Vector Folding: improving stencil performance via multi-dimensional SIMD-vector representation

Charles Yount, Intel Corporation, chuck.yount@intel.com
2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC)
©2015 IEEE, DOI 10.1109/HPCC-CSS-ICESS.2015.27

Abstract—Stencil computation is an important class of algorithms used in a large variety of scientific-simulation applications. Modern CPUs are employing increasingly longer SIMD vector registers and operations to improve computational throughput. However, the traditional use of vectors to contain sequential data elements along one dimension is not always the most efficient representation, especially in the multicore and hyper-threaded context where caches are shared among many simultaneous compute streams. This paper presents a general technique for representing data in vectors for 2D and 3D stencils. This method reduces the number of memory accesses required by storing a small multi-dimensional block of data in each vector compared to the single dimension in the traditional approach. Experiments on an Intel Xeon Phi Coprocessor show performance speedups over traditional vectors ranging from 1.2x to 2.7x, depending on the problem size and stencil type. This technique is independent of and complementary to a variety of existing stencil-computation tuning algorithms such as cache blocking, loop tiling, and wavefront parallelization.

I. INTRODUCTION

Stencil computation is a class of algorithms used in a large variety of scientific simulation applications. Conceptually, the kernel of a typical 3D iterative Jacobian stencil computation can be shown by the following pseudo-code that iterates over the points in a 3D grid:

```plaintext
for t = 1 to T do
  for i = 1 to Dx do
    for j = 1 to Dy do
      for k = 1 to Dz do
        u(t + 1, i, j, k) ← S(t, i, j, k)
      end for
    end for
  end for
end for
```

where \( T \) is the number of time-steps; \( D_x, D_y, \) and \( D_z \) are the problem-size dimensions; and \( S(t, i, j, k) \) is the stencil function. For the stencil functions under consideration here, there are no dependencies between any values in the 3D grid within a time-step, implying that the \( D_x \times D_y \times D_z \) values for the \( t + 1 \) time-step can be calculated in any order. This allows straightforward application of threading and vectorization techniques within a time-step. More advanced techniques that apply across time-steps can enable even more performance opportunities.

The stencil function itself typically inputs the center grid element \( u(t, i, j, k) \) and multiple neighboring values. The number of values input is referred to as the number of points in the stencil. Samples of different 3D stencil shapes and sizes are illustrated in Figure 1. For example, the 25-point 3-axis stencil (b) can implement functions such as this 8th-order\(^1\) isometric 3D finite-difference equation:

\[
S(t, i, j, k) \equiv c_0 u(t, i, j, k) + \sum_{r=1}^{4} c_r [u(t, i - r, j, k) + u(t, i + r, j, k) + u(t, i, j - r, k) + u(t, i, j + r, k) + u(t, i, j, k - r) + u(t, i, j, k + r)]
\]

The number and variety of possible stencils are practically unlimited. Stencils may be 1D, 2D, or 3D and may or may not be symmetric about the origin. Some stencils input temporal neighbors, e.g., \( u(t - 1, i, j, k) \), or values from other grids. Jacobian-type stencils output a single value for the next time-step at \( u(t + 1, i, j, k) \). Other types of stencils output multiple values. This paper discusses a specific vectorization technique and measures its impact on Jacobian-type 3D stencils. Nonetheless, the technique is applicable to many other types of stencils.

\(^1\)The order of this stencil is twice the radius, indicated by the summation of \( r \) from 1 to 4.
The remainder of the paper is organized as follows: Section II covers related work. Section III defines the concept of vector folding and provides some architecture-independent cost metrics. Section IV discusses the implementation of a harness to compare performance to traditional vectors on a specific architecture. Section V shows the results of the performance experiments. Section VI concludes with a summary and future work.

II. RELATED WORK

CPUs that support Single-Instruction, Multiple-Data (SIMD) vectors and instructions that operate on those vectors have been available for over four decades [1]. SIMD was initially only available in specialty supercomputers, but it soon began to appear in consumer CPUs to accelerate media applications such as photo-editing software. SIMD vector lengths in volume CPUs have also increased over recent years. For example, Intel Corporation CPUs supported vector lengths of 64 bits in MMX, 128 bits in SSE, 256 bits in AVX, and 512 bits in the Intel Xeon Phi and AVX-512 instruction sets [2].

Reduction of memory accesses is especially advantageous in multicore and hyper-threaded architectures due to the additional pressure on shared caches, memory load and store units, etc. Much work has been done on removing memory bandwidth as the primary limiting factor in stencil computations [3]–[9]. Our work focuses on reducing memory accesses at the vector level and is thus complementary to these techniques. Additional stencil work has focused on vectorization [10]–[12]. Finally, there have been advances in auto-tuning stencil algorithms [5], [13]–[15] that we leverage. To the best of our knowledge, the particular technique presented herein of packing elements across multiple dimensions into each hardware vector and an associated comparison of performance to traditional techniques have not previously been published.

III. SIMD VECTOR FOLDING

Stencil computations can readily take advantage of SIMD vector code, but the most straightforward usage does not generally result in the minimum number of memory accesses required to compute a time-step update. This section reviews the traditional in-line method of applying SIMD to stencil codes and describes a multi-dimensional method that reduces the number of memory reads for many stencil types.

A. In-line vectors

The most straightforward use of SIMD for 3D stencils is to apply vectorization in-line, i.e., in the same direction of the inner-most loop. An auto-vectorizing compiler is often able to do this implicitly (without code modification) when consecutive inner-most loop indices correspond to accesses of consecutive locations in memory. Such vectorization can also be specified explicitly by the coder via special variable and/or loop directives, “intrinsic” functions, etc. Whether the code is generated implicitly or explicitly, such in-line vectorization can be described as follows:

\[
\begin{align*}
    k &\equiv [k, \ldots, k + L_v - 1] \\
    u(t + 1, i, j, k) &\leftarrow S(t, i, j, k)
\end{align*}
\]

where \(L_v\) is the vector length in vector elements. For example, \(L_v = 8\) for double-precision floats in a 512-bit vector. The term \(k\) indicates that operations are performed to calculate the \(L_v\) elements between \(k\) and \(k + L_v - 1\), inclusive, using appropriate SIMD vectors and instructions implementing the vectorized function \(S(t, i, j, k)\). In the above pseudo-code, we apply vectorization in the \(z\)-dimension, assuming that sequential \(k\) indices correspond to consecutive memory locations.

B. Multi-dimensional vectors

We now generalize this concept to multiple dimensions:

\[
\begin{align*}
    \text{for } t = 1 \text{ to } T &\text{ do} \\
    \text{for } i = 1 \text{ to } D_x \text{ by } L_{vx} &\text{ do} \\
    \text{for } j = 1 \text{ to } D_y \text{ by } L_{vy} &\text{ do} \\
    \text{for } k = 1 \text{ to } D_z \text{ by } L_{vz} &\text{ do} \\
        i &\equiv [i, \ldots, i + L_{vx} - 1] \\
        j &\equiv [j, \ldots, j + L_{vy} - 1] \\
        k &\equiv [k, \ldots, k + L_{vz} - 1] \\
        u(t + 1, i, j, k) &\leftarrow S(t, i, j, k)
    \end{align*}
\]

Here, vectorization is applied in more than one dimension simultaneously: \(L_{vx}\) elements in the \(x\)-dimension, \(L_{vy}\) in \(y\), and \(L_{vz}\) in \(z\). From an implementation standpoint, each vector register used in the stencil calculation contains values from multiple dimensions instead of just one in the traditional approach. At this point, we introduce some terms used in the remainder of the paper:

- **Vector block** Any sub-block of size \(L_{vx}\times L_{vy}\times L_{vz}\) within the overall 3D grid. In order to apply \(L_v\)-wide SIMD calculations efficiently, \(L_{vx}\times L_{vy}\times L_{vz} = L_v\).
- **Vector-block dimensions** The values of \(L_{vx}\), \(L_{vy}\), and \(L_{vz}\).
- **Vector-folding** The overall process of decomposing a grid and calculating stencils using vector blocks. Also called multi-dimensional vectorization.

**Aligned vector block** A vector block starting at \(u(i_n, j_n, k_n)\) such that \(i_n\) is a multiple of \(L_{vx}\), \(j_n\) is a multiple of \(L_{vy}\), and \(k_n\) is a multiple of \(L_{vz}\).

**Unaligned vector block** A vector block such that at least one of its starting indices is not a multiple of the corresponding vector-block dimension.

**Fold** A specific set of values for the vector-block dimensions. A fold is named with the shorthand notation \(L_{vx}\times L_{vy}\times L_{vz}\). For example, the fold where \(L_{vx} = 1\), \(L_{vy} = 2\), and \(L_{vz} = 4\) is referred to as the 1x2x4 fold.

**0D fold** A degenerate fold for \(L_v = 1\). This is for scalar (non-vector) operation.

**1D fold** A fold in which only one of its dimensions is greater than one (1). If that dimension is also in the direction of the inner-most loop, this corresponds to traditional in-line vectorization.
Higher-dimension folds are possible and may be useful for other applications. They are not discussed in this paper.

2D fold A fold in which exactly two of its dimensions are greater than one.

3D fold A fold in which all of its dimensions are greater than one.

Figure 2 illustrates three folds for \( L_v = 8 \) and their application to the stencil in Figure 1b. For each fold, two subdiagrams are shown: 1. the logical layout of a single 8-element vector block in the upper-left corner (which is also the shape of the \( t + 1 \) result vector block) and 2. the 8 elements that must be input for each of the 25 points in the stencil (some of these overlap). For additional clarity, one stencil position is highlighted in color along with its corresponding element in the single vector block. Observe that the number of results produced by each fold is the same (8), but the number of input elements differs due to varying amounts of vector-block overlap.

C. Memory layout

To load from and store to SIMD vectors most efficiently when using vector folding, memory layout for the grid should not follow a standard element-based row-major or column-major layout. Rather, all \( L_v \) elements of each aligned vector block should be located consecutively in memory. For example, for the 2x2x2 fold, the elements might appear in this order in memory: \((1,1,1), (1,1,2), (1,2,1), (1,2,2), (2,1,1), (2,1,2), (2,2,1), (2,2,2), (3,1,1), (3,1,2), (3,2,1), (3,2,2), (4,1,1), (4,1,2), (4,2,1), (4,2,2)\), where each row is a single aligned vector block of 8 elements. Thus, an access of an aligned vector block can be performed with one load or store without needing gather or scatter, even though it might contain elements across two or three dimensions. An access of an unaligned vector block requires loads of the two or more constituent aligned vector blocks and subsequent shifting and merging operations to construct the required elements within a hardware vector.

To further generalize, consider that the memory layout can be implemented as any 4D-to-1D mapping function \( A_n \leftarrow \mathcal{M}(t_n, i_n, j_n, k_n) \), where \( A_n \) is the memory address of the specified aligned vector block. By abstracting the memory layout in this way and completely separating it from the vectorization implementation, we enable a number of independent performance-tuning opportunities.

D. Memory/compute trade-off

As discussed above, various folds overlap differently when applied to the same stencil and require different numbers of instructions. For each fold in Figure 2, we can quantify several related cost factors that impact performance.

First, the 1x1x8 fold in Figure 2a requires 19 unique aligned vector reads: one in the center (starting from the red block), eight each along the \( x \) and \( y \) axes, and two along the \( z \) axis. Only two additional reads are required along the \( z \) axis because the vectorization is also in that direction, creating overlap. The overlap is not without cost, however; the construction of each of the 8 unaligned vector blocks on the \( z \) axis requires the merging of two constituent aligned vector blocks. We will define this cost as two generic “blend” operations\(^2\) per element, or 16 blends for all 8 \( z \)-axis points. A sequence of stencil calculations is usually performed in an inner loop as discussed in Section I. In the steady state, some of the vector blocks will most likely remain in cache between calculations. Thus, the absolute number of reads per stencil may not be as predictive of performance as the number of additional reads required for each stencil iteration. This number is dependent on the direction of the inner loop and can be determined by projecting the set of required aligned vector blocks onto a 2D plane perpendicular to the inner-loop direction. Projecting the 1x1x8 vector blocks in Figure 2a onto the \( y \)-\( z \) plane gives a footprint of 11 reads in the \( x \) inner-loop direction. Projecting onto the \( x \)-\( z \) plane also gives a footprint of 11 in the \( y \) direction, but projecting onto the \( x \)-\( y \) plane gives a footprint of 17 in the \( z \) direction. To summarize, the costs of the 1x1x8 fold are 19 reads with \((x, y, z)\) footprint-tuple \((11,11,17)\) and 16 blends. (The same costs hold for the logically-equivalent 1x8x1 and 8x1x1 folds, except the footprint values would be respectively permuted.)

Next, the 1x2x4 fold in Figure 2b allows reuse in both the \( y \) and \( z \) dimensions compared to only the \( z \) dimension with the 1x1x8 fold above. It requires 15 reads with footprint \((7,11,13)\) and 20 blends. (The same costs hold for the 1x4x2, 2x1x4, 2x4x1, 4x1x2, and 4x2x1 folds).

Lastly, 2x2x2 3D fold in Figure 2c is most compact and allows reuse in all three dimensions. It requires 13 reads with footprint \((9,9,9)\) and 24 blends.

If performance on a given architecture is limited by cache misses and has sufficient computation headroom, we would expect the implementation with the smallest footprint to perform best. For the alternatives presented above, that would be the

\(^{2}\)Based on the features in a given instruction-set architecture (SIMD shifting, masking, etc.), these two blends may be implementable in fewer or more than two actual CPU instructions.
1x2x4 fold, which has a footprint of 7 when the inner loop is in the x direction. (This does not account for any cache sharing on a multi-core and/or hyper-threaded architecture that might effectively further reduce the footprint of a multi-threaded implementation.)

The preceding discussion covered the costs of the three unique fold shapes for \( L_v = 8 \) on the stencil in Figure 1b. Table I presents the costs of the four unique fold shapes for \( L_v = 16 \) on the three basic stencil shapes in Figure 1 at orders (radii) of 4, 8 and 16 (except 16th-order planar because its large size of 817 points is considered impractical). The minimal footprint for each stencil across the four folds is shown in bold text. For the 4th-order stencils, the minimal footprint occurs in all three 2D and 3D folds. For the 8th and 16th-order stencils, the minimal fold is uniquely the 1x4x4 fold. The next section describes an implementation of a vector-folding kernel harness to experimentally measure the results on actual hardware.

IV. IMPLEMENTATION

This section describes our implementation of the vector-folding concept for performance evaluation. We discuss the target platform, the code generator, the kernel test harness, and the auto-tuning mechanism.

A. Target platform

We ran our experiments on the Intel® Xeon Phi™ Coprocessor 7120A [16]. The coprocessor board contains 16GB of RAM. Its 61 cores run at 1.238 GHz, and each supports 4-way hyper-threading for a maximum of 244 hardware threads. The hardware and instruction set support 512-bit SIMD: 16 single-precision or 8 double-precision floating-points values per each. There are 32 SIMD vector registers per thread and two FP ALUs per core. Thus, the maximum FP throughput is 16 \( \times \) 2 (for FMA) \( \times \) 1.238GHz \( \times \) 61 \( \approx \) 2416 SP GFLOPs/sec or 1208 DP GFLOPs/sec. Each core contains a 512KB L2 cache, and the caches are shared across cores for a combined L2 capacity of over 30MB. Much more coding and micro-architectural information is available in white-papers from Intel [2].

The Xeon Phi instruction set supports all the expected SIMD operations and two specific features that enable an efficient vector-folding implementation. The first is a set of instructions such as VALIGND and VPERMD that rearrange elements in vectors. The second is write-masking that controls which elements are written to a vector at the end of most SIMD instructions.

B. Code generator

Leveraging in-line vectorization is relatively straightforward. Code can be written directly in high-level languages such as C, C++, or FORTRAN and compiled using an auto-vectorizing compiler. Some optimizations such as ones that help avoid unaligned loads can also be successfully handwritten using “intrinsic” functions that allow access to the instructions listed above without resorting to writing assembly code. On the other hand, we have found that manually writing the code to correctly implement 2D and 3D vector folding is quite complex and error-prone, especially when constructing unaligned vector blocks not along a major axis.

To simplify this process, we developed a software framework that translates straightforward scalar stencil code into SIMD code with arbitrary vector folding. The framework is implemented in C++ and leverages operator overloading to extract an abstract-syntax tree (AST) from the stencil code. The AST is then traversed for each point in the target fold to generate the SIMD code. Grid-access function operators programmatically generate the required aligned memory reads and blend operations based on the offsets from the center point. Constant FP values and standard math operators complete the code to produce the correct SIMD vector output. Following is an example input stencil function that implements Equation 1:

```cpp
virtual GridValue nextValue(Grid& u, int t, int i, int j, int k) const {
    const float c[1 + order/2] = { ... };
    GridValue v = c[0] * u(t, i, j, k);
    for (int r = 1; r <= order/2; r++) {
        v += c[r] * u(t, i-r, j, k);
        v += c[r] * u(t, i+r, j, k);
        v += c[r] * u(t, i, j-r, k);
        v += c[r] * u(t, i, j+r, k);
        v += c[r] * u(t, i, j, k+r);
    }
    return v;
}
```

This code is compiled into the code-generator framework, creating a binary generator for the provided stencil. Then, the resulting generator is run with options specifying the value of the “order” variable and the desired vector fold. When the above example code is compiled and run for order=8 and a 4x2x2 fold, it outputs over 400 lines of C++ code including explanatory comments. Following is a slightly-simplified snippet of the code generated from the first line in the for-loop above when \( r = 1 \):

```cpp
// Read aligned vector blocks from memory.
float v0 = grid.readVec(0, i, j, k);
float v1 = grid.readVec(0, i-4, j, k);
// Construct the unaligned vector block
// starting at -1, 0, 0 with 4 elements
// from v1 and 12 from v0.
float v2 = _mm512_alignr_epi32(v1, v1, 3);
v2 = _mm512_mask_alignr_epi32(v2, v0, v0, 15, 0x0000);
// Accumulate result.
resultVec += 0.9f * v2;
```

In this code, “floatv” is a vector-block type, and “grid.readVec()” is an in-line function that reads one aligned vector block from memory. The “_mm512_alignr_epi32()” statement emits the VALIGND instruction that right-rotates the aligned vector block from \( u(0, i-4, j, k) \) by three elements. The “_mm512_mask_alignr_epi32()” statement then rotates the vector block from \( u(0, i, j, k) \) by 15 elements and uses the bit-mask “0x0000” to overlay 12 of the resulting elements onto the previous result to produce the final desired unaligned vector block starting at \( u(0, i-1, j, k) \). These two reads and two VALIGND instructions correspond to the read and blend costs discussed in Subsection III-D. Finally, the value is multiplied by the constant value from c[1] and added to the running sum using a vector FMA instruction.
TABLE I. Cost metrics for example stencils for $L_v = 16$.

<table>
<thead>
<tr>
<th>order</th>
<th>stencil</th>
<th>$1 \times 1 \times 16$</th>
<th>$1 \times 2 \times 8$</th>
<th>$1 \times 4 \times 4$</th>
<th>$2 \times 2 \times 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4th</td>
<td>13-pt 3-axis</td>
<td>11 (7,7,9); 8</td>
<td>9 (5,7,7); 12</td>
<td>9 (5,7,7); 16</td>
<td>7 (5,5,5); 16</td>
</tr>
<tr>
<td>37-pt 9-axis</td>
<td>35 (15,15,17); 40</td>
<td>29 (9,15,15); 60</td>
<td>29 (9,15,15); 80</td>
<td>19 (9,9,9); 80</td>
<td></td>
</tr>
<tr>
<td>61-pt 3-plane</td>
<td>43 (15,15,25); 72</td>
<td>29 (9,15,15); 108</td>
<td>29 (9,15,15); 144</td>
<td>19 (9,9,9); 144</td>
<td></td>
</tr>
<tr>
<td>8th</td>
<td>25-pt 3-axis</td>
<td>19 (11,11,17); 16</td>
<td>15 (7,11,13); 24</td>
<td>13 (5,11,11); 24</td>
<td>11 (7,7,9); 28</td>
</tr>
<tr>
<td>73-pt 9-axis</td>
<td>67 (27,27,33); 80</td>
<td>59 (15,27,33); 120</td>
<td>49 (9,27,27); 120</td>
<td>43 (15,15,25); 140</td>
<td></td>
</tr>
<tr>
<td>217-pt 3-plane</td>
<td>115 (27,28,81); 272</td>
<td>71 (15,27,45); 408</td>
<td>49 (9,27,27); 408</td>
<td>43 (15,15,25); 476</td>
<td></td>
</tr>
<tr>
<td>16th</td>
<td>49-pt 3-axis</td>
<td>35 (19,19,25); 32</td>
<td>27 (11,19,25); 44</td>
<td>25 (9,21,21); 48</td>
<td>21 (13,13,17); 56</td>
</tr>
<tr>
<td>145-pt 9-axis</td>
<td>131 (51,51,65); 160</td>
<td>119 (27,51,69); 220</td>
<td>129 (25,65,65); 240</td>
<td>117 (41,41,57); 280</td>
<td></td>
</tr>
</tbody>
</table>

Each table entry shows the number of reads (x, y, z footprint); number of blends for given fold.

Via polymorphism, classes derived from a generic code-generation class can produce code in different formats from the same input. The code shown above was created from a class that generates C++ code for Xeon Phi. Generating code for FORTRAN or other languages is possible, even though the input stencil code is written in C++.

C. Kernel evaluation harness

Performance data were gathered using a harness that essentially implements the pseudo-code from Subsection III-B. It adds traditional cache-blocking loops with blocking factors specified separately in each dimension. Grid memory is allocated in one consecutive block with enough space for $2 \times D_{ux} \times D_{uy} \times D_{uz}$ single-precision FP values plus halo (or ghost) borders. The multiplier of 2 provides memory for one “current” (t) and one “next” (t + 1) time-step, and their roles are swapped after each iteration as is typical for Jacobian code. Inter-process and inter-node halo exchanges using MPI or similar technologies are independent of vector folding and were thus not explored; all code is run in one OpenMP-threaded process. The order of the three spatial loops can be interchanged, and OpenMP threading may be applied within the cache-blocked regions to the outer-most spatial loop or “collapsed” across the two outer-most loops.

For measuring performance, each stencil implementation is executed for 4 trials of 50 time-step iterations each. These values were deemed sufficient because they exhibited small variations between trials compared to differences between experiments with different parameters. Each trial is timed separately, and the lowest time is reported.

D. Auto-tuning

In order to fairly compare the multicore performance of in-line vectorization compared to vector folding, we attempt to determine the near-best performance of both under various conditions. This requires tuning of the various settings provided in the harness described above. Since vector unfolding impacts the cache footprint and other behavior, the best settings without folds generally would not be the best with folds and vice-versa. The number of possible cache-blocking sizes in the three dimensions along with other settings creates an intractable enumeration of combinations that would require years to explore exhaustively. Thus, we used a genetic-algorithm (GA) auto-tuning system similar to the one described in [14] to find near-optimum settings for each particular stencil. For each desired result, the GA is run for at least three separate experiments to ensure that a local minimum is not prematurely found. The best result from the multiple experiments is reported.

V. EXPERIMENTAL RESULTS

In this section, we compare traditional in-line vectorization with vector folding for a variety of stencil shapes and sizes under different conditions. Each experiment is performed using the implementation described in the preceding section. Trials are run on the same eight stencils shown in Table I. Results are reported in GFLOPs/sec, which is a valid comparison metric because the number of FLOPs required is constant for any given stencil, regardless of the fold used to implement it.

A. Single-thread cache-local experiment

This experiment measures the benefits of vector folding in a single-threaded application on small problem size of $64^3$ cells. Approximately 4–5 MBytes of data are required, which easily fits within the shared L2 cache of the Xeon Phi. For each stencil, we run a set of control (in-line vector) trials and a set of vector-folding trials.

Each 1D in-line control trial fixes the fold to $1 \times 1 \times 16$ and the inner loop to the z dimension. We enable two orderings of the outer two spatial loops: z, y and y, z. We apply OpenMP to either just the outer or collapsed over both loops. We do not consider cache-blocking due to this this small problem size. Since the number of combinations is small, we exhaustively cover the search space.

Each vector-folding trial allows all of the possible vector folds for $L_v = 16$. We consider all 12 possible combinations of x, y and z-loop nesting orders. We allow OpenMP alternatives and disallow cache blocking as in the control trails, and we employ a similar exhaustive search.

Table II shows the results. For the in-line trails, we report the highest throughput for the $1 \times 1 \times 16$ fold. For the “best” result, we report the name of the highest-performing fold and its throughput. Note that the best performance in each case is achieved by a fold with the minimal footprint shown by the bold text in Table I. The speedup ranges from 1.22x to 1.85x with a geometric mean of 1.46x, showing that vector folding provides significant benefit over in-line vectors on a small problem size.
TABLE II. PERFORMANCE RESULTS IN GFLOPS/SEC FROM SMALL SINGLE-THREAD EXPERIMENT.

<table>
<thead>
<tr>
<th>order</th>
<th>stencil</th>
<th>in-line perf</th>
<th>best fold perf</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4th</td>
<td>13-pt 3-axis</td>
<td>1.985</td>
<td>1x2x8</td>
<td>2.656</td>
</tr>
<tr>
<td></td>
<td>37-pt 9-axis</td>
<td>2.853</td>
<td>2x2x4</td>
<td>3.483</td>
</tr>
<tr>
<td></td>
<td>61-pt 3-plane</td>
<td>3.654</td>
<td>2x2x4</td>
<td>4.480</td>
</tr>
<tr>
<td>8th</td>
<td>25-pt 3-axis</td>
<td>2.287</td>
<td>1x4x4</td>
<td>4.053</td>
</tr>
<tr>
<td></td>
<td>73-pt 9-axis</td>
<td>2.801</td>
<td>1x4x4</td>
<td>4.084</td>
</tr>
<tr>
<td></td>
<td>217-pt 3-plane</td>
<td>4.335</td>
<td>1x4x4</td>
<td>5.281</td>
</tr>
<tr>
<td>16th</td>
<td>49-pt 3-axis</td>
<td>2.283</td>
<td>1x4x4</td>
<td>4.014</td>
</tr>
<tr>
<td></td>
<td>145-pt 9-axis</td>
<td>1.552</td>
<td>1x4x4</td>
<td>2.874</td>
</tr>
<tr>
<td>geometric mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE III. PERFORMANCE RESULTS IN GFLOPS/SEC FROM LARGE MULTICORE EXPERIMENT.

<table>
<thead>
<tr>
<th>order</th>
<th>stencil</th>
<th>in-line perf</th>
<th>best fold perf</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4th</td>
<td>13-pt 3-axis</td>
<td>234.8</td>
<td>2x2x4</td>
<td>339.4</td>
</tr>
<tr>
<td></td>
<td>37-pt 9-axis</td>
<td>228.5</td>
<td>1x2x8</td>
<td>418.7</td>
</tr>
<tr>
<td></td>
<td>61-pt 3-plane</td>
<td>418.5</td>
<td>1x2x8</td>
<td>650.2</td>
</tr>
<tr>
<td>8th</td>
<td>25-pt 3-axis</td>
<td>249.0</td>
<td>1x4x4</td>
<td>538.7</td>
</tr>
<tr>
<td></td>
<td>73-pt 9-axis</td>
<td>297.5</td>
<td>1x4x4</td>
<td>622.0</td>
</tr>
<tr>
<td></td>
<td>217-pt 3-plane</td>
<td>451.3</td>
<td>1x4x4</td>
<td>727.9</td>
</tr>
<tr>
<td>16th</td>
<td>49-pt 3-axis</td>
<td>225.3</td>
<td>1x2x8</td>
<td>609.4</td>
</tr>
<tr>
<td></td>
<td>145-pt 9-axis</td>
<td>228.5</td>
<td>1x2x8</td>
<td>418.7</td>
</tr>
<tr>
<td>geometric mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B. Multicore cache-blocking experiment

This experiment measures the benefit of vector folding in a multicore environment on a larger problem size of 512² cells. Approximately 1.2 GBytes of data are required, which exceeds the shared L2 cache of the Xeon Phi.

The trails are run similar to the ones for the smaller problem size above except for the following differences:

- We use 60 cores of the Xeon Phi with 4 hyper-threads each.
- We allow any size of cache blocking from one to 512 (no blocking) along each axis.
- Since the number of possible combinations is intractable to search exhaustively, we use the genetic algorithm (GA) approach described in the preceding section and report the best performance for each stencil.

Table III shows the results with the in-line and best data as before. The best performance again corresponds to the fold with the smallest footprint except for the 16th-order stencils. In those cases, the best-performing fold (1x2x8) has a slightly larger footprint, but a lower number of blends; this may have been a deciding factor in this more complex multicore scenario. The speedup ranges from 1.45x to 2.71x with a geometric mean of 1.82x, showing that vector folding provides even more benefit over in-line vectors in a large problem size compared to the smaller one shown earlier.

VI. CONCLUSIONS

This paper described a general technique dubbed “vector folding” for representing data in vectors for 2D and 3D stencils. This method reduces the number of memory accesses required by storing a small multi-dimensional block of data in each vector, thereby increasing overlap between points for the stencil computation. Experiments on an Intel Xeon Phi show performance speedups over traditional vector representation ranging from 1.2x to 2.7x, depending on the problem size and stencil type.

Future work includes integration of this technique with other stencil algorithms, frameworks, libraries, domain-specific languages, and/or compilers.

REFERENCES


