Using Streaming SIMD Extensions 2 (SSE2) to Find the Maximum/Minimum Element of a Double-Precision Floating-point Vector and its Corresponding Index

Version 2.0

7/00

Order Number: 248602-001

Table of Contents

Revision History

Revision	Revision History	Date
2.0	Pentium® 4 processor update	7/00
1.0	Original publication of document	9/99

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. Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, 1991
2. *Using the Streaming SIMD Extensions to Find the Maximum/Minimum Element of a Single-Precision Floating-point Vector and its Corresponding Index*, AP-805, Intel Corporation, Order Number 243639-001

1 Introduction

The Streaming SIMD Extensions 2 (SSE2) technology introduces new Single Instruction Multiple Data (SIMD) double-precision floating-point instructions and new SIMD integer instructions into the IA-32 Intel® architecture. The double-precision SIMD instructions extend functionality in a manner analogous to the single-precision instructions introduced by the SSE instructions. The 128-bit SIMD integer extensions are a full superset of the 64-bit integer SIMD instructions, with additional instructions to support more integer data types, conversion between integer and floating-point data types, and efficient operations between the caches and system memory. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, spatial (3D) audio, video encoding/decoding, encryption, and scientific application. This application note contains both the code and a description of how the SSE2 instructions can be used to find the maximum or minimum double-precision floating-point element of a vector and the element's corresponding index.

2 Maximum/Minimum Algorithm

This algorithm searches a single dimensional array (called a vector) filled with double-precision floating-point numbers in order to find the maximum or minimum value. Once this value is found, it returns the value along with the index of where that number can be found in the vector.

SSE2 technology allows us to load and compare two double-precision floating-point numbers at a time, which greatly improves the performance of this algorithm.

For the purpose of this application note, we will focus on finding the maximum value, since the algorithm is implemented in the same way (with minor changes) for both the minimum and the maximum.

2.1 Implementing the Maximum/Minimum Algorithm

The implementation of this algorithm in C generally begins with initializing the maximum value with the first number in the vector. This number is then compared to the next number and the maximum and index values are updated if a new maximum is found. The algorithm continues to repeat this process until all of the numbers in the vector have been compared against the maximum.

The C implementation of this algorithm is inefficient for two reasons. First, a branch is introduced after each comparison to determine if the maximum value and index should be updated as shown in the following example:

```
if(maxDouble < the_array[i])
{
     maxDouble = the_array[i];
     maxIndex = i;
}
```

Second, most C compilers generally do not take advantage of the SSE2 instructions.

From an algorithmic viewpoint, the C implementation makes N-1 comparisons, which is the least number of comparisons that it must perform.

2.1.1 Techniques

One technique for improving the performance of the C implementation would be to remove the branch (the `if` statement) contained within the loop.

One way of doing this is to compare all of the elements in the vector to find the maximum value (this requires N-1 comparisons). Once the maximum value is obtained, compare the elements in the vector against the maximum value until the index is located.

In the best case, the maximum value is the first element of the vector and the implementation has only N comparisons. In the worst case, the maximum value is the last element in the vector and the implementation has 2N-2 comparisons. Algorithmically, searching through a vector once or twice is still of linear complexity. However, removing the branch must significantly improve performance to justify searching through the vector more than once.

A second technique for improving the performance is to use SSE2 instructions to compare more than one double-precision floating-point number at a time.

These techniques are actually combined in the example code to provide the best performance improvement obtainable. The rest of this section discusses how these techniques are combined.

The SSE2 architecture provides eight 128-bit registers for floating-point calculations. These eight registers can each hold two 64 bit double-precision floating-point numbers. SSE2 instructions include several very valuable SIMD instructions that allow us to do the following:

1. Load two aligned double-precision floating-point numbers into a register. This instruction is the assembly instruction `movapd`.
2. Compare two 128-bit registers containing two double-precision floating-point numbers and store the two maximum values in one of the registers. This instruction is the assembly instruction `maxpd`.
3. Compare two 128-bit registers containing two double-precision floating-point numbers to see if they are equal and store the result in one of the registers. This instruction is the assembly instruction `cmpeqpd`.
4. Move values from 128-bit registers to 32 bit registers. This instruction is the assembly instruction `movmskpd`.
5. Logically OR 128-bit registers. This instruction is the assembly instruction orpd.
6. Other instructions that allow us to load unaligned numbers and shuffle numbers between registers.

In our first loop, we locate the maximum value in the vector by loading 7 registers at a time with 14 elements (1 register, `xmm0`, always maintains the maximum value) and comparing them. The assembly loop for accomplishing this is shown below:

```
maxloop:
   movapd xmm1,[edi - 16]
   movapd xmm2,[edi - 32]
   movapd xmm3,[edi - 48]
   movapd  xmm4,[edi - 64]
   movapd  xmm5,[edi - 80]
   movapd  xmm6,[edi - 96]
   movapd  xmm7,[edi - 112]
```

```
   sub    edi,112
   maxpd  xmm0,xmm1
   maxpd  xmm2,xmm3
   maxpd  xmm4,xmm5
   maxpd  xmm6,xmm7
   maxpd  xmm0,xmm2
   maxpd  xmm4,xmm6
   maxpd  xmm0,xmm4
   sub ecx,14
   cmp ecx,14
   jge maxloop
```

Notice that the only branch in this loop is the jump instruction at the end. As the processor correctly predicts this branch, there is no penalty for its execution (this is similar to the `for` loop in the C implementation). Architecturally, this loop is only limited by the speed at which the `maxpd` instructions can be scheduled. Consequently, it is extremely fast.

Once we have found the maximum value, we need to find its index. An example of the assembly code to do this follows:

```
indexloop:
   cmp    ecx,12
   jle    indexlast
   movapd    xmm0,[edi]
   movapd    xmm1,[edi + 16]
   movapd    xmm2,[edi + 32]
   movapd    xmm3,[edi + 48]
   movapd    xmm4,[edi + 64]
   movapd    xmm6,[edi + 80]
   add    edi,96
   add    edx,12
   sub  ecx,12

   cmpeqpd xmm0,xmm5
   cmpeqpd xmm1,xmm5
   cmpeqpd xmm2,xmm5
   cmpeqpd xmm3,xmm5
   cmpeqpd xmm4,xmm5
   cmpeqpd xmm6,xmm5
// OR our registers to see if the maximum value was found
   orpd xmm0,xmm1
   orpd xmm2,xmm3
```

```
   orpd xmm4,xmm6
   orpd xmm0,xmm2
   orpd xmm0,xmm4
// Move the result of the OR into eax
movmskpd eax,xmm0
   cmp    eax,0
   jz   indexloop
```

This loop is very similar to the loop in which we find the maximum value. The only difference is that we load 6 registers at a time, since one register holds our bit mask and we wish to have an even number of compares (to maximize the use of our `orpd` instruction).

The `movmskpd` instruction is an expensive instruction. In our example, we use the `orpd` instruction to identify to us that the index has been found (instead of a `movmskpd` each time). We then resolve the index once we exit the loop. The `orpd` and `movmskpd` instructions both use the same execution unit. Consequently, this loop is only limited by speed at which we can schedule and execute these instructions. Once again, the loop is extremely fast.

Combining these loops takes advantage of the SSE2 instructions and maximizes the speed of the algorithm. While the possibility of searching through a vector twice is counter intuitive, our ability to fully leverage SSE2 technology mitigates the linear increase in search time.

2.1.2 Tips and Tricks

1. Performance is greatly enhanced when the vector is aligned on a 16-byte memory boundary. The implementations discussed in this application note assume 8-byte or 16-byte alignment. Check your compiler documentation for ways to insure the proper alignment.
2. Prior to the index loop of the assembly code, the end of the vector is checked for the maximum value. We also do this for the front of the array. If the index is found, the code jumps to the label `indexdone`. This technique has two advantages:
   - It checks any unaligned elements at the front and rear of the array so that we don't have to worry about the alignment.
   - It allows us enter the index loop knowing that we will find the maximum value (and its corresponding index) before we walk off the end of the vector.
3. If you wish to find the minimum value, simply substitute the `minpd` instruction for the `maxpd` instruction in the assembly code. The same can be done in the SSE2 C++ Double-Precision Vector Class (DVEC) code by substituting the `simd_min` instruction for the `simd_max` instruction.
4. The first loop starts at the end of the vector and search to the front. The second loop starts at the front of the vector and search to the rear. This technique optimizes data cache usage.
5. Our worst case scenario is where the maximum value is 3 elements from the rear of the vector. We always check the last two before we begin looping.

3 Performance

3.1 Gains/Improvements

The assembly code implementation using SSE2 technology is more than twice as fast as its corresponding C implementation in the average case. This was owing to several factors:

1. The use of SSE2 instructions that allow us to operate on two double-precision floating-point elements at a time. These instructions are significantly faster than previous x87 instructions.
2. The optimization of the algorithm, which included the following:
   - Removing the conditional branch (the `if` statement) in the C code.
   - Pipelining the loading of elements from memory.
   - Operating on multiple registers filled with multiple elements during each loop iteration.

The algorithmic optimizations could only have occurred with the new SSE2 instructions.

3.2 Considerations

The SSE2 code examples provided in this application note perform extremely well under most conditions. These examples however, may not perform as well as other SSE2 implementations on large vector sizes (greater than 10,000 elements) in the worst case (maximum value at the end of the vector). In these cases, one of the following is recommended:

1. Break the vector up into smaller pieces where the vector size (in bytes) is smaller than the first level cache. Use this algorithm to operate on these smaller pieces.
2. Use a different implementation. Please refer to application note AP-805 if you want to see a different implementation.

4 Conclusion

Using SSE2 instructions can significantly improve the performance of this algorithm. A good example is the `maxpd` instruction used in this algorithm. These instructions allow us to operate on two double-precision floating-point elements simultaneously and provide us the ability to create more powerful algorithms for solving problems.

5 C/C++ Coding Example

```
double max_c(double *the_array, int array_size, int *index)
{
int maxIndex = 0;
// Initialize maxDouble with the value of the first item in the vector
double maxDouble = the_array[0];
for(int i=1; i<array_size; i++)
  {
// Compare maxDouble with remaining vector elements
// Keep track of the maximum value and its index
if(maxDouble < the_array[i])
    {
maxDouble = the_array[i];
maxIndex = i;
    }
  }
*index = maxIndex;
return(maxDouble);
}
```

6 SSE2 Technology DVEC Code Example

```
double maxw_dvec_unrolled(double *the_array, int array_size, int *index)
{

     // Assume 8 or 16 byte alignment
   assert(((unsigned int)&the_array[0] & (0x07)) == 0);
   // Use C code if array size is small
   if (array_size<=18)
   {
      int   maxIndex = 0;
     double maxDouble = the_array[0];

     for(int i=1; i<array_size; i++)
     {

       if(maxDouble  < the_array[i])
       {
         maxDouble  = the_array[i];
         maxIndex = i;
       }
     }
     *index = maxIndex;
     return(maxDouble);
   }

   F64vec2 r1(0.0,0.0), r2(0.0,0.0), r3(0.0,0.0), r4(0.0,0.0), r5(0.0,0.0),
r6(0.0,0.0), r7(0.0,0.0), r8(0.0,0.0);
   double max = 0.0;
   int i=0;
   int j, mask;
   F64vec2 *aligned_front_of_array;
   F64vec2 *aligned_end_of_array;
   *index = 0;
   int front_alignment,back_alignment;
   // Calculate alignments and compensate if 8 byte aligned
   if((((unsigned int)&the_array[0]) & (0x0F)) == 0) front_alignment = 1;
   else front_alignment = 0;
```

```
  if((((unsigned int)&the_array[array_size - 1]) & (0x0F)) == 0)
back_alignment = 1;
  else back_alignment = 0;

  if(!back_alignment) {
    aligned_end_of_array = (F64vec2 *)&the_array[array_size - 2];
  } else aligned_end_of_array = (F64vec2 *)&the_array[array_size - 1];

  if(!front_alignment) {
    aligned_front_of_array = (F64vec2 *)&the_array[1];
  } else aligned_front_of_array = (F64vec2 *)&the_array[0];

  r1 = _mm_loadu_pd(&the_array[array_size - 2]);
  r2 = _mm_loadu_pd(&the_array[0]);
  r1 = simd_max(r1,r2);
  Loop through the vector and find the maximum value
  j = array_size/16 * 8;
  for(i=1; i<j; i+=8) {
    r1 = simd_max(r1,*(aligned_end_of_array - i));
    r2 = simd_max(r2,*(aligned_end_of_array - (i+1)));
    r3 = simd_max(r3,*(aligned_end_of_array - (i+2)));
    r4 = simd_max(r4,*(aligned_end_of_array - (i+3)));
    r5 = simd_max(r5,*(aligned_end_of_array - (i+4)));
    r6 = simd_max(r6,*(aligned_end_of_array - (i+5)));
    r7 = simd_max(r7,*(aligned_end_of_array - (i+6)));
    r8 = simd_max(r8,*(aligned_end_of_array - (i+7)));
  }

  r1 = simd_max(r1,*(aligned_front_of_array));
  r2 = simd_max(r2,*(aligned_front_of_array+1));
  r3 = simd_max(r3,*(aligned_front_of_array+2));
  r4 = simd_max(r4,*(aligned_front_of_array+3));
  r5 = simd_max(r5,*(aligned_front_of_array+4));
  r6 = simd_max(r6,*(aligned_front_of_array+5));
  r7 = simd_max(r7,*(aligned_front_of_array+6));
  r8 = simd_max(r8,*(aligned_front_of_array+7));
  r1 = simd_max(r1,r2);
  r3 = simd_max(r3,r4);
  r5 = simd_max(r5,r6);
  r7 = simd_max(r7,r8);
  r1 = simd_max(r1,r3);
```

```
r5 = simd_max(r5,r7);
r1 = simd_max(r1,r5);

// Create a mask of maximum values in r5
r5 = unpack_low(r1,r1);
r1 = unpack_high(r1,r1);
r5 = simd_max(r5,r1);

_mm_store_sd(&max,r5); // Store the max value

// Calculate the index now (starting from the front of array in cache)
if(!front_alignment) {
  r1 = _mm_loadu_pd(&the_array[0]);
  r1 = cmpeq(r1,r5);
  mask = move_mask(r1);
  // If we are lucky, the max is in the front of the array
  if(mask) {
    if(mask == 3) mask = 1;
    *index = mask-1;
    return(max);
  }
  *index = 1;
}

// Last two doubles to look at
r1 = _mm_loadu_pd(&the_array[array_size - 2]);
r1 = cmpeq(r1,r5);
mask = move_mask(r1);
if(mask) {
  if(mask == 2) *index = array_size - 1;
  else  *index = array_size - 2;
  return(max);
}

i = 0;
// Go through array from the front and look for index
while(!mask) {
  r1 = cmpeq(*(aligned_front_of_array+i),r5);
  i++;
  r2 = cmpeq(*(aligned_front_of_array+i),r5);
```

```
      i++;
      r3 = cmpeq(*(aligned_front_of_array+i),r5);
      i++;
      r4 = cmpeq(*(aligned_front_of_array+i),r5);
      i++;
      r6 = cmpeq(*(aligned_front_of_array+i),r5);
      i++;
      r7 = cmpeq(*(aligned_front_of_array+i),r5);
      i++;

      r1 = _mm_or_pd(r1,r2);
      r3 = _mm_or_pd(r3,r4);
      r6 = _mm_or_pd(r6,r7);
      r1 = _mm_or_pd(r1,r3);
      r1 = _mm_or_pd(r1,r6);

      mask = move_mask(r1);
      if((i*2+12) >= array_size) break;
   }

   i -= 6;
   mask = 0;
   while(!mask) {
      r1 = cmpeq(*(aligned_front_of_array+i),r5);
      mask = move_mask(r1);
      i++;
    }
   i--;
   if(mask == 3) mask = 1;
   *index += 2*i + (mask-1);
   return(max);
}
```

7 SSE2 Technology Assembly Code Example

```
double maxw_asm(double *the_array, int array_size, int *index)
{
   double maximum;
   int indexvalue = 0;
   double *end_of_array,*aligned_end_of_array,*aligned_front_of_array;

   // Assume 8 or 16 byte alignment
   assert(((unsigned int)&the_array[0] & (0x07)) == 0);

   // Array size must be at least 18 elements or we use the C code
   if (array_size<=18)
   {
       int  maxIndex = 0;
     float maxFloat = the_array[0];

     for(int i=1; i<array_size; i++)
     {

       if(maxFloat < the_array[i])
       {
         maxFloat = the_array[i];
         maxIndex = i;
       }
     }
     *index = maxIndex;
     return(maxFloat);
   }

   end_of_array = &the_array[array_size - 1];
   int front_alignment,back_alignment;

   if((((unsigned int)&the_array[0]) & (0x0F)) == 0) front_alignment = 1;
   else front_alignment = 0;

   if((((unsigned int)&the_array[array_size - 1]) & (0x0F)) == 0)
back_alignment = 1;
```

```
   else back_alignment = 0;

   if(!back_alignment) {
     aligned_end_of_array = &the_array[array_size - 2];
   } else aligned_end_of_array = &the_array[array_size - 1];

   if(!front_alignment) {
     aligned_front_of_array = &the_array[1];
     indexvalue = 1;
   } else aligned_front_of_array = &the_array[0];


__asm {
   mov    esi,the_array
   mov    ecx,array_size
   mov    edx,ecx
   mov    edi,aligned_end_of_array
   mov    eax,end_of_array

   movupd xmm0,[eax - 8]
   movupd xmm1,[esi]
   maxpd  xmm0,xmm1
   sub    ecx,1

   // Loop where we find the maximum
maxloop:

   movapd xmm1,[edi - 16]
   movapd xmm2,[edi - 32]
   movapd xmm3,[edi - 48]
   movapd xmm4,[edi - 64]
   movapd xmm5,[edi - 80]
   movapd xmm6,[edi - 96]
   movapd xmm7,[edi - 112]
   sub    edi,112

   maxpd  xmm0,xmm1
   maxpd  xmm2,xmm3
```

```
   maxpd   xmm4,xmm5
   maxpd   xmm6,xmm7
   maxpd   xmm0,xmm2
   maxpd   xmm4,xmm6
   maxpd   xmm0,xmm4

   sub     ecx,14
   cmp     ecx,14
   jge     maxloop

   mov     edi,aligned_front_of_array

maxdone:

   movapd  xmm2,[edi]
   maxpd   xmm0,xmm2

   add     edi,16
   sub     ecx,2
   cmp     ecx,0
   jg   maxdone

   mov     edi,aligned_front_of_array

   shufpd  xmm5,xmm0,3
   maxpd   xmm5,xmm0
   shufpd  xmm5,xmm5,1
   maxpd   xmm5,xmm0   // Created mask of maximum values in xmm5
   movsd   maximum,xmm5   // Stored maximum value

   sub     edx,2
   movupd  xmm0,[eax - 8]
   cmpeqpdxmm0,xmm5
   movmskpd  eax,xmm0
   cmp     eax,0
   jne     indexdone

   xor     edx,edx
```

```
   movupd  xmm0,[esi]
   cmpeqpdxmm0,xmm5
   movmskpd  eax,xmm0
   cmp     eax,0
   jne     indexdone

   mov     ecx,array_size

   // Loop where we find the index
indexloop:

   cmp     ecx,12
   jle     indexlast
   movapd  xmm0,[edi]
   movapd  xmm1,[edi + 16]
   movapd  xmm2,[edi + 32]
   movapd  xmm3,[edi + 48]
   movapd  xmm4,[edi + 64]
   movapd  xmm6,[edi + 80]
   add     edi,96
   add     edx,12
   sub     ecx,12

   cmpeqpdxmm0,xmm5
   cmpeqpdxmm1,xmm5
   cmpeqpdxmm2,xmm5
   cmpeqpdxmm3,xmm5
   cmpeqpdxmm4,xmm5
   cmpeqpdxmm6,xmm5

   orpd xmm0,xmm1
   orpd xmm2,xmm3
   orpd xmm4,xmm6
   orpd xmm0,xmm2
   orpd xmm0,xmm4
   movmskpd eax,xmm0
   cmp     eax,0
   jz   indexloop
```

```
   sub    edx,12
   sub    edi,96

indexlast:

   movapd xmm0,[edi]
   cmpeqpd xmm0,xmm5
   movmskpd eax,xmm0
   add    edi,16
   add    edx,2
   cmp    eax,0
   jz   indexlast

   sub    edx,2
   add    edx,indexvalue

indexdone:

   cmp    eax,2
   jne    end
   add    edx,1

end:

   mov    indexvalue,edx

}

   *index = indexvalue;
   return(maximum);
}
```

Appendix A - Performance Data

Revision History

Revision	Revision History	Date
2.0	Updated with 1.2 GHz Pentium® 4 processor performance data	7/00
1.0	Original publication of document	9/99

Table 1: Performance Data of Max Implementations

Performance Data in Microseconds		
Cases	**Pentium® III Processor (733 MHz)**	**Pentium 4 Processor (1.2 GHz)**
C Code Middle* Case	7.74	4.94
SSE2 ASM Best* Case		1.09
SSE2 ASM Middle* Case		1.74
SSE2 ASM Worst* Case		2.22
SSE2 DVEC Best* Case		1.14
SSE2 DVEC Middle* Case		1.86
SSE2 Worst* Case		2.48

* Best Case is the maximum element at the front of the vector. Middle Case is the maximum element in the middle of the vector. Worst Case is the maximum element at the end of the vector (see algorithm).

Table 2: Speedups from Table 1 Performance Data

Implementations and Platforms	Speedup
Pentium 4 Processor (SSE2 ASM vs. C Code in Middle Case)	2.84
Pentium 4 Processor (SSE2 DVEC vs. C Code in Middle Case)	2.66
C Code on Pentium 4 Processor vs. C Code on PentiumIII processor	1.57

Table 1 and 2 measure the performance of a 1.2 GHz Pentium 4 processor and a 733 MHz Intel Pentium III processor when executing the max/min code. See Test System Configurations for a detailed description of the Pentium 4 and Pentium III processor systems. Performance on both processors was measured with test data in the first level cache.

A vector length of 1000 elements was chosen for representation. Obviously, the performance of the C versions does not change regardless of the position of the max/min element. Full compiler optimizations were used with the Intel® C/C++ Compiler to compile the C code. For the assembly (ASM) and vector class library (DVEC) code, performance is indicated by the position of the max/min element.

For max/min, one would expect the SIMD width of the SSE2 instructions to provide a 2x improvement over scalar code in the optimal case. However, using SSE2 technology, a speedup of 2.84x can be achieved (on the average – with max/min in the middle position). This is resulting from the improvements in the overall algorithm, which can successfully exploit SSE2 technology.

Test Systems Configuration

Table 3: Pentium III Configuration

Processor	Pentium III Processor at 733 MHz
System	Intel® Desktop Board VC820
Bios Version	VC82010A.86A.0028.P10
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster† Annihilator† Pro AGP nVidia GeForce256† DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows† 2000 Build 2195

Table 4: Pentium 4 Configuration

Processor	Pentium 4 Processor at 1.2 GHz
System	Intel Desktop Board D850GB
Bios Version	GB85010A.86A.0014.D.0007201756
Secondary Cache	256KB
Memory Size	128 MB RDRAM PC800-45
Ultra ATA Storage Driver	Production Candidate 6.00.012
Hard Disk	IBM DJNA-371800 ATA-66
Video Controller/Bus	Creative Labs 3D Blaster Annihilator Pro AGP nVidia GeForce256 DDR –32MB
Video Driver Revision	NVidia Reference Driver 5.22
Operating System	Windows 2000 Build 2195