Using the Intel® SSSE3 Instruction Set to Accelerate DNN Algorithm in Local Speech Recognition

By Li Alven

Overview
Over the past thirty years, speech recognition technology has made significant progress, starting in the lab to the market. Speech recognition technology is becoming important in people lives, and is found in our jobs, houses, automotive, medical and other fields. It’s is one of the TOP 10 merging technologies in the world.

As a result of this year’s developments, the main algorithm of speech recognition technology has changed from GMM(Gaussian Mixture Model) and, HMM-GMM(Hidden Markov Model-Gaussian Mixture Model) to DNN(Deep Neural Network). DNN functions similar to the way a human’s brain works, it is a very complicated, heavy calculation, and huge data based model. Thanks to the internet, we just only need a smartphone and don’t care about the huge number of servers in the remote computer room that make it all happen. Without internet, the speech recognition service in your mobile devices nearly useless, very few times it can listen to what you said and work.

Is it possible to move the DNN calculation process from server to the mobile end device? Phones? Tablets? The answer is YES.

With support for the SSSE3 instruction set on Intel’s CPU, we could easy run a DNN based speech recognition application without the internet. The accuracy is over 80% by our test, that’s very close to the result of online mode tests. Adding direct SSSE3 support creates a good user experience on mobile devices. In this article I will explain what is DNN and how the Intel® SSSE3 instruction set helps to accelerate DNN calculation progress.

Introduction
DNN is the abbreviation for Deep Neural Network, which contains a many hidden layer feed forward network. DNN is a hot spot in the field of machine learning in recent years, producing a wide range of applications. DNN has a deep structure, with tens of millions of parameters needed to be learned, and the lead time for trainning is very time consuming.
Speech recognition is a typical application case of DNN. To put it simply, speech recognition applications consist of an acoustical model, language model and a decoding process. The acoustical model is used to simulate the probability distribution of pronunciation. The language model is used to simulate the relationship between words. And the decoding process stage uses the above two models to translate the sound to into text. A neural network has the ability to simulate any word distribution. Where a deep neural network has a stronger expression ability than a shallow neural network, it simulates the deep structure of the brain and can “understand” more accurately the characteristics of things. So compared with other methods, the depth of the neural network can be a more accurately simulated acoustic and language model.

![Typical DNN Chart](image)

**Figure 1. DNN Application Field**

**Typical DNN Chart**

A typical DNN generally contains multiple alternate superposition of a linear and non-linear layer, as it shows below:
Figure 2. Including 4 hidden layer DNN acoustic model

In Figure 2, the linear layer is a fully connected relationship, and the input to output could be described by this formula:

\[ Y = XW + B \]

\( X \) is the row vector, and input is by neural network. In a speech recognition application, we generally put 4 frames of data to calculate together, so that we create a 4xM input matrix. \( W \) and \( B \) is the linear transformation matrix of the neural network and offset vector, usually the dimension is huge and square.

**Intel® SSSE3 Instruction Set**

Supplemental Streaming SIMD Extensions 3, or SSSE3 for short, is named by Intel and as the extension of SSSE3 instruction set. The SSSE3 instruction set is a part of SIMD technology, which has been integrated into Intel’s CPU and helps to improve the ability of multimedia processing, coding/decoding, and calculations. Using the SSSE3 instruction set, we can process multiple data inputs by a single instruction in a clock cycle, and then greatly improve the program’s efficiency. It works particularly for matrix calculations.

To use the SSSE3 instruction set, we should first declare and include the SIMD header files:
The header file “tmmintrin.h” is for SSSE3, and the functions defined in this file are below:

```c
1. #include <mmintrin.h> //MMX
2. #include <xmmmintrin.h> //SSE(include mmintrin.h)
3. #include <emmintrin.h>
4. #include <pmmintrin.h>
5. #include <wmmintrin.h>
6. #include <nmmintrin.h>
7. #include <smmintrin.h>
8. #include <tmmintrin.h>
9. #include <avx/include xwmmintrin.h>
10. #include <immintrin.h>

The header file “tmmintrin.h” is for SSSE3, and the functions defined in this file are below:

```c
1. /*Add horizontally packed [saturated] words, double words,
2. }{X},MM2/m{128,64} (b) to }{X},MM1 (a).*/
3. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)
4. //then r0=a0+a1, r1=a2+a3, r2=a4+a5, r3=a6+a7, r4=b0+b1, r5=b2+b3, r6=b4+b5, r7=b6+b7
5. extern __m128i __m128_mm_hadd_epi16 (__m128i a, __m128i b);
6. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
7. //then r0=a0+a1, r1=a2+a3, r2=b0+b1, r3=b2+b3
8. extern __m128i __m128_mm_hadd_epi32 (__m128i a, __m128i b);
9. //SATURATE_16(x) is ((x > 32767) ? 32767 : ((x < -32768) ? -32768 : x))
10. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)
11. //then r0=SATURATE_16(a0+a1), ..., r3=SATURATE_16(a6+a7),
12. //r4=SATURATE_16(b0+b1), ..., r7=SATURATE_16(b6+b7)
13. extern __m128i __m128_mm_hadds_epi16 (__m128i a, __m128i b);
14. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
15. //then r0=a0+a1, r1=a2+a3, r2=b0+b1, r3=b2+b3
16. extern __m64 __m64_mm_hadd_epi16 (__m64 a, __m64 b);
17. //a=(a0, a1), b=(b0, b1), then r0=a0+a1, r1=b0+b1
18. extern __m64 __m64_mm_hadd_epi32 (__m64 a, __m64 b);
19. //SATURATE_16(x) is ((x > 32767) ? 32767 : ((x < -32768) ? -32768 : x))
20. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
21. //then r0=SATURATE_16(a0+a1), r1=SATURATE_16(a2+a3),
22. //r2=SATURATE_16(b0+b1), r3=SATURATE_16(b2+b3)
23. extern __m64 __m64_mm_hsubs_epi16 (__m64 a, __m64 b);
24. /*Subtract horizontally packed [saturated] words, double words,
25. }{X},MM2/m{128,64} (b) from }{X},MM1 (a).*/
26. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)
27. //then r0=a0-a1, r1=a2-a3, r2=a4-a5, r3=a6-a7, r4=b0-b1, r5=b2-b3, r6=b4-b5, r7=b6-b7
28. extern __m128i __m128_mm_hsub_epi16 (__m128i a, __m128i b);
29. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
30. //then r0=a0-a1, r1=a2-a3, r2=b0-b1, r3=b2-b3
31. extern __m128i __m128_mm_hsub_epi32 (__m128i a, __m128i b);
32. //SATURATE_16(x) is ((x > 32767) ? 32767 : ((x < -32768) ? -32768 : x))
33. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)```
35. //then r0=SATURATE_16(a0-a1), ..., r3=SATURATE_16(a6-a7),
36. //r4=SATURATE_16(b0-b1), ..., r7=SATURATE_16(b6-b7)
37. extern __m128i __mm_hsubs_epi16 (__m128i a, __m128i b);
38. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
39. //then r0=a0-a1, r1=a2-a3, r2=b0-b1, r3=b2-b3
40. extern __m64 __mm_sub_pi16 (__m64 a, __m64 b);
41. //a=(a0, a1), b=(b0, b1), then r0=a0-a1, r1=b0-b1
42. extern __m64 __mm_sub_pi32 (__m64 a, __m64 b);
43. //SATURATE_16(x) is ((x > 32767) ? 32767 : ((x < -32768) ? -32768 : x))
44. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
45. //then r0=SATURATE_16(a0-a1), r1=SATURATE_16(a2-a3),
46. //r2=SATURATE_16(b0-b1), r3=SATURATE_16(b2-b3)
47. extern __m64 __mm_hsubs_pi16 (__m64 a, __m64 b);
48. 
49. /*Multiply and add packed words,
50. (X)(Y)MM2/m128,64 (b) to (X)(Y)MM1 (a).* /
51. //SATURATE_16(x) is ((x > 32767) ? 32767 : ((x < -32768) ? -32768 : x))
52. //a=(a0, a1, a2, ..., a13, a14, a15), b=(b0, b1, b2, ..., b13, b14, b15)
53. //then r0=SATURATE_16((a0*b0)+(a1*b1)), ..., r7=SATURATE_16((a14*b14)+(a15*b15))
54. //Parameter a contains unsigned bytes. Parameter b contains signed bytes.
55. extern __m128i __mm_maddubs_epi16 (__m128i a, __m128i b);
56. //SATURATE_16(x) is ((x > 32767) ? 32767 : ((x < -32768) ? -32768 : x))
57. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)
58. //then r0=SATURATE_16((a0*b0)+(a1*b1)), ..., r3=SATURATE_16((a6*b6)+(a7*b7))
59. //Parameter a contains unsigned bytes. Parameter b contains signed bytes.
60. extern __m64 __mm_maddubs_pi16 (__m64 a, __m64 b);
61. 
62. /*Packed multiply high integers with round and scaling,
63. (X)(Y)MM2/m128,64 (b) to (X)(Y)MM1 (a).* /
64. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)
65. //then r0=INT16(((a0*b0)+0x4000) >> 15), ..., r7=INT16(((a7*b7)+0x4000) >> 15)
66. extern __m128i __mm_mulhrs_epi16 (__m128i a, __m128i b);
67. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
68. //then r0=INT16(((a0*b0)+0x4000) >> 15), ..., r3=INT16(((a3*b3)+0x4000) >> 15)
69. extern __m64 __mm_mulhrs_pi16 (__m64 a, __m64 b);
70. 
71. /*Packed shuffled bytes
72. (X)(Y)MM2/m128,64 (b) to (X)(Y)MM1 (a).* /
73. //SELECT(a, n) extracts the nth 8-bit parameter from a. The 0th 8-bit parameter
74. //is the least significant 8-bits, b=(b0, b1, b2, ..., b13, b14, b15), b is mask
75. //then r0 = (b0 & 0x80) ? 0 : SELECT(a, b0 & 0x0f), ...
76. //r15 = (b15 & 0x80) ? 0 : SELECT(a, b15 & 0x0f)
77. extern __m128i __mm_shuffle_epi8 (__m128i a, __m128i b);
78. //SELECT(a, n) extracts the nth 8-bit parameter from a. The 0th 8-bit parameter
79. //is the least significant 8-bits, b=(b0, b1, ..., b7), b is mask
80. //then r0= (b0 & 0x80) ? 0 : SELECT(a, b0 & 0x07), ...
81. //r7=(b7 & 0x80) ? 0 : SELECT(a, b7 & 0x07)
82. extern __m64 __mm_shuffle_pi8 (__m64 a, __m64 b);
83. \n84. /*Packed byte, word, double word sign, \{X,\}MM2/m\{128,64\} (b) to \{X,\}MM1 (a).*/
85. //a=(a0, a1, a2, ..., a13, a14, a15), b=(b0, b1, b2, ..., b13, b14, b15)
86. //then r0=(b0 < 0) ? -a0 : ((b0 == 0) ? 0 : a0), ...
87. //r15= (b15 < 0) ? -a15 : ((b15 == 0) ? 0 : a15)
88. extern __m128i __mm_sign_epi8 (__m128i a, __m128i b);
89. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)
90. //r0=(b0 < 0) ? -a0 : ((b0 == 0) ? 0 : a0), ...
91. //r7= (b7 < 0) ? -a7 : ((b7 == 0) ? 0 : a7)
92. extern __m128i __mm_sign_epi16 (__m128i a, __m128i b);
93. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
94. //then r0=(b0 < 0) ? -a0 : ((b0 == 0) ? 0 : a0), ...
95. //r3= (b3 < 0) ? -a3 : ((b3 == 0) ? 0 : a3)
96. extern __m128i __mm_sign_epi32 (__m128i a, __m128i b);
97. //a=(a0, a1, a2, a3, a4, a5, a6, a7), b=(b0, b1, b2, b3, b4, b5, b6, b7)
98. //then r0=(b0 < 0) ? -a0 : ((b0 == 0) ? 0 : a0), ...
99. //r7= (b7 < 0) ? -a7 : ((b7 == 0) ? 0 : a7)
100. extern __m64 __mm_sign_pi18 (__m64 a, __m64 b);
101. //a=(a0, a1, a2, a3), b=(b0, b1, b2, b3)
102. //a=(a0, a1, b=(b0, b1, b2, b3), then r0=(b0 < 0) ? -a0 : ((b0 == 0) ? 0 : a0),
103. //r3= (b3 < 0) ? -a3 : ((b3 == 0) ? 0 : a3)
104. extern __m64 __mm_sign_pi16 (__m64 a, __m64 b);
105. //a=(a0, a1), b=(b0, b1), then r0=(b0 < 0) ? -a0 : ((b0 == 0) ? 0 : a0),
106. //r1= (b1 < 0) ? -a1 : ((b1 == 0) ? 0 : a1)
107. extern __m64 __mm_sign_pi32 (__m64 a, __m64 b);
108.
109. /*Packed align and shift right by n*8 bits,
110. \{X,\}MM2/m\{128,64\} (b) to \{X,\}MM1 (a).*/
111. //n: A constant that specifies how many bytes the interim result will be
112. //shifted to the right, If n > 32, the result value is zero
113. //CONCAT(a, b) is the 256-bit unsigned intermediate value that is a
114. //concatenation of parameters a and b.
115. //The result is this intermediate value shifted right by n bytes.
116. //then r= (CONCAT(a, b) >> (n * 8)) & 0xffffffffffffffff
117. extern __m128i __m_alignr_epi8 (__m128i a, __m128i b, int n);
118. //n: An integer constant that specifies how many bytes to shift the interim
119. //result to the right, If n > 16, the result value is zero
120. //CONCAT(a, b) is the 128-bit unsigned intermediate value that is formed by
121. //concatenating parameters a and b.
122. //The result value is the rightmost 64 bits after shifting this intermediate
123. //result right by n bytes
124. //then r = (CONCAT(a, b) >> (n * 8)) & 0xffffffff
125. extern __m64 __m_alignr_pi18 (__m64 a, __m64 b, int n);
126.
127. /*Packed byte, word, double word absolute value,
128. \{X,\}MM2/m\{128,64\} (b) to \{X,\}MM1 (a).*/
129. //a=(a0, a1, a2, ..., a13, a14, a15)
130. //then r0 = (a0 < 0) ? -a0 : a0, ..., r15 = (a15 < 0) ? -a15 : a15
131. extern __m128i _mm_abs_epi8 (__m128i a);
132. //a=(a0, a1, a2, a3, a4, a5, a6, a7)
133. //then r0 = (a0 < 0) ? -a0 : a0, ..., r7 = (a7 < 0) ? -a7 : a7
134. extern __m128i _mm_abs_epi16 (__m128i a);
135. //a=(a0, a1, a2, a3)
136. //then r0 = (a0 < 0) ? -a0 : a0, ..., r3 = (a3 < 0) ? -a3 : a3
137. extern __m128i _mm_abs_epi32 (__m128i a);
138. //a=(a0, a1, a2, a3, a4, a5, a6, a7)
139. //then r0 = (a0 < 0) ? -a0 : a0, ..., r7 = (a7 < 0) ? -a7 : a7
140. extern __m64 _mm_abs_pi8 (__m64 a);
141. //a=(a0, a1, a2, a3)
142. //then r0 = (a0 < 0) ? -a0 : a0, ..., r3 = (a3 < 0) ? -a3 : a3
143. extern __m64 _mm_abs_pi16 (__m64 a);
144. //a=(a0, a1), then r0 = (a0 < 0) ? -a0 : a0, r1 = (a1 < 0) ? -a1 : a1
145. extern __m64 _mm_abs_pi32 (__m64 a);

The data structure definition of __m64 and __m128 are in MMX's header file “mmintrin.h” and SSE header file “xmmintrin.h”.

__m64:

1. typedef union __declspec(intrin_type) _CRT_ALIGN(8) __m64
2. {
3.    unsigned __int64 m64_u64;
4.    float m64_f32[2];
5.    __int8 m64_i8[8];
6.    __int16 m64_i16[4];
7.    __int32 m64_i32[2];
8.    __int64 m64_i64;
9.    unsigned __int8 m64_u8[8];
10.   unsigned __int16 m64_u16[4];
11.   unsigned __int32 m64_u32[2];
12. } __m64;

__m128:

1. typedef union __declspec(intrin_type) _CRT_ALIGN(16) __m128 {
2.    float m128_f32[4];
3.    unsigned __int64 m128_u64[2];
4.    __int8 m128_i8[16];
5.    __int16 m128_i16[8];
6.    __int32 m128_i32[4];
7.    __int64 m128_i64[2];
8.    unsigned __int8 m128_u8[16];
9.    unsigned __int16 m128_u16[8];
10.   unsigned __int32 m128_u32[4];
11. } __m128;
Case study: using SSSE3 functions to accelerate DNN calculation

In this section, we take two functions as a sample to describe how SSSE3 is used to accelerate the DNN calculation process.

__m128i _mm_maddubs_epi16 (__m128i a, __m128i b) Saturated Accumulation Operation

This function is very critical for the matrix calculation in DNN, the parameter a is a 128bit register, used to store 16 unsigned integers which are 8bit, and parameter b is 16 signed integer which also is 8bit; the return result which included 8 signed 16bit integer. This function is perfect meet the requirement of matrix calculation. such as:

\[
\begin{align*}
    r0 & := \text{SATURATE}_16((a0*\text{b0}) + (a1*\text{b1})) \\
    r1 & := \text{SATURATE}_16((a2*\text{b2}) + (a3*\text{b3})) \\
    \vdots & \nonumber \\
    r7 & := \text{SATURATE}_16((a14*\text{b14}) + (a15*\text{b15})) \\
\end{align*}
\]

__m128i _mm_hadd_epi32 (__m128i a, __m128i b) Adjacent Elements Add Operation

This function can be called pair-wise add. The parameters a and b both are 128bit registers which store a 4 signed integer of 32bit. According to normal corresponding element add operation in two vector, it does the add operation with adjacent elements with input vector. Such as:

\[
\begin{align*}
    r0 & := a0 + a1 \\
    r1 & := a2 + a3 \\
    r2 & := b0 + b1 \\
    r3 & := b2 + b3 \\
\end{align*}
\]

Then, we suppose there’s a task of vector calculation in DNN process:

Q: There are five vectors a1, b1, b2, b3, b4. The a1 vector is 16 dimension unsigned-char integer, b1, b2, b3, b4 are both 16 dimension signed-char integers. We need the inner product of a1*b1, a1*b2, a1*b3, a1*b4, and to store the result in a signed int of 32bit.

If we used normal design and C program language to implement it, the coding would be as follows:

1. unsigned char b1[16], b2[16], b3[16], b4[16];
2. signed char a1[16];
3. int c[4], i;
4. //
5. Initialize b1, b2, b3, b4 and a1, for c, initialize with zeros
6. //
7. for(i=0; i<16; i++){
8.        c[0] += (short)a1[i]*(short)b1[i];
9.        c[1] += (short)a1[i]*(short)b1[i];
Suppose there is one multiplication and addition per clock cycle, this code fills 64 clock cycles.

Then we used the SSSE3 instruction set to implement it instead:

```c
register __m128i a1,b1,b2,b3,b4,c,d1,d2,d3,d4;
// initialize a1 b1 b2 b3 b4 c here, where c is set to zeros/
d1 = __mm_add_epi32(__mm_srai_epi32(__mm_unpacklo_epi16(d1, d1), 16),
    __mm_srai_epi32(__mm_unpackhi_epi16(d1, d1), 16));
d2 = __mm_add_epi32(__mm_srai_epi32(__mm_unpacklo_epi16(d2, d2), 16),
    __mm_srai_epi32(__mm_unpackhi_epi16(d2, d2), 16));
d3 = __mm_hadd_epi32(d1, d2);
d1 = __mm_add_epi32(__mm_srai_epi32(__mm_unpacklo_epi16(d1, d1), 16),
    __mm_srai_epi32(__mm_unpackhi_epi16(d1, d1), 16));
d2 = __mm_add_epi32(__mm_srai_epi32(__mm_unpacklo_epi16(d2, d2), 16),
    __mm_srai_epi32(__mm_unpackhi_epi16(d2, d2), 16));
d4 = __mm_hadd_epi32(d1, d2);
c = __mm_hadd_epi32(d3, d4);
```

We stored the result in a 128bit register of “c”, where it is jointed by 4 integers. Take in consideration of the pipeline, this process may cost 12 or 13 clock cycles. So, the potential results we could get from this task are:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>CPU Clock Cycles</th>
<th>Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal C Coding</td>
<td>64</td>
<td>-</td>
</tr>
<tr>
<td>Using SSSE3 Instruction Set</td>
<td>13</td>
<td>~ 500%</td>
</tr>
</tbody>
</table>

As we know, there are many matrix calculations in the DNN process of speech recognition, if we optimize each one in our code like this, it will achieve better performance on the IA platform than ever. We have cooperated with ISV Unisound, which provides speech recognition services in China. Unisound used the DNN process with an improvement in performance of over 10% on ARM devices.

**Summary**

DNN is becoming the main algorithm in speech recognition. It has been selected by Google Now, Baidu Voice, Tencent Wechat, iFlytek Speech Service, Unisound Speech Service, and many others. At the same time, we have the SSSE3 instruction set which could help to optimize the speech recognition process, if
all of these applications begin using it, I believe the speech service will give us a better experience and more increased usage of IA platform.

**About the Author**

Li Alven graduated from Huazhong University of Science and Technology, where he majored in Computer Science and Information Security at 2007. He joined Intel in 2013 as a senior application engineer in the Developer Relations Division Mobile Enabling Team. He is focused on differentiation and innovative enabling on the IA platform, Speech Recognition Technology, tuning performance, etc.

**Notices**

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.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

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. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document 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.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: [http://www.intel.com/design/literature.htm](http://www.intel.com/design/literature.htm)

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark* and MobileMark*, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.