Benefits of SIMD Programming

Introduction

Ticker Tape is a tech demo that encourages developers to implement more complex behavior in particle systems. The demo uses a number of techniques to improve performance, including multithreading and optimizations for Intel® Streaming SIMD Extensions (SSE). An overview of the article can be found at www.intel.com/software/tickertape and the demo can be downloaded from there as well. This article focuses on the performance gained in the Ticker Tape demo by the introduction of SSE instructions. However, before we get to those results we will go over an introduction to SSE programming, how to lay out your data so that it is most conducive to SSE, and finally, walk through an example of calculating dot products using SSE.

Background

SSE is an instruction set designed to make use of a type of architecture called SIMD (Single Instruction Multiple Data). In this type of design it is possible to perform a calculation on multiple pieces of data simultaneously, thus achieving a form of data parallelism. This is sometimes also called vector processing. For example, let’s say we have two arrays of numbers which we wish to multiply together.

```c
float a[nElements], b[nElements], c[nElements];
for (unsigned int i = 0; i < nElements; i++) {
    c[i] = a[i] * b[i];
}
```

Listing 1 - Scalar method of multiply two arrays, one element per iteration

Normally, as in Listing 1, there is a loop that iterates through all of the elements, multiplying each element together, and storing the results. What if instead of doing one multiplication per loop iteration we could do more? Let’s create a function called `MultiplyFourElements()`.

```c
float a[nElements], b[nElements], c[nElements];

void MultiplyFourElements(float *a, float *b, float *c) {
    c[0] = a[0] * b[0];
    c[1] = a[1] * b[1];
}

for (unsigned int i = 0; i < nElements; i += 4) {
    MultiplyFourElements(&a[i], &b[i], &c[i]);
}
```

Listing 2 - Scalar method of multiply two arrays, four elements per iteration

In Listing 2 we have created a function that can process four elements at once. This lets us iterate through the loop four times fewer. In terms of efficiency we have gained very little since the same math
calculations are being done; this is where SSE instructions come in. Instead of a function that calculates the four multiplications in series there is a single instruction that can calculate four multiplications simultaneously. SSE uses special registers on the CPU that are 128 bits wide. These registers can hold any type of data that will fit in 128 bits, like two double precision numbers, four single precision, or 16 bytes. There are two ways to program for SSE, either by coding the assembly instructions directly or by using intrinsics functions. Ticker Tape, and this article, focus exclusively on use of the intrinsic functions. Using intrinsics instead of assembly instructions is more straightforward and similar to standard C/C++ programming. Using intrinsics also allows a compiler to better optimize the code, the results of this will be seen later.

```c
// Assembly SSE Instruction
/// xmm0 and xmm1 are actual registers, not variables
mulps xmm0,xmm1

// Intrinsic SSE Instruction
__m128 a, b, c;
c = _mm_mul_ps(a, b);
```

Listing 3 - Assembly and Intrinsic SSE Instructions

The type __m128 is a definition for a 16 byte aligned variable that maps to one of the SIMD registers. The program first needs to explicitly load data into the SIMD registers which is done with the _mm_load_ps() intrinsic (not shown in Listing 3). The _mm_mul_ps() is the instruction that actually performs the calculation. When we have the result, we can store it into an output array using the _mm_store_ps() intrinsic.

```c
#include <xmmintrin.h>

float a[nElements], b[nElements], c[nElements];
__m128 A, B, C;

//... load data in a, b

for (unsigned int i = 0; i < nElements; i += 4) {
    A = _mm_load_ps(&a[i]);
    B = _mm_load_ps(&b[i]);
    C = _mm_mul_ps(A, B);
    _mm_store_ps(&c[i], C);
}
```

Listing 4 - SIMD method of multiplying two arrays

Graphically, the difference in operations is shown in Figure 1:
Data Layout

The bottleneck in many applications is not the calculation portion of the algorithm, but reading and writing data. In the previous example, three of the four instructions dealt with loading and storing the data. If the data you want to access is stored in different areas, loading it into the cache and then into the SIMD registers will eat up a significant amount of time. For example, if the arrays from the previous example were actually member variables in a class of which we had an array, it would be difficult and time consuming to load the data into the registers. The elements would have to be loaded into the SIMD registers element by element.
Listing 5 - Class with SIMD unfriendly data layout (Array of Structures)

In Listing 5 the data is loaded in an incorrect manner. It attempts to load contiguous memory into the SIMD register but this will be the wrong data. It is not impossible to load data into the registers but it does involve rearranging the data into the proper format which takes a number of steps. Regardless of the actual processing costs involved to rearranging the data, there is simply more data to move and potentially many cache lines to load into memory. However, even with all of that it can still be very beneficial to use SSE instructions, especially if there are lots of calculations being performed with the same data. The alternative layout is to arrange similar pieces of data sequentially so that it can be efficiently loaded and saved. The other benefit to arranging data like this is that when the size of SIMD registers increases it is much easier to change the code. You would simply load eight floats at a time instead of four. The above example, reworked with a more efficient data layout, would look like this:
```c
class foo {
    float *a;
    float *b;
    ... other data ... 
};

foo::foo(unsigned int nElements) {
    a = (float *)_mm_malloc(nElements * sizeof(float), 16);
    b = (float *)_mm_malloc(nElements * sizeof(float), 16);
}

void bar() {
    foo fooVariable(nElements);
    float c[nElements];
    __m128 A, B, C;

    // Non_SIMD Method
    for (unsigned int i = 0; i < nElements; i++) {
        c[i] = fooVariable.a[i] * fooVariable.b[i];
    }

    // SIMD Method
    for (unsigned int i = 0; i < nElements; i += 4) {
        A = _mm_load_ps(&fooVariable.a[i]);
        B = _mm_load_ps(&fooVariable.b[i]);
        C = _mm_mul_ps(A, B);
        _mm_store_ps(&c[i], C);
    }
}
```

Listing 6 - Class with SIMD friendly data layout (Structure of Arrays)

**Structure of Arrays**

```
X  X  X  X  ...  
Y  Y  Y  Y  ...  
Z  Z  Z  Z  ...  
W  W  W  W  ...  
```

**Array of Structures**

```
X  Y  Z  W  X  Y  Z  W  ...  
```

Figure 2 - SoA and AoS memory layout

SIMD optimizations in Ticker Tape
Optimizing Ticker Tape to make use of SIMD consisted of two major parts, rearranging the data to be more SIMD friendly and then rewriting a portion of the code to use SSE instructions. When we profiled the demo we found that the majority of time was being spent in a function called `Newton()`. This function calculates how various forces affect the particles. In the original architecture, it was being called once for every particle and would do the calculations inside four times, once per corner. A high level outline looked something like this:

```cpp
class RigidBody
{
    void Newton();
    // ... Bunch more functions ...

    D3DXVECTOR3 Position;
    D3DXVECTOR3 Rotation;

    // ... Lots of other data ...
};

void RigidBody::Newton()
{
    for (unsigned int = 0; i < 4; i++)
    {
        // ... some math goes on ...

        // Velocity_Ground
        D3DXVECTOR3 Vel_Ang_CM_global;
        D3DXVec3TransformCoord(&Vel_Ang_CM_global,
            &Velocity_Angular_CM, &this->LocalGlobal);
        D3DXVec3Cross(&tmp, &Radial_Vec, &(Vel_Ang_CM_global));
        Velocity_Ground = (this->Velocity_Linear_CM) + tmp;

        // ... more math ...
    }

    return;
}
```

Listing 7 - Original Ticker Tape Implementation

Each particle was represented as a distinct object and contained the methods to manipulate it. Elsewhere, there was a loop that iterated through each particle calling its member methods. The first in the transformation to SSE was to create another `Newton()` function called `NewtonArray()`. This did the exact same thing as the original, but acted on an entire array of particles instead of just one. As part of this step, the class `RigidBody` was made to contain entire arrays of data so that the program went from having an array of `RigidBody` objects to having one `RigidBody` comprised of arrays of data. Based on some testing that was done on different data layouts, the data was further broken down into its component parts (Listing 8).
After the data was reorganized, `NewtonArray()` was changed to use SSE instructions. The basic premise to making the transition was based on the fact that there was a nested loop structure. Every variable that used the inner loop index would be loaded in the SIMD registers normally whereas every variable that used the outer loop index would be loaded in duplicate in the registers. Essentially, we eliminated the inner loop (Listing 9). The instruction `_mm_set1_ps()` is similar to the `_mm_load_ps()` instruction except that it loads a single floating point value and replicates it into all four sections of the register.

The benefits of doing this optimization were immediately apparent, even after just a small portion of the function was transformed. After all the optimizations were completed, timings on the `NewtonArray()` function were made. Using the Microsoft compiler (MSVCC) with Visual Studio 2008 there was a 1.8x improvement in speed. Using the Intel® C++ Compiler (ICC) on the original code gave almost as good a benefit thanks to its automatic vectorizing capabilities. Using ICC on the handwritten SSE netted a huge improvement of 4.5x.

### Table 1 - Execution Time of Function Newton, in milliseconds

<table>
<thead>
<tr>
<th>Compiler:</th>
<th>MSVCC</th>
<th>MSVCC + SSE</th>
<th>ICC</th>
<th>ICC + SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (ms):</td>
<td>16.3</td>
<td>8.9</td>
<td>9.9</td>
<td>3.6</td>
</tr>
<tr>
<td>Improvement:</td>
<td>1.0x</td>
<td>1.8x</td>
<td>1.6x</td>
<td>4.5x</td>
</tr>
</tbody>
</table>

**Dot Product Example**
This section contains an example of using SSE instructions to compute a dot product (actually, four dot products) along with a walkthrough of the algorithm. The variables are all named xmm# because the eight SSE registers are xmm0 through xmm7. There is no requirement to name the variables in this manner nor are you limited to only having eight variables.

```c
// Dot Product
// Computes dot products on two arrays of vectors, 1 at a time
for (unsigned int i = 0; i < nElements; i++) {
    result[i] = v1[i].x * v2[i].x +
                v1[i].y * v2[i].y +
                v1[i].z * v2[i].z;
}

// Dot Product
// Computes dot products on two arrays of vectors, 4 at a time
for (unsigned int i = 0; i < nElements; i += 4) {
    xmm0 = _mm_load_ps(X1 + i);  // Load data into SIMD registers
    xmm1 = _mm_load_ps(Y1 + i);
    xmm2 = _mm_load_ps(Z1 + i);
    xmm3 = _mm_load_ps(X2 + i);
    xmm4 = _mm_load_ps(Y2 + i);
    xmm5 = _mm_load_ps(Z2 + i);
    xmm6 = _mm_mul_ps(xmm0, xmm3);  // Multiply x's together
    xmm7 = _mm_mul_ps(xmm1, xmm4);  // Multiply y's together
    xmm8 = _mm_mul_ps(xmm2, xmm5);  // Multiply z's together
    xmm0 = _mm_add_ps(xmm6, xmm7);  // Add all the values together
    xmm7 = _mm_add_ps(xmm0, xmm8);
    _mm_store_ps(result + i, xmm7);  // Save the results
}
```

Listing 10 - Example of SSE version of dot product

The first step in the algorithm is to load all of the data into SIMD registers which is done using the _mm_load_ps() instruction. As an argument, this takes the address from which to load the data. The ps suffix on all of the instructions indicate that they are the single precision versions. Many instructions have multiple versions, for example there is also a _mm_load_pd() for loading two double precision numbers. It is also important to note that the data must be 16 byte aligned. If it is not then a different instruction, _mm_loadu_ps(), must be used but it does not offer the same performance benefit.

After the data is loaded, the multiplication is done using the _mm_mul_ps() instruction. This multiplies corresponding elements of the two registers together.
After the multiplications are all done the results need to be added together. This is done with the 
_mm_add_ps() function. It works in the same manner as the multiplication version except it does
addition. Finally, the results are written out to a memory location using _mm_store_ps(). Again, the
address being written to needs to be 16 byte aligned. There is a code sample at
www.intel.com/software/tickertape that contains this function along with a simple test framework to
compare the performance of it against different methods.

Future Work

For Ticker Tape there are some clear next steps for SSE work. One would be to vectorize the
code further instead of just one function. Also, the SSE could be made to work with the upcoming Intel®
Advanced Vector Extensions (Intel® AVX) instruction set. Another area that would be interesting to
pursue is using the Intel® C++ Compiler to automatically vectorize the code. There are code patterns and
constructs that can be used in order to help the compiler do the vectorization. The Ticker Tape code
could be changed to use these so that SSE is still used but behind the scenes.

Further Reading / Resources

- "Ticker Tape: A Scalable 3D Particle System with Wind and Air Resistance" at
  http://www.intelsoftwaregraphics.com/?lid=NrY2bP+21e0=&siteid=cqMoF5H/37o=
- Ticker Tape source code and binaries at www.intel.com/software/tickertape
- SSE code samples and method comparisons at http://software.intel.com/file/25119
- "Creating a Particle System with Streaming SIMD Extensions" at http://software.intel.com/en-
  us/articles/creating-a-particle-system-with-streaming-simd-extensions/

About the Author

Quentin Froemke is a software engineer with Intel's Visual Computing Software Division where he works
with the game industry to more effectively utilize multi-core and other Intel technologies.