Getting Started with Intel® Cilk™ Plus SIMD Vectorization and SIMD-enabled functions

Introduction

SIMD Vectorization and SIMD-enabled Functions are a part of Intel® Cilk™ Plus feature supported by the Intel® C++ Compiler that provide ways to vectorize loops and user defined functions. Vectorization is the key to improving your applications' performance through taking advantage of the processor's Single Instruction Multiple Data (SIMD) capability to operate on multiple array (or vector) elements at a time. The Intel® Compilers provide unique capabilities to enable vectorization. The programmer may be able to help the compiler to vectorize more loops through a simple programming style and by the use of compiler features designed to assist vectorization. This article discusses how to use the vector SIMD-enabled functions, and the SIMD directive (#pragma simd) from the Intel® Cilk™ Plus, to help the compiler to vectorize C/C++ code and improve performance.

Document Organization

To demonstrate the usage and performance characteristics of SIMD vectorization and SIMD-enabled functions, you will:

- Establish a performance baseline by building and running the scalar (non-vectorized) version of a loop.
- Improve application performance by using SIMD-enabled Functions.
- Improve application performance further by using SIMD vectorization clauses.
- Evaluate the effect of different SIMD vector lengths on performance.
- Evaluate the effect of using the SIMD pragma without using SIMD-enabled functions.

System Requirements

To compile and run the example and exercises in this document you will need Intel® Parallel Composer XE 2011 Update 9 or higher, and an Intel® Pentium 4 Processor or higher with support for Intel® SSE2 or higher instruction extensions. **The exercises in this document were tested on a first generation Intel® Core™ i5 system supporting 128-bit vector registers.** The instructions in this tutorial show you how to build and run the example with the Microsoft Visual Studio* 2008. The example provided can also be built from the command line on Windows*, Linux*, and Mac OS* X using the following command line options:

```
Windows*: icl /Qvec-report2 /Qstd=c99 simdelemental.c fsqrt.c
/Fesimdelemental.exe

Linux and Mac OS* X: icc -vec-report2 -std=c99 simdelemental.c
fsqrt.c -o simdelemental
```

The declaration of the induction variable in the for-loop, used in this example, is a C99 feature that requires compiling with the "/Qstd=c99" option (Windows*) or "-std=c99" (Linux*). If you declare the induction variable outside the for-loop, you do not need to use the /Qstd=c99 compiler option.

SIMD Vectorization

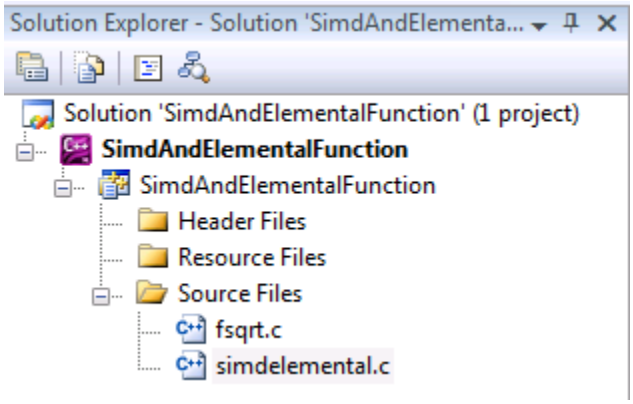
SIMD vectorization also known as "User-mandated vectorization" or "vectorization using #pragma simd" enables the user to instruct the compiler to enforce vectorization of loops. Pragma simd is designed to minimize the amount of source code changes needed in order to obtain vectorized code. Pragma simd supplements automatic vectorization just like OpenMP parallelization supplements automatic parallelization. Similar to the way that OpenMP 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.

Locating the Samples

To begin this tutorial, open the SimdAndElementalFunction.zip archive attached. Use these files for this tutorial:

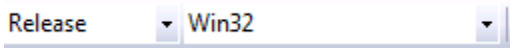
- `SimdAandElementalFunction.sln`
- `SimdAndElementalFunction.c`
- `Simdelemental.h`
- `Fsqrt.c`

1. Open the Microsoft Visual Studio* solution file, SimdAndElementalFunction.sln,



and follow the steps below to prepare the project for the SIMD vectorization exercises in this tutorial:

2. Select "Release" "Win32" configuration



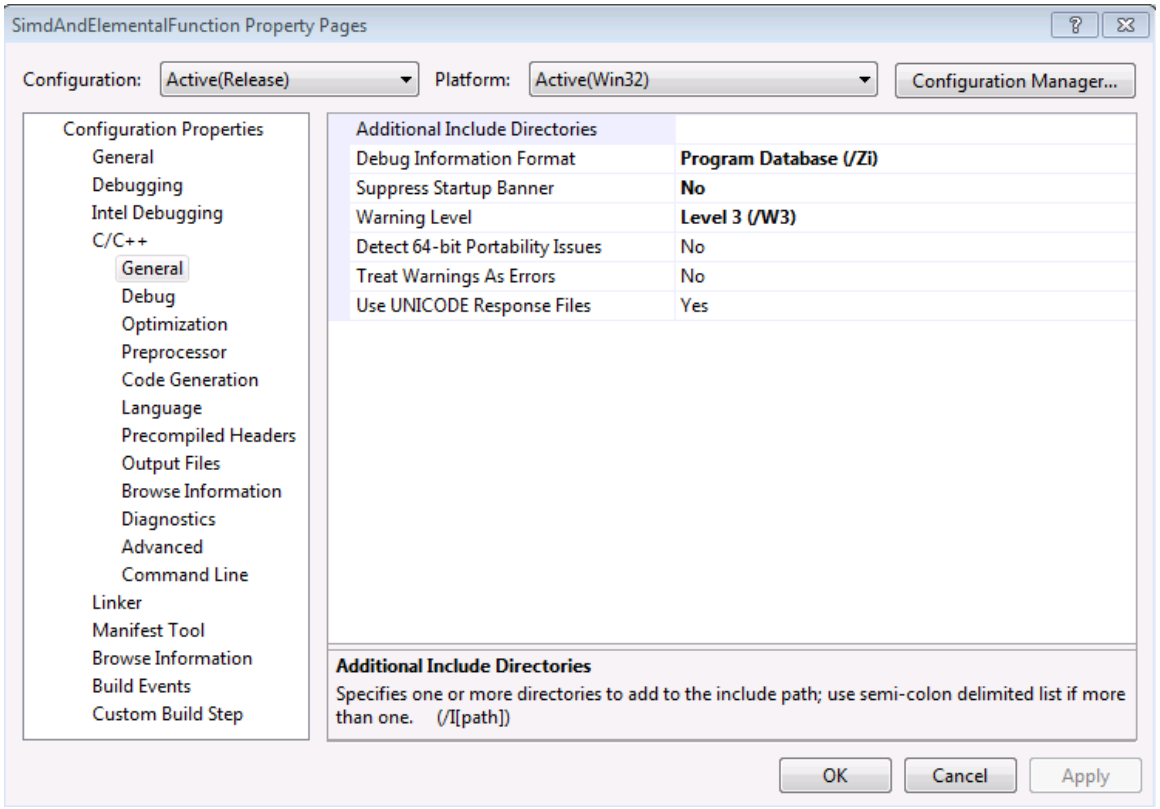
3. Clean the solution by selecting **Build > Clean Solution**.

You just deleted all of the compiled and temporary files association with this solution. Cleaning a solution ensures that the next build is a full build rather than changing existing files.

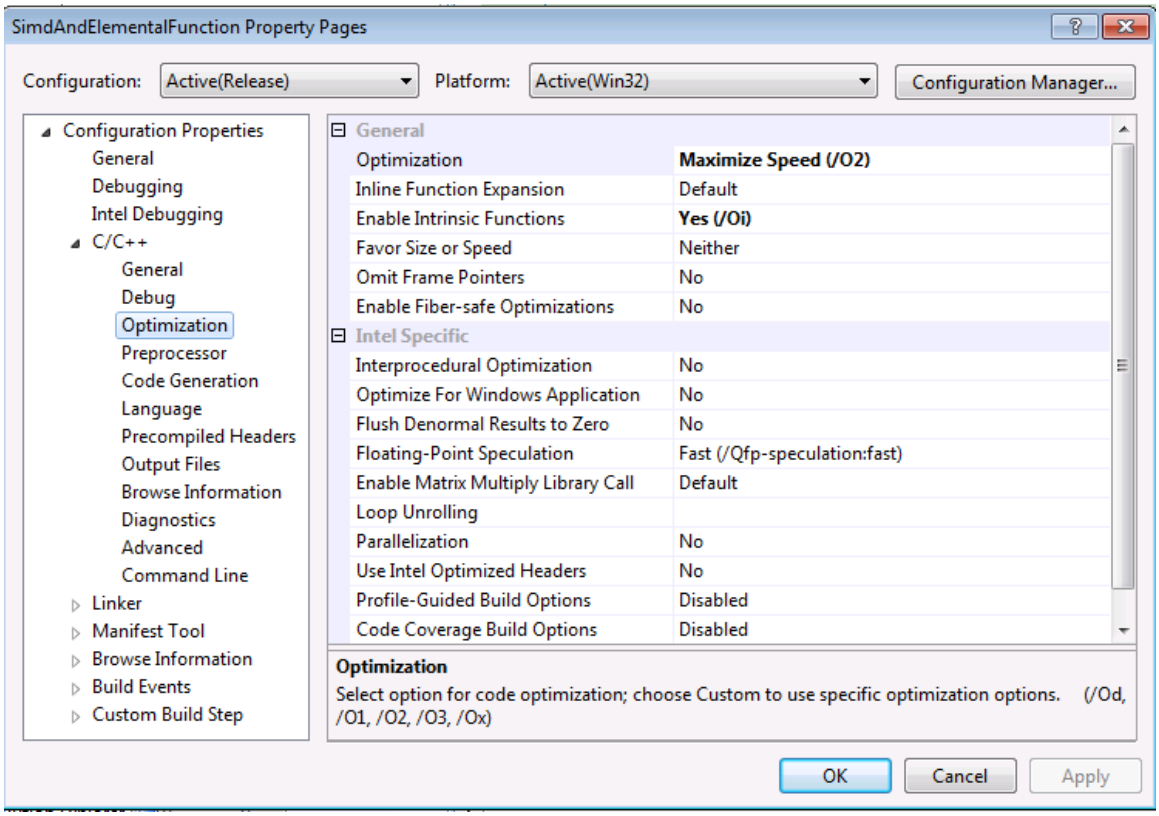
Establishing a Performance Baseline

To set a performance baseline for the improvements that follow in this tutorial, build your project with these settings:

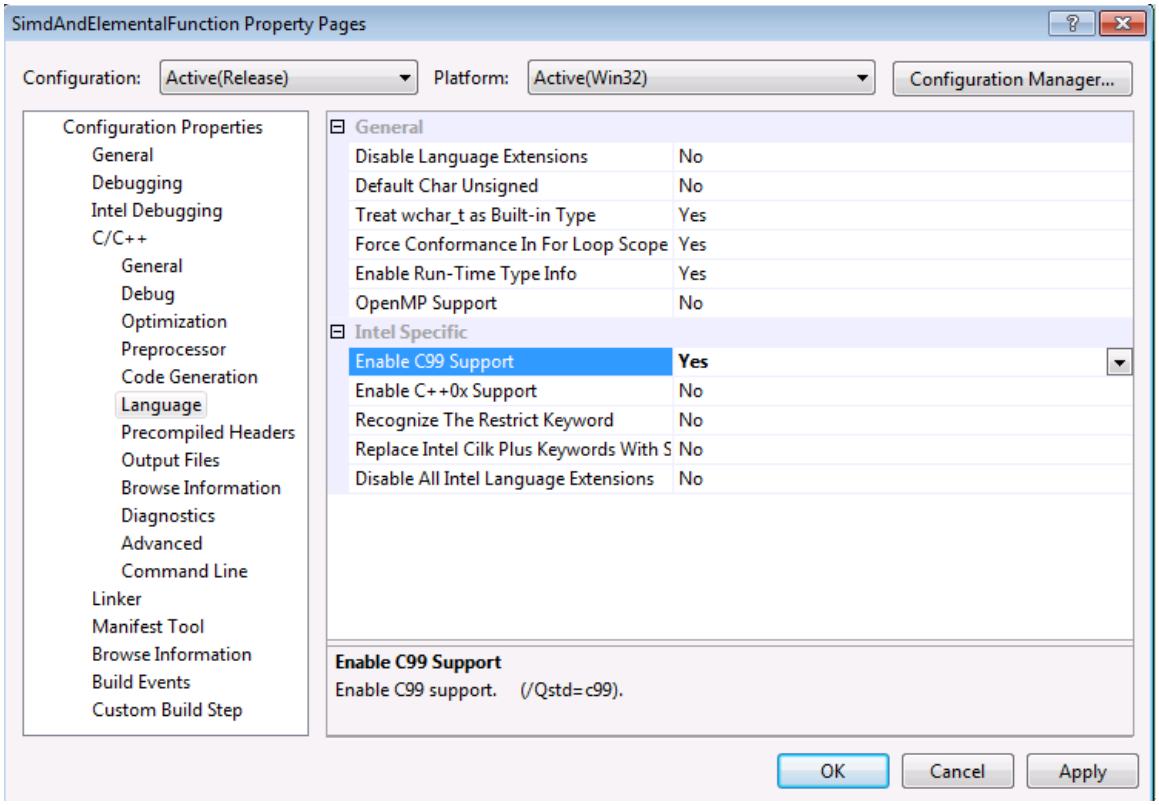
1. Select **Project > Properties > C/C++ > General > Suppress Startup Banner > No**.



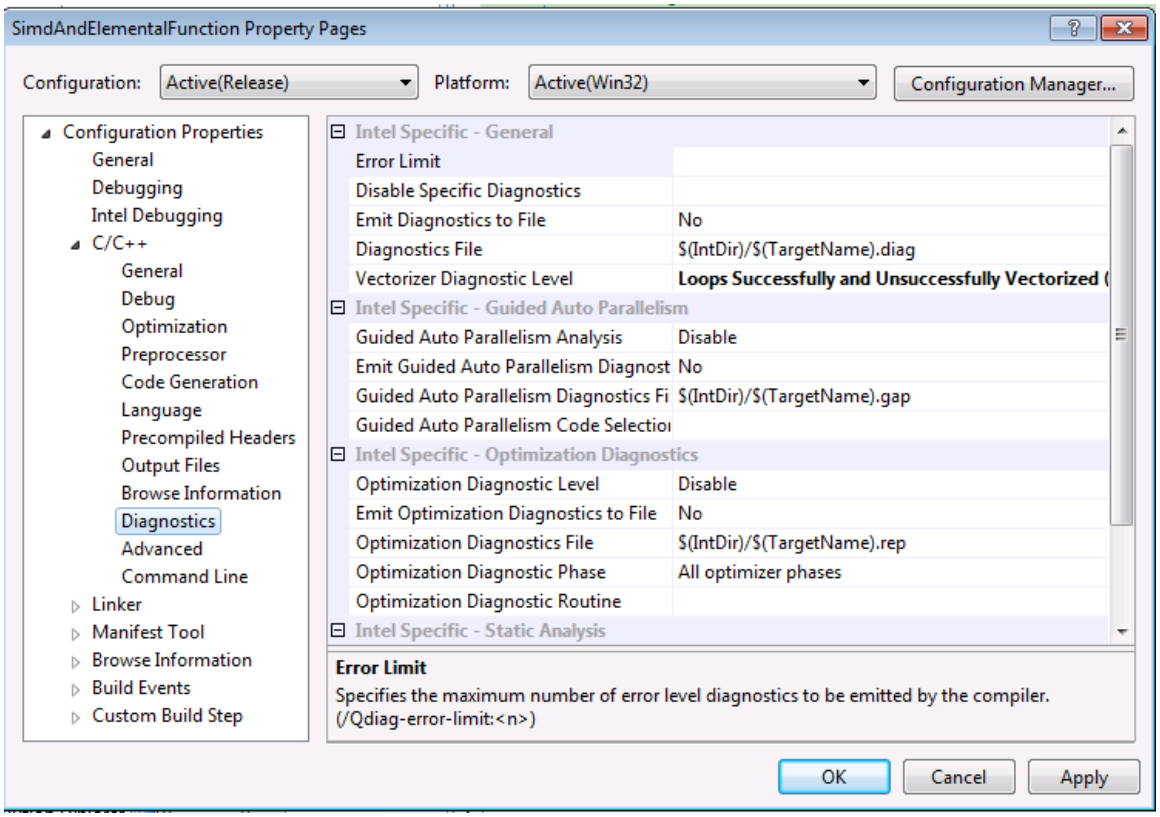
2. Select **Project > Properties > C/C++ > Optimization > Optimization > Maximize Speed (/O2)**



3. Select **Project > Properties > C/C++ > Language > Enable C99 Support > Yes**



4. Select **Project > Properties > Diagnostics > Vectorizer Diagnostic Level > Loops Successfully Vectorized (1) (/Qvec-report2)**



5. Rebuild the project, then run the executable (**Debug > Start Without Debugging**). Running the program results in starting a window that displays the program's execution time in seconds. Record the execution time reported in the output. This is the baseline against which subsequent improvements will be measured. In establishing the baseline performance it is a good practice to compare the vec-report2 results between -O2 and -O3 optimization levels because more vectorization candidates tend to appear at –O3. For this example, however, the –O2 and –O3 results are the same.

```
simdelemental.c(72): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(98): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(76): (col. 3) remark: nonstandard loop is
not a vectorization candidate.
simdelemental.c(92): (col. 5) remark: nonstandard loop is
not a vectorization candidate.
simdelemental.c(57): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(50): (col. 3) remark: LOOP WAS VECTORIZED.
```

The vectorization report indicates that the loops in the simdelemental.c at lines 76 and 92 are not vectorization candidates. The loop at line 76 does not vectorize because it is not an inner-most loop. The Intel® C++ Compiler is capable of vectorizing outer loops if they exhibit greater data-level parallelism and locality than the innermost loop. In this example the compiler does not see any benefits in

vectorizing the outer loop at line 76. Therefore, it does not vectorize it. You can force the outer loop to vectorize by removing the SIMD pragma from the inner loop and adding it above line 76, instead. A guideline for using the SIMD pragma is to apply it to loops with linear memory access on the hotspot.

The loop at line 92 does not vectorize because it has a function call in it. Loops containing user defined function calls do not auto vectorize, unless they are inlined at the call site, because the autovectorizer may not know about the function dependencies and other side effects. The hotspot in this example is the innermost loop at line 92 that we will vectorize in this guide.

Improving Performance by Using SIMD-enabled Functions

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

```
__declspec (vector)
float v_add(float x, float y) { return x+y;}
// caller:
for (int j = 0; j < N; ++j) { r[j] = v_add(a[j],b[j]); }
```

The use of __declspec(vector) indicates to the compiler that the function "v_add" is intended to be used as an SIMD-enabled function. When you declare a function as SIMD-enabled the Intel® C++ Compiler generates a vector form of the function internally, which receives a vector of arguments for x and a vector of arguments for y, and can perform the function's operation on multiple arguments in a single invocation, and returns a vector of results instead of a single result. The compiler internally generates the vector form of the function and implicitly calls it from a vector level parallel context such as "#pragma simd" or within the context of a "cilk_for" loop if the "cilk_for" loop is vectorizable.

```
// Vector version generated by the Intel® C++ Compiler
v_add(x0…x3, y0…y3) → r0…r3
v_add(x4…x7, y4…y7) → r4…r7
… …
```

The benefit may be enhanced performance without requiring the programmer to write a vector version of the function explicitly in low level intrinsic or assembly instructions. The compiler also generates a scalar implementation of the function, and invokes the function either on single arguments or array arguments, as appropriate. If the SIMD-enabled function is called from a loop with "pragma simd", the compiler will attempt to call the vector version of the function.

In this exercise we will declare the fSqrtMul() function in our example as an SIMD-enabled function as follows:

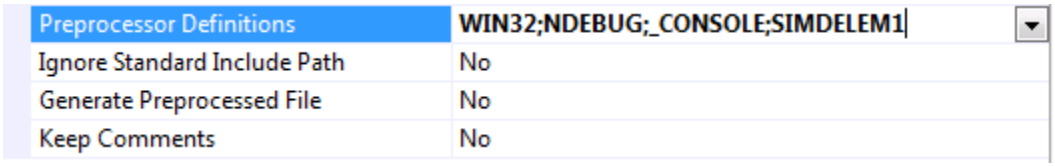
```
// Declaring fSqrtMul() function as an SIMD-enabled function
__declspec(vector)
```

```
extern float fSqrtMul(float *op1, float op2);
```

The invocation of an SIMD-enabled function and its declaration/definition need not be in the same file as long as all sites see the same set of SIMD vector annotations in the prototype. To illustrate this point the invocation and definition of the fSqrtMul() SIMD-enabled function in this example are in separate files.

To enable function form of the "fSqrtMul" function in this example:

1. Add the preprocessor definition, SIMDELEM1, by selecting **Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions**, then adding SIMELEM1 to the existing list of preprocessor definitions. This will enable the vector form of the "fSqrtMul" routine.



2. Rebuild the project. Notice that both the loop at line 92 and the fSqrtMul() function are now vectorized.

```
simdelemental.c(72): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(92): (col. 5) remark: SIMD LOOP WAS VECTORIZED.
simdelemental.c(76): (col. 3) remark: loop was not vectorized:
not inner loop.
simdelemental.c(98): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(57): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(50): (col. 3) remark: LOOP WAS VECTORIZED.
fsqrt.c(18): (col. 39) remark: FUNCTION WAS VECTORIZED
fsqrt.c(18): (col. 39) remark: FUNCTION WAS VECTORIZED.
```

3. Run the executable and record the new execution time. You should see a performance improvement due to SIMD loop and SIMD-enabled function vectorization. The SIMD-enabled function also uses the vector version of the "sqrt" function provided by the Intel® C++ Compiler library that helps improve the performance.

Improving Performance Further by Using SIMD Clauses

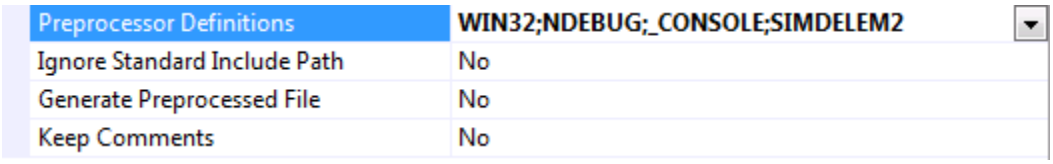
The compiler could generate more optimal code if it had more information about loop and vector functions characteristics such as memory access pattern, private vs shared variables, etc. The programmer can specify such information by using various vector clauses. In this exercise we will declare the SIMD-enabled function as follows:

```
// Declaring fSqrtMul() function as an SIMD-enabled function
//with simd clauses for the function parameters
__declspec(vector(linear(op1),
            uniform(op2)))
extern float fSqrtMul(float *op1, float op2);
```

The linear clause specifies that the "op1" parameter is a variable of scalar type with unit-stride memory access (accessing memory in adjacent locations). This allows the compiler to generate faster unit-stride vector memory load/store instructions (e.g. movaps or movups supported on Intel® SIMD hardware) rather than generating longer latency gather/scatter instructions that it would have to do otherwise. The uniform clause specifies that the "op2" parameter is shared across SIMD lanes of vector execution, because its value does not change within the iterations of the inner loop. This allows the compiler to load the value of the "op2" variable only once rather than loading it for every iteration of the inner loop.

To enable SIMD-enabled function with SIMD vectorization clauses:

1. Add the preprocessor definition, SIMDELEM2, by selecting **Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions**, then replacing SIMDELEM1 with SIMDELEM2 in the list of preprocessor definitions.



2. Rebuild the project.

```
simdelemental.c(72): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(92): (col. 5) remark: SIMD LOOP WAS VECTORIZED.
simdelemental.c(76): (col. 3) remark: loop was not vectorized:
not inner loop.
simdelemental.c(98): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(57): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(50): (col. 3) remark: LOOP WAS VECTORIZED.
fsqrt.c(18): (col. 39) remark: FUNCTION WAS VECTORIZED
fsqrt.c(18): (col. 39) remark: FUNCTION WAS VECTORIZED
```

3. Run the executable and record the new execution time. You should see a performance improvement due to use of the uniform and linear clauses.

Evaluating the Effects of Different SIMD Vector Lengths on Performance

Choosing the appropriate vector length for the target CPU could improve the SIMD loop performance. You may control the ideal vector length or unrolling by adding an optional parameter, vectorlength() to the pragma.

The logical vector length specified by the vectorlength clause on the SIMD pragma is recommended to be at least twice the physical vector length based on the data types, for achieving a better instruction-level parallelism on Intel® Architecture Processors.

> **NOTE:** Using a vector length of twice the physical vector length for more complex loops with many arrays, would increase the register pressure and might result in spills to memory.

For example:

If the physical vector register size is 128 bits:

Data Type	char (8-bit)	short (16-bit)	float/int (32-bit)	Long long / double (64-bit)
Physical Vector Length	16	8	4	2
Recommended Logical Vector Length to Use	32	16	8	4

If the physical vector register size is 256 bits as in Intel® AVX:

Data Type	char (8-bit)	short (16-bit)	float/int (32-bit)	Long long / double (64-bit)
Physical Vector Length	32	16	8	4
Recommended Logical Vector Length to Use	64	32	16	8

If the physical vector register size is 512 bits as in Intel® MIC Architecture:

Data Type	char (8-bit)	short (16-bit)	float/int (32-bit)	Long long / double (64-bit)
Physical Vector Length	64	32	16	8
Recommended Logical Vector Length to Use	128	64	32	16

Based on the above guideline and a target CPU with 128-bit physical vector registers , we use a vectorlength=8 for the float data types used in our example, to operate on 8 float data elements which improves performance relative to a smaller vector length such as vectorlength(4). When the vectorlength() parameter is added the line looks like this:

```
__declspec(vector(linear(op1),
             uniform(op2),vectorlength(8)))
extern float fSqrtMul(float *op1, float op2);
```

The vectorlength(8) clause states that the vector function operates on 8 elements at a time. On a CPU with 128-bit vector registers, with a vectorlength(8) clause, the Intel® C++ Compiler will vectorize the loop at line 92 in this example (e.g. computes b[0..7]) using two 128-bit SIMD registers for 8 32-bit float data elements, and each call to our vector function will use two 128-bit SIMD registers for 8 32-bit float data elements (e.g. to pass addresses of a[0..7] to the vector function). On a CPU with 256-bit vector registers such as Intel CPUs with **Intel® AVX** support, with a vectorlength(8) clause, the compiler will vectorize the loop using one 256-bit SIMD register for 8 floats, and each call to our SIMD-enabled function will use two 128-bit SIMD registers to pass 8 addresses for 8 floats. If your target CPU has 512-bit vector registers such as **Intel® MIC** architecture, you could use a vectorlength(16) upon which each call to the vector function will use one 512-bit SIMD register with 16 floats to vectorize the loop.

To see the effect of using vectorlength(8) on performance:

1. Add the preprocessor definition, VLEN=8, by selecting **Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions**, then adding VLEN=8 to the existing list of preprocessor definitions.

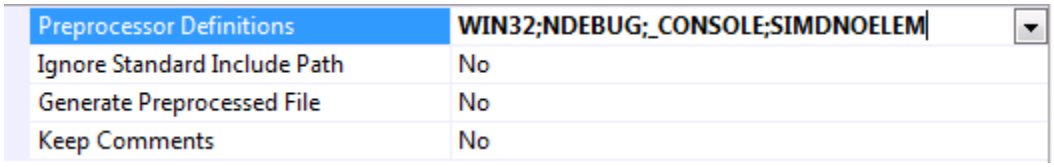
Preprocessor Definitions	**WIN32;NDEBUG;_CONSOLE;SIMDELEM2;VLEN=8**
Ignore Standard Include Path	No
Generate Preprocessed File	No
Keep Comments	No

2. Build and run the executable and record the new execution time. On a CPU with 128-bit SIMD registers you should see better performance than the previous exercise. Experiment with different vector lengths on different CPUs and observe the performance effects. For example, based on the above tables, you could experiment with vector lengths 16 and 32 for this example, on CPUs with 256-bit registers (Intel® AVX), and those with 512-bit SIMD registers (Intel® MIC).

Evaluating the Effect of pragma simd Vectorization without Using SIMD-enabled Functions

In this exercise we will vectorize the loop with "#pragma simd" without declaring the fSqrtMul() function as an SIMD-enabled function to see the performance affects.

1. Add the preprocessor definition, SIMDNOELEM, by selecting **Project > Properties > C/C++ > Preprocessor > Preprocessor Definitions**, then adding SIMDNOELEM to the list of preprocessor definitions as shown.



2. Rebuild the project. Notice that the compiler followed the #pragma simd directive and vectorized the loop at line 92.

```
simdelemental.c(72): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(92): (col. 5) remark: SIMD LOOP WAS
VECTORIZED.
simdelemental.c(98): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(76): (col. 3) remark: loop was not
vectorized: nonstandard oop is not a vectorization candidate.
simdelemental.c(57): (col. 3) remark: LOOP WAS VECTORIZED.
simdelemental.c(50): (col. 3) remark: LOOP WAS VECTORIZED.
```

3. Run the executable and record the new execution time. Although the loop is now vectorized you may not see much performance improvement over the non-vectorized baseline version of the loop. This is because adding the SIMD pragma alone, with the call to non-SIMD-enabled user function serializes the call. The compiler will break down the vector values into scalar pieces and then calls the scalar function. The pseudo code below shows how the compiler handles this case:

```
extern float foo(int);
float a[1024], b[1024];

Before Vectorization
#pragma simd vectorlength(8)
for (int k=0; k<1024; k++){
     x = foo(i);
     a[i] = x + b[i];
}

After vectorization (assuming 128-bit vector register size)
for (int k=0; k<1024; k+=8){
     x0 = foo(k+0);
     x1 = foo(k+1);
     x2 = foo(k+2);
     x3 = foo(k+3);
     vector_x = pack(x0, x1, x2, x3);
```

```
      a[k:k+3] = vector_x + b[k:k+3];

      x0 = foo(k+4);
      x1 = foo(k+5);
      x2 = foo(k+6);
      x3 = foo(k+7);
      vector_x = pack(x0, x1, x2, x3);
      a[k+4:k+7] = vector_x + b[k+4:k+7];
}
```

For the loop at line 92 with the function call, the performance improvement is achieved when SIMD vectorization and elemental function are used together.

References

For more information on SIMD vectorization, Intel Compiler automatic vectorization, SIMD-enabled Functions and examples of using other Intel® Cilk™ Plus constructs refer to:

"A Guide to Autovectorization Using the Intel® C++ Compilers"
"Requirements for Vectorizing Loops"
"Requirements for Vectorizing Loops with #pragma SIMD"
"Getting Started with Intel® Cilk™ Plus Array Notations"
"SIMD Parallelism using Array Notation"
"Intel® Cilk™ Plus Language Extension Specification"
"SIMD-enabled functions: Writing data parallel code in C/C++ using Intel® Cilk™ Plus"
"Using Intel® Cilk™ Plus to Achieve Data and Thread Parallelism"

Optimization Notice

Intel's compilers 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 SSE2, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804