Future Generation Computer Systems 92 (2019) 903–919

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Multi-level spatial and temporal tiling for efficient HPC stencil computation on many-core processors with large shared caches

Charles Yount [a,*], Alejandro Duran [b], Josh Tobin [c]

[a] Intel Corporation, USA
[b] Intel Corporation Iberia, Spain
[c] University of California at San Diego, CA, USA

HIGHLIGHTS

- Optimization of finite-difference stencils for Intel Xeon Phi processor.
- Discussion of multiple levels of tiling applied to three levels of memory cache.
- Focus on tuning and analysis of temporal wave-front tiling to maximize locality of large (16 GiB) global cache.
- Process to determine optimal sizing of temporal wave-front tiles.
- Presentation and analysis of performance data spanning a range of problem sizes and memory configurations.

ARTICLE INFO

Article history:
Received 7 April 2017
Received in revised form 15 September 2017
Accepted 25 October 2017
Available online 13 November 2017

Keywords:
Finite-difference method
Seismic modeling
Intel Xeon Phi
Temporal wave-front tiling
Vector-folding

ABSTRACT

Stencil computation is an important class of algorithms used in a large variety of scientific-simulation applications, especially those arising from finite-difference numerical solutions to differential equations representing the behavior of physical phenomenon such as seismic activity. The performance of stencil calculations is often bounded by memory bandwidth, and such code benefits from vectorization and tiling techniques to reuse data as much as possible once it is loaded from memory. These tiling algorithms are especially crucial for many-core CPU products that contain caches local to the individual cores, and this work provides a review of the use of techniques such as vector-folding and spatial tiling to maximize per-core cache resources. Recent many-core products also include special memory with much higher bandwidth than traditional DDR memory that is intended to provide additional performance for bandwidth-limited applications. On such platforms that also include DDR, the high-bandwidth RAM may be configurable either as separately addressable memory or as a large shared cache for the DDR. Examples of platforms with this feature include those containing products in the Intel® Xeon Phi™ x200 processor family (code-named Knights Landing), which use Multi-Channel DRAM (MCDRAM) technology to provide the higher bandwidth memory resources. In traditional sequential time-step stencil algorithms, the additional bandwidth can most easily be exploited when the stencil data fits into the faster memory, restricting the problem sizes that can be undertaken and under-utilizing the larger DDR memory on the platform. As stencil problem sizes become significantly larger than the fast-memory capacity, the sequential time-step algorithms create an overwhelming number of misses from the fast-memory shared cache, and the effective bandwidth approaches that of the DDR, significantly degrading performance. This paper illustrates this effect and explores the application of temporal wave-front tiling to alleviate it, simultaneously leveraging both the large cache's bandwidth and the DDR capacity. Two example applications are used to illustrate the optimizations: a single-grid isotropic approximation to the wave equation and a staggered-grid formulation for earthquake simulation. Details of the various tiling algorithms are given for both applications, and results on a Xeon Phi processor are presented, comparing performance across problem sizes and among four experimental configurations. Analyses of the bandwidth utilization and MCDRAM-cache hit rates are provided for one of the example applications, illustrating the correlation between these metrics and performance. It is demonstrated that temporal wave-front tiling can provide up to a 2.4x speedup compared to using the fast-memory cache without temporal tiling and 3.3x speedup compared to only using DDR memory for large problem sizes on the isotropic application. Respective speedups of 1.9x and 2.8x are demonstrated for the staggered-grid application.

* Corresponding author.
E-mail address: chuck.yount@intel.com (C. Yount).

https://doi.org/10.1016/j.future.2017.10.041

1. Introduction

Stencil computation is an important class of algorithms used in a large variety of scientific simulation applications. These techniques arise naturally from the application of finite-difference methods to find approximate numerical solutions to ordinary and partial differential equations (ODEs and PDEs) [1]. Conceptually, the kernel of a simple iterative single-stencil computation can be shown by the pseudo-code in Algorithm 1 that iterates over the points in a 1D temporal + 3D spatial grid where D_t is the number of time-steps; D_x, D_y, and D_z are the problem-sizes in the spatial dimensions; and $\mathcal{S}(t, i, j, k)$ is the stencil function [2].

Algorithm 1 Conceptual application of a single 1D temporal + 3D spatial stencil.

```
1: for t = 1 to D_t do
2:   for i = 1 to D_x do
3:     for j = 1 to D_y do
4:       for k = 1 to D_z do
5:         S(t, i, j, k)
6:       end for
7:     end for
8:   end for
9: end for
```

A simple stencil function might input the center value $u(t, i, j, k)$ and multiple neighboring values from a grid u and calculate a new result for $u(t + 1, i, j, k)$. The number of values input is often referred to as the number of *points* in the stencil. Samples of various symmetric 3D stencil shapes and sizes are illustrated in Fig. 1. For example, a stencil shaped like the 13-point one (a) contains the input pattern for this 4th-order 3D finite-difference formulation:

$$\begin{aligned}\mathcal{S}(t,i,j,k) \equiv u(t+1,i,j,k) \leftarrow c_0 u(t,i,j,k) + \\ \sum_{r=1}^{2} c_r \Big[u(t,i-r,j,k) + u(t,i+r,j,k) + \\ u(t,i,j-r,k) + u(t,i,j+r,k) + \\ u(t,i,j,k-r) + u(t,i,j,k+r) \Big]\end{aligned} \tag{1}$$

The 25-point stencil (b) might be used to formulate an 8th-order solution to the same differential equation to obtain higher accuracy from the same grid spacing or similar accuracy from a larger grid spacing [1]. The numerical solution to many differential equations and other useful simulation problems can be approximated with application of a single stencil. More complex problems require stencils that read from multiple grids or even multiple stencils that update more than one grid. These may be applied sequentially within the innermost spatial loop shown above or require multiple iterations of the spatial loops within each temporal iteration. This paper will show examples of both a single-stencil and a multiple-stencil problem.

For a significant set of stencil functions, like the one above, there are no dependencies between any two 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 vectorization, threading, and spatial tiling[1] techniques within a time-step. More advanced techniques that apply tiling across time-steps can enable even more performance opportunities, but they require careful coordination of the dependencies between time-step values. This paper describes and analyzes the performance of a suite of multi-level spatial and temporal tiling techniques that is designed to leverage multi-level caches on modern many-core CPUs. Experiments were run on a CPU from the Intel® Xeon Phi™ x200 processor family, which contains level-1 (L1) and level-2 (L2) caches local to one and two cores, respectively, as well as a large high-bandwidth memory that can be configured as a cache shared by all cores.

The remainder of the paper is organized as follows: Section 2 describes some aspects of the Intel Xeon Phi processor relevant to this work. Section 3 introduces a simple isotropic acoustic-wave stencil and discusses its implementation, focusing on the tiling algorithms used to leverage multiple levels of parallelism and caching. Section 4 introduces a more complex seismic problem whose solution is represented with a staggered-grid scheme. Sections 5 and 6 present the performance results on the target platform for the isotropic and staggered-grid stencils, respectively. Section 7 covers related work, and Section 8 concludes with a summary.

2. Target platform

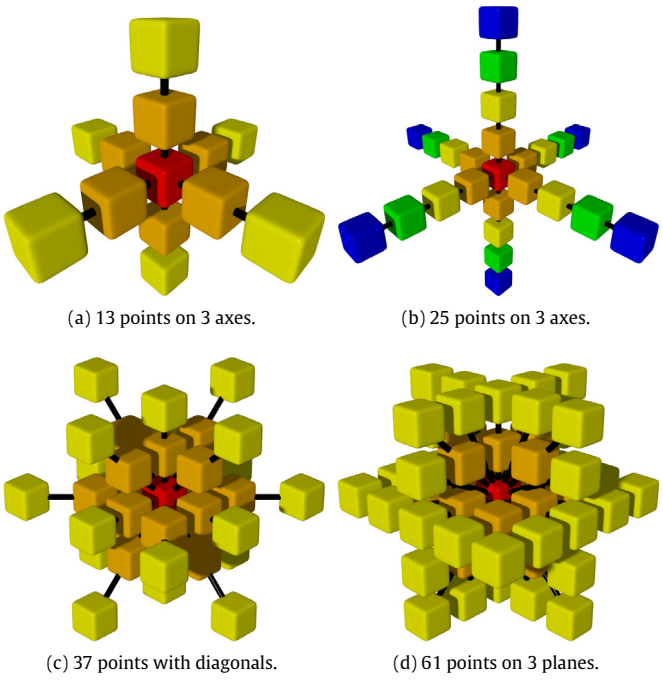
We are using in our study a system with a CPU from the Intel® Xeon Phi™ x200 processor family (code-named Knights Landing) [3]. These processors have a many-core architecture with up to 36 tiles per package. Each tile is composed of two cores that share a one-MiB second-level (L2) cache and an agent that connects the tile to a 2D mesh. This mesh connects all tiles and the memory controllers. Each core appears as four logical CPUs through hyper-threading. The four hyper-threads on a core share a 32 KiB first-level data (L1d) cache and a 32 KiB first-level instruction (L1i) cache. Each core is capable of issuing two instructions per cycle out-of-order, including vector and memory instructions. The product of the number of tiles, cores, and hyper-threads implies that there may be up to $36 \times 2 \times 4 = 288$ logical CPUs in a package, all visible to the operating system.

The Xeon Phi processors as well as the latest server products from Intel such as the Intel® Xeon® Scalable Processors implement a 512-bit SIMD instruction set known as AVX-512 [4]. Each SIMD register in this architecture may contain eight double-precision (DP) or sixteen single-precision (SP) floating-point values as well as a variety of integer data sizes. The AVX-512 instruction set also supports unaligned loads, fused-multiply and add, vector masking, shuffle and permutation instructions, advanced vector gather and scatter, histogram support, and hardware-accelerated transcendentals.[2]

To provide the cores with enough data to feed the computing capabilities, the Xeon Phi processors may be configured with an integrated on-package Multi-Channel DRAM (MCDRAM) memory of up to 16 GiB. The MCDRAM can deliver up to 490 GB/s of bandwidth compared to the main DDR4 memory on the same platform that can deliver about 90 GB/s [5]. This memory can be configured at boot time in one of three different modes shown in Fig. 2: flat, cache, or hybrid. In flat mode, the integrated memory isvisible

[1] The terms "tiling", "loop-tiling", "blocking", and "cache-blocking" are often used interchangeably in the literature.

[2] A few specialized instructions are specific to either the Xeon Phi or Xeon CPUs.



(a) 13 points on 3 axes. (b) 25 points on 3 axes.

(c) 37 points with diagonals. (d) 61 points on 3 planes.

Fig. 1. Example symmetrical 3D stencil shapes and sizes [2].

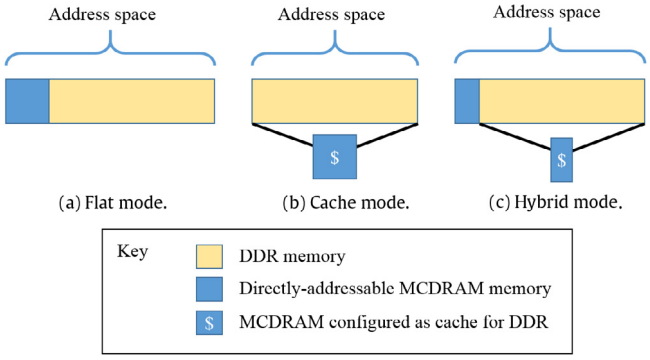


Fig. 2. Available Multi-Channel DRAM configuration modes.

to the programmer in the same memory space as regular system memory but as different NUMA domains. This provides the most control to programmers over which kind of memory is allocated for various purposes or datasets. In cache mode, the MCDRAM cannot be accessed directly by the programmer, but it acts as an additional level of cache in between the L2 caches and the DDR memory. This cache is a direct-mapped cache and is inclusive of the L2 caches. Access hits in the cache will be served directly by the MCDRAM, while misses are redirected to the DDR controllers which serve the data back to both the MCDRAM cache and the L2 that made the request. The hybrid mode allows one to configure a portion of MCDRAM as flat Mode and another in cache mode, each with the respective characteristics just described.

More information on the Xeon Phi and MCDRAM can be found at [4].

3. Single-stencil application and implementation

This section describes the first stencil equation we selected for evaluation and the stencil-optimization techniques targeting features of the Intel Xeon Phi architecture. We apply three hierarchical tiling algorithms at the single-processor node level: vector-folding to increase SIMD overlap, spatial cache blocking to increase per-core cache locality within threads, and temporal wave-front tiling to exploit the global MCDRAM cache. For multi-node (cluster) applications, standard techniques such as halo exchanges via MPI message-passing can be used, but they are not discussed in this paper.

3.1. Iso3DFD formulation

The tiling techniques described can be applied to a wide variety of stencil equations. We selected a high-order isotropic 3D finite-difference ("Iso3DFD") stencil as an example. The Iso3DFD stencil solves the following wave-equation:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \nabla^2 p$$

with the approximation shown in Eq. (2) [6,7], where $\mathcal{S}(t, i, j, k)$ is applied to the points in the problem domain as shown in Algorithm 1. The core of this equation is similar to Eq. (1), but this one has higher accuracy (16th-order accuracy in space and 2nd-order accuracy in time) and has an additional input from a constant v grid, which contains pre-computed values of $c^2 \Delta t^2$ at each spatial point. In total, the stencil inputs 52 points to calculate each new value.

$$\begin{aligned}
&\mathcal{S}(t, i, j, k) \equiv \\
&p(t+1, i, j, k) \leftarrow 2p(t, i, j, k) - p(t-1, i, j, k) + \\
&\quad v(i, j, k)\Big(c_0 p(t, i, j, k) + \\
&\quad \sum_{r=1}^{8} c_r \Big[p(t, i-r, j, k) + p(t, i+r, j, k) + \\
&\qquad p(t, i, j-r, k) + p(t, i, j+r, k) + \\
&\qquad p(t, i, j, k-r) + p(t, i, j, k+r) \Big] \Big)
\end{aligned} \tag{2}$$

3.2. Memory requirements

As Eq. (2) is written, there is a unique index in p for every time-step t. In most applications, though, it is not practical or even desired to save the results after every time-step. Results may be saved at some regular interval or only after the final time-step. Thus, memory may be reused because only results that need to be referenced to compute the result at $t + 1$ need to be kept. In Eq. (2), inputs for $t + 1$ are read from t and $t - 1$, so at most we would need to keep two previous results while calculating the next one. In addition, this equation has the property that the only input read from $t - 1$ is at spatial index (i, j, k), which is the same index as the one being updated at $t + 1$. Under this condition, we can make the further optimization that the algorithm can read from $p(t - 1, i, j, k)$, make the calculation for $p(t + 1, i, j, k)$, and store that new value back into the same memory location because it will not be needed by any other calculation. Therefore, if processing data in the order shown in the pseudo-code loops from Section 1, i.e., with the spatial loops inside the time loop, we need to only allocate memory for two 3D grids for p and one for v.

This is illustrated in Fig. 3 where all values for $t = 1$ and $t = 2$ have been calculated, and the values for $t = 3$ are being calculated in ascending x indices (left-to-right). In this case, the memory used to store the $t = 1$ values can be reused to store the $t = 3$ values. This can be generalized for any values of $t + 1$ and $t - 1$. (The key in Fig. 3 applies to all diagrams in this section.)

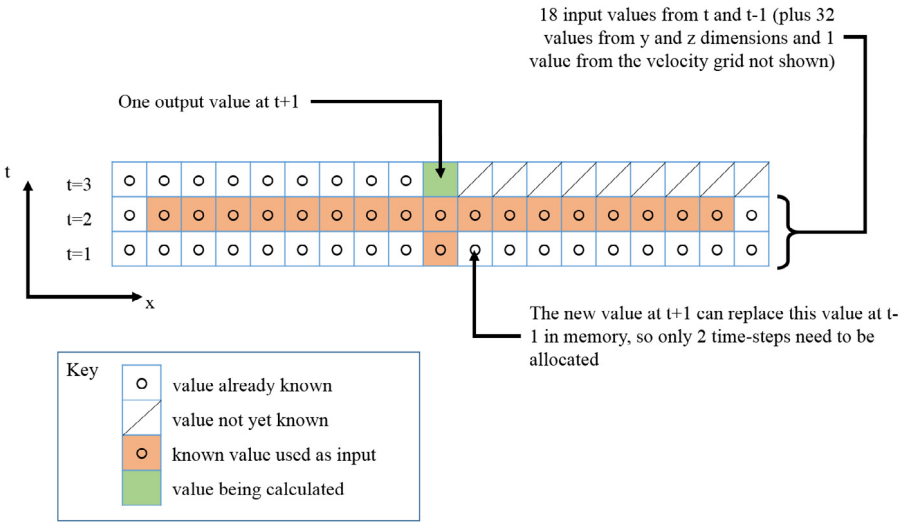


Fig. 3. Time–space diagram of the pressure grid from Eq. (2) (showing t and x dimensions only).

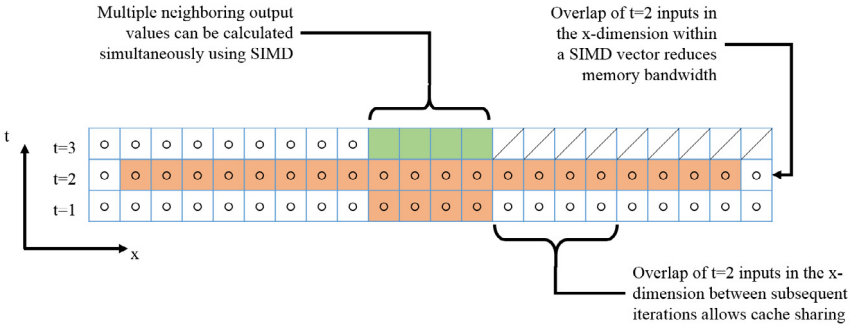


Fig. 4. Parallel update of values in a SIMD vector within a single time-step.

3.3. SIMD

Applying SIMD to stencil code is based on the observation that $p(t+1, i, j, k)$ does not depend on any other value of p at $t+1$ for all i, j, and k. Thus, multiple neighboring outputs can be calculated simultaneously in a given time-step, leveraging the SIMD throughput of a vector-enabled ALU. In addition to the benefits afforded by executing the floating-point operations in parallel, there is also a reduction of cache and/or memory bandwidth due to the overlap between inputs as illustrated in Fig. 4. This overlap often requires accessing SIMD vectors that are not aligned to a multiple of the SIMD width, often crossing cache-line boundaries as well. For this overlap to be advantageous, the architecture should support unaligned and/or cache-line splitting loads with little additional overhead. An alternative is for the software to load the two constituent aligned vectors and produce the unaligned data via one or more shuffle instructions.

As described in Section 2, the Intel Xeon Phi architecture SIMD registers may contain eight double-precision (DP) or sixteen single-precision (SP) floating-point values. A straightforward application of SIMD to stencils like Iso3DFD is to apply vectorization *in-line*, i.e., a 1D series of elements in the same direction of the inner-most spatial for-loop as shown in Algorithm 2. In the Xeon Phi, eight or sixteen results can be efficiently calculated simultaneously in each of two vector execution units. When processing fused-multiply and add instructions (FMAs), each execution unit can output up to one result each clock, for a total of $8 \times 2 \times 2 = 32$ DP or 64 SP floating-points operations each cycle (per core). Other floating-point operations can complete at half that rate. Additionally, the support of low-overhead unaligned loads and shuffle instructions enables increased performance from sharing between SIMD vector inputs.

Algorithm 2 In-line N_z-wide SIMD parallelization, assuming D_z is a multiple of N_z.

```
1:  // Problem of size Dt × Dx × Dy × Dz.
2:  for t = 1 to Dt do
3:    for i = 1 to Dx do
4:      for j = 1 to Dy do
5:        for k = 1 to Dz by Nz do
6:          // SIMD of size Nz.
7:          S(t, i, j, k to k + Nz − 1)
8:        end for
9:      end for
10:   end for
11: end for
```

3.4. Index-traversal notation

Before continuing with optimizations, we introduce a more generalized notation for traversing rectilinear solids. Algorithms 1 and 2 use a traditional nested-loop pseudo-code notation for scanning the three spatial dimensions. This notation implies that the points are visited with ascending indices in the specified nest order. Since any particular order of evaluation within a given time-step is not required, we will generalize the concept of *traversing, visiting,* or *scanning* points in a 3D space with the following notation:

for all $(i, j, k) \in [(X_{first}$ **to** X_{last} **by** $X_{step}) \times (Y_{first}$ **to** Y_{last} **by** $Y_{step}) \times (Z_{first}$ **to** Z_{last} **by** $Z_{step})]$ **do**
  $\mathcal{F}(t, i, j, k)$
end for

which indicates that $\mathcal{F}(t, i, j, k)$ is applied to the points in the 3D solid as follows:

- The indices to be visited in the x dimension are X_{first}, $X_{first} + X_{step}$, $X_{first} + 2 \times X_{step}$, ..., X_{last}, assuming that $(X_{last} - X_{first} + 1)$ is an exact multiple of X_{step}. The same applies to y and z.
- $\mathcal{F}(t, i, j, k)$ is applied to all possible combinations of these indices in x, y, and z.
- The total number of points visited is $\lfloor (X_{last} - X_{first} + 1)/X_{step} \rfloor \times \lfloor (Y_{last} - Y_{first} + 1)/Y_{step} \rfloor \times \lfloor (Z_{last} - Z_{first} + 1)/Z_{step} \rfloor$.
- The applied function $\mathcal{F}(t, i, j, k)$ may be a stencil or another nested scan.
- If the **by** modifier is not given, the default step is one (1) for that dimension.
- The order of traversal (path) is not implied. Traditional nested loops with any nesting order are the most straightforward paths. Various space-filling curves describe other alternatives.
- The assumption that $(X_{last} - X_{first} + 1)$ is an exact multiple of X_{step} (and equivalents in the other dimensions) allows $\mathcal{F}(t, i, j, k)$ to cover a sub-space exactly sized $X_{step} \times Y_{step} \times Z_{step}$ without special considerations. In practice, this restriction is not necessary, and $\mathcal{F}(t, i, j, k)$ must handle smaller sizes in any or all dimensions due to any remainder from $(X_{last} - X_{first} + 1)/X_{step}$ (and/or equivalents in the other dimensions). However, for clarity, the remainder of algorithms in this paper will make the simplifying assumption.
- This concept of scanning rectilinear solids may be extended to other shapes in more or fewer dimensions, including those that are not convex or continuous. Again, for clarity, these will not be considered in the following discussions.

Algorithm 3 shows the generalization of Algorithm 1 expressed in this notation. The notation clarifies that the time-steps are evaluated strictly in-order, but the spatial traversal order within each time-step is arbitrary.

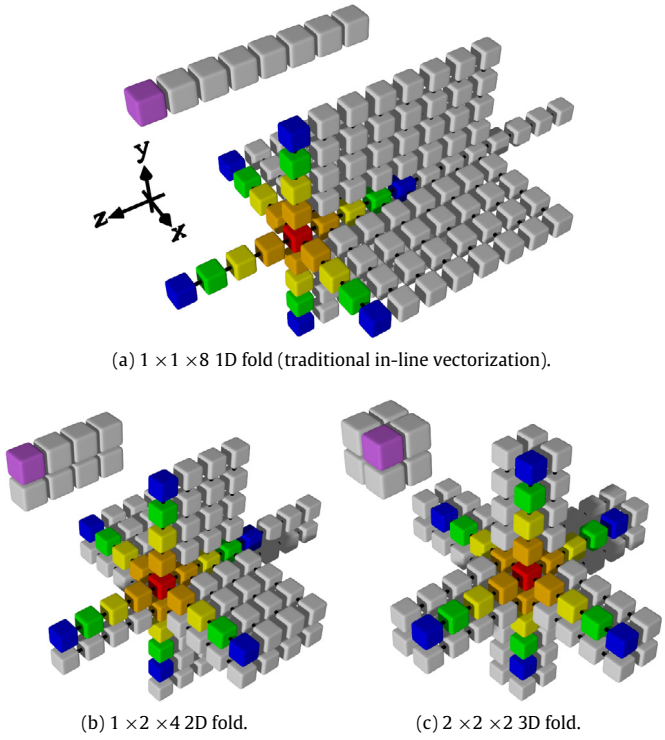
Algorithm 3 Single-stencil kernel with generalized spatial scanning.

```
1: // Problem of size D_t × D_x × D_y × D_z.
2: for t = 1 to D_t do
3:     for all (i, j, k) ∈ [(1 to D_x) × (1 to D_y) × (1 to D_z)] do
4:         S(t, i, j, k)
5:     end for
6: end for
```

3.5. Vector-folding

The traditional in-line vectorization approach described above does not always result in the most efficient use of cache or memory bandwidth because there is little reuse of data within the SIMD vectors between subsequent iterations of the stencil-evaluation loop. By applying a technique known as "vector-folding" [2], the reuse may be increased for certain stencils. This technique decreases the bandwidth demand at the expense of more element shuffling and a non-traditional data layout. The extensive and highly flexible shift and permute instructions available in AVX-512 make the required 2D or 3D shuffling cost relatively minor, and the customized data layout may be encapsulated within a software structure for little or no performance cost.

The central concept behind vector-folding is that small multi-dimensional tiles of data are stored within a SIMD register instead of the traditional one-dimensional representation. Fig. 5 illustrates three ways to pack eight double-precision floating-point values into a 512-bit SIMD register. Using 2D or 3D vector-folding extends the overlap illustrated in Fig. 4 from the one dimension shown to two or three dimensions, reducing the number of memory loads required for each stencil calculation, and thus reducing the memory-bandwidth demands. To obtain this efficiently, each aligned 2D or 3D vector block must be stored in consecutive bytes of memory. Unaligned vector blocks are constructed by first loading the constituent aligned blocks and then applying the required permutations. This is analogous to performing an unaligned load (or two aligned loads and a shuffle) in the 1D case. Fig. 6 compares the memory layout and aligned loads of in-line SIMD versus vector-folding (top and middle diagrams), and it shows an example of loading an unaligned folded vector (bottom diagram).



(a) 1 ×1 ×8 1D fold (traditional in-line vectorization).

(b) 1 ×2 ×4 2D fold.

(c) 2 ×2 ×2 3D fold.

Fig. 5. Various folds of 8 elements [2]. The smaller diagram in the upper-left of each sub-figure illustrates a single SIMD layout, and the larger diagram shows the input values needed for the 25-point stencil from Fig. 1(b).

Earlier experiments on the Intel Xeon Phi x100 product-family coprocessor showed significant performance speedups over traditional vector representation, depending on the problem size and stencil type [2]. Since the Xeon Phi x200 family processors implement similar instructions to the x100 family and additionally contain a richer set of shift and permute instructions, the vector-folding technique is easily extended to the x200 family and also shows significant speedup [8]. Using the notation introduced above, application of a vector-fold to a single stencil is illustrated in Algorithm 4.

3.6. Threading and spatial cache-blocking

The next level of tiling we apply is targeted to effectively utilize the many cores and threads of the Intel Xeon Phi architecture and also to increase cache locality. The parallelization used in this technique is based on the same observation that $p(t + 1, i, j, k)$ does not depend on any other value of p at $t + 1$. Thus, not only

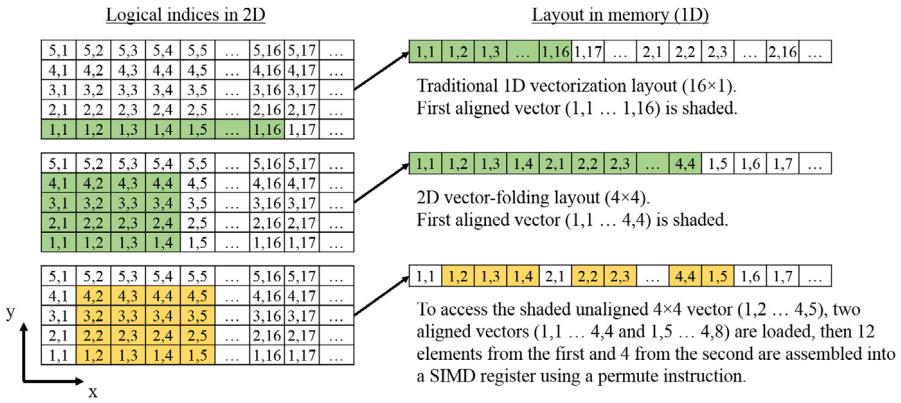


Fig. 6. Extracting efficiency from vector folding requires modifying the memory layout and generating code to assemble unaligned vectors from aligned loads [8].

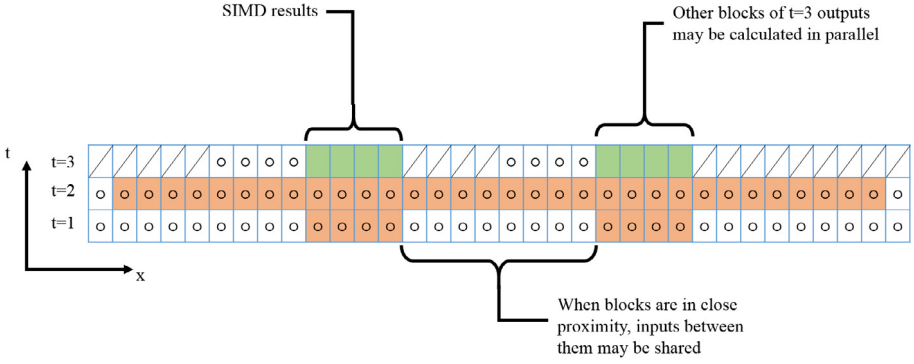


Fig. 7. Parallel update of values in multiple blocks within a single time-step.

Algorithm 4 $N_x \times N_y \times N_z$ vector-folded SIMD parallelization.

```
1: // Problem of size D_t × D_x × D_y × D_z.
2: for t = 1 to D_t do
3:     for all (i, j, k) ∈ [(1 to D_x by N_x) × (1 to D_y by N_y) × (1 to
       D_z by N_z)] do
4:         // Vector-folded SIMD of size N_x × N_y × N_z.
5:         S(t, i to i + N_x − 1, j to j + N_y − 1, k to k + N_z − 1)
6:     end for
7: end for
```

Algorithm 5 Parallel update of multiple blocks within a single time-step.

```
1: // Problem of size D_t × D_x × D_y × D_z.
2: for t = 1 to D_t do
3:     // Blocks evaluated in parallel.
4:     for all (i_1, j_1, k_1) ∈ [(1 to D_x by B_x) × (1 to D_y by B_y) × (1 to
       D_z by B_z)] do
5:         // Block of size B_x × B_y × B_z.
6:         for all (i, j, k) ∈ [(i_1 to i_1 + B_x − 1 by N_x) × (j_1 to
           j_1 + B_y − 1 by N_y) × (k_1 to k_1 + B_z − 1 by N_z)] do
7:             // Vector-folded SIMD of size N_x × N_y × N_z.
8:             S(t, i to i + N_x − 1, j to j + N_y − 1, k to k + N_z − 1)
9:         end for
10:    end for
11:    // End of parallel section.
12: end for
```

can neighboring values be updated simultaneously as in the SIMD case, but any arbitrary values within a time-step can be updated in parallel as illustrated in Fig. 7, where the values for $t = 3$ are not necessarily updated in ascending order in the x dimension. This concept may be directly extended to the y and z dimensions.

When applying threading, we divide the $D_x \times D_y \times D_z$ spatial problem size into smaller blocks, each of size $B_x \times B_y \times B_z$. The blocks are sized to increase the cache locality of the hardware unit ultimately responsible for evaluating the block of results. If B_n does not divide evenly into D_n for any dimension n, smaller remainder blocks are created as noted earlier. In addition, B_x must be a multiple of the vector-fold dimension N_x to avoid requiring partial-vector calculations in the x dimension. The same holds for the other spatial dimensions.[3] Parallelization can then be applied to the resulting blocks as shown in Algorithm 5 using OpenMP or similar technologies.

When applying SIMD, vector-folding, and spatial cache-blocking, there are no additional memory requirements. In the Iso3DFD example, allocating memory for two temporal values across all spatial indices is still sufficient.

3.7. Cache sharing optimization within cache blocks

The most straightforward scheme for mapping CPUs to cache blocks is to assign software threads to blocks via a simple OpenMP parallel for-loop. Unfortunately, this is not optimum because each thread would be working on separate portions of memory with very little inter-block sharing (except possibly a small amount at neighboring-block boundaries). Additionally, in the Intel Xeon Phi architecture, the eight threads in each tile (recall that a tile is a pair of cores) would be competing for space in the L2, and the four threads in each core would be competing for space in the L1 (see

[3] Partial-vector calculations are possible using either SIMD masking or scalar instructions, but this specialization is not discussed here for clarity.

Section 2). This can be mitigated by using fewer threads or making the cache blocks very small, but there is a more optimal approach.

To increase L1 and L2 cache sharing *between* cores on a tile and threads in a core, we can use two-level nested OpenMP parallel regions. At the outer level, we assign Xeon Phi tiles to blocks via an OpenMP parallel-for loop. Then, at the inner level, we assign threads to *sub-blocks* within each block via a nested OpenMP parallel-for loop as shown in Algorithm 6 . Because of the high-order nature of the Iso3DFD stencil, there is a significant amount of reuse between the sub-blocks, allowing the threads in a tile to cooperate instead of compete for cache space. This can be seen in Fig. 7, where the two blocks being updated at $t = 3$ share overlapping dependencies from $t = 2$.

Algorithm 6 Pseudo-code for parallel update of multiple blocks with nested parallelism within each block.

```
1:  // Problem of size D_t × D_x × D_y × D_z.
2:  for t = 1 to D_t do
3:    // Blocks evaluated in parallel by Xeon Phi tiles.
4:    for all (i_1, j_1, k_1) ∈ [(1 to D_x by B_x) × (1 to D_y by B_y) × (1 to
      D_z by B_z)] do
5:      // Block of size B_x × B_y × B_z.
6:      // Sub-blocks evaluated in parallel by Xeon Phi hyper-threads.
7:      for all (i_0, j_0, k_0) ∈ [(i_1 to i_1 + B_x − 1 by S_x) × (j_1 to
        j_1 + B_y − 1 by S_y) × (k_1 to k_1 + B_z − 1 by S_z)] do
8:        // Sub-block of size S_x × S_y × S_z.
9:        for all (i, j, k) ∈ [(i_1 to i_1 + S_x − 1 by N_x) × (j_1 to
          j_1 + S_y − 1 by N_y) × (k_1 to k_1 + S_z − 1 by N_z)] do
10:         // Vector-folded SIMD of size N_x × N_y × N_z.
11:         S(t, i to i + N_x − 1, j to j + N_y − 1, k to k + N_z − 1)
12:       end for
13:     end for
14:     // End of sub-blocks in nested parallel section.
15:   end for
16:   // End of blocks in outer parallel section.
17: end for
```

3.8. Temporal wave-front tiling

As stated above, spatial blocking is relatively straightforward for stencils like the one in Eq. (2) because the values calculated for $t + 1$ depend only on values in previous time-steps. Thus, the order in which values or spatial blocks of values are calculated is completely arbitrary and can be done in parallel. Temporal tiling[4] is a category of techniques whereby the spatial blocking technique described above is extended so that multiple time-steps are evaluated in a subset of the overall problem domain [9–11].

As opposed to the spatial-blocking schemes, one must more carefully consider the dependencies between time-steps when applying temporal tiling. Calculations of new values at the border between two tiles are illustrated in Fig. 8. In this diagram, values are shown being calculated in ascending x indices (left-to-right). Updates for this tile ends where indicated for $t = 1$, and we wish to calculate values for $t = 2$ before the remaining values at $t = 1$ are provided. Due to the dependencies between $t = 1$ and $t = 2$, we must stop updating values for $t = 2$ at a lower x index than we did for $t = 1$. Specifically, for the 16th-order stencil in Eq. (2), we must stop 8 indices earlier. The ratio of 8 spatial indices to each temporal index is referred to as the *wave-front angle* in the x dimension. This continues for each time-step, creating the *wave-front* boundary indicated by the bold line in the diagram. This results in a trapezoid-shaped set of known values to the left of the boundary. We will refer to the last values calculated along the boundary as the *trailing-edge* of the wave-front.

The wave-front shape can be logically extended into the three spatial dimensions. For this stencil, the wave-front angle is also eight (8) in the y and z dimensions. For example, consider a 2048^3 overall-problem domain broken into tiles no larger than 512^3 elements in the spatial dimensions. In the first tile calculated starting in the (1, 1, 1) "corner" of the overall space, the $t = 1$ values can be calculated for all 512^3 values, [1, 512] in each spatial dimension. Then, the $t = 2$ values can only be calculated for [1, 504] in each dimension, the $t = 3$ values for [1, 496], etc., up to the last value of t to be evaluated for this tile, creating a hyper-trapezoid of values.

Assuming tiles are being calculated sequentially, after the first tile's values are calculated, another tile can be calculated adjacent to it. Again referring to Fig. 8, after the tile to the left of the wave-front boundary is completed, the tile to the right of it may be calculated. For this second tile, the values for $t = 1$ start at the x index immediately following the last one completed in the first tile for $t = 1$. Similarly, for $t = 2$, the starting x index follows the last one completed in the first tile for $t = 2$, thus "filling-in" the missing values caused by the wave-front angle. We refer to the first values calculated along the boundary as the *leading-edge* of the wave-front, fitting exactly against the trailing-edge left by the previous tile. If the second tile is started after the first tile is completed, all dependencies will be available and the results of the calculations will be the same as if temporal tiling were not used. If the second tile extends to the end of the problem domain, another trapezoid shape of values will result because the final x values will be the same for all t. However, if the second tile is an "interior" one (i.e., not abutting the edge of the problem domain), a parallelogram of values will be calculated. Again, this can extend to multiple spatial dimensions, and the resulting hyper-parallelogram-shaped tiles will ultimately calculate all values for the desired number of time-steps. At that point, another temporal layer of tiles may be calculated for additional time-step values.

Perhaps surprisingly, temporal wave-front tiling requires no additional memory beyond that required for non-blocked or spatial cache-blocking schemes. As each time-step is calculated, values along the trailing edge of the wave-front boundary are "left-behind" in memory. Unlike in the non-temporally-blocked scheme, the values in memory are not from two time-steps only. Rather, the memory contains values from all the time-steps calculated, but only those along the trailing-edge boundary, including the "top" of the trapezoid. This is illustrated in Fig. 9, which shows values from five time-steps left in memory along the trailing edge of the left-hand-side tile. When the next tile is constructed, those values are exactly what are needed when computing values along the leading edge. After all tiles are completed, the values in memory are the same as they would have been if temporal wave-front tiling had not been used.

It is possible for temporally-blocked tiles to be calculated in parallel under certain conditions. For example, in a single spatial dimension, diamond-shaped tiles may be created in parallel. In diamond-tiling, trailing edges are first created on both ends of a tile. Then, leading edges along both ends of another phase of tiles fill in between the earlier trailing edges. Extending this to multiple spatial dimensions requires applying different techniques to other dimensions or creating more complex interlocking shapes [12].

In this work, however, we employ a more straightforward scheme using *sequential* tile updates specifically to take advantage of the large shared MCDRAM cache in the Intel Xeon Phi processors. A diagram of four tiles evaluated sequentially for a single spatial dimension is shown in Fig. 10. The temporal wave-front tiles are sized to fit within the MCDRAM cache to sizes $W_x \times W_y \times W_z$ spatially. The number of time-steps in a tile, W_t is chosen empirically as discussed in Section 5. Within each time-step of a

[4] Although the terms "block" and "tile" are often used interchangeably, in this section, we [arbitrarily] use the term "block" for spatial-only grouping and "tile" when multiple temporal updates are allowed.

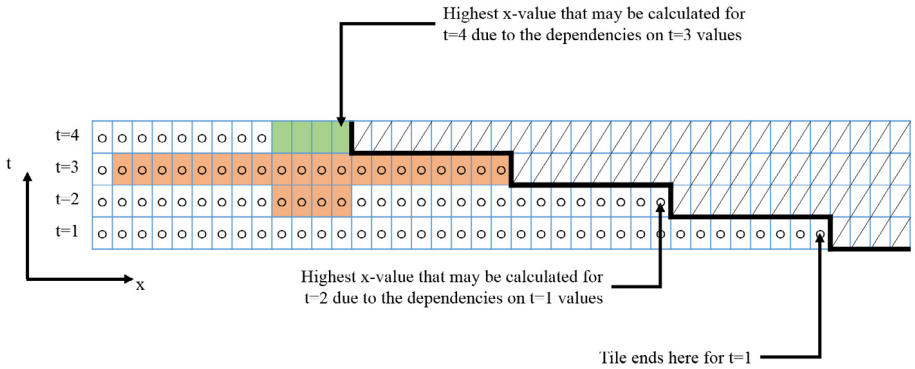


Fig. 8. Update of values in a wave-front tile for multiple time-steps.

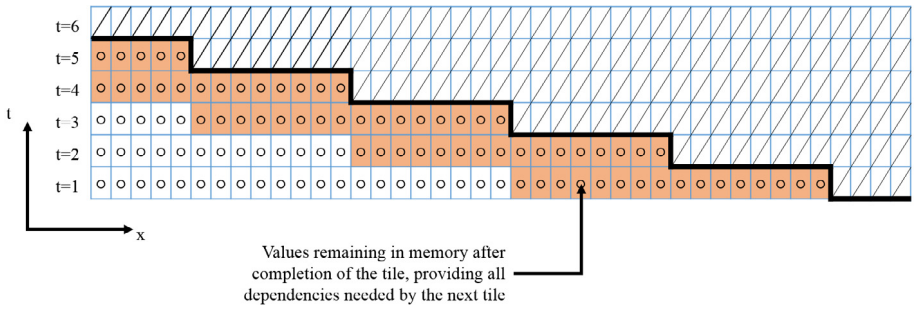


Fig. 9. Memory sharing along a wave-front tile boundary.

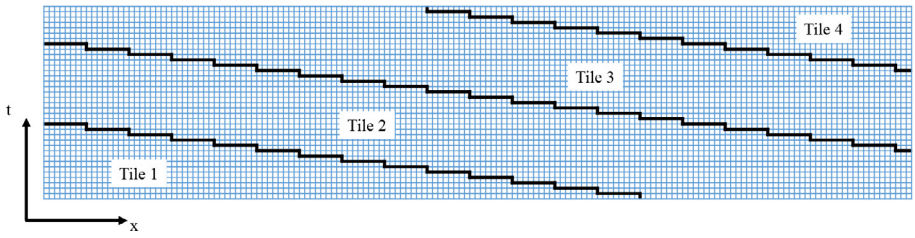


Fig. 10. Multiple wave-front tiles evaluated sequentially.

tile, SIMD, vector-folding, and multi-threaded spatial blocking are applied as shown in Algorithm 7. This algorithm illustrates how the wave-front angles A_x, A_y, and A_z are used to offset the cache-block starting and ending indices in each time-step. The algorithm makes the simplifying assumption that A_x has been rounded up to multiple of N_x, etc. This avoids evaluating partial SIMD vector results.

Since the MCDRAM is shared among all cores, the cost of an access to DDR is amortized across all cores. Calculating the first time-step in a given tile causes almost all accesses to miss the MCDRAM cache. However, subsequent time-steps cause misses only along the leading-edge boundaries like that of the second tile described above. All other accesses not on the boundaries are likely[5] to hit in the MCDRAM cache, so if successful, the resulting bandwidth (and thus overall performance) should be dominated primarily by MCDRAM bandwidth and degraded only slightly by the DDR accesses. This scheme is implemented and its impact on performance is quantified in Section 5.

[5] Cache-tag collisions will cause misses even in the interior.

4. Staggered-grid application and implementation

This section describes the second stencil equation we selected for evaluation, which involves the update of multiple grids, and

4.1. AWP formulation

AWP-ODC is a software package that simulates the propagation of seismic waves through a heterogeneous 3D medium. "AWP-ODC" is an acronym for "Anelastic Wave Propagation - Olsen, Day, Cui", named after its three main developers. It is widely used, particularly within the Southern California Earthquake Center, for example [13].

The governing equations are given by the velocity–stress formulation of the elasto-dynamic equations of motion, and AWP-ODC additionally incorporates frequency-dependent anelastic attenuation using the memory-efficient course-grained approach of [14]. The velocity–stress formulation results in a system of nine coupled PDEs, which are solved numerically with the finite-difference method. A central-difference staggered-grid scheme is used that is fourth-order accurate in space and second-order accurate in time. The system is staggered both temporally and

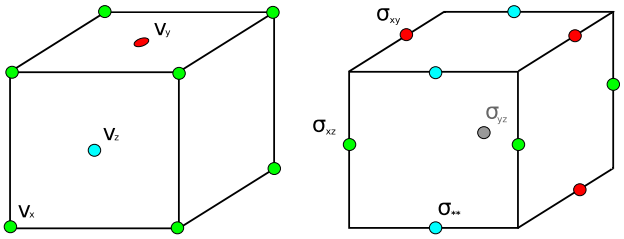
Algorithm 7 Temporal wave-front tiling encompassing nested parallel cache-blocking and vector-folding.

```
1:  // Problem of size D_t × D_x × D_y × D_z.
2:  // Wave-front angles A_x, A_y, A_z.
3:  // Wave-front tiles evaluated sequentially.
4:  for t_2 = 1 to D_t by W_t do
5:    for i_2 = 1 to D_x + (W_t × A_x) by W_x do
6:      for j_2 = 1 to D_y + (W_t × A_y) by W_y do
7:        for k_2 = 1 to D_z + (W_t × A_z) by W_z do
8:          // Initialize wave-front offsets for this tile.
9:          O_x = O_y = O_z = 0
10:         // Wave-front tile of size W_t × W_x × W_y × W_z.
11:         for t_1 = t_2 to t_2 + W_t − 1 do
12:           // Blocks evaluated in parallel by Xeon Phi tiles.
13:           for all (i_1, j_1, k_1) ∈ [(i_2 − O_x to
              i_2 − O_x + W_x − 1 by B_x) × (j_2 − O_y to
              j_2 − O_y + W_y − 1 by B_y) × (k_2 − O_z to
              k_2 − O_z + W_z − 1 by B_z)] do
14:             // Block of size B_x × B_y × B_z.
15:             // Sub-blocks evaluated in parallel by Xeon Phi
                hyper-threads.
16:             for all (i_0, j_0, k_0) ∈ [(i_1 to i_1 + B_x − 1 by S_x) × (j_1 to
                j_1 + B_y − 1 by S_y) × (k_1 to k_1 + B_z − 1 by S_z)] do
17:               // Sub-block of size S_x × S_y × S_z.
18:               for all (i, j, k) ∈ [(i_1 to i_1 + S_x − 1 by N_x) × (j_1 to
                  j_1 + S_y − 1 by N_y) × (k_1 to k_1 + S_z − 1 by N_z)] do
19:                 if i ∈ [1, D_x] and j ∈ [1, D_y] and k ∈ [1, D_z] then
20:                   // Vector-folded SIMD of size N_x × N_y × N_z.
21:                   S(t, i to i+N_x−1, j to j+N_y−1, k to k+N_z−1)
22:                 end if
23:               end for
24:             end for
25:             // End of sub-blocks in nested parallel section.
26:           end for
27:           // End of blocks in outer parallel section.
28:           // Add wave-front angles to offsets for next time-step.
29:           O_x = O_x + A_x
30:           O_y = O_y + A_y
31:           O_z = O_z + A_z
32:         end for
33:         // End of one wave-front tile.
34:       end for
35:     end for
36:   end for
37: end for
38: // End of all wave-front tiles.
```

spatially: the velocity and stress grids are temporally offset, and the components of the velocity vector $(\upsilon_x, \upsilon_y, \upsilon_z)$ and stress tensor σ are spatially offset. This is illustrated in Fig. 11.

Due to the staggered grid, typically, first the velocity values throughout the problem domain are updated, and then the stress tensor values are updated using the newly-updated velocity values. The three velocity components, υ_x, υ_y, υ_z, are updated using 13-point stencils of radius 2, as depicted in Fig. 1(a). The three off-diagonal stress components, σ_{xy}, σ_{xz}, σ_{yz} are updated with 13-point stencils, while the diagonal components σ_{xx}, σ_{yy}, σ_{zz} are updated with 9-point stencils.

In addition to the nine 3D grids required to store the velocity and stress values throughout the problem domain, there are three 3D grids to store the density and material parameters of the problem region's underlying geology, one 3D grid used to implement absorbing boundary conditions, and thirteen additional 3D grids for the coarse-grained frequency-dependent anelastic attenuation. In total, this formulation requires twenty-six 3D grids.



(a) Staggered velocity components.

(b) Staggered stress components.

Fig. 11. The positions of the velocity υ and stress σ components in their respective staggered grids. σ_{**} denotes the diagonal stress components σ_{xx}, σ_{yy}, σ_{zz}. σ_{yz} is located at the center of grid, indicated by a gray color in the illustration.

4.2. Memory requirements

The stencil inputs to calculate a new value $(t+1, i, j, k)$ for any given velocity grid includes one point from that *same* grid at (t, i, j, k) and multiple points from various stress grids at time t. Since only one point is read from the grid being updated, and since that point has the same spatial index as the output, we only need to store one time-step value in memory for each velocity grid.

This is illustrated in Fig. 12 where all stress and velocity values for $t = 1$ have been calculated, and the velocity values for $t = 2$ are being calculated in ascending x indices (left-to-right). In this case, the memory used to store the $t = 1$ velocity values can be reused to store the $t = 2$ values. This can be generalized for any values of $t + 1$ and t. (The key in Fig. 12 applies to all diagrams in this section.)

Similarly, the stencil inputs to calculate a new value $(t+1, i, j, k)$ for any given stress grid includes one point from that same grid at (t, i, j, k) and multiple points from various velocity grids at time t. Thus, we only need to store one time-step value in memory for each stress grid as well.

4.3. Dependencies between stencils

We observe that any velocity-grid value $(t + 1, i, j, k)$ does not depend on any other value in that grid at $t + 1$ for all i, j, and k. Thus, multiple values at $t + 1$ within any given velocity grid may be calculated simultaneously. Furthermore, any velocity-grid value $(t+1, i, j, k)$ does not depend on any value in any *other* velocity grid at $t + 1$ for all i, j, and k. Thus, multiple values at $t + 1$ across all velocity grids may be calculated simultaneously. This also holds for stress-grid updates.

In contrast to the independence among the various velocity grids, each velocity-grid value $(t + 1, i, j, k)$ does depend on multiple stress-grid values at time-index t. Thus, all the required values in the stress grids at t must be calculated before the dependent velocity-grid values can be calculated. Similarly, all the required values in the velocity grids at $t + 1$ must be calculated before the dependent stress-grid values can be calculated.

This pattern repeats through time: given initial conditions in velocity and stress grids at $t = 0$, velocity-grid values at $t = 1$ depend on velocity and stress-grid values at $t = 0$, then stress-grid values at $t = 1$ depend on stress-grid values at $t = 0$ and velocity-grid values at $t = 1$, then velocity-grid values at $t = 2$ depend on velocity and stress-grid values at $t = 1$, and so forth. Therefore, the most straightforward approach is to calculate all values in all velocity grids at a given time $t + 1$, then all values in all stress grids at $t+1$, then all values in all velocity grids at $t+2$, etc. This approach is illustrated in Algorithm 8, which is analogous to Algorithm 3 but with multiple stencils. In this algorithm, $\upsilon(t, i, j, k)$ represents the application of *all* velocity stencils at the given point, and $\sigma(t, i, j, k)$ represents the application of all stress stencils.

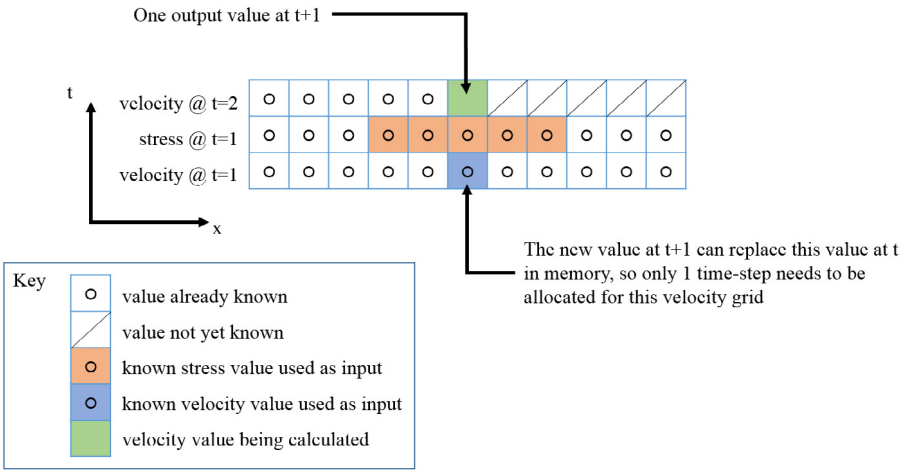


Fig. 12. Time–space diagram of one velocity and one stress grid for AWP (showing t and x dimensions only).

Algorithm 8 Conceptual AWP kernel.

```
1: // Problem of size D_t × D_x × D_y × D_z.
2: for t = 1 to D_t do
3:    // Update all velocity grids at t + 1 using velocity stencils υ.
4:    for all (i, j, k) ∈ [(1 to D_x) × (1 to D_y) × (1 to D_z)] do
5:       υ(t, i, j, k)
6:    end for
7:    // Update all stress grids at t + 1 using stress stencils σ.
8:    for all (i, j, k) ∈ [(1 to D_x) × (1 to D_y) × (1 to D_z)] do
9:       σ(t, i, j, k)
10:   end for
11: end for
```

4.4. SIMD and thread parallelization

As with the Iso3DFD example, it is convenient and efficient to update multiple values in any given grid using SIMD parallelization, and vector-folding is also applicable to this application to increase overlap, reducing bandwidth demands. Similarly, multiple spatial cache-blocks within a velocity grid can be updated using multiple software threads running in parallel on multiple hardware cores and threads. These techniques are illustrated in Fig. 13. In addition, nested OpenMP parallelism can be applied to improve cache locality using sub-blocks as was discussed for the Iso3DFD application. The same optimizations apply to the stress grids.

In short, all the optimizations from Sections 3.3 through 3.7 apply directly to AWP while calculating the velocity or stress updates at a given time-step. This is illustrated in Algorithm 9. Note that the details of the stress update are not shown; the loop structures and parallelism are exactly the same as those of the velocity update. In this algorithm, the size of blocks and sub-blocks are the same for both velocity and stress stencils, but this does not have to be the case. On the other hand, it is recommended that the shape of the folded vectors be the same across all grids because each stencil calculation accesses multiple grids, and the most efficient processing is obtained when all SIMD inputs share the same layout.

4.5. Temporal wave-front tiling

Application of temporal wave-front tiling to a staggered-grid problem is similar to that of the single-grid problem as described in Section 3.8, except that dependencies between grids need to be considered. The calculation of points in one velocity and one stress grid in the t and x dimensions for one tile is illustrated in Fig. 14. Here, we first calculate points for the velocity grid at $t = 1$ (dependencies on the initial conditions in the velocity and stress grids at $t = 0$ are not shown), and we stop at some x index determined by the tile size. Next, we may proceed with calculating values for the stress at $t = 1$ within this tile even though all the values for the velocity at $t = 1$ are not calculated, but we must be careful to stop at the x index for which the last dependencies are available. In the case of this AWP formulation, that limit is two (2) points before the x index where we stopped calculating velocity values. Thus, the wave-front angle for the stress calculations is two steps in the x dimension. It so happens in this application that the wave-front angles for all spatial dimensions in all velocity and stress grids are the same, but that is not generally the case. Algorithm 10 shows the final pseudo-code for the AWP application with temporal wave-front tiling.

Algorithm 9 Sequential update of dependent staggered grids with nested parallel cache-blocking and vector-folding within each.

```
1:  // Problem size D_t × D_x × D_y × D_z.
2:  for t = 1 to D_t do
3:     // Update all velocity grids at t + 1 using velocity stencils υ.
4:     // Blocks evaluated in parallel by Xeon Phi tiles.
5:     for all (i_1, j_1, k_1) ∈ [(1 to D_x by B_x) × (1 to D_y by B_y) × (1 to
       D_z by B_z)] do
6:        // Block of size B_x × B_y × B_z.
7:        // Sub-blocks evaluated in parallel by Xeon Phi hyper-threads.
8:        for all (i_0, j_0, k_0) ∈ [(i_1 to i_1 + B_x − 1 by S_x) × (j_1 to
          j_1 + B_y − 1 by S_y) × (k_1 to k_1 + B_z − 1 by S_z)] do
9:           // Sub-block of size S_x × S_y × S_z.
10:          for all (i, j, k) ∈ [(i_1 to i_1 + S_x − 1 by N_x) × (j_1 to
             j_1 + S_y − 1 by N_y) × (k_1 to k_1 + S_z − 1 by N_z)] do
11:             // Vector-folded SIMD of size N_x × N_y × N_z.
12:             υ(t, i to i + N_x − 1, j to j + N_y − 1, k to k + N_z − 1)
13:          end for
14:       end for
15:       // End of sub-blocks in nested parallel section.
16:    end for
17:    // End of velocity blocks in outer parallel section.

18:    // Repeat lines 3–17, applying stress stencils σ to stress grids.
19: end for
```

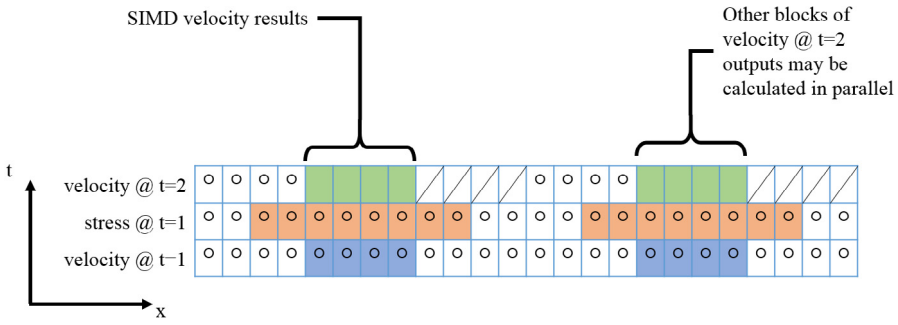


Fig. 13. Parallel update of values in multiple velocity blocks within a single time-step.

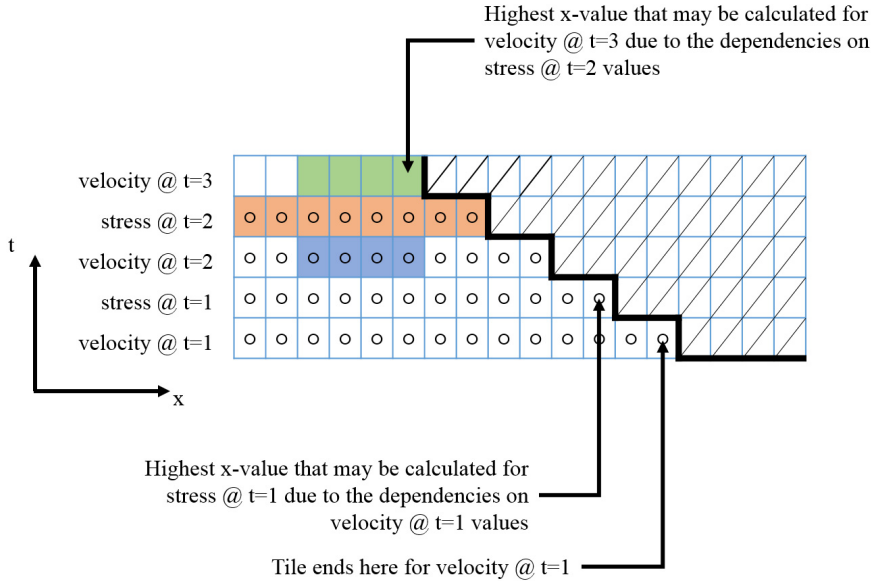


Fig. 14. Update of values in a wave-front tile of multiple grids.

The other observations made in Section 3.8 apply to AWP. In particular, the same hyper-trapezoid and hyper-parallelogram shapes will be created as subsequent tiles are calculated, and the memory requirements do not increase due to temporal tiling.

5. Experimental results for single-stencil application

In this section, we use the Iso3DFD problem from Section 3 that is available as an example stencil within the YASK software package [8] to evaluate tiling strategies on the Intel Xeon Phi 7250 processor.

5.1. Configuration

Table 1 summarizes the target system and the software stack that we used for this evaluation. In this Intel Xeon Phi system, the DDR memory capacity is six times that of the MCDRAM.

There are many options and parameters available in the YASK framework to tune the performance of any given stencil application. In this paper, we focus on the tuning and evaluation of the temporal wave-front tiling feature at various problem sizes and hold other settings constant for ease of comparison. The reader is referred to [8] for a discussion on the impacts of tuning options not detailed here, such as threading and cache-block sizes. The settings selected for the following experiments are shown in Table 2, most of which are for optimizations described in Section 3.

Table 1
Characteristics of the system used for the evaluation.

Processor model	Intel® Xeon Phi™ 7250
Nominal frequency	1.4 GHz
Number of tiles	34
Number of cores	68 (2per tile)
Hyper-threads per core	4
Mesh mode	Quadrant
DDR4 size	96 GiB
MCDRAM size	16 GiB
MCDRAM modes	flat and cache (see text)
Operating system	Red Hat Enterprise Linux Server release 7.2
C++ compiler	Intel® compiler version 17.0.2.174
Performance-data collector	Intel® VTune™ Amplifier XE 2016 Update 4

Table 2
YASK software settings for Iso3DFD.

Floating-point precision	single (32 bits)
SIMD width	16 elements (512 bits)
Problem size ($D_t \times D_x \times D_y \times D_z$)	Varies (see text)
Wave-front size ($W_t \times W_x \times W_y \times W_z$)	Varies (see text)
Block size ($B_x \times B_y \times B_z$)	96 × 96 × 96 (cubes)
Sub-block size ($S_x \times S_y \times S_z$)	1 × 96 × 96 (slabs)
Vector-folding ($N_x \times N_y \times N_z$)	4 × 4 × 1 (2D fold)
Scanning paths	Nested loops: *x* outer, then *y*, then *z* inner
OpenMP threads across blocks	34 (one per Xeon Phi tile)
OpenMP threads across sub-blocks	8 (2 cores ×4 hyper-threads per tile)

Algorithm 10 Temporal wave-front tiling for AWP.

```
1:  // Problem of size D_t × D_x × D_y × D_z.
2:  // Wave-front angles A_x, A_y, A_z for both velocity and stress stencils.
3:  // Wave-front tiles evaluated sequentially.
4:  for t_2 = 1 to D_t by W_t do
5:    for i_2 = 1 to D_x + (W_t × A_x) by W_x do
6:      for j_2 = 1 to D_y + (W_t × A_y) by W_y do
7:        for k_2 = 1 to D_z + (W_t × A_z) by W_z do
8:          // Initialize wave-front offsets for this tile.
9:          O_x = O_y = O_z = 0
10:         // Wave-front tile of size W_t × W_x × W_y × W_z.
11:         for t_1 = t_2 to t_2 + W_t − 1 do
12:           // Update all velocity grids at t + 1 using velocity stencils
              υ.
13:           // Blocks evaluated in parallel by Xeon Phi tiles.
14:           for all (i_1, j_1, k_1) ∈ [(i_2 − O_x to
              i_2 − O_x + W_x − 1 by B_x) × (j_2 − O_y to
              j_2 − O_y + W_y − 1 by B_y) × (k_2 − O_z to
              k_2 − O_z + W_z − 1 by B_z)] do
15:             // Block of size B_x × B_y × B_z.
16:             // Sub-blocks evaluated in parallel by Xeon Phi
                hyper-threads.
17:             for all (i_0, j_0, k_0) ∈ [(i_1 to i_1 + B_x − 1 by S_x) × (j_1 to
                j_1 + B_y − 1 by S_y) × (k_1 to k_1 + B_z − 1 by S_z)] do
18:               // Sub-block of size S_x × S_y × S_z.
19:               for all (i, j, k) ∈ [(i_1 to i_1 + S_x − 1 by N_x) × (j_1 to
                  j_1 + S_y − 1 by N_y) × (k_1 to k_1 + S_z − 1 by N_z)] do
20:                 if i ∈ [1, D_x] and j ∈ [1, D_y] and k ∈ [1, D_z] then
21:                   // Vector-folded SIMD of size N_x × N_y × N_z.
22:                   υ(t, i to i+N_x−1, j to j+N_y−1, k to k+N_z−1)
23:                 end if
24:               end for
25:             end for
26:             // End of sub-blocks in nested parallel section.
27:           end for
28:           // End of velocity blocks in outer parallel section.
29:           // Add wave-front angles to offsets for stress stencils.
30:           O_x = O_x + A_x
31:           O_y = O_y + A_y
32:           O_z = O_z + A_z
33:           // Repeat lines 12–28, applying stress stencils σ to stress
              grids.
34:           // Add wave-front angles to offsets for velocity stencils at
              next time-step.
35:           O_x = O_x + A_x
36:           O_y = O_y + A_y
37:           O_z = O_z + A_z
38:         end for
39:         // End of one wave-front tile.
40:       end for
41:     end for
42:   end for
43: end for
44: // End of all wave-front tiles.
```

The experimental process used to evaluate the effectiveness of temporal wave-front tiling is to run a series of trials across a variety of problem sizes ranging from those that fit easily within the MCDRAM size up to those that use most of the DDR size. For each problem size, we evaluate four different Xeon Phi memory configurations:

DDR The system MCDRAM is configured in flat mode, and the memory allocation policy is set to prefer DDR memory, only using MCDRAM if DDR is exhausted. No temporal wave-front tiling was used in this version.

MCDRAM-flat The system MCDRAM is configured in flat mode and the memory allocation policy is set to *prefer* the MCDRAM memory. This means that system will start allocating in MCDRAM, and when that is filled, it will start allocating in DDR. No temporal wave-front tiling was used in this version.

MCDRAM-cache without wave-front tiling The system MCDRAM is configured in cache mode, but no temporal wave-front tiling is used.

MCDRAM-cache with wave-front tiling The system MCDRAM is configured in cache mode, and temporal wave-front tiling is used.

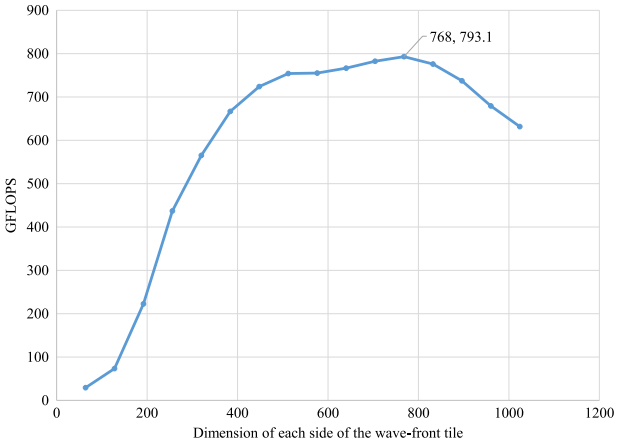


Fig. 15. Overall Iso3DFD performance obtained in MCDRAM-cache mode at various spatial wave-front tile sizes.

5.2. Tile-size selection

For the MCDRAM-cache configuration with wave-front tiling, in addition to the settings shown in Table 2, we must also select settings for the size of the temporal wave-front tiles. In order to select these settings, we run two sets of experiments in MCDRAM-cache mode: one to select the spatial size of the tiles, and one to select the temporal size.

To select a spatial size for the wave-front tiles, we run a series of trials with various tile sizes. We fix the problem size at 1920^3 points in x, y, and z. This requires a memory allocation of 81.2 GiB, using most of the DDR size on the test platform. Each trial consists of two runs of 30 time-steps each, and we record the performance of the better of the two runs. The number of time-steps within each wave-front tile (its temporal size) is set to the full length of each run, or 30 time-steps. The trials vary the tile sizes from 64^3 to 1024^3, all cube-shaped.[6] All other settings are those in Table 2. The results of this experiment are shown in Fig. 15. Performance is reported in GFLOPS (billion floating-point operations per second), which is a consistent and comparable unit of real work in this kernel because each grid update requires 61 floating-point operations. The best-performing tile size is 768^3, and we use that setting for the remaining experiments.

Recall that only two temporal copies of the pressure grid are needed, and there is no temporal dimension in the velocity grid. Since single-precision floating-point values are four bytes each, this implies the minimum footprint of a tile is $768^3 \times 3 \times 4 \approx 5.1$ GiB. Adding in the additional memory required to store the values in the halo brings this to approximately 5.4 GiB. The actual working footprint is slightly larger due to the shifting in the three spatial dimensions after each time step based on the wave-front angles

[6] We experimented with other tile shapes, but did not find any benefit to non-cube ones.

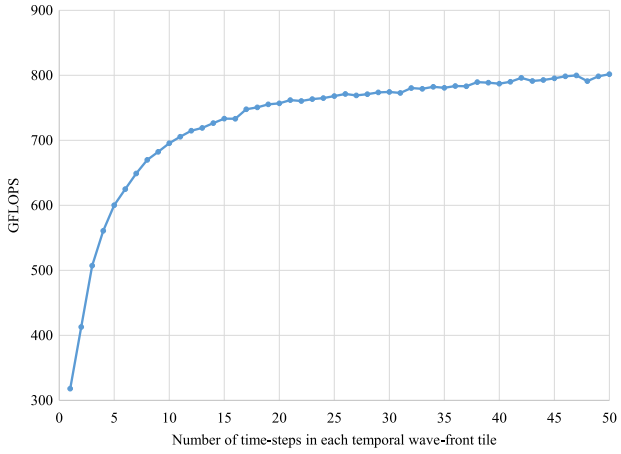


Fig. 16. Overall Iso3DFD performance obtained in MCDRAM-cache mode at various temporal sizes (number of time-steps) of temporal wave-front tiles.

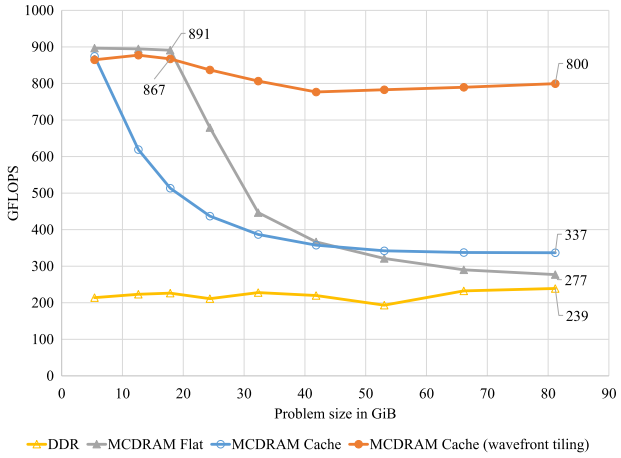


Fig. 17. Overall Iso3DFD performance obtained for the different configurations at various problem sizes.

described in Section 3.8. Because the MCDRAM cache is direct-mapped, the number of cache-tag collisions can grow rapidly as the working footprint approaches the MCDRAM size. Thus, a footprint significantly lower than the MCDRAM size will often provide fewer collisions and thus higher performance, as seen in this case. Optimal tile sizes will likely be different for other stencil equations and problem sizes.

To select a temporal size for the wave-front tiles, we run a series of trials with various numbers of time-steps within each wave-front. The problem size and spatial tile size are fixed as discussed above. All other settings are those in Table 2. The results of this experiment are shown in Fig. 16. As expected, the incremental performance benefit of adding more time-steps decreases asymptotically as the high cost of the initial time-step due to a high number of cache misses is amortized across the remaining time-steps with fewer misses. We elected to use 30 time-steps as the temporal size of the tiles in the remaining experiments because it was well-within the "flat" part of the curve and within 3% of the maximum observed.

After running these preparatory experiments to select the wave-front tile settings, we can now run the desired experiments to determine the impact of wave-front tiling.

5.3. Wave-front results and analysis

We run trials across a variety of cube-shaped problem sizes ranging from 768^3 to 1920^3 points. The smallest size uses approximately 5.4 GiB of memory, and the largest requires 81.2 GiB. Thus, we are evaluating problem sizes that fit well within the 16 GiB of the MCDRAM through those that are much larger than the MCDRAM capacity and approach the DDR capacity.

Fig. 17 shows the overall performance obtained by each of the four configurations across the different problem sizes. The DDR version obtains a relatively low performance—around 200 GFLOPS for all sizes. For the smallest size, all versions running out of MCDRAM perform very well—around 900 GFLOPS. As the size increases, first the version running in cache mode without the wave-front tiling starts to degrade, then the performance of the version running in flat mode starts to degrade. Both asymptotically approach the performance of DDR, which suggests that for large sizes, they are limited by the same bottleneck. Only the version running in cache mode that uses the temporal wave-front tiling is capable of performing well – around 800 GFLOPS – for all problem sizes.

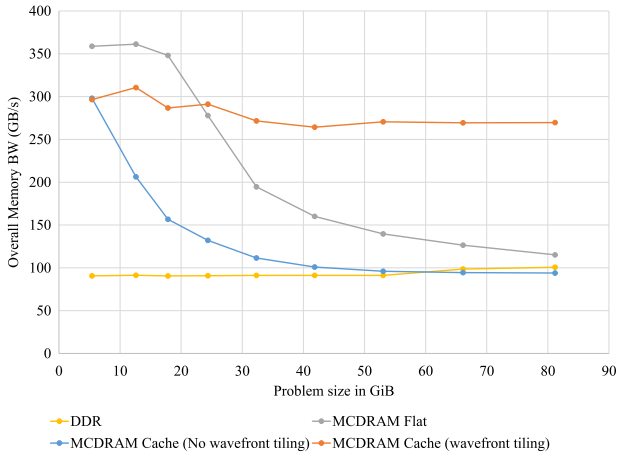


Fig. 18. Iso3DFD bandwidth provided by the memory subsystem for the different configurations.

To confirm our hypotheses regarding the relative performance between these configurations, we collect memory bandwidth and cache data using the Intel® VTune™ Amplifier XE performance-data collection tool.

Fig. 18 shows the total memory bandwidth (the combination of MCDRAM plus DDR bandwidth) that the different configurations obtain as we increase the grid size. We observe that the DDR version memory bandwidth is flat, as it is always limited by the bandwidth of the DDR memory. The other versions that run out of MCDRAM obtain high bandwidth for the small grid sizes, but as the grid size increases, their behavior changes. The version running in flat mode sustains the bandwidth until the amount of data overflows in the MCDRAM and the system starts allocating part of the grid in the DDR memory. As a larger part of the grid is in the DDR, the bandwidth that is obtained approaches the one obtained by the DDR version. The version that runs in cache mode without tiling degrades quickly, as it is unable to use the MCDRAM cache effectively. The version that runs in cache mode with the temporal wave-front tiling starts with a bandwidth slightly below that of the Flat version, but it is able to sustain that high bandwidth even as the grid size increases.

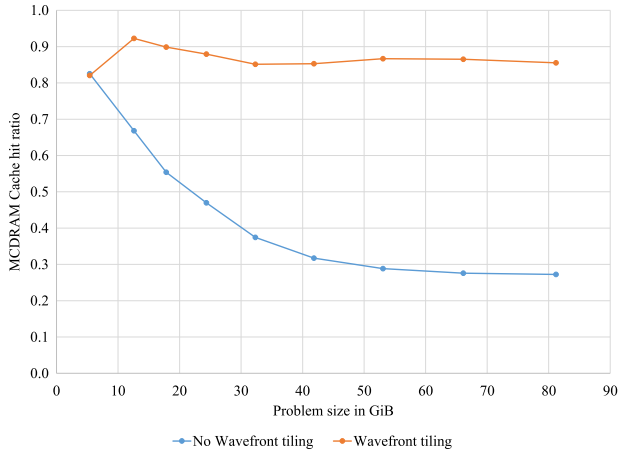


Fig. 19. Iso3DFD MCDRAM cache hit rates for the two MCDRAM-cache configurations.

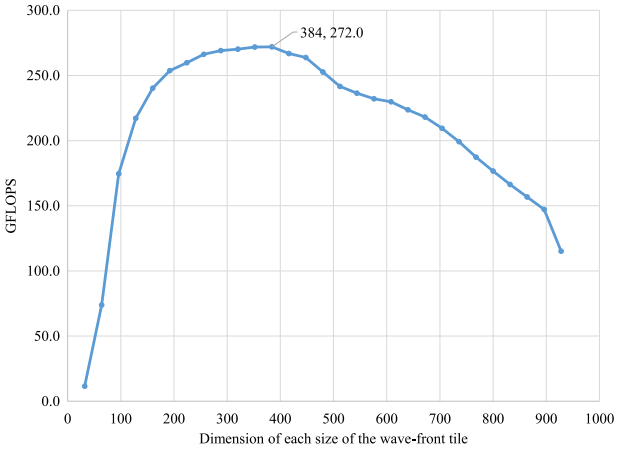


Fig. 20. Overall AWP performance obtained in MCDRAM-cache mode at various spatial wave-front tile sizes.

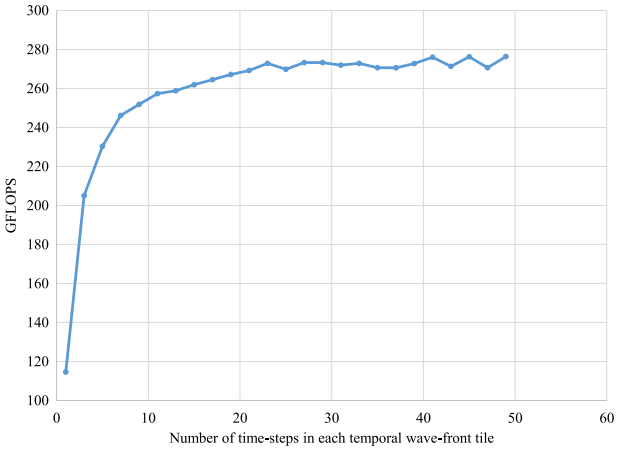


Fig. 21. Overall AWP performance obtained in MCDRAM-cache mode at various temporal sizes (number of time-steps) of temporal wave-front tiles.

To explain the difference in memory bandwidth between the versions running in cache mode, we need to look into the effectiveness of the MCDRAM cache. Fig. 19 shows the hit ratio of accesses to the MCDRAM cache. We observe that the version with the temporal wave-front tiling maintains a hit ratio close to 90% even for large grid sizes, whereas in the version without tiling, the effectiveness of the MCDRAM cache degrades quickly as we increase the grid size with a ratio below 30% for the largest grid size. The increase in reuse is what allows the MCDRAM to continue to provide a high bandwidth for the version with temporal wave-front tiling, which in turn translates into maintaining performance as the problem size increases beyond the MCDRAM capacity.

The data in Fig. 18 strongly correlates with that in Fig. 17, which suggests that the memory bandwidth is the most important factor to obtain performance for the Iso3DFD kernel. Similarly, the data in Fig. 19 correlates with the MCDRAM-cache data in Figs. 17 and 18. Our results indicate that applying locality optimizations at the MCDRAM-cache level similar to those used to improve performance at other caches in the memory hierarchy can enable effective use of the MCDRAM of the Xeon Phi processors in MCDRAM-cache mode. If such optimizations are not applied, the performance of bandwidth-limited workloads like stencil codes may instead trend toward the performance provided by DDR only as the memory footprint exceeds the MCDRAM capacity.

6. Experimental results for staggered-grid application

In this section, we use the AWP problem from Section 4 that is available as an example stencil within the YASK software package [8] to evaluate tiling strategies on the Intel Xeon Phi 7250 processor. Performance results not including the temporal-wavefront feature are discussed in [15]. Here, we focus on tuning the temporal-wavefront tiles.

6.1. Configuration

The experimental system is the same as the platform described in Section 5.1. The YASK software is configured for AWP as shown in Table 3. The four Xeon Phi memory configurations are also the same as those Section 5.1.

6.2. Tile-size selection

The process of selecting spatial and temporal sizes for the wave-front tiles is the same as described in Section 5.2 except as noted below.

For selecting the spatial tile size, we fix the problem size at 928^3 points in x, y, and z. This requires a memory allocation of 84.8 GiB, using most of the DDR size on the test platform as with Iso3DFD. The trials vary the tile sizes from 64^3 to 928^3, all cube-shaped. All other settings are those in Table 3. The results of this experiment are shown in Fig. 20. The best-performing tile size is 384^3, and we use that setting for the remaining experiments.

For selecting a temporal size for the wave-front tiles, we fix the problem size and spatial tile size as discussed above. The results of this experiment are shown in Fig. 21. As with Iso3DFD, we elected to use 30 time-steps as the temporal size of the tiles in the remaining experiments.

6.3. Wave-front results

We run trials across a variety of cube-shaped problem sizes ranging from 384^3 to 928^3 points. The smallest size uses approximately 6.2 GiB of memory, and the largest requires 84.8 GiB. Thus, we are evaluating problem sizes that fit well within the 16 GiB of the MCDRAM through those that are much larger than the MCDRAM capacity and approach the DDR capacity.

Fig. 22 shows the overall performance obtained by each of the four configurations across the different problem sizes. Note that the GFLOPS scores are comparable within data points on this figure,

Table 3
YASK software settings for AWP.

Floating-point precision	single (32 bits)
SIMD width	16 elements (512 bits)
Problem size ($D_t \times D_x \times D_y \times D_z$)	Varies (see text)
Wave-front size ($W_t \times W_x \times W_y \times W_z$)	Varies (see text)
Block size ($B_x \times B_y \times B_z$)	$32 \times 32 \times 32$ (cubes)
Sub-block size ($S_x \times S_y \times S_z$)	$1 \times 32 \times 32$ (slabs)
Vector-folding ($N_x \times N_y \times N_z$)	$4 \times 4 \times 1$ (2D fold)
Scanning paths	Nested loops: x outer, then y, then z inner
OpenMP threads across blocks	34 (one per Xeon Phi tile)
OpenMP threads across sub-blocks	8 (2 cores ×4 hyper-threads per tile)

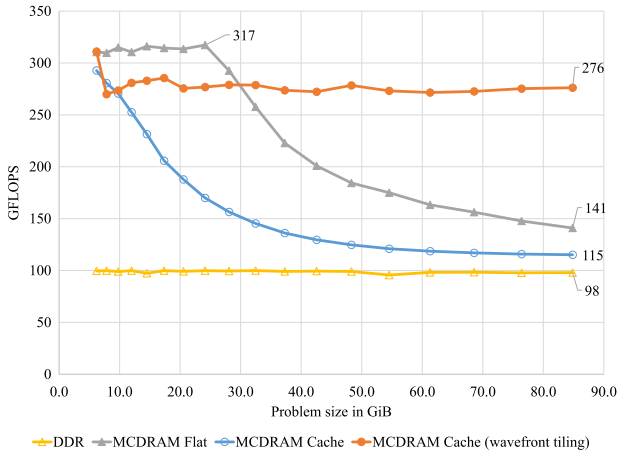


Fig. 22. Overall AWP performance obtained for the different configurations at various problem sizes.

but not to those in Fig. 17 because the arithmetic intensity of the AWP problem is less than that of Iso3DFD. The DDR version obtains a relatively low performance—around 100 GFLOPS for all sizes. For the smallest size, all versions running out of MCDRAM perform very well—around 300 GFLOPS. As the size increases, first the version running in cache mode without the wave-front tiling starts to degrade, then the performance of the version running in flat mode starts to degrade. Both asymptotically approach the performance of DDR, which suggests that for large sizes, they are limited by the same bottleneck. Only the version running in cache mode that uses the temporal wave-front tiling is capable of performing well – around 270 GFLOPS – for all problem sizes after the smallest one.

7. Related work

Reduction of memory accesses is especially advantageous in multi-core and hyper-threaded architectures due to the additional pressure on shared caches, memory load and store units, etc. Significant work has been done on reducing the large memory-bandwidth demand often observed in stencil algorithms. Much of this work has focused on developing and/or analyzing tiling strategies at one or more cache levels. Kamil et al. [16] reported on several implicit cache-oblivious approaches and a cache-aware algorithm on cache-based architectures and compared them to cache-aware computations on an explicitly-managed memory hierarchy architecture. Frigo et al. [17] presented and evaluated a cache-oblivious algorithm for stencil computations. Datta et al. [18] discussed a number of optimization strategies and an auto-tuning environment to search over the space of the techniques and their parameters for high-performance solutions. Dursun et al. [19] described a multi-level parallelization framework combining multi-node, multi-thread, and SIMD (single-instruction, multiple data) parallelization. Peng et al. [20] also presented a multi-level framework, focusing on spatial decomposition, critical-section-free multithreading, and SIMD. Holewinski et al. [21] described a code generation scheme for stencil computations on GPU accelerators, which optimizes the code by trading an increase in the computational workload for a decrease in the required global memory bandwidth. Renganarayanan et al. [22] detailed an approach for parameterized tiled code generation. Bandishti et al. [23] described a tiling technique that ensures concurrent start-up as well as perfect load-balance whenever possible.

Temporal tiling extends the notion of spatial blocking to perform more than one time-step calculation within a spatial subset of the problem domain. Lamport [9] introduced the general concept of hyperplane or wave-front parallelization of dependent loops in 1974. Ramanujam and Sadayappan [10] presented a method of aggregating a number of loop iterations into tiles where the tiles execute atomically. Wonnacott [11] introduced the notion of time-skewing as applied to stencil operations. Strzodka et al. [24] presented a time-skewing algorithm for breaking the so-called memory wall for certain iterative stencil computations. Nguyen et al. [25] presented a "3.5D"-blocking algorithm (2.5D-spatial and temporal blocking) that reduces memory bandwidth usage and reduces overhead due to blocking. Malas et al. [12] combined the ideas of multicore wave-front temporal blocking and diamond tiling to arrive at stencil update schemes that show reductions in memory pressure compared to existing approaches.

In contrast to these related studies, this work extends the notion of temporal tiling to an additional level of hierarchy above the traditional CPU-cache blocks to specifically target very large shared caches. This article builds upon the results presented in [26] by providing additional analysis and discussion on the selection of temporal-tiling parameters and extending the application of temporal tiling from a simple single-stencil problem to a staggered-grid model requiring several stencils and updated grids.

8. Conclusions

This paper described an implementation of two stencil-based numerical simulations: a high-order isotropic 3D finite-difference (Iso3DFD) stencil, and a staggered-grid seismic model (AWP), both using a variety of optimizations. The optimizations targeted a hierarchy of tiling strategies: SIMD and vector-folding to increase L1-cache and processor ALU usage, spatial cache-blocking to leverage multi-core parallelism and L2-cache locality, and temporal wave-front tiling to expose the benefit of large shared caches. Experiments were run on an Intel® Xeon Phi™ 7250 processor (code-named Knights Landing), which implements a large shared cache with MCDRAM technology. The MCDRAM can be configured in "flat" and "cache" modes and has one sixth the capacity the DDR provided in the system under test. Performance results were compared across problem sizes ranging from those smaller than the MCDRAM to those approaching the DDR capacity. For the Iso3DFD problem, temporal wave-front tiling provided a 2.4x speedup compared to using the MCDRAM cache without temporal tiling and a 3.3x speedup compared to only using DDR at the largest problem size. Measurements of MCDRAM+DDR bandwidth and MCDRAM-cache hit rates were presented and shown to correlate

with performance. For the AWP problem, temporal wave-front tiling provided a 1.9x speedup compared to using the MCDRAM cache without temporal tiling and a 2.8x speedup compared to only using DDR at the largest problem size.

8.1. Code availability

An implementation of all the techniques described in this article is available in a software package known as YASK (Yet Another Stencil Kernel) under the MIT open-source license at https://github.com/intel/yask. See https://01.org/yask for related announcements and downloads. More details of the features found in the software framework may be found in [8].

8.2. Future work

Near-term plans include the development of a well-documented API (application programmer interface) for YASK to facilitate integration of the generated code into production software and integrating YASK as a back-end into the Devito symbolic finite-difference computation package [27,28]. The latter will provide a mechanism for programmers to directly input the differential equations that describe the physical system being modeled; these will be automatically transformed into high-performance finite-difference kernels using the techniques discussed in this article.

References

[1] R.J. LeVeque, Finite Difference Methods for Ordinary and Partial Differential Equations, SIAM, 2007.

[2] C. Yount, Vector folding: Improving stencil performance via multi-dimensional SIMD-vector Representation, in: Proceedings of the IEEE 17th International Conference on High Performance Computing and Communications, HPCC, 2015, pp. 865–870 http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.27.

[3] Intel Corp. product specifications, http://ark.intel.com.

[4] Intel corp. software, https://software.intel.com.

[5] Optimizing memory bandwidth in knights landing on stream triad, https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-in-knights-landing-on-stream-triad.

[6] C. Andreolli, Eight optimizations for 3-dimensional finite difference (3DFD) code, https://software.intel.com/en-us/articles/eight-optimizations-for-3-dimensional-finite-difference-3dfd-code-with-an-isotropic-iso..

[7] C. Andreolli, P. Thierry, L. Borges, G. Skinner, C. Yount, Characterization and optimization methodology applied to stencil computations, in: J. Reinders, J. Jeffers (Eds.), High Performance Parallelism Pearls, first ed., Elsevier, 2016.

[8] C. Yount, J. Tobin, A. Breuer, A. Duran, YASK–Yet another stencil kernel: A framework for HPC stencil code-generation and tuning, in: Proceedings of the 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing held as part of ACM/IEEE Supercomputing 2016, (SC16), WOLFHPC'16, 2016, http://dx.doi.org/10.1109/WOLFHPC.2016.08.

[9] L. Lamport, The parallel execution of do loops, Commun. ACM 17 (2) (1974) 83–93. http://dx.doi.org/10.1145/360827.360844.

[10] J. Ramanujam, P. Sadayappan, Tiling multidimensional iteration spaces for nonshared memory machines, in: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, ACM, New York, NY, USA, 1991, pp. 111–120. http://dx.doi.org/10.1145/125826.125893.

[11] D. Wonnacott, Achieving scalable locality with time skewing, Int. J. Parallel Program. 30 (3) (2002) 181–221. http://dx.doi.org/10.1023/A:1015460304860.

[12] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, D. Keyes, SIAM J. Sci. Comput. 37 (4) (2015) C439–C464.

[13] Y. Cui, K.B. Olsen, T.H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D.K. Panda, A. Chourasia, et al., Scalable earthquake simulation on petascale supercomputers, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010, pp. 1–20.

[14] S. Day, C. Bradley, Memory-efficient simulation of anelastic wave propagation, Bull. Seismol. Soc. Amer. 91 (3) (2001) 520–531.

[15] J. Tobin, A. Breuer, A. Heinecke, C. Yount, Y. Cui, Accelerating seismic simulations using the intel xeon phi knights landing processor, in: Proceedings of ISC High Performance 2017, ISC17, 2017.

[16] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, K. Yelick, Implicit and explicit optimizations for stencil computations, in: Proceedings of the 2006 Workshop on Memory System Performance and Correctness, in: MSPC '06, ACM, New York, NY, USA, 2006, pp. 51–60. http://dx.doi.org/10.1145/1178597.1178605.

[17] M. Frigo, V. Strumpen, The memory behavior of cache oblivious stencil computations, J. Supercomput. 39 (2) (2007) 93–112. http://dx.doi.org/10.1007/s11227-007-0111-y.

[18] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 4:1–4:12.

[19] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, R.K. Kalia, A. Nakano, P. Vashishta, A multilevel parallelization framework for high-order stencil computations, in: Proceedings of the 15th International Euro-Par Conference on Parallel Processing, Euro-Par '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 642–653.

[20] L. Peng, R. Seymour, K.-i. Nomura, R.K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. Volz, C. Wong, High-order stencil computations on multicore clusters, in: Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on, 2009, pp. 1–11, http://dx.doi.org/10.1109/IPDPS.2009.5161011.

[21] J. Holewinski, L.-N. Pouchet, P. Sadayappan, High-performance code generation for stencil computations on GPU architectures, in: Proceedings of the 26th ACM International Conference on Supercomputing, ICS '12, ACM, New York, NY, USA, 2012, pp. 311–320. http://dx.doi.org/10.1145/2304576.2304619.

[22] L. Renganarayanan, D. Kim, M.M. Strout, S. Rajopadhye, Parameterized loop tiling, ACM Trans. Program. Lang. Syst. 34 (1) (2012) 3:1–3:41. http://dx.doi.org/10.1145/2160910.2160912.

[23] V. Bandishti, I. Pananilath, U. Bondhugula, Tiling stencil computations to maximize parallelism, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 40:1–40:11.

[24] R. Strzodka, M. Shaheen, D. Pajak, H.P. Seidel, Cache accurate time skewing in iterative stencil computations, in: 2011 International Conference on Parallel Processing, 2011, pp. 571–581.

[25] A. Nguyen, N. Satish, J. Chhugani, C. Kim, P. Dubey, 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–13. http://dx.doi.org/10.1109/SC.2010.2.

[26] C. Yount, A. Duran, Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling, in: Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems held as part of ACM/IEEE Supercomputing 2016, SC16, PMBS'16, 2016.

[27] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, G. Gorman, Devito: towards a generic finite difference dsl Using Symbolic Python, in: 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), 2016, pp. 67–75.

[28] M. Lange, N. Kukreja, F. Luporini, M. Louboutin, C. Yount, J. Hückelheim, G. Gorman, Optimised finite difference computation from symbolic equations, in: Python in Science Conference Proceedings, 2017, pp. 89–96 (SciPy, Texas). http://conference.scipy.org/proceedings/scipy2017/michael_lange.html.

Charles Yount received his Ph.D. degree from the Department of Electrical and Computer Engineering at Carnegie Mellon University. He is currently a Principal Engineer in the Software and Services Group at Intel Corporation. His work includes developing analysis and optimization techniques for HPC applications on many-core products including the YASK open-source software framework for stencil-code optimization.

Alejandro Duran received his Ph.D. degree from Universitat Politècnica de Catalunya and was a Senior Researcher at the Barcelona Supercomputing Center. He is currently an Application Engineer in the Data Center Group at Intel Corporation Iberia. His work is focused on support for parallel computing including OpenMP and scalable operating systems.

Josh Tobin received his Ph.D. from the Department of Mathematics at the University of California at San Diego, where he was additionally a member of the High Performance Geocomputing group at the San Diego Supercomputer Center. His research centers around spectral graph theory and its applications.