Optimized Matrix Library for use with the Intel® Pentium® 4 Processor's Streaming SIMD Extensions (SSE2)

Contributed by [Ronen Zohar]

Introduction

On January 2000, Intel published an optimized matrix library (4D single-precision matrix and vector classes) for use with Pentium® III Streaming SIMD (Single Instruction Multiple Data) Extensions, or SSE, in an article in www.gamasutra.com.
Since then, a new processor was introduced – the Intel® Pentium® 4 processor. Its new SSE2 instructions are devoted to double-precision calculations. While a Pentium® III processor's SSE register holds four single-precision elements, the Pentium® 4 processor's SSE2 register holds two double-precision elements.

Using the new SSE2 instructions, Intel has completed an enhanced version of the optimized matrix library. The new library contains similar classes to those of its successor, and additional classes with the same functionality implemented using double-precision arithmetic.

In this article, we describe the new library and its classes and provide some examples of its use. At the end of the article we provide links to the library itself and other helpful resources, such as a free evaluation copy of the Intel® C/C++ Compiler.

The Library

This library includes three types of classes. Each class type has two variants – a single-precision implementation and a double-precision implementation. The single-precision variants are implemented using SSE instructions, and are optimized for both Pentium® III and Pentium® 4 processors. The double-precision variants are implemented using SSE2 instructions, and are optimized for Pentium® 4 processor. The class types are:

- Matrix classes: The variants are the `SPMatrix` and the `DPMatrix` classes. They implement a matrix of 4x4 elements, which can represent 3D transformations in the homogenous space.
- 4-elements vector classes: The variants are the `SPVector` and the `DPVector` classes. They represent 4D vectors or 3D vectors in the homogeneous space.
- 3-elements vector classes: The variants are the `SPVector3` and the `DPVector3` classes. They represent pure 3D vectors in space.

Since all the classes are written using SSE/SSE2 instructions, all instances must be allocated in a 16-bytes aligned memory address.
For static-allocated instances, the Intel® C/C++ Compiler handles the alignment.

However, for run-time allocated instances, to the developer must ensure proper alignment.

The Classes

The SPMatrix and DPMatrix classes

The `SPMatrix` class is a 4x4 matrix of `float`s (single precision floating point numbers), while the `DPMatrix` class is a 4x4 matrix of `double`s (double precision floating point numbers).

```
class SPMatrix {
    union {
        struct {
            __m128 _L1, _L2, _L3, _L4;
        };
        struct {
            float  _11, _12, _13, _14;
            float  _21, _22, _23, _24;
            float  _31, _32, _33, _34;
            float   _41, _42, _43, _44;
        };
    };
    // ...
};
```

```
class DPMatrix {
    union {
        struct {
            __m128d _L1, _L2, _L3, _L4;
        };
        struct {
            __m128d _L1a, _L1b,
                    _L2a, _L2b,
                    _L3a, _L3b,
                    _L4a, _L4b;
        };
        struct {
            double  _11, _12, _13, _14;
            double  _21, _22, _23, _24;
            double  _31, _32, _33, _34;
            double  _41, _42, _43, _44;
        };
    };
    // ...
};
```

Each class contains 16 elements (`_11` to `_44`). The elements are placed in four lines (`_L1` to `_L4`). For the `SPMatrix` class, each line is represented as one SSE variable. For the `DPMatrix` class, each line is divided into two halves and each half is represented as one

SSE2 variable (`_L1a` to `_L4b`), since an SSE2 register can hold only two double-precision elements.
Data elements may be referenced by their row and column: `Mat._12` is also `Mat[0][1]` or `Mat(0,1)`.

The SPVector and DPVector classes

The `SPVector` class is a vector of four `float`s, while the `DPVector` class is a vector of four `double`s.

```
class SPVector {
    union {
        __m128 vec;
        struct {
            float x,y,z,w;
        };
    };
    // ...
};
```

```
class DPVector {
    union {
        __m256d vec;
        struct {
            __m128d xy,zw;
        };
        struct {
            double  x,y,z,w;
        };
    };
    // ...
};
```

Each class contains the `x`, `y`, `z` and `w` elements of the vector. The `SPVector` class represents all the elements as one SIMD variable. For the `DPVector` class, the `x` and `y` elements are stored together in the `xy` variable while the `z` and `w` elements are stored together in the `zw` variable.
Data elements may be referenced by their place: `Vec.z` is also `Vec[2]` or `Vec(2)`.

The SPVector3 and DPVector3 classes

The `SPVector3` and `DPVector3` classes are variants of the `SPVector` and `DPVectror` classes respectively. These classes do not have a `w` element, so they hold "pure" 3D vectors. However, for alignment and for other reasons, the `w` element, which is not used, is replaced with a spacer.

Constructors & Operators

Operators on SPMatrix and DPMatrix:

A * B	matrices multiplication
A ± B	matrices addition/subtraction
± A	matrix unary plus/minus
A * s	matrix multiplication with scalar
A *= B	matrix multiplied by matrix
A *= s	matrix multiplied by scalar
A ±= B	matrix added/subtracted by matrix
	matrix transpose
	matrix inverse
	matrix determinant
	matrix minimal/maximal element

SPMatrix and DPMatrix Constructors:

	Identity matrix
	Zero matrix
	Rotation matrices (around the X axis, Y axis and Z axis)
	Translation matrices
	Scaling matrices

Operators on SPVector, SPVector3, DPVector and DPVector3:

v * M	vector multiplication with matrix
v * s	vector multiplication with scalar
v * w	vectors dot (inner) product
v % w	vectors cross product (in 3D)
v | w	vectors elements product
v ± w	vectors addition/subtraction
v *= M	vector multiplied by matrix
v *= s	vector multiplied by scalar
v |= w	vector elements multiplied by vector elements
v ±= w	vector added/subtracted by vector
± v	vector unary plus/minus
~ v	normalized vector
	vector length
	vector normalization

Header Files

All the `SP*` classes are declared in the `SPMatrix.h` header file.
All the `DP*` classes are declared in the `DPMatrix.h` header file.
Both the `SP*` and `DP*` classes are included within the `Matrices.h` header file.

- There are two different ways to use the library.
  To use only the `SP*` classes or only the `DP*` classes, include the appropriate header file – `SPMatrix.h` or `DPMatrix.h`.
- To use both the `SP*` classes and the `DP*` classes at the same time, do not simply include the two header files together, since some functions share names. Instead, include the header file `Matrices.h`. This header file includes the other two header files, but removes ambiguous functions (i.e., constructors that are separated only by the return value). It also declares conversion functions from the `SP*` classes to the corresponding `DP*` classes, and vice-versa.

Examples

Calculation of an exponent

The first example, in `Exponent.cpp`, demonstrates calculation of a matrix exponent.

An exponent of a real number can be calculated using a Tylor Series.
Similarly, an exponent of matrix $\mathbb{M}$ is defined:

$$e^{\mathbb{M}} = \sum_{n=0}^{\infty} \frac{1}{n!} \mathbb{M}^n \qquad \left[\mathbb{M}^0 = \mathbb{I}\mathrm{d}\right]$$

Calculation of square root

The second example, in `Sqrt.cpp`, demonstrates an iterative method of calculating matrix square root.

An iterative numerical analysis method called the Newton-Raphson method can be used to calculate a square root of a positive number. However, it can also be used to calculate a square root of a matrix.
One iteration for approximating a square root of $\mathbb{M}$ is:

$$\mathbb{X}_{n+1} = \frac{1}{2}\left(\mathbb{X}_n + \mathbb{M} \cdot \mathbb{X}_n^{-1}\right) \qquad \left[\mathbb{X}_0 = \mathbb{I}\mathrm{d}\right]$$

Usually, this method is not recommended for the calculation of matrix square root, since even if the root is reversible (which is not always true), the method is highly sensitive to the rounding errors. However, for demonstration proposes, we may ignore the limitations.

Results

Below we compare four versions for calculating the series from both examples:

A. Using single-precision scalar code (using Microsoft* `D3DXMATRIX` class).
B. Using the `SPMatrix` class.
C. Using double-precision scalar code.
D. Using the `DPMatrix` class.

The following table shows the average time an iteration takes for each version:

Version	Average time for **Exponent**		Average time for **Square Root**	
Single-precision scalar code	398	393	681	654
`SPMatrix` code	170	122	322	254
Double-precision scalar code	352		584	
`DPMatrix` code	216		402	

Note that using the `DPMatrix` instead of scalar code gives an improvement of up to x1.6, and using the `SPMatrix` instead of scalar code gives greater improvement of x2.1-x3.2 !

Links

- Download the library <<link to license.htm attached to this CR>>
- Download a free evaluation copy of Intel® C/C++ Compiler from http://developer.intel.com/software/products/compilers/c60/c60eval.htm
- Read the original article introducing the optimized matrix library for the Pentium® III Streaming SIMD (Single Instruction Multiple Data) Extensions, or SSE, published 31 January, 2000 in www.gamasutra.com at http://www.gamasutra.com/features/20000131/barad_01.htm.