Optimizing YUV-RGB Color Space Conversion Using Intel's SIMD Technology

Étienne Dupuis

August 2003

Abstract

Multimedia players must deal with MPEG video streams. Rendering these highly compressed streams on screen requires intensive and lengthy computations. Improving the speed of these computations may have a noticeable impact on the image quality and fluidity. In this document we explore the conversion of a decoded image in YUV color format to the RGB color format, suitable for rendering by the Microsoft Windows operating system. The final step of our optimization uses Intel's Pentium SIMD instructions to drastically improve the speed of the algorithm.

1 YUV Color Space

A large number of computer users have already encountered the RGB color space. A color space provides an association between a set of values and a color. The RGB color space represents colors in terms of red, blue and green intensity. The combination of these values by the electron beam inside a classical monitor allows it to display virtually any color.

However, there are some drawbacks with this representation. For example, the brightness of a pixel may not be changed easily, as it must be computed from the RGB components. These components must then be recalculated with the new intensity in order to obtain the same color, brighter. Standard video signals like PAL[1], NTSC[2] or SECAM[3] hence uses an alternative color scheme, YUV. The Y component represents luminance, or intensity, which is suitable for black and white display devices. With the introduction of color TV two additional signals, U and V, were added to represent color.

As part of the development of a world-wide digital video standard, YUV was redefined as YCbCr and equations were laid out such that the values of the

[1] Phase Alternation Line
[2] National Television System Committee
[3] Séquentiel couleur avec mémoire

three components (which we will still call Y, U and V) fit in the 0-255 range. The equation[4] to convert from the RGB color space is

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix},$$

while the inverse conversion can be achieved with

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.164 & 0 & 1.596 \\ 1.164 & -0.391 & -0.813 \\ 1.164 & 2.018 & 0 \end{bmatrix} \left(\begin{bmatrix} Y \\ U \\ V \end{bmatrix} - \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} \right).$$

Note that if $0 \leq R, G, B \leq 255$ then $16 \leq Y \leq 235$ and $16 \leq U, V \leq 240$, which is perfect for a byte representation.

We conclude this section with a biology fact. The human eye is more sensitive to luminosity than color. Hence the U and V components may be undersampled to lower an image's byte size, thus improving transmission speeds and saving disk space. For example, YCbCr 4:2:0 uses one byte per pixel for the Y component and one byte for each 2×2 pixel region for the two color components.

2 Algorithm to Optimize

The algorithm we will optimize is the conversion from YCbCr 4:2:0 to RGB. The input consists of three byte buffers containing Y, U and V values. For a $w \times h$ image, the Y buffer is wh bytes long, values being stored line by line:

$$Y_{i,j} = y_{jw+i},$$

where y_k denotes the kth byte of the buffer and $Y_{i,j}$ is the Y value for pixel in ith column, jth row. The U and V buffers are both $wh/4$ bytes long, as every value is shared by four pixels, in 2×2 pixel squares. The correct offset is computed with

$$U_{i,j} = u_{\lfloor \frac{jw}{4} + \frac{i}{2} \rfloor},$$

where $\lfloor x \rfloor$ is the integer part of a real number. The output buffer is $4wh$ bytes long, each pixel being represented by a 32-bit value: 8 bit for the blue component, 8 bit for green, 8 bit for red and 8 bits which must be set to zero.

For the remaining of our discussion, we will make the following assumptions :

1. The image has even width and height.
2. There are no "gaps" between lines of the image in memory. This is often the case for monochrome image formats (one bit/pixel), where lines always start on a new byte, even if the width is not a multiple of 8.

[4]There are in fact different equations for SDTV and HDTV but they differ only in the matrix coefficients. In this document, we have used coefficients for SDTV.

3. A **char** occupies one byte of memory, a **short** two and an **int** four. This is the case on all compilers for the Pentium processor.

4. Integers are stored in memory in little-endian format. Again this is the case for the Pentium processor.

A straightforward C/C++ implementation of the algorithm is given by

```
for (int h = 0; h < height; h++) {
  for (int w = 0; w < width; w++) {
    int k = h * width + w;
    int i = (h / 2) * (width / 2) + (w / 2);

    int Y = y[k];
    int U = u[i];
    int V = v[i];

    double R = 1.164 * (Y - 16) + 1.596 * (V - 128);
    double G = 1.164 * (Y - 16) - 0.391 * (U - 128) - 0.813 * (V - 128);
    double B = 1.164 * (Y - 16) + 2.018 * (U - 128);

    if (R < 0.0)
      R = 0.0;
    if (G < 0.0)
      G = 0.0;
    if (B < 0.0)
      B = 0.0;

    if (R > 255.0)
      R = 255.0;
    if (G > 255.0)
      G = 255.0;
    if (B > 255.0)
      B = 255.0;

    unsigned int RGB =
      (unsigned int)(B + 0.5) |
      ((unsigned int)(G + 0.5) << 8) |
      ((unsigned int)(R + 0.5) << 16);

    rgb[k] = RGB;
  }
}
```

and is encapsulated in function prototype

```
void YUVRGBConversion(
  const unsigned char *y,
  const unsigned char *u,
  const unsigned char *v,
  int width,
  int height,
  unsigned int *rgb
);
```

In the above implementation, the RGB values have been saturated between 0 and 255. This operation would be superfluous if we could be sure that the YUV values were computed from a valid RGB triplet. However, the YUV values may come from a stream where noise or compression artefact have pushed the values out of the predefined range. Finally, adding 0.5 before converting the floating

point values to integers is a simple trick to round the real value to the nearest integer rather than simply truncating the fractional part.

3 Basic Optimizations

All the optimizations described in this section will be performed on the C/C++ code.

3.1 Removing Floating-Point Computations

The first observation relates to the use of floating point arithmetic. The precision provided by floating-point computations is of no concern in this algorithm. For a sufficiently large integer K, the following approximation satisfies our needs:

$$\frac{1}{K}\begin{bmatrix}\lfloor 1.164K\rceil & 0 & \lfloor 1.596K\rceil \\ \lfloor 1.164K\rceil & \lfloor -0.391K\rceil & \lfloor -0.813K\rceil \\ \lfloor 1.164K\rceil & \lfloor 2.018K\rceil & 0\end{bmatrix} \approx \begin{bmatrix}1.164 & 0 & 1.596 \\ 1.164 & -0.391 & -0.813 \\ 1.164 & 2.018 & 0\end{bmatrix},$$

where $\lfloor x\rceil$ the nearest integer to the real number x. We compute the new matrix coefficients as follow:

```
static const int Precision = 32768;
static const int CoefficientY = (int)(1.164 * Precision + 0.5);
static const int CoefficientRV = (int)(1.596 * Precision + 0.5);
static const int CoefficientGU = (int)(0.391 * Precision + 0.5);
static const int CoefficientGV = (int)(0.813 * Precision + 0.5);
static const int CoefficientBU = (int)(2.018 * Precision + 0.5);
```

Note that we have chosen a power of two for K (Precision) as dividing by a power of two may be accomplished with a bit shift, which is much faster than an ordinary division. We also added 0.5 before truncating in order to round the value to the nearest integer, thus minimizing errors due to the approximation. Our inner loop now looks like

```
    int R = CoefficientY * (Y - 16) + CoefficientRV * (V - 128);
    int G = CoefficientY * (Y - 16) - CoefficientGU * (U - 128)
                    - CoefficientGV * (V - 128);
    int B = CoefficientY * (Y - 16) + CoefficientBU * (U - 128);

    R = (R + Precision / 2) / Precision;
    G = (R + Precision / 2) / Precision;
    B = (R + Precision / 2) / Precision;

    if (R < 0)
      R = 0;
    if (G < 0)
      G = 0;
    if (B < 0)
      B = 0;

    if (R > 255)
      R = 255;
    if (G > 255)
```

```
      G = 255;
    if (B > 255)
      B = 255;

    rgb[k] = B | (G << 8) | (R << 16);
```

The reason for adding Precision/2 before dividing is, as before, to improve the accuracy of the approximation induced by the truncation. This step could be omitted without introducing much error though. Note that we leave to the compiler the role of optimizing registry usage to compute $Y - 16$ only once for example.

3.2 Removing Multiplications

Multiplication is a costly operation on any processor. In the case of our algorithm, multiplications consist of a constant times a single byte value. We can hence replace them with table lookups. Since there are five different coefficients in the matrix, we need five tables with an entry for each possible byte value. Here is how they look like:

```
static const int CoefficientsGU[256] = {
  -CoefficientGU * (0x00 - 128),
  -CoefficientGU * (0x01 - 128),
  -CoefficientGU * (0x02 - 128),
  -CoefficientGU * (0x03 - 128),
...

static const int CoefficientsY[256] = {
  CoefficientY * (0x00 - 16) + (Precision / 2),
  CoefficientY * (0x01 - 16) + (Precision / 2),
  CoefficientY * (0x02 - 16) + (Precision / 2),
  CoefficientY * (0x03 - 16) + (Precision / 2),
...
```

You will note that the value Precision/2 have been included in the table for the Y coefficient. It saves us from adding it after. The matrix multiplication in our inner loop becomes

```
    int R = CoefficientsY[Y] + CoefficientsRV[V];
    int G = CoefficientsY[Y] + CoefficientsGU[U] + CoefficientsGV[V];
    int B = CoefficientsY[Y] + CoefficientsBU[U];

    R /= Precision;
    G /= Precision;
    B /= Precision;
```

3.3 Removing Conditional Tests

Many modern processors suffer a penalty when jumping from one point of execution to another. These jumps may occur when evaluating `if` statements, depending on the result. Although the more recent Pentium have complex jump

prediction algorithms built-in, the best is still to avoid jumps. After investigation, we note that the RGB values as computed by our algorithm are bounded. The minimal value is attained with $(Y, U, V) = (0, 0, 0)$ and the maximal value with $(Y, U, V) = (255, 255, 255)$, in which cases $B = -297.984$ and $R = 683.580$. Thus, even with the approximations in the computations, the values fir in a 1024-wide interval:

$$-320 \leq R, G, B < 704. \tag{1}$$

We hence define a table T with 1024 entries such that

$$T_i = \min\{\max\{i - 320, 0\}, 255\}.$$

In fact, we will build a table for each RGB component such that the table includes the bitshift we use to put in place the component in the final RGB value. Moreover, using a pointer to T_{320} as the base pointer of our array allows us to index the array with negative values:

```
static unsigned int _CoefficientsR[1024] = {
  0x000000,
  0x000000,
  ...
  0x000000,
  0x010000,
  0x020000,
  ...
  0xFE0000,
  0xFF0000,
  0xFF0000,
  ...

unsigned int *CoefficientsR = &_CoefficientsR[320];
```

Our loop is now quite tight:

```
int k = 0;
for (int h = 0; h < height; h++) {
  for (int w = 0; w < width; w++, k++) {
    int i = (h / 2) * (width / 2) + (w / 2);

    int Y = y[k];
    int U = u[i];
    int V = v[i];

    int R = CoefficientsY[Y] + CoefficientsRV[V];
    int G = CoefficientsY[Y] + CoefficientsGU[U] + CoefficientsGV[V];
    int B = CoefficientsY[Y] + CoefficientsBU[U];

    rgb[k] = CoefficientsR[R / Precision] |
          CoefficientsG[G / Precision] |
          CoefficientsB[B / Precision];
  }
}
```

3.4 Four pixels at once

The final optimization is based on computing four pixels per loop iteration, the U and V values being shared by precisely four pixels. This implies processing

pixels on two scan lines simultaneously. We also replace the array access to the image data by moving pointers. Here is the code:

```
const unsigned char *y0 = y;
const unsigned char *y1 = y + width;

unsigned int *rgb0 = rgb;
unsigned int *rgb1 = rgb + width;

for (int h = 0; h < height; h += 2) {
  for (int w = 0; w < width; w += 2) {
    int U = *u++;
    int V = *v++;

    int RUV = CoefficientsRV[V];
    int GUV = CoefficientsGU[U] + CoefficientsGV[V];
    int BUV = CoefficientsBU[U];

    int Y = *y0++;
    int R = CoefficientsY[Y] + RUV;
    int G = CoefficientsY[Y] + GUV;
    int B = CoefficientsY[Y] + BUV;
    *rgb0++ = CoefficientsR[R / Precision] |
          CoefficientsG[G / Precision] |
          CoefficientsB[B / Precision];

    Y = *y0++;
    R = CoefficientsY[Y] + RUV;
    G = CoefficientsY[Y] + GUV;
    B = CoefficientsY[Y] + BUV;
    *rgb0++ = CoefficientsR[R / Precision] |
          CoefficientsG[G / Precision] |
          CoefficientsB[B / Precision];

    Y = *y1++;
    R = CoefficientsY[Y] + RUV;
    G = CoefficientsY[Y] + GUV;
    B = CoefficientsY[Y] + BUV;
    *rgb1++ = CoefficientsR[R / Precision] |
          CoefficientsG[G / Precision] |
          CoefficientsB[B / Precision];

    Y = *y1++;
    R = CoefficientsY[Y] + RUV;
    G = CoefficientsY[Y] + GUV;
    B = CoefficientsY[Y] + BUV;
    *rgb1++ = CoefficientsR[R / Precision] |
          CoefficientsG[G / Precision] |
          CoefficientsB[B / Precision];
  }

  rgb0 += width;
  rgb1 += width;
  y0 += width;
  y1 += width;
}
```

4 Advanced SIMD Optimizations

SIMD stands for Single Instruction, Multiple Data. Introduced with the Pentium MMX, this technology allows the cpu to perform computations on up to eight data registers simultaneously. The Pentium MMX and its successors con-

tain eight 64-bit MMX registers. Computations with these registers operate simultaneously on either 2 four-byte values, 4 two-byte values or 8 single bytes.

Our algorithm produces for each pixel four bytes of data: a null byte and three color bytes. Hence it is a likely candidate to be reworked to use SIMD instructions. Since there is an instruction to pack 4 two-byte values into four bytes, the ideal would be to do all of our computations within 16-bit registers. Doing so requires the Precision constant, which control the precision of the integer calculations, to be 64. In such a case, as pointed by the inequality 1, the intermediate values for R, G and B would be between -20480 and 45056, the latter value being still to large to fit in a signed 16-bit register. However, if YUV values are bounded by 235, 240 and 240, the maximum intermediate value is 30780, which is smaller than 2^{15}. The solution is to incorporate into the coefficient tables these bounds:

```
#define RGBY(i) { \
  (short)(1.164 * 64 * (i - 16) + 0.5), \
  (short)(1.164 * 64 * (i - 16) + 0.5), \
  (short)(1.164 * 64 * (i - 16) + 0.5), \
  0x00, \
}

static const short CoefficientsRGBY[256][4] = {
  RGBY(0x10), RGBY(0x10), RGBY(0x10), RGBY(0x10),
  RGBY(0x10), RGBY(0x10), RGBY(0x10), RGBY(0x10),
  RGBY(0x10), RGBY(0x10), RGBY(0x10), RGBY(0x10),
  RGBY(0x10), RGBY(0x10), RGBY(0x10), RGBY(0x10),
  RGBY(0x10), RGBY(0x11), RGBY(0x12), RGBY(0x13),
  RGBY(0x14), RGBY(0x15), RGBY(0x16), RGBY(0x17),
  ...
  RGBY(0xE8), RGBY(0xE9), RGBY(0xEA), RGBY(0xEB),
  RGBY(0xEB), RGBY(0xEB), RGBY(0xEB), RGBY(0xEB),
  RGBY(0xEB), RGBY(0xEB), RGBY(0xEB), RGBY(0xEB),
  RGBY(0xEB), RGBY(0xEB), RGBY(0xEB), RGBY(0xEB),
  RGBY(0xEB), RGBY(0xEB), RGBY(0xEB), RGBY(0xEB),
  RGBY(0xEB), RGBY(0xEB), RGBY(0xEB), RGBY(0xEB),
};

#define RGBU(i) { \
  (short)(2.018 * 64 * (i - 128) + 0.5), \
  (short)(-0.391 * 64 * (i - 128) + 0.5), \
  0x00, \
  0x00, \
}

static const short CoefficientsRGBU[256][4] = {
  RGBU(0x10), RGBU(0x10), RGBU(0x10), RGBU(0x10),
  ...

#define RGBV(i) { \
  0x00, \
  (short)(-0.813 * 64 * (i - 128) + 0.5), \
  (short)(1.596 * 64 * (i - 128) + 0.5), \
  0x00, \
}

static const short CoefficientsRGBV[256][4] = {
  RGBV(0x10), RGBV(0x10), RGBV(0x10), RGBV(0x10),
  ...
```

The tables have also been laid out such that a single table entry is composed of

4 two-byte values. This layout allows to load in one operation a 64-bit MMX register with the coefficients for all three color components.

Before going further, we should consider the approximation errors introduced by setting Precision to 64. Each table entry has been rounded to the nearest integer and hence may be up to 0.5 off from the exact value. The sum of three of these entries has thus an error less than or equal to 1.5. The computed value is then divided by 64 and rounded again. The error, 1.5/64 (1/48), at worst causes rounding to the incorrect integer value after the division. The maximum error on each RGB component is hence ± 1, which is quite reasonable.

Using the MMX registers in an optimal way requires the use of assembly language. With Microsoft's C/C++ Compiler, directives __declspec(naked) and __cdecl can be used to write an assembly language function without any interference from the compiler. The body of our function starts with

```
__asm {
  pushad
  finit
```

The **pushad** instruction pushes all registers on the stack. They will be restored before leaving the function. This is always safer when mixing assembly language within C/C++ code as compilers make some assumptions on which registers are preserved by a function call. The **finit** instruction resets the floating-point unit. MMX registers and FPU registers are in fact the same and can not be used simultaneously. Switching from FPU registers to MMX mode requires the floating point register unit to be empty, hence the **finit** instruction.

```
  xor       eax, eax
  mov       ebx, [esp + 32 + 20]
  mov       ecx, [esp + 32 + 16]
  mov       edx, [esp + 32 + 4]
  mov       edi, [esp + 32 + 8]
  mov       esi, [esp + 32 + 12]
  mov       ebp, [esp + 32 + 24]
```

We then load all variables in registers: image width in **ecx**, height in **ebx**, pointers to YUV buffers in **edx**, **edi** and **esi**, pointer to output RGB buffer in **ebp**. Register **eax** is zeroed for later use. The offset of 32 bytes is caused by the **pushad** instruction, which stores 32 bytes of data on the stack.

```
hloop :
  push      ebx
  mov       ebx, ecx
```

This is the beginning of the loop on the image lines. At this point, **ebx** contains the number of lines left to process. We pushes it on the stack and set **ebx** to the image width, in order to proceed with the next loop.

```
wloop :
  push      ebx
  xor       ebx, ebx
```

Now the loop on each line pixels. **ebx**, the number of pixels left to process on the line, is pushed on the stack and zeroed for later use.

```
mov     al, [edi]
mov     bl, [esi]
movq    mm0, [CoefficientsRGBU + 8 * eax]
paddw   mm0, [CoefficientsRGBV + 8 * ebx]
```

In the above four lines, we start by loading U and V in **al** and **bl**. Since **eax** and **ebx** were previously zeroed, the upper 24 bits are zero hence the registers can be immediately used to compute the correct offset in the coefficient tables. The third instruction loads 64 bits of data (four 2-byte values) into MMX register **mm0**, which will contain

mm0	B_U	G_U	0000	0000

,

where B_U is the integer part of $2.018 * 64 * (i - 128) + 0.5$ and so on. Finally, the fourth instruction adds in the values for V. All four values are added simultaneously:

mm0	B_U	$G_U + G_V$	R_V	0000

.

All that in four instructions!

```
mov     al, [edx]
mov     bl, [edx + 1]
movq    mm1, [CoefficientsRGBY + 8 * eax]
movq    mm2, [CoefficientsRGBY + 8 * ebx]
```

Similarly, the above instructions load in **mm1** and **mm2** the results of the matrix coefficient multiplication with $Y - 16$ for two consecutive pixels:

mm1	B_{Y_1}	G_{Y_1}	R_{Y_1}	0000

,

mm2	B_{Y_2}	G_{Y_2}	R_{Y_2}	0000

.

```
mov     al, [edx + ecx]
mov     bl, [edx + ecx + 1]
movq    mm3, [CoefficientsRGBY + 8 * eax]
movq    mm4, [CoefficientsRGBY + 8 * ebx]
```

We do the same for the two pixels on the next scan line. As with our C/C++ algorithm, we will process four pixels in a single loop iteration.

```
paddw   mm1, mm0
paddw   mm2, mm0
paddw   mm3, mm0
paddw   mm4, mm0

psraw   mm1, 6
psraw   mm2, 6
psraw   mm3, 6
psraw   mm4, 6
```

The above eight instructions finish the computation by adding Y coefficients to the previously computed values and arithmetically shifting right by 6 bits, thus dividing by 64. The registers now contain

mmi	B_i	G_i	R_i	0000

.

Note that the values are in the $-20480/64 = -320$ to $30780/64 = 481$ range, as explained earlier.

```
packuswb   mm1, mm2
packuswb   mm3, mm4
```

Now the powerful instruction **packuswb**. This instruction packs eight 2-byte values from two MMX registers in a single register. Values lower than zero are set to zero while ones superior to 255 are saturated, which is exactly what we need! After this operation, we have

mm1	B_2	G_2	R_2	00	B_1	G_1	R_1	00

,

mm3	B_3	G_3	R_3	00	B_4	G_4	R_4	00

.

The four pixels are ready to be stored in memory.

```
movq       [ebp], mm1
movq       [ebp + 4 * ecx], mm3
```

We will see in the next section that these two memory write instructions introduce a considerable latency in the flow of execution.

```
add        ebp, 8
add        edx, 2
add        edi, 1
add        esi, 1
```

Pointers are incremented according to the size of the data required to represent two consecutive pixels on a single row.

```
pop        ebx
sub        ebx, 2
jnz        wloop
```

We loop on pairs of pixels. Recall that the value pop from the stack is the number of pixels left to process.

```
lea        ebp, [ebp + 4 * ecx]
add        edx, ecx
```

At the end of each line, we must adjust pointers to skip a line as our inner loop processes pixels from two lines simultaneously. In this code, the **lea** instruction is used to add four times the value of **ecx** to **ebp** in short simple way.

```
  pop       ebx
  sub       ebx, 2
  jnz       hloop
```

We loop on every line pairs of the image.

```
  emms
  popad
  ret
}
```

The function ends with two cleanup instruction and a return. The first instruction, **emms**, frees the MMX processing unit, leaving it ready to eventually perform floating-point calculations. The second instruction, **popad** restores the values of all registers modified by the function.

The complete code of the function is listed below. It is composed of only 52 instructions and runs much faster than our last C/C++ function. The results are consigned in section 6.

```
__asm {
  pushad
  finit

  xor       eax, eax
  mov       ebx, [esp + 32 + 20]
  mov       ecx, [esp + 32 + 16]
  mov       edx, [esp + 32 + 4]
  mov       edi, [esp + 32 + 8]
  mov       esi, [esp + 32 + 12]
  mov       ebp, [esp + 32 + 24]

hloop :
  push      ebx
  mov       ebx, ecx

wloop :
  push      ebx
  xor       ebx, ebx

  mov       al, [edi]
  mov       bl, [esi]
  movq      mm0, [CoefficientsRGBU + 8 * eax]
  paddw     mm0, [CoefficientsRGBV + 8 * ebx]

  mov       al, [edx]
  mov       bl, [edx + 1]
  movq      mm1, [CoefficientsRGBY + 8 * eax]
  movq      mm2, [CoefficientsRGBY + 8 * ebx]

  mov       al, [edx + ecx]
  mov       bl, [edx + ecx + 1]
  movq      mm3, [CoefficientsRGBY + 8 * eax]
  movq      mm4, [CoefficientsRGBY + 8 * ebx]

  paddw     mm1, mm0
  paddw     mm2, mm0
  paddw     mm3, mm0
  paddw     mm4, mm0

  psraw     mm1, 6
  psraw     mm2, 6
```

```
  psraw       mm3, 6
  psraw       mm4, 6

  packuswb    mm1, mm2
  packuswb    mm3, mm4

  movq        [ebp], mm1
  movq        [ebp + 4 * ecx], mm3

  add         ebp, 8
  add         edx, 2
  add         edi, 1
  add         esi, 1

  pop         ebx
  sub         ebx, 2
  jnz         wloop

  lea         ebp, [ebp + 4 * ecx]
  add         edx, ecx

  pop         ebx
  sub         ebx, 2
  jnz         hloop

  emms
  popad
  ret
}
```

5 Further Optimizations?

A few years ago, it was still possible to count clock cycles *before* actually executing the code. Modern Pentium processors include technology features which greatly complexify such a task. Instructions are decoded and translated into micro-instructions, which may be executed out-of-order, cache misses may introduce considerable latency, instructions are fetched simultaneously into various processing units from different ports and so on. Moreover, as advanced features are added to processors, optimizations may be more or less effective, as the results in the next section show.

Careful experimentation with the above function leads us to an interesting fact: most of the time is spent in the two memory write instructions! In such a case, Intel's optimization guide [2] suggests to use non-temporal stores. These special memory write instructions hint the processor to bypass cache allocation for the memory to be written. The benefit is double: cache "pollution" is reduced and write latency is improved. However, these instructions must be used with care as severe penalties may occur if the written memory is accessed shortly after. We thus replace the two write instructions with

```
  movntq      [ebp], mm1
  movntq      [ebp + 4 * ecx], mm3
```

The effect of this modification varies drastically from processor models to others.

Intel's optimization guide also suggest to write sequentially to memory whenever possible. As our assembly version processes two scan lines simultaneously, memory writes alternate between two regions. The solution to prevent these non-sequential memory access would be to process only two pixels per iteration. The price to pay is that more instructions will be executed to perform the full image conversion. We leave the realization of this version as an exercice to the interested reader. Results, which are quite instructive, are given int the following section.

6 Results

In order to test the speed improvements given by the optimizations described in this document, the code has been compiled with two different compilers on three different processors. The code was first ran on a 886×806 bitmap depicting "il Festino degli dei", a painting by the Early Renaissance Italian artist Giovanni Bellini. It was also ran on a randomly generated 4000×3000 image.

The three computer configurations tested were:

1. A 750MHz Pentium III on a laptop computer running Windows 2000 under 256Mb of RAM.
2. A 2GHz Athlon XP running Windows 2000 under 512Mb of RAM.
3. A 2.4GHz Pentium IV running Windows XP under 512Mb of RAM.

The two compilers where Microsoft C/C++ Compiler 13.0, included in the .NET Framework, and mingw, a Win32 port of GCC 3.2. No special optimization compilation directives were used. The results, expressed in milliseconds, are summarized in tables 1 and 2. No particular care was taken to make very precise timings, hence from one run to another small variations were noticed. The values printed are the minimum values obtained in a few runs.

These results clearly show that the optimizations are more or less effecient depending on the target processor and the compiler used. For example, let us consider the replacement of multiplications by table lookups. With GCC, the speed improvement is significant, from 1.36 to 1.61 times faster with the random image while with Microsoft's compiler, the improvement factor is at most 1.24.

Another interesting result is the effect of non-temporal stores. On Intel's processors, the effect is clearly benefic. However, on AMD's Athlon, there is an outstanding penalty when we use this special feature. However, when we put one **movntq** instruction and one **movq** instruction, there is no penalty but rather a 33% speed improvement! As this strange behavior seems specific to the Athlon, the results were not included in the tables.

	P3		Athlon		P4	
	gcc	vc++	gcc	vc++	gcc	vc++
Floating-Point Code	182.3	142.4	47.1	55.2	39.5	106.6
Integer Multiplication	70.5	41.9	22.6	18.7	30.0	21.1
No Multiplications	40.3	38.4	14.7	15.1	12.6	13.6
No Conditionals	30.9	32.9	12.6	16.0	8.6	10.9
Four Pixels	29.4	27.9	10.8	13.7	7.2	8.1
Assembly Code		17.7		5.0		4.4
Non-Temporal Stores		16.2		11.9		3.5
Two Pixels		19.7		2.8		2.8

Table 1: Timings for 886×806 Giovanni Bellini painting

	P3		Athlon		P4	
	gcc	vc++	gcc	vc++	gcc	vc++
Floating-Point Code	3356.4	2971.9	996.2	1124.3	1083.9	2174.2
Integer Multiplication	1496.3	991.6	488.3	409.4	782.2	664.2
No Multiplications	957.0	943.3	356.9	366.8	483.8	531.9
No Conditionals	654.5	547.4	270.7	277.8	335.4	294.6
Four Pixels	646.1	483.4	233.2	234.2	236.7	189.2
Assembly Code		286.6		84.9		72.8
Non-Temporal Stores		258.2		196.7		57.3
Two Pixels		318.8		48.5		56.1

Table 2: Timings for 4000×3000 random image

In conclusion, SIMD instructions did brought a significant speed improvement, especially on the more recent processors. However, extreme care must be taken with memory writes as they may considerably slow down an algorithm, as the better timings for processing two pixels per iteration rather than of four demonstrate.

References

[1] Instruction set reference. In *IA-32 Intel Architecture Software Developer's Manual*, volume 2. Intel Corporation, 1997-2003.

[2] *IA-32 Intel Architecture Optimization.* Intel Corporation, 1999-2003.

[3] Intel Corporation. Using MMX instructions to convert RGB to YUV color conversion. 1996.

[4] Keith Jack. *Video demystified: a handbook for the digital engineer*, chapter 3, pages 15–34. LLH Technology Publishing, 2001.