6,853,473 members and growing! (22,947 online)
Email Password Sign in Join Remember me? Lost your password?
THE CODE PROJECT Your Development Resource
FREE TRIAL DISCOVER SCALABLE PERFORMANCE distributed caching is the solution!

License: The Code Project Open License (CPOL)

VC7.1Win2K, WinXP, Win2003, MFC, VS.NET2003, Dev

Introduction to SSE Programming

By Alex Fr

Posted: 10 Jul 2003
Views: 221,980
Bookmarked: 88 times

An article describes programming floating-point calculations using Streaming SIMD Extensions

Subscribe Now

Download source files - 60 Kb

Introduction

The Intel Streaming SIMD Extensions technology enhance the performance of floating-point operations. Visual Studio .NET 2003 supports a set of SSE Intrinsics which allow the use of SSE instructions directly from C++ code, without writing the Assembly instructions. MSDN SSE topics **[2]** may be confusing for the programmers who are not familiar with the SSE Assembly progamming. However, reading the Intel Software manuals **[1]** together with MSDN gives the opportunity to understand the basics of SSE programming.

SIMD is a single-instruction, multiple-data (SIMD) execution model. Consider the following programming task: computing of the square root of each element in a long floating-point array. The algorithm for this task may be written by such way:

Collapse

```
for each  f in array
    f = sqrt(f)
```

Let's be more specific:

Collapse

```
for each  f in array
{
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

Processor with the Intel SSE support have eight 128-bit registers, each of which may contain 4 single-precision floating-point numbers. SSE is a set of instructions which allow to load the floating-point numbers to 128-bit registers, perform the arithmetic and logical operations with them and write the result back to memory. Using SSE technology, algorithms may be written as:

Collapse

```
for each  4 members in array
{
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```

The C++ programmer writing a program using SSE Intrinsics doesn't care about registers. He has a 128-byte `__m128` type and a set of functions to perform the arithmetic and logical operations. It's up to the C++ compiler to decide which SSE register to use and to make code optimizations. SSE technology may be used when some operation is done with each element of a long floating-point arrays.

SSE Programming Details

Include Files

All SSE instructions and `__m128` data type are defined in `xmmintrin.h` file:

Collapse

```
#include <xmmintrin.h>
```

Since SSE instructions are compiler intrinsics and not functions, there are no lib-files.

Data Alignment

Each `float` array processed by SSE instructions should have 16 byte alignment. A static array is declared using the `__declspec(align(16))` keyword:

Collapse

```
__declspec(align(16)) float m_fArray[ARRAY_SIZE];
```

Dynamic array should be allocated using new `_aligned_malloc` function:

Collapse

```
m_fArray = (float*) _aligned_malloc(ARRAY_SIZE * sizeof(float), 16);
```

Array allocated by the `_aligned_malloc` function is released using the `_aligned_free` function:

Collapse


```
_aligned_free(m_fArray);
```

__m128 Data Type

Variables of this type are used as SSE instructions operands. They should not be accessed directly. Variables of type _m128 are automatically aligned on 16-byte boundaries.

Detection of SSE Support

SSE instructions may be used if they are supported by the processor. The Visual C++ **CPUID** sample **[4]** shows how to detect support of the SSE, MMX and other processor features. It is done using the cpuid Assembly command. See details in this sample and in the Intel Software manuals **[1]**.

SSETest Demo Project

SSETest project is a dialog-based application which makes the following calculation with three float arrays:

⊟ Collapse

```
fResult[i] = sqrt( fSource1[i]*fSource1[i] + fSource2[i]*fSource2[i] ) + 0.5

i = 0, 1, 2 ... ARRAY_SIZE-1
```

ARRAY_SIZE is defined as 30000. Source arrays are filled using sin and cos functions. The Waterfall chart control written by Kris Jearakul **[3]** is used to show the source arrays and the result of calculations. Calculation time (ms) is shown in the dialog. Calculation may be done using one of three possible ways:

- C++ code;
- C++ code with SSE Intrinsics;
- Inline Assembly with SSE instructions.

C++ function:

⊟ Collapse

```
void CSSETestDlg::ComputeArrayCPlusPlus(
          float* pArray1,                   // [in] first source array
          float* pArray2,                   // [in] second source array
          float* pResult,                   // [out] result array
          int nSize)                        // [in] size of all arrays
{
    int i;

    float* pSource1 = pArray1;
    float* pSource2 = pArray2;
    float* pDest = pResult;

    for ( i = 0; i < nSize; i++ )
    {
        *pDest = (float)sqrt((*pSource1) * (*pSource1) + (*pSource2)
                 * (*pSource2)) + 0.5f;

        pSource1++;
        pSource2++;
        pDest++;
    }
}
```

Now let's rewrite this function using the SSE Instrinsics. To find the required SSE Instrinsics I use the following way:

- Find Assembly SSE instruction in Intel Software manuals **[1]**. First I look for this instruction in Volume 1, Chapter 9, and after this find the detailed Description in Volume 2. This description contains also appropriate C++ Intrinsic name.
- Search for SSE Intrinsic name in the MSDN Library.

Some SSE Intrinsics are composite and cannot be found by this way. They should be found directly in the MSDN Library (descriptions are very short but readable). The results of such search may be shown in the following table:

Required Function	Assembly Instruction	SSE Intrinsic
Assign float value to 4 components of 128-bit value	movss + shufps	_mm_set_ps1 (composite)
Multiply 4 float components of 2 128-bit values	mulps	_mm_mul_ps
Add 4 float components of 2 128-bit values	addps	_mm_add_ps
Compute the square root of 4 float components in 128-bit values	sqrtps	_mm_sqrt_ps

C++ function with SSE Intrinsics:

⊟ Collapse

```
void CSSETestDlg::ComputeArrayCPlusPlusSSE(
          float* pArray1,                   // [in] first source array

          float* pArray2,                   // [in] second source array

          float* pResult,                   // [out] result array

          int nSize)                        // [in] size of all arrays

{
    int nLoop = nSize/ 4;

    __m128 m1, m2, m3, m4;

    __m128* pSrc1 = (__m128*) pArray1;
    __m128* pSrc2 = (__m128*) pArray2;
    __m128* pDest = (__m128*) pResult;


    __m128 m0_5 = _mm_set_ps1(0.5f);        // m0_5[0, 1, 2, 3] = 0.5


    for ( int i = 0; i < nLoop; i++ )
    {
        m1 = _mm_mul_ps(*pSrc1, *pSrc1);        // m1 = *pSrc1 * *pSrc1

        m2 = _mm_mul_ps(*pSrc2, *pSrc2);        // m2 = *pSrc2 * *pSrc2

        m3 = _mm_add_ps(m1, m2);                // m3 = m1 + m2

        m4 = _mm_sqrt_ps(m3);                   // m4 = sqrt(m3)

        *pDest = _mm_add_ps(m4, m0_5);          // *pDest = m4 + 0.5


        pSrc1++;
        pSrc2++;
        pDest++;
    }
}
```

This doesn't show the function using inline Assembly. Anyone who is interested may read it in the demo project. Calculation times on my computer:

- C++ code - 26 ms
- C++ with SSE Intrinsics - 9 ms
- Inline Assembly with SSE instructions - 9 ms

Execution time should be estimated in the Release configuration, with compiler optimizations.

SSESample Demo Project

`SSESample` project is a dialog-based application which makes the following calculation with `float` array:

⊟ Collapse

```
fResult[i] = sqrt(fSource[i]*2.8)

i = 0, 1, 2 ... ARRAY_SIZE-1
```

The program also calculates the minimum and maximum values in the result array. ARRAY_SIZE is defined as 100000. Result array is shown in the listbox. Calculation time (ms) for each way is shown in the dialog:

- C++ code - 6 ms on my computer;
- C++ code with SSE Intrinsics - 3 ms;
- Inline Assembly with SSE instructions - 2 ms.

Assembly code performs better because of intensive using of the SSX registers. However, usually C++ code with SSE Intrinsics performs like Assembly code or better, because it is difficult to write an Assembly code which runs faster than optimized code generated by C++ compiler.

C++ function:

⊟ Collapse

```
// Input: m_fInitialArray

// Output: m_fResultArray, m_fMin, m_fMax

void CSSESampleDlg::OnBnClickedButtonCplusplus()
{
    m_fMin = FLT_MAX;
    m_fMax = FLT_MIN;

    int i;

    for ( i = 0; i < ARRAY_SIZE; i++ )
    {
        m_fResultArray[i] = sqrt(m_fInitialArray[i]  * 2.8f);

        if ( m_fResultArray[i] < m_fMin )
            m_fMin = m_fResultArray[i];

        if ( m_fResultArray[i] > m_fMax )
            m_fMax = m_fResultArray[i];
    }
}
```

C++ function with SSE Intrinsics:

⊟ Collapse

```
// Input: m_fInitialArray

// Output: m_fResultArray, m_fMin, m_fMax

void CSSESampleDlg::OnBnClickedButtonSseC()
{
    __m128 coeff = _mm_set_ps1(2.8f);      // coeff[0, 1, 2, 3] = 2.8

    __m128 tmp;

    __m128 min128 = _mm_set_ps1(FLT_MAX);  // min128[0, 1, 2, 3] = FLT_MAX

    __m128 max128 = _mm_set_ps1(FLT_MIN);  // max128[0, 1, 2, 3] = FLT_MIN


    __m128* pSource = (__m128*) m_fInitialArray;
    __m128* pDest = (__m128*) m_fResultArray;

    for ( int i = 0; i < ARRAY_SIZE/4; i++ )
    {
        tmp = _mm_mul_ps(*pSource, coeff);      // tmp = *pSource * coeff

        *pDest = _mm_sqrt_ps(tmp);              // *pDest = sqrt(tmp)


        min128 =  _mm_min_ps(*pDest, min128);
        max128 =  _mm_max_ps(*pDest, max128);

        pSource++;
        pDest++;
    }

    // extract minimum and maximum values from min128 and max128

    union u
    {
        __m128 m;
        float f[4];
    } x;

    x.m = min128;
    m_fMin = min(x.f[0], min(x.f[1], min(x.f[2], x.f[3])));

    x.m = max128;
    m_fMax = max(x.f[0], max(x.f[1], max(x.f[2], x.f[3])));
}
```

Sources

1. Intel Software manuals.
    - Volume 1: Basic Architecture, CHAPTER 9, PROGRAMMING WITH THE STREAMING SIMD EXTENSIONS
    - Volume 2: Instruction Set Reference http://developer.intel.com/design/archives/processors/mmx/index.htm
2. MSDN, Streaming SIMD Extensions (SSE). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclang/html/vcrefstreamingsimdextensions.asp
3. Waterfall chart control written by Kris Jearakul. http://www.codeguru.com/controls/Waterfall.shtml
4. Microsoft Visual C++ CPUID sample. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcsample/html/vcsamcpuiddeterminecpucapabilities.asp
5. Matt Pietrek. Under The Hood. February 1998 issue of Microsoft Systems Journal. http://www.microsoft.com/msj/0298/hood0298.aspx

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

About the Author

Alex Fr

Occupation: Software Developer

Location: Israel

Member

Other popular Algorithms & Recipes articles:

- Symbolic Differentiation
  This article demonstrates differentiating expressions using a stack and displaying the input expression and its derivative.
- Manipulating colors in .NET - Part 1
  Understand and use color models in .NET
- Neural Networks on C#
  The articles describes a C# library for neural network computations, and their application for several problem solving.
- Fast Mathematical Expressions Parser
  Code for a fast math parser library
- State of the Art Expression Evaluation
  A tutorial on how to realize an expression evaluator in CSharp with ANTLR

Article Top

Sign Up to vote for this article

You must Sign In to use this message board.