Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 269764, 14 pages
http://dx.doi.org/10.1155/2015/269764

Research Article

Effective SIMD Vectorization for Intel Xeon Phi Coprocessors

Xinmin Tian,[1] Hideki Saito,[1] Serguei V. Preis,[2] Eric N. Garcia,[1] Sergey S. Kozhukhov,[2] Matt Masten,[1] Aleksei G. Cherkasov,[2] and Nikolay Panchenko[2]

[1]*Mobile Computing and Compilers Software and Service Group, Intel Corporation, Santa Clara, CA 95054, USA*
[2]*Mobile Computing and Compilers Software and Service Group, Intel Corporation, 6/1 Prospect Akademika, Novosibirsk 125009, Russia*

Correspondence should be addressed to Xinmin Tian; xinmin.tian@intel.com

Received 15 May 2014; Accepted 29 September 2014

Academic Editor: Sunita Chandrasekaran

Efficiently exploiting SIMD vector units is one of the most important aspects in achieving high performance of the application code running on Intel Xeon Phi coprocessors. In this paper, we present several effective SIMD vectorization techniques such as less-than-full-vector loop vectorization, Intel MIC specific alignment optimization, and small matrix transpose/multiplication 2D vectorization implemented in the Intel C/C++ and Fortran production compilers for Intel Xeon Phi coprocessors. A set of workloads from several application domains is employed to conduct the performance study of our SIMD vectorization techniques. The performance results show that we achieved up to 12.5x performance gain on the Intel Xeon Phi coprocessor. We also demonstrate a 2000x performance speedup from the seamless integration of SIMD vectorization and parallelization.

1. Introduction

The Intel Xeon Phi coprocessor is based on the Intel Many Integrated Core (Intel MIC) architecture, which consists of many small, power efficient, in-order cores, each of which has a powerful 512-bit vector processing unit (SIMD unit) [1]. It is designed to serve the needs of applications that are highly parallel, make extensive use of SIMD vector operations, or are memory bandwidth bound. Hence, it is targeted for highly parallel, high performance computing (HPC) workloads [2] in a variety of fields such as computational physics, chemistry, biology, and financial services [3]. The Intel Xeon Phi Coprocessor 5110P has the following key specifications:

(i) 60 cores, 240 threads (4 threads/core),
(ii) 1.053 GHz,
(iii) 1 TeraFLOP double precision theoretical peak performance,
(iv) 8 GB memory with 320 GB/s bandwidth,
(v) 512 bit wide SIMD vector engine,
(vi) 32 KB L1, 512 KB L2 cache per core,
(vii) fused multiply-add (FMA) support.

One Teraflop theoretical peak performance is computed as follows: 1.053 GHz × 60 cores × 8 double precision elements in SIMD vector × 2 flops per FMA. As such, any compute bound applications trying to achieve high performance on Intel Xeon Phi coprocessors need to exploit a high degree of parallelism and wide SIMD vectors. Using a 512-bit vector unit, 16 single precision (or 8 double precision) floating point (FP) operations can be performed as a single vector operation. With the help of the fused multiply-add (FMA) instruction, up to 32 FP operations can be performed at each core at each cycle. In comparison to the current 128-bit SSE and 256-bit AVX vector extensions, this new coprocessor can pack up to 8x and 4x the number of operations into a single instruction, respectively.

Wider SIMD vector units cannot be effectively utilized by simply extending the vectorizer for Intel SSE and Intel AVX architecture. Consider the following simple example. There exists a scalar loop that executes N-iterations. Using the vector length of VL, a vector loop would execute floor (N/VL) full vector iterations followed by N mod VL scalar remainder iterations. Unless N is sufficiently larger than VL, executing N mod VL scalar iterations can still be a significant portion of the vector execution of such a loop. In what follows, we

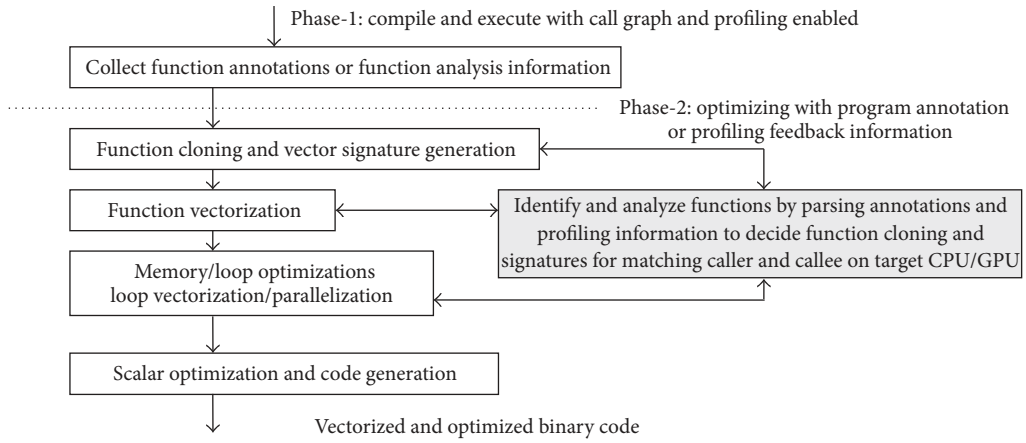

Figure 1: SIMD vector compilation infrastructure for function and loop vectorization.

will discuss two approaches in handling such "less-than-full-vector" situations: the first technique is masked vectorization and the second technique is small matrix optimization and 2-dimensional (2D) vectorization.

Furthermore, architectural or microarchitectural differences between Intel Xeon Phi coprocessors and Intel Xeon processors necessitate that new compiler techniques be developed. This paper focuses on three SIMD vectorization techniques and makes the following contributions.

(i) We propose an extended compiler scheme to vectorize short trip-count loops and peeling and remainder loops that are classified as "less-than-full-vector" cases, with a masking capability supported by the Intel MIC architecture.

(ii) We describe our specific data alignment strategies for achieving optimal performance through vectorization, as the Intel MIC architecture is much more demanding on memory alignment than the Intel AVX architecture [4].

(iii) We describe our 2-dimensional vectorization method which is beyond the conventional loop vectorization for small matrix transpose and multiplication operations by fully utilizing long SIMD vector units, swizzle, shuffle, and masking support on the Intel MIC architecture.

The rest of this paper is organized as follows: Section 2 provides a high-level overview of Intel C/C++ and Fortran compilers. In Section 3, the compiler details of "less-than-full-vector" loop vectorization are described and discussed. Specific data alignment strategies for the Intel Xeon Phi coprocessor and the schemes of performing data alignment optimization are discussed in Section 4. Section 5 presents the 2D vectorization methods for small matrix transpose and multiplication. Section 6 discusses related work. Section 7 provides the performance results with a set of workloads and microbenchmarks. Section 8 concludes the paper.

2. Compiler Architecture for Vectorization

This section describes the Intel C/C++ and Fortran compiler support for the Intel Xeon Phi coprocessor at a high level with respect to loop vectorization and the translation and optimization of SIMD *vector* extensions [5–7]. The compiler translates serial C/C++ and Fortran code via automatic loop analysis or based on annotations using the SIMD pragma and *vector* attributes into SIMD instruction sequences. The compilation process is amenable to many optimizations such as loop parallelization, memory locality optimizations, classic loop transformations and optimizations, redundancy elimination, and dead code elimination before and after the loop/function vectorization. Figure 1 depicts the SIMD compilation infrastructure of the Intel C/C++ and Fortran compilers for automatic loop vectorization and compiling SIMD pragma, *vector* function annotations, and associated clauses. The framework consists of four major parts.

(i) Perform automatic loop analysis and identify and analyze programmer annotated functions and loops by parsing and collecting function and loop vector properties. In addition, our compiler framework can apply interprocedural analysis and optimization with profiling and call-graph creation for automatic function vectorization.

(ii) Generate vectorized function variants with properly constructed signatures via function cloning and vector signature generation.

(iii) Vectorize SIMD for loops that are identified by the compiler or annotated using SIMD extensions (#pragma SIMD can be used to vectorize outer loops) and cloned vector function bodies and all arguments by leveraging and extending our automatic loop vectorizer.

(iv) Enable classical scalar, memory, and loop optimizations and parallelization effectively, before or after loop and function vectorization, for achieving good performance.

```
float x, y[31];
for (k=0; k<31; k++) {
    x = x + fsqrt(y[k]);
}
```

Algorithm 1

```
float foo(float *y, int n)
{   int  k; float x = 10.0f;
    for (k = 0; k < n; k++) {
      x = x + fsqrt(y[k])
    }
    return x;
}
```

Algorithm 2

3. Less-than-Full-Vector Loop Vectorization with Masking

Intel Xeon Phi coprocessor provides long (512-bit) SIMD vector hardware support for exploiting more vector-level parallelism. The long SIMD vector unit imposes the requirement of packing more scalar loop iterations into a single vector loop iteration, which also results in more iterations in the peeling loop, and/or in the remainder loop remaining nonvectorized, due to the fact that they do not constitute the full SIMD vector (or less-than-full-vector) unit of Intel MIC architecture. For example, consider the short trip-count loop as shown in Algorithm 1.

When the loop is vectorized for Intel SSE2 with vector length = 4 (128-bit), the remainder loop will have 3 iterations. When the loop is vectorized for the Intel MIC architecture with vector length = 16 (512-bit), the remainder loop will have 15 iterations. In another situation, if the loop is unrolled by 16, then the remainder loop will have 15 iterations, leaving the remaining 15 iterations in a scalar execution form. Thus, vectorizing the peeling and remainder loops (i.e., short trip-count loop in general) is very important for the Intel MIC architecture. This section describes how to apply vectorization, with masking support, to peeling and remainder loops (i.e., short trip-count loop) with special guarding masks to prevent the SIMD code from exceeding original loop and memory access boundaries. At a high level, the following steps describe our vectorization scheme without vectorization of peeling and remainder loops.

(i) s0: select alignment, vector length, and unroll factor.

(ii) s1: generate alignment setup code.

(iii) s2: compute the trip count of the peeling loop.

(iv) s3: emit the scalar peeling loop.

(v) s4: generate the vector loop initialization code.

(vi) s5: emit the main vector loop.

(vii) s6: compute the trip count of the remainder loop.

(viii) s7: emit the scalar remainder loop.

Given the simple example as shown in Algorithm 2, the loop trip-count "n" and the pointer "y" ($\&y[0]$) have a memory alignment that is unknown at compile time.

On the Intel MIC architecture the vector length is 512 bits, which requires 64-byte alignment for efficient memory accesses. To achieve 64-byte aligned memory loads/stores, we need to pack 16 float (32-bit) elements for each single vector iteration and generate a peeling loop. Pseudocode 1 shows the vectorized loop based on the vectorization steps [s0, s1, ..., s7] described above. The "less-than-full-vector" loops, that is, the peeling and remainder loops, are not vectorized.

Note that we performed loop unrolling for the main vectorized loop, which allows the hardware to issue more instructions per cycle by hiding memory access latency and reducing branching. To enable the "less-than-full-vector" (i.e., peeling loop, remainder loop, or short trip-count loop) vectorization, the loop vectorization scheme is extended as below.

(i) s0: select alignment, vector length and unroll factor.

(ii) s1: generate alignment setup code.

(iii) s2: compute the trip count of peeling loop.

  (a) Create a vector of 16 elements with value $\langle 0, \ldots, 15\rangle$.

  (b) Create a vector of 16 elements with value $\langle$peeledTripCount, ..., peeledTripCount$\rangle$.

(iv) s3: emit the vectorized peeling loop with masking operations.

(v) s4: generate the main vector loop initialization code.

(vi) s5: emit the main vector loop.

(vii) s6: compute the trip count of the remainder loop.

  (a) Create a vector of 16 elements with the value $\langle$mainTripCount, ..., mainTripCount+15$\rangle$.

  (b) Create a vector of 16 elements with the value $\langle$origTripCount, ..., origTripCount$\rangle$.

(viii) s7: emit the vectorized remainder loop with masking operations.

Pseudocode 2 shows the vectorized loops based on the extended vectorization schemes [s0, s1, ..., s7] described as above.

In the cases of short trip-count loop vectorization of peeling and remainder loops with runtime trip-count and alignment checking, loops are vectorized as efficiently as possible. These loops are vectorized with optimal vector lengths and an optimal amount of profitable unrolling regardless of a known loop trip count. This provides better utilization of SIMD vector hardware without sacrificing the performance of short loops. This scheme allows us to completely eliminate

```
misalign = &y[0] & 63
peeledTripCount = (63 – misalign)/sizeof(float)
x = 10.0f;
do k0 = 0, peeledTripCount-1  // peeling loop
   x = x + fsqrt(y[k0])
enddo
x1_v512 = (m512)0
x2_v512 = (m512)0
mainTripCount = n – ((n – peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
  x1_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1:16]),x1_v512)
  x2_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo
// perform vector add on two vector x1_v512 and x2_v512
x1_v512 = _mm512_add_ps(x1_v512, x2_512);
// perform horizontal add on all elements of x1_v512, and
// the add x for using its value in the remainder loop
x = x + _mm512_hadd_ps(x1_512)
do k2 = mainTripCount, n // Remainder loop
   x = x + fsqrt(y[k2])
enddo
```

Pseudocode 1: Pseudocode without vectorizing "less-than-full-vector" loops.

```
misalign = &y[0] & 63
peeledTripCount = (63 – misalign) / sizeof(float)
x = 10.0f;
// create a vector: <0,1,2,…15>
k0_v512 = _mm512_series_pi(0, 1, 16)
// create vector: all 16 elements are peeledTripCount
peeledTripCount_v512 = _mm512_broadcast_pi32(peeledTripCount)
x1_v512 = (m512)0
x2_v512 = (m512)0
do k0 = 0, peeledTripCount-1, 16
   // generate mask for vectorizing peeling loop
   mask = _mm512_compare_pi32_mask_lt(k0_v512, peeledTriPCount_v512)
   x1_v512 = _mm512_add_ps_mask(_mm512_fsqrt(y[k0:16]), x1_v512, mask)
enddo
mainTripcount = n – ((n – peeledTripCount) & 31)
do k1 = peeledTripCount, mainTripCount-1, 32
  x1_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1:16]), x1_v512)
  x2_v512 = _mm512_add_ps(_mm512_fsqrt(y[k1+16:16]), x2_v512)
enddo
// create a vector: <mainTripCount,mainTripCount+1 … mainTripCount+15>
k2_v512 = _mm512_series_pi(mainTripCount, 1, 16)
// create a vector: all 16 elements has the same value n
n_v512 = _mm512_broadcast_pi32(n)
step_v512 = _mm512_broadcast_pi32(16)
do k2 = mainTripCount, n, 16 // vectorized remainder loop
   mask = _mm512_compare_pi32_mask_lt(k2_v512, n_v512)
   x1_v512 = _mm512_add_ps_mask(_mm512_fsqrt(y[k2:16]), x1_v512, mask)
   k2_v512 = _mm512_add_ps(k2_v512, step_v512)
enddo
x1_v512 = _mm512_add_ps(x1_v512, x2_512);
// perform horizontal add on 8 elements and final
// reduction sum to write the result back to x.
x = x + _mm512_hadd_ps(x1_512)
```

Pseudocode 2: Pseudocode with vectorizing "less-than-full-vector" loops using mask.

scalar execution of the loop in favor of masked SIMD vector code generation. Special properties of the mask are used to match unmasked code generation in most cases. For example, masked scalar memory loads that could be unsafe under an empty mask are considered safe under a remainder mask since it is never empty.

Without adding the capability of short trip-count loop vectorization, the loops in the ConvolutionFFT2D benchmark with 7 iterations and double precision data type would end up as a fully scalar execution. Applying vectorization with masking to these short trip-count loops results in a ~2x to ~5x speedup for the 7-iteration short trip-count (or less-than-full-vector) loops in the ConvolutionFFT2D benchmarks on the Intel MIC Architecture.

4. Alignment Strategy and Optimization

The Intel Xeon Phi coprocessor is much more sensitive to data alignment than the Intel Xeon E5 processor, so developing an Intel MIC oriented alignment strategy and optimization schemes is one of the key aspects for achieving optimal performance.

(i) Similar to Intel SSE4.2, the SIMD load+op instructions require vector size alignment, which is 64-byte alignment for the Intel MIC architecture. However, simple load/store instructions require the alignment information to be known at compile time on the Intel Xeon Phi coprocessor.

(ii) Different from prior Intel SIMD extensions, *all* SIMD load/store instructions including gather/scatter require at least element size alignment. Misaligned elements will cause a fault. This necessitates the Intel MIC architecture ABI [8] to require that all memory accesses be elementwise aligned.

(iii) There are no special unaligned load/store instructions in the Intel Initial Many Core Instruction (Intel IMCI) set. This is overcome by using unpacking loads and packing stores that are capable of dealing with unaligned (element-aligned) memory locations. Due to their unpacking and packing nature, these instructions cannot be directly used for masked loads/stores, except under special circumstances.

(iv) The faulting nature of masked memory access instructions in Intel IMCI adds extra complexity to those instructions addressing data outside paged memory and may fail even if actual data access is masked out. The exceptions are gather/scatter instructions.

Therefore, the compiler aggressively performs data alignment optimizations using traditional techniques such as alignment peeling and alignment multiversioning.

Alignment peeling implies the creation of a preloop that executes several iterations on unaligned data in order to reach an aligned memory address. As a result, most of these iterations are executed using aligned SIMD operations. The preloop can be vectorized with masking as described in Section 2. Unfortunately, this scheme works only for one set of coaligned memory addresses, and the others are assumed to be unaligned. In addition, our multiversioning optimization can be applied to the second set of coaligned locations by examining them dynamically. Aligned or unaligned operations are used based on the results of the examination.

For unmasked unaligned (element-aligned) vector loads and stores, the compiler uses unpacking/packing load and store instructions. They are safe in this scenario and perform much better than gather/scatter instructions. If the compiler cannot prove the safety of the entire address range of a particular memory access, it inserts a zero-mask check in order to avoid a memory fault. All instructions with the same mask are emitted under a single check to avoid execution under the empty mask and to eliminate multiple checks of the same condition.

Unpacking and packing instructions may cause fault when they are used with a mask, as they may address masked-out invalid memory. On-the-fly data conversion may cause fault even without masking. Thus, for unaligned masked and/or converting loads/stores, the compiler uses gather/scatter instructions instead of safety, even though this degrades performance. Memory faults would never happen if each memory access had at least one vector (64 bytes) of memory paged after its initial address. This can be achieved by padding each data section in the program and each dynamically allocated object with 64 bytes. For developers who are willing to do the padding to achieve optimal performance from masked code, the compiler knob-opt-assume-safe-padding was introduced. Under this knob, unaligned masked and/or converting load/store operations are emitted as unpacking loads/packing stores.

(i) In unmasked converting cases, as well as cases with peel/remainder masks, the compiler emits loads/stores directly. The mask in this case will work since it is dense.

(ii) For an arbitrary masking scenario, an unmasked load unpack instruction is used, which is safe due to the padding assumption, followed by a masked move (blend). The "nonempty-mask" check guarantees that the 64-byte padding is always enough for safety; that is, at least one item within the vector is to be loaded. Thus, the tail end of the memory access is within 64 bytes from meaningful data.

The safe-padding optimization has provided notable improvements on a number of benchmarks, for example, 10% gain on BlackScholes and selected Molecular Dynamics kernels.

5. Small Matrix Operations 2D Vectorization

Frequently seen in HPC workloads, operations on small matrices are a growing, profitable set of calculations for vectorization on Intel Xeon Phi coprocessors. With the wider SIMD unit support, the Intel C/C++ and Fortran compilers are enhanced to vectorize common operations on small matrices along 2 dimensions. Small matrices are matrices whose data can reside entirely in one or two 512-bit SIMD

```
real, dimension(4,4):: A, B, C
real sum
integer j, l, i
do j = 1, 4
  do l = 1, 4
    sum = 0.0
    do i = 1, 4
      sum = sum + A(i,l) * B(i,j)
    enddo
    C(l,j) = sum
  enddo
enddo
```

Algorithm 3: Small matrix multiplication summation.

Table 1: Contents of vector register A_v512 after load.

A_v512	A[1][1]	A[1][2]	A[1][3]	A[1][4]
	A[2][1]	A[2][2]	A[2][3]	A[2][4]
	A[3][1]	A[3][2]	A[3][3]	A[3][4]
	A[4][1]	A[4][2]	A[4][3]	A[4][4]

Table 2: Contents of vector register B_v512 after load.

B_v512	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[2][1]	B[2][2]	B[2][3]	B[2][4]
	B[3][1]	B[3][2]	B[3][3]	B[3][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]

Table 3: A′_v512 after zero initialization.

A′_v512	0	0	0	0
	0	0	0	0
	0	0	0	0
	0	0	0	0

registers. Consider the example Fortran loop nest with 32-bit float (or real) type as shown in Algorithm 3.

With nonunit stride references present in the inner loop of Algorithm 3, the conventional inner loop vectorization will not provide the most efficient vectorization of the loop nest. The outer loop vectorization faces similar issues. The Intel C/C++ and Fortran compiler employs the wider SIMD vector unit of the Intel MIC architecture and vectorizes this example loop across all three loop nesting levels, named as 2-dimensional (2D) vectorization on small matrices.

The vectorization approach is detailed below with vector intrinsic pseudocode. For visualization, Tables 1–13 depict a snapshot of the various vector unit contents after each corresponding instruction. Tables 1–13 represent a vector unit, whose name is in the leftmost column and its contents in the rightmost four columns. Of the rightmost four columns, the lowest addressed element is in the top left corner and each consecutive element follows a row-major addressing order.

Table 4: Vector register contents after first shuffle.

A′_v512	A[1][1]	0	0	0
	0	A[2][2]	0	0
	0	0	A[3][3]	0
	0	0	0	A[4][4]

Table 5: Vector register contents after second shuffle.

A′_v512	A[1][1]	0	0	A[4][1]
	A[1][2]	A[2][2]	0	0
	0	A[2][3]	A[3][3]	0
	0	0	A[3][4]	A[4][4]

Table 6: Vector register contents after third shuffle.

A′_v512	A[1][1]	0	A[3][1]	A[4][1]
	A[1][2]	A[2][2]	0	A[4][2]
	A[1][3]	A[2][3]	A[3][3]	0
	0	A[2][4]	A[3][4]	A[4][4]

Table 7: Vector register contents after the final shuffle.

A′_v512	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][2]	A[2][2]	A[3][2]	A[4][2]
	A[1][3]	A[2][3]	A[3][3]	A[4][3]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]

Table 8: Vector register contents after load with broadcast.

t1_v512	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][1]	A[2][1]	A[3][1]	A[4][1]
	A[1][1]	A[2][1]	A[3][1]	A[4][1]

Table 9: Vector register contents illustrating swizzle.

t2_v512	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[1][1]	B[1][2]	B[1][3]	B[1][4]
	B[1][1]	B[1][2]	B[1][3]	B[1][4]

First, array data is loaded into a vector unit. With a wider SIMD vector unit, the compiler is able to load the entire A and B matrix each into a single vector unit.

(a) Matrices A and B are loaded into two SIMD registers:

```
//Load A matrix from memory into vector
register,
A_v512 = ⟨A[1][1], A[1][2], ... ...,
  A[4][3], A[4][4]⟩.
```

For more details see Table 1.

```
//Load B matrix from memory into vector
register,
B_v512 = ⟨B[1][1], B[1][2], ... ...,
B[4][3], B[4][4]⟩.
```

For more details see Table 2.

Table 10: C_v512 vector unit contains elementwise product of t1_v512 and t2_v512.

C_v512	t1_v512 * t2_v512

Table 11: t1_v512 vector register contents illustrating final load with broadcast.

t1_v512	A[1][4]	A[2][4]	A[3][4]	A[4][4]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]
	A[1][4]	A[2][4]	A[3][4]	A[4][4]

Table 12: t2_v512 vector register contents illustrating final swizzle.

t2_v512	B[4][1]	B[4][2]	B[4][3]	B[4][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]
	B[4][1]	B[4][2]	B[4][3]	B[4][4]

Table 13: Final C_v512 vector unit contains sum of existing values of C_v512 and elementwise products t2_v512 and t_v512.

C_v512	t2_v512 * t1_v512 + C_v512

Next, the compiler optimizes the multiplication operation between matrix A and matrix B, through a series of data layout transformations and vector multiplication and addition operations. The compiler identifies a matrix multiplication in this loop and permutes the elements in matrix A and matrix B setting up simple vector multiplications and additions.

(b) We can simplify the multiplication needed through a transposition of the elements of A, followed by a multiply and add of each row B and with each row of transposed A. We start by transposing the elements of A.

```
//First, create a vector unit of zeros.
A′_v512 = _mm512_setzero()
```

For more details see Table 3.

For the transpose operation, we use a set of new Intel MIC _mm512_mask_shuf128 × 32( ) intrinsic calls. Similarly in classic architecture, this shuffle intrinsic is bound by four 128-bit "lanes" in each vector register. Thus, this intrinsic contains arguments for permutation patterns for each of the four 128-bit lanes, as well as a permutation pattern for each of the four 32 bit boundaries within each of those lanes. The arguments are as follows:

_m512 res = _mm512_mask_shuf128 × 32(_m512 v1, (I16) vmask, _m512 v2, (SI32)perm128, (SI32)perm32),

(i) res: result vector unit,

(ii) v1: blend-to-vector unit; the values in this vector unit will be blended with the shuffled elements of the v2, according to the write mask,

(iii) vmask: write mask; the write mask is a bit vector specifying which elements to overwrite in v1 with the shuffle elements of v2,

(iv) v2: incoming data vector unit; this vector unit holds the elements which are to be shuffled,

(v) perm128: 128-bit lane permutation; this value specifies the permutation order of the vector unit's 128-bit lanes,

(vi) perm32: elementwise permutation; this value specifies the permutation order of the each of the four 32 bit boundaries within each 128-bit lane,

```
//Begin transpose operation by
shufflingelements into
//desired order. Shuffle used to insert
matrix diagonal
//into transpose result vector unit,
A′_v512 = _mm512_mask_shuf128 ×32(A′_
v512, 0 × 8421,A_v512,
_MM_PERM_DCBA, _MM_PERM_DCBA).
```

For more details see Table 4.

```
//Shuffle the next four elements and
blend-in with the
//elements written from previous
shuffle,
A′_v512 = _mm512_mask_shuf128 × 32(A′_
v512, 0 × 4218,A_v512, _MM_PERM_CBAD,
_MM_PERM_ADCB).
```

For more details see Table 5.

```
//Shuffle the next four elements and
blend-in with the
//elements written from previous
shuffle,
A′_v512 = _mm512_mask_shuf128 × 32(A′_
v512, 0 × 2184, A_v512, _MM_PERM_BADC,
_MM_PERM_BADC)
```

For more details see Table 6.

```
//Shuffle the final four elements and
blend-in with the
//elements written from previous shuffle
to obtain the
//complete transpose,
A′_v512 = _mm512_mask_shuf128 × 32(A′_
v512, 0 × 1842, A_v512, _MM_PERM_ADCB,
_MM_PERM_CBAD).
```

For more details see Table 7.

After the elements of matrix A have been permuted through transposition, each element of A and B is now in the correct position within each vector unit for a vector product, resulting in the same behavior as the dot product of rows and columns.

(c) Next, we perform the multiplication of each row of the transposed A with each row of B, maintaining a sum of the products from row to row:

```
//Load the first row of A′_v512 and
broadcast that row to
//each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_v512[0 : 4],
_MM_FULLUPC_NONE, _MM_BROADCAST_4 × 16,
0).
```

For more details see Table 8.

Another useful intrinsic used in this optimization is the Intel MIC _mm512_swizzle_ps( ) intrinsic. This intrinsic is similar to that of the shuffle above except it only permutes each 128-bit lane and not each of the 32 boundaries within those lanes. The arguments are as follows:

_m512 res = _mm512_swizzle_ps(_mm512 v1, SI32 perm)

(i) res: result vector unit,

(ii) v1: incoming data vector unit to be permuted,

(iii) perm: permutation pattern for each 128-bit lane,

```
//Load the first row of B_v512 and
broadcast that row to
//each of the remaining three rows
t2_v512 = _mm512_swizzle_ps (B_v512,
_MM_SWIZ_REG_AAAA).
```

For more details see Table 9.

```
//Multiply each element of t1_v512
with each element of
//t2_v512 and store result in C_v512
C_v512 = _mm512_mul_ps (t1_v512, t2_v512).
```

For more details see Table 10.

```
//Load the second row of A′_v512 and
broadcast that row
//to each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_
v512[4 : 8], _MM_FULLUPC_NONE, _MM_
BROADCAST_4 × 16, 0)
//Load the second row of B_v512 and
broadcast that row to
//each of the remaining three rows
t2_v512 = _mm512_swizzle_ps (B_v512,
_MM_SWIZ_REG_BBBB).
```

Each subsequent multiplication must be accumulated for each row. These multiplications and additions are the corresponding dot product of rows and columns found in matrix multiplication, but because of the earlier transpose, no further permuting is required:

```
//Add the existing values of C_v512
with the product of
//t1_v512 and t2_v512 and store
result in C_v512
C_v512 = _mm512_madd213_ps (t2_v512,
t1_v512, C_v512)

//Load the third row of A′_v512
and broadcast that row to
//each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_
v512[8 : 12], _MM_FULLUPC_NONE,
_MM_BROADCAST_ 4 × 16, 0)

//Load the third row of B_v512 and
broadcast that row to
//each of the remaining three rows
t2_v512 = _m512_swizzle_ps (B_v512,
_MM_SWIZ_REG_CCCC)

//Add the existing values of C_v512
with the product of
//t1_v512 and t2_v512 and store result
in C_v512
C_v512 = _mm512_madd213_ps (t2_v512,
t1_v512,C_v512)

//Load the fourth row of A′_v512
and broadcast that row
//to each of the remaining three rows
t1_v512 = _mm512_extload_ps(A′_
v512[12:16], _MM_FULLUPC_NONE,
_MM_BROADCAST_4 × 16, 0).
```

For more details see Table 11.

```
//Load the fourth row of B_v512
and broadcast that row to
//each of the remaining three rows
t2_v512 = _mm512_swizzle_ps (B_v512,
_MM_SWIZ_REG_DDDD).
```

For more details see Table 12.

```
//Add the existing values of C_v512
with the product of
//t1_v512 and t2_v512 and store result
in C_v512
C_v512 = _mm512_madd213_ps (t2_v512,
t1_v512, C_v512).
```

For more details see Table 13.

After the simplified matrix multiplication, the loop further requires that results be stored in the C matrix. With all elements correctly computed and residing in vector unit only one store operation is generated.

(d) Finally, the result vector unit of values is stored to the C array:

```
//The elements of vector register
C_v512 are then stored
```

```
//to memory at &C[1][1]
   ⟨C[1][1], &C[1][2], ... C[4][3],
   &C[4][4]⟩ = C_v512.
```

The 512-bit long SIMD vector unit of the Intel MIC architecture supports consumption of both matrix dimensions for 2D vectorization, fitting an entire small matrix (4×4 float type) into one 512-bit SIMD vector register. This enables more efficient flexible vectorization and optimizations for small matrix operations. For example, the scalar version of single precision 4×4 matrix multiply computation naively executes 128 memory loads, 64 multiplies, 64 additions, and 16 memory stores. The small matrix 2D vectorization reduces instructions to 2 vector loads from memory, 4 multiplications, 4 shuffles, 4 swizzles, 3 additions, and 1 vector store to memory for a reduction of approximately 15x in number of instructions.

6. Performance Evaluation

This section presents the performance results measured on an Intel Xeon Phi coprocessor system using a set of workloads and microbenchmarks.

6.1. Workloads. We have selected a set of workloads to demonstrate the performance benefits and importance of SIMD vectorization on the Intel MIC architecture. These workloads exhibit a wide range of application behavior that can be found in areas such as high performance computing, financial services, databases, image processing, searching, and other domains. These workloads include the following.

6.1.1. NBody. NBody computations are used in many scientific applications such as astrophysics [9] and statistical learning algorithms [10]. The main computation involves two loops that iterate over the bodies and computes a pairwise interaction between them.

6.1.2. 2D 5×5 Convolution. Convolution is a common image filtering computation used to apply effects such as blur and sharpen. For a given 2D image and a 5×5 spatial filter containing weights, this convolution computes the weighted sum for the neighborhood of the 5×5 set of pixels.

6.1.3. Back Projection. Back projection is commonly used for performing cone-beam image reconstruction of CT projection values [11]. The input consists of a set of 2D images that are "back-projected" onto a 3D volume in order to construct a 3D grid of density values.

6.1.4. Radar (1D Convolution). The 1D convolution is widely used in applications such as radar tracking, graphics, and image processing.

6.1.5. Tree Search. In memory tree structured index search is a commonly used operation in database applications. This benchmark consists of multiple parallel searches over a tree with different queries, where the path through the tree is determined based on the comparison of results of the query and node value at each tree level.

Table 14: Target system configuration.

System parameters	Intel Xeon Phi processor
Chips	1
Cores/threads	61 and 244
Frequency	1 GHz
Data caches	32 KB L1, 512 KB L2 per core
Power budget	300 W
Memory capacity	7936 MB
Memory technology	GDDR5
Memory speed	2.75 (GHz) (5.5 GT/s)
Memory channels	16
Memory data width	32 bits
Peak memory Bandwidth	352 GB/s
SIMD vector length	512 bits

6.2. System Configuration. The detailed information on the configuration of the Intel Xeon Phi Coprocessor used for the performance study and for evaluating the effectiveness of SIMD vectorization techniques is provided in Table 14.

6.3. Performance Results. All benchmarks were compiled as native executable using the Intel 13.0 product compilers and run on the Intel Xeon Phi coprocessor system specified in Table 14. To demonstrate the performance gains obtained through the SIMD vectorization, two versions of the binaries were generated for each workload. The baseline version was compiled with OpenMP parallelization only (-mmic -openmp -novec); the vectorized version is compiled with vectorization (default ON) and OpenMP parallelization (-mmic -openmp).

The performance scaling is derived from the OpenMP-only execution and OpenMP with 512-bit SIMD vector execution on the Intel Xeon Phi coprocessor system that we described at beginning of this section. That is, when the workload contains 32-bit single precision computations, 16-way vectorization may be achieved. When the workload contains 64-bit double-precision computations, 8-way vectorization is achieved.

Figure 2 shows the normalized SIMD performance speedup of five workloads. The generated SIMD code of these workloads achieved SIMD speedup ranging from 2.25x to 12.45x. Besides those classical HPC applications with regular array accesses and computations, the workload with a large amount of branching codes, such as tree search used in database applications, achieves 2.25x speedup as well with SIMD vectorization based on the masking support in the Intel MIC architecture.

6.3.1. Impact of Less-than-Full-Vector Loop Vectorization. To examine the impact of the less-than-full-vector loop vectorization, a simple microbenchmark was written with three

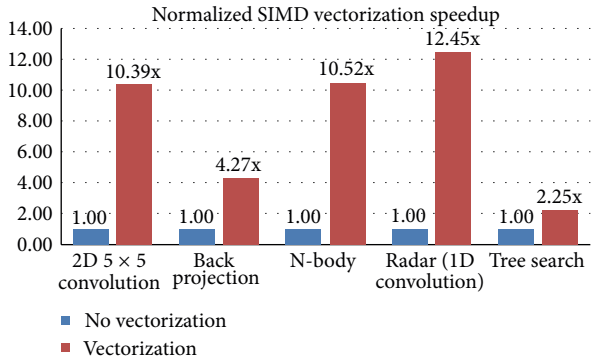

Figure 2: Performance results of workloads.

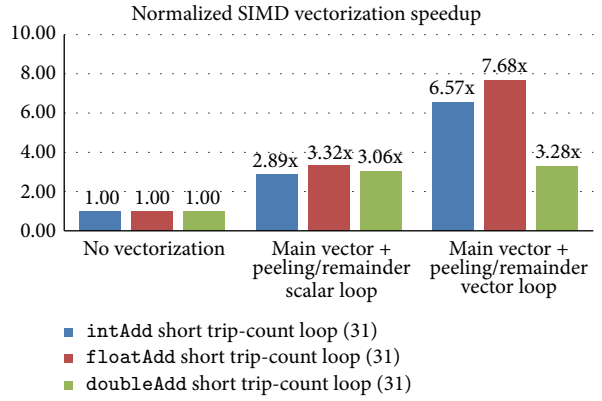

Figure 3: Performance gain with "less-than-full-vector" loop vectorization.

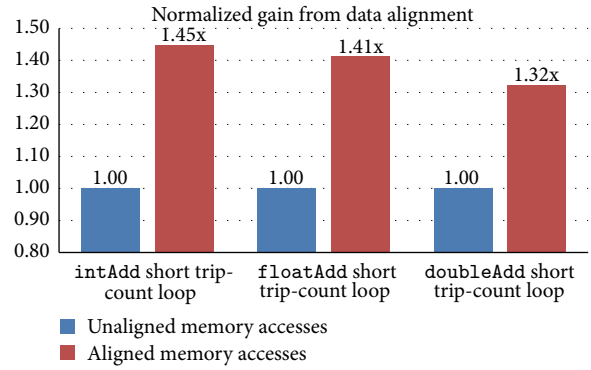

Figure 4: Performance gain with data alignment.

small kernel functions: `intAdd`, `floatAdd`, and `doubleAdd`. Each of them has a short trip-count loop that takes 3 arrays, a, b, c of size 31, and does an elementwise addition with respect to `int`, `float`, and `double` data types. The vector length is 16 iterations for loops in the `intAdd` and `floatAdd` kernels and 8 iterations for the loop in the `doubleAdd` kernel function. This experimental setup ensures the `intAdd` and `floatAdd` loops contain a 15-iteration remainder loops, and the `doubleAdd` loop contains a 7-iteration remainder loop which can be vectorized with the "*less-than-full-vector*" loop SIMD vectorization technique using masking support described in the Section 2.

Figure 3 shows performance gains from vectorization without "*less-than-full-vector*" loop vectorization and with "*less-than-full-vector*" loop vectorization for three short trip-count loops in the `intAdd`, `floatAdd`, and `doubleAdd` kernel functions. The generated SIMD code of these loops achieves a speedup ranging from 2.89x to 3.32x without "*less-than-full-vector*" loop vectorization. With "*less-than-full-vector*" loop vectorization, the performance speedup is improved significantly and ranges from 3.28x to 7.68x. Note that, in this measurement, all data are 64-byte aligned, there are no peeling loops generated, and the aligned memory load/store instructions such as `vmovaps` and `vmovapd` [1] are generated to achieve optimal performance. The next subsection shows the data alignment impact on the Intel MIC architecture.

6.3.2. Impact of Data Alignment. These kernel loops used in Section 6.3.1 are reused for this measurement. In this study, the difference is that we do not provide alignment information of the arrays `a`, `b`, and `c`. Without alignment information, given these loops are short trip-count loops with constant trip count, the compiler generates SIMD instructions:

(i) `vloadunpackld` and `vloadunpackhd` to load data from unaligned memory locations and `vpackstoreld` and `vpackstorehd` [1] to store data to unaligned memory locations for the vectorized main loop,

(ii) `vgatherdps` and `vscatterdps` instructions [1] to load and store for the vectorized remainder loop with write mask.

As shown in Figure 4, with data alignment information, the performance of SIMD execution is 1.45x, 1.41x, and 1.32x better than unaligned cases with respect to `int`, `float`, and `double` types of three kernel functions. The alignment optimization described in Section 3 is critical to achieving optimal performance on Intel MIC architecture.

6.3.3. Impact of Small Matrix 2D Vectorization. Small matrix operations such as addition and multiplication have served as important parts of many HPC applications. A number of classic compiler optimizations such as loop complete unrolling, partial redundancy elimination (PRE), scalar replacement, and partial summation have been developed to achieve optimal vector execution performance. The conventional inner or outer loop vectorization for 3-level loop nests of 4 × 4 matrix operations is not performing well on Intel Xeon Phi coprocessor due to

(i) less effective use of 512-bit long SIMD unit, for example, for 32-bit float data type, when either inner loop or outer loop is vectorized. In this case 4-way vectorization is used instead of 16-way vectorization,

(ii) side-effects on classic optimizations, for example, the partial redundancy elimination, partial summation, and operator strength reduction, when the loop is vectorized.

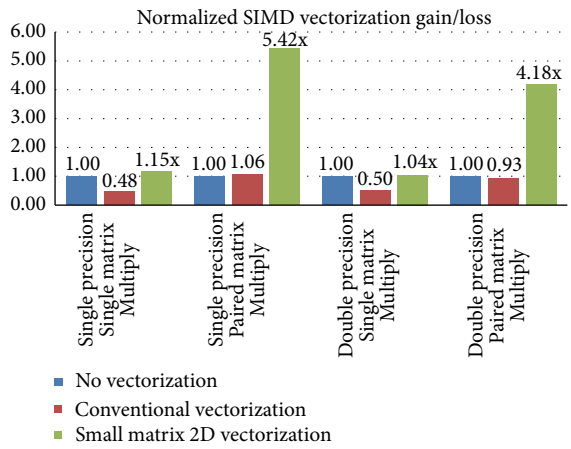

FIGURE 5: Performance gain/loss with SIMD vectorization.

As shown in Figure 5, the convention loop vectorization on small matrix (4 × 4) operations does cause performance degradation. For both cases of single precision and double precision matrix (4 × 4) multiplications, the performance degradation is ~50% when comparing against cases without vectorization, which are used as the baseline performance. In the case of the paired matrix multiplication, there are two matrix (4 × 4) multiplications done in a single loop nest, and matrix B is transposed for computing `sumy` (for more details see Algorithm 4).

The classical loop optimizations are not as effective as for the single matrix multiplication case due to the transpose operation of `matrix B` and paired matrix multiplications in the loop. Thus, the performance achieved with classical loop optimization is on-par with applying conventional loop vectorization, and no notable performance difference is observed as shown in Figure 5. Promisingly, applying the small matrix 2D vectorization we proposed in Section 4, we achieved a performance speedup 1.15x/1.04x for single matrix (4 × 4 float/double type) multiplication and a speedup 5.42x/4.18x for paired matrix (4 × 4 float/double type) transpose and multiplication, which demonstrates the effectiveness of small matrix 2D vectorization using long SIMD vector unit supported by Intel Xeon Phi coprocessor.

7. Seamless Integration with Threading

Effectively exploiting the power of a coprocessor like Xeon Phi requires that both thread- and vector-level parallelism are exploited. While the parallelization topic is beyond the scope of this paper, we would still like to highlight that the SIMD vector extensions can be seamlessly integrated with threading models such as OpenMP* 4.0 supported by the Intel compilers. Given the Mandelbrot example Mandelbrot computes a graphical image representing a subset of the Mandelbrot set (a well-known 2D fractal shape) out of a range of complex numbers. It outputs the number of points inside and outside the set.

In the *mandelbrot* workload, the function "*mandel*" in the *mandelbrot* program is a hot function and a candidate for SIMD vectorization, so we can annotate it with `#pragma omp declare SIMD`. At the caller site, the hot loop is a double nested *for* loop, the outer *for* loop is asserted with "omp parallel for" for threading, and the inner loop is asserted with "omp SIMD" for vectorization as shown in Algorithm 5. Note that the "guided" scheduling type is used for achieving a good load balance, as each call to "mandel" function does varying amount of work in terms of execution time due to "break" exit of the loop.

Figure 6 shows that the SIMD vectorization alone delivers a ~16x speedup, built with option –mmic –openmp –std=c99–O3 over the serial execution. The OpenMP parallelization delivers a 62.09x speedup with 61 threads using 61 cores with Hyperthreading OFF, a speedup 131.54x with 244 threads (61 cores with Hyperthreading ON, 4 HT threads per core) over the serial execution. The OMP PARALLEL FOR and SIMD combined execution delivers an OMP PAR + SIMD speedup 2067.9x with 244 threads, running on an Intel Xeon Phi system, which has 61-core on the chip with Hyperthreading ON. The performance scaling from 1 thread to 61 threads is close to linear. In addition, the Hyperthreading support delivers a ~2x performance gain by comparing the 244-thread speedup with the 61-thread speedup, which is better than the well-known 20%–30% expectation on the performance gain from Hyperthreading technology due to the nature of less computing resource contention in the workload, and 4 busy HT threads did hide latency well. For the system information details see Section 6.2.

8. Related Work

The compiler vectorization technology [12] had been one of the key loop transformations for traditional vector machine decades ago. However, the recent proliferation of modern SIMD architecture [1, 4] poses new constraints such as data alignment, masking for control flow, nonunit stride access to memory, and the fixed-length nature of SIMD vectors that shall demand more advanced vectorization technologies and vectorization friendly programming language extensions [7].

In the past three plus decades, the rich body of SIMD vectorization capabilities has been incorporated in a number of industry and research compilers [5, 6, 12–16]. These include works based on ICC (the Intel compiler) [5, 6], XLC (the IBM compiler) [13, 16], VAST [17], GCC [18, 19], and the SUIF compiler [20]. However, there are many unknown program factors such as loop trip count, memory access stride and patterns, alignment, and control flow complexity at compile-time that pose challenges to the modern optimizing compiler's ability to apply advanced and practical vectorization techniques and fulfill the semantic gap between application programs and the modern processors such as Intel Xeon Phi coprocessor for harnessing its computational power.

Compared to the conventional loop vectorization [5, 12, 20], the "less-than-full-vector" vectorization technique brings extra performance benefits for those vectorizable short

```
do j = 1, 4
  do k = 1, 4
    sumx = 0.0
    sumy = 0.0
    do i = 1, 4
      sumx = sumx + matrixA(i,k) * matrixB(i,j)
      sumy = sumy + matrixA(i,k) * matrixB(j,i)
    enddo
    matrixC(k,j) = sumx
    matirxD(j,k) = sumy
  enddo
enddo
```

Algorithm 4

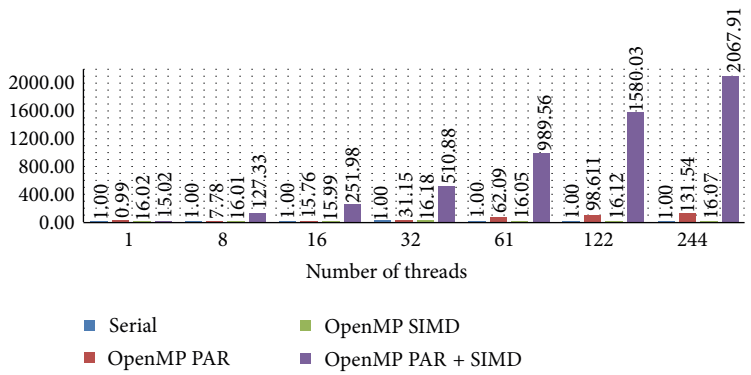

Figure 6: OpenMP* parallel for and SIMD speedup of mandelbrot workload.

trip-count loops, especially when the processor provides the long SIMD unit masking capability like the Intel Xeon Phi coprocessor. Our alignment optimizations are built on top of existing dynamic alignment optimizations as presented in [5, 6]. However, the alignment strategy described in this paper is designed to satisfy the requirement of Intel MIC architecture with optimal SIMD instruction selection and mask utilization for safe and optimal performance. Beyond traditional single-level loop vectorization [5, 12, 16, 18, 19, 21], the small matrix operation 2D vectorization increases vector-parallelism and improves the utilization efficiency of the long SIMD vector unit, swizzle, shuffle, broadcast, and mask support in Intel MIC architecture significantly.

In addition, programming language extensions such as OpenMP* SIMD extensions [22, 23] and Cilk Plus [3, 7] function vectorization and loop vectorization through the compiler has been paving the way to enable more effective vector-level parallelism [7, 22] in both C/C++ and Fortran programming languages. To support these SIMD vector programming models on the Intel Xeon Phi coprocessor effectively, the practical and effective vectorization techniques described in this paper are essential for achieving optimal performance and ensuring SIMD code execution safety on an Intel Xeon Phi coprocessor system.

9. Conclusions

Driven by the increasing prevalence of SIMD architecture in the Intel Xeon Phi coprocessor, we proposed and implemented new vectorization techniques to explore the effective use of its long SIMD units. This paper presented several practical SIMD vectorization techniques such as less-than-full-vector loop vectorization, Intel MIC specific data alignment optimizations, and small matrix operations 2D vectorization for the Intel Xeon Phi coprocessor. A set of workloads from several domains was employed to evaluate the benefits of our SIMD vectorization techniques. The results show that we achieved up to 12.5x performance gain on Intel Xeon Phi coprocessor. Mandelbrot workload demonstrated the seamless integration of SIMD vector extensions with threading and showed a 2067.91x performance speedup with the combined use of OpenMP "parallel for" and "SIMD" constructs using Intel C/C++ compilers on an Intel Xeon Phi coprocessor system.

Intel C/C++ and Fortran compilers are highly enhanced for programmers to harness the computational power of Intel Xeon Phi coprocessors for accelerating highly parallel applications found in chemistry, visual computing, computational physics, biology, financial services, pixel, multimedia,

```
#pragma omp declare SIMD uniform(max_iter) SIMDlen(32)
uint32_t mandel(fcomplex c, uint32_t max_iter)
{
    // Computes number of iterations(count variable)
    // that it takes for parameter c to be known to
    // be outside mandelbrot set
    uint32_t count = 1; fcomplex z = c;
    for (int32_t i = 0; i < max_iter; i += 1) {
      z = z * z + c;
      int t = (cabsf(z) < 2.0f);
      count += t;
      if (t == 0) { break;}
    }
    return count;
}
Caller site code:
int main() {
   ……
    #pragma omp parallel for schedule(guided)
    for (int32_t y = 0; y < ImageHeight; ++y) {
      float c_im = max_imag - y * imag_factor;
      #pragma omp SIMD safelen(32)
      for (int32_t x = 0; x < ImageWidth; ++x) {
        fcomplex in_val;
        in_val = (min_real + x*real_factor) + (c_im*1.0iF);
       count[y][x] = mandel(in_val, max_iter);
      }
    }
    ……
}
```

Algorithm 5: An example of OpenMP* parallel for and SIMD combined usage.

graphics, and HPC applications by effectively exploiting the use of the Intel MIC architecture SIMD vector unit beyond traditional loop SIMD vectorization.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] Intel Corporation, "Intel Xeon Phi Coprocessor System Software Developers Guide," 2012, http://software.intel.com/en-us/mic-developer.

[2] N. Satish, C. Kim, J. Chhugani et al., "Can traditional programming bridge the Ninja performance gap for parallel computing applications?" in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*, pp. 440–451, June 2012.

[3] J. Reinders, "An Overview of Programming for Intel Xeon processor and Intel Xeon Phi Coprocessor," 2012.

[4] Intel Corporation, *Intel Advanced Vector Extensions Programming Reference*, Document Number 319433-011, Intel Corporation, 2011.

[5] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian, "Automatic intra-register vectorization for the intel architecture," *International Journal of Parallel Programming*, vol. 30, no. 2, pp. 65–98, 2002.

[6] A. J. C. Bik, D. L. Kreitzer, and X. Tian, "A case study on compiler optimizations for the Intel Core^{TM} 2 duo processor," *International Journal of Parallel Programming*, vol. 36, no. 6, pp. 571–591, 2008.

[7] X. Tian, H. Saito, M. Girkar et al., "Compiling C/C++ SIMD extensions for function and loop vectorizaion on multicore-SIMD processors," in *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops (IPDPSW '12)*, pp. 2349–2358, May 2012.

[8] H. J. Lu, M. Garkar, M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell, "System V Application Binary Interface K1OM Architecture Processor Supplement," Version 1.0, 2012, http://software.intel.com/en-us/forums/topic/278102.

[9] S. J. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithm*, Cambridge Monographs on Mathematical Physics, Cambridge University Press, Cambridge, UK, 2003.

[10] A. G. Gray and A. W. Moore, "'N-body' problems in statistical learning," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 521–527, 2000.

[11] M. Kachelrieb, M. Knaup, and O. Bockenbach, "Hyperfast perspective cone-beam backprojection," in *Proceedings of the IEEE Nuclear Science Symposium Conference Record*, pp. 1679–1683, November 2006.

[12] R. Allen and K. Kennedy, "Automatic translation of FORTRAN programs to vector form," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491–542, 1987.

[13] A. E. Eichenberger, K. O'Brien, P. Wu et al., "Optimizing compiler for the CELL processor," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*, pp. 161–172, IEEE, St. Louis, Mo, USA, September 2005.

[14] R. Karrenberg and S. Hack, "Whole-function vectorization," in *Proceedings of the 9th International Annual IEEE/ACM Symposium on Code Generation and Optimization*, pp. 141–150, Charmonix, France, April 2011.

[15] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, pp. 145–156, June 2000.

[16] P. Wu, A. E. Eichenberger, and A. Wang, "Efficient SIMD code generation for runtime alignment and length conversion," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*, pp. 153–164, March 2005.

[17] Crescent Bay Software, *VAST-F/AltiVec: Automatic Fortran Vectorizer for PowerPC Vector Unit*, 2004.

[18] D. Nuzman and A. Zaks, "Outer-loop vectorization—revisited for short SIMD architectures," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 2–11, Toronto, ON, Canada, October 2008.

[19] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *Proceedings of the 4th International Symposium on Code Generation and Optimization (CGO '06)*, pp. 281–294, New York, NY, USA, March 2006.

[20] G. Cheong and M. S. Lam, "An optimizer for multimedia instruction sets," in *Proceedings of the 2nd SUIF Compiler Workshop*, August 1997.

[21] J. Shin, M. Hall, and J. Chame, "Superword-level parallelism in the presence of control flow," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '05)*, pp. 165–175, IEEE Computer Society, March 2005.

[22] M. Klemm, A. Duran, X. Tian, H. Saito, D. Caballero, and X. Martorell, "Extending OpenMP* with vector constructs for modern multicore SIMD architectures," in *OpenMP in a Heterogeneous World: 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11–13, 2012. Proceedings*, Lecture Notes in Computer Science, pp. 59–72, Springer, Berlin, Germany, 2012.

[23] OpenMP Architecture Review Board, "OpenMP Application Program Interface," Version 4.0 (Release Candidate RC1), 2012.