Exceptional service in the national interest

Sandia National Laboratories

KokkosKernels: Compact Layouts for Batched Blas and Sparse Matrix-Matrix multiply

Siva Rajamanickam,

Kyungjoo Kim, Andrew Bradley, Mehmet Deveci, Christian Trott,

Si Hammond

Batched BLAS Workshop, 2017, Atlanta

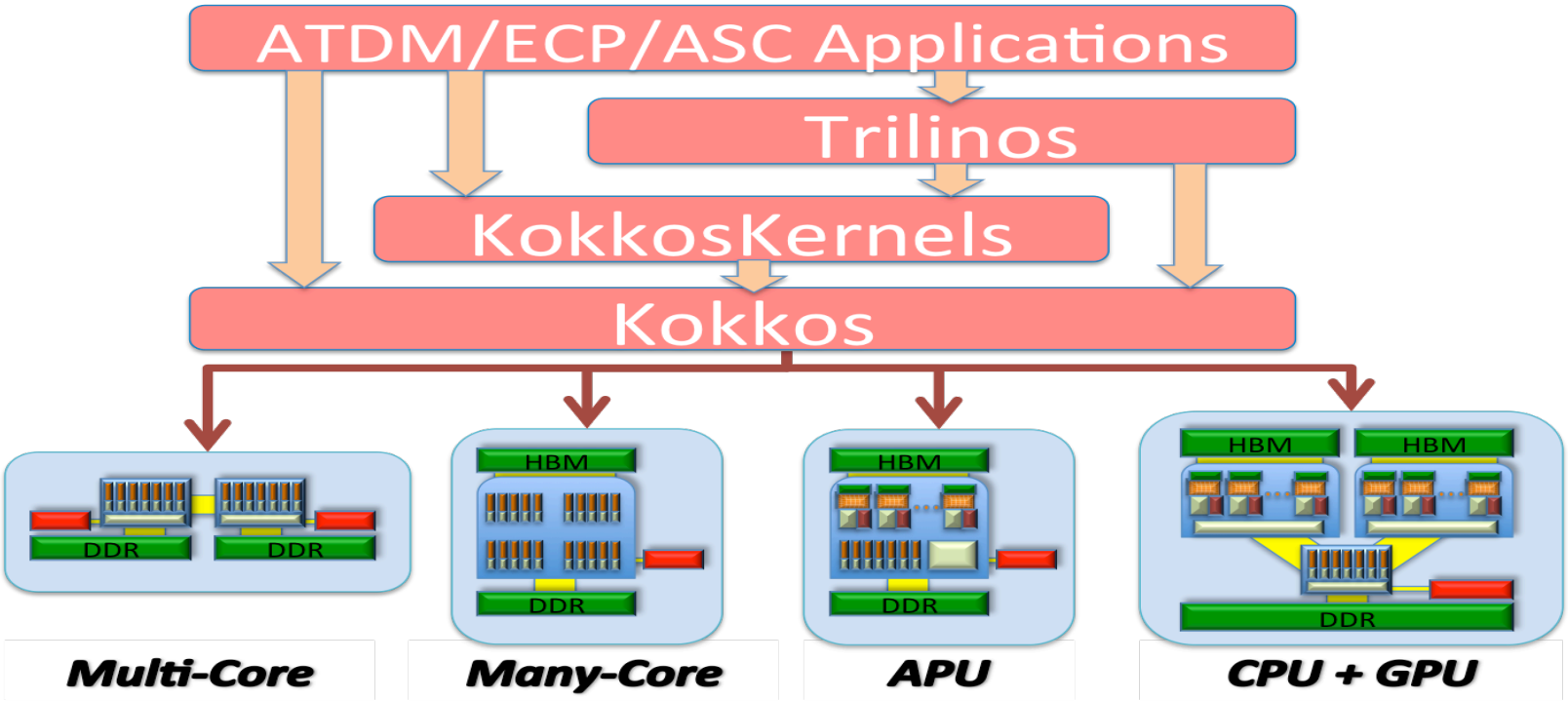
U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

CCR Center for Computing Research

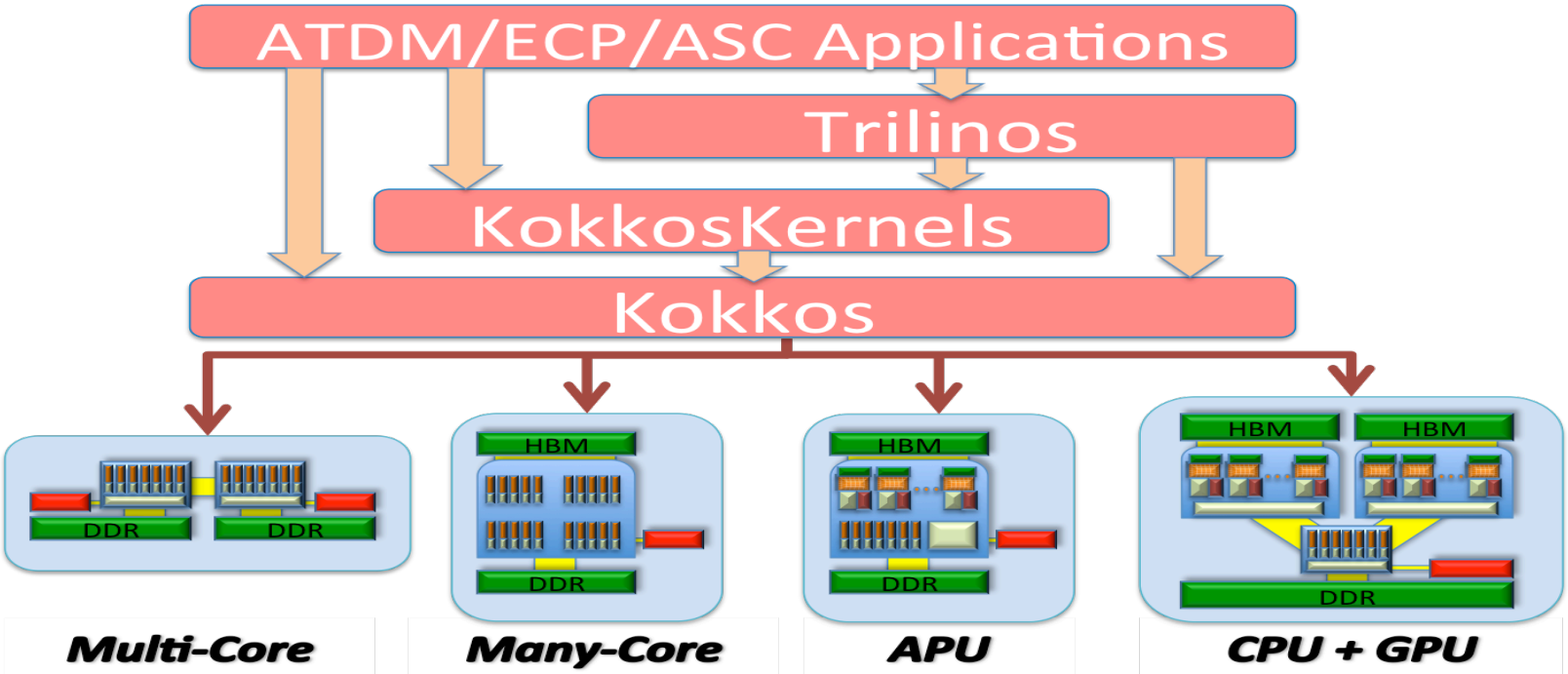
Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

KokkosKernels : Overview



- Layer of **performance portable** kernels on top of Kokkos
  - Sparse linear algebra kernels
  - Dense linear algebra kernels (Batched BLAS as well as traditional BLAS)
  - Graph kernels
  - Tensor Contraction kernels (upcoming)

KokkosKernels : Overview

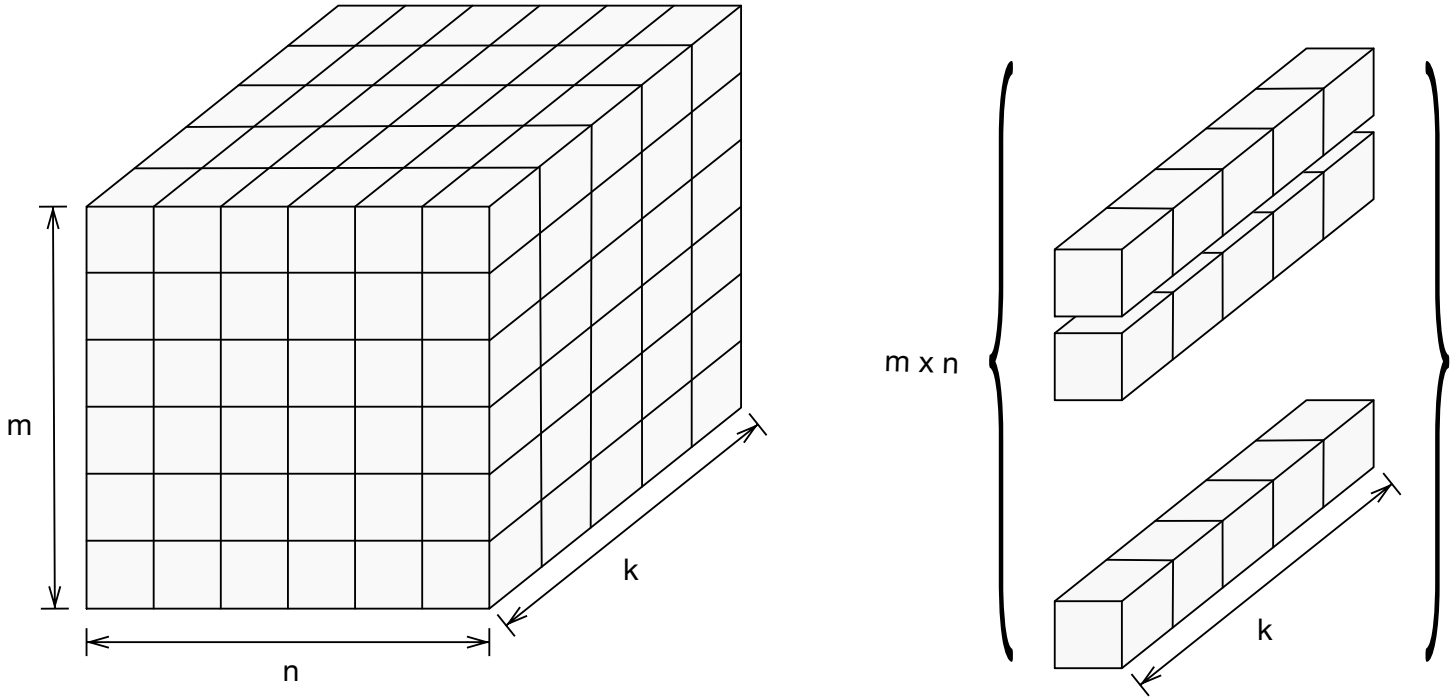


- No dependencies other than Kokkos
- Node-level only (No MPI)
- Provide kernels for all-levels of parallelism (wherever applicable) : *Device level, Team level, Thread level, Serial*
- Copyright received last week
  - Will reside in the Kokkos github organization

KokkosKernels : Current Kernels

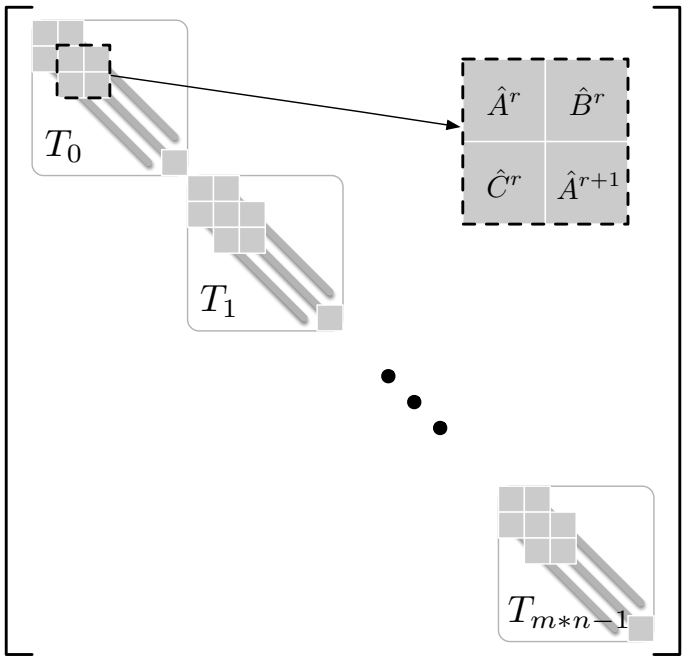
- Sparse linear algebra kernels
  - CrsMatrix – fill
  - Sparse Matrix Vector Multiply
  - **Sparse Matrix Matrix Multiply – Mehmet Deveci**
  - (Symmetric) Gauss Seidel
- Dense linear algebra kernels (BLAS)
  - BLAS1, some BLAS2
  - **Batched BLAS – Kyungjoo Kim**
- Graph kernels
  - Graph coloring
- Other Utilities
  - HashMap
  - Uniform Memory Allocator

Motivation for Batched BLAS with Compact Layouts



- Sandia application characteristics
  - One dimension of the mesh more important than the others when preconditioning
  - Multiple degrees of freedom per element gives rise to tiny blocks

Motivation for Batched BLAS/LAPACK

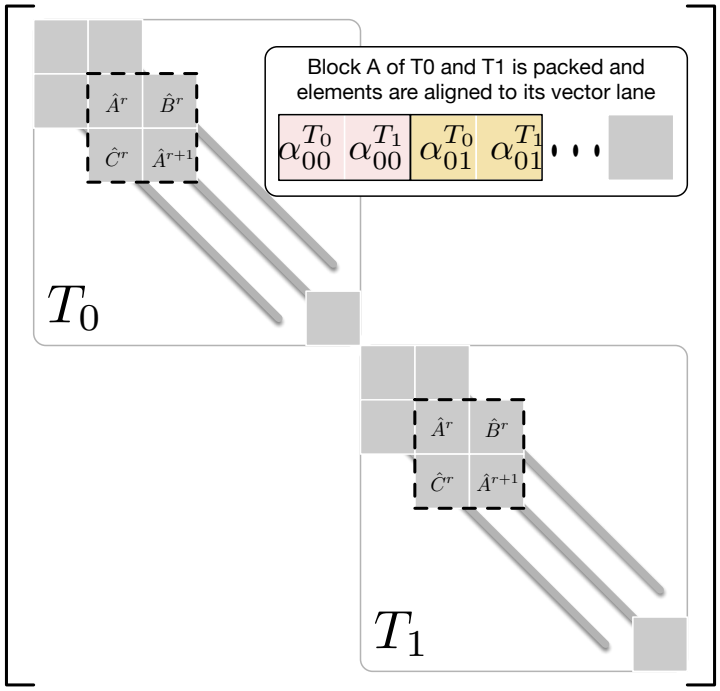


Algorithm 1: Reference impl. TriLU

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

- Block Jacobi preconditioner where each block is a Tridiagonal matrix
- Every scalar in the tridiagonal matrix is a small block matrix
  - Block sizes 5x5, 9x9, 15x15 etc
- Typical number of diagonal blocks 512-1024
- **Key kernels needed DGEMM, LU, TRSM**

KokkosKernels Compact Layouts for Batched BLAS



Algorithm 2: Batched impl. TriLU

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

- Data Layout for better vector intrinsics
  - Pack entries from up to vlen block diagonal matrices, vlen is the vector length (vector length = 2 shown)
  - Use vector intrinsics on the new data vector data with operator overloading
- Scalar Performance is due to explicit loop unrolling

Path Forward for Compact Layouts for Batched BLAS

- KokkosKernels:
  - A Performance-Portable Reference Implementation for compact layouts
- Collaborations would be ideal
  - Intel MKL team for compact layouts in MKL (ongoing)
    - ***Thanks to T. Costa, M. Guney, S. Knepper, S. Story***
    - Disseminate the ideas to broader community
  - "Shared Fate Milestones" (Exascale Computing Project) with the MAGMA team
    - ***Thanks to S. Tomov, J. Dongarra***
    - Extend the work to other kernels (E.g: Tensor contractions)

Compact Layouts for Batched BLAS: Experiments

MKL Test Setup

```
Kokkos::parallel_for(Kokkos::RangePolicy(N),
  KOKKOS_LAMBDA(const int k) {
    auto aa = Kokkos::subview(a, k, Kokkos::ALL(), Kokkos::ALL());
    auto bb = Kokkos::subview(b, k, Kokkos::ALL(), Kokkos::ALL());
    auto cc = Kokkos::subview(c, k, Kokkos::ALL(), Kokkos::ALL());

    cblas_dgemm(CblasRowMajor,
                CblasNoTrans, CblasNoTrans,
                BlkSize, BlkSize, BlkSize,
                1.0,
                (double*)aa.data(), aa.stride_0(),
                (double*)bb.data(), bb.stride_0()
                1.0,
                (double*)cc.data(), cc.stride_0());
  });
```

```
MKL_INT blksize[1] = { BlkSize };
MKL_INT lda[1] = { a.stride_1() };
MKL_INT ldb[1] = { b.stride_1() };
MKL_INT ldc[1] = { c.stride_1() };
CBLAS_TRANSPOSE transA[1] = { CblasNoTrans };
CBLAS_TRANSPOSE transB[1] = { CblasNoTrans };
double one[1] = { 1.0 };
MKL_INT size_per_grp[1] = { N };
```

```
cblas_dgemm_batch(CblasRowMajor,
    transA, transB,
    blksize, blksize, blksize,
    one,
    (const double**)aa, lda,
    (const double**)bb, ldb,
    one,
    cc, ldc,
    1,
    size_per_grp);
```

Compact Layouts for Batched BLAS: Experiments

KokkosKernels Test Setup

```
// Scalar version
Kokkos::View<double***,HostSpaceType>
          a(“a", N, BlkSize, BlkSize);

// Vector version
Kokkos::View<Vector<VectorTag<AVX<double>,4> >***,HostSpaceType>
          a("a", N/VectorLength, BlkSize, BlkSize),

Kokkos::parallel_for( Kokkos::RangePolicy(/* N or N/VectorLength */),
  KOKKOS_LAMBDA(const int k) {
    auto aa = Kokkos::subview(a, k, Kokkos::ALL(), Kokkos::ALL());
    auto bb = Kokkos::subview(b, k, Kokkos::ALL(), Kokkos::ALL());
    auto cc = Kokkos::subview(c, k, Kokkos::ALL(), Kokkos::ALL());

    KokkosKernels::Serial::
    Gemm<Trans::NoTranspose,Trans::NoTranspose,AlgoTagType>::
    invoke(1.0, aa, bb, 1.0, cc); });
  });
```

KokkosKernels Batched BLAS : DGEMM Performance

KNL, 1x68x4, 1.4 Ghz, Intel 17.1.132

DGEMM GFLOP/s	Blocksize 5	9	15	20	Speedup w.r.t. MKL
Number of threads 1	1.64	2.58	2.45	5.74	4.3
2	2.70	4.90	4.77	10.6	3.7
4	3.64	8.72	9.46	20.9	3.1
8	4.15	13.3	18.3	41.9	2.5
16	4.25	16.3	33.9	79.9	1.9
34	3.77	15.4	53.2	130	1.3
68	3.91	16.1	76.3	179	0.7

- Intel Knights Landing Architecture
- GFLOP/s (numbers) and speedup w.r.t MKL (colors) shown for 512 worksets
- Data flushed after each GEMM

KokkosKernels Batched BLAS : TRSM Performance

KNL, 1x68x4, 1.4 Ghz, Intel 17.1.132

TRSM GFLOP/s	Blocksize 5	9	15	20	Speedup w.r.t. MKL
Number of threads 1	0.80	1.61	1.90	3.83	33.9
2	1.06	2.79	3.57	7.16	28.6
4	1.07	3.89	6.35	13.0	23.2
8	1.06	4.58	10.4	21.8	17.9
16	0.96	4.40	14.6	31.9	12.5
34	0.80	3.84	16.6	38.2	7.2
68	0.84	3.66	18.8	43.2	1.8

- Intel Knights Landing Architecture
- GFLOP/s (numbers) and speedup w.r.t MKL (colors) shown for 512 worksets
- Data flushed after each TRSM

KokkosKernels Batched BLAS : LU Performance

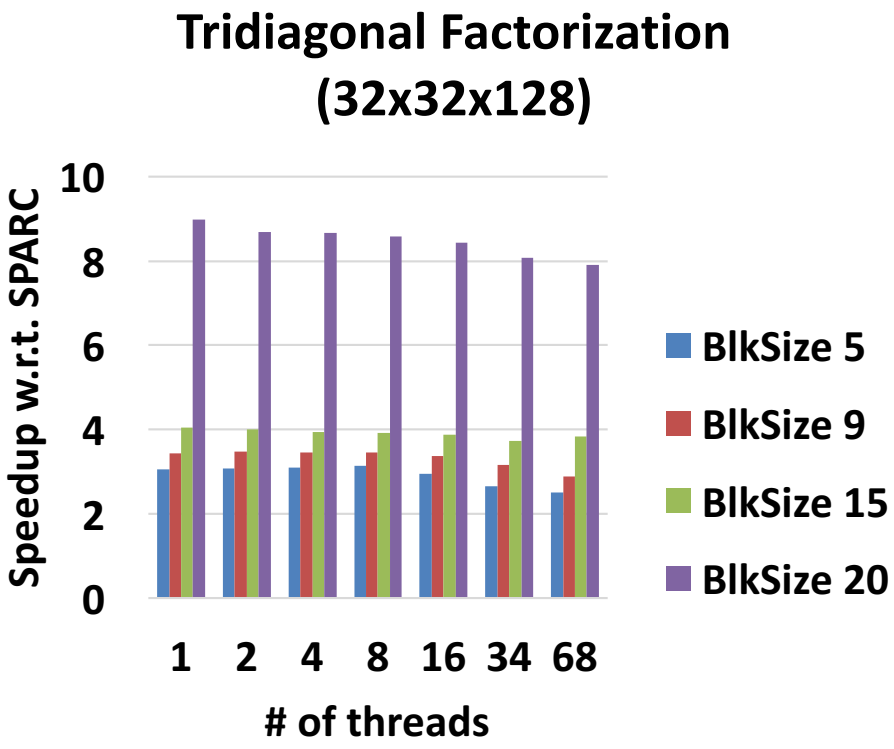
KNL, 1x68x4, 1.4 Ghz, Intel 17.1.132

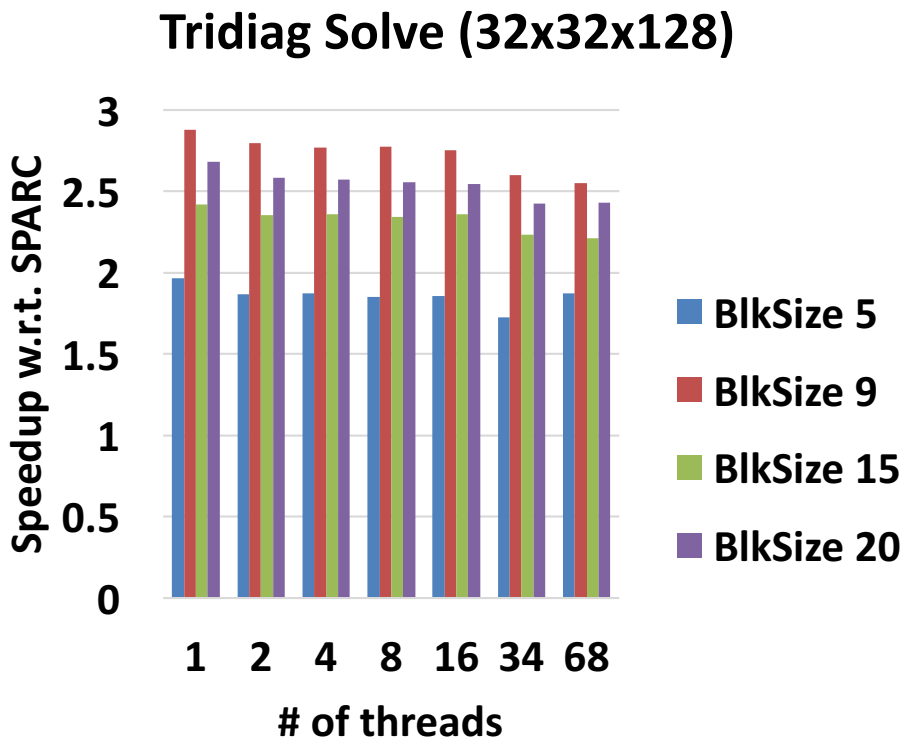
Number of threads	LU GFLOP/s (Blocksize 5)	9	15	20	Speedup w.r.t. MKL
1	0.53	1.26	1.73	3.18	20.9
2	0.69	1.93	3.25	6.01	17.7
4	0.68	2.68	5.60	10.6	14.5
8	0.69	3.04	8.52	17.4	11.3
16	0.64	3.03	11.2	24.2	8.1
34	0.53	2.57	11.6	27.6	4.9
68	0.59	2.52	13.3	29.8	1.6

- Intel Knights Landing Architecture
- GFLOP/s (numbers) and speedup w.r.t MKL (colors) shown for 512 worksets
- Data flushed after each LU

KokkosKernels Batched BLAS : Usage within Preconditioner

KNL, 1x68x4, 1.4 Ghz, Intel 17.1.132





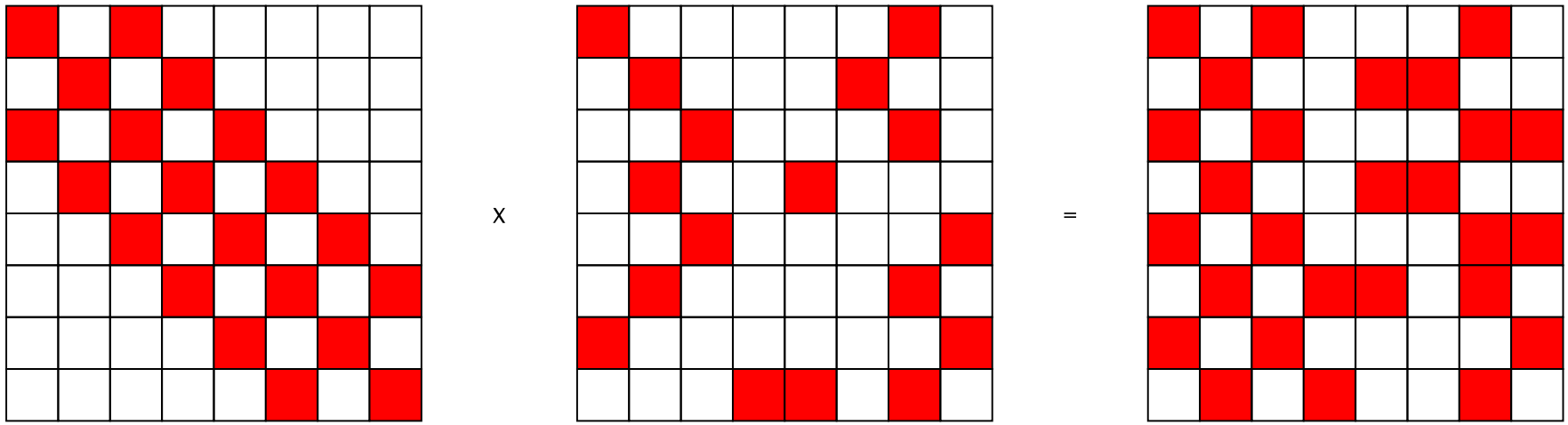
- Performance comparisons for Large-Block Jacobi Small-Block Tridiagonal factorization and Triangular Solve
- One right hand side per solve
- Speedups against a hand-tuned version of the code within the application

Compact Layouts for Batched BLAS: Discussion

- Path forward
  - Batched implementation of other kernels than DGEMM
  - Integrating with other linear algebra codes (FASTILU, direct methods)
  - Implementation of Compact/Packed Layouts in other libraries
- Smaller block sizes are an important use case for Sandia applications
- Need careful interface design for reuse of the structure
- C++20 standardization of the “packed double” or SIMT vector

Sparse Matrix-Matrix Multiplication (SpGEMM)

Sparse Matrix-Matrix Multiplication Problem



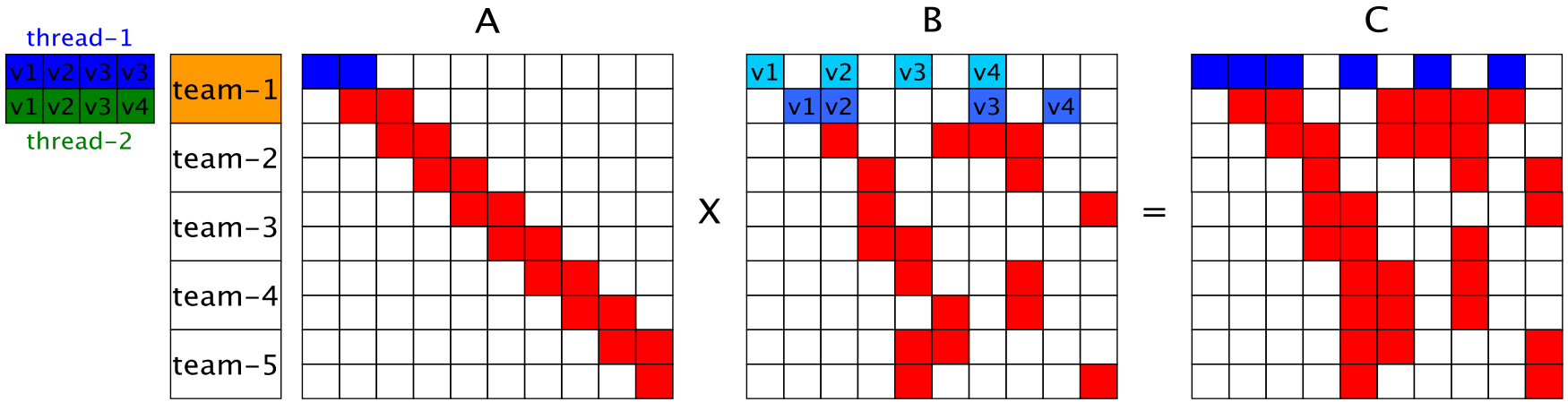
- SPGEMM: fundamental block for
  - Algebraic multigrid
  - Various graph analytics problems: clustering, betweenness centrality…
- Extra irregularity: nnz of C is unknown beforehand.

SpGEMM: Previous Work

- **Distributed Memory algorithms:**
  - 1D Trilinos, 2D Combinatorial Blas [Buluç 12],

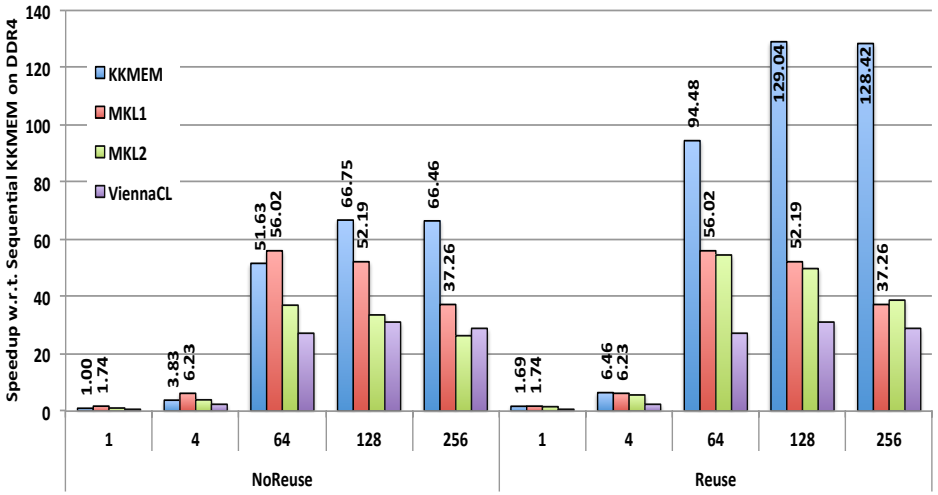
  3D [Azad 15], Hypergraph-based: [Akbudak 14], [Ballard 16]
- Most of the shared algorithms are based on 1D-Gustavson algorithm [Gustavson 78]
- **Multi-threaded algorithms:**
  - Dense Accumulator [Patwary 15]
  - Sparse Heap accumulators: ViennaCL, CommBlass
  - Sparse accumulators: MKL
- **GPUs**:
  - CUSP: 3D outer product O(FLOPS) memory
  - Hierarchical: cuSPARSE, bhSparse [Liu 14]

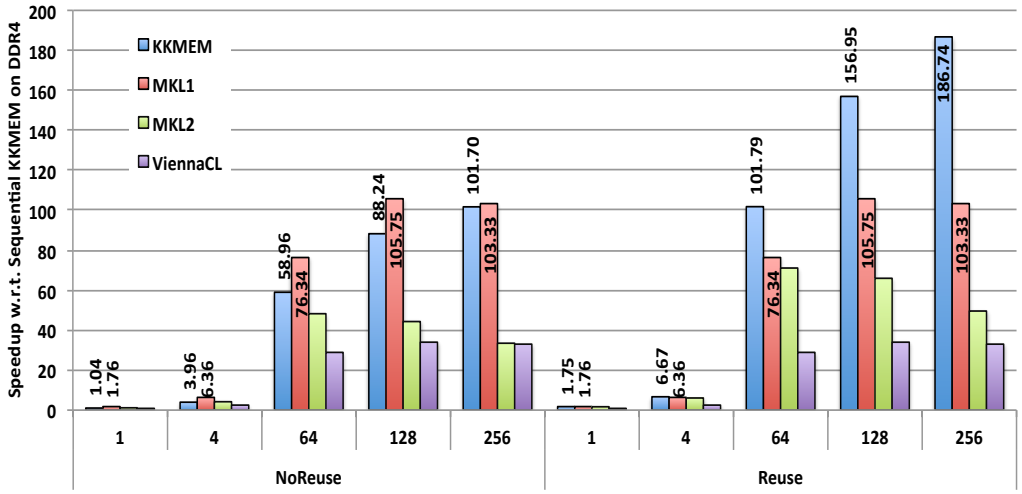
KokkosKernels Portable SPGEMM Method



- Two Phase Method : Symbolic and Numeric Phase
- Each team works on a bunch of rows
  - Team: Block (GPU), group of hyperthreads in core (CPU)
- Each worker in team works on consecutive rows.
  - Worker: Warp(GPUs), hyperthread(CPU)
  - More coalesced access on GPUs, better L1-cache usage on CPUs.
- Each vectorlane in a worker works on a different multiplication within a row:
  - Vectorlane: Threads in a Warp (GPUs), vector units (CPU)
- ***See Mehmet Deveci's talk on Tuesday @CSE for more details***

KokkosKernels Portable SPGEMM Method KNL Experiments





- Comparing KokkosKernels SPGEMM and two SPGEMM in Intel MKL and ViennaCL on Intel Knights Landing
- Geometric Mean Speedups w.r.to sequential KokkosKernel SPGEMM for 20 different matrix multiplications
- Reusing the symbolic structure is key to better performance on applications

KokkosKernels Portable SPGEMM Method on GPUs

numbers with other SPGEMM methods on P100 GPUs.

	KKMEM		Speedup w.r.t.			
	time	gflops	CUSP	bhSPARSE	ViennaCL	cuSPARSE
2cubes_sphere	0.02	3.631	4.54	1.20	1.06	3.62
cage12	0.03	2.396	3.13	0.75	1.22	2.74
webbase	0.27	0.521	0.66	0.54	5.18	2.30
offshore	0.03	4.304	5.25	1.33	1.21	7.08
filter3D	0.03	4.918	5.78	0.83	1.47	4.30
hugebubbles20_0	0.10	3.804	4.99	4.81	1.94	12.14
Europe	0.18	2.669	3.41	5.57	2.57	2.50
cant	0.04	12.001	12.83	1.05	1.42	0.77
hood	0.08	13.944	14.22	0.97	1.77	1.72
pwtk	0.07	17.717	17.88	1.13	2.06	1.53
Empire_R_AP	0.04	4.734		0.89	0.65	0.88
Empire_RA_P	0.08	2.316		1.03	0.41	0.68
Laplace_R_A	0.39	2.041		0.68	0.73	2.71
Laplace_A_P	0.15	5.398		2.57	1.00	11.65
Laplace_R_AP	0.19	5.466		2.36	1.24	5.24
Laplace_RA_P	0.47	2.203		1.67	0.65	3.32
Brick_R_A	0.64	2.381		1.16	1.82	4.91
Empire_R_A	0.43	5.934		1.09	1.06	1.11
Empire_A_P	0.30	8.463		3.60	1.05	1.48
Brick_RA_P	0.61	6.326		1.26	0.43	1.14
ldoor	0.32	14.910		1.09	1.88	1.76
delaunay_n24	0.41	3.086			1.74	1.12
Brick_R_AP	0.24	6.349			0.76	1.91
channel	0.43	7.054			1.51	3.10
Brick_AP	0.49	7.954			0.95	4.54
cage15	1.56	2.660				4.86
Bump	0.88	13.126				1.58
audi	1.31	12.345				1.54
dielFilterV3real	1.80	9.679				1.85
		Geomean:	5.25	1.36	1.22	2.43

- Comparing KokkosKernels SPGEMM and four other GPU implementations (CUSPARSE, CUSP, bhSparse, ViennaCL)
- Both multigrid and data analysis style multiplications
- Reusing the symbolic structure is key to better performance on applications

Sparse Matrix-Matrix multiplication Discussion

- Raising Importance of SPGEMM
  - Data Analysis community is driving lot of the work
  - Key for scalability of algebraic multigrid setup
- An opportunity to address a gap for important applications
- Addressing the symbolic reuse portion is an important usecase for several applications
- One performance-portable reference implementation available
  - Vendor collaborations and other reference implementation needed

Thank you

srajama@sandia.gov

KokkosKernels Portable SPGEMM Method

- **2 level Hashmap Accumulator:**
  - 1st level uses GPUs shared memory or a small memory that will fit in L1 cache
  - 2nd level goes to global memory
- **Uniform Memory Pool:**
  - Only some of the workers need 2nd level hash map. They request memory from memory pool.
- **Compression: Symbolic works performs unions on rows. Binary relations that can be done with BitWiseO**