SAND2017-13020C

Designing Vector-Friendly Compact BLAS and LAPACK Kernels

Kyungjoo Kim
Sandia National Laboratories
Albuquerque, New Mexico
kyukim@sandia.gov

Timothy B. Costa
Intel Corporation
Hillsboro, Oregon
timothy.b.costa@intel.com

Mehmet Deveci
Sandia National Laboratories
Albuquerque, New Mexico
mndevec@sandia.gov

Andrew M. Bradley
Sandia National Laboratories
Albuquerque, New Mexico
ambradl@sandia.gov

Simon D. Hammond
Sandia National Laboratories
Albuquerque, New Mexico
sdhammo@sandia.gov

Murat E. Guney
Intel Corporation
Hillsboro, Oregon
murat.e.guney@intel.com

Sarah Knepper
Intel Corporation
Hillsboro, Oregon
sarah.knepper@intel.com

Shane Story
Intel Corporation
Hillsboro, Oregon
shane.story@intel.com

Sivasankaran Rajamanickam
Sandia National Laboratories
Albuquerque, New Mexico
srajama@sandia.gov

ABSTRACT

Many applications rely on the use of BLAS/LAPACK routines on large groups of very small matrices. For example, many PDE-based simulations and machine learning applications require batched BLAS/LAPACK routines. While existing batched BLAS APIs provide meaningful speedup over alternatives like OpenMP loops around traditional BLAS/LAPACK kernels, there exists potential for significant speedup by considering a non-canonical data layout that allows for cross-matrix vectorization in batched BLAS/LAPACK routines. In this paper we propose a new compact data layout that interleaves matrices in blocks according to the architecture's SIMD vector length and investigate its benefits. Second, we combine the compact data layout with a new compact batched layer/interface to BLAS/LAPACK routines that can be used within a hierarchical parallel application. We demonstrate significant benefits provided by this layer due to increased locality and reduced synchronization costs. Third, we discuss the compact batched BLAS/LAPACK implementations and APIs in two libraries, an open-source reference implementation (KokkosKernels) and a high-performance vendor implementation (The Intel® Math Kernel Library®) and present performance results for both libraries. In our experiments, the compact batched data layout provides up to 4.5×, 16.5× and 21.6× speedup against OpenMP loops around `dgemm`, `dtrsm` and `dgetrf` kernels, respectively, with block size 5 on the Intel Knights Landing architecture. Finally, we demonstrate the benefits of the compact batched routines in a line solver for coupled PDEs by comparing it with the original line solver implementation in a computational fluid dynamics code. The compact batched routines provide 2×-6× speedup for problem sizes of interest.

ACM Reference format:
Kyungjoo Kim, Timothy B. Costa, Mehmet Deveci, Andrew M. Bradley, Simon D. Hammond, Murat E. Guney, Sarah Knepper, Shane Story, and Sivasankaran Rajamanickam. 1997. Designing Vector-Friendly Compact BLAS and LAPACK Kernels. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colorado USA, November 2017 (SC17),* 14 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Dense linear algebra subroutines have a long history of standardization [5, 9, 15], performance optimization [10, 11] and use in applications. While these standards have been foundational in multiple generations of high performance computing, application and architectural changes today require new designs [2, 8]. Several applications, such as PDE based simulations and machine learning, are starting to rely on a large number of linear algebra operations (BLAS/LAPACK) applied to very small matrices. Also, the evolution of hardware to allow massive parallelism and increasing vector lengths impact the implementation of foundational linear algebra subroutines.

For groups of small matrix problems the community has developed batched BLAS approaches, which enhance performance by introducing parallelism over the individual BLAS/LAPACK operations. However, existing batched BLAS/LAPACK implementations based on column or row major data layouts have limited performance for small matrix sizes. For very small sizes there is simply too little data to take full advantage of the SIMD vector length in modern processors. In this work we consider a SIMD-friendly data layout which can take full advantage of the long SIMD length in modern processors for groups of small BLAS/LAPACK operations through cross-matrix vectorization. Performance optimization of such kernels, especially for a large number of small problems, depends on a number of design choices. The key performance optimizations that we explore in this work are *data layouts, vectorization, and cache-friendly interface design.* To motivate the methods developed in this paper, we approach the problem from the perspective of an entire application: a line solver for coupled PDEs. The impact on the application greatly influences our design choices. Specifically, while the community focus on batched kernels has been around fixed

SC17, Denver, Colorado USA
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

or variable sized interfaces for batched kernels [8, 17], GPU implementations [1, 3, 4], or group based interfaces [27], we introduce a two-level interface that results in better cache-locality when composing multiple linear algebra kernels. The result is a set of highly efficient, vector-friendly BLAS/LAPACK kernels for small matrices typical in applications. The proposed data layout, two-level interface and implementation is called *compact batched* BLAS/LAPACK throughout the paper for brevity.

Contributions. The primary contribution of this paper is the introduction of new BLAS/LAPACK kernels based on the *compact data layout*. The proposed compact BLAS/LAPACK kernels are implemented in two libraries, a performance-portable open source implementation (Github[1]) and a vendor specific library targeting architecture specific improvements. While the focus of batched BLAS/LAPACK to date has been on DGEMM performance, we extend this to the more complex routines DTRSM and DGETRF. On Intel Knights Landing architecture, our implementations of DGEMM, DTRSM and DGETRF result in up to 14×, 45×, and 27× speedup against OpenMP loops around highly optimized DGEMM, DTRSM and DGETRF kernels, respectively. Finally, we demonstrate the efficacy of the compact batched BLAS/LAPACK subroutines by using them in a line solver for coupled PDEs. The compact batched routines provide 2×-6× speedup for problem sizes of interest. Detailed performance analysis for vector utilization and arithmetic intensity of the kernels based on a custom PIN tool APEX [12].

The rest of the paper is organized as follows. The motivating applications are described first in Section 2. The compact layout, interface, and implementation are described in Section 3. We then demonstrate the strengths of our design with performance results of the compact batched kernels (Section 4.1) and the line solver application (Section 4.2). Vector utilization and arithmetic intensity measures are used to analyze the performance using a roofline model (Section 4.3).

2 MOTIVATING APPLICATIONS

A fundamental distributed data structure in HPC is a *block* sparse matrix, implemented in either compressed row storage (CSR) or compressed column storage (CSC) formats. A CSR or CSC graph encodes dependencies between blocks, i.e., the graph. Blocks may be one fixed size throughout the matrix, or have variable size. In this paper we consider one fixed block size $b \times b$. A higher-level data structure can be used to build matrices having variable block sizes from matrices having fixed block size [7]. Each block is typically dense.

PDE-based simulations form one class of HPC application using block matrices. The discretization is over a mesh. A block is associated with a mesh entity, such as a node, edge, face, or cell center. The block size is the number of degrees of freedom associated with the mesh entity. For example, a 3D compressible fluid dynamics model using the ideal gas model and with all degrees of freedom at the cell center has 5×5 blocks. As another example, a 3D solid mechanics nodal finite element model using the linear elasticity material model has 3×3 blocks. In simulation codes modeling complicated physical phenomena, b can be several tens [20].

Compared with a *point* sparse matrix, where a *point* can be understood as corresponding to a 1×1 block or, in other words, a scalar, a block sparse matrix uses b^2 fewer ordinals to encode the graph. In addition, computations within and between a block and a vector do not require indexing except to compute the offsets to the block quantities.

Computations within a block sparse matrix (e.g., an incomplete factorization), between two matrices (e.g., a matrix-matrix multiplication), and between a matrix and a multivector (e.g., matrix-vector product, triangular solve) use many small BLAS 1, 2, and 3 subroutine calls. The structure of the computation determines the extent to which these calls may occur in parallel.

In a typical PDE-based application, a block sparse matrix is filled with discretization coefficient values. Then a preconditioner is formed as a function of the matrix. Finally, an iterative linear solver performs a sequence of matrix-vector products and preconditioner applications.

One commonly used preconditioner is the *line* smoother or solver. It arises in a simulation in two or more dimensions in which independent equations are solved along one dimension, either as an approximation within a preconditioner, because of decoupling of time or space scales, or because of a mix of implicit and explicit time integration. For example, Tuminaro et al. [22] solve independent equations in the vertical direction of an ice sheet as the smoother in an algebraic multigrid preconditioner. U. of Minnesota's US3D [6], NASA's DPLR [26], and Sandia National Laboratories' SPARC use a line smoother in a fixed-point iteration [26] to solve the Navier-Stokes equations for compressible and reacting flow. Lines are formed with the intention that they be approximately orthogonal to the shocks that form in the simulation. Nonhydrostatic atmosphere solvers use the horizontally explicit, vertically implicit (HEVI) time integration method to remove the vertical acoustic wave speed from the time step restriction [24]. Segall et al. [19] solve for pressure and temperature orthogonal to a fault, with no coupling along fault.

In each of these applications, a large number of independent block-tridiagonal matrices are formed. Operations on and with the block-tridiagonal matrices may be performed in parallel. We refer to this kind of parallel work as *batch* parallelism. Because of the typical topologies of the meshes, the block-tridiagonal matrices often have the same size.

There are a number of divide-and-conquer methods to expose parallelism within an operation on or with a single tridiagonal matrix, such as cyclic reduction [21] and prefix product [18]. Each method recursively forms independent smaller problems; the recursion depth can be adjusted. Each method is slightly work inefficient. Thus, if the number of block-tridiagonal matrices times the amount of parallelism within a single block operation is at least a few times greater than the available hardware parallelism, it is optimal to exploit only batch parallelism. If the number is less than the available hardware parallelism, then algorithmic methods can be used with a recursion depth that uses batch parallelism maximally. This paper focuses only on batch parallelism.

[1] https://github.com/kokkos/kokkos-kernels

3 COMPACT BLAS/LAPACK

Traditional BLAS implementations based on the conventional data layout (either column major or row major dense matrices) have limited performance for small problem sizes. With small matrix sizes there is too little data to take advantage of all of the vector registers and the data is too small to fill the vector registers that are used, resulting in limited benefit from vectorization. For a single BLAS operation, performance can be improved through the use of kernels specifically tuned for the problem size. For example, one can use Just-in-Time (JIT) code-generation [13]. For general matrix multiplication (GEMM) this can be very effective in improving performance. It remains to be seen if this approach can be beneficial for a broader set of more complicated BLAS or LAPACK functions. Additionally, a JIT strategy for generating problem-size tuned kernels still does not address the fundamental problem of vector register fill for small problems.

When there are many matrix operations to be performed simultaneously we can consider alternative data layouts that allow the application to benefit from kernel vectorization. The compact data layout, which is the subject of this work, is a SIMD-friendly layout with considerable advantages in performance for groups of many small matrix operations.

In this section we first describe the compact data layout, identifying its benefit in the context of BLAS/LAPACK kernel vectorization. We then explore the key differences in optimized implementations of GEMM and TRSM in the traditional layout and the compact layout. Finally, we describe compact batched BLAS/LAPACK implementation in two libraries: an open source implementation, Kokkos Kernels, and a vendor library, Intel® Math Kernel Library® (Intel MKL).

3.1 Compact Data Layout

To illustrate the compact layout, consider a collection of V·P matrices A, each having the same size, with which we need to perform some BLAS operation. Here P is a positive integer and V is the SIMD vector length of the underlying hardware; e.g. for an AVX512 machine in double precision V = 8. We identify element (i, j) of matrix m by A(m, i, j).

The compact layout is most easily understood as a modified 3D tensor. First, consider a collection of V·P matrices as a 3D tensor. Organizing the data such that m is the fastest index makes vectorization natural even for a very small matrix size. We can replace a scalar operation, e.g. a multiply and add operation,

$$C(m, i, j) \mathrel{+}= B(m, k, j) \times A(m, i, k),$$

by a vector operation,

$$\begin{aligned} C(m{:}m{+}V{-}1, i, j) \mathrel{+}= {} & B(m{:}m{+}V{-}1, k, j) \\ & \times A(m{:}m{+}V{-}1, i, k). \end{aligned}$$

where C(m:m+V-1, i, j) refers to the $(i,j)^{th}$ element of V matrices stored contiguously in memory. Here += and × are applied elementwise. The vector registers can be filled completely if V is equal to the vector register length (SIMD width).

Notice, however, that as P grows large the distance in memory between elements of an individual matrix grows large, penalizing spatial locality. To remedy this the compact layout organizes the matrices in a packed data structure (*packs*) whose length is given by the vector length V. Specifically, the packs

$$A(nV{:}(n{+}1)V{-}1, :, :), \quad n \in \{0, \ldots, P{-}1\},$$

are each individually organized as 3D tensors, again with the matrix number m as the fastest index. We illustrate the layout for four 3×3 matrices with V=2 in Figure 1. Pack n + 1 is stored subsequently to pack n in memory, for each n. This layout fills SIMD vectors for each instruction in our BLAS/LAPACK kernel while minimizing the distance in memory between elements of the same matrix. In the next section we will discuss the implementation of optimized GEMM and TRSM kernels with the compact layout and compare these with their standard implementations.

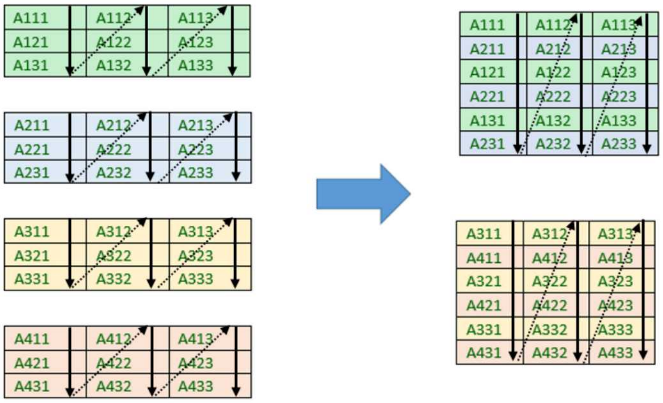

Figure 1: Illustration of compact data layout: four 3×3 matrices in packs of length two.

3.2 Implementation

To focus on the details related to the compact layout we will compare kernels written for simplified GEMM and TRSM operations. In particular we will ignore matrix strides and scaling operations, as these do not affect the basic strategy of optimizing the inner kernel for compact BLAS/LAPACK routines. For GEMM we consider only the non-transpose non-transpose case, and for TRSM we will isolate the left, lower, non-transpose, non-unit diagonal case. To avoid confusion, we separate notation for matrices A, B and C stored in column major layout from a collection of matrices in compact layout, Ac, Bc and Cc. All matrices have real, double precision. Let M, N, and K be positive integers. A and Ac(m, :, :) have M rows and K columns, where m ∈ {1, . . . , V×P}; for brevity, we write this size as M×K. Similarly, B and Bc(m, :, :) are K×N, and C and Cc(m, :, :) are M×N.

We begin by considering the GEMM operation

$$C \mathrel{+}= A \times B,$$

where here × denotes matrix-matrix multiplication. The simplified non-transpose non-transpose matrix multiplication is given in Algorithm 1.

Methods for optimizing GEMM and other level 3 BLAS operations for large matrix sizes are well understood [10], [11]. It is outside the scope of this paper to review all of the techniques involved in obtaining high performance for large GEMMS. Instead, we briefly

Algorithm 1: Simplified non-transpose, non-transpose GEMM

```
for $j$ in $\{0, \cdots, N-1\}$ do
    for $i$ in $\{0, \cdots, M-1\}$ do
        for $k$ in $\{0, \cdots, K-1\}$ do
            C(i,j) += A(i,k) * B(k,j)
```

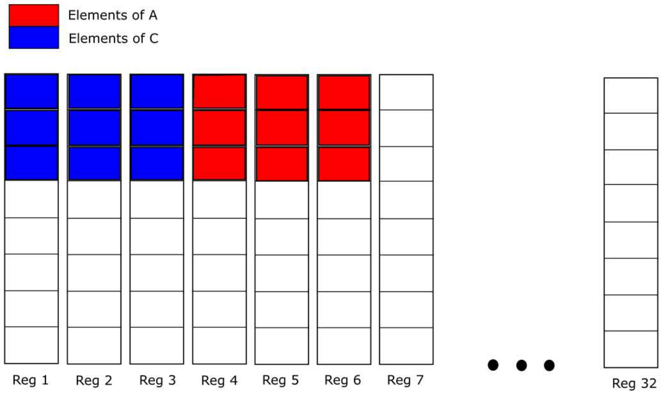

Figure 2: AVX512 register use in 3×3×3 DGEMM.

review the design of a GEMM kernel for a 3×3×3 DGEMM on a KNL. First we load columns of C and A into vector registers. For the 3×3×3 problem, the entire A and C matrices can be loaded into registers. Figure 2 illustrates the resulting fill of the vector registers. In a non-transpose non-transpose matrix multiplication, columns of A are scaled by elements of B, and the results are added to columns of C. So, we will not work with vectors of B, but rather we will crawl over B, broadcast elements to vectors and then perform vector FMAs. A nice feature of the FMA instruction on Intel's AVX512 architecture is the ability to take a memory address as an operand; the FMA instruction performs an implicit broadcast. Thus we do not need to explicitly broadcast elements of B into registers before performing FMAs.

Notice in Figure 2 that every vector instruction will need to be masked since the A and C registers are only partially filled with 3 out of 8 packed elements for an AVX512 machine in double precision (or worse, 3 out of 16 packed elements in single precision). There are two obvious performance limitations here. First, our theoretical peak is limited to 3/8ths of the core's peak simply due to the low vector fill. Second, masked FMAs have a higher latency than non-masked FMAs. The figure illustrates that a fundamental problem with very small BLAS operations is that we don't have enough data to make full use of the CPU core.

We turn our attention now to the collection of matrices stored in compact format and consider the reference compact GEMM algorithm given in Algorithm 2, where we isolate a single pack of V matrices for clarity of exposition. The key difference is that in the inner-most loop we are performing the same operation on the same indices of the Ac, Bc and Cc matrices, changing only the matrix index. Since we have stored the matrices in packs of length V within which the matrix number is the fastest index, we can use only vector instructions – loads, stores and FMAs – with no masks. Additionally, the broadcast of B elements is replaced by simple loads since we are performing abstractly scalar operations.

Algorithm 2: Reference compact GEMM kernel

```
for $j$ in $\{0, \cdots, N-1\}$ do
    for $i$ in $\{0, \cdots, M-1\}$ do
        for $k$ in $\{0, \cdots, K-1\}$ do
            for $p$ in $\{0, \cdots, V-1\}$ do
                Cc(p,i,j) += Ac(p,i,k) * Bc(p,k,j)
```

Algorithm 3: Simplified left, lower, non-transpose, non-unit diagonal TRSM kernel

```
for $j$ in $\{0, \cdots, N-1\}$ do
    for $i$ in $\{0, \cdots, M-1\}$ do
        B(i,j) /= A(i,i) for $ii$ in $\{i+1, \cdots, M-1\}$ do
            B(ii,j) -= B(i,j) * A(ii,i)
```

The story is even more dramatic for TRSM. Recall that our simplified TRSM operation solves the equation

$$AX = B$$

for X, where A is a lower triangular M×M matrix and X and B are M×N matrices. TRSM overwrites B with the solution X. For matrices stored in standard column major format, this operation is given in Algorithm 3. We see that there are two main components of the TRSM operation: a divide step to solve for element B(i,j) at the top of the i-loop, and then a forward substitution step. A typical algorithmic strategy for optimizing this operation for large sizes is to (i) thread over the j-index and (ii) block over the i and ii loops so that the forward substitution can be written as a GEMM call. For large sizes the cost of the i and ii blocked TRSM that occurs before the ensuing GEMM call is relatively small compared to the performance of the large GEMM, and an optimized library can obtain performance that is within 80% of GEMM performance for the same sizes. The blocking of the i and ii indexes can be sized to ensure the GEMM in the substitution is properly aligned to memory boundaries. However, for very small problems we cannot benefit from the performance of a GEMM-based forward substitution. To make matters worse there is absolutely no opportunity for vectorization of the initial divides. Further, the divides are followed by substitutions which cannot be aligned properly to memory boundaries since the beginning of the forward substitution increments with each i index increment.

If we work on our V·P matrices in compact format, we can improve upon this situation dramatically. Consider the reference compact TRSM algorithm presented in Algorithm 4, again isolating a single pack for clarity of exposition.

Notice that we can again replace every operation with a vector instruction, including the division. Also, memory boundaries are determined by the location of the first element of a pack of matrices and the pack length, resulting in aligned operations regardless of the i or ii index.

3.2.1 *Open Source Impl. - Kokkos Kernels.*

KokkosKernels is a new package built on top of Kokkos providing a collection of kernel

Algorithm 4: Reference compact left, lower, non-transpose, non-unit diagonal TRSM kernel

```
for $j$ in $\{0, \cdots, N-1\}$ do
    for $i$ in $\{0, \cdots, M-1\}$ do
        for $p$ in $\{0, \cdots, V-1\}$ do
            B(p,i,j) /= A(p,i,i)
        for $ii$ in $\{i+1, \cdots, M-1\}$ do
            for $p$ in $\{0, \cdots, V-1\}$ do
                B(p,ii,j) -= B(p,i,j) * A(p,ii,i)
```

algorithms that are commonly used in scientific applications. It operates within fine-grained parallelism on diverse architectures. Following the parallel abstractions in Kokkos, our computational kernels support the following interfaces:

- *serial* - a single thread is used in the kernel;
- *team* - a team of threads are cooperatively used in parallel
- *procedure* - the entire execution space is used in parallel.

In this work, we focus on the implementation of the serial interface that is used in `parallel for`, and we follow Kokkos' parallel loop scheduling and thread mapping.

SIMD requires aligned data which is packed contiguously along the vector length. Our compact data layout allows aligned memory access, but this might lead our implementation to be hardware specific. To make our code portable, we use a template vector data type encapsulating vector registers with arithmetic operator overloading. This allows us to reuse scalar BLAS/LAPACK algorithms with the vector data type, which also means that a good scalar code is more likely to be a good vectorized code. We follow the practice described in [14, 23]. Their core idea is to use a highly optimized architecture specific micro kernel around the loop packing and blocking data for the kernel according to the architecture cache hierarchy. In our work, we use a 4×4 rank-4 update as our inner kernel. Implemented using the vector data type, the inner kernel fully exploits SIMD instructions. In the case of the AVX512 architecture, the update becomes a $4 \times 4 \times 8$ vectorized rank-4 update. As we target small matrices, loop unrolling in this kernel is enough to achieve high performance.

3.2.2 Vendor Library - Intel Math Kernel Library. Intel MKL 2018 introduces compact batched GEMM, TRSM, and non-pivoting, incomplete LU. Intel MKL's initial implementation uses the compact layout and techniques described earlier in this section, loop unrolling, and compiler intrinsics to achieve performant kernels.

4 PERFORMANCE

We present performance results on the second generation Intel Xeon Phi 7250, code-named Knights Landing. The processor consists of 34 tiles interconnected by a two-dimensional mesh. Each tile comprises two four-way threaded cores running at 1.40GHz with 1MB of shared L2 cache. The processor core of Knights Landing has a private 32KB L1 data cache and two AVX512 vector units per core. The processor is equipped with 16GB of MCDRAM that provides approximately 480 GB/s of STREAM Triad bandwidth. All benchmark codes used in this section are compiled using the Intel compiler 17.0.1 with `-O3 -g` options. First we evaluate the performance of the compact data layout and implementations in synthetic benchmarks against OpenMP loops around BLAS/LAPACK kernels, the standard batched BLAS/LAPACK, and libxsmm. We then consider a line preconditioner application.

4.1 Batched BLAS/LAPACK

In this section we evaluate the performance of the compact batched BLAS/LAPACK implementations for `dgemm`, `dtrsm` and `dgetrf` by comparing the performance to the standard batched BLAS implementation (where available) as well as the use of a `parallel for` around the respective BLAS function. This section provides a view into the performance of the individual BLAS/LAPACK functions that will be evaluated in concert in the context of block tridiagonal factorization in the next section.

For each function, we evaluate the performance for square matrices with sizes 3, 5, 10, and 15, with a batch size of 16384. These sizes were chosen in collaboration with domain experts for our motivating application, but also provide a window into a variety of small sizes with wide applicability in high-performance computing.

Figures 3 and 6 evaluate the performance of the compact layout for `dgemm`. In Figure 3 we compare the Intel MKL (MKL Compact) and KokkosKernels (KokkosKernels) compact implementations with the standard Intel MKL Batched `dgemm` (MKL Batch), an OpenMP loop around Intel MKL `dgemm` (MKL OpenMP) calls, and to an OpenMP loop around libxsmm (libxsmm) library calls for small matrix multiplications. For sizes 3, 5, and 10 we see considerable performance improvement from both compact implementations over the other methods. For size 15, the performance of libxsmm is comparable, although we note no such library exists for BLAS functions other than `dgemm`, and libxsmm shows much weaker performance than the batched compact functions for smaller sizes. In Figure 6 we present a heatmap showing the speedups of libxsmm, the Intel MKL Batched `dgemm`, and the Intel MKL compact `dgemm` routines over an OpenMP loop around Intel MKL `dgemm` calls with 68 threads. We see that while the standard batched `dgemm` approach provides 1.1-2.2× improvements and the libxsmm approach provides 1.4-3.6× improvements the compact implementation provides up to 13.8× improvements over the OpenMP loop strategy.

Figures 4 and 7 evaluate the performance of the compact layout for `dtrsm`. In Figure 4 we compare the Intel MKL (MKL Compact) and KokkosKernels (KokkosKernels) compact implementations with the standard Intel MKL Batched `dtrsm` (MKL Batch) and an OpenMP loop around Intel MKL `dtrsm` (MKL OpenMP) calls. In this case we see very large improvements for the compact implementations for all sizes and all core counts. In Figure 7 we present a heatmap showing the speedups of the Intel MKL Batched `dtrsm` and the Intel MKL compact `dtrsm` routines over an OpenMP loop around Intel MKL `dtrsm` calls with 68 threads. We see that while the standard batched `dtrsm` approach provides 1.2-1.6× improvements over the OpenMP strategy, we see up to 45.2× improvements with the compact layout.

Finally, Figures 5 and 8 evaluates the performance of the compact layout for `dgetrf` with no pivoting. In Figure 5 we compare the Intel MKL (MKL Compact) and KokkosKernels (KokkosKernels) compact implementations with an OpenMP loop around Intel MKL `dgetrf` (MKL OpenMP) calls. In this case we do not compare against a

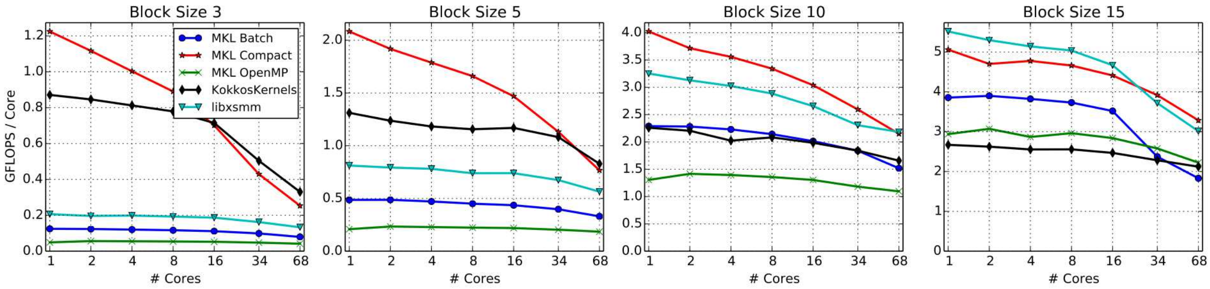

Figure 3: Batched `dgemm` performance with a batch size $N = 16384$ on Intel KNL 7250.

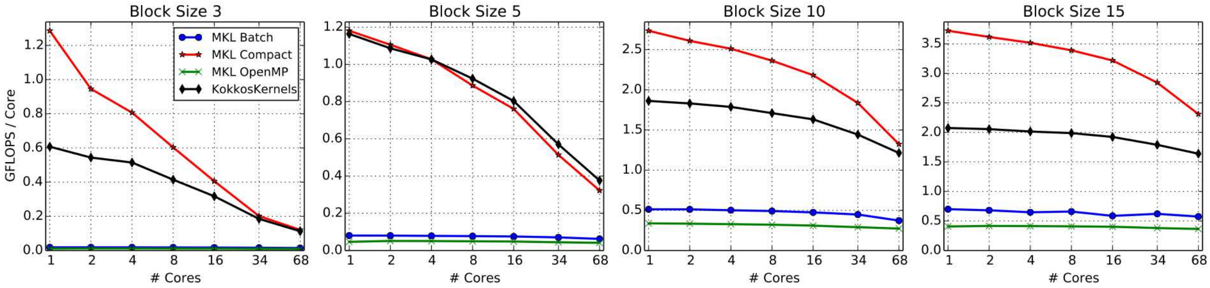

Figure 4: Batched `dtrsm` performance with a batch size $N = 16384$ on Intel KNL 7250.

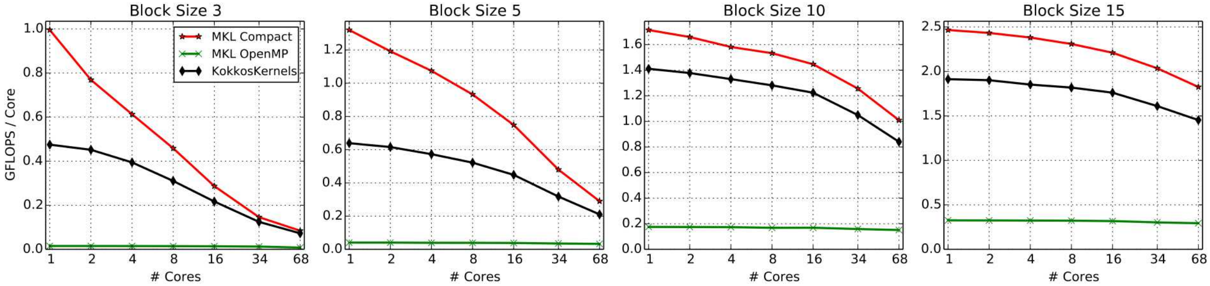

Figure 5: Batched `dgetrf` performance with a batch size $N = 16384$ on Intel KNL 7250.

standard batched implementation for `dgetrf` as this function is not available in a batched implementation in the Intel MKL. Similar to the `dtrsm` case we see extremely large speedups for all sizes and core counts for compact `dgetrf`. In Figure 8 we present a heatmap showing the speedup of the Intel MKL compact `dgetrf` implementation over an OpenMP loop around Intel MKL `dgetrf` calls with 68 threads. We see that the compact layout provides 1.7-27.4× improvements for the `dgetrf` function depending on matrix and batch sizes.

4.2 Line Preconditioner

We consider a block sparse system of equations $Ax = b$ arising from coupled PDEs. The problem is discretized on a domain depicted in Fig. 9 and lines are extracted along the k dimension. The standard stationary iterative procedure is applied for preconditioning the problem by splitting $A = M - N$, where M consists of block tridiagonal matrices corresponding to the extracted lines of elements. At the solution, $Ax = b$; hence

$$\begin{aligned} Mx &= b + Nx \\ x &= M^{-1}(b + Nx) \\ x &= x + M^{-1}(b - Ax)\,. \end{aligned}$$

The final line suggests the iteration

$$x^{k+1} = x^k + M^{-1}\left(b - Ax^k\right).$$

Alternatively, M may be used as a preconditioner in a Krylov subspace method, such as GMRES or CG. In either approach, M is factorized once per solution of $Ax = b$, and its factorization is applied once per iteration of the stationary or Krylov subspace method. There are algorithmic variants for parallel tridiagonal solvers mostly

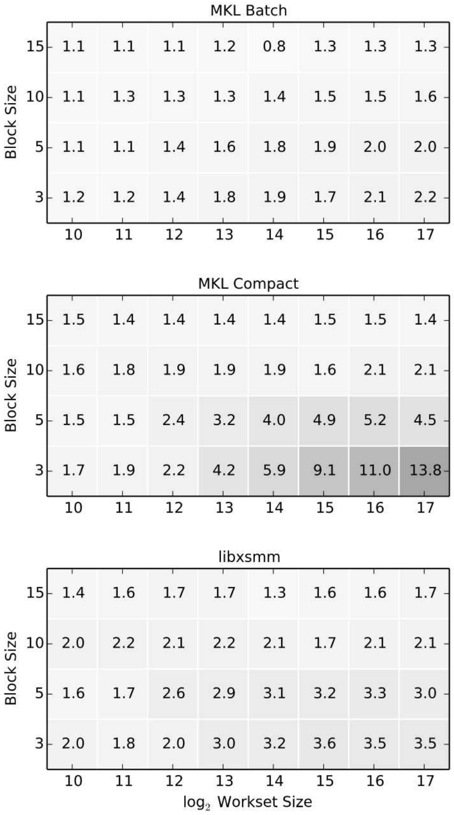

Figure 6: Speedup of compact batched `dgemm` **over OpenMP loops around** `dgemm` **with $N = 128^2$ on Intel KNL 7250.**

Algorithm 5: Block tridiagonal LU factorization

```
for $T$ in $\{T_0, T_1, \cdots, T_{m\times n-1}\}$ do in parallel
    for $r \leftarrow 0$ to $k-2$ do
        $\hat{A}^r := LU(\hat{A}^r)$;
        $\hat{B}^r := L^{-1}\hat{B}^r$;
        $\hat{C}^r := \hat{C}^r U^{-1}$;
        $\hat{A}^{r+1} := \hat{A}^{r+1} - \hat{C}^r\hat{B}^r$;
    $\hat{A}^{k-1} := LU(\hat{A}^{k-1})$;
```

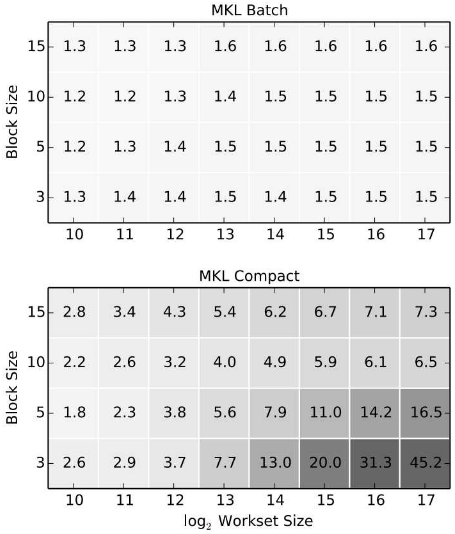

Figure 7: Speedup of compact batched `dtrsm` **over OpenMP loops around** `dtrsm` **with $N = 128^2$ on Intel KNL 7250.**

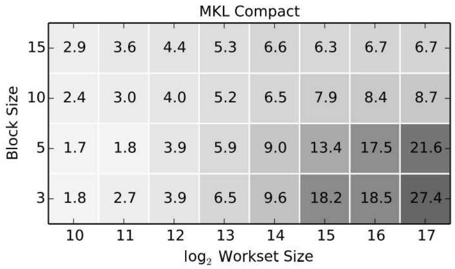

Figure 8: Speedup of compact batched `dgetrf` **over OpenMP loops around** `dgetrf` **with $N = 128^2$ on Intel KNL 7250.**

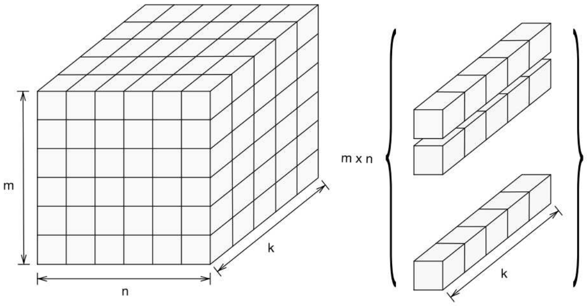

Figure 9: Left: discretization on a cubic domain. Right: lines of elements extracted in the k dimension.

based on the divide-and-conquer methods [21]. In this study, we do not consider the parallel tridiagonal factorization but use a sequential algorithm solving many tridiagonal systems within `parallel for`.

Block tridiagonal matrices are extracted and packed in the compact data layout from the block sparse matrix A as illustrated in Fig. 10. Then, an LU factorization is applied to those block tridiagonal matrices according to Alg. 5 in a parallel batch. There are two important aspects of this batched tridiagonal factorization. As the blocks are dense, the performance of this setup phase largely depends on efficient use of level 3 BLAS/LAPACK functions. In particular, we evaluate the code on the small block sizes $n_b = 3$, 5, 10 and 15. These small block sizes are typical in scientific applications, as we previously described. Additionally, the batched block tridiagonal factorization requires application of a sequence of BLAS/LAPACK operations along a block tridiagonal matrix. The

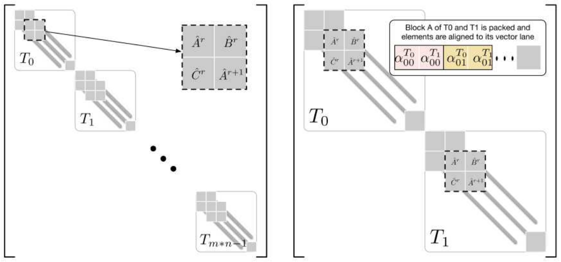
T_0
$\hat{A}^r$ $\hat{B}^r$
$\hat{C}^r$ $\hat{A}^{r+1}$
T_1
T_{m*n-1}
Block A of T0 and T1 is packed and elements are aligned to its vector lane
$\alpha_{00}^{T_0}$ $\alpha_{00}^{T_1}$ $\alpha_{01}^{T_0}$ $\alpha_{01}^{T_1}$ ...
T_0
T_1

Figure 10: Left: block tridiagonal matrices. Right: illustration of compact data layout with a vector length 2.

standard batched BLAS/LAPACK interface significantly limits the performance as it loses data locality after a single batched operation sweeps over blocks. In our proposed APIs, we expose the short and packed batch interface aligned to the hardware vector length. This provides building blocks for us to efficiently compose a new batched functions that can be used in `parallel for`.

Fig. 11 shows the performance of our implementation using the compact layout compared against the hand-tuned reference implementation on the Intel KNL. As `gemv` and `trsv` are not available from our industry partner, we do not evaluate the vendor library version for the solve phase. It is worth noting that we use the MCDRAM on the KNL as cache. In modern software design, application codes use modules to improve software productivity and our line preconditioner is also provided as a part of a solver module. In this context, relatively small HBM memory is shared with other components and using MCDRAM as cache is the most plausible testing environment. We also assume that applications decompose the problem domain so that each computing node can hold a block sparse matrix that fits into the HBM. Thus, we use the 128×128×128 mesh for blocks $n_b = 3, 5$ and $64 \times 64 \times 128$ mesh is used for blocks $n_b = 10, 15$. This setup generates problems ranging between 5.2 million and 10.5 million unknowns on each node.

We compare our vectorized implementation of the preconditioner based on the compact data layout against SPARC - a massively parallel Computational Fluid Dynamics (CFD) code. This code is developed and maintained by Sandia National Laboratories. SPARC has a straightforward implementation of the line smoother. It does not use a compact layout, and it relies on the compiler to vectorize loops. It uses a template specialization for block sizes of primary interest. In this reference study, we have used specializations for $n_b = 5, 15$ but not for $n_b = 3, 10$.

As shown in the figure, substantial performance improvements – 6.03×, 2.23×, 11.42× and 5.8× speed-up for block sizes 3, 5, 10 and 15 respectively in the factor step – are obtained by vectorizing the code with the compact data layout compared to hand-tuned version of the code with the conventional data layout. In particular, our code shows significant speed-up for small block matrices, which are considered difficult to solve efficiently using conventional optimization techniques.

4.3 Performance Analysis

In order to better understand the performance of the various BLAS kernel implementations on the Knights Landing we use the recently developed APEX application characterization tool described in [12]. APEX is a customized Intel PIN [16] tool which performs dynamic instruction, memory operation, arithmetic, control flow and logical operation analysis on executing multi-threaded binaries in order to extract low-level performance characteristics and operation counts. Several aspects of the Knights Landing core and the AVX512 vector units make instruction analysis of executing applications particularly challenging. These include: (1) the ability of the vector units to mask out lanes when performing memory, arithmetic and logical operations, thus affecting operation counting; (2) the significant increase in the capabilities of the AVX512 vector units in terms of operations and datatype support; (3) memory gather/scatter operations and, finally, (4) the increased parallelism available, placing pressure on any analysis tool to scale to significantly higher thread counts than previous Xeon-based processor designs. It is worth noting for the reader that the publicly exposed performance counters available on Knights Landing cores cannot provide the level of detail exposed in APEX or the level of detail required for the analysis described here.

The APEX toolkit is optimized in several ways to provide accurate operation counting at high speed to permit scaling to long duration executions and the size of application binaries used in the production computing environment at Sandia (typically hundreds of megabytes to gigabytes in size). The instrumentation of application instructions is performed in two potential modes – the first performs analysis of basic blocks within the application counting instructions which have no masking properties. We call these static operation counts as they will always execute the same number of operations when the instruction is executed. Each basic block is instrumented so that on entry it atomically increments the number of times it has been executed permitting tallies to be maintained quickly and across threads. The second class of instruction presents a greater challenge - these are instructions for which masking operations are used. For these operations an additional handler is installed prior to each instruction execution which traps the masking register/value being used and then performs a population count over the mask to count the number of active entries. We call these dynamic operations as the number of operations is a data dependent property and can change on each execution of the instruction. Complex arithmetic operations such as fused-multiply-adds/subtracts *etc.* are associated with multipliers to ensure the final operation tallies match the expectation of the application programmers. APEX can be considered to provide an optimistic approach to operation counting as there are possible uses of vector operations that can provide mask-equivalent operations that are not as easily tracked, but, in practice, we have found good correlation to algorithmic and hardware counters (where comparison is application) for a number of test cases that we have used from practical and contrived code examples during the tool's development. For the purposes of this analysis we provide a clear distinction between arithmetic floating-point operations from those which still utilize the vector units but do not perform mathematical operations (such as vector comparisons, logical operations and load/store operations), instead, we count each of these classes separately to ensure a more accurate representation of application behavior. An important aspect to the operation counts supplied here is that they are as instrumented and

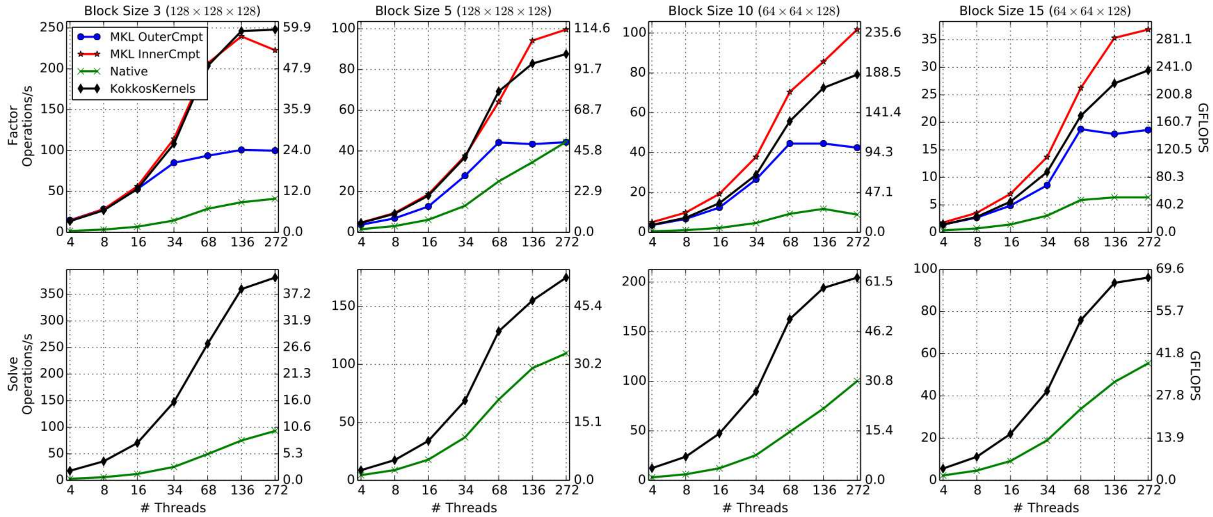

Figure 11: Parallel performance in the setup phase (top) and the solve phase (bottom) on Intel KNL 7250.

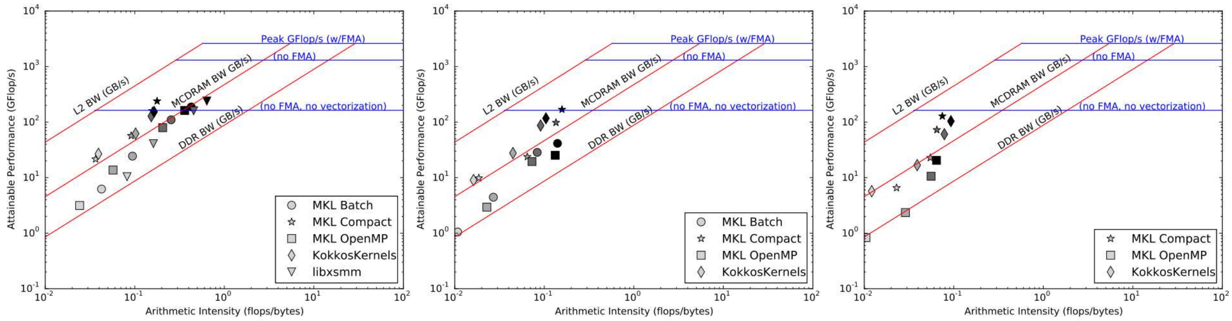

Figure 12: Roofline analysis for different methods for 3 kernels with 68 threads and $N = 16384$ on Intel KNL 7250. Each method is shown with a different marker, and increasing scale of grays are used for increasing block sizes (3, 5, 10 and 15).

Table 1: Average Double Precision Vector Utilization for different kernels with 68 threads and $N = 16384$ on Intel KNL 7250 (Maximum is 16 resulting from a 8–wide vector unit with FMA capabilities).

	DGEMM Methods					DGETRF Methods			DTRSM Methods			
Blk Size	MKL Compact	Kokkos Kernels	MKL Batch	MKL OpenMP	LIBXSMM	MKL Compact	Kokkos Kernels	MKL OpenMP	MKL Compact	Kokkos Kernels	MKL Batch	MKL OpenMP
3	15.87	12.96	10.30	10.30	9.99	12.28	12.80	1.00	15.68	13.13	6.44	6.24
5	15.97	14.34	12.40	12.40	11.99	13.42	13.42	2.18	13.64	15.39	8.89	8.74
10	15.99	15.14	14.43	14.43	15.01	14.70	14.72	3.65	14.65	15.93	10.66	10.61
15	15.99	15.41	14.94	14.94	15.87	15.15	15.17	4.94	15.05	15.98	12.44	12.42

perceived by the executing processor and are the subject of code generation and optimization, thus, differences between programmer estimates and final profile-based operation and instruction tallies are not uncommon once inlining, unrolling and vectorization (or the lack of vectorization) take place.

Using the APEX Knights Landing analysis tool, we have been able to capture a broad range of low-level application behavior metrics. By using a subset of these, we have been able to capture the average vector utilization of different kernels and formulate Roofline Model [25] diagrams of kernel behavior (see Figure 12) to show how the various kernel implementations utilize the hardware resources of the Knights Landing processor.

Table 1 gives the average vector lane utilization per floating-point arithmetic instruction for the various implementations of DGEMM, DTRSM, and DGETRF. For the purposes of this analysis we define the vector utilization metric as the summation of all floating point arithmetic operations (including those which execute as scalar (*i.e.* they execute in the 0th lane of the vector unit, as well as with and without masking applied) divided by the number of instructions which execute floating point arithmetic. While we include legacy X87-based instructions in this count, these are virtually never generated by modern Intel compilers and so can be considered to be either zero or an insignificant part of the operation count. Each kernel can have a maximum utilization of at most 16 double precision floating point operations per arithmetic instruction which would result from 8 double-precision vector lanes and the ability to perform a fused-multiply-add on each lane (giving 2 operations per vector lane per instruction). MKL Compact and KokkosKernels achieve the best vector utilization overall. The vector utilization of all methods is higher with increasing block size as there is an increase in operands available to pack into each vector instruction. While all methods achieve decent vector utilization for DGEMM, the compact kernels achieve close to 16 for all sizes, beating the other methods. However, utilization for the non-compact methods is much lower for DTRSM and DGETRF, consistent with our earlier analysis, while the compact layout allows the use of full vector operations for these more complicated functions. As a result, the performance difference for DTRSM and DGETRF is larger, as shown in Figure 4 and Figure 5. These results mostly correlate with the performance achieved, with the exception of the LIBXSMM performance which is able to provide strong performance at relatively lower levels of vector intensity. Although it has lower vector utilization, it has higher arithmetic intensity (indicating reduced load/store operations and higher register use) as described by the kernel's roofline model analysis.

The Roofline performance model diagrams of the three kernels are shown in Figure 12 when using 68 threads and $N = 16384$. In order to generate the arithmetic intensity used for these plots we profile all floating-point arithmetic operations/instructions (also required for the vector utilization metric above), as well as all load-/store/gather/scatter operations. Although APEX tracks data movement between registers, we explicitly exclude this in our arithmetic intensity calculation since we are interested in the bandwidth requirements and data movement across the processor. Movements between vector registers are exceptionally fast versus loads/stores even from local data caches to the point that we regard them as free in the context of investigating broader hardware performance and bottlenecks. The first conclusion to note is the strong correlation with MCDRAM performance for most smaller kernel executions which cluster either at or close to the MCDRAM bandwidth limit. Increasing block sizes help to push the kernel performance over the MCDRAM bandwidth line and closer to the L2 bandwidth limit as the increased number of operands permits higher vector utilization (permitting more efficient load/stores), and, allows the compiler and processor to keep a greater number of loaded values in registers or cache, thereby reducing cache/memory accesses and access times for each block, increasing achieved performance. For all kernels shown, the use of compact memory layouts (MKL compact and KokkosKernels) provides the highest performance which we attribute to the greater levels of efficiency resulting from full vector utilization, as well as, operation independence (since each function is performed on independent operands in each vector lane), reducing the overheads of managing kernel execution and allowing for very efficient load/stores of operands at full vector-widths. We argue that these effects combine to reduce the total number of instructions required to compute the kernels over all operands and provide the processor with a much more efficient instruction stream that presents itself as higher achieved performance.

Although we have gone to considerable lengths in our code design and the discussion in this paper to convince the reader of our increased use of SIMD operations (as reflecting in Table 1), the Roofline models may be interpreted as counter to this discussion. The reality is that the roofline limits (shown as blue lines/labels in our plots) show the peak the hardware is capable of in terms of instructions per second and clock rates in the absence of other micro-architectural bottlenecks or instruction mixes which contain non-floating point arithmetic operations or register/operand dependencies. The result of such effects is shown in our roofline plots and is routinely felt by application developers on modern hardware systems – that even well optimized, vectorized algorithms rarely achieve high fractions of computational peak because *useful* instruction sequences will almost always require dependencies between operands or a great deal of book-keeping and memory operations. The strong correlation of performance to MCDRAM bandwidth points to hope that future high-performance processors will continue to provide increases in both bandwidths and capacity so that our kernels will execute faster and that we will be able to increase the maximum problem size for which our approaches are profitable.

5 CONCLUSION

Small matrix problems suffer from reduced performance on modern hardware due to limited available parallelism and vectorization. In part this comes from these problems having a small number of operands that do not fill wide vector units. Data layout can also contribute significantly to lower performance, as unoptimized layouts for dense matrices can create high overhead when load/store operations only utilize fractions of a cache line or require gather/scatter operations to load operands. By batching together many small matrix operations, the community has addressed the limited available parallelism through batched BLAS/LAPACK. However, batched BLAS/LAPACK does not address the limited vectorization potential observed in small matrix problems, leading to limited core utilization in these routines.

In this paper we introduce a SIMD-friendly data layout for groups of small matrices, which packs matrices together to enable cross-matrix vectorization while maximizing spatial locality in BLAS/LAPACK kernels. We call the resulting methods the compact BLAS/LAPACK and focus in this paper on compact general matrix multiplication, triangular matrix solvers, and LU factorization with no pivoting. We describe the process of designing efficient compact BLAS/LAPACK kernels, while examining how the compact layout benefits performance for small matrices, focusing on GEMM and TRSM.

To motivate the method we consider a line solver for coupled partial differential equations. We demonstrate the performance of compact BLAS/LAPACK routines in an open-source reference (KokkosKernels) and a high-performance vendor implementation (Intel MKL) on the Intel Knights Landing architecture by first considering synthetic problems with sizes inspired by our application of interest. In these tests we see up to 14×, 45× and 27× speedup for compact DGEMM, DTRSM and DGETRF(no pivoting), respectively over OpenMP loops around hightly optimized DGEMM, DTRSM and DGETRF. We then demonstrate the performance of the compact BLAS/LAPACK through use in the line solver application. We compare full batch v inner kernels. Compared to a hand-tuned version of the line solver, we observe 2×-6× performance improvement. Finally, we perform a detailed analysis and comparison of processor core utilization in the compact BLAS/LAPACK implementation using the APEX analysis suite. This analysis shows significant increase in vector lane utilization for arithmetic operations (up to 30%) when using compact BLAS/LAPACK functions. The generation of Roofline Performance models using our analysis data shows the strong correlation of benchmark performance with high-bandwidth memory available on the Knights Landing as well as the higher arithmtic intensity of the compact kernels compared to batch and standard function calls. We conclude that the compact implementation of BLAS/LAPACK kernels makes much more efficient use of the computational resources of the processor cores available on leading high-performance processors.

Compact BLAS/LAPACK shows impressive performance potential for an important class of problems in HPC, machine learning, and elsewhere. As the community continues to push towards ever more aggressive processor designs to reach Exascale within an efficient energy budget, mathematics libraries, such as BLAS/LAPACK, will need to be rewritten to maximize their use of compute resources. In this paper we have demonstrated a high-performance path to providing BLAS/LAPACK functions for small matrices – a set of operations which have typically been challenging to optimize and which traditionally have performed considerably slower than their large-matrix kin.

Further statement about the importance of considering architecture in obtaining performance.

ACKNOWLEDGMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of EnergyâĂŹs National Nuclear Security Administration under contract DE-AC04-94AL85000.

The APEX Performance Analysis tools used in this paper were developed under Early-Career Laboratory Directed Research and Development funding from Sandia National Laboratories.

REFERENCES

[1] CUDA Toolkit Documentation. (????). http://docs.nvidia.com/cuda/cublas/index.html, last accessed Mar 2017.

[2] Workshop on Batched, Reproducible and Reduced Precision BLAS. (????). bit.ly/Batch-BLAS-2017, last accessed Mar 2017.

[3] Ahmad Abdelfattah, Marc Baboulin, Veselin Dobrev, Jack Dongarra, Christopher Earl, Joël Falcou, Azzam Haidar, Ian Karlin, Tz Kolev, Ian Masliah, and others. 2016. High-performance tensor contractions for GPUs. *Procedia Computer Science* 80 (2016), 108–118.

[4] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2016. Performance, design, and autotuning of batched GEMM for GPUs. In *International Conference on High Performance Computing*. Springer, 21–38.

[5] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and others. 1999. *LAPACK Users' guide*. SIAM.

[6] Graham V Candler, Heath B Johnson, Ioannis Nompelis, Vladimyr M Gidzak, Pramod K Subbareddy, and Michael Barnhardt. 2015. Development of the US3D Code for Advanced Compressible and Reacting Flow Simulations. In *53rd AIAA Aerospace Sciences Meeting*. 1893.

[7] Eric C Cyr, John N Shadid, and Raymond S Tuminaro. 2016. Teko: A block preconditioning capability with concrete example applications in Navier–Stokes and MHD. *SIAM Journal on Scientific Computing* 38, 5 (2016), S307–S331.

[8] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J Higham, Jonathon Hogg, Pedro Valero-Lara, Samuel D Relton, Stanimire Tomov, and others. 2016. A proposed API for batched basic linear algebra subprograms. (2016).

[9] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)* 16, 1 (1990), 1–17.

[10] Kazushige Goto and Robert A Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)* 34, 3 (2008), 12.

[11] Kazushige Goto and Robert Van De Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)* 35, 1 (2008), 4.

[12] S.D. Hammond. 2015. *Towards Accurate Application Characterization for Exascale (APEX)*. Technical Report SAND2015-8051. Sandia National Laboratories, NM, USA.

[13] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Press, Piscataway, NJ, USA, 84:1—-84:11. http://dl.acm.org/citation.cfm?id=3014904.3015017

[14] Jianyu Huang and Robert A Van˜de˜Geijn. 2016. *BLISlab: A Sandbox for Optimizing GEMM*. FLAME Working Note #80, TR-16-13. The University of Texas at Austin, Department of Computer Science. http://arxiv.org/pdf/1609.00076v1.pdf

[15] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)* 5, 3 (1979), 308–323.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 190–200.

[17] Samuel Relton and Mawussi Zounon. Batched BLAS API and Memory Layouts. (????). http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/Batched-BLAS-2017/talk02-relton.pdf, last accessed Mar 2017.

[18] Sudip K Seal, Kalyan S Perumalla, and Steven P Hirshman. 2013. Revisiting parallel cyclic reduction and parallel prefix-based algorithms for block tridiagonal systems of equations. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 273–280.

[19] Paul Segall and Andrew M Bradley. 2012. The role of thermal pressurization and dilatancy in controlling the rate of fault slip. *Journal of Applied Mechanics* 79, 3 (2012), 031013.

[20] John Shadid, Scott Hutchinson, Gary Hennigan, Harry Moffat, Karen Devine, and Andrew G Salinger. 1997. Efficient parallel computation of unstructured finite element reacting flow solutions. *Parallel Comput.* 23, 9 (1997), 1307–1325.

[21] Paul N Swarztrauber. 1977. The methods of cyclic reduction, Fourier analysis and the FACR algorithm for the discrete solution of PoissonâĂŹs equation on a rectangle. *Siam Review* 19, 3 (1977), 490–501.

[22] R Tuminaro, Mauro Perego, I Tezaur, A Salinger, and Stephen Price. 2016. A matrix dependent/algebraic multigrid approach for extruded meshes with applications to ice sheet modeling. *SIAM Journal on Scientific Computing* 38, 5 (2016), C504–C532.

[23] Field G Van˜Zee and Robert A Van˜de˜Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Trans. Math. Software* 41, 3 (jun 2015), 14:1—-14:33. http://doi.acm.org/10.1145/2764454

[24] Hilary Weller, Sarah-Jane Lock, and Nigel Wood. 2013. Runge–Kutta IMEX schemes for the horizontally explicit/vertically implicit (HEVI) solution of wave equations. *J. Comput. Phys.* 252 (2013), 365–381.

[25] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[26] Michael J Wright, Graham V Candler, and Deepak Bose. 1998. Data-parallel line relaxation method for the Navier-Stokes equations. *AIAA journal* 36, 9 (1998), 1603–1609.

[27] Zhang Zhang. Introducing Batch GEMM Operations. (????). https://software.intel.com/en-us/articles/introducing-batch-gemm-operations

A ARTIFACT DESCRIPTION: [DESIGNING VECTOR-FRIENDLY COMPACT BATCHED BLAS AND LAPACK KERNELS]

A.1 Abstract

We describe the artifacts associated with the compact batched BLAS/LAPACKin this appendix. We describe the computational environment, software and testing methodology used in the experiments in detail. With the right hardware environment and software libraries listed here it should be straightforward for a user to replicate the results in this paper.

A.2 Description

A.2.1 Check-list (artifact meta information). Fill in whatever is applicable with some informal keywords and remove the rest

- **Algorithm:** DGEMM, DTRSM, DGETRF, **and** BCRS
- **Compilation: Intel C++ compiler icpc version 17.0.1, Intel MKL 2018**
- **Binary:**
- **Data set: Generated within the tests.**
- **Run-time environment: Intel MKL, Kokkos, Memkind, Libxsmm**
- **Hardware: Intel Knights Landing processors**
- **Execution: Scripts provided to repeat the runs in this paper**
- **Output: Block sizes, time, average and maximum floating point operations**
- **Experiment workflow: Script based workflow to run different kernels and the preconditioner**
- **Experiment customization: Parameters in scripts can be modified to adjust the block sizes, number of threads, workset sizes.**
- **Publicly available: Yes**

A.2.2 How software can be obtained (if available). We described a reference implementation and a vendor implementation of compact batched BLAS/LAPACK. The reference implementation is open source and publicly available[2]. The specific version of our code used in the experiments can be accessed with the SHA-id `8a8bb3547d...` in the git repository. The vendor version of our code is freely available to download as a library. The specific improvements described in this paper will be made publicly available in Intel MKL 2018.

A.2.3 Hardware dependencies. All the tests were run on Intel Knights Landing processors. The kernel test results used the processors in the “quad-flat” memory mode, which allows all the data to be stored in the 16GB high bandwidth memory with coherency directory lookups performed in the closest quadrant of the processor mesh. The preconditioner test results used the processors in the “quad-cache” mode where the data resides in the DDR memory and high bandwidth memory is used as a large direct-mapped cache. The “quad-cache” mode was used to accommodate the increased memory requirements in the preconditioner.

A.2.4 Software dependencies. The reference implementation depends on the open source Kokkos library[3]. The version of the Kokkos library we used can be accessed using the git SHA-id `b8bce49f5f7c...`. The reference implementation also depends on memkind (version 20160811) and Intel compilers (version 17.1.132).

[2] `https://github.com/kokkos/kokkos-kernels`

[3] `https://github.com/kokkos/kokkos`

The tests can also use the vendor version of the compact BLAS/LAPACK, libxsmm (version xxx.xxx) and Intel MKL 2017 for comparison purposes. The test scripts in the github repository can be used to test just the reference implementation without other comparisons.

A.2.5 Datasets. Our tests generate small block matrices of different block sizes and work set sizes. These matrices are used in the testing of our kernels. Our preconditioner tests generate block-diagonal matrices where block-diagonal is a tridiagonal matrix with small blocks for each entry in it. We use this matrix to evaluate the preconditioner creation and application. Data layout is an important factor in tests such as these. As our applications can switch to compact layouts with a small change (a template parameter) it is customary for them to store the data in a format that is best suitable for performance. Our tests store the input in the compact layout. Even if there was an application where it is not possible to store the data in compact layouts, the cost of allocation and copying into the compact data layout can be amortized over several preconditioner creation and solves. We avoid such expensive reformatting and store the data in compact layouts.

A.3 Installation

Installation of the reference implementation of compact batched BLAS/LAPACK uses a simple Makefile system. Users can provide the path to Kokkos installation and get the batched BLAS/LAPACKlibraries built. We build the Kokkos library with the configuration options "−with-openmp −with-serial −arch=KNL -with-options= aggressive_vectorization". All our code is compiled with `-O3 -g` options to the Intel C++ compiler. The vendor library is provided in binary form and linked to our tests.

A.4 Evaluation and expected result

The tests output the average and maximum GFLOPS/sec for XX iterations of the kernel. The preconditioner test output.

A.5 Experiment customization

All the experiments in the paper uses a technique called "cold cache". In each run, we flush the small matrices out of the cache by allocating and initializing a large dataset that flushes the cache. This is very conservative estimate of the performance of our kernels. In typical usage such as our line preconditioner the data resides in cache for different kernels due to dependencies. The standard way to run our tests still uses the cold cache mode. However, we provide an option to evaluate the kernels in "hot cache" mode. This options demonstrates the improved performance that can be achieved if the data is reused between different kernels. For example, DGEMM kernel with 68 threads will result in XXX GFLOP/thread for block size of BB and work set size of YY (See Figure 3). Users interested in hot cache approach could run our tests with "−hot cache" and evaluate the improved performance. For example, DGEMM kernel with 68 threads will result in XXX GFLOP/thread for block size of BB and work set size of YY with hot cache.

A.6 Notes

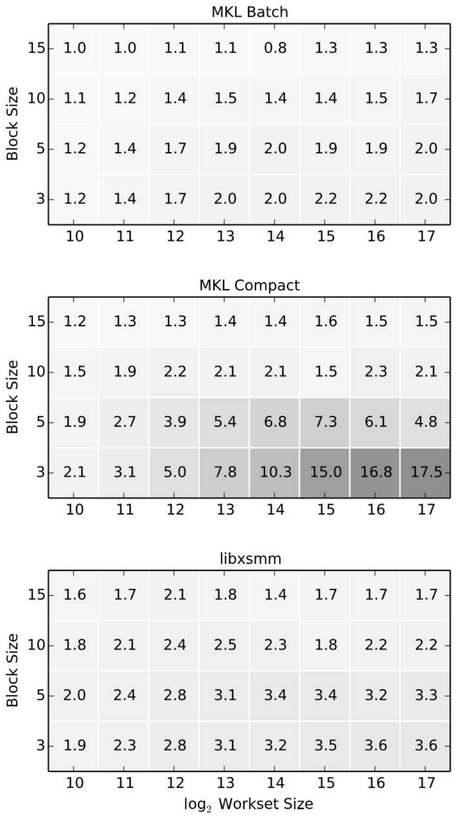

Figure 13: dgemm hot cache speedup heat map

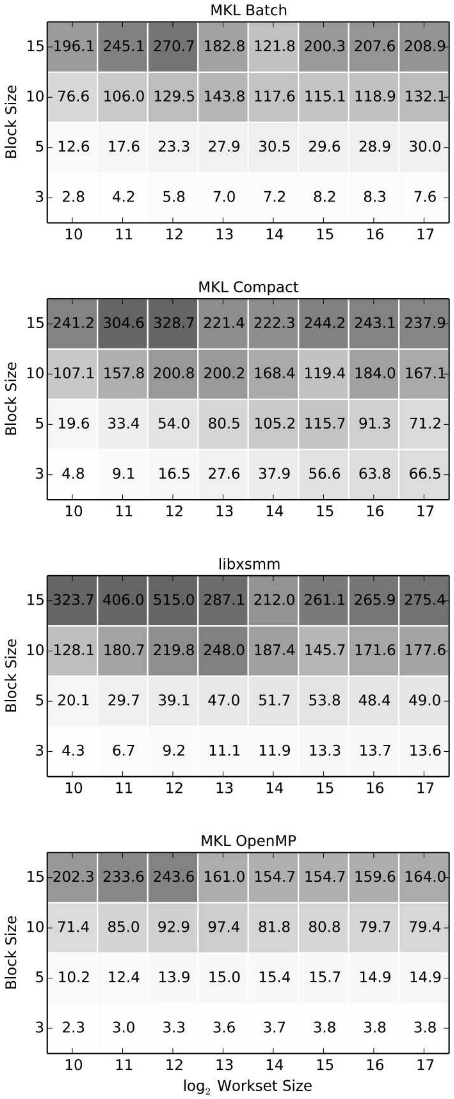

Figure 14: dgemm hot cache gflops heat map