Forskning er vanskelig

I 2012 ble *Beastie Boys'* siste album kalt *Hot Sauce Committee Part Two* gitt ut. Del 1 var planlagt for utgivelse i 2009, men Adam "MCA" Yauch hadde blitt diagnostisert med kreft det året og døde rett etter utgivelsen av del 2 i 2012. Hans død var et stort tap for både musikk og film. Det må nevnes at del 2 hadde nesten samme sporliste som den annonserte del 1, så det jevnet seg ut til slutt. Så året hadde både gode nyheter og ekstremt dårlige nyheter.

I 2012 skjedde også noe helt forventet: Et team av forskere hoppet på "C-programmering er vanskelig"-trenden. I et dokument kalt *Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines*[10], publisert av en lang liste av navn fra MIT, Stanford og Adobe, hevder de

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

Det er en svært modig påstand. Og jeg mener modig slik ordet brukes i den klassiske TV-serien "Javel, herr statsråd".

Selvfølgelig prøvde de å lansere sitt eget lekespråk for å løse de antatte problemene. Det kunne ikke stå ubesvart, så jeg påstod isteden at **rask C-kode alltid er pen og velorganisert.** Når jeg tenker på det, det er delvis hva denne boken handler om. Merkelig.

De startet artikkelen med noen eksempler for å bevise påstanden sin: Her er det de kalte "Clean C" (referanse 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;
}
```

Her er det de kalte "Fast C++" (referanse 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] https://people.csail.mit.edu/jrk/halide12/halide12.pdf

Jeg har kopiert formateringen eksakt. Jeg vil kalle det henholdsvis amatør-C og amatør-C++.

Hvordan er det mulig å skrive sånn kode? Bare å samle krøllparentesene i en haug på slutten, hva er det for noe? Hvis jeg var en konspirasjonsteoretiker, ville jeg si at dette tullet ble produsert av de som selger CPU-tid per sekund. Jeg tenker på retningslinjene fra *Real Programmers Code of Conduct* fra *PDP-11*-dagene: "If it was hard to write, it should be hard to read." Disse retningslinjene er en eldgammel spøk, men det passet veldig godt her.

Koden skal visstnok være et 3x3 boksfilter. Som påpekt etterpå av en av de få som leser greiene jeg skriver, 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*

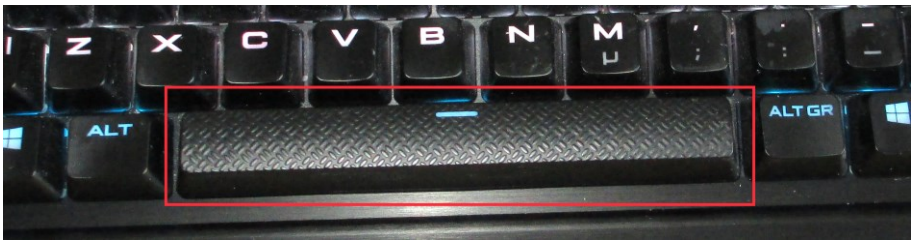
Det høres fornuftig ut. Test det selv! For meg var saken bare enda en dataprosesseringsjobb, jeg brydde meg ikke om matematikken. Ser ikke ut som at forfatterne av artikkelen brydde seg heller, hvilket var merkelig. Så: Enten kaste bort hele dagen på å sette seg inn i hvordan boksfiltere fungerer, eller stole på Zimmermann. Jeg valgte å stole på Zimmerman.

Så det var på tide å børste støvet av SSE2. En rask gjennomgang av noen grunnleggende SSE2-intrinsics: Intel rotet det ikke fullstendig til som ARM, bare litt, så det meste er enkelt og greit: *__m128i* er den grunnleggende datatypen. *_mm_load_si128()/store_si128()* laster og lagrer data. *_mm_add_epi16()* adderer 16-bits verdier og wrapper, mens *_mm_adds_epi16()* saturerer. *_mm_mulhi16()* multipliserer 16-bit verdier, genererer 32-bit resultater og pakker de høye 16 bitene i hvert svar. Veldig beleilig. Og veldig enkelt.

Tilbake til artikkelen. Strømmen av modige påstander fortsatte uforstyrret (min understreking):

> *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>*

Da er det bare å prøve å finne ut nøyaktig hva de mener, ved å bruke en radikal tilnærming. Symbolet mellomrom, den brede tasten nederst på tastaturet, kan brukes til å forbedre lesbarheten, og blanke linjer kan brukes til å gruppere ting. Og LED-lys i tastaturet er underholdende. I omtrent 20 sekunder.



En omformatering av referanse B gir:

```
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 );
                }
            }
        }
    }
}
```

Det gjorde det lesbart, og det er virkelig ikke så veldig vanskelig. Neste steg kunne være å gi noen variabler logiske navn, men det er rett og slett noe grunnleggende galt her. Husk at artikkelen ble skrevet i 2012. Den midlertidige bufferkonstruksjonen skulle ikke være nødvendig med moderne Intel-prosessorer fra det året. Ved nærmere ettersyn fant jeg følgende avsnitt i teksten:

> *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).*

Så det er en uspesifisert *Core2 Quad*, en lett ubrukelig mobilversjon, noen andre brikker jeg ikke brydde meg om på den tiden og en superdyr GPU. Den kostet $3,999 og ble lansert i 2011. Det er ikke en spøk. Uansett rotet jeg rundt i kjelleren og fant en gammel *Intel Core2 Quad Q9550 CPU*, lansert i 2008, der koden i referanse B faktisk kjørte ok. Så: Koden ble optimalisert for en Intel-arkitektur som var minst 4 år den gangen. Et gufs fra fortiden!

Siden 2008 hadde det skjedd noe annet som også var helt forventet: Nyere og bedre CPU-er hadde blitt lansert. En OK og rimelig CPU i 2012 var *Intel i7 "Sandy Bridge" 2600K* som hadde betydelig forbedret hurtigbuffer. Jeg hadde en av dem på hylla. Brukte den ikke siden enda bedre ting hadde kommet ut, men tilgjengelig for testing.

En ting jeg hoppet over den gangen, var å se på hvordan løkkene var konstruert. Vurder den første indreløkken i referanse B. For brikker som ikke har nødvendig I/O tilgjengelig, som jeg antar dekker *OMAP* og den uspesifiserte *Core2*, vil det sannsynligvis være fornuftig å bruke en midlertidig buffer som de gjorde. Eller ville det det? Jeg er ikke fan av å bruke to ujusterte lastinger og en justert når bare en justert og noe enkel omformatering er nødvendig. Det kan gå begge veier siden data kommer til å være i L1-hurtigbufferen ganske raskt. Og så er det spørsmålet om utrulling. Og hvor kostbare ville de ujusterte lastingene være på en så gammel CPU? Mange spørsmål, få svar. Mitt *Core2 Quad*-system er dessverre dødt og begravet, så det er umulig å teste det nå.

Den gangen prøvde jeg bare å gjøre det på rett måte, dvs. så enkelt som mulig. Å lese tre linjer og skrive en var det enkleste jeg kunne tenke meg:

```
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;
        }

    }

}
```

Trivielt. Men ettersom hurtigbufferen var betydelig forbedret på i7-2600K, så var det verdt å prøve å lese fire og skrive to:

```
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;

    }

}
```

Jeg testet å bruke flere, men i7-2600K nådde toppen ved 4 + 2. Ville være morsomt å prøve andre varianter på nyere CPU-er idag. Legg merke til hvor enkelt det er å bruke korte og logiske variabelnavn og formatere koden slik at ting som hører sammen henger sammen!

Det var det eneste eksemplet som ble gitt i dokumentet. Det var mye snakk om hvordan ting fungerte, og litt prat om mer avanserte filtere. Men det var bare prat, ikke kode. Jeg hadde mistet interessen for lenge siden.

Jeg gjorde noen målinger på forskjellige CPU-er den gang for å finne ut hva de brukte. Først ut er en levning fra fortiden, sannsynligvis i samme nabolag som CPU-en forskerne brukte. Iterasjoner satt til 10, fordi den var fryktelig treg:

```
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! Koden deres er faktisk rask på den CPU-en, sammenlignet med den mer moderne tilnærmingen.

Noe fra 2009, året etter: *Intel i7-950*. Fortsatt ganske gammel på den tiden:

```
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%
```

Interessant. En CPU fra 2009 kjører den trivielle SIMD-versjonen mye raskere.

Noe attraktivt priset fra tidsperioden, *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%
```

Også interessant. Mye raskere.

Dette ser ut til å være et tilbakevendende problem helt til i dag. Jeg kan huske å ha sett flere vitenskapelige artikler fylt med eksepsjonelt dårlige eksempler i C og intrinsics. Det er nesten som om de aldri har skrevet C før, eller programmer. Det kan imidlertid unnskyldes hvis grunnideen er god. Det var den ikke her. Det virket som de hadde satt opp en testbenk av rester de hadde liggende og slenge i lageret, og som tilfeldigvis ga gode resultater for det dårlige eksempelet de hadde kokt sammen. Noe som er merkelig, tatt i betraktning at artikkelen har en liste av forfattere fra universiteter og firmaer som har et godt rykte.