Application Note

Fast Color Conversion Using Streaming SIMD Extensions and MMX™ Technology

November 28, 2001

Software Solutions Group

Table of Contents

TABLE OF FIGURES

1.0 Introduction

A very basic understanding of the Pentium® 4 architecture and instructions is expected for understanding this technique. Information on the Pentium® 4 processor including documentation, optimization guide, and reference manuals can be found at: http://developer.intel.com/ids.

Datatype color conversions are a common requirement in 3-D application pipelines. In a simple lighting scheme, these conversions happen at least once per color channel, red, green, blue (R, G, B) per vertex, and in a more realistic lighting scheme, such values are often calculated multiple times for different light components. Color conversion from single-precision floating point values to integer values is typically done by using simple casts, for example intvalR = (int)fpvalR; where intval is an integer and fpval a single-precision floating point number.

2.0 Background & Basics

This approach can be less than optimal on an out-of-order execution CPU architecture with a large execution pipeline such as the Pentium® 4 processor. The technique described in this paper makes possible an alternative, more architectural friendly approach to using simple high-level-language casts to solve this problem on data sets of specific configurations. The result is a faster method for doing such conversions using the Streaming SIMD Extension (SSE) instructions and their associated registers, in conjunction with the MMX™ technology registers and instructions.

2.1 A Common Example

The following pseudocode applies diffuse and specular lighting to a vertice:

```
// D3D MACRO
D3DRGB(r, g, b) \
(0xff000000L | ( ((long)((r) * 255)) << 16) | \
(((long)((g) * 255)) << 8) | (long)((b) * 255))

void CLightTransform::Light_x87c()
{
// Set initial ambient RGB

// Calculate the local diffuse RBB color

for( idx=0; idx<=numVertices; idx++ )
{
// Calculate Diffuse RGB contribution
// Note this is done per vertex!
DiffuseColor.r =
DiffuseColor.g =
DiffuseColor.b =

// Assign color to screen vertices
ScreenVertices[idx].color = D3DRGB(DiffuseColor.r, DiffuseColor.g, DiffuseColor.b );

// Calculate specular RGB contribution
// Note this is done per vertex!
```

```
20  SpecularColor.r =
21  SpecularColor.g =
22  SpecularColor.b =

23  // Assign color to screen vertices
24  ScreenVertices[idx].specular = D3DRGB(SpecularColor.r, SpecularColor.g,
    SpecularColor.b );
25  }
26  } // Light_x87c
```

Listing 1

On lines 17 and 24 in Listing 1, we call the D3DRGB macro (as defined on lines 2-4 and by the Microsoft DirectX API interface) to perform our casts and color conversions for both specular and diffuse lighting. Using Microsoft* Visual Studio 6.0 with the Microsoft C++ compiler, these casts each break down to the following simple example:

```
1   float fpnum;
2   intnum = (long)dy_sum ;

3   fld       dword ptr [fpnum]
4   call      __ftol
5   mov       dword ptr [intnum], eax
```

Listing 2

Whereas each call to __ftol on line 4 of Listing 2 equates to:

```
1   __ftol:
2   push      ebp
3   mov       ebp,esp
4   add       esp,fffffff4
5   wait
6   fnstcw    [ebp-02]
7   wait
8   mov       ax,word ptr [ebp-02]
9   or        ah,0c
10  mov       word ptr [ebp-04],ax    ; Partial register stall
11  fldcw     [ebp-04]                ; Instruction stream serialization
12  fistp     qword ptr [ebp-0c]
13  fldcw     [ebp-02]                ; Instruction stream serialization
14  mov       eax,dword ptr [ebp-0c]  ; memory stall
15  mov       edx,dword ptr [ebp-08]
16  leave
17  ret
```

Listing 3

2.2 A Closer Look at the Instruction Stream

This instruction stream contains a few issues that constrain the processor from optimally processing the code. These issues are highlighted in the comments contained in Listing 3 above. To summarize:

- The partial stall occurs on line 10 when a 16-bit register (for example, AX) is read immediately after an 8-bit register (for example, AL, AH) is written, the read is stalled until the write retires.
- The instruction stream becomes serialized on line 11 as the floating-point control word is loaded with the `fldcw` instruction. Serializing instructions constrain

speculative execution by defeating the dynamic execution feature of the processor. Examples of offending instructions include `fldcw` and `cpuid`.

- The memory stall occurs on line 14 because the 32-bit load to eax was preceded by a 16-bit store from ax.

See the Pentium® 4 processor documentation, optimization guide, and reference manuals at: http://developer.intel.com/ids for full details on specific architectural features and coding pitfalls.

The casts performed on each R, G, B channel for diffuse and specular lighting in Listing 1 can be improved. The method described below improves this method in two ways:

1) It avoids the less-optimal code paths and the computational cost associated with such methods.

2) It improves the throughput of the data by using a SIMD method. For example, in the case of the Pentium® 4 processor, throughput increases from processing 1 data element at a time to processing 4 data elements at a time.

3.0 The New and Improved Algorithm

We'll limit our further discussion to converting the diffuse light components, although the same technique would apply to the specular components.

This technique will make two assumptions:

1) Data for SIMD instructions is efficiently organized in a structure of arrays format (SOA), where the vertex data is in a XXXX YYYY ZZZZ WWWW format in the registers or data types, rather than a XYZW XYZW XYZW XYZW, array of structures, (AOS) format.

2) The rendering engine used requires data to be submitted as XYZW, this is a common practice for both hardware accelerators and software rasterizers, and required by most APIs.

It is possible to use variations of this technique with different data structures, however that discussion is outside the scope of this document.

In its simplest form, the algorithm takes SIMD data (RGB-SOA) in the form:

$R3_{FP}$	$R2_{FP}$	$R1_{FP}$	$R0_{FP}$
$G3_{FP}$	$G2_{FP}$	$G1_{FP}$	$G0_{FP}$
$B3_{FP}$	$B2_{FP}$	$B1_{FP}$	$B0_{FP}$

Figure 1: RGB-SOA Data

And converts it to integers in the following (RGB-AOS) SIMD format:

--	$R3_{INT}$	$G3_{INT}$	$B3_{INT}$
--	$R2_{INT}$	$G2_{INT}$	$B2_{INT}$
--	$R1_{INT}$	$G1_{INT}$	$B1_{INT}$
--	$R0_{INT}$	$G0_{INT}$	$B0_{INT}$

Figure 2: RGB-AOS Data

Using the following steps:

1) Convert SIMD-Floating Point data into SIMD-Integer data
2) Shift data left to prepare for packing
3) Combine the integer data in a packed format using logical operations

3.1 Digging a Little Deeper

1) Convert SIMD-FP data into SIMD-INT data. Assume the following components are arranged in the registers as described in figure 5.

	127-96	95-64	63-32	31-0		63..32	31..0	
xmm2	$R3_{FP}$	$R2_{FP}$	$R1_{FP}$	$R0_{FP}$	to	$000000R1_{INT}$	$000000R0_{INT}$	mm6
						$000000R3_{INT}$	$000000R2_{INT}$	mm7
xmm5	$G3_{FP}$	$G2_{FP}$	$G1_{FP}$	$G0_{FP}$	to	$000000G1_{INT}$	$000000G0_{INT}$	mm1
						$000000G3_{INT}$	$000000G2_{INT}$	mm4
xmm1	$B3_{FP}$	$B2_{FP}$	$B1_{FP}$	$B0_{FP}$	to	$000000B1_{INT}$	$000000B0_{INT}$	mm2
						$000000B3_{INT}$	$000000B2_{INT}$	mm5

Figure 3: initial data configuration

Using the following code,

```
cvtps2pi       mm6,  xmm2            // mm6 = (int)r1,(int)r0
shufps         xmm2, xmm2, 0xEE      // r3,r2,r3,r2
cvtps2pi       mm7,  xmm2            // mm7 = (int)r3,(int)r2
cvtps2pi       mm1,  xmm5            // mm1 = (int)g1,(int)g0
shufps         xmm5, xmm5, 0xEE      // g3,g2,g3,g2
cvtps2pi       mm4,  xmm5            // mm4 = (int)g3,(int)g2
cvtps2pi       mm2,  xmm1            // mm2 = (int)b1,(int)b0
shufps         xmm1, xmm1, 0xEE      // b3,b2,b3,b2
cvtps2pi       mm5,  xmm1            // mm5 = (int)b3,(int)b2
```

The `cvtps2pi` will convert packed single precision floating-point data to a packed integer representation, the truncate / chop-rounding mode is implicitly encoded in the instruction, thereby taking precedence over the rounding mode specified in the MXCSR register. This can eliminate the need to change the rounding mode from round-nearest, to truncate / chop, and then back to round-nearest to resume computation.

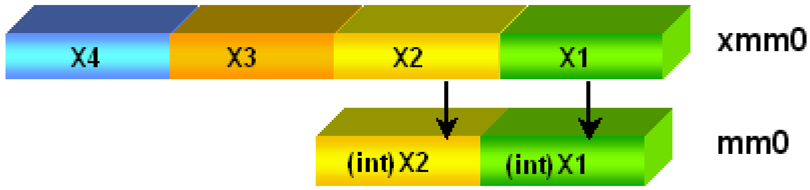

Figure 4: cvtps2pi instruction

We then shuffle the data using the `shufps` instruction. The shuffle instructions use a one byte immediate, where the high and low order nibbles correspond to the two low and two high elements in the destination register, each two bits are then used to identify which element in the source register to move to that location in the destination register. The best way to understand the functionality is through an example. If we were to take the instruction `shufps xmm0, xmm1, 0x2D`, we would achieve the result in figure 5.

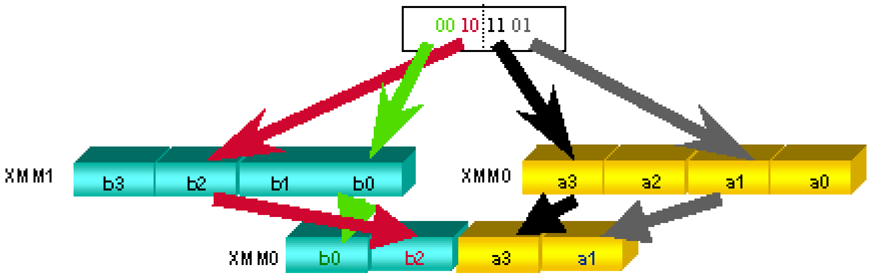

Figure 5: Shuffle Example

2) Shift data left to prepare for packing using the `pslld` instruction 16, 8, or 0 bits to prepare for packing while shifting in 0's. We won't be shifting the blue components, as they will simply be OR'ed together to achieve the final result.

63..32	31..0	
$00R1_{INT}0000$	$00R0_{INT}0000$	mm6
$00R3_{INT}0000$	$00R2_{INT}0000$	mm7
$0000G1_{INT}00$	$0000G0_{INT}00$	mm1
$0000G3_{INT}00$	$0000G2_{INT}00$	mm4
$000000B1_{INT}$	$000000B0_{INT}$	mm2
$000000B3_{INT}$	$000000B2_{INT}$	mm5

Figure 6: Shifting the Data

Using the following code:

```
pslld        mm6, 0x10             // mm6 = r1<<16,r0<<16
pslld        mm7, 0x10             // mm7 = r3<<16,r2<<16
pslld        mm1, 0x08             // mm1 = g1<<8,g0<<8
pslld        mm4, 0x08             // mm4 = g3<<8,g2<<8
```

3) Combine the integer data in a packed format using the logical OR operation `por` to achieve the final result.

63..32	31..0	
$00R1_{INT}\ G1_{INT}\ B1_{INT}$	$00R0_{INT}\ G0_{INT}\ B0_{INT}$	mm6
$00R3_{INT}\ G3_{INT}\ B3_{INT}$	$00R2_{INT}\ G2_{INT}\ B2_{INT}$	mm7

Figure 7: Final Results

Using the following code:

```
por            mm6, mm1              // bitwise OR red(0,1) and green(0,1)
por            mm6, mm2              // bitwise OR with blue(0,1)  // result 0,1
por            mm7, mm4              // bitwise OR red(2,3) and green(2,3)
por            mm7, mm5              // bitwise OR with blue(2,3)  // result 2,3
```

4.0 Wrapping It Up

This description provides the basic understanding of a technique of combining alternative instruction streams to improve algorithm performance when using specific data arrangement. Numerous assumptions were made, and variations and improvements are possible that can be adapted to best fit your data structures and performance requirements. A complete and commented function source listing is available in Appendix A.

Appendix A: Example Source

```
//---------------------------------------------------------------------------
// SOAtoAOS - SOA to AOS with SSE and MMX™ Technology ASM.
//
// This routine gets its input as SOA. In each iteration it transposes
// four vertices into AOS format (xyz) and puts the info back in
// m_pScreenVertices to be drawn with D3D, we're using MMX code to do the
// color conversion rather than just calling D3DMAKE.
//
// The code below should replace the following C code.
//
//      for (i=0, j=0; i<numVecs; i++, j+=VectorSize)
//      {
//              psx             = (float*)&(QScreenVertices.sx[i]);
//              psy             = (float*)&(QScreenVertices.sy[i]);
//              psz             = (float*)&(QScreenVertices.sz[i]);
//              prhw    = (float*)&(QScreenVertices.rhw[i]);
//              pdiffuseR       = (float*)&(QScreenVertices.diffuseR[i]);
//              pdiffuseG       = (float*)&(QScreenVertices.diffuseG[i]);
//              pdiffuseB       = (float*)&(QScreenVertices.diffuseB[i]);
//              pspecularR      = (float*)&(QScreenVertices.specularR[i]);
//              pspecularG      = (float*)&(QScreenVertices.specularG[i]);
//              pspecularB      = (float*)&(QScreenVertices.specularB[i]);
//
//              m_pScreenVertices[j].sx         =       psx[0];
//              m_pScreenVertices[j+1].sx       =       psx[1];
//              m_pScreenVertices[j+2].sx       =       psx[2];
//              m_pScreenVertices[j+3].sx       =       psx[3];
//              m_pScreenVertices[j].sy         =       psy[0];
//              m_pScreenVertices[j+1].sy       =       psy[1];
//              m_pScreenVertices[j+2].sy       =       psy[2];
//              m_pScreenVertices[j+3].sy       =       psy[3];
//              m_pScreenVertices[j].sz         =       psz[0];
//              m_pScreenVertices[j+1].sz       =       psz[1];
//              m_pScreenVertices[j+2].sz       =       psz[2];
//              m_pScreenVertices[j+3].sz       =       psz[3];
//              m_pScreenVertices[j].rhw        =       prhw[0];
//              m_pScreenVertices[j+1].rhw      =       prhw[1];
//              m_pScreenVertices[j+2].rhw      =       prhw[2];
//              m_pScreenVertices[j+3].rhw      =       prhw[3];
//              m_pScreenVertices[j].color      =       D3DRGB(pdiffuseR[0], pdiffuseG[0], diffuseB[0]);
//              m_pScreenVertices[j].specular =         D3DRGB(pspecularR[0], pspecularG[0], pspecularB[0]);
//              m_pScreenVertices[j+1].color  =         D3DRGB(pdiffuseR[1], pdiffuseG[1], pdiffuseB[1]);
//              m_pScreenVertices[j+1].specular         =       D3DRGB(pspecularR[1], pspecularG[1],
pspecularB[1]);
//              m_pScreenVertices[j+2].color  =         D3DRGB(pdiffuseR[2], pdiffuseG[2], pdiffuseB[2]);
//              m_pScreenVertices[j+2].specular         =       D3DRGB(pspecularR[2], pspecularG[2],
pspecularB[2]);
//              m_pScreenVertices[j+3].color  =         D3DRGB(pdiffuseR[3], pdiffuseG[3], pdiffuseB[3]);
//              m_pScreenVertices[j+3].specular         =       D3DRGB(pspecularR[3], pspecularG[3],
pspecularB[3]);
//      }
//
// "unpckhps xmm1, xmm2"   works like this (and low is just the opposite):
//
//                      -------------                         -------------
//              XMM1    |x4|x3|x2|x1|                 XMM2    |y4|y3|y2|y1|
//                      -------------                         -------------
//
//                                            -------------
//                                    XMM1    |y4|x4|y3|x3|
//                                            -------------
//
// "shufps xmm1, xmm2, 0x44" which selects bits 0-31 & 32-63
//
//
//
```

```
//                   -------------                       -------------
//             XMM1   |x4|x3|x2|x1|               XMM2   |y4|y3|y2|y1|
//                   -------------                       -------------
//
//                                      -------------
//                               XMM1   |x2|x1|y2|y1|
//                                      -------------
//
// Inputs:     pVertex - pointer to the vertices
//             numVecs - number of vectors
// Outputs:    m_pScreenVertices - the transformed, lit vertices
// Return Value: none
//-------------------------------------------------------------------------------
inline void SOAtoAOS( D3DVERTEX *pVertex, int numVecs )
{
       int i, j;

       __m128 c = _mm_set_ps1(255.0f);
       float *pin, *pout;
       int idx = m_Size*4;
       for (i=0, j=0; i<numVecs; i++, j+=VectorSize)
       {
              pin = ((float*)&(QScreenVertices.sx[i]));
              pout = ((float*)&(m_pScreenVertices[j].sx));

              int offsetFromBaseSOA = j*4;  // Used as an offset to arrive at the correct memory

              _asm
              {
                     mov          eax,idx          // number of vectors per point (e.g. x) into eax
                     mov          edx, pin         // address start of our SOA data

                     movaps       xmm7,[edx];      // x4,x3,x2,x1
                     movaps       xmm4,xmm7        // copy x4,x3,x2,x1
                     add          edx, eax
                     unpcklps     xmm7,[edx]       // y2,x2,y1,x1
                     unpckhps     xmm4,[edx]       // y4,x4,y3,x3 // Data of address [edx] already in cac

                     add          edx, eax
                     movaps       xmm6,[edx]       // z4,z3,z2,z1
                     movaps       xmm2, xmm6       // copy z4,z3,z2,z1
                     add          edx, eax
                     unpcklps     xmm6,[edx]       // rhw2,z2,rhw1,z1
                     unpckhps     xmm2,[edx]       // rhw4,z4,rhw3,z3
                     movaps       xmm3, xmm7       // copy y2,x2,y1,x1

                     shufps       xmm7,xmm6,0x44          // rhw1,z1,y1,x1      Got our 1st result!
                     shufps       xmm3,xmm6,0xEE          // rhw2,z2,y2,x2      Got our 2nd result!

                     movaps       xmm6,xmm4               // copy y4,x4,y3,x3
                     shufps       xmm4,xmm2,0x44          // rhw3,z3,y3,x3      Got our 3rd result!
                     shufps       xmm6,xmm2,0xEE          // rhw4,z4,y4,x4      Got our 4th result!

                     add          edx, eax                // pointer to diffuseR
                     movaps       xmm1, [c]
                     movaps       xmm2, [edx]
                     mulps        xmm2, xmm1              // diffuseR  r4,r3,r2,r1 * 255.0f
                     add          edx, eax
                     movaps       xmm5, [edx]
                     mulps        xmm5, xmm1              // diffuseG  g4,g3,g2,g1 * 255.0f
                     add          edx, eax
                     mulps        xmm1, [edx]             // diffuseB  b4,b3,b2,b1 * 255.0f
                     cvtps2pi     mm0, xmm2               // mm0 = (int)r2,(int)r1
                     shufps       xmm2, xmm2, 0xEE        // r4,r3,r4,r3
                     cvtps2pi     mm3, xmm2               // mm3 = (int)r4,(int)r3
                     cvtps2pi     mm1, xmm5               // mm1 = (int)g2,(int)g1
                     shufps       xmm5, xmm5, 0xEE        // g4,g3,g4,g3
                     cvtps2pi     mm4, xmm5               // mm4 = (int)g4,(int)g3
                     cvtps2pi     mm2, xmm1               // mm2 = (int)b2,(int)b1
                     shufps       xmm1, xmm1, 0xEE        // b4,b3,b4,b3
                     cvtps2pi     mm5, xmm1               // mm5 = (int)b4,(int)b3
```

```
                pslld       mm0, 0x10           // mm0 = r2<<16,r1<<16
                pslld       mm3, 0x10           // mm3 = r4<<16,r3<<16
                pslld       mm1, 0x08           // mm1 = g2<<8,g1<<8
                pslld       mm4, 0x08           // mm4 = g4<<8,g3<<8
                por         mm0, mm1            // bitwise OR red(1,2) and green(1,2)
                por         mm0, mm2            // bitwise OR with blue(1,2)  // result 1,2
                por         mm3, mm4            // bitwise OR red(3,4) and green(3,4)
                por         mm3, mm5            // bitwise OR with blue(3,4)  // result 3,4

                add         edx, eax            // pointer to specularR
                movaps      xmm1, [c]
                movaps      xmm2, [edx]
                mulps       xmm2, xmm1          // specularR  r4,r3,r2,r1 * 255.0f
                add         edx, eax
                movaps      xmm5, [edx]
                mulps       xmm5, xmm1          // specularG  g4,g3,g2,g1 * 255.0f
                add         edx, eax
                mulps       xmm1, [edx]         // specularB  b4,b3,b2,b1 * 255.0f
                cvtps2pi    mm6, xmm2           // mm6 = (int)r2,(int)r1
                shufps      xmm2, xmm2, 0xEE    // r4,r3,r4,r3
                cvtps2pi    mm7, xmm2           // mm7 = (int)r4,(int)r3
                cvtps2pi    mm1, xmm5           // mm1 = (int)g2,(int)g1
                shufps      xmm5, xmm5, 0xEE    // g4,g3,g4,g3
                cvtps2pi    mm4, xmm5           // mm4 = (int)g4,(int)g3
                cvtps2pi    mm2, xmm1           // mm2 = (int)b2,(int)b1
                shufps      xmm1, xmm1, 0xEE    // b4,b3,b4,b3
                cvtps2pi    mm5, xmm1           // mm5 = (int)b4,(int)b3

                pslld       mm6, 0x10           // mm6 = r2<<16,r1<<16
                pslld       mm7, 0x10           // mm7 = r4<<16,r3<<16
                pslld       mm1, 0x08           // mm1 = g2<<8,g1<<8
                pslld       mm4, 0x08           // mm4 = g4<<8,g3<<8
                por         mm6, mm1            // bitwise OR red(1,2) and green(1,2)
                por         mm6, mm2            // bitwise OR with blue(1,2)  // result 1,2
                por         mm7, mm4            // bitwise OR red(3,4) and green(3,4)
                por         mm7, mm5            // bitwise OR with blue(3,4)  // result 3,4

                movq        mm1, mm0            // copy diffuse1, diffuse2
                punpckldq   mm0, mm6            // interleave mm0 = specular1, diffuse1
                punpckhdq   mm1, mm6            // interleave mm1 = specular2, diffuse2

                movq        mm2, mm3            // copy diffuse4, diffuse3
                punpckldq   mm3, mm7            // interleave mm3 = specular3, diffuse3
                punpckhdq   mm2, mm7            // interleave mm1 = specular4, diffuse4

                mov         edx, pout           // Move the base address of the memory
                                                // where SOA vertices will be stored into edx

                //Write final values to new SOA memory segment
                movups      [edx], xmm7         //Write out vertex
                movq        [edx+0x10], mm0     //Write out diffuse and specular color 1
                movups      [edx+0x20], xmm3
                movq        [edx+0x30], mm1     //Write out diffuse and specular color 2
                movups      [edx+0x40], xmm4
                movq        [edx+0x50], mm3     //Write out diffuse and specular color 3
                movups      [edx+0x60], xmm6
                movq        [edx+0x70], mm2     //Write out diffuse and specular color 4
            }
        }
        EMMS    // emms instruction macro
    } // SOAtoAOS
    #endif //
```