# Introduction to AVX2 optimizations in x264

x264 is a good example of a program with many integer SIMD functions that operate on a wide variety of data types and perform a variety of different operations. Few of these are simple enough that we can change register sizes from 128-bit to 256-bit trivially; many are frequency transforms, FIR filters, and other functions with a lot of data mixing and dependency. Just like moving from 64-bit to 128-bit registers rarely gave an exact doubling of performance, moving from 128-bit to 256-bit rarely will here, too.

The shift from SSE2-derived integer SIMD to AVX2 is rather analogous to the change from MMX to SSE2 back during the era of the Pentium 4, Athlon 64, and Core 2 CPUs. However, there's a few core differences.

When SSE2 was introduced, the first implementations were often quite slow. The most extreme examples, the Athlon 64 and Core 1, only had 64-bit execution units, so an SSE2 function was often equal to or slower than the MMX equivalent in speed. It wasn't until the Core 2 Penryn (45nm) that this finally ended, when Intel blessed Penryn with a lightning-fast 128-bit shuffle unit[1], making MMX operations and their SSE2 equivalents finally equal in speed for all classes of instructions. With AVX2, this isn't a problem, at least with Haswell – Intel has long marketed Haswell's 256-bit load and arithmetic units, so 256-bit AVX2 operations can be largely expected to be just as fast as 128-bit SSE.[2]

Secondly, AVX2 operations are often extended from 128-bit in a somewhat unusual way. Instead of being logically expanded in the expected fashion, they operate on 128-bit *lanes* of data, with a few special inter-lane instructions like *vpermd*. This lets Intel implement larger SIMD by copy-pasting smaller SIMD units, but it also adds yet another challenge in extending 128-bit code to 256-bit code.[3]

Thirdly, x264 has many functions that are smaller in width than 256-bit AVX registers, making it more difficult (or sometimes, impossible) to effectively use AVX2. This issue existed when moving from 64-bit MMX to 128-bit SSE2, of course, but it's more dramatic now since there's quite a lot of functions that aren't wide enough to make easy use of AVX2.

To facilitate the move to AVX2, we've extended the x264asm abstraction layer (*x86inc.asm*) to to include AVX operations and registers. This let us port a number of rather complex functions to AVX2 with far fewer changes than we would have expected. As always, x264asm is available under a BSD-like license for anyone to use – if you're working on lots of x86 assembly code, I strongly recommend trying it!

I'd love to write an AVX2 optimization guide; there's plenty of tricks and subtleties that we've discovered along the way. Many of the functions I wrote before I had access to a Haswell turned out to be grossly suboptimal. Unfortunately, I'm pretty sure I'm still bound by the NDA here; the most I think I'm currently allowed to do is post performance numbers for individual functions, which you'll see on the next page.

---

1 Two of them, actually!

2 Like always, there might be some minor exceptions: for example, on Sandy Bridge, most AVX floating point operations had 256-bit execution units, but rcpps (reciprocal approximation) did not, likely to save hardware. This is also no guarantee for future (especially non-Intel) CPUs; for example, Bulldozer supported AVX, but only uses 128-bit execution units, so it's probably no faster than SSE for floating point math.

3 Each 128-bit lane is operated on separately, so for example, pshufb shuffles 2 groups of 16 bytes separately instead of 1 group of 32 bytes. The latter would be more convenient, but require more hardware and greater circuit depth, thus increasing latency or lowering clock speed.

## *Results from various functions*

Some quick disclaimers here before we dive into individual functions. Many of these functions have not gotten the same level of optimization treatment that a lot of x264's x86 assembly has[4], so there probably are still future performance gains to be had. We've learned a lot already from working on the Haswell; many functions have been significantly revised multiple times as we discovered faster ways to perform common operations. We can only expect there's still more to learn in the future!

There's also some functions that still could use AVX2 assembly and don't have any yet, so our work isn't complete, though I do feel we're relatively close.[5] Also, this is not at all a complete list; I omitted some less important or less interesting functions.

Furthermore, keep in mind that x264 is not 100% assembly code. About half of x264's running time is spent in either optimized C code or mostly-non-SIMD assembly code, where AVX2 will not provide any gains. Even if all the SIMD assembly functions doubled in speed, x264 would only get 33% faster because of this bottleneck.[6]

| Function | Description | Cycles (C code) | Cycles (pre-AVX2) | Cycles (AVX2) | Gain with AVX2[7] | Gain vs. C | Notes |
|---|---|---|---|---|---|---|---|
| **add8x8_idct** | 4x 4×4 inverse transforms on 8×8 coefficient block | 326.8 | 41.8 | 31.4 | **33.1%** | **10.4x** | x264 does almost all inverse transforms in 8×8 groups, so I did not optimize the 16×16 variant, even though it would have gained more from AVX2. |
| **avg_16x16** | 6-bit fixed point weighted average on 16×16 pixel block | 673.7 | 63.5 | 49.2 | **29.1%** | **13.7x** | Widths smaller than 16 were too small to gain anything. |
| **denoise_dct** | adaptive deadzone frequency-domain denoiser on 4×4 or 8×8 coefficient block | 152.9 | 18.1 | 10.2 | **77.5%** | **15.0x** | |
| **dequant_8×8** | coefficient dequantization on 8×8 coefficient block | 182.9 | 13.3 | 7.8 | **70.5%** | **23.4x** | 4×4 gains similarly. |
| **hadamard_ac_16x16** | sum of absolute AC coefficients with 8×8 and 4×4 Hadamard transforms on 16×16 pixel block | 1190.8 | 188.5 | 102.5 | **83.9%** | **11.6x** | 16x8 gains similarly. |
| **hpel_filter** | 6-tap separable filter to calculate hpel pixels (resize by 2x in each direction) | 8883.9 | 1239.8 | 885.5 | **40.0%** | **10.0x** | |

4 I've only had the Haswell for about a month!

5 Contributions welcome, seriously! Poke us on #x264dev on Freenode IRC if you have any ideas or want to help. No video compression knowledge necessary, just love of SIMD hacking – and really, you don't have to be good at it; any skill level is okay. We love playing with this kind of thing.

6 A total running time of 0.5 seconds C code + 0.5 seconds SIMD code would become 0.5 + 0.25, or 0.75 seconds for every 1 second before. 1 / 0.75 is 1.33, or 33% faster. This is a case of Amdahl's Law.

7 Defined such that a gain of 100% means twice as fast.

| Function | Description | Cycles (C code) | Cycles (pre-AVX2) | Cycles (AVX2) | Gain with AVX2 | Gain vs. C | Notes |
|---|---|---|---|---|---|---|---|
| **intra_predict_16x16_p** | 16×16 planar intra prediction mode | 434.1 | 53.9 | 44.9 | **20.0%** | **9.7x** | 8×8/8x16 gain similarly. |
| **lowres_init** | bilinear downscaling filter for lowres lookahead | 9613.8 | 643.9 | 444.8 | **44.8%** | **21.6x** | |
| **mbtree_propagate** | part of x264's mb-tree algorithm; mostly Newton's Method for fast division approximation | 2903.9 | 240.4 | 182.5 | **31.7%** | **15.9x** | x264's only floating point function |
| **mc_chroma_8x8** | 6-bit fixed point bilinear motion compensation on 2x 8×8 pixel blocks | 567.5 | 53.9 | 44.1 | **22.2%** | **12.9x** | Smaller widths have some, albeit smaller, gains. |
| **quant_8x8** | coefficient quantization on 8×8 coefficient block | 247.6 | 15.5 | 9.0 | **72.2%** | **27.5x** | 4×4 gains similarly. |
| **sa8d_satd_16x16** | combined satd/sa8d calculation on 16×16 pixel block | 1546.7 | 189.9 | 113.6 | **67.2%** | **13.6x**[8] | |
| **sad_x4_16x16** | SAD comparisons between 4x 16×16 pixel blocks and one reference block | 1855.5 | 61.6 | 53.8 | **14.5%** | **34.5x** | This was a function that was expected to have no real gain with AVX2 for width reasons, but nevertheless we were able to shave off a good few cycles. sad_x3 gains similarly. |
| **satd_16x16** | 16x 4×4 SATDs on 16×16 pixel block | 725.9 | 115.9 | 63.1 | **83.7%** | **11.5x**[8] | x264's most heavily optimized function, and very important. Smaller widths gain significantly less, but still some. |
| **ssd_16x16** | sum of squared differences between two 16×16 pixel blocks | 306.5 | 51.8 | 36.9 | **40.4%** | **8.3x** | Widths smaller than 16 were too small to gain anything. |
| **sub16x16_dct** | 16x 4×4 forward transform on 16×16 pixel blocks | 787.2 | 148.9 | 115.5 | **28.9%** | **6.8x** | Smaller widths gain less, but aren't used very often. |
| **sub16x16_dct8** | 4x 8×8 forward transform on 16×16 pixel blocks | 1523.8 | 239.5 | 145.8 | **64.3%** | **10.5x** | Smaller widths gain less, but aren't used very often. |
| **var_16x16** | variance of 16×16 pixel block | 339.3 | 48.6 | 31.1 | **56.3%** | **10.9x** | Widths smaller than 16 were too small to gain anything. |
| **weight_w16** | 6-bit fixed point weight of width-16 pixel block | 697.0 | 53.6 | 34.3 | **56.3%** | **19.8x** | Widths smaller than 16 gained much less, but still some. |

8 The C code for SATD is already munged into this wonderfully evil pseudo-SIMD code that's about 60% faster than the naive C.