SAND2017-12419C

Designing Vector-Friendly Compact BLAS and LAPACK Kernels

Kyungjoo Kim[1] Timothy B. Costa[2] Mehmet Deveci[1]
Andrew M. Bradley[1] Simon D. Hammond[1] Murat E. Guney[2]
Sarah Knepper[2] Shane Story[2] Sivasankaran Rajamanickam[1]

[1] *Center for Computing Research, Sandia National Labs*

[2] *Intel® Math Kernel Library Team, Intel Corporation*

SC17, Denver, CO

Introduction

- Vectorization
- Block Line Preconditioner
- Implementation Choices

Compact Data Layout

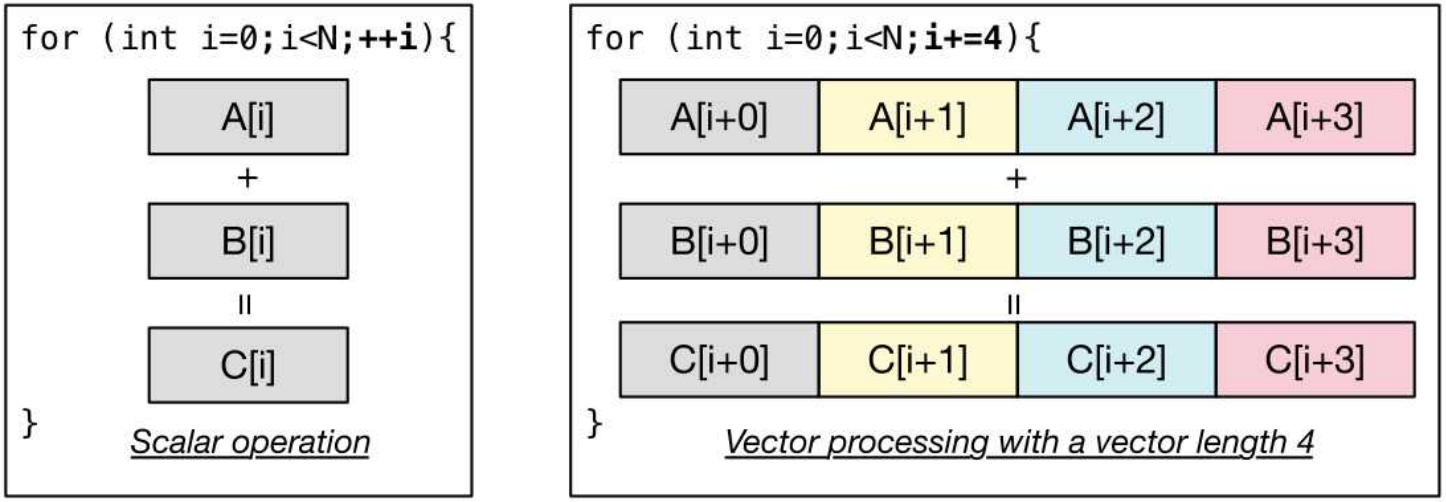
- Example of $3 \times 3 \times 3$ Matrix-Matrix Multiplication (GEMM)
- Compact Batched APIs
  - Intel MKL 18
  - KokkosKernels

Numerical Experiments

- Compact batched LU, TRSM, GEMM
- Roofline Analysis
- Block Line Preconditioner

Conclusion

What Is Vectorization ?



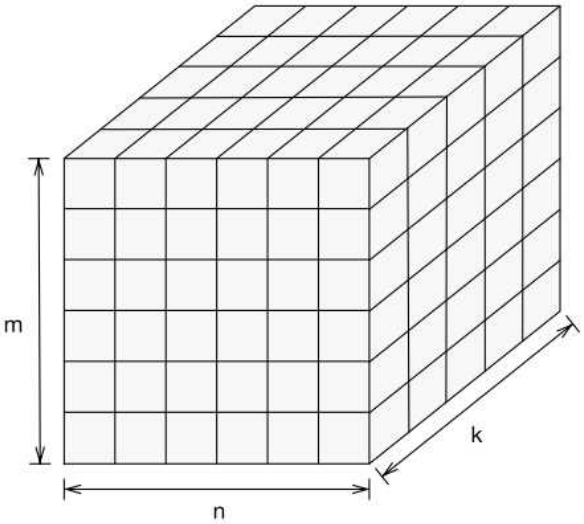
- Vectorization transforms a code with Single Instruction Multiple Data (SIMD) instructions, exploiting instruction-level parallelism.
- Significant speedup can be achieved by vectorizing a code with SIMD.
- Modern HPC systems use wide vector units to achieve peak performance.

How can we vectorize a code ?

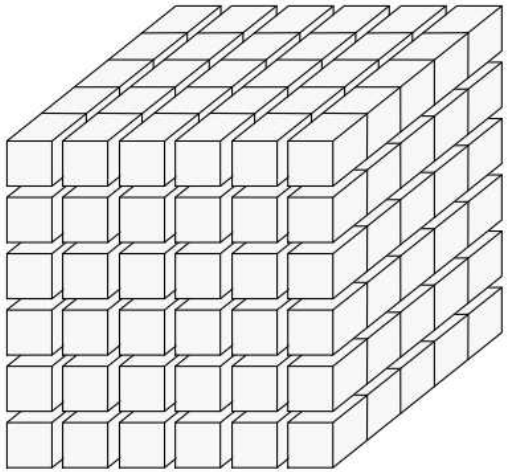
- A code can be auto-vectorized by a compiler, or developers can write a code with vector intrinsics or assembly kernels.
- *Developers (or code generators) should be able to express algorithms with fine-grained regular parallelsim.*

Problem: Block Line Preconditioner

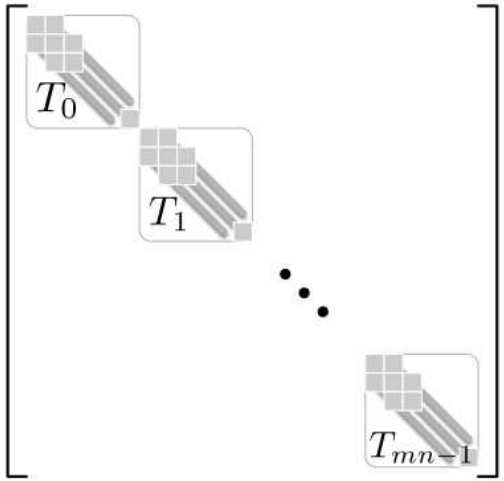
- Consider a block sparse system arising from coupled multi-physics problems.
- Line preconditioner is built by approximating the problem domain as a collection of lines of elements.
- A collection of lines of elements results in a set of block tridiagonal matrices.
- Block tridiagonal matrices are factorized once per solution (or every nonlinear iteration) and applied (triangular solve) multiple times.



Problem domain



Extracted line elements

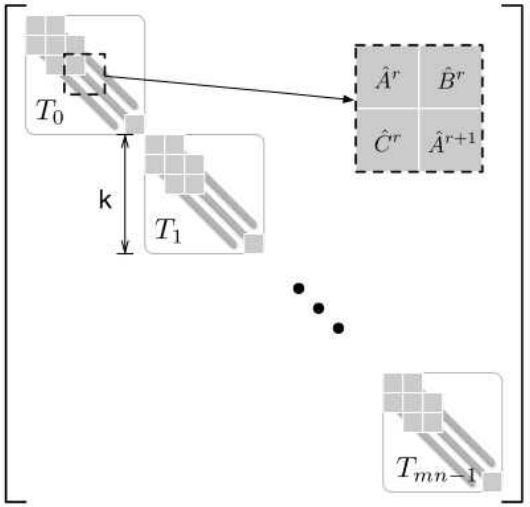


A set of block tridiagonal matrices

Problem Setup

- Typical blocksize b is selected as 3, 5, 9 and 15, which are related to scientific applications e.g., elasticity, ideal gas and multi-physics fluid problems.
- Limit memory usage up to 16 GB i.e., MCDRAM on KNL and GPU device memory.
- With this memory constraint, typical local problems ($m \times n \times k$) are selected as $128 \times 128 \times 128$ for $b = 3, 5$ and $64 \times 64 \times 128$ for $b = 10, 15$.

→ Batch parallelism is used running a sequential block tridiagonal factorization consisting of `GETRF`, `TRSM` and `GEMM` within `parallel_for`.



1 **for** T *in* $\{T_0, T_1, \cdots, T_{m\times n-1}\}$ **do in parallel**
2 **for** $r \leftarrow 0$ **to** $k-2$ **do**
3 $\hat{A}^r := LU(\hat{A}^r)$;
4 $\hat{B}^r := L^{-1}\hat{B}^r$;
5 $\hat{C}^r := \hat{C}^r U^{-1}$;
6 $\hat{A}^{r+1} := \hat{A}^{r+1} - \hat{C}^r \hat{B}^r$;
7 $\hat{A}^{k-1} := LU(\hat{A}^{k-1})$;

Line preconditioner setup with batch parallelism

Implementation Choices

- BLAS/LAPACK with OpenMP
- Batched BLAS/LAPACK
- Do-It-Yourself

Problems Using Dense Linear Algebra Libraries

Sandia National Laboratories

- BLAS/LAPACK with OpenMP

```
#pragma omp parallel for
for (i=0;i<m*n;++i) {
  for (r=0;i<k-1;++r) {
    getrf(A(i,r));
    trsm('L', A(i,r), B(i,r));
    trsm('U', A(i,r), C(i,r));
    gemm(C(i,r), B(i,r), A(i,r+1));
  }
  getrf(A(i,k-1));
}
```

  - BLAS/LAPACK is not optimized for such small problem sizes as 3, 5, 9 and 15.

- Batched BLAS/LAPACK

```
for (r=0;i<k-1;++r) {
  batch_getrf(A(:,r));
  batch_trsm('L', A(:,r), B(:,r));
  batch_trsm('U', A(:,r), C(:,r));
  batch_gemm(C(:,r), B(:,r), A(:,r+1));
}
batch_getrf(A(:,k-1));
```

  - Batched BLAS/LAPACK is designed to compute many dense problems in parallel.
  - The sequence of batched operations does not exploit temporal data locality.

Compact Data Layout

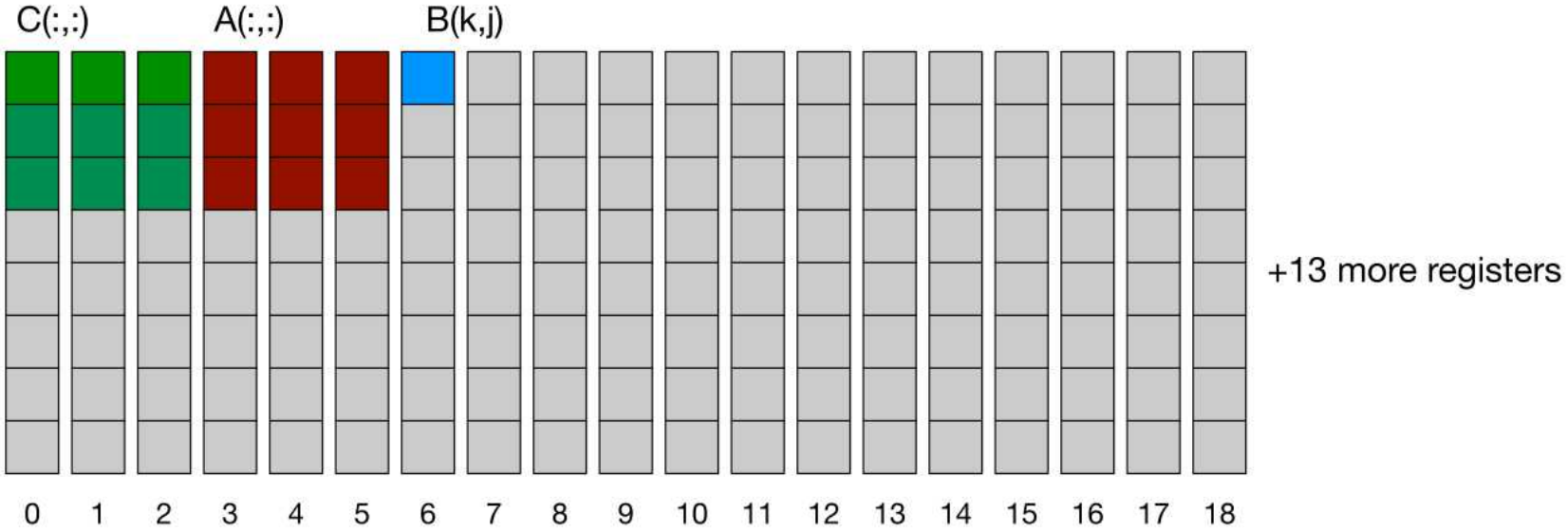
- Allows to exploit temporal locality in a sequence of batch calls.
- Efficiently uses SIMD units for small matrix computations.

Problem in Standard $3 \times 3 \times 3$ Matrix-Matrix Multiplication (GEMM)

- Matrices are stored in a standard column-major (row-major) order.

```
// C += A B;
for (int j=0;j<3;++j)
  for (int k=0;k<3;++k)
    // C(0:2,j) += A(0:2,k)*B(k,j)
    fused_mult_add(C(mask(0:2), j), A(mask(0:2), k), B(k, j));
```

- FLOP ($54 = 2 \cdot m \cdot n \cdot k$) per memory ops. (6 vector load with masks, 9 scalar load and 3 vector store with mask) is **3**.
- **Blocksize of interest (3,5,10,15) is too small to use wide vector units (AVX512) on KNL.**

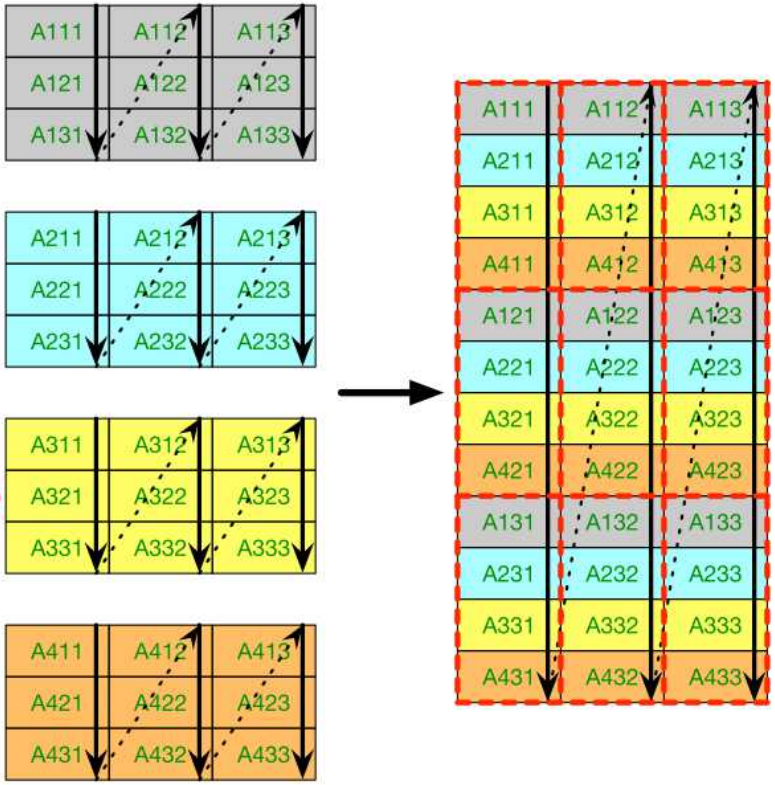


Register usage: A and C are unrolled and B is loaded elementwisely

Our Solution: Compact Data Layout (SIMD type)

- Modern computing architectures achieve peak performance through vectorization.
- Recall that our focus is on solving multiple problems in parallel.
- Compact data layout interleaves data across matrices.
- SIMD type becomes basic computing unit and all scalar operations are transformed to vector operations.

```
// computing unit
struct VectorAVX256D {
  union {
    __m256d v;
    double s[4];
  };
};

// overload arithmetic operators (+-*/)
VectorAVX256D
operator+(VectorAVX256D const &a,
          VectorAVX256D const &b) {
  return __mm256_add_pd(a, b);
}
```



Standard data layout

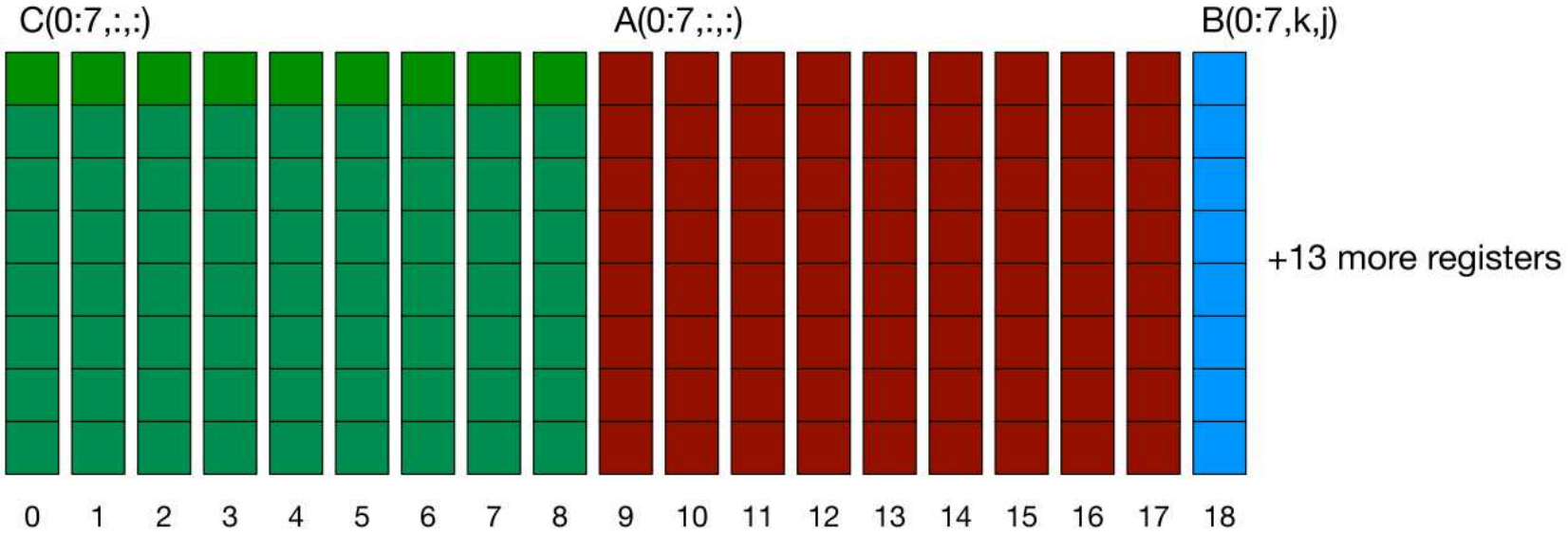
Compact data layout using vector length of 4

Compact $3 \times 3 \times 3$ Matrix-Matrix Multiplication (GEMM)

- Matrices are batched, with batch size the architecture vector length. In a batch, matrix entries are interleaved.

```
// batched C += A B;
for (int i=0;i<3;++i)
  for (int j=0;j<3;++j)
    for (int k=0;k<3;++k)
      // C(0:7,i,j) += A(0:7,j,k)*B(0:7,k,j)
      fused_mult_add(C(0:7, i, j), A(0:7, j, k), B(0:7, k, j));
```

- FLOP ($432 = 8 \cdot 2 \cdot m \cdot n \cdot k$) per memory ops. (27 vector load and 9 vector store) is **12**.
- **Code is purely vectorized**.



Register usage: packed matrix A and C is unrolled and packed B is loaded elementwisely

Compact Batched BLAS/LAPACK

Sandia National Laboratories

Intel® MKL 2018

- Compact BLAS/LAPACK APIs[1]:
  - `mkl_?gemm_compact` matrix-matrix multiplication,
  - `mkl_?trsm_compact` triangular matrix solve,
  - `mkl_?potrf_compact` Cholesky factorization,
  - `mkl_?getrfnp_compact` LU without pivoting,
  - `mkl_?geqrf_compact` QR, etc.
- Example of `mkl_?gemm_compact`:

```
mkl_?gemm_compact(// conventional BLAS interface
                  layout, transa, transb, m, n, k,
                  alpha, *ap, ldap, *bp, ldbp, beta, *cp, ldcp,
                  // compact format description
                  format, // MKL_COMPACT_{SSE/AVX/AVX512}
                  nm);    // # of matrices in compact format
```

- Single pack operation ($nm = 1$) can be used in "parallel for".
- New batch functionality can be efficiently composed by using compact BLAS/LAPACK e.g., batched block tridiagonal factorization.

1. software.intel.com/en-us/mkl-developer-reference-c-blas-and-lapack-compact-routines

Compact Batched BLAS/LAPACK

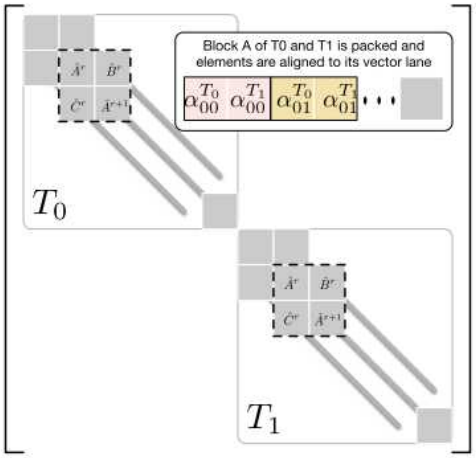
KokkosKernels[2]

- Provides portable C++ implementations of local computational kernels for linear algebra and graph operations using Kokkos shared-memory programming model.
- Layered interface i.e., serial(vector), team and device corresponding to hierarchical parallelism.
- Supports LU factorization without pivoting, TRSM and GEMM.

```
// SIMD type encapsulates vector storage and operations
class Vector<SIMD<T>, VectorLength>;

// multidimensional array, Kokkos::View, abstracts rank-3 packed matrices
// nm - # of matrices in compact format; m,n,k - matrix dimensions for gemm
Kokkos::View<Vector<SIMD<T> >***> A(nm, m, k), B(nm, k, n), C(nm, m, n);

// device: compose a batch operation using Kokkos parallel programming models
Kokkos::parallel_for(nm, KOKKOS_LAMBDA(int i) {
  // extract rank-2 array from input array of matrices
  auto Ac = Kokkos::subview(A, i, ALL, ALL);
  auto Bc = Kokkos::subview(B, i, ALL, ALL);
  auto Cc = Kokkos::subview(C, i, ALL, ALL);

  // serial: single pack interface using compact data format
  KokkosBatched::SerialGemm<TransA,TransB,AlgorithmTag>
               ::invoke(alpha, Ac, Bc, beta, Cc);
});
```

2. github.com/kokkos/kokkos-kernels/tree/master/src/batched

Line Preconditioner Impl. Using Compact Batched BLAS/LAPACK



1 **for** *a pair* T *in* $\{(T_0,T_1),(T_2,T_3),\cdots,(T_{m\times n-2},T_{m\times n-1})\}$ **do in parallel**
2 **for** $r \leftarrow 0$ **to** $k-2$ **do**
3 $\hat{A}^r := LU(\hat{A}^r)$;
4 $\hat{B}^r := L^{-1}\hat{B}^r$;
5 $\hat{C}^r := \hat{C}^r U^{-1}$;
6 $\hat{A}^{r+1} := \hat{A}^{r+1} - \hat{C}^r\hat{B}^r$;
7 $\hat{A}^{k-1} := LU(\hat{A}^{k-1})$;

Block tridiagonal factorization using compact batched BLAS/LAPACK packed with a vector length 2

Some Issues

- As it performs cross-matrix vectorization, pivoting in LU is not feasible.

→ For preconditioning, this does not matter.

- There is repacking overhead when the standard format is used.

→ Block tridiagonal matrices are extracted and repacked at the same time.

Numerical Experiments

Numerical Experiments

Sandia National Laboratories

Testbed: Intel Knights Landing

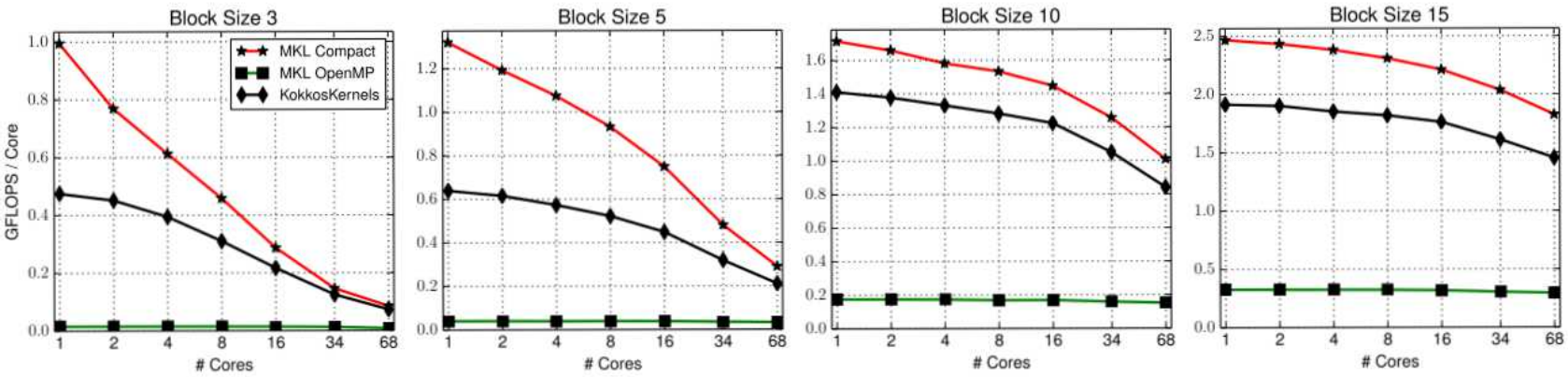
- 34 Tiles, 2 Cores/tile, 4 Threads/core, 2x **AVX512** units/core, 1MB L2
- 3+ TFLOPs in double precision, 400+ GB/s (MCDRAM)

Benchmark

- Compact batched LU, TRSM and GEMM are compared against
  - 1) MKL with OpenMP, 2) MKL batched APIs, 3) `libxsmm`[3]
- Roofline analysis on batched LU, TRSM, GEMM.
- Our impl. of block line preconditioner is compared with an optimized mini-app version of SPARC.
  - SPARC: Sandia production code for solving Navier-Stokes equations for compressible and reacting flows.

3. https://github.com/hfp/libxsmm

Batched LU

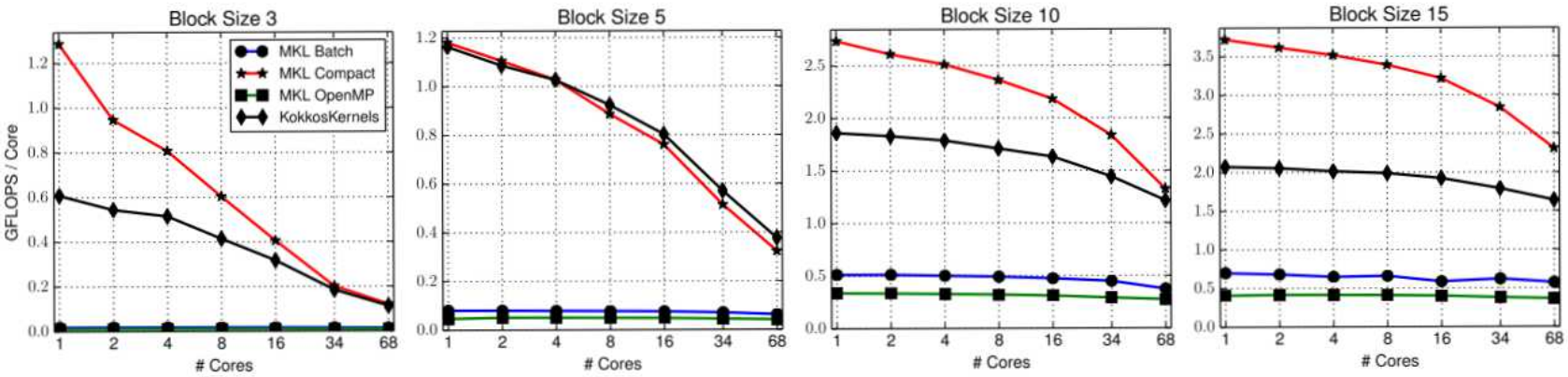


Comparison of compact batched LU against MKL DGETRF with OpenMP where the batch size (N) is 16384.

	Vector utilization			Speedup of MKL Compact	
Blocksize	MKL OpenMP	KokkosKernels	MKL Compact	1 thread	68 threads
3	1.00	12.80	12.28	67.61	7.88
5	2.18	13.42	13.42	34.49	9.69
10	3.65	14.72	14.70	10.01	6.84
15	4.94	15.17	15.15	7.64	6.24

Vector utilization (closer to 16 is better) with 68 threads and speedup against MKL `DGETRF` with OpenMP using 1 and 68 threads.

Batched TRSM

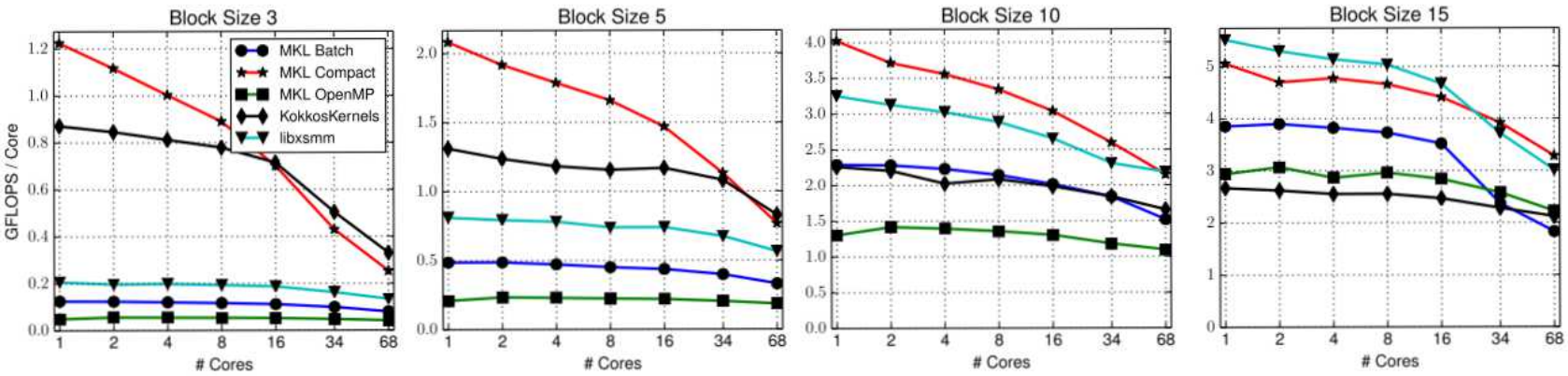


Comparison of compact batched TRSM against MKL batched DTRSM where the batch size (N) is 16384.

	Vector utilization			Speedup of MKL Compact	
Blocksize	MKL Batched	KokkosKernels	MKL Compact	1 thread	68 threads
3	6.44	13.13	15.68	73.17	9.38
5	8.89	15.39	13.64	14.90	5.36
10	10.66	15.93	14.65	5.37	3.44
15	12.44	15.98	15.05	5.33	4.09

Vector utilization (closer to 16 is better) with 68 threads and speedup against MKL batched DTRSM using 1 and 68 threads.

Batched GEMM



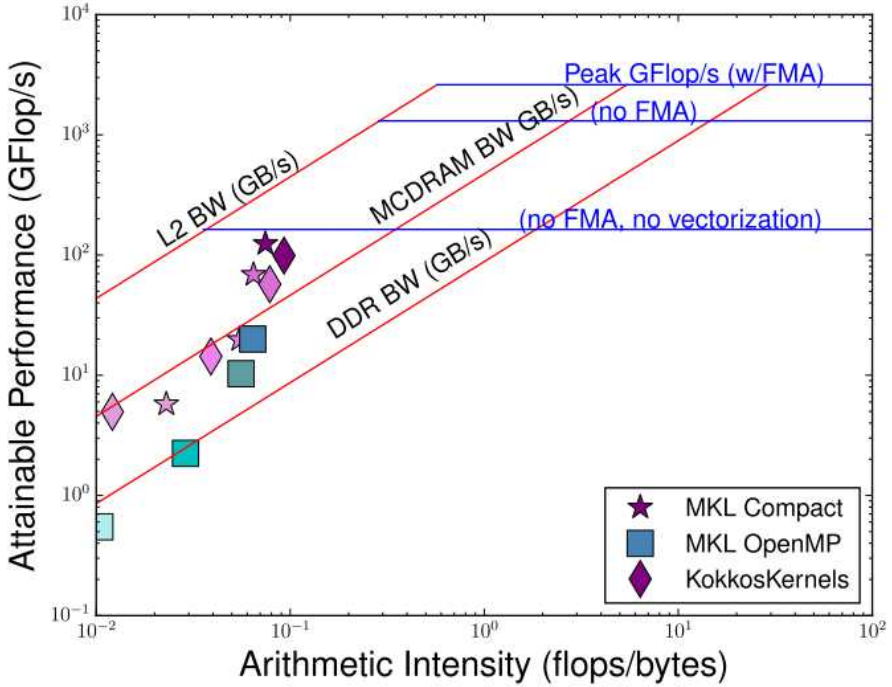
Comparison of compact batched `GEMM` against MKL batched `DGEMM` and `libxsmm` where the batch size (N) is 16384.

	Vector utilization				Speedup (Compact/Batch)	
Blocksize	MKL Batched	libxsmm	KokkosKernels	MKL Compact	1 thread	68 threads
3	10.30	9.99	12.96	15.87	10.26	3.46
5	12.40	11.99	14.34	15.97	4.32	2.34
10	14.43	15.01	15.14	15.99	1.76	1.45
15	14.94	15.87	15.41	15.99	1.32	1.27

Vector utilization (closer to 16 is better) with 68 threads and speedup against MKL batched `DGEMM` using 1 and 68 threads.

Roofline Analysis LU

- Roofline is obtained by $P = \min(P_c, B \cdot I)$ where P, P_c, B and I are attainable performance, peak compute performance, peak bandwidth and arithmetic intensity respectively.
- APEX[4] toolkit is used for performance analysis.
- Compact BLAS/LAPACK fully utilizes high bandwidth memory.

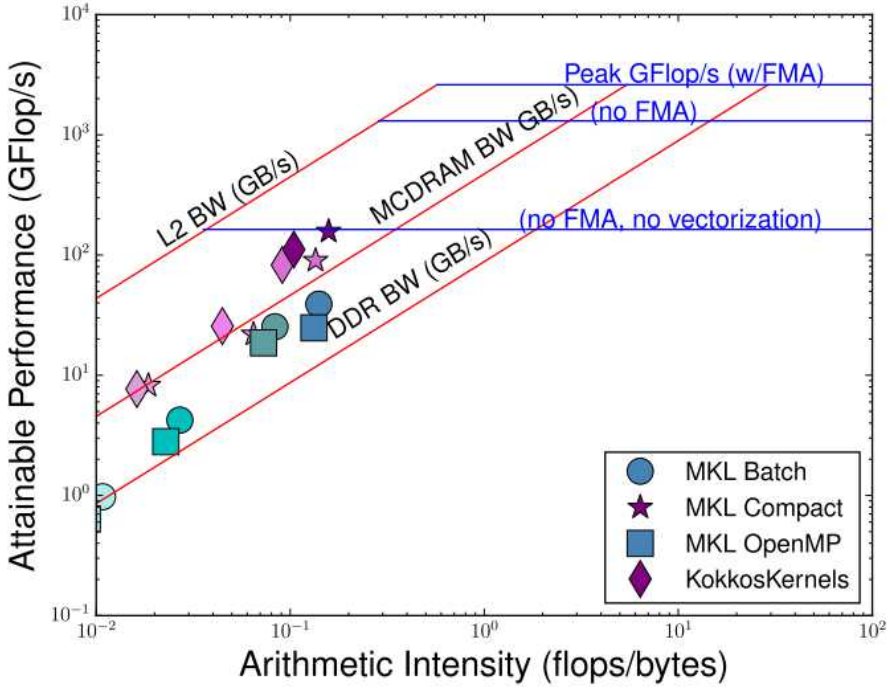


Roofline analysis of batched LU: darker color represents a bigger blocksize among 3,5,10 and 15.

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

Roofline Analysis TRSM

- Roofline is obtained by $P = \min(P_c, B \cdot I)$ where P, P_c, B and I are attainable performance, peak compute performance, peak bandwidth and arithmetic intensity respectively.
- APEX[4] toolkit is used for performance analysis.
- Compact BLAS/LAPACK fully utilizes high bandwidth memory.

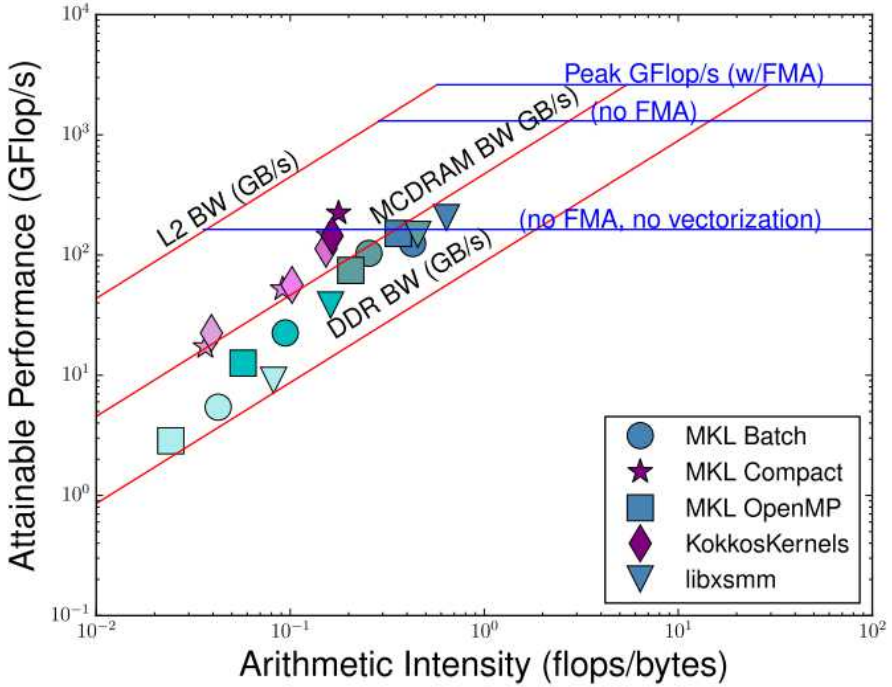


Roofline analysis of batched TRSM: darker color represents a bigger blocksize among 3,5,10 and 15.

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

Roofline Analysis GEMM

- Roofline is obtained by $P = \min(P_c, B \cdot I)$ where P, P_c, B and I are attainable performance, peak compute performance, peak bandwidth and arithmetic intensity respectively.
- APEX[4] toolkit is used for performance analysis.
- Compact BLAS/LAPACK fully utilizes high bandwidth memory.

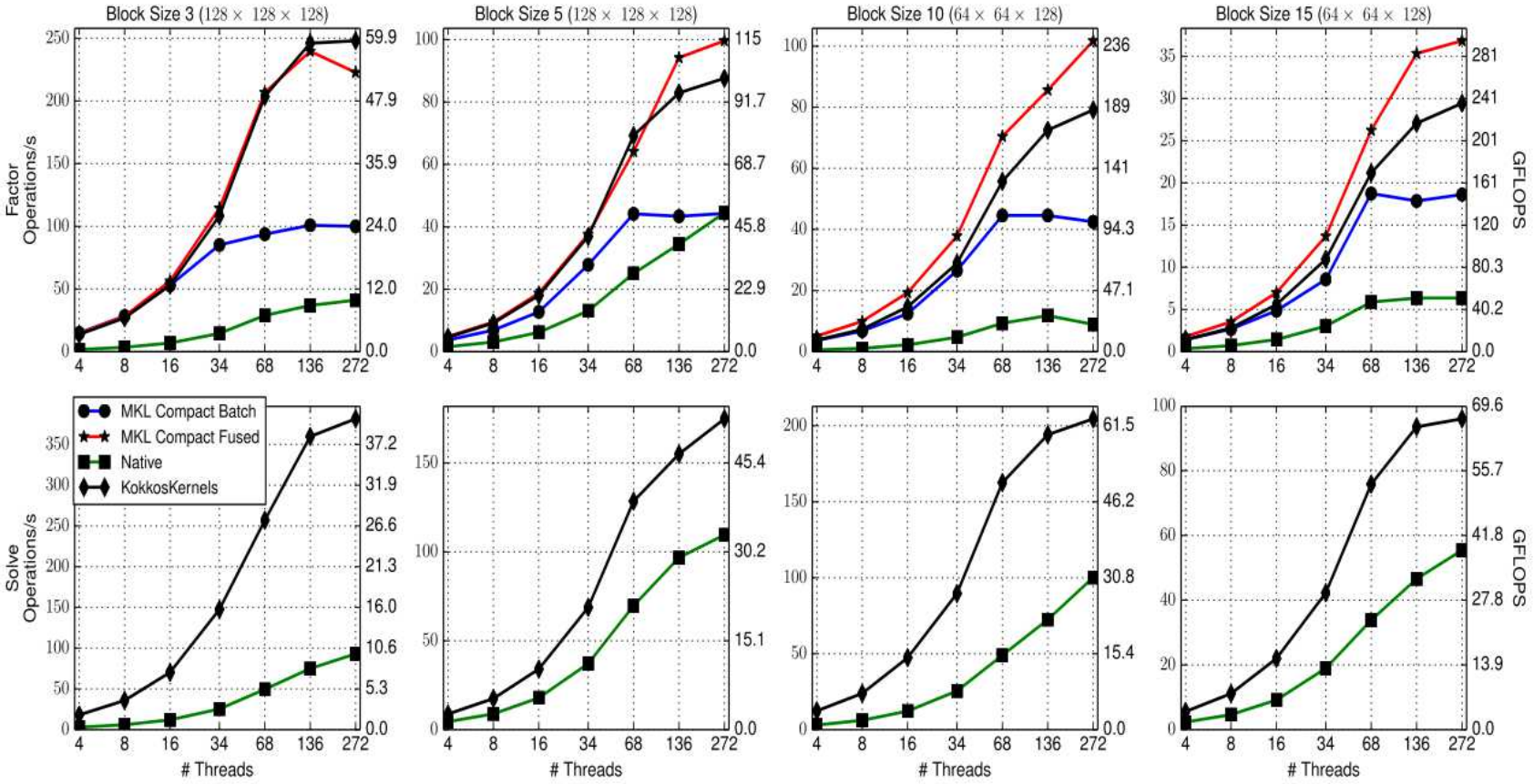


Roofline analysis of batched GEMM: darker color represents a bigger blocksize among 3,5,10 and 15.

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

Block Line Preconditioner: Performance Improvement on KNL

- KokkosKernels is compared with an optimized mini-app version of SPARC.



Performance comparison of line preconditioner on KNL against SPARC mini-app implementation

Conclusion

Sandia National Laboratories

- New compact batched BLAS/LAPACK is proposed and developed.
- Using SIMD friendly data layout, compact batched BLAS/LAPACK almost purely vectorized.
- Significant speedup is achieved for small sized problems.
- A new batch functionality e.g., batched block tridiagonal factorization can be efficiently implemented using compact batched BLAS/LAPACK APIs.
- Compared with an optimized mini-app version of SPARC, 1.7$\times$ - 6$\times$ speedup is observed for initializing and applying block line preconditioner.
- Sustainable performance improvement is expected through the standardization of compact batched BLAS/LAPACK.
- Compact BLAS/LAPACK is available in Intel® MKL 2018.