Using Streaming SIMD Extensions 3 in Algorithms with Complex Arithmetic

Version 1.0

01/04
Using Streaming SIMD Extensions 3 in Algorithms with Complex Arithmetic

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel’s Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Intel, Pentium, VTune Performance Analyzer are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries.

The Pentium® 4 processors may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 processor supporting HT Technology and a Hyper-Threading Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See http://www.intel.com/info/hyperthreading/ for more information including details on which processors support HT Technology.

† Other brands and names may be claimed as the property of others.

Copyright © Intel Corporation 2002-2004
# Table of Contents

1. Introduction ................................................................................................................................. 5
   1.1. SIMD Instructions .................................................................................................................. 5
2. Complex Arithmetic ......................................................................................................................... 6
   2.2. Complex Multiply ..................................................................................................................... 7
      2.2.1. Using Streaming SIMD Extensions 3 .................................................................................. 7
   2.3. Complex Divide ....................................................................................................................... 7
      2.3.1. Using Streaming SIMD Extensions 3 .................................................................................. 8
3. Mandelbrot Set ............................................................................................................................... 8
   3.1. Evaluating Mandelbrot Set ....................................................................................................... 9
      3.1.1. Coding Strategy for Mandelbrot Set ................................................................................... 9
   3.2. Single Precision with SSE3 ..................................................................................................... 10
   3.3. Double Precision with SSE3 ................................................................................................... 11
      3.3.1. Pipelining ......................................................................................................................... 11
      3.3.2. Algorithm ......................................................................................................................... 11
   3.4. Comparing Results ............................................................................................................... 12
      3.4.1. Controlling x87 Precision ................................................................................................. 13
      3.4.2. Controlling Reassociation ............................................................................................... 13
   3.5. Comparing Performance ......................................................................................................... 13
      3.5.1. Reviewing x87 Performance ............................................................................................. 14
      3.5.2. Reviewing SSE3 Performance ........................................................................................ 15
4. Conclusion ..................................................................................................................................... 17
5. Single Precision SSE3 Code Example ........................................................................................... 18
6. Double Precision SSE3 Code Example ......................................................................................... 20
7. Configuration .................................................................................................................................. 22
   7.1. Test System ........................................................................................................................... 22
   7.2. Build Environment ............................................................................................................... 22
   7.3. Using VTune™ Performance Analyzer to Estimate Retirement Bandwidth ................. 23
8. Appendix ....................................................................................................................................... 24
   8.1. Double-Precision Complex Multiply .................................................................................. 24
   8.2. Double-Precision Complex Division ............................................................................... 24
Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Revision History</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Public Release</td>
<td></td>
</tr>
</tbody>
</table>

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1 Introduction

Single Instruction Multiple Data (SIMD) programming enables software to achieve higher performance by processing more data with fewer instructions. The Streaming SIMD Extensions 3 (SSE3) is a new extension in the IA-32 Intel® Architecture that enhances the SIMD capabilities of Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2). SSE3 enables software to accelerate data processing in specific areas, for example, complex arithmetic and video decoding. This application note focuses on the use of SSE3 in algorithms that perform arithmetic using complex numbers.

1.1 SIMD Instructions

A large part of the SSE and SSE2 instructions that deal with arithmetic operations have focused on processing parallel data elements in a vertical computing model. In the vertical computing model, data elements in the source and destination operands are operated uniformly by the same operation. (For more information, see “Data Arrangement” in Chapter 5 of IA-32 Intel® Architecture Software Optimization Reference Manual). Using SIMD instructions targeted to the vertical computation model implies that

- One must pre-formulate the data arrangement to suit the vertical computation model, or
- It may be necessary to introduce additional data-swizzling operations into the algorithm to compensate for any deviations from the homogeneous operations in the vertical computation model.

SSE3 offers several new instructions that allow programmers to accelerate SIMD processing when data elements must be operated in a non-homogeneous or asymmetric manner. Two examples of the new instructions that can handle asymmetric or non-homogeneous operations on multiple data elements are described below.

ADDSUBPS is a new instruction that performs asymmetric arithmetic operations. It adds two pairs of single-precision floating-point values and subtracts two pairs of floating-point values. This type of asymmetric arithmetic operations are similar to basic operations used in complex algebra.

<table>
<thead>
<tr>
<th>ADDSUBPS xmm1,xmm2/m128</th>
</tr>
</thead>
<tbody>
<tr>
<td>xmm1</td>
</tr>
<tr>
<td>xmm2/ m128</td>
</tr>
<tr>
<td>xmm1</td>
</tr>
<tr>
<td>X4</td>
</tr>
<tr>
<td>X3</td>
</tr>
<tr>
<td>X2</td>
</tr>
<tr>
<td>X1</td>
</tr>
<tr>
<td>Y4</td>
</tr>
<tr>
<td>Y3</td>
</tr>
<tr>
<td>Y2</td>
</tr>
<tr>
<td>Y1</td>
</tr>
<tr>
<td>X4 + Y4</td>
</tr>
<tr>
<td>X3 - Y3</td>
</tr>
<tr>
<td>X2 + Y2</td>
</tr>
<tr>
<td>X1 - Y1</td>
</tr>
</tbody>
</table>
HADDPD is a new instruction that performs arithmetic operations on data elements that deviate from the vertical computing model. In the case of HADDPD, the arithmetic operation applies horizontally to two adjacent data elements in the operands. Horizontal addition or subtraction can be useful for evaluating dot products or cross products.

<table>
<thead>
<tr>
<th>xmm1</th>
<th>xmm2/m128</th>
</tr>
</thead>
<tbody>
<tr>
<td>X2</td>
<td>X1</td>
</tr>
<tr>
<td>Y2</td>
<td>Y1</td>
</tr>
<tr>
<td>Y2 + Y1</td>
<td>X2 + X1</td>
</tr>
</tbody>
</table>

2 Complex Arithmetic

A complex number $z$ consists of a real part and an imaginary part, i.e. $z = x + y \cdot i$. On a complex plane, the real axis extends horizontally from negative infinity to positive infinity and the imaginary axis extends vertically from negative infinity to positive infinity. The complex number $z$ can be identified as a vector in a two-dimensional plane, extending from the origin to the point represented by $z$.

From an algebraic point of view, both the set of real (R) and complex (C) numbers are fields. A field is a mathematical structure where all four arithmetic operations (+, -, *, /) are allowed. But the set of complex numbers possesses a property not shared by the set of real numbers: it is algebraically closed. This simply means that an algebraic equation is always solvable in C. For example, the algebraic equation of degree " $z^2 + 1 = 0$" is not solvable for $z$ defined in R but it is solvable in C, and the two solutions are $i$ and $-i$. In general, implementing vectorized arithmetic operations using SIMD instructions must pay attention to the direction of data flow between data elements of SIMD registers. Some arithmetic operations, e.g. complex addition or subtraction, are naturally suited to a vertical computation model. This is because, in the rectangular representation of a complex number, $z = x + y \cdot i$, the data flow of the real (imaginary) parts between the output and the input operands is unchanged for each data element.

For multiplication and division of complex numbers, the real part of the output of such arithmetic operations involves both the real parts and the imaginary parts of the input operands. Thus, using SSE or SSE2 alone with the vertical computation model will require additional data permutations between intermediate results stored in the SIMD registers. SSE3 provides several new instructions that can perform asymmetric arithmetic operations and suits complex data stored in natural order.

There is more than one coding strategy to implement complex arithmetic:

The first is a “brute-force” approach. Decompose the problem in fine-grain steps to process each data element serially, and separate arithmetic operations on the real and imaginary parts sequentially. This can be done by programming in C language or using x87 assembly.

The second is to identify inherent parallelism in the algorithm, arrange data layout and the processing between data elements to suit the common SIMD computation model. For example, one can separate arithmetic operations on the
real and imaginary parts sequentially, and identify any orthogonal parallelism in the problem, such as in the outer loop. Then it may be possible to unroll the outer loop twice or four times to take advantage of the number of slots available in SIMD instructions. This approach requires pre-arranging data to fit the vertical computation model, and may be feasible for some class of problems dealing with complex arithmetic.

A third approach is similar to the second, but takes advantage of SSE3 by arranging the SIMD instructions to handle arithmetic operations on the real and imaginary part in parallel. SSE3 provides more flexibility for programmers to choose the optimal data layout and use SSE3 in conjunction with SSE or SSE2 to do more work with fewer instruction and/or data permutations.

### 2.2 Complex Multiply

To multiply two complex numbers \((a + bi)\) and \((c + di)\), find the result \((x + yi)\) where:

- \(x = a*c - b*d\)
- \(y = a*d + b*c\)

A complex multiply requires four scalar multiplies, one add, and one subtract to calculate one result. A simple implementation would require separate instructions for each operation.

#### 2.2.1 Using Streaming SIMD Extensions 3

When implementing complex multiplication using SSE3 it is possible to reduce the number of instructions needed by using SSE3 in conjunction with SSE2 or SSE, depending on the desired numeric precision. Using SSE3 in single-precision complex arithmetic, one can evaluate a pair of complex multiplies \((a_k + b_ki) * (c_k + d_ki)\), with four complex numbers, and compute two complex results \((x_k + y_ki)\) in one iteration.

This algorithm assumes a natural layout where the real and imaginary parts are stored together.

```c
struct complexf { float r, i; }
```

Thus, the multipliers are stored together \((a_0 \ b_0 \ a_1 \ b_1)\), as are the multiplicands \((c_0 \ d_0 \ c_1 \ d_1)\).

The algorithm uses the new instructions MOVSLDUP and MOVSHDUP to setup two copies of each of the multipliers. It then does eight multiplies using two MULPS instructions and completes the operation with two adds and two subtracts using a single ADDSUBPS instruction. The following is an instruction sequence used to evaluate two complex multiply and store two complex numbers in the destination:

```
movsldup xmm0, [eax]    ; multiplier real  (a1,   a1,   a0,   a0)
movaps xmm1, [ebx]      ; multiplicand     (d1,   c1,   d0,   c0)
mulps xmm0, xmm1       ; temp1            (a1d1, a1c1, a0d0, a0c0)
shufps xmm1, xmm1, 0xB1 ; shuf multiplicand(c1,   d1,   c0,   d0)
movshdup xmm2, [eax]    ; multiplier imag  (b1,   b1,   b0,   b0)
mulps xmm2, xmm1       ; temp2            (b1c1, b1d1, b0c0, b0d0)
addsubps xmm0, xmm2    ; b1c1+a1d1, a1c1-b1d1, a0d0+b0d0, a0c0-b0c0
movaps [edx], xmm0     ; store the results (y1,x1,y0,x0)
```

This shows that SSE3 can be used to calculate two complex multiply results in relatively few instructions.

### 2.3 Complex Divide

To divide the complex number \((a + bi)\) by \((c + di)\), find the result \((x + yi)\) where:

\[ t = c^2 + d^2 \]
\[ x = (a^2 + b^2) / t \]
\[ y = (b^2 - a^2) / t \]

Complex division requires six multiplies, two adds, one subtract, and two divides to calculate one result.

### 2.3.1 Using Streaming SIMD Extensions 3

The algorithm for dividing complex numbers assumes a similar data layout as the multiplication algorithm. The algorithm first calculates the numerator values using a similar method to the complex multiply:

```
movshdup xmm0, [eax] ; dividend imag (b1, b1, b0, b0)
movaps xmm1, [ebx] ; divisor (d1, c1, d0, c0)
mulps xmm0, xmm1 ; temp (b1d1, b1c1, b0d0, b0c0)
shufps xmm1, xmm1, 0xB1 ; reorder (c1, d1, c0, d0)
movsldup xmm2, [eax] ; dividend real (a1, a1, a0, a0)
mulps xmm2, xmm1 ; temp (a1c1, a1d1, a0c0, a0d0)
addsubps xmm0, xmm2 ; a1c1+b1d1, b1c1-a1d1, a0c0+b0d0, b0c0-a0d0
```

The algorithm then calculates the denominator values using MULPS and ADDPS:

```
mulps xmm1, xmm1 ; c1c1, d1d1, c0c0, d0d0
movaps xmm2, xmm1 ; c1c1, d1d1, c0c0, d0d0
shufps xmm2, xmm2, 0xB1 ; d1d1, c1c1, d0d0, c0c0
addps xmm2, xmm1 ; c1c1+d1d1, c1c1+d1d1, c0c0+d0d0, c0c0+d0d0
```

Final DIVPS calculates four division results and the values are reordered before storing (the values were calculated in this order to enable the use of ADDSUBPS):

```
divps xmm0, xmm2
shufps xmm0, xmm0, 0xB1 ; (b1c1-a1d1)/(c1c1+d1d1),
                         ; (a1c1+b1d1)/(c1c1+d1d1),
                         ; (b0c0-a0d0)/(c0c0+d0d0),
                         ; (a0c0+b0d0)/(c0c0+d0d0)
movaps [edx], xmm0 ; store the results (y1,x1,y0,x0)
```

This shows that SSE3 can be used to calculate two complex divide results in relatively few instructions.

### 3 Mandelbrot Set

“The Mandelbrot set is the most complex object in mathematics, its admirers like to say. An eternity would not be enough to see it all... but here is a paradox: to send a full description over a transmission line requires just a few dozen characters of code” (Gleick)

“The Mandelbrot set is a collection of points in the complex plane where every number is either in the set or out of the set. To test a point, its value is squared and then the original value is added. This process is repeated. If the total goes to infinity, the point is not in the set; otherwise, the point is in the set.” (Gleick)
“When rendering an image of the Mandelbrot set using a grid of complex numbers black is often used to represent numbers that are in the set. Sometimes white is used to represent numbers that are not in the set, but often colors are used to represent numbers that are not in the set with different colors used to represent the number of iterations needed before the point is identified as not in the set. The resulting images contain intense color near the boundaries, as the boundary cases require the most iterations.” (Gleick)

3.1 Evaluating Mandelbrot Set

The Mandelbrot set can generate images that are interesting to scientists as well as to lay people. It is an example of a class of simple algorithms that can generate a wide variety of imagery with rich textures. One popular fractal image can be generated using a simple complex algebraic equation translated into pseudo-code shown below, with \( Z \) and \( C \) being complex numbers, \( \text{LengthSQ} \) is a floating point number, \( \text{Count} \) is an integer and \( C \) is initialized to a complex number based on the point being considered.

\[
\begin{align*}
Z &= 0; \quad \text{Count} = 0; \\
\text{do} &\quad \{ \\
&\quad \quad Z = Z \times Z + C; \\
&\quad \quad \text{LengthSQ} = Z.\text{real}^2 + Z.\text{imag}^2 \\
&\quad \quad \text{count}++ \\
\text{while} &\quad ((\text{LengthSQ} < 4.0) \land (\text{Count} < 255));
\end{align*}
\]

The routine will return 0 if the do loop exited because of count exceeding the loop limit of 255; otherwise it returns the count. This count is then assigned as the color value of a pixel in the two-dimensional plane, with 0 generating black while values approaching 255 generating very bright colors.

3.1.1 Coding Strategy for Mandelbrot Set

The Mandelbrot set problem allows more than one coding strategy to perform complex arithmetic operations. The pseudo code shown above can be implemented in high-level language such as C, low-level assembly using x87 instructions, intrinsics or inline assembly using SSE3.

With high-level language implementations, performance is limited to the degree a compiler can generate efficient code for the target processor. Hand-tuned x87 code by an experienced programmer with microarchitectural knowledge of the target processor can achieve higher performance than compiler-generated code. With optimized hand-tuned x87 code, its performance will be limited by the scalar nature of the x87 instructions.

The coding examples used in the following sections take advantage of the parallel nature of the spatial geometry to process two pixels in parallel using SSE3 in conjunction with SSE or SSE2 instructions. These examples demonstrate the flexibility of using natural data arrangement (array of structures) and SIMD processing of the vertical computation model.

These examples demonstrate a technique of transforming spatial parallelism into instruction level SIMD processing. Complex arithmetic operations are decomposed into intermediate stages of scalar arithmetic operations, with the intermediate results of each stage stored in a XMM register; each slot of the XMM register stores one or more scalar components associated with adjacent pixels. The single precision implementation uses the above complex multiply and thus conserves registers (XMM5 is unused). The double precision implementation uses a different approach based on algebraic simplifications from squaring one complex number vs. multiplying two different complex numbers. This second approach requires more registers. In both of these examples, one can expect that when complex arithmetics is implemented using SSE3, roughly half as many loops are needed to complete the same amount of work as compared to x87 implementation.
3.2 Single Precision with SSE3

To implement the single-precision Mandelbrot set using SSE3, it is more efficient to load data and compute complex multiply on two adjacent pixels at once.

TestPixels:

```assembly
    // max iteration counter
    add    ecx,1

    // partial results
    movslldup xmm0,xmm1          // c c a a -- real pairs
    movshdup xmm2,xmm1          // d d b b -- imaginary pairs
    mulps  xmm0,xmm1          // dc cc ba aa - ri ii ri rr
    shufps xmm1,xmm1,0xB1     // c d a b -- reordered 'Z' value
    mulps  xmm2,xmm1          // dc dd ba bb - ri rr ri rr

    // finish the complex multiply
    addsubps xmm0,xmm2          // dc+cd cc-dd ba+ba aa-bb - d' c' b' a'
```

In addition to the complex multiply of \(Z \times Z\), the constant ‘C’ is added to \(Z\) with ADDPS and the LengthSQ is calculated using MULPS and the new instruction HADDPS.

```assembly
    // add constant C with result back in xmm1
    movaps xmm1,xmm7          // xmm7 has C1i C1r C0i C0r
    addps xmm1,xmm0          // Z = Z + C

    // length sq
    movaps xmm3,xmm1          // copy of 'Z'
    mulps  xmm3,xmm1          // dd cc bb aa
    haddps xmm3,xmm3          // dd+cc bb+aa dd+cc bb+aa
```

Loop control is accomplished with CMPLTPS setting mask bits based on a comparison of LengthSQ and 4.0. The mask bits are then added as a negative counter using PADDD and sent to EAX using MOVMSKPS. While the count is per pixel the loop terminates when both points exit. Neighboring points tend to have similar values except near boundaries so the extra computations needed because of exiting after both points satisfy the exit condition should be less than the computations saved by computing two points in parallel.

```assembly
    //----- Loop Control -----  
    cmpltps xmm3,xmm6          // L1<4.0 L0<4.0 L1<4.0 L0<4.0
    paddd xmm4,xmm3          // add the counters
    movmskps eax,xmm3          // extract sign bits
    cmp   eax,0
    je    Done
    cmp   ecx,255
    jl    TestPixels
```
3.3 Double Precision with SSE3

The double-precision Mandelbrot implementation employs a slightly different coding technique. This technique combines the benefits of SIMD instructions in general (complete the same amount of work with fewer instructions) and an algebraic simplification specific to the Mandelbrot problem (reduce the number of required arithmetic operations to complete the same amount of work). The algebraic simplification is based on the decomposition of the square of a complex number into scalar terms.

In the typical complex multiply the new real value is the difference of the products of real terms minus the products of the imaginary terms. In this Mandelbrot equation, the new real value is the difference of the real part squared and the imaginary part squared. The difference is calculated with the new horizontal instruction HSUBPD.

3.3.1 Pipelining

The first implementation used the following general algorithm for the TestPixels loop:

- Increment the Iteration Counter
- temp = z.real^2 - z.imag^2 + Cr
- z.imag = z.real*z.imag + z.real*z.imag + Ci
- z.real = temp
- lengthsq = z.real^2 + z.imag^2
- loop exit test

Performance tests showed a long data dependency chain with the computation of lengthsq and the loop exit test. Each instruction depended on the results of the previous, which limited out-of-order execution. To improve performance the computation of “temp = z.real^2 - z.imag^2 + Cr” was pipelined. The computation for the first iteration was moved to setup while the computation for the later iterations was inserted in the middle of the test for loop exit. During loop iteration ‘N’ temp is computed for iteration ‘N+1’. The performance improved noticeably with the pipelined optimization, so the optimization was included in the final code.

3.3.2 Algorithm

The double-precision algorithm begins with the computation of temp for the first iteration. This is done in setup code as part of pipelining. As xmm2 and xmm3 start at zero, the subtract instruction (HSUBPD xmm2,xmm3) was not needed in the setup code. The ADDPD instruction is used to add Cr to the temp value.

```
// pipelining setup
addpd xmm2,xmm4 // add Cr ---> r1 r0
```

The TestPixels loop starts with incrementing the iteration counter. The code for the computation of temp is not at the top of the loop as it has been moved.

```
//----- Test Pixels -----
TestPixels:
// max iteration counter
inc ecx
// temp = z.real^2 - z.imag^2 + Cr (pipelined) - moved
```

In the typical complex multiply the new imaginary value is the sum of each point’s real term multiplied by the other point’s imaginary term. This approach multiplies the real term by the imaginary term and doubles it (xmm0 has r1 r0 and xmm1 has i1 and i0).

```
// z.imag = z.real*z.imag + z.real*z.imag + Ci
```
Using Streaming SIMD Extensions 3 in Algorithms with Complex Arithmetic

mulpd xmm1,xmm0 // result=r1*i1 r0*i0
addpd xmm1,xmm1 // *2
addpd xmm1,xmm5 // add Ci ---> i1 i0

Next the temp value is assigned to z.real.

// z.real
movapd xmm0,xmm2 // r1 r0

The LengthSQ is calculated with two MULPD and the new instruction HADDPD. Before calculating LengthSQ the values must be re-arranged with the points in separate registers.

// re-arrange to prepare for lengthsq
movapd xmm3,xmm2 // r1 r0
shufpd xmm3,xmm1,3 // srcH,destH or i1 r1
shufpd xmm2,xmm1,0 // srcL,destL or i0 r0
// lengthsq = z.real^2 + z.imag^2
mulpd xmm3,xmm3 // I1^2 r1^2 - need to have r at low end
// as hsubpd does low-high
mulpd xmm2,xmm2 // I0^2 r0^2
movapd xmm7,xmm2
haddpd xmm7,xmm3 // L1 L0

The loop control (CMPLTPD nad MOVMSKPD) includes the pipelined computation of temp (HSUBPD and ADDPD) and uses general-purpose registers to count iterations. The general-purpose registers are used because this approach requires more SSE2 registers than the single precision algorithm above.

// test
cmpltpd xmm7,xmm6 // 4 4
// temp = z.real^2 - z.imag^2 + Cr (pipelining)
hsubpd xmm2,xmm3 // (A) xmm2=10^2 r0^2 r0=I=1^2 r1^2 ---> diff1 diff0
addpd xmm2,xmm4 // (B) add Cr ---> r1 r0
// continue the test
movmskpd edx,xmm7
cmp edx, 0
je Done
mov ebx,edx
and edx,1
shr ebx,1
add esi,edx
add edi,ebx
cmp ecx,255
jl TestPixels

3.4 Comparing Results

Each of the hand-optimized implementation of Mandelbrot set code (x87 and SSE3) represents a compute-bound workload. Single-precision and double-precision routines were coded separately for x87 and SSE3. These routines plus corresponding C version are programmed into an executable to run under an operating system. Using the Intel® VTune™ Performance Analyzer, all potential coding pitfalls (See Chapter 2 of IA-32 Intel Architecture Optimization Reference Manual) were minimized. In addition, each implementation is multithreaded using data-decomposition threading model so that each Mandelbrot Map (4096 x4096) can be calculated using two threads if
running on a Pentium® 4 processor supporting Hyper-Threading Technology. The configuration of the test system is described in Section 7. (See disclaimer on page 2.)

The executable can be run with Hyper-Threading Technology disabled to compare the performance benefits of SSE3 implementation over hand-optimized x87 implementation. When running in a configuration with Hyper-Threading Technology enabled, one can compare the performance benefits of SSE3 when the processor's execution resources are utilized more heavily.

To compare the performance of different implementations, it is important to make sure the same amount of computation is being compared against each other. For this reason, single-precision should be compared against another single-precision implementation, similarly for double-precision implementations. Cross comparisons between a single-precision implementation and a double-precision implementation may not be a valid performance comparison.

Due to the architectural differences between the x87 FPU and floating-point operations in the XMM registers, it is well-known that numerical results from some arithmetic operations may differ slightly when different register sets are used. This can affect the Mandelbrot map color generation problem, because the computation at each pixel is determined by the number of iterations to reach a predefined threshold condition. To ensure the number of iteration at each pixel remain the same between the SSE3 implementation and the equivalent x87 implementation, we used two techniques that are described next.

### 3.4.1 Controlling x87 Precision

The x87 instructions normally operate with 80-bits of precision internally while SSE/SSE2/SSE3 operate in either single precision with 32-bits or in double-precision with 64-bits of precision. The following code was inserted to control the x87 operation such that it generates the same color map as the SSE/SSE2/SSE3 implementations.

To set single precision:

```c
_control87( PC_24 | RC_NEAR, MCW_PC | MCW_RC );
```

To set double precision:

```c
_control87( PC_53 | RC_NEAR, MCW_PC | MCW_RC );
```

In the above code the number of bits refers to the number of bits in the mantissa. A 32-bit single-precision floating point uses 24-bits for the mantissa while a 64-bit double-precision floating point uses 53-bits for the mantissa.

### 3.4.2 Controlling Reassociation

When comparing results of floating point operations it is important to remember that floating-point, as an imprecise representation, does not support reassociation. Thus the expression $a + b + c$ will possibly yield different results if $a + b$ is added first than when $b + c$ is added first. With assembly language it is easy to control the order of expressions as the coder has complete control. With a higher level language such as ‘C’ the compiler may use reassociation to optimize for performance. The Intel® C++ Compiler when optimizing for speed in a release build does this optimization. The compiler supports an option ‘/Op’ which improves conformance with IEEE arithmetic. When checking for correctness by comparing the results of the ‘C’ and assembly language algorithms the program was built with the ‘/Op’ option.

### 3.5 Comparing Performance

After verifying the correctness of each of these implementations, we evaluated the performance of each implementation, and compared the performance benefits of SSE3 as well as Hyper-Threading Technology.
The program reports the number of milliseconds needed to calculate the colors of a 4096x4096 surface. The system was rebooted between test runs. A test run consisted of a single test (such as single precisions x87) repeated five times. The result was the average of the five runs.

The following comparisons express performance improvement as follows:

\[ \text{Performance Improvement} = \frac{\text{Old Value} - \text{New Value}}{\text{New Value}} \]

The values reported in the tables below and the horizontal axes are all in milliseconds.

### 3.5.1 Reviewing x87 Performance

**C code Results:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>11765.8</td>
</tr>
<tr>
<td>Double Precision</td>
<td>12062.2</td>
</tr>
</tbody>
</table>

**x87 Results:**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision HT disabled</td>
<td>9059.0</td>
</tr>
<tr>
<td>Single precision HT enabled</td>
<td>5993.8</td>
</tr>
<tr>
<td>Double precision HT disabled</td>
<td>9572.0</td>
</tr>
<tr>
<td>Double precision HT enabled</td>
<td>6409.2</td>
</tr>
</tbody>
</table>

The hand-optimized x87 implementation is practically free of potential coding pitfalls (more so for the single-precision implementation). So it is no surprise that hand-optimized x87 code performs significantly better than compiler-optimized C code. The x87 code without Hyper-Threading Technology betters the ‘C’ code by 30% and 26% for single and double precision. Nevertheless, the performance of hand-optimized x87 code is limited by the scalar nature of x87 instructions and the latency of the x87 execution units.

The double-precision version is slightly slower than the single-precision version because the double-precision implementation has a small amount of residual store-to-load forwarding issue. The impact of the store-to-load forwarding in the double-precision x87 code is estimated to be about 5%, which may be improved further with more code tuning effort.
Hyper-Threading Technology boosts performance of the compute-bound x87 code by an additional 51% and 49% for single-precision and double-precision. Several factors are key contributing factors to the performance gains of Hyper-Threading Technology:

- Long dependency chain: The performance of this workload is limited by the chain of dependent code in the inner loop, which exposes the aggregate latency of floating-point add, floating-point multiply, and loading floating-point data.
- Resource utilization: One can estimate the percentage of retirement bandwidth consumed by this workload relative to the peak retirement bandwidth of the processor. For the optimized x87 implementations executing with only one thread, approximately ~21% of the peak retirement bandwidth are consumed.
- Minimizing coding pitfalls: The lack of coding pitfall in each thread enables the processor’s out-of-order engine to schedule micro-ops originated from either thread using shared execution resource more effectively.

The combination of the presence of long dependency chains, the available retirement bandwidth, and the lack of coding pitfall impeding the processor’s out-of-order execution engine are likely the key factors for utilizing Hyper-Threading Technology successfully.

### 3.5.2 Reviewing SSE3 Performance

**SSE3 Results:**

<table>
<thead>
<tr>
<th></th>
<th>Single precision HT disabled</th>
<th>Single precision HT enabled</th>
<th>Double precision HT disabled</th>
<th>Double precision HT enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5524.8</td>
<td>4459.4</td>
<td>7328.2</td>
<td>5100.0</td>
</tr>
</tbody>
</table>

**SSE3 Gains More Performance Than x87**

The hand-optimized SSE3 implementations are also implemented to minimize potential coding pitfalls. Consequently, SSE3 speeds up the performance of Mandelbrot map calculation by varying amount ranging from 64% to 31% in single-threaded environment, depending on data type (See chart below). The performance gains of using SSE3 demonstrates the effectiveness of the SIMD coding strategy: using fewer instructions to do the same amount of work. For example, comparing SSE3 versus x87 code, the number of inner loops in the SSE3 implementations are approximately half of that in the x87 code.
Hyper-Threading Technology Adds Even More Performance

It is worth noting that Hyper-Threading Technology can further boost performance by additional 24% and 44% (See chart below) after SSE3 code significantly increased the processors’s floating-point throughput. In these SSE3 implementations, the three factors discussed previously as contributing to the effectiveness of Hyper-Threading Technology also apply.

Performance Benefits of SSE3 with Shared Execution Resources

Just as the performance gains of Hyper-Threading Technology is robust across different instruction sets, the performance gains of SSE3 over x87 is also robust across single-threaded and multithreaded environments. This is shown in the chart below.

Two Performance Enhancing Technology

The Intel® Pentium® 4 Processor with Hyper-Threading Technology and 90 nm technology includes two performance enhancing technology: SSE3 and Hyper-Threading Technology. It is of interest to note the cumulative performance gains due to these two performance enhancements over hand-optimized x87 code can reach 103% and 88% respectively for single-precision and double-precision Mandelbrot map calculation.
Using Streaming SIMD Extensions 3 in Algorithms with Complex Arithmetic

4 Conclusion

Streaming SIMD Extensions 3 (SSE3) can accelerate complex arithmetics significantly. Using SSE3 along with SSE and SSE2, applications have flexibility to implement complex arithmetic with different data storage requirements. When applied to the Mandelbrot fractal image generation and using optimizing techniques to minimize potential coding pitfalls, the performance gains of using SSE3 over x87 can be as much as 64%. The Intel Pentium 4 Processor with Hyper-Threading Technology and 90 nm technology includes two performance enhancing technology: SSE3 and Hyper-Threading Technology. The cumulative performance gains due to these two performance enhancements can be as much as 103% over hand-optimized x87 code for this Mandelbrot calculation.
5 Single Precision SSE3 Code Example

///
/// SSE3_cal_pixel_S
///
void SSE3_cal_pixel_S (complexS c0, complexS, DWORD * pCount0, DWORD * pCount1)
{
    complexS * pC0 = &c0;
    float fours[4] = { 4.0, 4.0, 4.0, 4.0 }; 
    float * pFours = fours;

    asm {
    //----- Setup ----- 
    push        ebx
    // xmm7 - load the two Cs: Clir C0i C0r
    mov         eax,pC0
    movups      xmm7,[eax]
    // xmm6 - four copies of exit test: 4.0 4.0 4.0 4.0
    mov         ecx,pFours
    movups      xmm6,[ecx]
    // xmm4 - counter - using negatives and initialized to one
    // as 'C' and x87 asm count each time thru
    pcmpeqd     xmm4,xmm4       // FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
    // xmm1 - initial Z values Z1 and Z0: d c b a or i1 r1 i0 r0
    pxor        xmm1,xmm1
    // setup the counters
    xor         ecx,ecx         // loop counter

    //----- Test Pixels ----- 
    TestPixels:
    // max iteration counter
    add         ecx,1
    // partial results
    movslldup   xmm0,xmm1       // c c a a -- real pairs
    movshdup    xmm2,xmm1       // d d b b -- imaginary pairs
    mulps       xmm0,xmm1       // dc cc ba aa - ri ii ri rr
    shufps      xmm1,xmm1,0xB1  // c d a b -- reordered 'Z' value
    mulps       xmm2,xmm1       // dc dd ba bb - ri rr ri rr
    // finish the complex multiply
    addsubps    xmm0,xmm2       // dc+cd cc-dd ba+ba aa-bb - d' c' b' a'
    // add constant C with result back in xmm1
    movaps      xmm1,xmm7
    addps       xmm1,xmm0       // Z = Z + C
    // length sq
    movaps      xmm3,xmm1       // copy of 'Z'
    mulps       xmm3,xmm1       // dd cc bb aa
    haddps       xmm3,xmm3       // dd+cc bb+aa dd+cc bb+aa

    //----- Loop Control ----- 
    cmpltps     xmm3,xmm6       // L1<4.0 L0<4.0 L1<4.0 L0<4.0
    paddd       xmm4,xmm3       // add the counters
    movmskps    eax,xmm3        // extract sign bits
    cmp         eax,0
    je          Done
    cmp         ecx,255
jl TestPixels

//----- Counters -----
Done:
  movd edx, xmm4       // Count0
  pshufd xmm4, xmm4, 0xB1 // reordered
  neg edx             // we counted negatives
  mov eax, pCount0
  movd ebx, xmm4      // Count1
  mov [eax], edx
  neg ebx
  mov ecx, pCount1
  mov [ecx], ebx

//----- Cleanup -----
pop ebx

// Output Values
if (*pCount0 > 255)
  *pCount0 = 0;
if (*pCount1 > 255)
  *pCount1 = 0;
} // end SSE3_cal_pixel_S
6 Double Precision SSE3 Code Example

// SSE3_cal_pixel_D

void SSE3_cal_pixel_D (double2 cR, double2 cI, DWORD * pCount0, DWORD * pCount1) {
  double2 * pCReal = &cR;
  double2 * pCImag = &cI;
  double fours[2] = { 4.0, 4.0 };
  double * pFours = fours;

  __asm {
    push   edi
    push   esi
    //----- Setup -----
    // initial values for Z are zero
    xorpd    xmm0,xmm0      // r1 r0
    xorpd    xmm1,xmm1      // i1 i0
    xorpd    xmm2,xmm2      // r0^2 I0^2
    xorpd    xmm3,xmm3     // r1^2 I1^2
    // Load Constants
    mov    eax,pCReal
    movupd   xmm4,[eax]     // Cr1 Cr0
    mov    eax,pCImag
    movupd   xmm5,[eax]     // Ci1 Ci0
    // xmm6 - four copies of exit test: 4.0 4.0 4.0 4.0
    mov    eax,pFours
    movupd   xmm6,[eax]
    // Counter - start
    mov    edi,1
    mov    esi,1
    // setup the counters
    xor      ecx,ecx        // loop counter
    // pipelining setup
    addpd    xmm2,xmm4     // add Cr ---> r1 r0

    //----- Test Pixels -----
  }

TestPixels:
  // max iteration counter
  inc ecx

  // temp = z.real^2 - z.imag^2 + Cr (pipelined) - moved

  // z.imag = z.real*z.imag + z.real*z.imag + Ci
  mulpd    xmm1,xmm0     // result=r1*i1 r0*i0
  addpd    xmm1,xmm1     // *2
  addpd    xmm1,xmm5     // add Ci ---> il i0

  // z.real
  movapd   xmm0,xmm2     // r1 r0

  // re-arrange to prepare for lengthsq
  movapd   xmm3,xmm2     // r1 r0
shufpd  xmm3,xmm1,3  // srcH,destH or i1 r1
shufpd  xmm2,xmm1,0  // srcL,destL or i0 r0

// lengthsq = z.real^2 + z.imag^2
mulpd   xmm3,xmm3  // I1^2 r1^2 - need to have r at low end
            // as hsubpd does low-high
mulpd   xmm2,xmm2  // I0^2 r0^2
movapd  xmm7,xmm2
haddpd  xmm7,xmm3  // L1 L0

// test
cmpltdp  xmm7,xmm6  // 4 4

// temp = z.real^2 - z.imag^2 + Cr  (pipelining)
hsubpd  xmm2,xmm3  // (A) xmm2=I0^2 r0^2  xmm3=I1^2 r1^2 ---> diff1 diff0
addpd   xmm2,xmm4  // (B) add Cr ---> r1 r0

movmskpd edx,xmm7
cmp    edx, 0
je     Done
mov    ebx,edx
and    edx,1
shr    ebx,1
add    esi,edx
add    edi,ebx
cmp    ecx,255
jl     TestPixels

Done:

mov    eax,pCount0
mov    edx,pCount1
mov    [eax],esi
mov    [edx],edi
pop    esi
pop    edi

// Output Values
if (*pCount0 > 255)
        *pCount0 = 0;
if (*pCount1 > 255)
        *pCount1 = 0;
} // end SSE3_cal_pixel_D
7 Configuration

7.1 Test System
The performance results were collected on production hardware:

- Processor: Intel® Pentium® 4 Processor with HT Technology\(^1\) 3.40E GHz
- Motherboard/Chipset: Intel® D875PBZL 875P
- BIOS: BZ87510A.86A.0083.P18.0312121502
- Memory: DDR4000 3-3-3; 1GB (2x512) dual channel
- Hard Disk: Maxtor 6L080J4
- Hard Disk Driver: Intel Application Accelerator RAID Edition 3.5 with RAID ready
- Chipset Driver: Intel® Chipset Installation Utility (INF) 5.01.1015
- AGP Graphics: ATI* Radeon* 9800 Pro 8x AGP graphics with 128 MB DDR memory
- Graphics Driver: ATI Catalyst 3.5 Driver Suite: display driver version: 6.14.10.6360 for Windows XP (wxp-w2k-radeon-7-90-030605m-009437c-efg.exe) and control panel (control-panel-7-90-030605m-009437c-efg.exe)
- Resolution: 1024x768 by 32-bit colors for all benchmarks
- Operating System: Microsoft* Windows XP Professional* (build 2600) with service pack 1
- Graphics Architecture: Microsoft DirectX* 9.0b
- File System: NTFS file system
- Sound Card: Creative* SoundBlaster Audigy* PCI (Gamer version. Model # SB0090) driver version 5.12.01.0129-1.00.0010 (Creative SB Audigy property)
- Network: Intel Pro/1000 MT desktop Gigabit PCI LAN adapter. A78408-xxx Intel part #: PWLA839MT. driver: 7.0.37.0

7.2 Build Environment
The build system ran Microsoft Windows XP Professional with Service Pack 1. The software was built using Intel® C++ Compiler for Windows 8.0 Beta running in the Microsoft* Visual Studio NET* IDE. The Intel C++ Compiler was used as it supports programming with SSE3. Program performance was tuned using the Intel® Vtune™ Performance Analyzer 7.1 Beta (12746).

Release build C++ command line options:

```
/c /Qvc7.1 /Qlocation,link,"C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\bin" /O3 /G7 /FD
/EHsc /MT /Zi /nologo /W3 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D "_MBCS" /Yu"StdAfx.h"
```

\(^1\) Hyper-Threading Technology requires a computer system with an Intel® Pentium® 4 processor supporting HT Technology and a Hyper-Threading Technology enabled chipset, BIOS and operating system. Performance will vary depending on the specific hardware and software you use. See http://www.intel.com/info/hyperthreading/ for more information including details on which processors support HT Technology.
Release build Link command line options:

/OUT:"Release/Mandelbrot_SSE3.exe" /INCREMENTAL:NO ddraw.lib winmm.lib /DEBUG
/MACHINE:IX86 kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib
ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib

7.3 Using VTune™ Performance Analyzer to Estimate Retirement Bandwidth

To estimate the average retirement bandwidth consumed by a given workload (for simplicity, we assume single-thread execution), one can use the following processor events (events are processor specific):

- Non-Halted Clockticks
- Uops Retired

Dividing “Non-Halted Clockticks” by “Uops Retired” gives the number of cycles required on average to retire a micro-op for a given workload. The peak retirement bandwidth of a processor based on the Intel® NetBurst® microarchitecture corresponds to 0.33333 cycles per micro-op.
8 Appendix

This appendix provides snippets of sample code that can be adapted to implement double precision complex multiplication and division using SSE3. These code snippets presume the double-precision complex data is stored with the real part, followed by the imaginary part. Furthermore, the input data is stored as an array of this complex data structure, and each complex data is aligned to 16 byte boundary.

8.1 Double-Precision Complex Multiply

// Compute (a+ bi) * (c+di)
movddup  xmm0, Src1 [8] ; a, a (high , low qword of xmm0)
movapd  xmm1, src2     ; d, c
mulpd   xmm0, xmm1      ; ad, ac
shufpd  xmm1, xmm1, 1  ; c, d
movddup  xmm2, Src1[8]  ; b, b
mulpd   xmm2, xmm1      ; bc, bd
addsubpd xmm0, xmm2     ; bc+ad, ac –bd (the final result )

8.2 Double-Precision Complex Division

// Compute (a+ bi) / (c+di)
movddup  xmm0, Src1[8]  ; b, b (high , low qword of xmm0)
movapd  xmm1, src2      ; d, c
mulpd   xmm0, xmm1      ; bd, bc
shufpd  xmm1, xmm1, 1  ; c, d
movddup  xmm2, Src1     ; a, a
mulpd   xmm2, xmm1      ; ac, ad
addsubpd xmm0, xmm2     ; ac+bd, bc -ad
mulpd   xmm1, xmm1      ; cc, dd
movpd   xmm2, xmm1      ; cc, dd
shufpd  xmm2, xmm2, 1   ; dd, cc
addpd   xmm2, xmm1      ; cc+dd, cc+dd
divpd   xmm0, xmm2      ; bc –ad)/(cc+dd), (ac+bd)/(cc+dd)