3BA26 : Concurrent Systems I: SIMD II

David Gregg

Department of Computer Science
Trinity College Dublin

April 22, 2009
"Newton's method, also called the Newton-Raphson method, is a root-finding algorithm that uses the first few terms of the Taylor series of a function f(x) in the vicinity of a suspected root. Newton's method is sometimes also known as Newton's iteration, although in this work the latter term is reserved to the application of Newton's method for computing square roots."

The Newton-Rhapson Reciprocal

One Iteration of the Newton Rhapson-Method is all we need to increase precision for the reciprocal intrinsic. Note that this is still faster than using a divide!

\[ rcp_{nr}(x) = 2 \times \frac{1}{x} - \left( \frac{1}{x} \times (x \times \frac{1}{x}) \right) \]
```c
__m128 rcp_nr(const __m128 &a)
{
    const __m128 r = _mm_rcp_ps(a);
    return _mm_sub_ps(_mm_add_ps(r, r), _mm_mul_ps(_mm_mul_ps(r, a), r));
}
```
NR Reciprocal SQRT

\[ \frac{1}{2} \times \text{rsqrtps}(x) \times (3 - x \times \text{rsqrtps}(x) \times \text{rsqrtps}(x)) \]

_m128 rsqrt_nr(const __m128 &a)
{
    const __m128 half = _mm_set1_ps(0.5f);
    const __m128 three = _mm_set1_ps(3.0f);

    const __m128 r = _mm_rsqrt_ps(a);
    return _mm_mul_ps(_mm_mul_ps(half, r),
                      _mm_sub_ps(three,
                                  _mm_mul_ps(_mm_mul_ps(_mm_mul_ps(a, r), Ra0))));
}
SSE provides a variety of bitwise operations that can be used to manipulate individual bits within a 128 bit vector value.

For example, it is possible to do a bitwise and of all 128 bits in two vector words:

```c
__m128 a = _mm_set_ps(0.0f, 1.0f, 2.0f, 3.0f);
__m128 b = _mm_set_ps(3.0f, 2.0f, 1.0f, 0.0f);
__m128 c;

c = _mm_and_ps(a, mask);
```
Some Bitwise Operations

__m128 _mm_and_ps(__m128 a, __m128 b)  \[ r = a \text{ and } b \]
__m128 _mm_or_ps(__m128 a, __m128 b)  \[ r = a \text{ or } b \]
__m128 _mm_andnot_ps(__m128 a, __m128 b)  \[ r = \text{not } a \text{ and } b \]
__m128 _mm_xor_ps(__m128 a, __m128 b)  \[ r = a \text{ xor } b \]
Comparisons

SSE allows us to compare 4 values at a time against 4 other values.

\[
a = \begin{bmatrix} 99 & 88 & 77 & 66 \end{bmatrix}
\]

\[
b = \begin{bmatrix} 88 & 77 & 66 & 55 \end{bmatrix}
\]

\[a > b\] is obviously true in this case, but what if...

\[
b = \begin{bmatrix} 88 & 99 & 66 & 55 \end{bmatrix}
\]
Some Comparison Operations

\[
\begin{align*}
\_\_m128 \_mm\_cmpeq\_ps(\_\_m128 \ a, \_\_m128 \ b) &= = \\
\_\_m128 \_mm\_cmplt\_ps(\_\_m128 \ a, \_\_m128 \ b) &= < \\
\_\_m128 \_mm\_cmple\_ps(\_\_m128 \ a, \_\_m128 \ b) &= \leq \\
\_\_m128 \_mm\_cmpgt\_ps(\_\_m128 \ a, \_\_m128 \ b) &= > \\
\_\_m128 \_mm\_cmpge\_ps(\_\_m128 \ a, \_\_m128 \ b) &= \geq \\
\_\_m128 \_mm\_cmpneq\_ps(\_\_m128 \ a, \_\_m128 \ b) &= \neq \\
\_\_m128 \_mm\_cmpnlt\_ps(\_\_m128 \ a, \_\_m128 \ b) &= \lt! \\
\_\_m128 \_mm\_cmpnle\_ps(\_\_m128 \ a, \_\_m128 \ b) &= \leq! \\
\_\_m128 \_mm\_cmpngt\_ps(\_\_m128 \ a, \_\_m128 \ b) &= \gt! \\
\_\_m128 \_mm\_cmpnge\_ps(\_\_m128 \ a, \_\_m128 \ b) &= \geq!
\end{align*}
\]
Comparison instructions return a bitmask indicating which of the constituent parts of the SSE register passed and which failed. In the previously listed instructions we have four results returned, packed into an __m128. So that we can write code such as:

```c
if( a > b ) do_a(); else do_b();
```

SSE provides the ability to convert the __m128 mask into a 4 bit integer using the _mm_movemask_ps intrinsic.

<table>
<thead>
<tr>
<th>Bitmask</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111</td>
<td>Comparison True for all 4 floats</td>
</tr>
<tr>
<td>0000</td>
<td>Comparison False for all 4 floats</td>
</tr>
<tr>
<td>1100</td>
<td>Comparison True for first two floats</td>
</tr>
<tr>
<td>1010</td>
<td>Comparison True for first and third floats</td>
</tr>
</tbody>
</table>
__m128 a = _mm_set1_ps(0.0f);
__m128 b = _mm_set1_ps(1.0f);

__m128 r = _mm_cmpgt_ps(a, b);

if( _mm_movemask_ps(r) == 0xF)
    printf("a is greater than b\n");
else if ( _mm_movemask_ps(r) == 0)
    printf("a is NOT greater than b");
else
    printf("mixed result");
More direct masking

Comparison instructions return a bitmask indicating which of the constituent parts of the SSE register passed and which failed.

It is possible to use this mask directly rather than using the _mm_movemask_ps intrinsic.

The result of a comparison is four values, one for each of the numbers compared. If the comparison is false then the result is zero. If the comparison is true, then the result is minus one.

Note that minus one in two’s complement is represented by all the bits being set to one. So the result of a comparison is all the bits in the result being set to zero, or all the bits set to one.
More direct masking

So that we can write code such as:

```c
__m128 a = _mm_set_ps(0.0f, 1.0f, 2.0f, 3.0f);
__m128 b = _mm_set_ps(3.0f, 2.0f, 1.0f, 0.0f);
__m128 c, mask;

mask = _mm_cmpgt_ps(a, b);
c = _mm_and_ps(a, mask);
```

The variable `c` now contains those numbers from `a` that are greater than the corresponding numbers in `b`.

Problem: Write a function `max(a, b)` which returns a vector containing the maximum values of `a` and `b`.
SSE also supports max and min operations directly:

```c
__m128 a = _mm_set_ps(0.0f, 1.0f, 2.0f, 3.0f);
__m128 b = _mm_set_ps(3.0f, 2.0f, 1.0f, 0.0f);
__m128 max, min;

max = _mm_max_ps(a, b);
min = _mm_min_ps(a, b);
```

Using these operations it is possible to build quite complex bigger functions.
To properly harness the power of SIMD we may need to re-order our data so that it can be more efficiently loaded into registers. For example, we have four separate data streams of floats, each loaded from a separate file. We must add up the floats in each file yielding 4 separate results.

```c
float total1, total2, total3, total4;
float *data1, *data2, *data3, *data4;

for(i=0; i<SIZE; i++){
    total1 += data1[i];
    total2 += data2[i];
    total3 += data3[i];
    total4 += data4[i];
}
```

An obvious solution might be to do the following to use SIMD instructions to speed up the loop.

```c
__m128 totals;
float *data1, *data2, *data3, *data4;

for(i=0; i<SIZE; i++){
    __m128 v = _mm_setr_ps(data1[i], data2[i], data3[i], data4[i]);
    totals = _mm_add_ps(totals, v);
}
```
Reordered Data

\[
data1 = \begin{bmatrix}
d1_1 & d1_2 & d1_3 & d1_4 & \ldots \\
\end{bmatrix}
\]

\[
data2 = \begin{bmatrix}
d2_1 & d2_2 & d2_3 & d2_4 & \ldots \\
\end{bmatrix}
\]

\[
data3 = \begin{bmatrix}
d3_1 & d3_2 & d3_3 & d3_4 & \ldots \\
\end{bmatrix}
\]

\[
data4 = \begin{bmatrix}
d4_1 & d4_2 & d4_3 & d4_4 & \ldots \\
\end{bmatrix}
\]

Reordered Data

\[
data = \begin{bmatrix}
d1_1 & d2_1 & d3_1 & d4_1 & d1_2 & d2_2 & d3_2 & d4_2 & \ldots \\
\end{bmatrix}
\]
Reordered Solution

Be reordering our data we can get it into a SIMD register more efficiently.

```c
__m128 totals;
float *data;

for(i=0; i<SIZE; i+=4){
    __m128 v = _mm_load_ps(&data[i]);
    totals = _mm_add_ps(totals, v);
}
```
SSE provides a shuffle instruction that can be used to reorder the four values within a vector word.

It actually operates on two separate vector words and takes two 32-bit values from each of them.

```c
__m128 a = _mm_set_ps(0.0, 1.0, 2.0, 3.0);
__m128 b = _mm_set_ps(4.0, 5.0, 6.0, 7.0);
__m128 c;

c = _mm_shuffle_ps(a, b, _MM_SHUFFLE(1, 0, 3, 2);
/* c now has value {2.0, 3.0, 4.0, 5.0}*/
Horizontal Operations

The operations we have used so far are horizontal operations such as:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\]

\[+
\]

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\]

\[=
\]

\[
\begin{array}{cccc}
2 & 4 & 6 & 8 \\
\end{array}
\]

Occasionally we may find it useful to be able to perform an operation across all constituent values in a single vector.

\[a = \begin{array}{cccc}
2 & 4 & 6 & 8 \\
\end{array}\]

\[\text{horizontal add}(a) = 20;\]
Intel’s SSE doesn’t give us exactly those type of horizontal operations, but it does provide operations (as of SSE3) which operate horizontally in the following manner

\[ c = \text{__mm_hadd_ps}(a, b) \]

\[
\begin{align*}
a &= \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \end{bmatrix} \\
b &= \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix}
\end{align*}
\]

After the operation, \( c \) contains the following

\[
\begin{align*}
a &= \begin{bmatrix} b_0 + b_1 & b_2 + b_3 & a_0 + a_1 & a_2 + a_3 \end{bmatrix}
\end{align*}
\]

\text{__mm_hsub_ps} operates in a similar fashion.
Scalar Operations

The horizontal operations we have used so far have all been *Packed Operations*. These operations operate on all constituents of the vector. *Scalar Operations* operate only on the least significant portion of the vector.

c = a + b

\[
a = \begin{array}{c|c|c|c}
a_0 & a_1 & a_2 & a_3 \\
\end{array}
\]

\[
b = \begin{array}{c|c|c|c}
b_0 & b_1 & b_2 & b_3 \\
\end{array}
\]

After the operation, \( c \) contains the following

\[
c = \begin{array}{c|c|c|c}
a_0 & a_1 & a_2 & a_3 + b_3 \\
\end{array}
\]
By convention SSE Packed Instructions are suffixed with _ps. Scalar Instructions are suffixed with _ss. Many of the instructions we’ve encountered so far have a scalar version. Check the xmmmintrin.h header file for a full list.