Solving Large Quantities of Small Matrix Problems on Cache-Coherent Many-Core SIMD Architectures

Bryce Adelstein Lelbach, Hans Johansen, and Samuel Williams
Lawrence Berkeley National Laboratory, Computational Research Division
{balelbach, hjohansen, swwilliams}@lbl.gov

***Abstract*—A number of computational science algorithms lead to discretizations that require a large number of independent small matrix solves. Examples include small non-linear coupled chemistry and flow systems, one-dimensional subsystems in climate and diffusion simulations and semi-implicit time integrators, among others. We introduce an approach for solving large quantities of independent matrix problems on cache-coherent many-core SIMD architectures. Unlike many vectorized or batched approaches that rely on reusing the matrix factorization across multiple solves, our algorithm supports sets of matrices that are different (due to spatial variation or non-linear solvers, for example). We present an optimized implementation of our solver for diagonally-dominant tridiagonal systems that uses only compiler directives, tiling, data layout and memory access patterns. Performance is evaluated on three Intel microarchitectures with different cache, vectorization, and threading features: Intel Ivy Bridge, Haswell, and Knight's Landing. Finally, we show that our solver improves on existing approaches and achieves ~90% of STREAM Triad effective bandwidth on all three target platforms.**

I. Introduction

One important class of problem in computational science is solving smaller-dimensional matrix subproblems that are duplicated across many degrees of freedom in a larger two (or more) dimension computation. Such problems appear in pointwise chemistry systems in the context of larger flow simulations (found in cloud microphysics [1] and combustion [2] models), one-dimensional systems that represent a numerically stiff direction for a physical phenomenon (found in atmospheric radiation [3], groundwater penetration [4] and cloud convection [5] models) and implicit solvers that need to couple these kinds of subsystems (such as semi-implicit time integrators [6]).

In most cases, these matrices are relatively small (ranging from $O(30-100)$ chemistry components or **vertical levels** in a climate application), may be sparse or dense and must be solved repeatedly with different entries each time to advance the overall simulation.

Thus, because these are often non-linear matrix systems with space- and time-dependent entries, these applications may not use a "factor once, solve many times" approach that seeks to prevent amortizing setup and factorization costs across multiple right-hand sides. This approach, for example, is used by LAPACK routines such as `dptsv()`, `dtsvb()`, `dgtsv()` and `dgttrs()` [7]. In our case, it is usually sub-optimal on SIMD many-core and GPGPU architectures to simply call an optimized linear algebra library, such as Intel's Math Kernel Library (MKL) [7] and NVIDIA's CUDA BLAS library (cuBLAS) [8]. These libraries may not achieve peak performance for batch solves of small matrices because they are not designed to simultaneously solve multiple systems and right-hand sides and take advantage of the data locality, memory access patterns and vectorization opportunities exposed by such interleaving.

We have developed a model problem that mimics the conditions encountered in these kinds of large-scale simulations. Key aspects of the test problem include:

- A 3D Cartesian grid ((i,j,k) indices), where different matrix systems in the **vertical dimension** k are generated at each **horizontal coordinate** (i,j) and the extent of k is $O(30-100)$.
- Each matrix is tridiagonal, diagonally-dominant and must be solved for all values in the k dimension.
- The matrix is derived from a finite difference discretization for the 1D diffusion equation, which allows it to be solved **without pivoting**.

In §III, we describe the Simultaneous Streaming Thomas Algorithm (SSTA), which is designed to solve the class of problem described above. The algorithm computes the solution to multiple systems **simultaneously**, facilitating the use of vector instructions. Additionally, SSTA iterates through memory in unit stride to enable software and hardware prefetching facilities to easily track data streams. SSTA supports different data layouts and tiling schemes, and we demonstrate how they can significantly influence performance. We compare performance against STREAM Triad [9] and a baseline solver derived from a production climate code which utilizes Intel MKL. Results presented in §V show that SSTA attains ~90% of STREAM Triad effective bandwidth on all three of our test platforms and provides a ~12*x* speedup over the MKL solver on our Intel Xeon Phi Knight's Landing platform.

II. Related Work

Approaches to solving large numbers of small matrices have been developed before in a variety of contexts. Libraries like Blaze [10] and the Numerical Template Toolkit (nt2) [11] are intended to support SIMD and/or SIMT vectorization for standard vector sizes as well as batched computation for sparse and dense matrices. Many implementations only support batched solves where the same matrix is applied to multiple right hand sides, such as Intel's MKL [7] or NVIDIA's cuBLAS [8]. As we demonstrate, this approach fails to expose vector parallelism opportunities which can be exploited on SIMD and SIMT architectures. Pipelined and batched versions of the Thomas algorithm [12], LU factorization [13] and similar algorithms which interleave the solution of multiple matrices have been developed, but most target SIMT and not SIMD architectures. There are few approaches that are designed to **simultaneously solve** many small problems which do not share the same matrix, support different data layouts and tiling schemes, and target **both** many-core SIMD and GPGPU SIMT architectures.

III. Implementation

Suppose we have a $nk \times nk$ **diagonally-dominant** tridiagonal matrix A and two nk element vectors u^s and u^{s+1}. We wish to solve $Au^{s+1} = u^s$ for u^{s+1}:

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & \dots & \\ & & \dots & \dots & c_{nk-2} \\ 0 & & & a_{nk-1} & b_{nk-1} \end{bmatrix} \begin{bmatrix} u_0^{s+1} \\ u_1^{s+1} \\ \dots \\ \dots \\ u_{n-1}^{s+1} \end{bmatrix} = \begin{bmatrix} u_0^s \\ u_1^s \\ \dots \\ \dots \\ u_{n-1}^s \end{bmatrix}$$

Our matrix A is stored as a set of three vectors: an $nk-1$ element sub-diagonal vector a, an nk element diagonal vector b and an $nk-1$ element super-diagonal vector c.

A. The Thomas Algorithm

We can use a simplified form of Gaussian elimination that does not require pivoting, known as the Thomas algorithm or the tridiagonal matrix algorithm (TDMA) [14], to solve the type of system described in the previous section. The Thomas algorithm takes advantage of sparsity to be $O(nk)$ in time, and can be extended to banded matrices and LU factorizations. It is also a significant improvement over dense Gaussian elimination, which is $O(nk^3)$ in time. We use a formulation of the Thomas algorithm which does not require any storage for temporary values, but overwrites the b vector and solves for u^{s+1} in place, overwriting u^s.

The Thomas algorithm consists of two passes. First, a forward pass is performed to eliminate the a_i elements:

```
for (auto k = 1; k < nk; ++k) {
  auto const m = a[k] / b[k - 1];
  b[k] -= m * c[k - 1];
  u[k] -= m * u[k - 1];
}
```

Then, an abbreviated form of back substitution is performed to obtain the solution:

```
u[nk - 1] = u[nk - 1] / b[nk - 1];

for (auto k = nk - 2; k >= 0; --k)
  u[k] = (u[k] - c[k] * u[k + 1]) / b[k];
```

The Thomas algorithm has very low algorithmic intensity (AI) [15]. During the course of our work, we developed a **theoretical peak performance model** for the Thomas algorithm.

First, we count the number of FLOPs performed. We will consider multiplications, additions and division operations as FLOPs. Each iteration of the forward elimination loop contains 1 division, 2 multiplications and 2 subtractions. This gives us a total of either $3(nk-1)$ FLOPs on fused-multiply-add (FMA) architectures [16] or $5(nk-1)$ FLOPs on non-FMA architectures. Next, the pre-substitution operation (the assignment to `u[nk - 1]` in the above snippet) performs a single division. Finally, the back substitution loop performs 1 multiplication, 1 subtraction and 1 division, adding either $2(nk-1)$ FLOPs (FMA architectures) or $3(nk-1)$ FLOPs (non-FMA architectures). In total, this gives us either $5nk-4$ FLOPs (FMA) or $8nk-7$ FLOPs (non-FMA) for the entire Thomas algorithm.

Our data movement model for the Thomas algorithm assumes that we will achieve optimal performance when data is cached between the forward elimination and back substitution loop. That is, we assume that accesses to a, b, c and u will **not** go to main memory in the back substitution loop because the arrays still reside in the cache hierarchy from the forward elimination loop accesses.

The Thomas algorithm accesses four arrays, reading from all of them (a, b, c and u) and writing to two of them (b and u). b and u have extent nk, so we store $2nk$ elements. a and c have extent $nk-1$, so we load $2(nk-1)+2nk$ elements. In total, $6nk-2$ elements are moved. Assuming double precision, $48nk-16$ bytes are moved to and from main memory during execution of the Thomas algorithm.

This gives us a FLOPs/byte ratio of $(5nk-4)/(48nk-16)$ (FMA) or $(8nk-7)/(48nk-16)$ (non-FMA). The lower bound for the Thomas algorithm's arithmetic intensity is $1/32$ FLOPs/byte when $nk=1$. The upper bound is $5/48$ FLOPS/byte (FMA) or $1/6$ FLOPs/byte (non-FMA) as nk approaches infinity. Based on this analysis, we can conclude that the Thomas algorithm is **memory-bandwidth bound**. Thus, we use effective bandwidth, not FLOPs, as our performance metric.

We verified our analytic model by measuring hardware performance counters which track memory traffic with the Intel VTune Amplifier XE profiler [17]. We found that memory bandwidth measured via hardware counters generally agreed with our model.

B. Batching and Vectorization

In the applications described in §I we need to apply the Thomas algorithm to each vertical column in a $ni \times nj \times nk$ 3D Cartesian grid, where i and j are **horizontal dimensions** and k is the **vertical dimension**. These computations are known as the **vertical solves**. Since each vertical solve is independent of the others, this is an embarrassingly parallel problem. The matrix coefficients for each column depend on the problem state, so a unique matrix for each column needs to be constructed before each solve. There are two different approaches to computing these batch solves.

The most straightforward approach is **solve** each column **independently** (the **independent solve strategy**; Figure 1). An $nk \times nk$ tridiagonal matrix is constructed for each column, and then the Thomas algorithm is used to solve the system formed by the matrix and the vertical column. Because the vertical solves are independent, they can be executed concurrently via task-level parallelism. Vectorization of the nk loop is not possible as each iteration of the loops in the Thomas algorithm is dependent on the previous iteration (e.g. loop-carried dependency) [12].

The other approach is to **simultaneously solve** multiple columns (the **simultaneous solve strategy**; Figure 2). A block $nk \times nk$ tridiagonal matrix is constructed for the whole grid; each block contained within the matrix is an $ni \times nj$ horizontal plane (e.g. the block matrix is a 4D space). To perform the vertical solves, we view the entire 3D Cartesian

Figure 1: Independent Solve Strategy: A vertical column of *nk* elements is extracted from a $ni \times nj \times nk$ 3D Cartesian grid and used as the right-hand side in a tridiagonal linear system. The system for each column is solved independently from other columns. This approach exposes task parallelism, but vectorization is not possible in the vertical dimension k due to its small extent and loop-carried dependencies present in the Thomas algorithm [12]. This strategy is used by our MKL baseline solver.

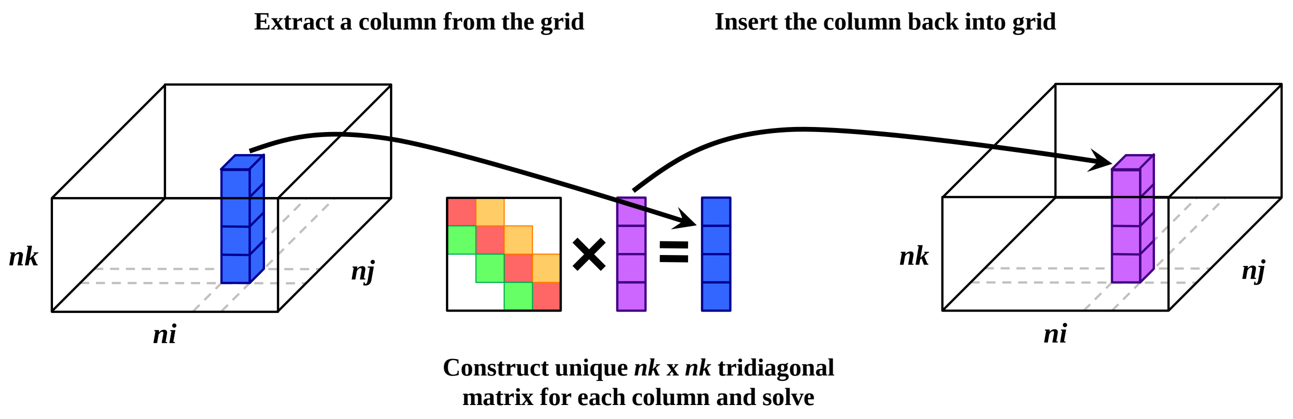
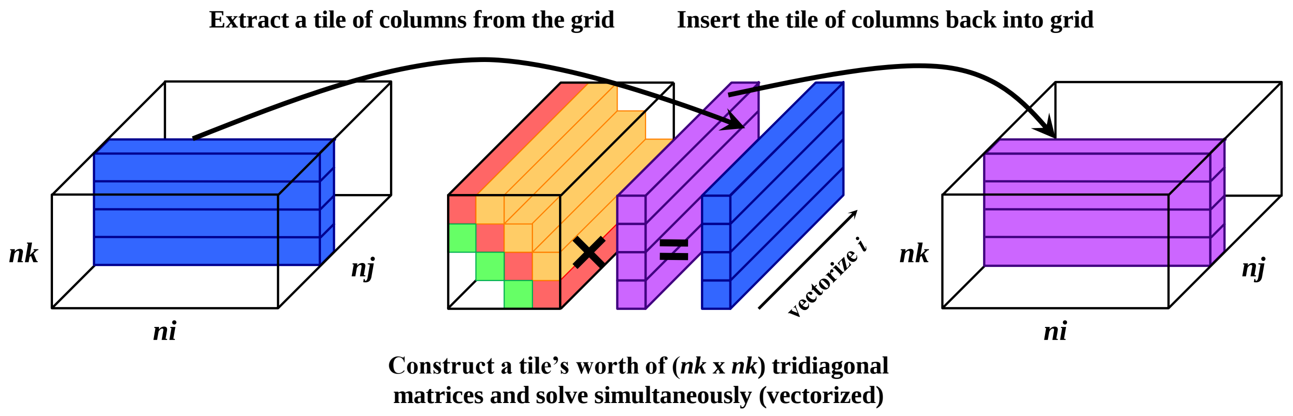

Extract a column from the grid
Insert the column back into grid
nk
nj
ni
nk
nj
ni
Construct unique nk x nk tridiagonal matrix for each column and solve

Figure 2: Simultaneous Solve Strategy: A tile of vertical columns, each containing *nk* elements, is extracted from a $ni \times nj \times nk$ 3D Cartesian grid. All the columns in the tile are solved simultaneously, interleaving the computation of individual solves. This approach exposes task parallelism, exhibits good data locality and enables vectorization in one of the horizontal dimensions (i). SSTA uses this strategy.


Extract a tile of columns from the grid
Insert the tile of columns back into grid
nk
nj
ni
vectorize i
nk
nj
ni
Construct a tile's worth of (nk x nk) tridiagonal matrices and solve simultaneously (vectorized)

grid as an *nk* block vector of $ni \times nj$ planes and apply it to the block matrix. Each step of the algorithm is applied across an entire plane. The forward sweep becomes:

```
for (auto k = 1; k < nk; ++k)
 for (auto j = 0; j < nj; ++j)
  for (auto i = 0; i < ni; ++i) {
    auto const m = a[i][j][k]
                 / b[i][j][k - 1];
    b[i][j][k] -= m * c[i][j][k - 1];
    u[i][j][k] -= m * u[i][j][k - 1];
  }
```

This approach facilities both task-level parallelism and vectorization. The grid and block matrix can be tiled into smaller subgrids, and the Thomas algorithm can be applied to each subgrid independently. Each step of the Thomas algorithm can be vectorized across the $ni \times nj$ horizontal plane that it is operating on: e.g. the i loop in the above snippet can be vectorized.

Our MKL baseline solver uses the independent solve approach, while SSTA uses the simultaneous solve strategy. The vector parallelism exposed by the simultaneous approach offers a major benefit over the independent approach. It is necessary to vectorize even for a memory-bandwidth bound problem like the Thomas algorithm, as peak memory bandwidth cannot be achieved without exploiting wider vector loads and stores. Such vectorization is not possible with most of the loops in the independent approach due to data dependencies between iterations of the Thomas algorithm [12]. Even if it was possible to vectorize in the vertical dimension k, it would still be undesirable to do so. The extent of the vertical dimension tends to be very small in the applications we are concerned with ($O(30-100)$), so loop and vectorization overheads would be hard to amortize.

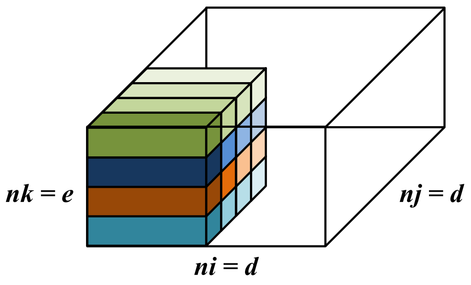
C. Data Layout

The data layout of the Cartesian grid has a huge impact on the performance of our solver. Throughout the course of our research, our understanding of the impact of different data layouts has evolved substantially. We have investigated three different schemes: kji, ijk and kji (Table I).

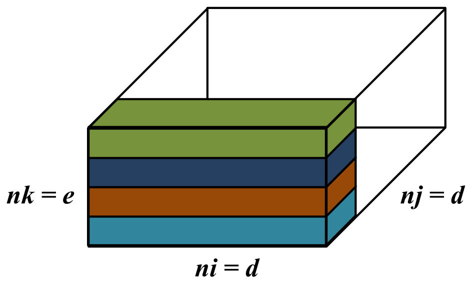
The production code that our MKL baseline solver is derived from uses the ijk-layout. The vertical columns that

Figure 3: Example of Different Layouts and Tiling Schemes: The SSTA solver supports four layout and tiling scheme combinations. The figures below show how the different combinatons would partition an $d \times d \times e$ grid into four tiles each containing $(d^2/4)(e)$ elements (one tile is shown). With the tile-ij scheme we get $d/2 \times d/2 \times e$ tiles, and with the tile-i scheme we get $d \times d/4 \times e$ tiles. Different colors indicate different contiguous regions within the tile. Regions with similar colors have greater locality with each other.

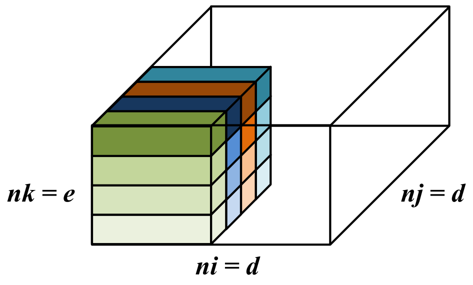
(a) ijk**-Layout with the Tile-**ij **Scheme:** $(d/2)(e)$ contiguous i-rows each containing $d/2$ elements. This layout and tiling combination exhibits the worst contiguity out of the the four different options.



(b) ijk**-Layout with the Tile-**j **Scheme:** e contiguous ij-planes, each containing $d^2/4$ elements. This layout and tiling combination exhibits the best contiguity for the ijk-layout.



(c) ikj**-Layout with the Tile-**ij **Scheme:** $(d/2)(e)$ contiguous i-rows each containing $d/2$ elements. This layout and tiling combination exhibits the worst contiguity for the ikj-layout.



(d) ikj**-Layout with the Tile-**j **Scheme:** One contiguous region containing $(d^2/4)(e)$ elements. This layout and tiling combination exhibits the greatest contiguity out of the four different options, and provides the best performance on Knight's Landing.

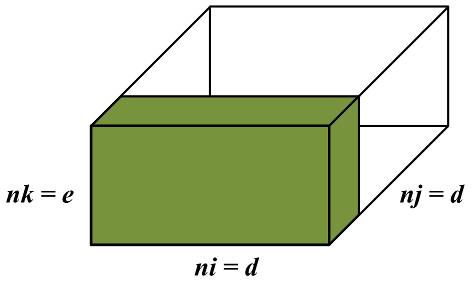


Table I: **Data Layouts for the 3D Cartesian Grid**

Name	i**-stride**	j**-stride**	k**-stride**
	(Horizontal)		**(Vertical)**
ijk, Column-Major	1	ni	$(ni)(nj)$
kji, Row-Major	$(nj)(nk)$	nk	1
ikj	1	$(ni)(nz)$	ni

we need to pass in to MKL as the right-hand side (u) are non-contiguous, as the vertical dimension k is the dimension with greatest stride. Currently, the production codebase allocates a temporary gather buffer, copies data from the grid into the buffer, calls the LAPACK solver, and then copies the result from the buffer back to the grid.

We have found the kji-layout to be optimal for our MKL baseline solver. With the kji-layout, we have contiguous vertical columns which can be passed to MKL without temporary allocations or unnecessary copies. We observed a noticeable performance increase from this optimization with very little source code change, although we have not studied the impact of this layout change on other components in more complex, multi-kernel applications.

Our SSTA solver uses either the ijk- or ikj-layout. SSTA vectorizes in one of the horizontal directions, so it is desirable to use layouts where the horizontal dimension that we are vectorizing (i) is the unit stride dimension, avoiding strided vector loads. The ikj-layout arose as an optimization to combat data locality and prefetching issues which hindered performance on the Intel Xeon Phi Knight's Landing architecture, and is described in greater detail in the following section and in §V.

D. Tiling

To ensure good CPU cache utilization, it is often necessary to break a larger grid into smaller **tiles** which better amortize nested loop overheads, expose greater data locality and keep data resident in a particular level of the cache hierarchy [18]. This technique is also known as **cache blocking**. Additionally, partitioning a grid with dimensions known only at application initialization into fixed size tiles can provide some of the performance benefits of compile-time-fixed dimensions (via compiler optimizations) with the flexibility of a dynamically sized problem [19]. Tiling also provides a useful abstraction for parallel work distribution, as each tile can be computed independent of other tiles and has no overlapping data accesses. Tiling is particularly important for memory-bandwidth bound computations like tridiagonal matrix solves.

The production code which our MKL baseline solver is derived from is not task parallelized or explicitly tiled. Since each column is solved independently, there is little opportunity to improve locality and reuse via tiling (shown in §V). Conceptually, since each column is solved independently in the production code, it is **effectively** tiled, albeit with a very small tile size (a single column, e.g. $O(30-100)$). Our MKL baseline solver is tiled solely to facilitate task parallelization.

On the other hand, tiling is very fundamental to our SSTA solver. Our performance model (§III-A) is based on caching assumptions which we rely on tiling to enforce. In particular, we assume that all four arrays (a, b, c and u) remain in the cache hierarchy in between the loops. If any of these four arrays need to be reloaded from main memory in between the forward elimination and back substitution loops, the amount of main memory data movement is significantly increased and the algorithmic intensity of the algorithm decreases dramatically. While our untiled SSTA results outperform the MKL baseline results (Figure 6), they were well below the manufacturer-specified bandwidth and STREAM Triad effective bandwidth. On the platforms covered in this paper, we typically aim to fit within the L2 cache.

It is important to note the distinction between **per-array tile size** (e.g. the tile size for one of the four arrays) and the **total tile size** (e.g. the sum of the per-array tile sizes). The total tile size is the amount of data we need to remain in cache between the forward elimination and back substitution loops. Unless stated otherwise, when we refer to "tile size" we mean total tile size.

Due to the loop-carried dependencies in the Thomas algorithm and the small extent of nk, we could not tile the vertical dimension. So, we explored two tiling schemes which partition the horizontal ij-plane.

The **tile-ij** scheme partitions both the i and j dimensions, yielding tiles with arbitrary horizontal shape. The **tile-j** scheme partitions only the j dimension, producing tiles containing contiguous i-rows. There is a trade-off between these two schemes, with the tile-ij scheme offering greater flexibility in tile size and the tile-j scheme offering greater contiguity.

For example, suppose we partition an ijk-layout grid into four tiles each containing $(d^2/4)(e)$ elements, with the tile-ij scheme producing $d/2 \times d/2 \times e$ tiles (Figure 3a) and the tile-i scheme producing $d \times d/4 \times e$ tiles (Figure 3b). For the tile-ij scheme, tiles contain $(d/2)(e)$ contiguous i-rows of only $d/2$ elements each. For the tile-j scheme, tiles contain e contiguous ij-planes of $d^2/4$ elements each.

While the tile-j scheme offers greater contiguity than the tile-ij in this example, the tile-j tiles are not a single contiguous memory region with the ijk-layout. The only way to obtain completely contiguous tiles in the ijk-layout would be to tile the dimension with the greatest stride (k), which is not an option as we never partition the vertical dimension.

This limitation led us to the ikj-layout for the SSTA solver (see §V). With this layout, the direction of vectorization (i) is still the unit stride dimension, avoiding strided loads and assisting hardware prefetching. Additionally, the tile-j tiles are completely contiguous regions of memory, decreasing translation-lookaside buffer (TLB) pressure, increasing locality for caching and further assisting hardware prefetching.

Consider switching to the ikj-layout in the previous example. The tile-ij tiles would be the same as before: $d/2 \times d/2 \times e$ tiles consisting of $(d/2)(e)$ contiguous i-rows of only $d/2$ elements each (Figure 3c). However, the tile-j tiles would be one contiguous region containing all $(d^2/4)(e)$ elements (Figure 3d).

The only notable downside to the tile-j scheme is the loss of flexibility in tile sizes. The smallest tile size for the tile-j scheme is a $d \times 1 \times e$ tile (an ik plane), while the smallest tile size for the tile-ij scheme is a $1 \times 1 \times e$ tile (a single column). We have not found this restriction to be overly restrictive.

We present results with the tile-j scheme and both the ijk- and ikj-layout in this paper.

IV. Experimental Setup

We developed the Tridiagonal Solve Benchmarks (TSB) suite during the course of our research [20]. The TSB suite is freely available on Github under the Boost Software License, version 1.0. The benchmark suite has three major components: a multi-dimensional array abstraction, generic solver components and a test harness.

A. Test Problem

The benchmarks in the TSB suite solve a matrix in each vertical column derived from a simple implicit Backward Time, Centered Space (BTCS) finite difference stencil with Dirichlet boundary conditions every time step. The diagonally-dominant tridiagonal matrix takes the following form, where D is a dimensionless diffusion coefficient ($D > 0$), Δt is the time step size, Δk is the vertical grid spacing and $r = D\Delta t/(\Delta k)^2$:

$$A = \begin{bmatrix} 1 & 0 & & & 0 \\ -r & 1+2r & -r & & \\ & \ldots & \ldots & \ldots & \\ & & -r & 1+2r & -r \\ 0 & & & 0 & 1 \end{bmatrix}$$

An identical form of the problem is initialized in every vertical column of the grid, although the matrix for each is constructed separately for each solve.

B. Hardware Platforms

The TSB suite currently targets x86-64 microarchitectures with SIMD vector units running POSIX-compliant operating systems. The results presented in this paper were collected from both Intel Xeon and Xeon Phi systems.

Our two Intel Xeon platforms, Edison and Cori Phase 1, are homogeneous Cray supercomputers, consisting of traditional dual-socket x86-64 nodes [21]. Edison features Intel Xeon E5-2695 v2 Ivy Bridge (IVB) processors [22], and Cori Phase 1 has Intel Xeon E5-2698 v3 Haswell (HSW) processors [23]. In general, the two Xeon platforms have a very similar performance profile for our application. Commensurate with MPI parallelization for distributed memory, in all experiments we restricted ourselves to a single NUMA nodel

Our Xeon Phi testbed, which features Intel Xeon Phi 7210 Knight's Landing (KNL) processors [24], is notably different from the two Xeon systems. Knight's Landing is a many-core design with a 2D mesh of **lightweight** cores optimized for throughput and explicit parallelism at the expense of increased latency and reduced complexity in other areas (branch prediction, out-of-order execution facilities, pipeline depth, etc). The KNL microarchitecture has a number of novel features including on-package high-bandwidth MCDRAM, 4 hyper-threads per core and dual 512-bit AVX512 vector units. [25], [26]

Knight's Landing processors can be configured for different Non-Uniform Memory Access (NUMA) topologies. Additionally, the MCDRAM can be configured as programmable memory, a direct-mapped last-level cache or a combination of the two [26]. For all experiments in this paper, we used a `quadcache` [25] configuration, where all processing units are in a single NUMA domain and all 16GB of MCDRAM are used as a cache.

C. Toolchain

TSB is written in ISO C++14 [27] and has no external software dependencies. The code contains no vector intrinsics or vector assembly. We rely entirely on the compiler vectorization engine for performant vector code generation, utilizing compiler directives to guide parallelization and vectorization, and indicate alignment, loop trip count and aliasing assumptions.

SSTA is task-parallelized using OpenMP `#pragma`s [28]. Since the problem is embarrassingly parallel in the horizontal dimensions, it is easy to statically load balance (dimension extents and tile sizes permitting).

The Intel C++ Compiler [29] was used to compile the TSB suite for all of the results presented in this paper. We used the 2017 Beta Update 2 version (`17.0.0 20160517`). We also used the Intel VTune Amplifier XE general-purpose profiler [17] during our research.

D. Statistical Considerations

Effective bandwidth is our primary performance metric. We define effective bandwidth as **effective data movement / solver execution time**.

We use the theoretical peak performance model for data movement in the Thomas algorithm defined in §III-A to estimate effective data movement. To measure solver execution time, we record and average the wall-clock execution time of each timestep. We include OpenMP parallelization in our measurements, e.g. we time the `#pragma omp for` loop instead of measuring the duration of each tile-iteration individually. Thus, parallel overheads are included in our recorded execution times.

We performed each individual experiment a statistically significant number of times on different nodes of our test systems. Sample sizes varied between experiments and platforms, but were usually between 100 to 200 independent executions per data point. We estimated variance between **different executions** of the benchmark with identical parameters by computing sample standard deviation. All measurements presented are **95% confidence intervals** (depicted visually with bars on all graphs) constructed from the mean and sample standard deviation of the dataset.

We used STREAM [9] Triad results as an reference for peak effective bandwidth. We measured STREAM Triad bandwidth between 49.74 GB/s and 49.86 GB/s on our Ivy Bridge system, between 60.36 GB/s and 60.44 GB/s on our Haswell system and between 418.2 GB/s and 419.6 GB/s on our Knight's Landing system.

We believe there are two potential sources of non-trivial **systemic observational error** in our results. First, mistakes in our analytic performance model for data movement could potentially skew our data. We mitigated this by verifying the model with hardware-based profilers (Intel VTune Amplifier XE [17]), and we are confident in its validity. Second, variance in the execution time of **individual timesteps** is not accounted for. We currently average the execution time of all timesteps within a single execution of a benchmark, but do not compute and record sample standard deviation as a variance estimation **within** the solver. Removing this source of systemic error would be straightforward, but would require extending our postprocessing framework to compute running sample standard deviations [30] which we have not yet implemented.

V. Results

Figure 5: SSTA Effective Bandwidth vs. Total Tile Size (Knight's Landing): Shown below are the results of a parameter sweep on our Intel Xeon Phi Knight's Landing platform showing the effect of data layout and total tile size on the performance of the SSTA solver. MKL baseline (lower-bound) and STREAM Triad (upper-bound) results are shown for reference. ijk-layout performance is hampered by poor data contiguity which incurs expensive TLB and hardware prefetching penalties on Knight's Landing. This effect is amplified at smaller tile sizes. The best ikj-layout result has between 91 GB/s and 109 GB/s higher effective bandwidth than the best ijk-layout result, and reaches between 88% and 92% of STREAM Triad performance. Results are shown with 95% confidence.

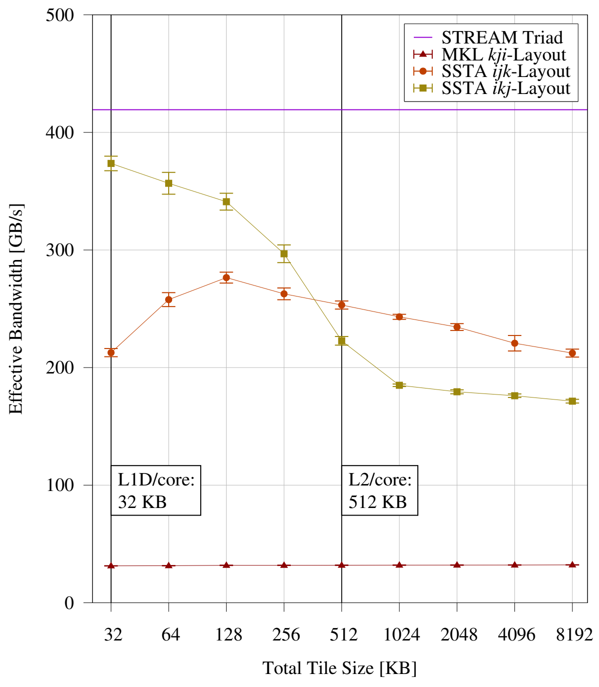
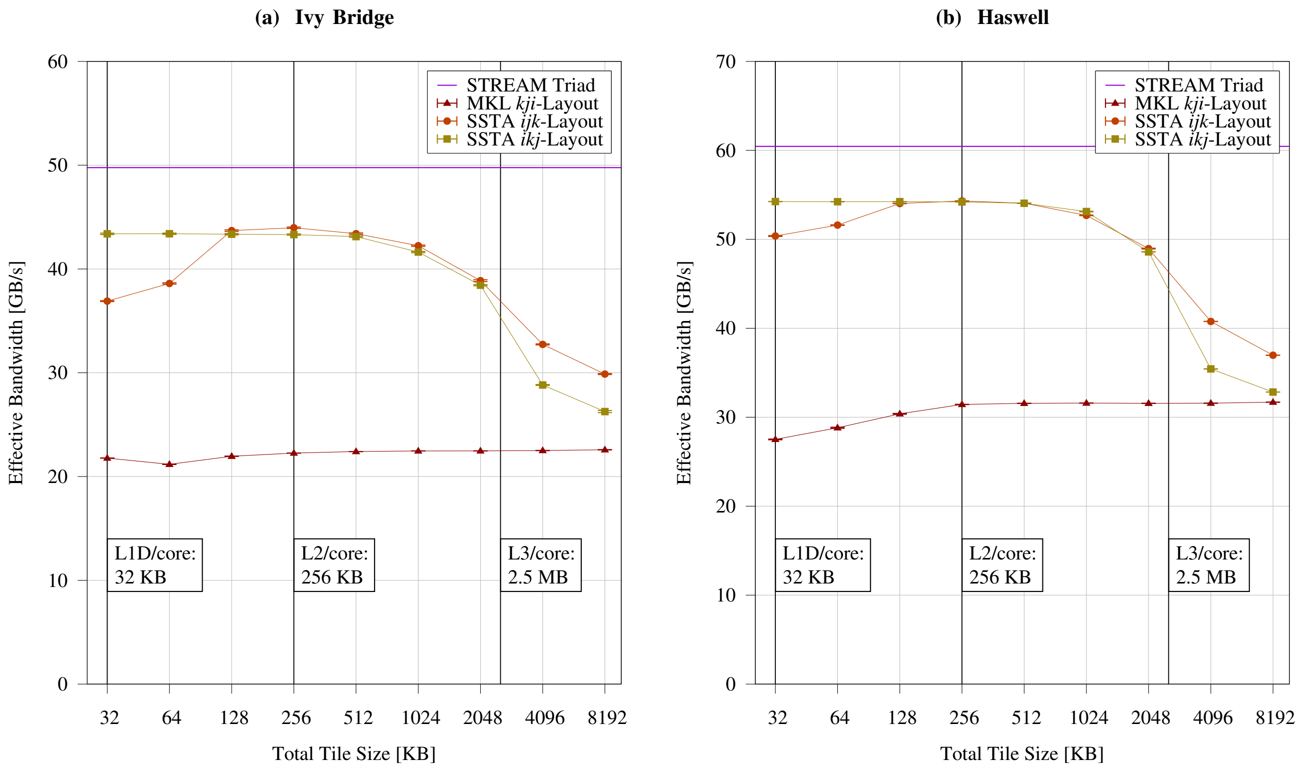


Figure 4: SSTA Effective Bandwidth vs. Total Tile Size: The results of a parameter sweep on our two Intel Xeon platforms demonstrating the effect of total tile size on the performance of the SSTA solver is shown below (MKL baseline and STREAM Triad results shown as lower- and upper-bounds). The *ikj*-layout, which exhibits better contiguity and memory access patterns, performs better at smaller tile sizes, although the best *ikj*-layout result does not outperform the best *ijk*-layout result on these platforms. Performance degrades for tile sizes which exceed L3 capacity per core. Both SSTA variants substantially outperform the MKL baseline results and reach ~90% of STREAM Triad effective bandwidth with optimal tile sizes. Results are shown with 95% confidence.



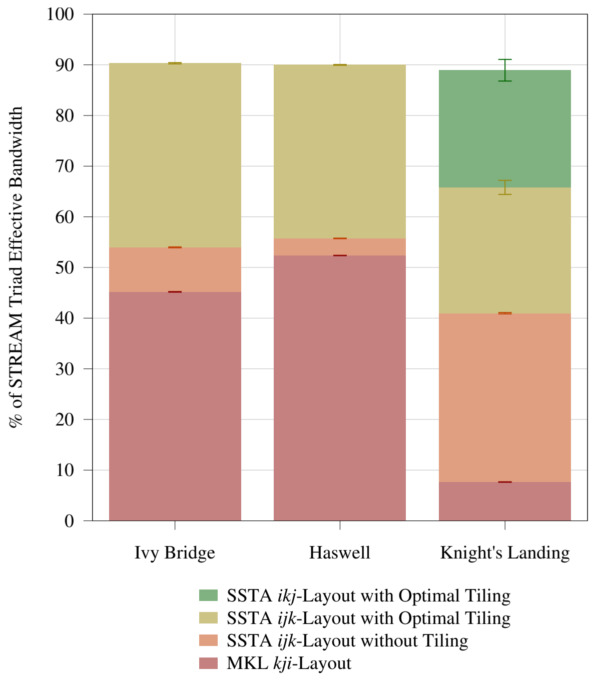
We developed SSTA to replace the MKL-based vertical column solver in our production climate application because we believed better vectorization and data movement would be achieved by solving multiple vertical columns simultaneously. We also theorized that such a solver would be highly sensitive to **data layout changes** and **total tile size** (§III-D). Parallel application performance is often driven by parameters such as tile size, which control the amount of work in each parallel task. In memory-bandwidth bound applications, data layout and tiling can be especially important as they not only influence the amount of task- and vector-parallelism exposed but also the working set size and degree of cached data reuse in each individual task. We anticipated that we would need to study the effect of different data layouts and tile sizes in order to understand the performance of our new solver. The first step was to determine the **optimal total tile size** for the different solver variants in the TSB suite.

We conducted a parameter sweep of total tile size on our Xeon and Xeon Phi platforms. We predicted that we would see the following:

- The MKL baseline solver would be very insensitive to total tile size, excluding extremely small (high overhead) and extremely large (insufficient parallelism) sizes since it solves each column independently and thus cannot exploit vector operations and data locality like the SSTA solver does.
- Total tile sizes small enough to fit into the L1D cache would not be feasible because the overheads of loop constructs and parallelization/vectorization setup would be too great relative to the execution time of useful work per inner-loop iteration. These tile sizes would either require vertical extents below the sizes we are interested in ($nk < 16$) or horizontal dimensions too small to vectorize efficiently ($ni < 16$).
- The optimal total tile size would fit into the L2 cache (but not the L1D cache), since it is the fastest cache which has the capacity to contain a feasible tile size.
- Excluding the L1D, as we move from a tile size which fits into the capacity of a particular cache to a tile size which does not, we should see a drop in effective bandwidth. On the Xeon platforms, the boundaries are L2 $\rightarrow$ L3 and L3 $\rightarrow$ DRAM (main memory). On Knight's Landing, the boundaries are L2 $\rightarrow$ MCDRAM and MCDRAM $\rightarrow$ DRAM (main memory).

Note that when a total tile size is equal to a particular cache capacity, we claim that it will **not** entirely fit into that cache, since it is not reasonable to assume that our arrays will be able to consume the entire cache's capacity.

Figure 6: Percentage of STREAM Triad Effective Bandwidth Attained by SSTA: Across all three microarchitectures, the SSTA solver with an optimal tile size improves performance substantially over the MKL baseline, and achieves ~90% of STREAM Triad effective bandwidth. Although untiled SSTA results beat the MKL solver, tiling contributes significantly to overall performance: between 35.9% and 36.1% on Ivy Bridge, 34.22% and 35.38% on Haswell and 24% and 26% on Knight's Landing. Between 21% and 25% of STREAM performance attained is attributed to the switch to the *ikj*-layout on Knight's Landing. Results are shown with 95% confidence.



The results of this parameter sweep for both the ijk- and ikj-layout are shown in Figure 4a (Ivy Bridge), Figure 4b (Haswell) and Figure 5 (Knight's Landing). Results from the TSB suite's MKL baseline solver are shown as a lower bound, and STREAM Triad results are shown as an upper bound. A $32 \times 147456 \times 32$ grid of double precision floating point values (4.5GB) was used on all platforms. All results were run on a full socket, with one application-thread per core.

A. Analysis

We found that the MKL baseline solver was largely insensitive to tile size, as anticipated. Across the range of total tile sizes we tested, we did not observe performance variation that was statistically significant enough to distinguish it from random observational error.

As we predicted, total tile sizes small enough to fit into the L1D cache were impractically small. We would have to either reduce the vertical extent or the horizontal extent ni to generate a tile size small enough to fit into the 32KB L1D cache with the tile-j scheme. We ran preliminary experiments with $ni = 16$ (not shown) to observe the effect of 16KB total tile sizes. The results indicated that tile sizes smaller than 32KB performed worse for both the ijk- and ikj-layouts. At $ni = 16$, the trip count on some of the loops in SSTA is so small that the compiler cannot perform as much unrolling after vectorization as it would for larger i extents. We have not determined if switching to the tile-ij scheme might make smaller tile sizes feasible without decreasing the extent of the horizontal dimension ni or decreasing performance via worse data contiguity.

On Knight's Landing, the optimal total tile size for the SSTA ijk-layout variant is 128KB, which is small enough to fit within the L2 but too large for the L1D. On Ivy Bridge and Haswell, the optimal total tile size for the ijk-layout is 256KB. While it is difficult to distinguish between the 128KB, 256KB and 512KB results in Figures 4a and 4b because the magnitude of the difference is relatively small, the confidence intervals do not overlap and the trend of the dataset supports our conclusion (see Figure 4).

We were intrigued to find that, contrary to our predictions, the optimal total tile size for the ijk-layout on the Xeon platforms was too large to fit within the L2. Additionally, while the optimal total tile size for the ijk-layout on Knight's Landing would fit within the L2 cache, performance was not as close to peak as it was on the Xeon platforms (see Table II). Profiling indicated that the SSTA ijk-layout variant experienced a high number of L1 DTLB store misses. As we decreased total tile size, we observed progressively worse TLB performance in our profiling traces. We determined this was due to the lack of fully contiguous tiles in the ijk-layout (see §III-D). As the per-array tile size is decreased, the size of each contiguous plane decreases and the frequency of non-contiguous jumps through memory increases, negatively impacting TLB, L1D and hardware prefetching performance, especially on Knight's Landing (See Figure 5).

We conducted a parameter sweep with the SSTA ijk-variant and a smaller vertical extent ($nk = 16$, not shown) to further verify the cause of this issue. With fewer vertical levels, the number of contiguous regions in each tile would be decreased and the size of each contiguous region would be increased, improving contiguity. We observed performance improvements with $nk = 16$; on Knight's Landing the impact was significant.

The poor performance of the ijk-layout on Knight's Landing led us to develop the ikj-layout variant of SSTA described in §III-C and §III-D. The optimal total tile size for the ikj-layout was 32KB on all three platforms. On the Xeon platforms, the best ikj-layout result does not outperform the best ijk-layout result, although the ikj-layout variant does perform better than the ijk-layout variant at smaller tile sizes. However, on Knight's Landing the best ikj-layout result achieves between 91 GB/s and 109 GB/s higher effective bandwidth than the best ijk-layout result (see Table II), a significant improvement.

Our prediction that the optimal total tile size would be larger than L1D capacity but smaller than L2 capacity held for the ikj-layout on all platforms, and for the ijk-layout on Knight's Landing. For the ijk-layout on the Xeon platforms, the optimal total tile size was too large to fit into the L2. We believe the difference between optimal total tile sizes on the Xeon platforms and Knight's Landing is due to the lack of a traditional L3 cache on the latter, making the performance penalty greater for L2 misses. We can draw the conclusion

Table II: **Summary of SSTA Performance Results:** Optimal tile sizes, effective bandwidth, percentage of STREAM Triad performance attained and speedup over the MKL baseline solver are given below for all SSTA experiments. ~90% of STREAM Triad effective bandwidth is obtained on all platforms. All measurements are averages of a statistically significant number of samples and are reported with 95% confidence.

Solver	Optimal Total Tile Size	Effective Bandwidth Bandwidth	Percentage of STREAM TRIAD Bandwidth	Speedup vs. MKL
Ivy Bridge				
MKL kji-layout	Any	22.49 ± 0.003 GB/s	45.2 ± 0.05 %	1x
SSTA ijk-layout	256 KB	44.0 ± 0.07 GB/s	90 ± 0.1 %	~2x
SSTA ikj-layout	32 KB	43.0 ± 0.07 GB/s	87 ± 0.2 %	~1.9x
Haswell				
MKL kji-layout	Any	31.65 ± 0.006 GB/s	52.4 ± 0.04 %	1x
SSTA ijk-layout	256KB	54.3 ± 0.01 GB/s	90.0 ± 0.07 %	~1.7x
SSTA ikj-layout	32KB	54.2 ± 0.01 GB/s	89.7 ± 0.06 %	~1.7x
Knight's Landing				
MKL kji-layout	Any	32 ± 0.2 GB/s	7.6 ± 0.04 %	1x
SSTA ijk-layout	128KB	280 ± 5 GB/s	66 ± 1 %	~9x
SSTA ikj-layout	32KB	370 ± 6 GB/s	90 ± 2 %	~12x

that, for the platforms surveyed, the optimal total tile size was for the ijk-layout is large enough to reside in the last-level associative on-die cache (the L3 on the Xeon platforms and the L2 on Knight's Landing).

The total tile size parameter sweep somewhat supports our prediction that we would see drops in performance as we moved from tile sizes that would fit into a particular cache to tile sizes that would not. We see no such decrease at the L2 capacity boundary on the Xeon platforms, but we do see a drop when the tile size exceeds L3 capacity per core. On Knight's Landing, we do see a drop as we cross the L2 capacity boundary. We posit that a modified version of our prediction is more accurate; as we move from a tile size which will fit into the last-level associative on-die cache to a tile size which will not, there is a drop in SSTA's effective bandwidth.

VI. Conclusion

We introduced the Simultaneous Streaming Thomas Algorithm (SSTA) solver, which can efficiently solve many small tridiagonal matrix systems simultaneously, and addresses a performance bottleneck that arises in a number of scientific application domains. We have demonstrated the impact of different tiling schemes, total tile sizes and data layouts, and how they interact with the memory subsystem. A parameter sweep demonstrated the sensitivity of the SSTA solver to changes in tile size and data layout. We found that the ijk-layout version of SSTA provides the best performance on Xeon platforms with a total tile size that is small enough to fit into the L3 cache but is too large to fit in L2. On Knight's Landing, the ikj-layout yields the best performance with a tile size which is small enough to fit in L2.

Our results are summarized in Table II. They show SSTA is a highly efficient solver capable of achieving 90% of STREAM Triad bandwidth on Intel Xeon and Xeon Phi systems, including the new Knight's Landing microarchitecture. Our algorithm is a substantial improvement over the MKL-based kji-layout baseline solver; it is ~2x faster on Xeon platforms and ~12x faster on Knight's Landing.

VII. Future Work

There are a number of optimizations which we developed for the SSTA solver but have not presented here.

The version of the TSB suite described in this paper uses what we refer to as the "full-grid" scheme, where the tridiagonal matrix is built and stored for all vertical columns. The alternative "rolling-grid" scheme builds the matrix on the fly with a tile-sized a, b and c. In addition to offering storage savings, we believe the rolling-grid scheme will further improve data locality and hardware prefetching performance as the storage for the matrix will be reused for all tiles that a processing unit executes. This would be especially appropriate for non-linear solvers, where the matrix needs to be updated every iteration. Further research is also needed to explore smaller total tile sizes, as well as the tile-ij scheme, as it may be beneficial to production applications that are willing to accept some performance trade-off for increased flexibility in the extent of the horizontal dimension i.

We would like to extend the range of hardware platforms and configurations in future experiments. On Knight's Landing, further effort is needed to study the trade-off between utilizing MCDRAM as a cache versus programmable memory. We are also interested in testing SSTA on GPGPU architectures.

There are also a number of related numerical problems which we believe could be addressed by some of the techniques we describe in this paper. The algorithm needs to be extended to include matrix assembly based on solution values, and put inside a fully non-linear iteration with a Jacobian calculation, such as those used in Newton iterations. The challenge here is that some vertical columns may converge faster than others, and vector masks or scalar iterative refinement may be required. Opportunities to extend the algorithm to banded or other sparse matrices, and incomplete LU-type algorithms are straight-forward for diagonally-dominant cases, but pivoting has the potential to hinder vectorization and memory access patterns.

Finally, more complex multi-kernel applications integrate multiple solvers and computationally phases, such as traditional finite difference stencil operations and time integration algorithms. We will have to explore the performance trade-offs between the *kji*-, *ijk*- and *ikj*-layouts in these applications. Our efforts in the immediate future will revolve around research into those trade-offs and integration of SSTA into our production software.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. This work was partially supported by the Intel Parallel Computing Center (IPCC) at Lawrence Berkeley National Laboratory, and used resources at the National Energy Research Scientific Computing Center (NERSC), which is supported by the U.S. Department of Energy Office of Science's Advanced Scientific Computing Research (ASCR) program under contract number DE-AC02-05CH11231.

References

[1] Gettelman, A., and Morrison, H. Advanced Two-Moment Bulk Microphysics for Global Models Part I: Off-Line Tests and Comparison with Other Schemes. *Journal of Climate 28*, 3 (2015), 1268–1287.

[2] Will E. Pazner, e. a. A High-Order Spectral Deferred Correction Strategy for Low Mach Number Flow with Complex Chemistry. *Combustion Theory and Modelling 20*, 3 (2016), 521–547.

[3] Michael J. Iacono, e. a. Radiative Forcing by Long-Lived Greenhouse Gases: Calculations with the AER Radiative Transfer Models. *Journal of Geophysical Research: Atmospheres 113*, D13 (2008).

[4] Guoping Tang, e. a. Addressing Numerical Challenges in Introducing a Reactive Transport Code into a Land Surface Model: a Biogeochemical Modeling Proof-of-Concept with CLM-PFLOTRAN 1.0. *Geoscientific Model Development 9*, 3 (2016), 927–946.

[5] Khairoutdinov, M., Randall, D., and DeMott, C. Simulations of the Atmospheric General Circulation using a Cloud-Resolving Model as a Superparameterization of Physical Processes. *Journal of the Atmospheric Sciences 62*, 7 (2005), 2136–2154.

[6] Weller, H., Lock, S.-J., and Wood, N. Runge-Kutta IMEX Schemes for the Horizontally Explicit/Vertically Implicit (HEVI) Solution of Wave Equations. *Journal of Computational Physics 252* (2013), 365–381.

[7] Intel Math Kernel Library (MKL). http://goo.gl/ZzbZ5B.

[8] NVIDIA CUDA BLAS Library (cuBLAS). http://goo.gl/nB4lO5.

[9] McCalpin, J. D. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (dec 1995), 19–25.

[10] Blaze source code repository. http://goo.gl/3l0D3t.

[11] The Numerical Template Toolkit (nt2) source code repository. http://goo.gl/KSld9R.

[12] Povitsky, A. *Parallelization of the Pipelined Thomas Algorithm.* Institute for Computer Applications in Science and Engineering (ICASE), 1998. http://goo.gl/tO0Z23.

[13] Azzam Haidar, e. a. Optimization for Performance and Energy for Batched Matrix Computations on GPUs. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs* (2015), pp. 59–69.

[14] Conte, S. D., and Boor, C. W. D. *Elementary Numerical Analysis: An Algorithmic Approach*, 3rd ed. McGraw-Hill Higher Education, 1980.

[15] Williams, S., Waterman, A., and Patterson, D. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM 52*, 4 (2009), 65–76.

[16] *Intel 64 and IA-32 Architectures Software Developers' Manual*, 325383-059US ed., 2016. http://goo.gl/OzZKIA.

[17] Intel VTune Amplifier XE. http://goo.gl/F218Pw.

[18] Lam, M. D., Rothberg, E. E., and Wolf, M. E. The Cache Performance and Optimizations of Blocked Algorithms. *ACM SIGPLAN Notices 26*, 4 (1991), 63–74.

[19] Edwards, H. C., Trott, C. R., and Sunderland, D. Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing 74*, 12 (2014), 3202–3216.

[20] Tridiagonal Solve Benchmarks (TSB) suite source code repository. http://goo.gl/iuhHst. Available under the Boost Software License 1.0 (a BSD-style open source license).

[21] National Energy Research Scientific Computing Center (NERSC) Systems. http://goo.gl/0OgfCP.

[22] Intel Xeon Processor E5-2695 v2 (30M Cache, 2.40 GHz) Specifications. http://goo.gl/84vkJd.

[23] Intel Xeon Processor E5-2698 v3 (40M Cache, 2.30 GHz) Specifications. http://goo.gl/t0GdZr.

[24] Intel Xeon Phi Processor 7210 (16GB, 1.30 GHz, 64 core) Specifications. http://goo.gl/xLtmjm.

[25] Douglas Doerfer, e. a. Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor. In *Intel Xeon Phi User Group Workshop Annual US Meeting* (2016).

[26] Sodani, A. Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor. In *27nd Hot Chips Symposium* (2015). http://goo.gl/AkIzBk.

[27] *ISO/IEC 14882:2014, Standard for Programming Language C++*, 2014. http://goo.gl/bCZssW.

[28] *OpenMP Specification Version 4.5*, 2015. http://goo.gl/QaZ1FD.

[29] Intel C++ Compiler. http://goo.gl/Vf7VIJ.

[30] Lelbach, B. A. Benchmarking C++ Code. In *CppCon 2015* (2015). http://goo.gl/S83V34.