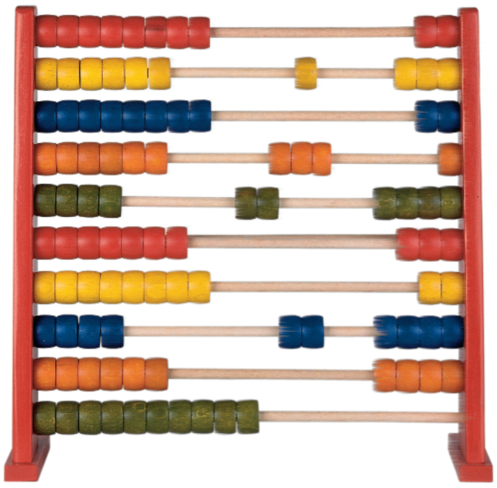
Optimizing Programs for SSE2-capable Processors

Go Faster



Intel's x86 processors have been SIMD-capable since the introduction of the Pentium MMX: SIMD (Single Instruction, Multiple Data) means that the CPU can perform four simultaneous computations. Being familiar with and applying appropriate optimization techniques can provide considerable performance benefits to your programs. BY STEPHAN SIEMEN

Current IA32 family processors are quick enough for scientific computations and other tasks that require large amounts of CPU power, such as multimedia applications. However, this only applied to integer operations until a short while ago; other processor families performed floating point operations a lot more quickly [1].

Intel adopted a new approach with the Pentium MMX processor: SIMD extensions (Single Instruction, Multiple Data) are designed to considerably accelerate integer and floating point operations in comparison to previous architectures, as a multiple computations can be performed using a single instruction.

Instructions with Wide-Ranging Effects

Multimedia applications are where SIMD architectures shine. Instead of processing various data pieces of different types, the processor is required to handle streams of the same data type using the same instruction.

Thus, an SIMD instruction will handle multiple data of the same type at the same time. This contrasts with the SISD technique (Single Instruction, Single Data) used by previous processors.

THE AUTHOR

Stephan Siemen works as a scientist at the University of Essex (UK) where he is involved with creating software for 3D representation of weather systems and teaches computer graphics and programming.
Additional information on this subject is available from his website: http://prswww.essex.ac.uk/stephan/3D/.

Figure 1 juxtaposes SISD and SIMD. Unfortunately, SIMD only processors were not flexible enough to handle SISD tasks quickly and without too heavy an impact on system resources. A combination of SISD and SIMD is ideally suited to handling both simple tasks and mass data quickly and effectively.

There is nothing new about the way SIMD works. Many other processor architectures, such as MIPS and SPARC, for example, also use this technique. Even gaming consoles like the Sony Playstation 2 with its Emotion Engine use SIMD.

Well-known: SIMD in the GPU

SIMD units were introduced to run-of-the-mill PCs years ago – to the video adapter (GPU, Graphics Processing Unit) to be more precise. SIMD is ideally suited to handling graphic data. The GPU often needs to perform operations, such as geometric transformations, on large amounts of the same type of data (vertices in this case).

There are numerous technologies that utilize hardware implementations of SIMD operations. This article uses SIMD to refer to the variant provided by newer x86 processors. The "SIMD Extensions in IA32 Processors" provides details on the common techniques MMX, SSE, SSE2, and 3DNow.

Modern Processor Extensions

To run an SIMD instruction, a program needs to load the data adjacently into a register. MMX (Multimedia Extension) uses the FPU (Floating Point Unit) to do this, using 64 bits of each of the eight FPU registers (each of which provides 80 bits), and assigning the aliases *mm0* through *mm7*. The disadvantage is that the programmer needs to specify

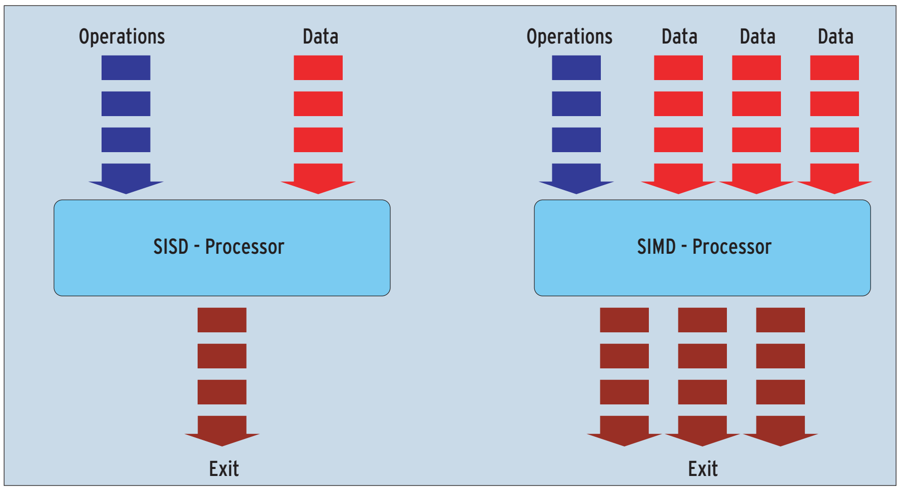


Figure 1: An SISD processor (left) handles a single data field per instruction, whereas an SIMD processor (right) processes multiple units of data simultaneously

whether a code section should perform FPU or MMX operations.

Intel explicitly advises against mixing FPU and MMX computations. For one thing both operation types affect the FPU status flags, and for another mixing ops can severely impact code executions speeds. The Pentium III uses a workaround involving eight new registers. These registers provide 128 bits (16 bytes) of memory and are referenced as *xmm0* through *xmm7*. The Pentium 4 also provides these SSE registers.

SIMD programs need to pay attention to how they organize the data. To allow the processor to efficiently load data into its SSE2 registers, the data need to occupy contiguous memory blocks that start and end on 16 byte borders. This is often referred to as data alignment. If the data is not aligned, the processor will need a lot of time to perform alignment at runtime.

Data Vector Composition

The data processed in a single step by an SIMD command build a vector. The number of elements in the vector depends on the size of the registers. In the case of MMX, each register is 64 bits; this extends to 128 bits for SSE and SSE2; thus an SSE2 vector can contain two 64-bit elements or sixteen 8-bit values, whereas an MMX vector can only handle a single 64 bit value or eight 8-bit elements.

Figure 2 shows the vector composition for MMX, SSE, and SSE2. The class names used by Intel are shown on the right. The figure shows that SSE2 has an advantage over MMX, particularly with regard to integer arithmetic: it can handle twice the amount of numbers at a single step and additionally leaves the x87 FPU free for other tasks. Thus, programs should avoid MMX and opt for SSE2 if at all possible.

There are two variants of every integer vector: signed and unsigned. The C++ class names indicate this by adding an *s* or an *u* after the first letter; thus *I64vec2* becomes *Is64vec2* (signed) or *Iu64vec2* (unsigned). Floating point vectors always require a sign.

SSE2 on Linux

New processor extensions need operating system and compiler support, if applications are to leverage them. Operating system support is vital as the dialog between processes involves storing the state of all registers, and this in turn means that the SSE and SSE2 registers must be known. The newer Linux ker-

Listing 1: C and Assembler

```
// Compute scalar product in C:
float scalarproduct(float x[], float y[])
{
  return x[0]*y[0] + x[1]*y[1] + x[2]*y[2] + x[3]*y[3];
}

// Compute scalar product in Assembler:
float scalarproductAssembler(float x[], float y[])
{
  vector tmp;
  asm(
     "movl   %1,       %%esi;"
     "movl   %2,       %%edi;"
     "movaps (%%esi), %%xmm0;"
     "mulps  (%%edi), %%xmm0;"
     "movaps %%xmm0,  %0;"
    :"=g"(tmp)        /* Output */
    :"r"(x),"r"(y)    /* Input */
  );
  return tmp.f[0] + tmp.f[1] + tmp.f[2] + tmp.f[3];
}
```

Table 1: SIMD-aware x86 Processors

Processor	MMX	SSE	SSE2	3Dnow
Intel Pentium and older Pentium MMX, Pentium II,	3			
Celeron Pentium III, Celeron II	3	3		
Pentium 4	3	3	3	
AMD K6	3			
K6 II/III, Athlon, Duron	3			3
Athlon 4, Athlon XP	3	3		3

IA32 Processor SIMD Extensions

MMX: Intel supplied the first SIMD extension, known as the Multimedia Extension, or MMX for short, in the form of the Pentium MMX way back in 1997. These extensions use new commands to perform two, four, or even eight integer instructions in parallel. Despite a large-scale advertising campaign, it took quite a while for developers to start applying themselves to the new extensions. But MMX had no effect on slower floating point computations.

SSE: When Intel introduced the Pentium III in 1999 it added the Streaming SIMD Extension (SSE) to MMX. Besides 70 new instructions the processor was give eight additional registers to allow it to perform single precision floating point computations using SIMD techniques.

SSE2: The Pentium 4 processor (November 2000) saw the introduction of the Streaming SIMD Extension 2 (SSE2) to Intel processors. This extension accelerates operations with double precision integers and floating points.

3DNow: Intel competitor AMD has developed its own SIMD extensions, referred to as 3DNow. 3DNow adds (single precision) floating point capabilities to MMX, using new commands to do so. The manufacturer has committed itself to supporting this instruction set in future, but seems to be heading in the direction of SSE and SSE2.

Table 2: Compiler SIMD Options

Option	Meaning
ICC 7.1	
-vec_reporto through -vec_report3	Issues a status report on the automatic vectorizer (Default: 1)
-xi, -xM, -xK, -xW	Enables the automatic vectorizer (i = Pentium Pro and Pentium II, M = MMX, K = SSE, W = SSE2)
-axi, -axM, -axK, -axW	Enables the automatic vectorizer, but also creates conventional code for non-SIMD processors
GCC 3	
-mfpmath=Unit	Specifies the floating point arithmetic the program will use; units can be *387*, *sse* or *sse,387*
-mmmx, -msse, -msse2	Allows the programmer to use the integrated SIMD function
(Version 3.3 or later), -m3dnow	[5] in GCC

nels (specifically the 2.4 series) provide the required support: *cat /proc/cpuinfo* shows the CPU capabilities supported by a system, as you can see in Figure 3.

Manual Code Optimization

Developers typically need to negotiate a few obstacles before they can leverage the capabilities provided by a CPU in their own programs. The problem with C and C++ is that neither language specifies standardized SIMD functions. Thus, SSE2-aware code depends to a greater extent on the capabilities provided by the compiler you are using. However, instead of relying on compiler functionality, developers can write performance critical code in Assembler.

MMX, SSE, and SSE2 all have their own Assembler instructions, and it is important to distinguish between scalar and packed instructions. Scalar instructions resemble conventional Assembler instructions, modifying only the least significant element of a vector. In contrast to this, packed instructions simultaneously apply the operation to each element. Figure 4 shows an example.

The Scalar Product of Two Vectors

Listing 1 shows two functions that calculate the scalar product of two vectors.

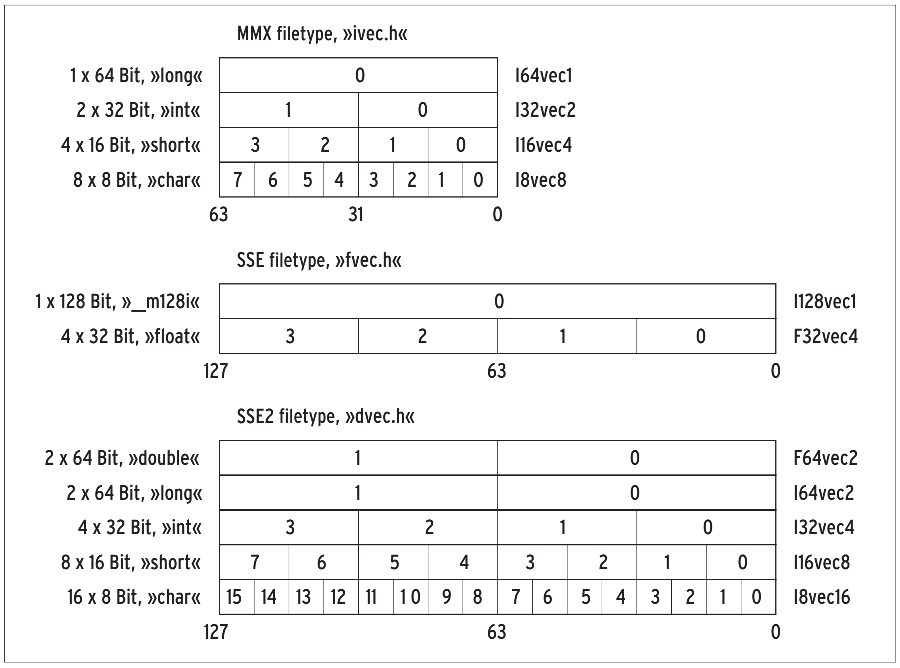


Figure 2: The MMX, SSE, and SSE2 data vectors use different data types. For each vector in Intel's class library, the number, size and type of the elements are shown on the left, and the class name is shown on the right

scalarproduct() is written entirely in C, whereas *scalarproductAssembler()* contains SSE Assembler instructions. The example shows the advantages and disadvantages of SIMD commands: the Assembler variant uses a single multiplication in contrast to four in the C code, but needs more commands to store the result in a variable. Both solutions compute four additions.

Assembler programming also has the disadvantage that the code quickly becomes illegible, and that makes later revisions difficult. However, if you are interested in more detail on the SSE2 Assembler instruction set for the Pentium 4, useful documentation is available from [3]. If you are a bit chary of Assembler, Intel may provide the answer: the processor and compiler manufacturer provides so-called intrinsics [4], compiling SIMD Assembler instructions to C type functions, which can be applied directly to C and C++ code. The programmer does not need to take care of register specifics, as the intrinsics will accept parameters just like normal functions.

Intrinsics Provide Legible Code

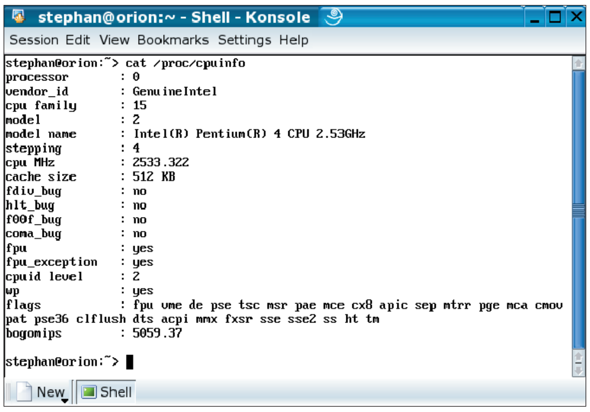
You will need to include one of the following header files to use intrinsics:

- *mmintrin.h* for MMX
- *xmmintrin.h* for SSE
- *emmintrin.h* for SSE2

In the case of SSE, this will add the *__m128 _mm_mul_ps(_ _m128 a, __m128 b)* command, for example. The command calls the Assembler command *MULPS*, which multiplies multiple fields

Listing 2: Intrinsics and C++

```
// Include SIMD Header
#ifdef __INTEL_COMPILER
#  include <fvec.h>   // also contains xmmintrin.h
#else
#  include <xmmintrin.h>
#endif

typedef union{
  __m128 m; float f[4];
} vector;

// Compute scalar product using Intrisics:
float scalarproductIntrinsics(float x[], float y[])
{
  __m128 *vecX = (__m128 *)x;
  __m128 *vecY = (__m128 *)y;
  vector tmp;

  tmp.m = _mm_mul_ps(vecX[0], vecY[0]);

  return tmp.f[0] + tmp.f[1] + tmp.f[2] + tmp.f[3];
}

// Compute scalar product with F32vec4:
float scalarproductSSE(float x[], float y[])
{
  F32vec4 *vecX = (F32vec4 *)x;
  F32vec4 *vecY = (F32vec4 *)y;
  F32vec4 tmp;

  tmp = vecX[0] * vecY[0];

  return tmp[0] + tmp[1] + tmp[2] + tmp[3];
}
```


stephan@orion:~ - Shell - Konsole
Session Edit View Bookmarks Settings Help
New
Shell

```
stephan@orion:~> cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Pentium(R) 4 CPU 2.53GHz
stepping        : 4
cpu MHz         : 2533.322
cache size      : 512 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
bogomips        : 5059.37

stephan@orion:~> 
```

Figure 3: The *cat /proc/cpuinfo* command the processor characteristics supported by your kernel. The flags line indicates that this Linux version is aware of MMX, SSE, and SSE2 (Kernel 2.4.20)

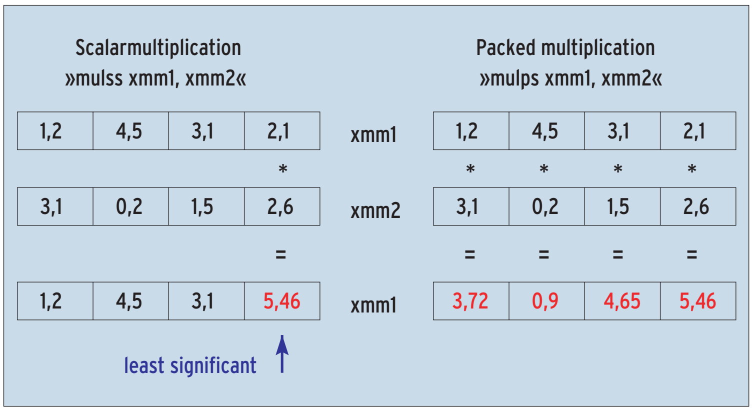


Figure 4: The difference between scalar and packed multiplication: The first multiplies only two operands (blue arrow), the second handles every element in the data vectors

in a single step (*MUL*), that is, it performs a packed operation (*P*, packed). It interprets its operands as single precision floating points (*S*).

Developers can also use GCC to add SIMD commands without resorting to Assembler [5]; the instruction set is enabled by stipulating the appropriate compiler option (see Table 2). Newer GCC versions also support Intel's intrinsics functions.

Automatic Optimization

Assembler, intrinsics and GCC extensions all have one major disadvantage: a program's source code is not easily ported to other systems or even other compilers. But you can typically avoid performing manual SIMD optimization, as many compilers automatically create machine code to leverage the SIMD capabilities of the current processor (see Table 2).

Developers using Intel compilers should use the *-xkW* option to enable the automatic vectorizer, which converts commands to SIMD instructions. However, not every piece of code lend itself to automation. The Intel compiler only vectorizes loops, and only does so provided they do not access elements of the same stream. ICC cannot handle the following loop:

```
for(i=1;i<1000;i++)
   x[i] = x[i-1] + c[i];
```

At each step, the computation has to wait for the results of the previous execution, and this prevents the compiler from performing multiple loops within a single step. The *-vec_report3* option tells the developer how well specific code sections will respond to vectorization.

Union Facilitates Access to SIMD Fields

Contiguous data storage within a vector is vital to SIMD operations. A 128-bit vector is defined as the *__m128* datatype in the SSE Intrinsics headers. The type is always aligned, and its structure is known.

Thus a *union* is well-suited to assigning values to its elements. The following code shows how to do this for four 32-Bit *float* datatypes:

```
union {
  __m128  m128;
  float   fElement[4];
} vector;
```

Intel provides a number of vector classes to save developers the effort of creating their own. Figure 2 shows these types and the appropriate header files.

The program in Listing 2 shows how to compute the scalar product from Listing 1 using intrinsics functions or Intel's SIMD classes (C++). The first function (line 13) uses intrinsics. It first converts the data to *__m128*. The type definition in line 8 facilitates access to the individual elements of the vector with the temporary results *tmp*.

Table 3: Runtimes for Matrix Multiplication

Compiler call	Time (Listing 3, pure C)	Time (Listing 4, Intrinsics)
g++ -O2	0,8 s	–
g++ -O2 -msse -mfpmath=sse	0,82 s	0,09 s
icc -O2	0,05 s	0,01 s
icc -O2 -axW	0,23 s	0,01 s

Listing 3: Multiplication in C

```
[...]
  // Matrix vector multiplication in C

  // Matrix:
  float matrix[4][4]={
    { 2.5, 3.4, 7.9, 1.2 },
    { 1.2, 7.7, 3.7, 0.5 },
    { 3.1, 8.2, 7.1, 3.6 },
    { 7.8, 0.4, 1.2, 5.2 }
  };


  // Vector:
  float vector[4]={ 1.0, 1.0, 1.0, 1.0 };

  // Compute result:
  float result[4]={0};
  int i,j;
  for (i=0;i<4;i++)
  {
    result[i]=0;
    for(j=0;j<4;j++)
    {
      result[i] +=
matrix[i][j] * vector[j];
    }
  }
}
```

The second function (line 25) uses the SSE class *F32vec4* to create a vector for SIMD computations. The vector comprises four elements (*vec4*), and each element contains a single precision floating point number (*F32*, 32 bit float).

The practical thing about these classes is the fact that developers do not need to learn new commands to perform simple operations: C++ allows you to overload operators, thus the multiplication in line 31 can use familiar notation.

The Matrix

The performance boost provide by SIMD extensions is easily demonstrated using a matrix-vector multiplication example. The standard C code for this is shown in Listing 3.

Listing 4 shows the same computation using SSE intrinsics. The intrinsics variant uses *union* constructs to access the *float* array or *__m128* vector. The code uses *_mm_add_ps* to add two vectors, and *_mm_set_ps1* sets all four elements in the vector to a value. Table 3 shows the results: the SSE implementation (right column) is a lot quicker than standard code. Unfortunately, GCC cannot compile SSE code without SSE options, thus it is impossible to assess the performance without the additional optimizations. Looking at the values for the Intel compiler indicates that the normal code runs a lot slower after applying the vectorizer. Automatic optimization has failed here, so manual optimization is well worth while.

The processor manufacturers, Intel [6] and AMD [7], are a useful source for more details, as is the documentation for the GCC [5] and ICC compilers. Intel has published a number of PDF documents on the subject of code optimization with its own compilers.

Conclusion and Lookout

Modern compilers like GCC and ICC provide developers with a number of techniques for leveraging the SIMD extensions of newer x86 processors. These extensions can considerably improve execution speeds, and do not necessarily require in-depth knowledge of Assembler programming on the part of the developer.

To make your programs future-proof, you should investigate the SIMD extensions. In addition to the 32-bit processors, they provide support for Intel and AMD's 64-bit CPU SSE extensions. ■

INFO

[1]	Processor benchmarks: *http://www.spec.org*
[2]	Single Instruction, Multiple Data: *http://www.wikipedia.org/wiki/SIMD*
[3]	Pentium 4 Manual: *http://developer.intel.com/design/Pentium4/manuals/*
[4]	ICC User Guide: *http://www.intel.com/software/products/compilers/techtopics/c_ug_lnx.pdf*
[5]	GCC documentation: *http://gcc.gnu.org/onlinedocs/*
[6]	Intel Homepage: *http://www.intel.com*
[7]	AMD Homepage: *http://www.amd.com*

Listing 4: SSE Intrinsics

```
01 [...]
02   // Matrix vector multiplication using SSE
   Intrinsics code.
03
04   //  Assign matrix memory:
05   union {
06     __m128 m128[4]; // by column
07     float  f[4][4];
08   } matrix;
09
10   // Memory for vector and result:
11   union {
12     __m128 m128;
13     float  f[4];
14   } vector, result;
15
16   // Initialize matrix:
17   matrix.m128[3] = _mm_set_ps( 2.5, 1.2, 3.1, 7.8
   );  // 1. Column
18   matrix.m128[2] = _mm_set_ps( 3.4, 7.7, 8.2, 0.4
   );  // 2. Column
19   matrix.m128[1] = _mm_set_ps( 7.9, 3.7, 7.1, 1.2
   );  // 3. Column
20   matrix.m128[0] = _mm_set_ps( 1.2, 0.5, 3.6, 5.2
   );  // 4. Column
21
22   // Initialize vector:
23   vector.m128    = _mm_set_ps(1.0,1.0,1.0,1.0);
24   result.m128    = _mm_setzero_ps();
25
26   // Compute multiplication:
27   __m128 tmp_multiplier;
28   __m128 tmp_column1;
29   __m128 tmp_column2;
30
31   tmp_multiplier = _mm_set_ps1(vector.f[0]);
32   tmp_column1    = _mm_mul_ps(matrix.m128[0],
   tmp_multiplier);
33
34   tmp_multiplier = _mm_set_ps1(vector.f[1]);
35   tmp_column2    = _mm_mul_ps(matrix.m128[1],
   tmp_multiplier);
36   tmp_column1    = _mm_add_ps(tmp_column1,
   tmp_column2);
37
38   tmp_multiplier = _mm_set_ps1(vector.f[2]);
39   tmp_column2    = _mm_mul_ps(matrix.m128[2],
   tmp_multiplier);
40   tmp_column1    = _mm_add_ps(tmp_column1,
   tmp_column2);
41
42   tmp_multiplier = _mm_set_ps1(vector.f[3]);
43   tmp_column2    = _mm_mul_ps(matrix.m128[3],
   tmp_multiplier);
44   result.m128    = _mm_add_ps(tmp_column1,
   tmp_column2);
45
46   result.m128    = _mm_loadr_ps(result.f);
47 }
```