A Guide to Vectorization with Intel® C++ Compilers

Contents

1. Introduction

The goal of this Guide is to provide guidelines for enabling compiler vectorization capability in the Intel® C++ Compilers. This document is aimed at C/C++ programmers working on systems based on Intel® processors or compatible, non-Intel processors that support SIMD instructions such as Intel® Streaming SIMD Extensions (Intel® SSE). This includes Intel 64 and most IA-32 systems, but excludes systems based on Intel® Itanium® processors. The examples presented refer to Intel SSE, but many of the principles apply also to other SIMD instruction sets. While the examples used are specific to C++ programs, much of the concepts discussed are equally applicable to Fortran programs.

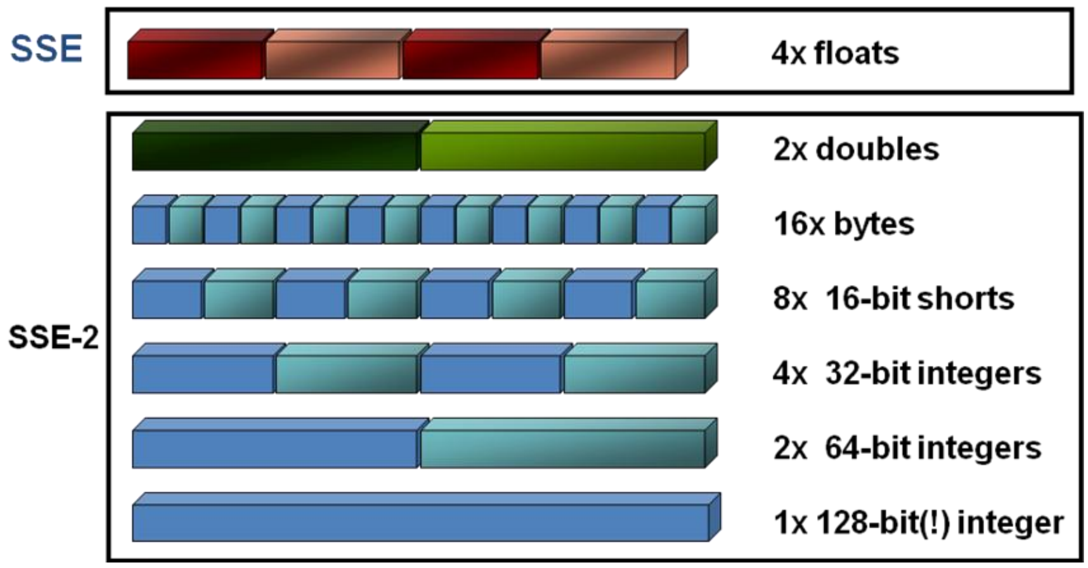
2. What is Vectorization in the Intel Compiler?

In this context, it is the unrolling of a loop combined with the generation of packed SIMD instructions by the compiler. Because the packed instructions operate on more than one data element at a time, the loop can execute more efficiently. It is sometimes referred to as auto-vectorization, to emphasize that the compiler identifies and optimizes suitable loops on its own, without requiring any special action by the programmer.

If you are familiar with this concept, you may choose to skip the additional technical background below and move on to the next section, "When does the compiler try to vectorize?"

Today's CPUs are highly parallel processors with different levels of parallelism. We find parallelism everywhere from the parallel execution units in a CPU core, up to the SIMD (Single Instruction, Multiple Data) instruction set and the parallel execution of multiple threads. The use of the Intel SSE instruction set, which is an extension to the x86 architecture, is called vectorization. In Computer science the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector (series of adjacent values) is called vectorization. SIMD instructions operate on multiple data elements in one instruction and make use of the 128-bit SIMD floating-point registers.

Intel originally added eight new 128-bit registers known as XMM0 through XMM7. For the 64-bit extensions additional eight registers XMM8-XMM15 were added. Programmers can exploit vectorization to speedup certain parts of their code. Writing vectorized code can take additional time but is mostly worth the effort, because the performance increase may be substantial. One major research topic in computer science is the search for methods of automatic vectorization: seeking methods that would allow a compiler to convert scalar algorithms into vectorized algorithms without human assistance. The Intel® Compiler can automatically generate Intel SSE instructions. This Guide will focus on using the Intel® Compiler to automatically generate SIMD code, a feature which will be referred as auto-vectorization henceforth. We also give some guidelines that will help the compiler to auto-vectorize. However, in some cases, certain keywords or directives have to be applied in the code for auto-vectorization to occur.

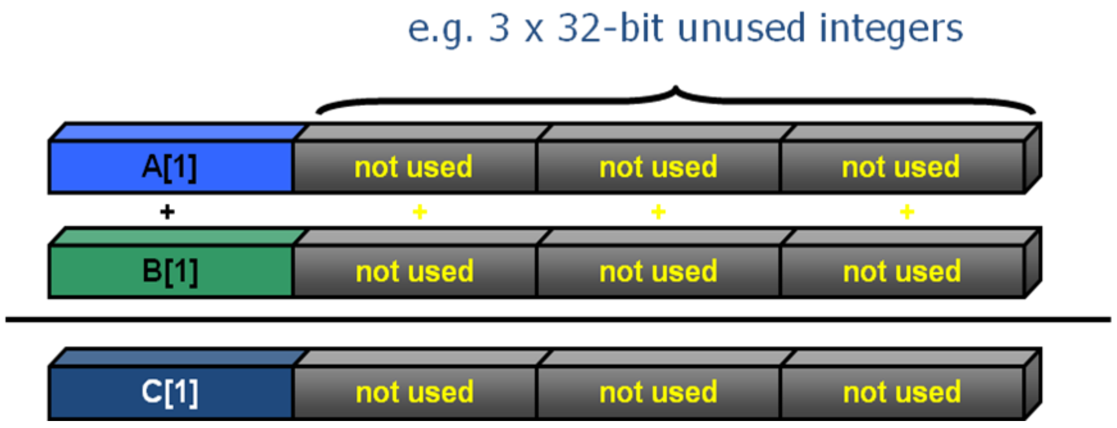


So where does the vectorization speedup come from? Let's look at a sample code fragment, where a, b and c are integer arrays:

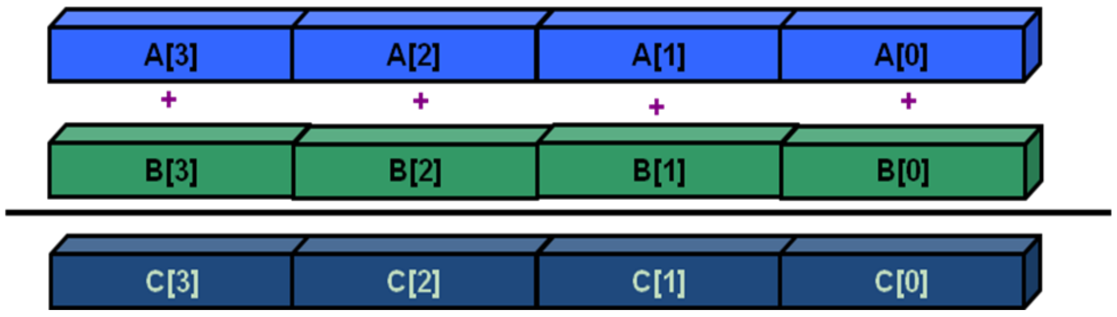
```
for(i=0;i<=MAX;i++)

   c[i]=a[i]+b[i];
```

If vectorization is not enabled (e.g. because we used /Od, /O1 or /Qvec-), the compiler will do something like this:



So we have a lot of unused space in our SIMD registers which could hold three additional integers. If vectorization is enabled, the compiler may use the additional registers to perform 4 additions in a single instruction:



2.1 When does the compiler try to vectorize?

The compiler will look for vectorization opportunities whenever you compile at default optimization (-O2) or higher. These options are available for both Intel® and non-Intel microprocessors but they may result in more optimizations for Intel microprocessors than for non-Intel microprocessors. To allow comparisons of vectorized with unvectorized code, vectorization may be disabled with the switch /Qvec- (Windows*) or -no-vec (Linux* or Mac OS* X). The compiler will not vectorize at –O1 or below.

2.2 How do I know whether a loop was vectorized?

The vectorization report (described more fully in the Vectorization Reports section) may be enabled with the switch /Qvec-report (Windows) or –vec-report (Linux* or Mac OS* X). This will result in a one line message for every loop that is vectorized, of the form:

```
> icl /Qvec-report MultArray.c

MultArray.c(92): (col.  5) remark: LOOP WAS VECTORIZED.
```

The source line number (92 in the above example) refers to the beginning of the loop. From the IDE, select Project \ Properties \ C/C++ \ Diagnostics \ Optimization Diagnostic Level as "minimum" and Optimization Diagnostic Phase as "The High Performance Optimizer phase, /Qopt-report-phase:hpo".

By choosing instead /Qvec-report2 (-vec-report2), or "medium" instead of "minimum" in the IDE, you will also get a diagnostic message for every loop that was not vectorized, with a brief explanation of why not.

2.3 What difference does it make?

Vectorization should improve loop performance in general. To illustrate this point, please open an Intel® Compiler command line window. For example, from the Start menu on Windows, select All Programs\Intel Parallel Composer\Command prompt\Intel64 Visual Studio 2008 mode. On Linux or Mac OS X, source the initialization script for your Linux compiler version in the compiler bin/intel64 or bin/ia32 directories as described in the corresponding release notes.

Navigate to the "example1" directory. The small application multiplies a vector by a matrix using the following loop:

```
    for (j = 0;j < size2; j++) {
      b[i] += a[i][j] * x[j];
    }
```

Now build and run the application by typing (on Windows):

```
icl /O2 /Qvec- MultArray.c /FeNoVectMult

NoVectMult

icl /O2 /Qvec-report MultArray.c /FeVectMult

VectMult
```

On Linux or Mac OS X, type equivalently:

```
icc -O2 -no-vec  MultArray.c -o NoVectMult

./NoVectMult

icc -O2 -vec-report MultArray.c -o VectMult

./VectMult
```

Compare the timing of the two runs. You may see that the vectorized version ran faster.

3. What sort of loops can be vectorized?

To be vectorizable, loops must meet the following criteria:

1. **Countable**
   The loop trip count must be known at entry to the loop at runtime, though it need not be known at compile time. (I.e., the trip count can be a variable, but the variable must remain constant for the duration of the loop). This implies that exit from the loop must not be data-dependent.

2. **Single entry and single exit**
   This is implied by countable. Here is an example of a loop that is not vectorizable due to a second data-dependent exit:

```
    void no_vec(float a[], float b[], float c[])
    {
         int i = 0.;
         while (i < 100) {
           a[i] = b[i] * c[i];
    //   this is a data-dependent exit condition:
           if (a[i] < 0.0)
              break;
           ++i;
         }
    }

> icc -c -O2 -vec-report2 two_exits.cpp
two_exits.cpp(4) (col.  9): remark: loop was not vectorized:
nonstandard loop is not a vectorization candidate.
```

3. **Straight-line code** Because SIMD instructions perform the same operation on data elements from multiple iterations of the original loop, it is not possible for different iterations to have different control flow, i.e., they must not branch. Thus switch statements are not allowed. However, "if" statements are allowed if they can be implemented as masked assignments, which is usually the case. The calculation is performed for all data elements, but the result is stored only for those elements for which the mask evaluates to true. Thus, the following example may be vectorized:

```
    #include <math.h>
    void quad(int length, float *a, float *b,
               float *c, float *restrict x1, float *restrict x2)
       {
         for (int i=0; i<length; i++) {
           float s = b[i]*b[i] - 4*a[i]*c[i];
           if ( s >= 0 ) {
              s = sqrt(s) ;
              x2[i] = (-b[i]+s)/(2.*a[i]);
              x1[i] = (-b[i]-s)/(2.*a[i]);
           }
           else {
              x2[i] = 0.;
              x1[i] = 0.;
           }
         }
       }
       > icc -c -restrict -vec-report2 quad.cpp
       quad5.cpp(5) (col.  3): remark: LOOP WAS VECTORIZED.
```

4. **The innermost loop of a nest** The only exception is if an original outer loop is transformed into an inner loop as a result of some other prior optimization phase, such as unrolling, loop collapsing or interchange.

5. **No function calls** The two major exceptions are for intrinsic math functions and for functions that may be inlined. Even a print statement is sufficient to render a loop unvectorizable. The vectorization report message is typically "nonstandard loop is not a vectorization candidate".

Intrinsic math functions such as sin(), log(), fmax(), etc., are allowed, since the compiler runtime library contains vectorized versions of these functions. Table 1 lists these functions. Most exist in both float and double versions.

Table 1. Functions for which the compiler has a vectorized version.

acos	ceil	fabs	round
acosh	Cos	floor	sin
asin	Cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	Erfc	log	tan
atan2	Erfinv	log10	tanh
atanh	Exp	log2	trunc
cbrt	exp2	pow	

Thus the loop in the following example may be vectorized, since sqrtf() is vectorizable and func() gets inlined. Inlining is enabled at default optimization for functions in the same source file. (An inlining report may be obtained by setting /Qopt-report-phase ipo_inl (-opt-report-phase ipo_inl).)

```
float func(float x, float y, float xp, float yp) {
  float denom;
  denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
  denom = 1./sqrtf(denom);
  return denom;
}

float trap_int
      (float y, float x0, float xn, int nx, float xp, float yp) {

float x, h, sumx;
int i;

  h = (xn-x0) / nx;
  sumx = 0.5*( func(x0,y,xp,yp) + func(xn,y,xp,yp) );
  for (i=1;i<nx;i++) {
    x = x0 + i*h;
    sumx = sumx + func(x,y,xp,yp);
  }
  sumx = sumx * h;

  return sumx;
}
> icc -c -vec-report2 trap_integ.c
trap_int.c(16) (col.  3): remark: LOOP WAS VECTORIZED.
```

4. Obstacles to vectorization

The following do not always prevent vectorization, but frequently either prevent it or cause the compiler to decide that vectorization would not be worthwhile.

4.1 Non-contiguous Memory Accesses

Four consecutive ints or floats, or two consecutive doubles, may be loaded directly from memory in a single SSE instruction. But if the four ints are not adjacent, they must be loaded separately using multiple instructions, which is considerably less efficient. The most common examples of non-contiguous memory access are loops with non-unit stride or with indirect addressing, as in the examples below. The compiler rarely vectorizes such loops, unless the amount of computational work is large compared to the overhead from non-contiguous memory access.

```
   // arrays accessed with stride 2
      for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[i];

   // inner loop accesses a with stride SIZE
      for (int j=0; j<SIZE; j++)  {
         for (int i=0; i<SIZE; i++)   b[i] += a[i][j] * x[j];
    }

   // indirect addressing of x using index array
      for (int i=0; i<SIZE; i+=2)  b[i] += a[i] * x[index[i]];
```

The typical message from the vectorization report is

```
"vectorization possible but seems inefficient"
```

Although indirect addressing may also result in

```
"Existence of vector dependence"
```

4.2 Data Dependencies

Vectorization entails changes in the order of operations within a loop, since each SIMD instruction operates on several data elements at once. Vectorization is only possible if this change of order does not change the results of the calculation.

- The simplest case is when data elements that are written (stored to) do not appear in any other iteration of the individual loop. In this case, all the iterations of the original loop are independent of each other, and can be executed in any order, without changing the result. The loop may be safely executed using any parallel method, including vectorization. All the examples considered so far fall into this category.

- **Read-after-write:** When a variable is written in one iteration and read in a subsequent iteration, there is a "read-after-write" dependency, also known as a flow dependency, as in this example:

```
A[0]=0;

for (j=1; j<MAX; j++)  A[j]=A[j-1]+1;

// this is equivalent to:

A[1]=A[0]+1;   A[2]=A[1]+1;   A[3]=A[2]+1; A[4]=A[3]+1;
```

The above loop cannot be vectorized safely because if the first two iterations are executed simultaneously by a SIMD instruction, the value of A[1] may be used by the second iteration before it has been calculated by the first iteration which could lead to incorrect results.

- **Write-after-read:** When a variable is read in one iteration and written in a subsequent iteration, this is a write-after-read dependency, (sometimes also known as an anti-dependency), as in the following example:

```
for (j=1; j<MAX; j++)  A[j-1]=A[j]+1;

// this is equivalent to:

A[0]=A[1]+1;   A[1]=A[2]+1;   A[2]=A[3]+1; A[3]=A[4]+1;
```

This is not safe for general parallel execution, since the iteration with the write may execute before the iteration with the read. However, for vectorization, no iteration with a higher value of j can complete before an iteration with a lower value of j, and so vectorization is safe (i.e., gives the same result as non-vectorized code) in this case. The following example, however, may not be safe, since vectorization might cause some elements of A to be overwritten by the first SIMD instruction before being used for the second SIMD instruction.

```
for (j=1; j<MAX; j++)  {
```

```
          A[j-1]=A[j]+1;

          B[j]=A[j]*2;

     }

     // this is equivalent to:

     A[0]=A[1]+1;   A[1]=A[2]+1;   A[2]=A[3]+1; A[3]=A[4]+1;
```

- **Read-after-read:** These situations aren't really dependencies, and do not prevent vectorization or parallel execution. If a variable is not written, it does not matter how often it is read.

- **Write-after-write**: Otherwise known as 'output', dependencies, where the same variable is written to in more than one iteration, are in general unsafe for parallel execution, including vectorization.

However, there is a very important exception that apparently contains all of the above types of dependency:

```
     sum=0;

     for (j=1; j<MAX; j++)  sum = sum + A[j]*B[j]
```

Although sum is both read and written in every iteration, the compiler recognizes such reduction idioms, and is able to vectorize them safely. The loop in example 1 was another example of a reduction, with a loop-invariant array element in place of a scalar.

These types of dependencies between loop iterations are sometimes known as loop-carried dependencies.

The above examples are of proven dependencies. However, the compiler cannot safely vectorize a loop if there is even a potential dependency. In the following example,

```
   for (i = 0; i < size; i++) {
     c[i] = a[i] * b[i];
   }
```

the compiler must ask itself whether, for some iteration i, c[i] might refer to the same memory location as a[i] or b[i] for a different iteration. (Such memory locations are sometimes said to be "aliased"). For example, if a[i] pointed to the same memory location as c[i-1], there would be a read-after-write dependency as in the earlier example. If the compiler cannot exclude this possibility, it will not vectorize the loop (at least, not without help, as discussed under "Helping the compiler to vectorize" under #pragma ivdep and the restrict keyword).

5. Guidelines for writing vectorizable code

5.1 General Guidelines

Here are some general guidelines for helping the compiler better vectorizer your code.

- Prefer countable single entry and single exit "for" loops. Avoid complex loop termination conditions – the loop lower bound and upper bound must be invariant within the loop. It's OK for it to be a function of the outer loop indices, for the innermost loop in a nest of loops.
- Write straight line code (avoid branches such as switch, goto or return statements, most function calls, or "if" constructs that can't be treated as masked assignments).
- Avoid dependencies between loop iterations, or at least, avoid read-after-write dependencies
- Prefer array notation to the use of pointers. C programs in particular impose very few restrictions on the use of pointers; aliased pointers may lead to unexpected dependencies. Without help, the compiler often cannot tell whether it is safe to vectorize code containing pointers.
- Use the loop index directly in array subscripts where possible, instead of incrementing a separate counter for use as an array address.
- Use efficient memory accesses
    - Favor inner loops with unit stride
    - Minimize indirect addressing
    - Align your data to 16 byte boundaries (for Intel® SSE instructions)
    - Align your data to 32 byte boundaries (for Intel® Advanced Vector Extensions (Intel® AVX))

Choose carefully a suitable data layout. Most multimedia extension instruction sets are rather sensitive to alignment. The data movement instructions of SSE/SSEx, for example, operate much more efficiently on data that is aligned at a 16-byte boundary in memory. Therefore, the success of a vectorizing compiler also depends on its ability to select an appropriate data layout that, in combination with code restructuring (like loop peeling), will result in aligned memory accesses throughout the program. If your compilation targets the Intel® AVX instruction set, you should try to align data on a 32-byte boundary. This may result in improved performance.

5.2 Use aligned data structures

Data structure alignment is the adjustment of any data object in relation with other objects. The Intel® Compilers may align individual variables to start at certain addresses to speed up memory access. Misaligned memory accesses can incur large performance losses on certain target processors that do not support them in hardware. Alignment is a property of a memory address, expressed as the numeric address modulo a power of 2. In addition to its address, a single datum also has a size. A datum is called naturally aligned if its address is aligned to its size and misaligned otherwise. For example, an 8-byte floating-point datum is naturally aligned if the address used to identify it is aligned to 8. A data structure is a way of storing data in a computer so that it can be used efficiently. Often a carefully chosen data structure will allow a more efficient algorithm to be used. A well-designed data structure allows a variety of critical operations to be performed, using as little resources, both execution time and memory space, as possible.

```
struct MyData{
        short   Data1;
        short   Data2;
        short   Data3;};
```

If the type ***short*** is stored in two bytes of memory then each member of the data structure depicted above would be aligned to a boundary of 2 bytes. Data1 would be at offset 0, Data2 at offset 2 and Data3 at offset 4. The size of this structure then would be 6 bytes. The type of each member of the structure usually has a required alignment, meaning that it will, unless otherwise requested by the programmer, be aligned on a pre-determined boundary. In cases where the compiler has taken sub-optimal alignment decisions, however, the programmer can use the directive `declspec(align(base,offset))`, where `0 <= offset < base` and base is a power of two, to allocate a data structure at offset from a certain base

Suppose, as an example, that most of the execution time of an application is spent in a loop of the following form:

```
double a[N], b[N];

.  .  .

for (i = 0; i < N; i++){

  a[i+1] = b[i] * 3;

}
```

If the first element of both arrays is aligned at a 16-byte boundary, then either an unaligned load of elements from "`b`" or an unaligned store of elements into "`a`", has to be used after vectorization. (Note that in this case, peeling off an iteration would not help.) However, the programmer can enforce the alignment shown below, which will result in two aligned access patterns after vectorization (assuming an 8-byte size for doubles):

```
_declspec(align(16, 8))  double a[N];

_declspec(align(16, 0))  double b[N];

/* or simply "align(16)"  */
```

If pointer variables are used, the compiler is usually not able to determine the alignment of access patterns at compile time. Consider the following simple fill function:

```
void fill(char *x) {
  int i;
  for (i = 0; i < 1024; i++){
       x[i] = 1;
  }
}
```

Without more information, the compiler cannot make any assumption on the alignment of the memory region accessed by this loop. At this point, the compiler may decide to vectorize this loop using unaligned data movement instructions or, as done by the Intel C/C++ compiler, generate the run-time alignment optimization shown here:

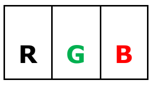
```
peel = x & 0x0f;
if (peel != 0) {
  peel = 16 - peel;
  /* runtime peeling loop */
  for (i = 0; i < peel; i++){
 x[i] = 1;
  }
}
/* aligned access */
for (i = peel; i < 1024; i++){
x[i] = 1;
}
```

Run-time optimization provides a generally effective way to obtain aligned access patterns at the expense of a slight increase in code size and testing. If incoming access patterns are guaranteed to be aligned at a 16-byte boundary, however, the programmer can avoid this overhead with the hint `__assume_aligned(x, 16);` in the function to convey this information to the compiler. Note that this hint must also be used with care because incorrect usage of aligned data movements will result in an exception for SSE.

5.3 Prefer Structure of Arrays (SoA) over Array of Structures (AoS)

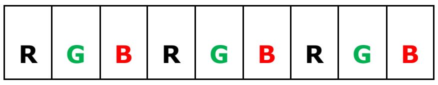
The most common and likely well-known data structure is the array, which contains a contiguous collection of data items that can be accessed by an ordinal index. This data can be organized as an Array Of Structures (AOS) or a Structure Of Arrays (SOA). While AOS organization is excellent for encapsulation it can be poor for use of vector processing. Selecting appropriate data structures can also make vectorization of the resulting code more effective. To illustrate this point, compare the traditional array-of-structures (AoS) arrangement for storing the r, g, b components of a set of three-dimensional points with the alternative structure-of-arrays (SoA) arrangement for storing this set.

Point Structure :

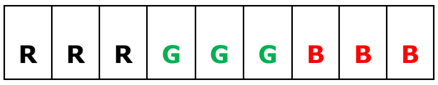


```
struct Point{ //AoS
    float r;
    float r;
    float g;
}
```

Structure of Arrays:



```
struct Points{ //SoA
    float* x;
    float* y;
    float* z;
}
```



With the AoS arrangement, a loop that visits all components of an RGB point before moving to the next point exhibits a good locality of reference because all elements in fetched cache lines are utilized. The disadvantage of the AoS arrangement is that each individual memory reference in such a loop exhibits a non-unit stride, which, in general, adversely affects vector performance. Furthermore, a loop that visits only one component of all points exhibits less satisfactory locality of reference because many of the elements in the fetched cache lines remain unused. In contrast, with the SoA arrangement the unit-stride memory references are more amenable to effective vectorization and still exhibit good locality of reference within each of the three data streams. Consequently, an application that uses the SoA arrangement may ultimately outperform an application based on the AoS arrangement when compiled with a vectorizing compiler, even if this performance difference is not directly apparent during the early implementation phase.

So before you start vectorization try to follow some simple rules:

- Make your data structures vector-friendly

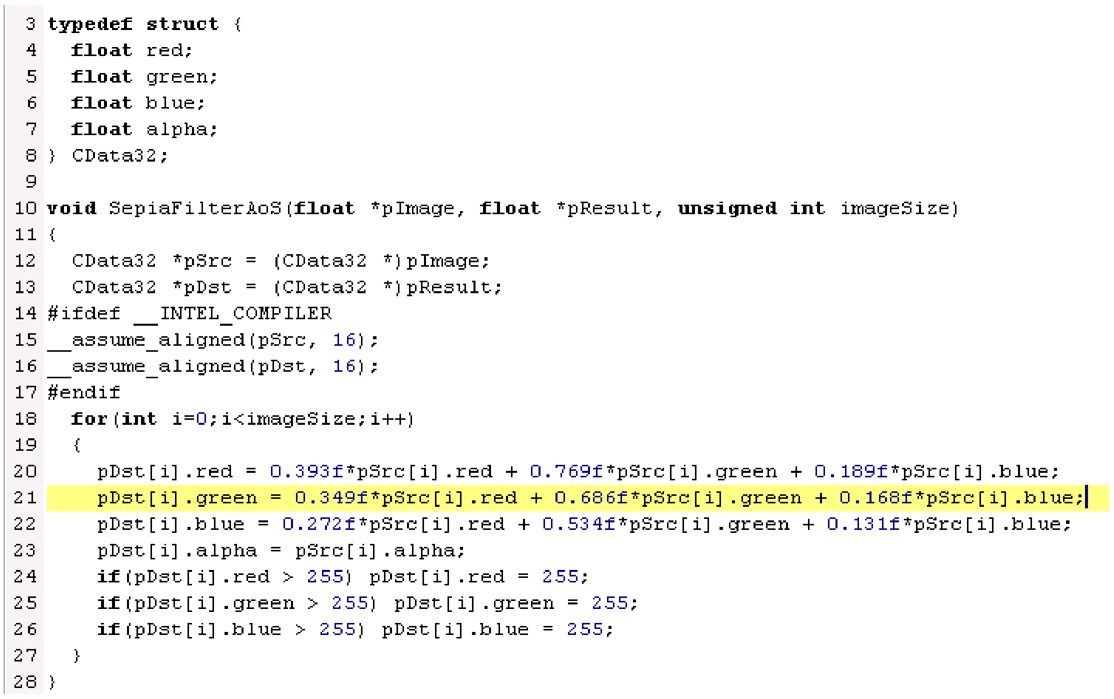
- Make sure that inner loop indices correspond to the outermost (last) array index in your data (row-major order).

- Prefer structure of arrays over array of structures

For instance when dealing with 3 dimensional coordinates, use 3 separate arrays for each component (SOA), instead of using one array of 3-component structures (AOS). To avoid dependencies between loops that will eventually prevent vectorization, it is therefore recommended to rather use 3 separate arrays for each component (SOA), instead of one array of 3-component structures (AOS), when processing 3 dimensional coordinates. In AOS, each iteration produces one result by computing XYZ_, but can at best use only 75% of the SSE unit, because the fourth component is not used. Sometimes, it may use only one component (25%). In SOA, each iteration produces 4 results by computing XXXX, YYYY and ZZZZ and using 100% of the SSE unit. A drawback for SOA is that your code will likely be 3 times as long. On the other hand, the compiler might not be able to vectorize AOS code at all.

If your original data layout is in AOS format, you may even want to consider a conversion to SOA on the fly, before the critical loop. If it gets vectorized, it may be worth the effort!

AoS Example:



```
 3 typedef struct {
 4   float red;
 5   float green;
 6   float blue;
 7   float alpha;
 8 } CData32;
 9
10 void SepiaFilterAoS(float *pImage, float *pResult, unsigned int imageSize)
11 {
12   CData32 *pSrc = (CData32 *)pImage;
13   CData32 *pDst = (CData32 *)pResult;
14 #ifdef __INTEL_COMPILER
15 __assume_aligned(pSrc, 16);
16 __assume_aligned(pDst, 16);
17 #endif
18   for(int i=0;i<imageSize;i++)
19   {
20     pDst[i].red = 0.393f*pSrc[i].red + 0.769f*pSrc[i].green + 0.189f*pSrc[i].blue;
21     pDst[i].green = 0.349f*pSrc[i].red + 0.686f*pSrc[i].green + 0.168f*pSrc[i].blue;
22     pDst[i].blue = 0.272f*pSrc[i].red + 0.534f*pSrc[i].green + 0.131f*pSrc[i].blue;
23     pDst[i].alpha = pSrc[i].alpha;
24     if(pDst[i].red > 255) pDst[i].red = 255;
25     if(pDst[i].green > 255) pDst[i].green = 255;
26     if(pDst[i].blue > 255) pDst[i].blue = 255;
27   }
28 }
```

`icl /c AoSvsSoA.cpp /DAOS /Qvec-report2`

```
AoSvsSoA.cpp

AoSvsSoA.cpp(18): (col. 3) remark: loop was not vectorized:
existence of vector dependence.
```

SoA Example:

```
34 #ifdef SOA
35
36 typedef struct {
37   float red[4];
38   float green[4];
39   float blue[4];
40   float alpha[4];
41 } CData32T4;
42
43 const float fIntensity = 0.0234;
44 extern CData32T4 * GetCurrentSrcPointer32T4(void);
45
46
47 void CrossfadeTransposedFloatKernel()
48 {
49   CData32T4 *pSrc1 = GetCurrentSrcPointer32T4();
50   CData32T4 *pSrc2 = GetCurrentSrcPointer32T4();
51   CData32T4 *pDst  = GetCurrentSrcPointer32T4();
52 #ifdef __INTEL_COMPILER
53 __assume_aligned(pSrc1, 16);
54 __assume_aligned(pSrc2, 16);
55 __assume_aligned(pDst, 16);
56 #endif
57 #pragma ivdep
58   for(int j=0;j<4;j++)
59   {
60     pDst->red[j] = (1.0f-fIntensity)*pSrc1->red[j] + fIntensity*pSrc2->red[j];
61     pDst->green[j] = (1.0f-fIntensity)*pSrc1->green[j] + fIntensity*pSrc2->green[j];
62     pDst->blue[j] = (1.0f-fIntensity)*pSrc1->blue[j] + fIntensity*pSrc2->blue[j];
63     pDst->alpha[j] = pSrc1->alpha[j];
64   }
65 }
66 #endif // SOA
```

```
icl /c AoSvsSoA.cpp /DSOA /Qvec-report2
```

```
    AoSvsSoA.cpp

    AoSvsSoA.cpp(58): (col.  3) remark: LOOP WAS VECTORIZED.
```

5.4 Vectorization Guidelines for Data Structures

- Use the smallest data types that give the needed precision, to maximize potential SIMD width. (If only 16-bits are needed, using a short rather than an int can make the difference between eight-way or four-way SIMD parallelism, respectively.)

- Avoid mixing vectorizable data types in the same loop (except for integer arithmetic on array subscripts). Vectorization of type conversions may be either unsupported or inefficient. The latest compilers, including the version 12 of the Intel® C++ Compilers for Windows* and Linux*, have greatly improved and can auto-vectorize mixed type loops. But avoid if possible. The example below illustrates the mixing of data types, which may prevent auto-vectorization.

```
void mixed(float *restrict a, double *restrict b, float *c)

{

        for(int i = 1; i < 1000; ++i)

        {

                b[i] = b[i] - c[i];

                a[i] = a[i] + b[i];

        }

}
```

- Avoid operations not supported in SIMD hardware. Arithmetic with (80 bit) long doubles on Linux, and the remainder operator "%" are examples of operations not supported in SIMD hardware.

- Use all the instruction sets available for your processor. Use the appropriate command line option for your processor type, or select the appropriate IDE option under "Project / Properties / C/C++ / Code Generation / Intel Processor-Specific Optimization" (/QxSSE4.1, /QxSSE4.2, etc., on Windows*) and (-xSSE4.1, -xSSE4.2, etc, on Linux* or Mac OS* X), if your application will run only on Intel processors, or "Project / Properties / C/C++ / Code Generation / Enable Enhanced Instruction Set" (/arch:SSE2, /arch:SSE3 on Windows) and (-msse2, -msse3 on Linux or Mac OS X), if your application may run on compatible, non-Intel processors.

  If your application will run only on the processor type on which it was built, you may simply choose the command line option /QxHost (Windows) or –xhost (Linux or Mac OS X), or in the IDE, select "Project / Properties / C/C++ / Code Generation / Intel Processor-Specific Optimization / The processor performing the compilation (/QxHost)".   This option is available for both Intel® and non-Intel microprocessors but it may perform more optimizations for Intel microprocessors than it performs for non-Intel microprocessors.

  For more details about the available processor-specific options, see the article at http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/

- Vectorizing compilers usually have some built-in efficiency heuristics to decide whether vectorization is likely to improve performance. The Intel® C/C++ Compiler will disable vectorization of loops with many unaligned or non-unit stride data access patterns. However, if experimentation reveals that vectorization will still improve performance, the programmer can override this behaviour with a "#pragma vector always" hint before the loop, which asks the compiler to vectorize any loop regardless of the outcome of the efficiency analysis (provided, of course, that vectorization is safe).

6. Vectorization Reports

The Intel® Compiler provides a vectorization report option that provides two important kinds of information. First, the vectorization report will inform you which loops in your code are being vectorized. The end result of a vectorized loop is an instruction stream for that loop that contains packed SIMD instructions. Secondly and at least as important, is report information about why the compiler did NOT vectorize a loop. This information assists a programmer by highlighting the barriers that the compiler finds to vectorization. With the Intel® compiler, one must enable the vectorization reporting mechanism, since it is not enabled by default. In the Microsoft* Visual Studio* IDE (Integrated Development Environment), this is done by selecting from the Project menu "Properties \ C/C++ \ Diagnostics \ Optimization Diagnostic Phase" as "The High Performance Optimizer phase, /Qopt-report-phase:hpo" and "Optimization Diagnostic Level" as "medium". This corresponds to n=2 below.

The -vec-report (Linux* or Mac OS X) or /Qvec-report (Windows*) options direct the compiler to generate the vectorization reports with different levels of information. The vectorization report option, -vec-report=<n>, uses the argument <n> to specify the information presented; from no information at -vec-report=0 to very verbose information at -vec-report ≥ 3. The arguments to -vec-report are:

n=0: No diagnostic information

n=1: Loops successfully vectorized

n=2: Loops not vectorized - and the reason why not

n=3: Adds dependency Information

n=4: Reports only non-vectorized loops

n=5: Reports only non-vectorized loops and adds dependency info

6.1 Examples of Vectorization Report Messages

"**Low trip count**": The loop does not have sufficient iterations for vectorization to be worthwhile

"Not Inner Loop": Only the inner loop of a loop nest may be vectorized.

“Existence of vector dependence”: The compiler did not vectorize the loop because of a proven or potential dependence. If you are sure that any potential dependencies are not in fact realized, you may invite the compiler to ignore them with **#pragma ivdep**.

"vectorization possible but seems inefficient": The compiler thinks that vectorization may not improve the performance of this loop. You may use **#pragma vector always** to override the compiler’s performance assessment, and ask the compiler to vectorize anyway if it is safe to do so.

“Condition may protect exception”: When the compiler tries to vectorize a loop containing an IF statement, it typically evaluates the RHS expressions for all values of the loop index, but only makes the final assignment in those cases where the conditional evaluates to TRUE. In some cases, the compiler may not vectorize out of concern that the condition may be protecting against accessing an illegal memory address.

An IVDEP pragma may be used to reassure the compiler that the conditional is not protecting against a memory exception in such cases.

“**data type unsupported on given target architecture**”: For example, this message might occur when compiling a loop containing complex arithmetic for a target processor that supported only the SSE2 instruction set. SSE3 instructions are needed for effective vectorization of arithmetic involving complex data types.

“**Statement cannot be vectorized**”: Certain statements, such as switch statements, can’t be vectorized.

“Subscript too complex”: An array subscript may be too complicated for the compiler to decipher the memory access pattern. Try to write subscripts as an explicit function of the main loop counter.

“Unsupported Loop Structure”, “Top test could not be found”: Loops that don’t fulfill the requirements of countability, single entry and exit, etc., may generate these messages.

“Operator unsuited for vectorization”: Certain operators, such as the “%” (modulus) operator, can’t be vectorized.

6.2 Help the compiler to vectorize

Here are several ways in the compiler can be provided with additional information that enable it to better vectorize the loop.

6.2.1 Pragmas

Please see the compiler user and reference guide for more information, but here are a few of the key ones.

- **#pragma ivdep** may be used to tell the compiler that it may safely ignore any potential data dependencies. (The compiler will not ignore proven dependencies). Use of this pragma when there are in fact dependencies may lead to incorrect results.

There are cases, where the compiler can't tell by a static analysis that it is safe to vectorize, but you as a developer would know. Consider the following loop:

```
void copy(char *cp_a, char *cp_b, int n) {

  for (int i = 0; i < n; i++) {

      cp_a[i] = cp_b[i];

  }

}
```

Without more information, a vectorizing compiler must conservatively assume that the memory regions accessed by the pointer variables cp_a and cp_b may (partially) overlap, which gives rise to potential data dependencies that prohibit straightforward conversion of this loop into SIMD instructions. At this point, the compiler may decide to keep the loop serial or, as done by the Intel C/C++ compiler, generate a run-time test for overlap, where the loop in the true-branch can be converted into SIMD instructions:

```
if (cp_a + n < cp_b || cp_b + n < cp_a)

  /* vector loop */

  for (int i = 0; i < n; i++) cp_a[i] = cp_b [i];

else

  /* serial loop */

  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];
```

Run-time data-dependence testing provides a generally effective way to exploit implicit parallelism in C or C++ code at the expense of a slight increase in code size and testing overhead. If the function copy is only used in specific ways, however, the programmer can assist the vectorizing compiler as follows. First, if the function is mainly used for small values of n or for overlapping memory regions, the programmer can simply prevent vectorization and, hence, the corresponding run-time overhead by inserting a #pragma novector hint before the loop. Conversely, if the loop is guaranteed to operate on non-overlapping memory regions, this information can be propagated to the compiler by means of a #pragma ivdep hint before the loop, which informs the compiler that conservatively assumed data dependencies that prevent vectorization can be ignored. This will result in vectorization of the loop without run-time data-dependence testing.

```
#pragma ivdep
void copy(char *cp_a, char *cp_b, int n) {

  for (int i = 0; i < n; i++) {

       cp_a[i] = cp_b[i];

  }

}
```

Or use the restrict keyword, see the keyword section below.

- **#pragma loop count (n)** may be used to advise the compiler of the typical trip count of the loop. This may help the compiler to decide whether vectorization is worthwhile, or whether or not it should generate alternative code paths for the loop.
- **#pragma vector always** asks the compiler to vectorize the loop if it is safe to do so, whether or not the compiler thinks that will improve performance.
- **#pragma vector align** asserts that data within the following loop is aligned (to a 16 byte boundary, for SSE instruction sets).
- **#pragma novector** asks the compiler not to vectorize a particular loop
- **#pragma vector nontemporal** gives a hint to the compiler that data will not be reused, and therefore to use streaming stores that bypass cache.

6.2.2 *restrict* Keyword

The ***restrict*** keyword may be used to assert that the memory referenced by a pointer is not aliased, i.e. that it is not accessed in any other way. The keyword requires the use of either the /Qrestrict (-restrict) for either .c or .cpp files, or /Qstd=c99 compiler option for .c files.

The example under #pragma ivdep above can also be handled using the restrict keyword.

The programmer may use the restrict keyword in the declarations of **cp_a** and **cp_b**, as shown below, to inform the compiler that each pointer variable provides exclusive access to a certain memory region. The restrict qualifier in the argument list will let the compiler know that there are no other aliases to the memory to which the pointers point. In other words, the pointer for which it is used provides the only means of accessing the memory in question in the scope in which the pointers live. If the loop vectorizes without using the restrict keyword, the compiler will be checking for aliasing at runtime. The compiler will not do any runtime checks for aliasing if you use the restrict keyword. Using such a language extension requires an extra compiler switch, such as -Qrestrict for the Intel C/C++ compiler.

```
void copy(char * restrict cp_a, char * restrict cp_b, int n) {

  for (int i = 0; i < n; i++) cp_a[i] = cp_b[i];

}
```

Another example is the following loop which may also not get vectorized, because of a potential aliasing problem between pointers a, b and c:

```
// potential unsupported loop structure
void add(float *a, float *b, float *c) {
    for (int i=0; i<SIZE; i++) {
        c[i] += a[i] + b[i];
    }
}
```

If the restrict keyword is added to the parameters, the compiler will trust you, that you will not access the memory in question with any other pointer and vectorize the code properly:

```
// let the compiler know, the pointers are safe with restrict
void add(float * restrict a, float * restrict b, float * restrict c) {
    for (int i=0; i<SIZE; i++) {
        c[i] += a[i] + b[i];
    }
}
```

Note, however, that both the loop-specific #pragma ivdep hint, as well as the pointer variable-specific restrict hint must be used with care, since incorrect usage may change the semantics intended in the original program.

The down-side of using restrict is, that not all compilers support this keyword, so your source code may lose portability. If you care about source code portability you may want to consider using the compiler options /Qalias-args- (-fargument-noalias) or /Qansi-alias (**-ansi-alias**) instead. The first option tells the compiler that function arguments cannot alias each other in the entire file, but they can alias global storage. The latter tells the compiler that the different types do not alias each other. This method is also convenient in case the exclusive access property holds for pointer variables that are used in a large portion of code with many loops, because it avoids the need to annotate each of the vectorizable loops individually. However, compiler options work globally, so you have to make sure it does not cause harm at other code fragments.

- __declspec(align(16)) may be used to ensure that data is aligned to a 16 byte boundary.

6.2.3 Switches

- Interprocedural optimization may be enabled by /Qipo (-ipo) across source files. This may give the compiler additional information about a loop, such as trip counts, alignment or data dependencies. It may also allow inlining of function calls. In the example under "Which loops can be vectorized", "no function calls", if the functions func() and trap_int() appear in separate source files, the loop in trap_int() may still be vectorized by compiling both with /Qipo (-ipo).

- Disambiguation of pointers and arrays. The switch /Oa (Windows) or –fno-alias (Linux or Mac OS X) may be used to assert there is no aliasing of memory references, that is, that the same memory location is not accessed via different arrays or pointers. Other switches make more limited assertions, for example, /Qalias-args- (-fargument-noalias) asserts that function arguments cannot alias each other (overlap). /Qansi-alias (-ansi-alias) allows the compiler to assume strict adherence to the aliasing rules in the ISO C standard. Use of these switches is the programmer's responsibility; use when memory is in fact aliased may lead to incorrect results.

- High level loop optimizations (HLO) may be enabled with /O3 (-O3). These additional loop optimizations may make it easier for the compiler to vectorize the transformed loops. This option is available for both Intel® and non-Intel microprocessors but they may result in more optimizations for Intel microprocessors than for non-Intel microprocessors.

  The HLO report, obtained with /Qopt-report-phase:hlo (-opt-report-phase hlo) or the corresponding IDE selection, may tell you whether some of these additional transformations occurred.

7. Let the compiler help you to help the compiler to vectorize

7.1 Guided Auto-Parallelization

The Guided Auto-Parallelization (GAP) option "/Qguide" causes the compiler to generate diagnostics suggesting ways to improve Auto-Vectorization, Auto-Parallelization, and data transformation. The "/Qguide" option requires an optimization level of "/O2" or higher. Otherwise, the compiler will ignore the option. Each piece of advice may be considered as a way to provide more information to the compiler. More information usually enables the compiler to perform more optimizations leading to better performance. If you use the "/Qguide" and the "/Qparallel" options together, the compiler may suggest advice on further parallelizing opportunities in your application.

The advice may include suggestions for source code medications, applying specific pragmas, or add compiler options. In all cases, applying a particular advice requires the user to verify that it is safe to apply that particular suggestion. For example, if the advice is to apply a particular pragma, the user has to understand the semantics of the pragma and carefully consider if it can be safely applied to the loop in question. If the user applies the pragma without verifying its validity based on the application data-access patterns, the compiler may generate incorrect code causing the application to execute incorrectly.

The compiler does not produce any objects or executables when the "/Qguide" option is specified. By default the compiler does not produce guidance on how to improve optimizations for parallelization, vectorization, and data transformation. If a loop does not vectorize, compile your code at the "/Qguide" option and follow the GAP recommendations when applicable.

Example:

Scaler_dep.cpp:

```
49 for (i=0; i<n; i++) {
50     if (A[i] > 0) {b=A[i]; A[i] = 1 / A[i]; }
51     if (A[i] > 1) {A[i] += b;}
52   }
```

7.1.1 Default - no vectorization on gap.cpp

```
icl /c /Qvec-report2 gap.cpp

         procedure: test_scalar_dep

      gap.cpp(49): (col.  1) remark: loop was not vectorized:
      existence of vector dependence.
```

7.1.2 Get GAP advice for vectorization

```
icl /c /Qguide gap.cpp

GAP REPORT LOG OPENED ON Tue Jun 08 17:10:55 2010

remark #30761: Add -parallel option if you want the compiler to
generate recommendations for improving auto-parallelization.

gap.cpp(49): remark #30515: (VECT) Loop at line 49 cannot be
vectorized due to conditional assignment(s) into the following
variable(s): b.  This loop will be vectorized if the variable(s)
become unconditionally initialized at the top of every iteration.
[VERIFY] Make sure that the value(s) of the variable(s) read in any
iteration of the loop must have been written earlier in the same
iteration.
Number of advice-messages emitted for this compilation session: 1.

END OF GAP REPORT LOG
```

7.1.3 Making changes suggested by GAP and rebuilding to vectorize gap.cpp

Scaler_dep.cpp:

```
  for (i=0; i<n; i++) {
        b = A[i];
    if (A[i] > 0) {A[i] = 1 / A[i];}
    if (A[i] > 1) {A[i] += b;}
  }
```

icl /c /Qvec-report1 /DTEST_GAP gap.cpp

```
gap.cpp(41) (col.  3): remark: LOOP WAS VECTORIZED.
```

7.2 User-mandated Vectorization (pragma simd)

User-mandated vectorization or "vectorization using #pragma simd", is a feature that allows the user to instruct the compiler to enforce vectorization of loops. Pragma simd is designed to minimize the amount of source code changes needed in order to obtain vectorized code. Pragma simd supplements automatic vectorization just like OpenMP parallelization supplements automatic parallelization. Similar to OpenMP that can be used to parallelize loops that the compiler does not normally auto-parallelize, pragma simd can be used to vectorize loops that the compiler does not normally auto-vectorize even with the use of vectorization hints such as "#pragma vector always" or "#pragma ivdep". You must add the pragma to a loop, recompile, and the loop is vectorized.

Pragma simd can assert or not assert an error if a #pragma simd annotated loop fails to vectorize. By default "#pragma simd" is set to "noassert", and the compiler will issue a warning if the loop fails to vectorize. To direct the compiler to assert an error when the #pragma simd annotated loop fails to vectorize, add the "assert" clause to the #pragma simd. If a #pragma simd annotated loop is not vectorized by the compiler, the loop holds its serial semantics

The countable loop for the pragma simd must be an innermost loop and must conform to the (C/C++) for-loop style usable for OpenMP* worksharing loop construct. See http://www.openmp.org/mp-documents/spec30.pdf (Section 2.5.1). While the example below will auto-vectorize without "#pragma simd", it does not conform to the above requirement for User-mandated vectorization:

```
void vec_copy(float *dest, float *src, int len)
{
  float ii;

#pragma simd
    for (int i = 0, ii = 0.0f; i < len; i++)
     dest[i] = src[i] * ii++;
}
```

icl /c simd2.cpp /Qvec-report2

```
simd2.cpp

simd2.cpp(6): error: the “for” statement following  pragma simd
must have an initializer of the form <index> = <expr>

      for (i = 0, ii = 0.0f; i < len; i++) {

      ^

simd2.cpp(6): error: invalid simd pragma

  #pragma simd

  ^
```

Note that using pragma simd to force vectorization of loops that are not vectorizable due to data dependencies or other reasons could cause the compiler to generate incorrect code. In such cases, using the simd clauses such as “reduction”, “private”, etc, might help the compiler to generate correct vector code. Please refer to the Intel® C++ Compiler 12.0 User’s Guide for details on the syntax and usage of the simd clauses.

In the following example, the compiler reports that the loop cannot be vectorized due to existence of vector dependence. The compiler will generate incorrect code if it is instructed to force vectorization of the loop in this example:

```
char foo(char *A, int n){

  int i;
  char x = 0;

#ifdef SIMD
#pragma simd
#endif
#ifdef REDUCTION
#pragma simd reduction(+:x)
#endif
#ifdef IVDEP
#pragma ivdep
#endif
  for (i=0; i<n; i++){
    x = x + A[i];
  }
  return x;
}
```

icl /c /DIVDEP -Qvec-report2 simd3.cpp

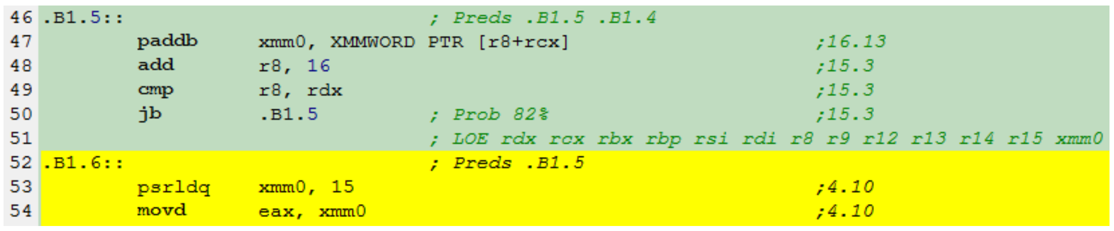
```
simd3.cpp
simd3.cpp(15) (col.  3): remark: loop was not vectorized: existence
of vector dependence.
```

```
Forcing the loop to vectorize will result in incorrect code being
generated:
```

`>icl /c /DSIMD -Qvec-report2 simd3.cpp`

```
simd3.cpp
simd3.cpp(15) (col.  3): remark: SIMD LOOP WAS VECTORIZED.
```

7.2.1 Incorrect Code



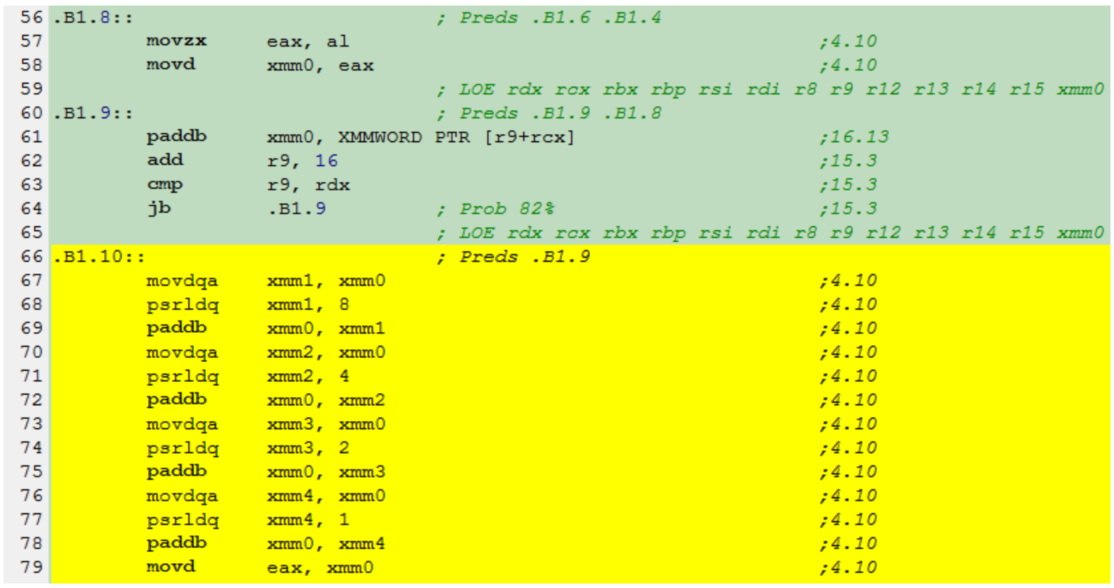
```
.B1.5::                                ; Preds .B1.5 .B1.4
        paddb     xmm0, XMMWORD PTR [r8+rcx]                        ;16.13
        add       r8, 16                                            ;15.3
        cmp       r8, rdx                                           ;15.3
        jb        .B1.5         ; Prob 82%                          ;15.3
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r9 r12 r13 r14 r15 xmm0
.B1.6::                         ; Preds .B1.5
        psrldq    xmm0, 15                                          ;4.10
        movd      eax, xmm0                                         ;4.10
```

Adding the reduction clause hints the compiler to generate correct code as shown below:

`>icl /c /DREDUCTION -Qvec-report2 simd3.cpp`

```
simd3.cpp
simd3.cpp(15) (col.  3): remark: SIMD LOOP WAS VECTORIZED.
```

7.2.2 Correct Code



```
.B1.8::                         ; Preds .B1.6 .B1.4
        movzx     eax, al                                           ;4.10
        movd      xmm0, eax                                         ;4.10
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r9 r12 r13 r14 r15 xmm0
.B1.9::                         ; Preds .B1.9 .B1.8
        paddb     xmm0, XMMWORD PTR [r9+rcx]                        ;16.13
        add       r9, 16                                            ;15.3
        cmp       r9, rdx                                           ;15.3
        jb        .B1.9         ; Prob 82%                          ;15.3
                                ; LOE rdx rcx rbx rbp rsi rdi r8 r9 r12 r13 r14 r15 xmm0
.B1.10::                        ; Preds .B1.9
        movdqa    xmm1, xmm0                                        ;4.10
        psrldq    xmm1, 8                                           ;4.10
        paddb     xmm0, xmm1                                        ;4.10
        movdqa    xmm2, xmm0                                        ;4.10
        psrldq    xmm2, 4                                           ;4.10
        paddb     xmm0, xmm2                                        ;4.10
        movdqa    xmm3, xmm0                                        ;4.10
        psrldq    xmm3, 2                                           ;4.10
        paddb     xmm0, xmm3                                        ;4.10
        movdqa    xmm4, xmm0                                        ;4.10
        psrldq    xmm4, 1                                           ;4.10
        paddb     xmm0, xmm4                                        ;4.10
        movd      eax, xmm0                                         ;4.10
```

Here an example where the loop does not auto-vectorize due to loop-carried dependencies even with the use of “pragma ivdep” but can be vectorized with “pragma simd" . There are no loop-carried dependencies within the vector lengths 4, 8, and 16 so the "vectorlength(16)" clause is used to tell the compiler that vectorizing the loop with vectorlength 4, 8, and 16 is legal but beyond that is not legal due to loop-carried dependencies.

```
13   short x = 0;
14 #ifdef SIMD
15 #pragma simd reduction(+:x)
16 #endif
17   for (i=0; i<n; i++) {
18     x = SAT2SI16(x + p[i]*q[i]);
19   }
20   return x;
21 }
22
23 int main(int argc, char **argv) {
24   short x = 0;
25   const __int64 startTime = __rdtsc();
26
27   for (int i=0; i<32767; i++) {
28     a[i] = 1;
29     b[i] = 1;
30   }
31
32 #ifdef SIMD
33 #pragma simd vectorlength(16)
34 #endif
35   for (int i=0; i<32767; i++) {
36     if (i >= 16 && i < 32767) {
37       b[i] = b[i-16] - 1;
38     }
39   printf("b[%d] = \n", b[i]);
40   }
41
42   x = sat2short(&a[0], &b[0], 32767);
```

>icl /c /Qvec-report2 simd4.cpp

```
Intel(R) C++ Compiler XE for applications running on Intel(R) 64,
Version 12.0.0.063 Build 20100721

simd4.cpp(27) (col. 3): remark: LOOP WAS VECTORIZED.
simd4.cpp(35) (col. 3): remark: loop was not vectorized: existence
of vector dependence.
simd4.cpp(35) (col. 3): remark: loop was not vectorized: existence
of vector dependence.
simd4.cpp(42) (col. 7): remark: loop was not vectorized: existence
of vector dependence.
simd4.cpp(17) (col. 3): remark: loop was not vectorized: existence
of vector dependence.
```

```
>icl /c /Qvec-report2 simd4.cpp /DSIMD
```

```
Intel(R) C++ Compiler XE for applications running on Intel(R) 64,
Version 12.0.0.063 Build 20100721

simd4.cpp
simd4.cpp(27) (col.  3): remark: LOOP WAS VECTORIZED.
simd4.cpp(35) (col.  3): remark: SIMD LOOP WAS VECTORIZED.
simd4.cpp(35) (col.  3): remark: SIMD LOOP WAS VECTORIZED.
simd4.cpp(42) (col.  7): remark: SIMD LOOP WAS VECTORIZED.
simd4.cpp(17) (col.  3): remark: SIMD LOOP WAS VECTORIZED.
```

Here is another example where the compiler does not vectorize the loop because it does not know whether the variable “t” is assigned in every iteration. With the programmer’s use of the “private clause” to assert that every iteration of the loop has its own private copy of “t”, the compiler can vectorize this loop correctly. Please note that the “private” clause may introduce sequential inconsistency which is well known for vectorization and parallelization. It is the user’s responsibility to maintain sequential consistency if needed. For example, in the example below, the serial execution result of “t” may not be the same as the SIMD execution. In the SIMD execution, the last value of “t” is from the last iteration of the loop, while for the serial execution, the value of “t” may not come from the last iteration depending on A[i] and B[i].

```
 1 void foo(int *A, int *B, int *restrict C, int n){
 2   int i;
 3   int t = 0;
 4
 5 #ifdef PRIVATE
 6 #pragma simd private(t)
 7 #endif
 8   for (i=0; i<n; i++){
 9     if (A[i] > 0) {
10       t = A[i];
11     }
12     if (B[i] < 0) {
13       t = B[i];
14     }
15     C[i] = t;
16   }
17 }
```

```
icl /c /Qvec-report2 /Qrestrict simd5.cpp
```

```
Intel(R) C++ Compiler XE for applications running on Intel(R) 64,
Version 12.0.0.063 Build 20100721

simd5.cpp
simd5.cpp(8) (col.  3): remark: loop was not vectorized: existence of
vector dependence.
```

```
>icl /c /Qvec-report2 /Qrestrict simd5.cpp /DPRIVATE

Intel(R) C++ Compiler XE for applications running on Intel(R) 64,
Version 12.0.0.063 Build 20100721

simd5.cpp
simd5.cpp(8) (col.  3): remark: SIMD LOOP WAS VECTORIZED.
```

Elemental Functions

An elemental function is a regular function, which can be invoked either on scalar arguments or on array elements in parallel. You define an elemental function by adding **"_declspec(vector)"** (on Windows*) and "**__attribute__((vector))"** (on Linux*) before the function signature:

```
 __declspec (vector)
  double ef_add (double x, double y){  return x + y;}
```

When you declare a function as elemental the compiler generates a short vector form of the function, which can perform your function's operation on multiple arguments in a single invocation. The compiler also generates a scalar implementation of the function, and the program can invoke the function either on single arguments or array arguments, at different places in the program. The vector form of the function can be invoked in parallel contexts in the following ways:

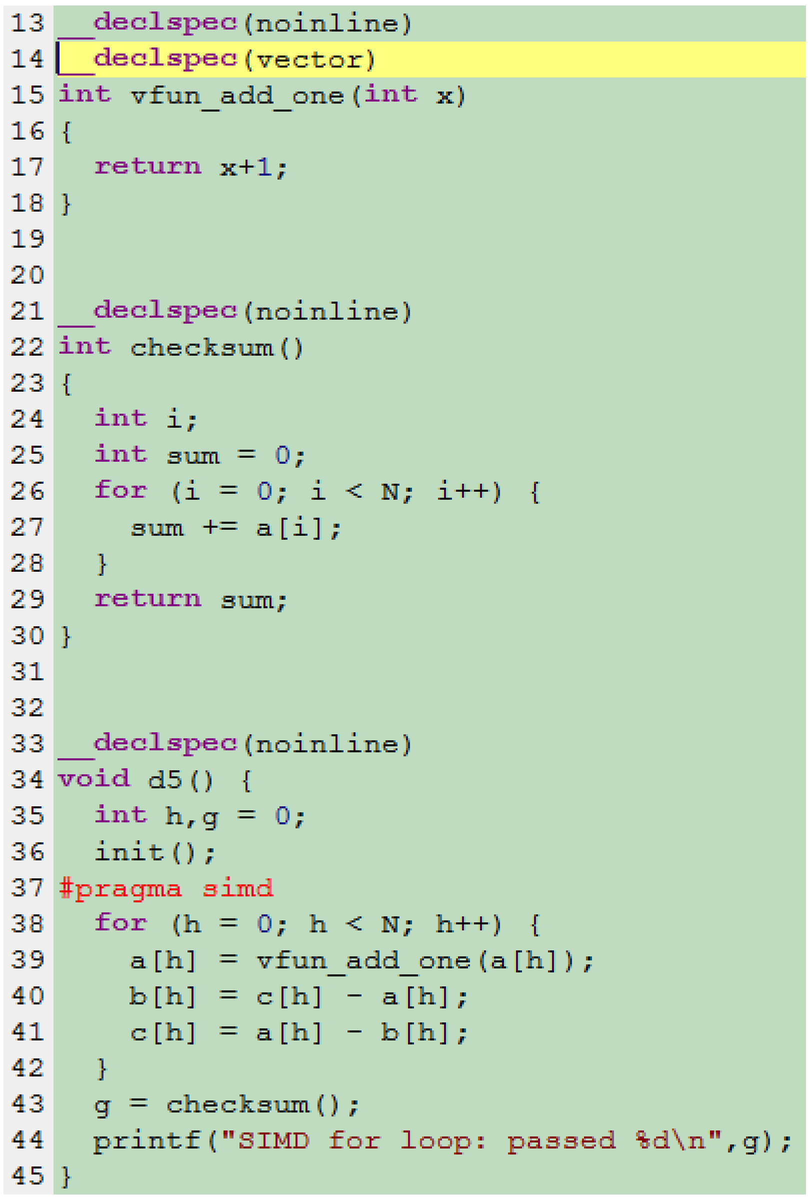
1. From a for-loop. It gets auto-vectorized; a loop that only has a call to an elemental function is always vectorizable, but the auto-vectorizer is allowed to apply performance heuristics and decide not to vectorize the function.

2. From a for-loop with pragma simd. If the elemental function is called from a loop with "pragma simd", the compiler no longer does any performance heuristics, and is guaranteed to call the vector version of the function. Note that if the loop contains a call to the "printf" C library function, the compiler will issue the following remark:

   "remark: loop was not vectorized: existence of vector dependence."

3. From a cilk_for

4. From an array notation syntax.



```
13 __declspec(noinline)
14 __declspec(vector)
15 int vfun_add_one(int x)
16 {
17   return x+1;
18 }
19
20
21 __declspec(noinline)
22 int checksum()
23 {
24   int i;
25   int sum = 0;
26   for (i = 0; i < N; i++) {
27     sum += a[i];
28   }
29   return sum;
30 }
31
32
33 __declspec(noinline)
34 void d5() {
35   int h,g = 0;
36   init();
37 #pragma simd
38   for (h = 0; h < N; h++) {
39     a[h] = vfun_add_one(a[h]);
40     b[h] = c[h] - a[h];
41     c[h] = a[h] - b[h];
42   }
43   g = checksum();
44   printf("SIMD for loop: passed %d\n",g);
45 }
```

```
>icl /c /Qvec-report2 vectorFunc.cpp
```

```
Intel(R) C++ Compiler XE for applications running on Intel(R) 64,
Version 12.0.0.063 Build 20100721

vectorFunc.cpp

vectorFunc.cpp(36) (col.  3): remark: LOOP WAS VECTORIZED.
vectorFunc.cpp(38) (col.  3): remark: SIMD LOOP WAS VECTORIZED.
vectorFunc.cpp(16) (col.  1): remark: FUNCTION WAS VECTORIZED.
vectorFunc.cpp(16) (col.  1): remark: FUNCTION WAS VECTORIZED.
vectorFunc.cpp(26) (col.  3): remark: LOOP WAS VECTORIZED.
vectorFunc.cpp(10) (col.  3): remark: LOOP WAS VECTORIZED.
```

For more details on elemental functions and their limitations please see the Intel® C++ Compiler XE 12.0 User and Reference Guide, and the article titled "Elemental functions: Writing data parallel code in C/C++ using Intel® Cilk Plus "

8. Conclusion

There are no hard and fast rules that guarantee auto-vectorization. However, applying the recommendations described above increases the success rate. Our experience shows that in cases where auto-vectorization is successful, the speedups are significant. It is worthwhile to try the auto-vectorization feature in the Intel Compiler before writing your own SIMD code because using the compiler is less time consuming than writing your own SIMD code. Furthermore, using the Intel Compiler to auto-vectorize is more future-proof. To update your optimization for new processors, you can recompile the code at that time to target the new processors rather than having to re-write the SIMD code. There will be cases when writing your own SIMD code is unavoidable, e.g. when the compiler cannot auto-vectorize or when you can vectorize the code much better than the compiler.

9. Appendix

Below are some performance and diagnostic switches available for use. Some of these options like the "-no-vec", are to be used for diagnostic purposes only.

These options are available for both Intel® and non-Intel microprocessors but they may result in more optimizations for Intel microprocessors than for non-Intel microprocessors.

Linux Flag	**Windows Flag**	**Description**
-vec-report0	/Qvec-report0	disable vectorization diagnostics (default)
-vec-report1	/Qvec-report1	report successfully vectorized code
-vec-report2	/Qvec-report2	also report which loops were not vectorized
-vec-report3	/Qvec-report3	also report all prohibiting data dependencies
-no-vec	/Qvec-	disable auto-vectorization regardless of any flags or pragmas in the code
-vec-threshold0	/Qvec-threshold0	will try to auto-vectorize loops even if the compiler doesn't think it improve performance; use with caution because it will vectorize a lot of loops that are unsuitable, and slow them down.

Linux Flag	Windows Flag	Description
-O0	/Od	disable optimizations
-O1	/O1	enable optimizations for code size
-O2	/O2	enable default optimizations for speed
-O3	/O3	enable aggressive loop optimizations for speed
-guide	/Qguide	enable guided automatic parallelization and vectorization; compiler gives advices on how to get loops to vectorize or parallelize; only available in 12.0 or later compiler
-S	/S	generate assembly file
-unroll=0	/Qunroll:0	disable loop unrolling
-opt-prefetch-	/Qopt-prefetch-	disable software prefetching
-ip	/Qip	enable interprocedural optimizations in a single source file
-ipo	/Qipo	enable interprocedural optimizations across source files, including inlining
-fargument-noalias	/Qalias-args-	Tells the compiler that function arguments cannot alias each other.
-fp-model precise	/fp:precise	improve floating-point consistency; may slow performance
-restrict	/Qrestrict	enable use of restrict keyword that conveys non-aliasing properties to the compiler
-parallel	/Qparallel	enable automatic parallelization
-openmp	/Qopenmp	enable parallelization with OpenMP extensions
-std=c99	/Qstd=c99	enable C99 extensions (ISO 1999)

Table 2. Pragmas that can help the compiler to auto-vectorize the following loop.

Compiler Hint	Description
#pragma ivdep	ignore potential (unproven) data dependences

Compiler Hint	Description
#pragma vector always	override efficiency heuristics
#pragma vector nontemporal	hint to use streaming stores
#pragma vector [un]aligned	assert [un]aligned property
#pragma novector	disable vectorization for following loop
#pragma distribute point	hint to split loop at this point
#pragma loop count (<int>)	hint for likely trip count
#pragma simd	enforces vectorization; Only available in 12.0 or later compiler. See Compiler User's Guide
restrict	keyword to assert exclusive access through pointer; requires command line option "-restrict". See examples above.

__declspec(align(<int>,<int>)) (Windows*)
 	
__attribute__(align(<int>,<int>)) (Linux*)
 	
_declspec(align(<int>,<int>)) (_icc recognizes _declspec but _declspec is more a Windows style syntax, on Linux the syntax used is use __attribute__.
	requires memory alignment

__assume_aligned(<var>,<int>)	assert alignment property

10. References

Intel® Software Documentation Library http://software.intel.com/en-us/articles/intel-software-technical-documentation/

The Software Vectorization Handbook, Aart Bik, Intel Press, 2004. ISBN 0-9743649-2-4

Vectorization with the Intel® Compilers (Part I), Aart Bik, http://software.intel.com/en-us/articles/vectorization-with-the-intel-compilers-part-i/

http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/

http://software.intel.com/en-us/articles/how-to-check-auto-vectorization/

http://software.intel.com/en-us/articles/vectorization-writing-cc-code-in-vector-format/

Image Processing Acceleration Techniques using Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions, Petter Larsson, Eric Palmer

http://software.intel.com/en-us/articles/image-processing-acceleration-techniques-using-intel-streaming-simd-extensions-and-intel-advanced-vector-extensions/

http://software.intel.com/en-us/articles/image-processing-acceleration-techniques-using-intel-streaming-simd-extensions-and-intel-advanced-vector-extensions/

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

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

Intel, the Intel logo and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, visit Intel http://www.intel.com/performance/resources/limits.htm

*Other names and brands may be claimed as the property of others.
The linked sites are not under the control of Intel and Intel is not responsible for the content of any linked site or any link contained in a linked site. Intel reserves the right to terminate any link or linking program at any time. Intel does not endorse companies or products to which it links and reserves the right to note as such on its web pages. If you decide to access any of the third party sites linked to this Site, you do this entirely at your own risk.
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, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © 2010, Intel Corporation. All Rights Reserved.