Research is Hard

2012 saw the release of Beastie Boys' final album called *Hot Sauce Committee Part Two*. Part 1 was planned for a 2009 release, but Adam "MCA" Yauch had been diagnosed with cancer that year and died right after the release of part 2 in 2012. His death was a great loss for both music and film. It has to be mentioned that part 2 had almost the same track listing as the announced part 1, so it evened out. As such, the year had good and extremely bad news.

Also in 2012, something completely expected happened: A team of researchers jumped on the "C programming is hard" bandwagon. In a paper called *Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines*[10] published by a lot of names from *MIT*, *Stanford*, and *Adobe*, they claim

> *Using existing programming tools, writing high-performance image processing code requires sacrificing readability, portability, and modularity.*

That's a very courageous claim. And I mean "courageous" as it's used in the classic TV series *Yes, Minister*.

Of course, they tried pitching their own toy language to solve those supposed problems. I would not let that stand and made a counter claim: **Fast C code is always pretty and highly organized.** Come to think of it, that's partly what this book is about. Weird.

As a start, they supplied some examples to prove their claim: Here's what they called "Clean C" (reference A):

```
void blur(const Image &in, Image &blurred) {
 Image tmp(in.width(), in.height());

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

 for (int y = 0; y < in.height(); y++)
  for (int x = 0; x < in.width(); x++)
   blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

Here's what they called "Fast C++" (reference B):

```
void fast_blur(const Image &in, Image &blurred) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i tmp[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
    __m128i *tmpPtr = tmp;
    for (int y = -1; y < 32+1; y++) {
     const uint16_t *inPtr = &(in(xTile, yTile+y));
     for (int x = 0; x < 256; x += 8) {
      a = _mm_loadu_si128((__m128i*)(inPtr-1));
      b = _mm_loadu_si128((__m128i*)(inPtr+1));
      c = _mm_load_si128((__m128i*)(inPtr));
      sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
      avg = _mm_mulhi_epi16(sum, one_third);
      _mm_store_si128(tmpPtr++, avg);
      inPtr += 8;
    }}
    tmpPtr = tmp;
    for (int y = 0; y < 32; y++) {
     __m128i *outPtr = (__m128i *)(&(blurred(xTile, yTile+y)));
     for (int x = 0; x < 256; x += 8) {
      a = _mm_load_si128(tmpPtr+(2*256)/8);
      b = _mm_load_si128(tmpPtr+256/8);
      c = _mm_load_si128(tmpPtr++);
      sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
      avg = _mm_mulhi_epi16(sum, one_third);
      _mm_store_si128(outPtr++, avg);
}}}}}
```

[10] http://people.csail.mit.edu/jrk/halide12/halide12.pdf

The code actually looked like that. I would call that "amateur C" and "amateur C++", respectively.

How is it possible to write code like that? Just piling up the curly brackets at the end, what's that about? If I'd been a conspiracy theorist, I'd say this crap was produced by the ones selling CPU time by the second. I'm thinking of the old saying from the *Real Programmers Code of Conduct* back in the *PDP-11* days: "If it was hard to write, it should be hard to read." The code of conduct is an old joke, but it was very appropriate here.

Let's get down to the details. The code is supposed to be a 3x3 box filter. As pointed out afterwards by one of the few who reads my stuff, Carl Zimmerman:

> *The box filter code at https://www.ignorantus.com/box_sse2/ overflows. Is there a reason you or Halide didn't use the saturating add instruction? i.e: _mm_add_epi16 becomes _mm_adds_epi16*

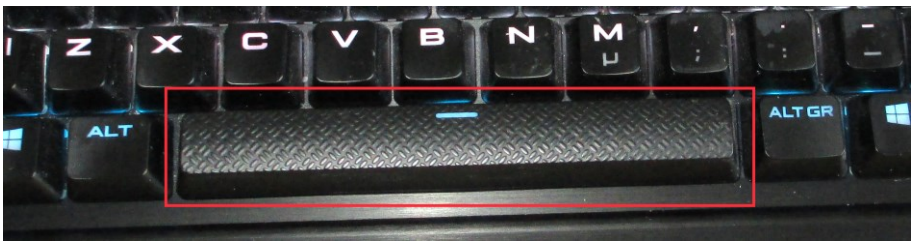
I guess he's correct about that. Test it yourself! For me, it was just another data processing job, I didn't care about the math. Looks like the authors of the article didn't care either, which is weird. Now, we can spend all day figuring out how box filters work, or we can trust Zimmerman. I'll go with Zimmerman on this one.

Time to dust off the old *SSE2* manual. Here's a quick review the intrinsics used. Intel didn't clutter it up totally like ARM did, only slightly, so most of them are simple and straightforward: *__m128i* is the basic datatype. *_mm_load_si128/store_si128* loads and stores stuff. *_mm_add_epi16* adds 16-bit values and wraps, whereas *_mm_adds_epi16* saturates. *_mm_mulhi16* multiplies 16-bit values, generating 32-bit results and packing the high 16 bits in each answer. Very convenient. And very simple.

Back to the article. The stream of courageous claims continued unabated (my underlining):

> *Using vectorization, multithreading, tiling, and fusion, we can make this algorithm more than 10× faster on a quad-core x86 CPU (b). <u>However, in doing so we've lost readability and portability.</u>*

Let's try to find out exactly what they mean, by using a radical approach. The symbol "space", the wide key at the bottom of the keyboard, can be used to improve readability, and blank lines can be used to group blocks of code. And LED-lights in the keyboard is entertaining. For about 20 seconds.



Let's reformat reference B:

```
void fast_blur_halide( const uint16_t *in, uint16_t *blurred, int width, int height )
{
    int x, y;
    int xTile, yTile;
    __m128i one_third = _mm_set1_epi16(21846);

    for( yTile = 0; yTile < height; yTile += 32 ) {

        __m128i a, b, c;
        __m128i sum, avg;
        __m128i tmp[(256/8)*(32+2)];

        for( xTile = 0; xTile < width; xTile += 256 ) {

            __m128i *tmpPtr = tmp;
```

```
            for( y = -1; y < 32+1; y++ ) {

                const uint16_t *inPtr = &in[(yTile+y)*width + xTile];

                for (x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128( (__m128i*)(inPtr-1) );
                    b = _mm_loadu_si128( (__m128i*)(inPtr+1) );
                    c = _mm_load_si128(  (__m128i*)(inPtr)   );
                    sum = _mm_add_epi16( _mm_add_epi16( a, b ), c );
                    avg = _mm_mulhi_epi16( sum, one_third );
                    _mm_store_si128( tmpPtr++, avg );
                    inPtr += 8;
                }

            }

            tmpPtr = tmp;

            for( y = 0; y < 32; y++ ) {

                __m128i *outPtr = (__m128i *)&blurred[(yTile+y)*width + xTile];

                for( x = 0; x < 256; x += 8 ) {
                    a = _mm_load_si128( tmpPtr+(2*256)/8 );
                    b = _mm_load_si128( tmpPtr+256/8     );
                    c = _mm_load_si128( tmpPtr++         );
                    sum = _mm_add_epi16( _mm_add_epi16( a, b ), c );
                    avg = _mm_mulhi_epi16( sum, one_third );
                    _mm_store_si128( outPtr++, avg );
                }
            }
        }
    }
}
```

Now it's readable, and it's really not that hard. A way forward could be renaming some variables to have logical names, but there's something just plainly wrong about it. The article was written in 2012. The temporary buffer construct shouldn't be necessary with modern Intel CPUs from that year. Looking a bit more closely at the text, I found the following paragraph:

> *Performance results are reported as the best of five runs on a 3GHz Core2 Quad x86 desktop, a 2.5GHz quad-core Core i7-2860QM x86 laptop, a Nokia N900 mobile phone with a 600MHz ARM OMAP3 CPU, a dual core ARM OMAP4 development board (equivalent to an iPad 2), and an NVIDIA Tesla C2070 GPU (equivalent to a midrange consumer GPU).*

So, there's an unspecified *Core2 Quad*, a semi-crap mobile version, some other chips I didn't care about then, and a hyper-expensive GPU. The sticker price was a cool $3,999 in 2011! It's not a joke. I rummaged around in the basement and found an old Intel *Core2 Quad Q9550* CPU, launched in 2008, where the code in reference B actually ran ok. So: The code was optimized for an Intel architecture that was at least 4 years old at the time. A blast from the past!

Since then, something completely expected had happened: Newer and better CPUs had been launched. An OK and reasonably priced CPU in 2012 was the Intel i7 "Sandy Bridge" 2600K that had significantly improved caches. I had one of them sitting on a shelf. Not using it since even better CPUs had come along, but available for testing.

One issue I missed back then was looking at how the loops were constructed. Consider the first inner loop in reference B. For chips that don't have the necessary I/O available, which I guess cover the *OMAPs* and the unspecified Core2, using a temporary buffer like they did would probably make sense. Or would it? I'm not a big fan of using two unaligned loads and one aligned when only one aligned and some simple reformatting is necessary. Could go either way since data is going to be in the L1 cache pretty quickly. And there's the question about unrolling. And how costly would the unaligned loads be on such an old CPU? Many questions, few answers. My Core2 Quad system is unfortunately dead and buried, so it's impossible to test that now.

Back then, I just tried doing it the right way, a.k.a. as simple as possible. Reading three lines and writing one is the simplest I could think of:

```
void fast_blur_horiz( const uint16_t *in, uint16_t *blurred, int width, int height )
{
    int x, y;
    __m128i one_third;
    __m128i *dst;

    one_third = _mm_set1_epi16(21846);
    dst = (__m128i *)blurred;

    for( y = 0; y < height; y++ ) {
        const uint16_t *row0, *rowp, *rown;

        row0 = &in[y*width];
        rowp = row0 - width;
        rown = row0 + width;

        for( x = 0; x < width; x += 8 ) {
            __m128i s0 ,s1 ,s2;
            __m128i r0, r1, r2;

            s0 = _mm_loadu_si128( (__m128i*)(row0-1) );
            s1 = _mm_loadu_si128( (__m128i*)(row0+1) );
            s2 = _mm_load_si128(  (__m128i*)(row0) );
            r0 = _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16( s0, s1 ), s2 ), one_third );

            s0 = _mm_loadu_si128( (__m128i*)(rowp-1) );
            s1 = _mm_loadu_si128( (__m128i*)(rowp+1) );
            s2 = _mm_load_si128(  (__m128i*)(rowp) );
            r1 = _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16( s0, s1 ), s2 ), one_third );

            s0 = _mm_loadu_si128( (__m128i*)(rown-1) );
            s1 = _mm_loadu_si128( (__m128i*)(rown+1) );
            s2 = _mm_load_si128(  (__m128i*)(rown) );
            r2 = _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16( s0, s1 ), s2 ), one_third );

            _mm_store_si128( dst++, _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16(r0,r1),r2 ),one_third ) );

            row0 += 8;
            rowp += 8;
            rown += 8;
        }

    }

}
```

Trivial. But seeing as the caching was significantly improved on the i7-2600K, reading one more, for a total of four, and writing two gave fantastic results:

```
void fast_blur_horiz2d( const uint16_t *in, uint16_t *blurred, int width, int height )
{
    int x, y;
    __m128i one_third = _mm_set1_epi16(21846);
    __m128i dst0 = (__m128i *)blurred;
    __m128i dst1 = (__m128i *)(blurred+width);

    for( y = 0; y < height; y += 2 ) {
        const uint16_t *row0, *row1, *row2, *row3;
        row1 = in + y*width;
        row0 = row1 - width;
        row2 = row1 + width;
        row3 = row2 + width;

        for( x = 0; x < width; x += 8 ) {
            __m128i s00 ,s01 ,s02;
            __m128i r00, r01, r02;

            s00 = _mm_loadu_si128( (__m128i *)(row0-1) );
            s01 = _mm_loadu_si128( (__m128i *)(row0+1) );
            s02 = _mm_load_si128(  (__m128i *)(row0) );
            r00 = _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16( s00, s01 ), s02 ), one_third );
```

```
            s00 = _mm_loadu_si128( (__m128i *)(row1-1) );
            s01 = _mm_loadu_si128( (__m128i *)(row1+1) );
            s02 = _mm_load_si128(  (__m128i *)(row1) );
            r01 = _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16( s00, s01 ), s02 ), one_third );


            s00 = _mm_loadu_si128( (__m128i *)(row2-1) );
            s01 = _mm_loadu_si128( (__m128i *)(row2+1) );
            s02 = _mm_load_si128(  (__m128i *)(row2) );
            r02 = _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16( s00, s01 ), s02 ), one_third );

            _mm_store_si128( dst0++,_mm_mulhi_epi16( _mm_add_epi16(_mm_add_epi16(r00,r01),r02), one_third));

            s00 = _mm_loadu_si128( (__m128i *)(row3-1) );
            s01 = _mm_loadu_si128( (__m128i *)(row3+1) );
            s02 = _mm_load_si128(  (__m128i *)(row3) );
            r00 = _mm_mulhi_epi16( _mm_add_epi16( _mm_add_epi16( s00, s01 ), s02 ), one_third );

            _mm_store_si128( dst1++,_mm_mulhi_epi16( _mm_add_epi16(_mm_add_epi16(r00,r01),r02), one_third));

            row0 += 8;
            row1 += 8;
            row2 += 8;
            row3 += 8;
        }

        dst0 += width>>3;
        dst1 += width>>3;

    }

}
```

And we could go on like that, but the *i7-2600K* peaked out at 4 + 2. Would be amusing to try other variants on newer CPUs. Notice how simple it is to use short and logical variable names and format it so pieces of code that belong together stick together!

It was the only example provided in the paper. There was a lot of talk there about how the new language worked, and some talk about more advanced filters. But it was just talk, not code. My interest had faded long ago.

I did some measurements on a set of CPUs back then to figure out what they used. Let's start with a blast from the past, probably the desktop CPU the researchers used. Iterations set to 10, since it was horrendously slow:

```
Image: 8192*8192, 10 iterations
CPU:   Q9550@2.8Ghz (launched Q1 2008)
RAM:   4x2GB@800Mhz
                 Time   Perc
Halide:        884382   0.0%
Corneliusen:  1250236 -41.4%
```

Aha! It's actually fast on that one, compared to the modern approach I outlined.

Let's try something from 2009, the year after: The Intel *i7-950*. Still pretty old at the time:

```
Image: 8192*8192, 100 iterations
CPU:   i7-950@3Ghz (launched Q2 2009)
RAM:   6x2GB@1066Mhz
                 Time   Perc
Halide:       6931559   0.0%
Corneliusen:  4421372  36.2%
```

Interesting. A CPU from 2009 runs the trivial SIMD version much faster.

Let's try something reasonably priced from the time period, the i7-2600K:

```
Image: 8192*8192, 100 iterations
CPU:   i7-2600K@3.4Ghz (launched Q1 2011)
RAM:   4x4GB@1600Mhz
                 Time   Perc
Halide:       4403894   0.0%
Corneliusen:  2608385  40.8%
```

Also interesting. Much faster.

This seems to be a recurring issue to this day. I recall reading several scientific papers that were riddled with exceptionally bad examples in C and intrinsics. It's almost like they'd never written C before, or programs. However, that can be excused if the basic idea is good. That wasn't the case here. It seemed like they had made a test bench of random crap from the recycle bin, which happened to give good results for the exceptionally bad example they were using. Which is odd, considering that the article is written by names from companies that have a good reputation.