White Paper

Program Optimization through Loop Vectorization

Overview

In this white paper, we will use a very simplified finite difference stencil computation of the following form:

```
for (i=0; i<dx; i++)
   for (r=1; r<=2; r++)
      u[i]+=c[r]*( v[i+ r*dx] + v[i- r*dx] );
```

to highlight some of the important points for taking advantage of the Intel® Many Integrated Core Architecture (Intel® MIC Architecture) vector units.

In this example, dx represents the leading dimension of a two dimensional array. For the sake of simplicity, we will assume arrays u and v are padded so that offsets [-2*dx,...,3*dx] can be accessed safely. A more general case is presented in Reference (1).

Single Instruction Multiple Data (SIMD) parallelism enhances the performance of computationally intensive applications that execute the same operation on distinct elements in a data set. With Intel® MIC Architecture 512-bit vector units, performance improvement in our single-precision example is obtained by performing operations applied to sixteen packed single-precision, floating-point values in the same clock cycle. Schematically, assume u[j:16]

as the pseudo-code representation of the sixteen single-precision, float-point values `u[j], u[j+1], ..., u[j+15]`, the SIMD vectorization can be represented as follows.

```
for (i=0; i<dx; i+=16)
   for (r=1; r<=2; r++)
      u[i:16]+=c[r]*( v[i+r*dx:16] + v[i-r*dx:16] );
```

In the above pseudo code, we assumed that dx is a multiple of 16 in order to keep it simple.

Vectorization Strategies

Intel® compilers provide the following ways to generate vector instructions:

- Inline assembly code
- SIMD vector intrinsics
- C++ SIMD Vector Class
- OpenMP* 4.0 (SIMD part)
- Intel® Cilk™ Plus (SIMD part)
    - SIMD pragma
    - SIMD enabled function
    - Array Notation
- Auto-vectorization

With inline assembly and SIMD vector instrinsics, the programmer has full responsibility and control over the vector code generated, but at the cost of writing very low level code, which negatively impacts productivity and maintainability. In this white paper, we try to always use higher-level coding with an eye on achieving optimal performance.

C++ SIMD Vector Class is a C++ Class abstraction to SIMD vector intrinsics. Overloaded operators, class member functions, and friend functions make the program easier to write and maintain (i.e., has the look and feel of normal C/C++ code). However, the class definition itself is tied to the underlying hardware, so the programmer still has to be operating on the 512-bit vector data at a time.

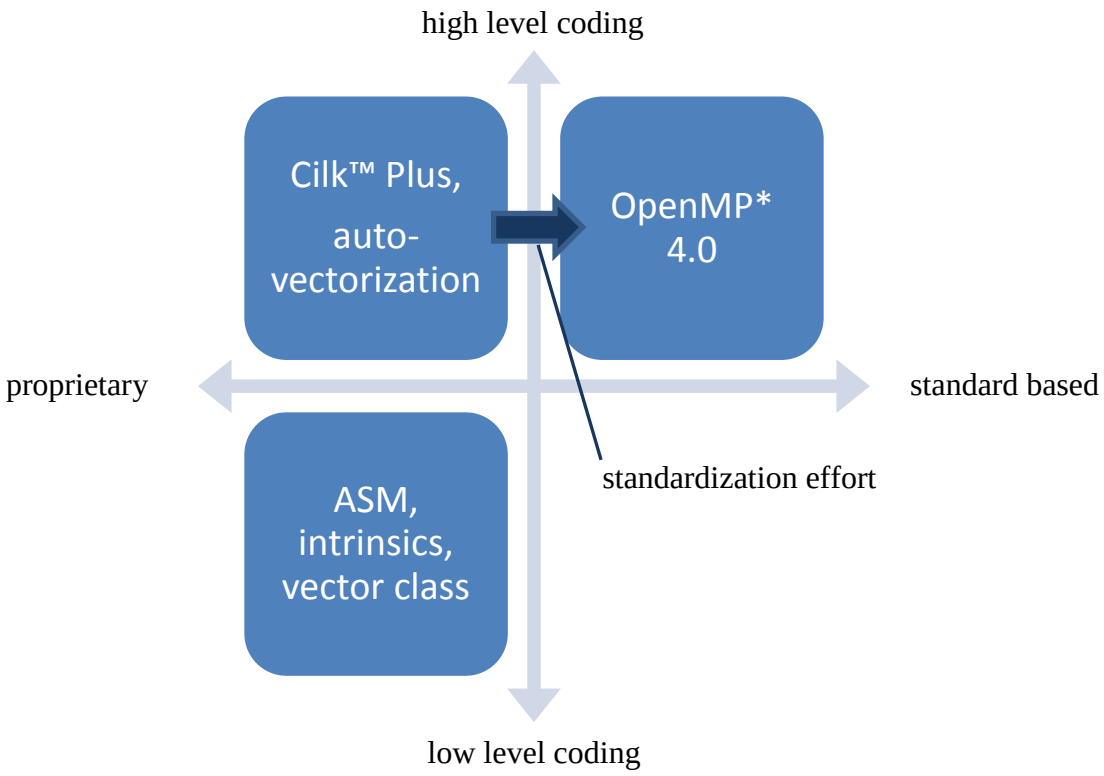
SIMD vector support in OpenMP 4.0 (2) provides a standard-based method of writing vector code explicitly. Intel has made a major contribution in the standardization effort. OpenMP has become a programmer's favorite way to write threaded code. Likewise, we expect standard-based vector programming to have a wide acceptance among the programming community.

Intel Cilk Plus (3) extension drives our standardization effort. A major part of it has been incorporated into OpenMP 4.0, and we have been in discussions with C/C++ language standard bodies to make the vector programming construct part of the language itself. Furthermore, we are working to add Intel Cilk Plus support to the GCC and LLVM Compilers.

Finally, Intel Compilers have an industry leading auto-vectorization technology. However, with 16-way vector hardware, simplistically speaking, even 80% vectorizable code, which is

rarely the case for auto-vectorization, will yield only 4x speedup potential, per Amdahl's law. For example, let's assume scalar execution of the code takes 100 seconds, and "80 seconds" of that can be 16-way vectorized, which achieves 16x speedup. The code will then take 25 seconds to execute: 20 seconds for the non-vectorized portion and 5 seconds for the vectorized portion. The total speedup is therefore only 4x (= 100/25). As such, it is imperative for programmers to look far beyond auto-vectorization to reach the full potential of the Intel® MIC Architecture vector units.

The following diagram illustrates the relationship among the different approaches.



Data Alignment Optimization

One of the most important aspects of vectorizing for Intel MIC Architecture is data alignment. When the data is aligned at 64-byte boundary and the compiler knows this alignment information, 512-bit vector load/store can be performed with one load/store instruction. The load can also be represented as the memory operand of an instruction such as vector add. If the compiler does not know the data alignment, which is typically the case, or is misaligned, the compiler is forced to generate less efficient code (in some cases significantly less efficient code sequences). As such, it is important to optimize the alignment of data and appropriately communicate the information to the compiler.

If you let the compiler vectorize (OpenMP, Intel Cilk Plus, or auto-vectorization), inspection of -vec-report6 output would show whether the compiler used aligned or unaligned memory

references. An alternative approach is deciphering the assembly code. When poor memory access behavior is detected, you may decide to improve data alignment to achieve better performance from vectorized code.

Aligned heap memory allocation can be achieved with calls to

```
void* _mm_malloc (size_t size, size_t base)
```

Alignment assertion `align(base)` can be used on statically allocated variables:

```
__declspec(align(64)) float temp[SIZE];  /* local temp array is 64byte aligned */
```

The equivalent Fortran directive is

```
!DIR$ ATTRIBUTES ALIGN: base :: variable
```

Alignment assertion `__assume_aligned(pointer, base)` can be used on pointers:

```
__assume_aligned(ptr, 64);  /* ptr is 64B aligned */
```

Fortran programmers would use the following to assert that the address of A(1) is 64-byte aligned.

```
!DIR$ ASSUME ALIGNED A(1): 64
```

Let the Compiler Generate Vector Code

Our example code was in the following form.

```
 for (i=0; i<dx; i++)
    for (r=1; r<=2; r++)
       u[i]+=c[r]*( v[i+ r*dx] + v[i- r*dx] );
```

Typical stencil code doing the same computation as above is written as

```
for (int i= 0; ix<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[i+2*dx] + v[i-2*dx]);
}
```

Figure 1: Initial version of code to implement our finite difference example.

and as such we use this as our starting point. Vectorizing this with OpenMP 4.0

```
#pragma omp simd  // OpenMP 4.0
for (int i= 0; ix<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[i+2*dx] + v[i-2*dx]);
}
```

and with Intel Cilk Plus SIMD pragma are strikingly similar since the former is modeled after the latter in our standardization effort.

```
#pragma simd // CilkPlus
for (int i= 0; ix<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[i+2*dx] + v[i-2*dx]);
}
```

With Intel Cilk Plus array notation, it can be written as a single statement

```
// Array Notation (long vector)
u[0:dx] += c1* (v[0+  dx:dx] + v[0-  dx:dx])
         + c2* (v[0+2*dx:dx] + v[0-2*dx:dx]);
```

or more explicitly in 512-bit short vector form.

```
dx1 = dx – (dx & 0xF);
for (int i= 0; ix<dx1; i+=16) {
   // Array Notation (short vector, 16 element at a time)
   u[i:16] += c1* (v[i+  dx:16] + v[i-  dx:16])
            + c2* (v[i+2*dx:16] + v[i-2*dx:16]);
}
for (int i= dx1; ix<dx; i++) { // remainder
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[i+2*dx] + v[i-2*dx]);
}
```

Auto-vectorization of our simple stencil loop is possible, and there are ways to improve the odds of success (e.g., writing source code that can be vectorized, using compiler flags to indicate lack of pointer aliasing, etc.). However, we will leave those topics to (4) and (5).

The compiler can use 'versioning' where data alignment is unclear by testing for alignment at runtime and branching the execution to a faster version of the loop assuming required alignment or a slower one assuming unaligned data. However, this will add more overhead, will increase code size and compile time, and cannot deal with exponential growth in the number of unknowns. As such, explicitly optimizing the data alignment is an essential part of obtaining optimal performance.

Coding with C++ SIMD Vector Class

A more aggressive approach is to program vectorized loops directly using SIMD instructions. There are multiple ways to accomplish this. The most labor intensive way is to program directly in assembly language. Alternatively, the compiler intrinsic support makes SIMD coding slightly easier and provides a straightforward way to combine hardware level SIMD instructions with high level C/C++ instructions in the same source code (6). When the application mainly consists of arithmetic and logical operations like our stencil code, Intel® C++ classes for SIMD are good alternatives to using SIMD intrinsic.

Intel C++ classes for SIMD operate on arrays, or vectors of data, in parallel. Integer vector (Ivec) classes and floating-point vector (Fvec) classes are supported. On Intel MIC

Architecture, the classes are named based on the underlying type of operation: F32vec16 operate on 16-packed 32-bit floating-point data, and F64vec8 operate on 8-packed 64-bit floating-point data. Standard operators are `+,+=,-,-=,*,*=,/,/=`.

Examples of advanced operators are Square Root, Reciprocal, and Reciprocal Square Root. It is worth noting that Intel MIC Architecture-aligned loads and stores require 64-byte data alignment. In our code example we added preamble loops that allow the main loop to issue SIMD instructions from aligned addresses of arrays *v* and *u*. Figure 2 exemplifies the use of the `F32vec16` class to implement vectorization in the *y*-direction loop via vector classes. The integer value `align_offset` is the array index obtained from a preamble loop that aligns the array pointers to a 64-byte boundary. Consequently, `vec_v` and `vec_u` are pointers to the aligned memory addresses `v+align_offset` and `u+align_offset`, respectively.

For clarity purposes, we assume arrays u and v with the same data alignment boundary. Advancing one single-precision array element in `vec_v` is equivalent to advancing sixteen single-precision elements in array `v`:. The loop trip count `vec_loop_trip` and stride `vec_dx` are 16 times smaller than their counterpart `dx` in the original scalar version. For each constant coefficient `ci` the constructor `F32vec16` packs sixteen identical copies of `ci` into `vec_ci`.

```
#include <micvec.h>

// Coefficients loaded in MIC vector registers
const F32vec16 vec_c1(c1), vec_c2(c2);

// Offset in v[] needed to get 64 byte boundary
const int align_offset = (64 - ((uintptr_t)v) % 64)/ sizeof(float);

// Scalar pre-loop (first non 64byte aligned elements)
#pragma loop count max(15)
for (int i=0; i< align_offset; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[x+2*dx] + v[i-2*dx]);
}

// Aligned vector register arrays – assume u & v with same alignment
F32vec16* vec_v = (F32vec16*)(v+align_offset);
F32vec16* vec_u = (F32vec16*)(u+align_offset);
const int vec_loop_trip = (dx - align_offset) / 16;
const int vec_dx = dx / 16;

// Loop on vector registers
for(int i=0; i<vec_loop_trip; i++) {
   vec_u[i] += vec_c1* (vec_v[i+  vec_dx] + vec_v[i-  vec_dx])
             + vec_c2* (vec_v[i+2*vec_dx] + vec_v[i-2*vec_dx]);
}

// Scalar post-loop (remaining elements)
#pragma loop count max(15)
for (int i=align_offset+16*vec_loop_trip; i<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[i+2*dx] + v[i-2*dx]);
}
```

Figure 2: Using Intel® C++ classes to implement SIMD vectorized loops. Each iteration in the main loop operates on sixteen packed elements of vec_v and vec_v.

These classes also provide a path to maintain assembler/intrinsics level code that can be easily transitioned to new hardware language extensions like Intel® Advanced Vector Extensions (Intel® AVX) (7). By generalizing the code with C macro definitions for the desired data alignment and the number of floating-point scalars per register, the code can be recompiled for Intel MIC Architecture, Intel AVX, or Intel® Streaming SIMD Extensions (Intel® SSE). Figure 3 extends our example to all Intel® Architecture by abstracting the machine specifics such as number of elements and alignment requirements, providing easy portability between Intel® Xeon® processors and Intel MIC Architecture. Note that C-macros like BUILD_FOR_MIC and BUILD_FOR_AVX, as in the example, can be substituted with the compiler-generated macros __MIC__ and __AVX__, respectively.

```
// Select hardware register size
#if defined(BUILD_FOR_MIC)
#  include <micvec.h>
#  define FLOAT_VECTOR F32vec16
#  define GOOD_ALIGN   64
#  define NUM_FLOATS   16
#elif defined(BUILD_FOR_AVX)
#  include <dvec.h>
#  define FLOAT_VECTOR F32vec8
#  define GOOD_ALIGN   32
#  define NUM_FLOATS   8
#elif defined(BUILD_FOR_SSE)
#  include <dvec.h>
#  define FLOAT_VECTOR F32vec4
#  define GOOD_ALIGN   16
#  define NUM_FLOATS   4
#else
#error “Choose the build target, please.”
#endif

// Coefficients loaded in the vector registers
const FLOAT_VECTOR vec_c1(c1), vec_c2(c2);

// Offset in v[] needed to get GOOD_ALIGN-byte boundary
const int align_offset = (GOOD_ALIGN - ((uintptr_t)v) % GOOD_ALIGN)/ sizeof(float);

// Scalar pre-loop (first non GOOD_ALIGN-byte aligned elements)
#pragma loop count max(NUM_FLOATS-1)
for (int i=0; i< align_offset; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[x+2*dx] + v[i-2*dx]);
}

// Aligned MIC vector register arrays – assume u & v with same alignment
FLOAT_VECTOR * vec_v = (FLOAT_VECTOR*)(v+align_offset);
FLOAT_VECTOR * vec_u = (FLOAT_VECTOR*)(u+align_offset);
const int vec_loop_trip = (dx - align_offset) / NUM_FLOATS;
const int vec_dx = dx / NUM_FLOATS;

// Loop on MIC vector registers
for(int i=0; i<vec_loop_trip; i++) {
   vec_u[i] += vec_c1* (vec_v[i+  vec_dx] + vec_v[i-  vec_dx])
             + vec_c2* (vec_v[i+2*vec_dx] + vec_v[i-2*vec_dx]);
}

// Scalar post-loop (remaining elements)
#pragma loop count max(NUM_FLOATS-1)
for (int i=align_offset+NUM_FLOATS*vec_loop_trip; i<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx])
         + c2* (v[i+2*dx] + v[i-2*dx]);
}
```

Figure 3: The Intel® C++ vector class vectorized loop example is extended to all Intel® Architecture by abstracting the machine specifics such as the number of elements and alignment requirements via C-macros. This allows easy portability between Intel® Xeon® processors and Intel MIC Architecture.

These classes also provide a path to maintain assembler/intrinsics-level code that can be easily transitioned to new hardware language extensions like Intel AVX (7; 8). By

generalizing the code with C macro definitions for the desired data alignment and the number of floating-point scalars per register, the code can be easily recompiled for Intel MIC Architecture, Intel AVX, or Intel SSE.

Alignment Optimization with Intel Compiler-Generated Vector Code

A variant of the code from Figure 2 can be formulated using array notation as shown in Figure 4. In this example we also demonstrate how to use `__assume_aligned()` declarations to guide the Intel compiler with data alignment information. With the right set of optimizations, compiler-generated vector code can perform similarly to intrinsic code. Declarations `__assume_aligned(u,64)` and `__assume_aligned(v,64)` indicate that base addresses of arrays u and v are 64-byte aligned. An alternative implementation could also use declarations of the type `__assume(<var> %16 == 0)` to indicate that the integer scalar `<var>` is a multiple of 16 and consequently if the address of `v[i]` is 64-byte aligned, then the offset address `v[i+ <var>]` is also 64-byte aligned. Note that using `__assume()` and `__assume_aligned()` is not required. They are resources for the programmer to minimize compiler code generation by asserting known alignment properties.

```
// Assume u and v[k*dx], k=-2,..,2, are 64-byte aligned arrays
__assume_aligned(u,64);
__assume_aligned(v,64);
float* v_pdx  = &(v[   dx]); __assume_aligned(v_pdx, 64);
float* v_mdx  = &(v[  -dx]); __assume_aligned(v_mdx, 64);
float* v_p2dx = &(v[ 2*dx]); __assume_aligned(v_p2dx,64);
float* v_m2dx = &(v[-2*dx]); __assume_aligned(v_m2dx,64);
const int vec_loop_trip = dx/16;

//  loop using vector registers (16 single precision floats)
for(int i=0; i<vec_loop_trip; i=+16) {
   u[i:16] += c1* (v_pdx [i:16] + v_mdx [i:16])
            + c2* (v_p2dx[i:16] + v_m2dx[i:16]);
}
```

Figure 4: Example of using Intel® Cilk™ Array Notation to explicitly implement vectorized loops. In this example, since all arrays are asserted to be 64-byte aligned and the loop trip is known to be a multiple of 16, the compiler generated only aligned loads and stores and no remainder loop code.

The code in Figure 4 can also be rewritten to simultaneously support Intel MIC Architecture and all Intel Xeon processor variants by using C-macros as shown in Figure 3.

Note that alignment information can also be provided to the compiler using pragmas/directives. When not all the arrays are known to be aligned, one can use selective options in the pragma/directive as shown in the Fortran example in Figure 5.

```
      integer n
      real v(*), w(*), u(*)
!dir$ ASSUME_ALIGNED v:64, w:64
      do i=1,n
         v(i) = u(i)
         w(i) = 5.0 * u(i)
      enddo
```

Figure 5: Fortran example of ASSUME_ALIGNED directive with hints for arrays v and w. Unaligned access is only required for array u.

More Realistic 3-D Stencil Example

In the following example we compare performance of three variants of a simple 9-point finite difference operator applied over the second dimension of a 3-dimensional array v. Figure 6 shows a plain C implementation of the finite differences computed over v. In Figure 7 the code is implemented with explicit array alignment handling. **Error! Reference source not found.** lists the C++ vector class implementation.

```
#pragma simd
for (int i=0; i<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx]) + c2* (v[i+2*dx] + v[i-2*dx])
         + c3* (v[x+3*dx] + v[i-3*dx]) + c4* (v[x+4*dx] + v[i-4*dx]);
}
```

Figure 6: Plain C implementation of a 9-point finite difference on the second dimension of the 3-dimensional array v.

```
const int align_offset = (GOOD_ALIGN - ((uintptr_t)v) % GOOD_ALIGN)/ sizeof(float);

#pragma loop count max(NUM_FLOATS-1)
for (int i=0; i< align_offset; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx]) + c2* (v[x+2*dx] + v[i-2*dx])
         + c3* (v[x+3*dx] + v[i-3*dx]) + c4* (v[x+4*dx] + v[i-4*dx]);
}

float * aligned_v = (float*)(v+align_offset);
float * aligned_u = (float*)(u+align_offset);
const int vec_loop_trip = (dx - align_offset) / NUM_FLOATS;

#pragma vector aligned
#pragma simd
for (int i=0; i<vec_loop_trip*NUM_FLOATS; i++) {
   aligned_u[ix] += c1* (aligned_v[i+  dimx] + aligned_v[i-  dimx])
                  + c2* (aligned_v[i+2*dimx] + aligned_v[i-2*dimx])
                  + c3* (aligned_v[i+3*dimx] + aligned_v[i-3*dimx])
                  + c4* (aligned_v[i+4*dimx] + aligned_v[i-4*dimx]);
}

#pragma loop count max(NUM_FLOATS-1)
for (int i=align_offset+NUM_FLOATS*vec_loop_trip; i<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx]) + c2* (v[i+2*dx] + v[i-2*dx])
         + c3* (v[x+3*dx] + v[i-3*dx]) + c4* (v[x+4*dx] + v[i-4*dx]);
}
```

Figure 7: A 9-point finite difference on the second dimension of the 3-dimensional array v implemented with explicit array alignment handling.

```
#include <immintrin.h>
#define PFTCH1(addr, DIST) _mm_prefetch((char const*) ((addr)+(DIST)), _MM_HINT_T0);
#define PFTCH2(addr, DIST) _mm_prefetch((char const*) ((addr)+(DIST)), _MM_HINT_T1);
#define L1_DIST   4
#define L2_DIST   16
#define PREFETCH(addr) \
  PFTCH1((addr), (L1_DIST)) \
  PFTCH2((addr), (L2_DIST))

const int align_offset = (GOOD_ALIGN - ((uintptr_t)v) % GOOD_ALIGN)/ sizeof(float);

#pragma loop count max(NUM_FLOATS-1)
for (int i=0; i< align_offset; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx]) + c2* (v[x+2*dx] + v[i-2*dx])
         + c3* (v[x+3*dx] + v[i-3*dx]) + c4* (v[x+4*dx] + v[i-4*dx]);
}

const FLOAT_VECTOR vec_c1(c1), vec_c2(c2), vec_c3(c3), vec_c4(c4);
FLOAT_VECTOR * vec_v = (FLOAT_VECTOR*)(v+align_offset);
FLOAT_VECTOR * vec_u = (FLOAT_VECTOR*)(u+align_offset);
const int vec_loop_trip = (dx - align_offset) / NUM_FLOATS;
const int vec_dx = dx / NUM_FLOATS;

#pragma noprefetch
for(int i=0; i<vec_loop_trip; i++) {
   PREFETCH(&vec_u[i])
   PREFETCH(&vec_v[i+  dimx])  PREFETCH(&vec_v[i-  dimx])
   PREFETCH(&vec_v[i+2*dimx])  PREFETCH(&vec_v[i-2*dimx])
   PREFETCH(&vec_v[i+3*dimx])  PREFETCH(&vec_v[i-3*dimx])
   PREFETCH(&vec_v[i+4*dimx])  PREFETCH(&vec_v[i-4*dimx])
   vec_u[i] += vec_c1*vec_v[i+  vec_dx] + vec_c1*vec_v[i-  vec_dx]
             + vec_c2*vec_v[i+2*vec_dx] + vec_c2*vec_v[i-2*vec_dx]
             + vec_c3*vec_v[i+3*vec_dx] + vec_c3*vec_v[i-3*vec_dx]
             + vec_c4*vec_v[i+4*vec_dx] + vec_c4*vec_v[i-4*vec_dx];
}

#pragma loop count max(NUM_FLOATS-1)
for (int i=align_offset+NUM_FLOATS*vec_loop_trip; i<dx; i++) {
   u[i] += c1* (v[i+  dx] + v[i-  dx]) + c2* (v[i+2*dx] + v[i-2*dx])
         + c3* (v[x+3*dx] + v[i-3*dx]) + c4* (v[x+4*dx] + v[i-4*dx]);
}
```

Figure 8: C++ vector class implementation of a 9-point finite difference on the second dimension of the 3-dimensional array v. This example also shows one alternative to directly control first-level (L1) and second-level (L2) array prefetching by using C prefetch intrinsics.

For these experiments, each variant of the code ran single threaded on a single Intel MIC Architecture core to highlight the differences in vectorization. This is a synthetic test configuration just to emphasize differences between these example implementations. In a real-case scenario, you should test such optimization changes in the actual parallel implementation of the code. Performance improvement was measured by how much faster the Array Alignment (Figure 7) and C++ Vector Class (**Error! Reference source not found.**) implementations ran compared with the baseline version in Figure 6. The C++ Vector Class has the additional feature of using explicit calls to C prefetch intrinsics to control the first-level (L1) and second-level (L2) prefetching of the arrays `vec_u` and `vec_v`.

For the tests, arrays `u` and `v` where defined with dimensions `dx=464, dy=224,` and `dz=840`. Sample codes were built with Intel C/C++ compiler version 14.0.0.080 and the single-thread tests ran on an Intel® Xeon Phi™ 7120 coprocessor. Ten iterations of the 9-point finite difference were applied to each point of the input array. Table 1shows the runtime in seconds for the three implementations described in Figure 6 through Figure 8. The first row describes runtimes for 32-byte aligned arrays and the second row contains runtimes for 64-byte aligned arrays. On these particular examples over 23% improvement was measured for source code that exploited the knowledge of aligned arrays in the same boundaries of 512-bit vector registers of the Intel MIC Architecture.

Table 1: Runtime in seconds for a single-threaded run of the three implementations. Results are shown with arrays v and u aligned at either 32-byte or 64-byte boundaries. The bottom row shows improvement when moving from 32-byte to 64-byte aligned array boundaries.

data alignment	baseline code	Array Alignment	C++ Vector Class
32 Bytes	6.0	4.0	3.9
64 Bytes	4.9	3.2	2.8
improvement:	23%	26%	37%

A second fact to note in Table 1is the performance gap between our different versions of code. In this case the baseline code runs between 50% (32-byte alignment) and 70% (64-byte alignment) slower than the C++ vector class version, and the array alignment code runs up to 13% (64-byte) slower when compared to the vector class counterpart version. One major reason for this performance difference is that the vector class implementation in Figure 8 contains explicit code to control first-level (L1) and second-level (L2) array prefetching via C prefetch intrinsics. In the next paragraph we take advantage of the Intel compiler features to control data prefetch and revisit both the baseline code and the array alignment code.

The compiler provides a prefetch distance tuning global option for the Intel MIC architecture:

```
-opt-prefetch-distance=n1[,n2]
        n1 specifies the distance for first-level prefetches into L2
        n2 specifies prefetch distance for second-level prefetches from L2 to L1 (n2<=n1)
```

Table 2 contains the same experiments as before, but now with executable binaries rebuilt with the additional compilation option "`-opt-prefetch-distance=8,2`" to fine-tune the distance used for prefetching arrays `v` and `u`. With specific prefetching tuning, the alignment array version of the code delivers similar runtimes to the vector class implementation. The performance of the baseline code also improves but still runs 8% slower for 32-byte alignment and 14% slower for 64-byte alignment. One can still see an overall improvement of 30% across all implementations by using arrays with 64-byte boundary alignment.

Table 2: Runtime in seconds for a single-threaded run of the three implementations compiled with option `-opt-prefetch-distance=8,2` to fine-tune L2 and L1 prefetching distances. Results are shown with arrays v and u aligned at either 32-byte or 64-byte boundaries. The bottom row shows improvement when moving from 32-byte to 64-byte aligned array boundaries.

data alignment	baseline code	Array Aligned	C++ Vector Class
32 Bytes	4.2	3.9	3.9
64 Bytes	3.3	2.9	2.8
improvement:	28%	36%	37%

It is worth noting that the Intel compiler provides a wide range of ways to control code generation for prefetching. For example, you could use the alternative of directive support for loop prefetches as either prefetch pragma support for C loops

```
#pragma prefetch var:hint:distance
```

or prefetch directive support for Fortran loops

```
CDEC$ prefetch var:hint:distance
```

The compiler documentation brings detailed information about prefetching. A deep-dive discussion is documented in the Compiler Prefetching for the Intel® Xeon Phi™ coprocessor (8) presentation.

Summary

Good vectorization is of fundamental importance to take full advantage of SIMD-based data parallelism. Intel Compilers provide a wide range of ways to generate well-optimized vector instructions. High-level coding can take advantage of auto-vectorization, Intel Cilk Plus extensions, and OpenMP 4.0 to improve the amount of vectorization by using either our proprietary syntax, or with the portability of a widely adopted standard like OpenMP. These high-level approaches are the recommended ways to improve vectorization because they are proven to generate good vector code; they are straightforward to implement, maintain; and are also very portable. In some cases, applications may contain very specific and delimited region(s) of code where the majority of the compute time is spent, and squeezing every bit of performance is a critical matter. Even under those circumstances, vector SIMD classes can often provide performance that matches intrinsic/assembly code, while maintaining the look and feel of high-level coding. Lower level coding with intrinsics and assembly code should be limited to a small section of code where overall performance gain justifies the higher program maintenance costs.

References

1. 3D Finite Differences on Multi-core Processors. [Online] http://software.intel.com/en-us/articles/3d-finite-differences-on-multi-core-processors/.
2. OpenMP 4.0 specification. [Online] http://www.openmp.org.
3. Intel® Cilk™ Plus Support. [Online] http://software.intel.com/en-us/articles/intel-cilk-plus-support.
4. A Guide to Auto-vectorization with Intel® C++ Compilers. [Online] http://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers/.
5. Quick-Reference Guide to Optimization with Intel® Compilers version 13. [Online] http://software.intel.com/sites/default/files/Compiler_QRG_2013.pdf.
6. Intel® C++ Intrinsics Reference. [Online] http://software.intel.com/sites/products/documentation/studio/composer/en-us/2011/compiler_c/intref_cls/common/intref_bk_intro.htm.
7. Intel Advanced Vector Extensions (AVX). [Online] http://software.intel.com/en-us/avx.
8. Compiler Prefetching for the Intel® Xeon Phi™ coprocessor. [Online] http://software.intel.com/sites/default/files/managed/5d/f3/5.3-prefetching-on-mic-4.pdf.

About the Authors

Leonardo Borges, Hideki Saito, and Philippe Thierry are members of the Intel Software & Services Group.

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

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

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

Intel, the Intel logo, Cilk, Xeon, and Xeon Phi are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others
Copyright© 2012 Intel Corporation. All rights reserved.

▲For more complete information about performance and benchmark results, visit www.intel.com/benchmarks

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 SSE3 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