incrementing all of them. Luckily, we don't have to: the first observation is that all the arrays we advance through have the same element stride (four bytes), so we don't actually need to keep incrementing 5 pointers, because the distances between the pointers always stay the same. We can just compute those distances, increment *one* of the pointers, and use [reg1+reg2] addressing modes to compute the rest on the fly.*

*The second trick is to use x86's scaling indexing addressing modes: we can not just use address expressions [reg1+reg2], but also [reg1+reg2*2], [reg1+reg2*4] and [reg1+reg2*8] (left-shifts by 0 through 3). The *8 version is not very useful to use unless we want to add a lot of padding since we're just dealing with 6 arrays, but that narrows us down to four choices:*

1. *[base_ptr]*
2. *[base_ptr + dist]*
3. *[base_ptr + dist*2]*
4. *[base_ptr + dist*4]*

and we need to address five arrays in the main loop. So spending only 2 registers isn't really practical unless we want to allocate a bunch of extra memory. I opted against it, and choose 2 "dist" registers, one being the negation of the other. That means we can use the following:

1. *[base_ptr]*
2. *[base_ptr + dist_pos]*
3. *[base_ptr + dist_pos*2]*
4. *[base_ptr - dist_pos] = [base_ptr + dist_neg]*
5. *[base_ptr - dist_pos*2] = [base_ptr + dist_neg*2]*

ta-daa, five pointers in three registers with only one increment per loop iteration. The layout I chose arranges the arrays as follows:

1. *BoxMaxX[size]*
2. *BoxMinX[size]*
3. *BoxMaxY[size]*
4. *BoxMinY[size]*
5. *BoxMaxZ[size]*
6. *BoxMinZ[size]*

which has the 5 arrays we keep hitting in the main loop all contiguous, and then makes base_ptr point at the fourth one (BoxMinY).

You can see this all in the C++ code already, although it really doesn't generate great code there. The pointer-casting to do bytewise additions and subtractions is all wrapped into "PtrAddBytes" to de-noise the C++ code. (Otherwise, you wouldn't able to see anything for the constant type casts.)

Reporting intersections (first version)

This version, unlike Pierre's original approach, "natively" processes multiple boxes at the same time, and only does one compare for the bunch of them.

In fact, the basic version of this approach is pretty canonical and would be easily written in something like ISPC (ispc.github.io). Since the goal of this particular version was to be as fast as possible, I didn't do that here though, since I wanted the option to do weird abstraction-breaking stuff when I wanted to. :)

Anyway, the problem is that now, we can find multiple pairs at once, and we need to handle this correctly.

Commit id 9e171cf6 has the basic approach: our new ReportIntersections gets a bit mask of reported intersections, and adds the pairs once by one. This loop uses an x86 bit scan instruction ("bsf") to find the location of the first set bit in the mask, remaps its ID, then adds it to the result array, and finally uses "mask &= mask - 1;" which is a standard bit trick to clear the lowest set bit in a value.

This was good enough at the start, although I later switched to something else.

The basic loop (SSE version)

I first started with the SSE variant (since the estimate said that should be faster as well) before trying to get AVX to run. Commit id 1fd8579c has the initial implementation. The main loop is pretty much like described in notes_avx.txt (edi is our base_ptr, edx=dist_neg, ecx=dist_pos).

In addition to this main loop, we also need code to process the tail of the array, when some of the boxes have a mMinX > MaxLimit. This logic is fairly easy: identify such boxes (it's just another compare, integer this time since we convertred the MinX array to ints) and exclude them from the test (that's the extra "andnps" - it excludes the boxes with mMinX > MaxLimit). This one is sligthly annoying because SSE2 has no unsigned integer compares, only signed, and my initial "MungeFloat" function produced unsigned integers. I fixed it up to produce signed integers that are currently ordered in an earlier commit (id 95eaaaac).

This version also needs to convert boxes to SoA layout in the first place, which in this version is just done in straight scalar C++ code.

Note that in this version, we're doing unaligned loads from the box arrays. That's because our loop counters point at an arbitrary box ID and we're going over boxes one by one. This is not ideal but not a showstopper in the SSE version; however, it posed major problems for...

The AVX version

As I wrote in "notes_avx.txt" before I started, "if the cache can keep up (doubt it!)". Turns out that on the (Sandy Bridge) i7-2600K I'm writing this on, writing AVX code is a great way to run into L1 cache bandwidth limits.

The basic problem is that Sandy Bridge has full 256-bit AVX execution, but "only" 128-bit wide load/store units (two loads and one store per cycle). A aligned 256-bit access keeps the respective unit busy for 2 cycles, unaligned 256b accesses are three (best case).

In short, if you're using 256-bit vectors on SNB, it's quite easy to swamp the L1 cache with requests, and that's what we end up doing.

The initial AVX version worked, but ended up being slightly slower than the (new) SSE2 version. Not very useful.

To make it competitive, it needed to switch to aligned memory operations. Luckily, in this case, it turns out to be fairly easy: we just make sure to allocate the initial array 32-byte aligned (and make sure the distance between arrays is a multiple of 32 bytes as well, to make sure that if one of the pointers is aligned, they all are), and then make sure to get us to a 32-byte aligned address before we enter the main loop.

*So that's what the first real AVX version (commit 19146649) did. I found it a bit simpler to round the current box pointer *down* to a multiple of 32, not up. This makes the first few lanes garbage if we weren't already 32-byte aligned; we can deal with this by masking them out, the same way we handled the mMinX > MaxLimit lanes in the tail code.*

And with the alignment taken care of, the AVX version was now at 3700 Kcycles for the test on my home machine, compared to about 6200 Kcycles for the SSE2 version! Success.

Cleaning up the setup

At this point, the setup code is starting to be a noticeable part of the overall time, and in particular the code to transform the box array from AoS to SoA was kind of ratty. However, transforming from AoS to SoA is a bog-standard problem and boils down to using 4x4 matrix transposition in this case. So I wrote the code to do it using SIMD instructions instead of scalar too, for about another 200 Kcycles savings (in both the AVX and SSE2 vers).

Reporting intersections (second version)

After that, I decided to take a look with VTune and discovered that the AVX version was spending about 10% of its time in ReportIntersections, and accruing pretty significant branch mis-prediction penalties along the way.

So, time to make it less branchy.

*As a first step, added some code so that instead of writing to "Container& pairs" directly, I get to amortize the work. In particular, I want to only do the "is there enough space left" check *once* per group of (up to) 8 pairs, and grow the container if necessary to make sure that we can insert those 16 pairs without further checks. That's what "PairOutputBuffer" is for. It basically grabs the storage from the given Container while it's in scope, maintains our (somewhat looser) invariants, and is written for client code that just wants to poke around in the pointers directly, so there's no data hiding here. That was finalized in commit 389bf503, and decreases the cost of ReportIntersections slightly.*

Next, we switch to outputting the intersections all at once, using SIMD code. This boils down to only storing the vector lanes that have their corresponding mask bit set. This is a completely standard technique. Nicely enough, Andreas Frediksson has written it up so I don't have to:

https://deplinenoise.files.wordpress.com/2015/03/gdc2015_afredriksson_simd.pdf

(The filter/"left packing" stuff). AVX only exists on machines that also have PSHUFB and POPCNT so we can use both.

This indeed reduced our CPU time by another ~250 Kcycles, putting the AVX version at about 3250 Kcycles! And that's where it currently still is on this machine. (That version was finalized in commit id f0ca3dc1).

Porting back improvements to SSE2/Intrinsics code

Finally, I decided to port back some of these changes to the SSE2 code and the C++ intrinsics version. In particular, port the alignment trick from AVX to SSE2 for a very significant win in commit d92dd5f9, and use the SSE2 version of the left-packing trick in commit 69baa1f1 (I could've used SSSE3 there, but I didn't want to gratuitously increase the required SSE version).

And then I later ported the left-packing trick for output to the C++ intrinsics version as well. (The alignment trick does not help in the intrinsics ver, since the compiler is not as good about managing registers and starts tripping all over its feet when you do it.)

Thank you Ryg.

That is quite a lot of stuff to digest. I guess we can first look at the results on my machines. There are 3 different versions in Fabian's last submits:

- an SSE2 version using intrinsics
- an SSE2 version using assembly
- an AVX version using assembly

And I now have 3 different PCs available: my old home desktop PC (one of the two used in the initial posts for this serie, the other one died), a new office desktop PC, and a new home laptop PC. So that's 9 results to report. Here they are (new entries in **bold**):

New office PC – Intel i7-6850K	Timings (K-Cycles)	Overall X factor
Version2 - base	66245	1.0
Version3 – don't trust the compiler	65644	-
Version4 - sentinels	58706	-
Version5 – hardcoding axes	55560	-
Version6a – data-oriented design	46832	-
Version6b – less cache misses	39681	-
Version7 – integer cmp	36687	-
Version8 – branchless overlap test	23701	-
Version9a - SIMD	18758	-
Version9b – better SIMD	10065	-
Version9c – data alignment	10957	-
Version10 – integer SIMD	12352	-
Version11 – the last branch	11403	-
Version12 - assembly	7197	-
Version13 – asm converted back to C++	8434	-
Version14a – loop unrolling	7511	-
Version14b – Ryg unrolled assembly 1	5094	~13.00
Version14c – better unrolling	5375	-
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**

Home laptop – Intel i5-3210M	Timings (K-Cycles)	Overall X factor
Version2 - base	62324	1.0
Version3 – don’t trust the compiler	59250	-
Version4 - sentinels	54368	-
Version5 – hardcoding axes	52196	-
Version6a – data-oriented design	43848	-
Version6b – less cache misses	37755	-
Version7 – integer cmp	36746	-
Version8 – branchless overlap test	28206	-
Version9a - SIMD	22693	-
Version9b – better SIMD	11351	-
Version9c – data alignment	11221	-
Version10 – integer SIMD	11110	-
Version11 – the last branch	10871	-
Version12 - assembly	9268	-
Version13 – asm converted back to C++	9248	-
Version14a – loop unrolling	9009	-
Version14b – Ryg unrolled assembly 1	5040	~12.36
Version14c – better unrolling	5301	-
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**

Home desktop PC	Timings (K-Cycles)	Overall X factor
Version2 - base	98822	1.0
Version3 – don’t trust the compiler	93138	-
Version4 - sentinels	81834	-
Version5 – hardcoding axes	78140	-
Version6a – data-oriented design	60579	-
Version6b – less cache misses	41605	-
Version7 – integer cmp	40906	-
Version8 – branchless overlap test	31383	-
Version9a - SIMD	34486	-
Version9b – better SIMD	32565	-
Version9c – data alignment	14802	-
Version10 – integer SIMD	16667	-
Version11 – the last branch	14512	-
Version12 - assembly	11731	-
Version13 – asm converted back to C++	12236	-
Version14a – loop unrolling	9012	-
Version14b – Ryg unrolled assembly 1	7600	-
Version14c – better unrolling	7558	-
Version14d – integer cmp 2	7386	~13.79
Version15a – SSE2 intrinsics	**16981**	**~5.81**
Version15b – SSE2 assembly	**6657**	**~14.84**
Version15c – AVX assembly	**Crash (AVX not supported)**	**0**

So first, we see that the performance of the SSE2 intrinsics version is quite different depending on where you run it. It is fine on my more recent machines, but it is quite bad on my older home desktop PC, where it is roughly similar to version 9c in terms of speed. That is a clear regression compared to our latest unrolled versions. I did not investigate what the problem could be, because even on modern PCs the performance is ultimately not as good as our best "version 14". On the other hand, this version (let's call it 15a) is not as ugly-looking as 14c or 14d. So provided we could fix its performance issue on the home desktop PC, it could be an interesting alternative which would remain somewhat portable.

Then, we have version 15b. This one is pretty good and offers a clear speedup over our previous SSE2 versions. This is interesting because it shows that without going all the way to AVX (i.e. without losing compatibility with some machines), the AVX "philosophy" if you want (re-organizing the code to be AVX-friendly) still has some potential performance gains. Ideally, we would be able to get these benefits in the C++ code as well. Admittedly this may not be easy since this is essentially what 15a failed to do, but we might be able to try again and come up with a better 15a implementation. Somehow.

Finally, version 15c is the actual AVX version. Unsurprisingly if I bypass the AVX check and run it on my home desktop PC, it crashes - since that PC does not support AVX. On the other hand, on the machines that do support it, performance is awesome: we get pretty much the advertised 2X speedup over regular SIMD that AVX was supposed to deliver. And thus, with this, we are now about **24X** to **27X** faster than the original code. Think about that next time somebody claims that low-level optimizations are not worth it anymore, and that people should instead focus on multi-threading the code: you would need a 24-core processor and perfect scaling to reach an equivalent speedup with multi-threading...

So, where do we go from here?

I had plans for where to move this project next, but now a whole bunch of new questions arose.

Some of these optimizations like the one done for reporting intersections in a less branchy way seem orthogonal to AVX, and could potentially benefit our previous versions as well. The way these AVX versions have been delivered, combining multiple new optimizations into the same new build, it is slightly unclear how much the performance changed with each step (although Ryg's notes do give us a clue). I usually prefer doing one optimization at a time (each in separate builds) to make things clearer. In any case, we could try to port the improved code for reporting intersections to version 14 and see what happens.

Fabian initially claimed that the design did not translate well to 8-wide, and thus we had to switch to SoA. I didn't give it much thoughts but I am not sure about this yet. I think the dismissed non-existing version where we would test 2 boxes at a time with AVX could still give us some gains. The movemask test becomes slightly more expensive, yes, but we still drop half of the loading+compare+movemask instructions. That must count for something? Maybe I am being naive here.

Another thing to try would be to dive into the disassembly of version 15a, figure out why it performs badly on the home desktop PC, and fix it / improve it. That could be a worthwhile goal because I really didn't want to move further into assembly land. Quite the opposite: one of the planned next posts was about checking the effects of these optimizations on ARM and different architectures. Assembly versions are a show-stopper there - we cannot even have inline assembly on Win64 these days. So at the very least versions 15b and 15c give us new targets and show us what is possible in terms of performance. But I will have difficulties keeping them around for long.

And this brings us to the obvious question about 15c: could we try it using *AVX intrinsics* instead of assembly? That could be a way to keep some portability (at least between Win32 and Win64) while still giving us some of the AVX performance gains.

Another thing that comes to mind is that we saw in part 3 that the sorting was costing us at best 140 K-Cycles (and in reality, much more). This was negligible at the time, but Ryg's latest optimizations were about saving ~200 K-Cycles, so this part is becoming relevant again. One strategy here, if we don't want to deal with this just yet, could be to reset the test and use more boxes. We used 10000 boxes so far, but we could just add a 0 there and continue our journey.

Beyond that, I had further optimizations planned for the whole project, which are completely orthogonal to AVX. So I could just continue with that and ignore the AVX versions for now. A new goal could be for me to reach the same performance as the AVX assembly version, but using another way, with regular SSE intrinsics.

Here is a potential TODO list for further reports:

1. Try the optimized intersections report in version 14.
2. Try an AVX version that tests 2 boxes at a time without SoA.
3. Analyze the 15a disassembly and try to fix it (make it fast on my home desktop PC).
4. Try a version using AVX intrinsics.
5. Revisit the sorting code (and general setup code) in version 14/15.
6. Go ahead with the initial plan and further non-AVX-related optimizations (that's at least 3 more blog posts there).
7. Once it's done, merge AVX optimizations to these new versions for further speedup.
8. When applicable, check the performance of all these versions on other platforms / architecture. That's when having a separate build per optimization pays off: do we have some optimizations that hurt instead of help on some platforms?
9. Investigate how the performance varies and which version is the fastest when the number of objects changes.
10. Explain what it takes to productize this and make it useful in a real physics engine (in particular: how you deal with sleeping objects and how you report new and deleted pairs instead of all of them).
11. Field test.

I don't know yet what I will try next, or when, but it seems that there is indeed, more than ever, still a lot to do.

What we learnt:

AVX works. It does not happen every day but it can make your code 2X faster than regular SSE.

You might need assembly for that to happen though. Like it or not, assembly wins again today.

Do not ignore low level optimizations. In our case there was a 24X performance gain on the table (so far), without changing the algorithm, without multi-threading. Typical multi-threading will give you much less than that.

I apologize for the lack of new material, it is really just the same as what Fabian published a year ago. I did a minor modification to Fabian’s code, to be able to run the SSE2 versions on AVX-enabled machines - so now you have to select the desired version with a define.