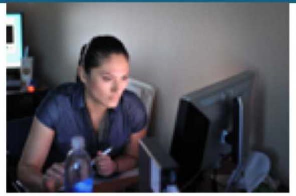
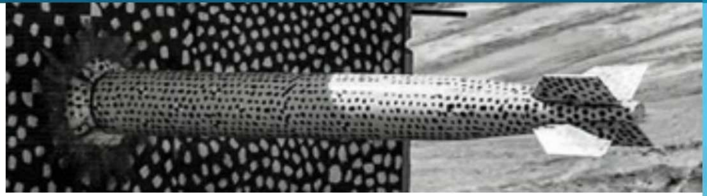
Sandia National Laboratories

SAND2019-14189C

Kokkos Kernels: Library Based Approach for Performance Portable Sparse/Dense linear algebra and Graph Kernels





PRESENTED BY

Siva Rajamanickam, S. Acer, L. Berger-Vergiat, V. Dang, N. Ellingwood, B. Kelley, K. Kim, C.R. Trott, J. Wilke

U.S. DEPARTMENT OF ENERGY NNSA

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND2019-XXXXXX

Approaches to Programming GPUs

Native Programming Models

- CUDA (NVIDIA), HIP (AMD), SYCL (Intel)
- Pros: Customized for each architecture, so low level control
- Cons: Rewrite code every time you buy a hardware from a new vendor

Directive Based Approach

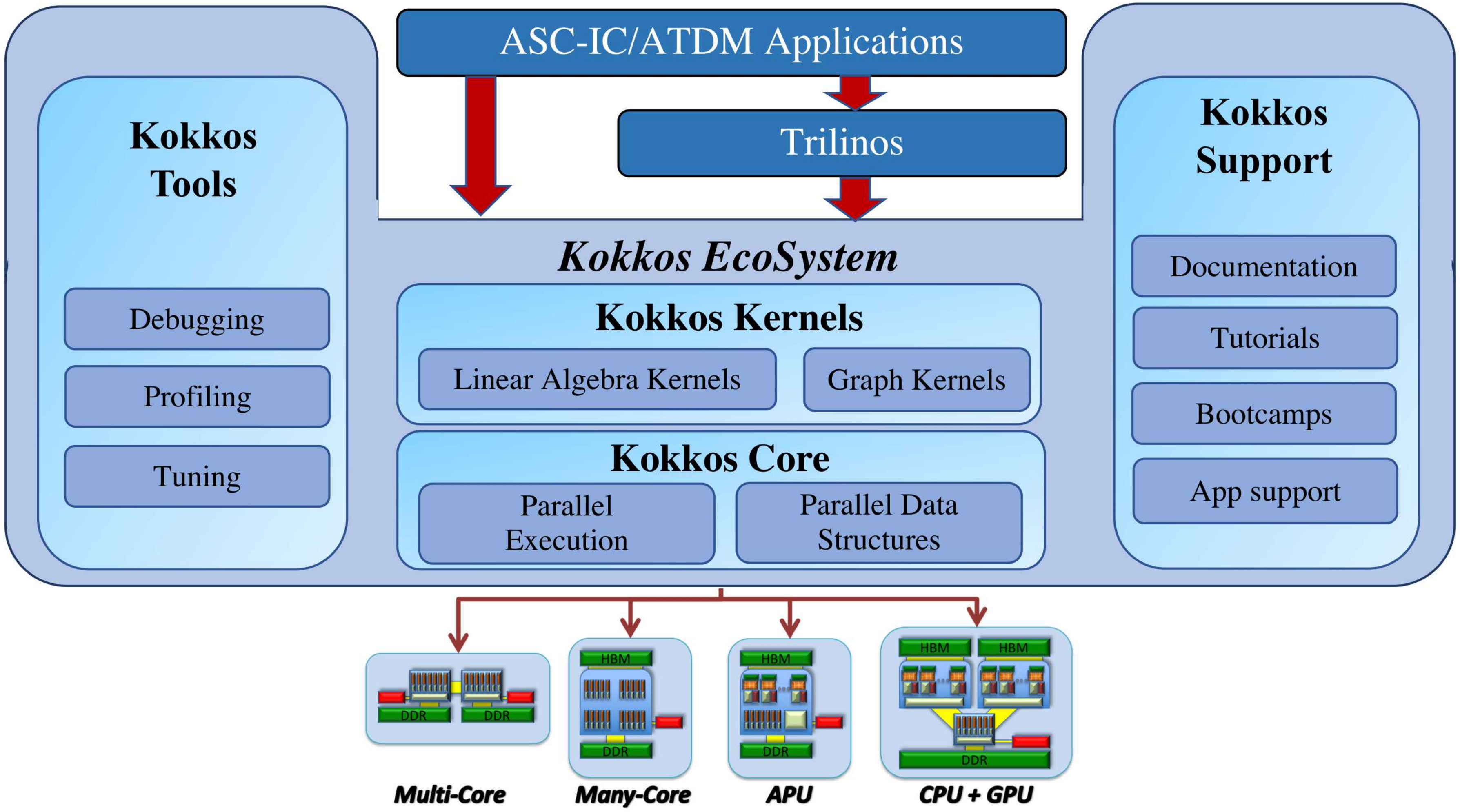
- OpenMP, OpenACC
- Pros: Standards based, General
- Cons: Long lag time between what is needed and when they are needed, Might have to resort to #ifdef after all, Different level of support from vendors

Library Based Approach

- Kokkos, RAJA
- Pros: Portable, Clean abstractions, Quicker turnaround, Reference implementations of standards
- Cons: Dependency on libraries

Library based performance portability allows for writing applications to several architectures with limited dependencies

Kokkos Ecosystem for Performance Portability



Kokkos Core: parallel patterns and data structures; supports several execution and memory spaces

Kokkos Kernels: performance portable BLAS; sparse, dense and graph algorithms

Kokkos Tools: debugging and profiling support

Write-once using Kokkos for portable performance on different architectures

Kokkos Ecosystem addresses complexity of supporting numerous many/multi-core architectures that are central to DOE HPC enterprise

Focus of Kokkos Kernels

Deliver ***portable*** sparse/dense linear algebra and graph kernels

- These are the kernels that are in 80% of time for most applications
- Key problems: Kernels might need different algorithms/implementations to get the best performance
- Ninja programming needs in addition to Kokkos
- Users of the kernels do not need to be ninja programmers
- ***Focus on performance of the kernels on all the platforms of interest to DOE***

Deliver ***robust software ecosystem*** for other software technology projects and applications

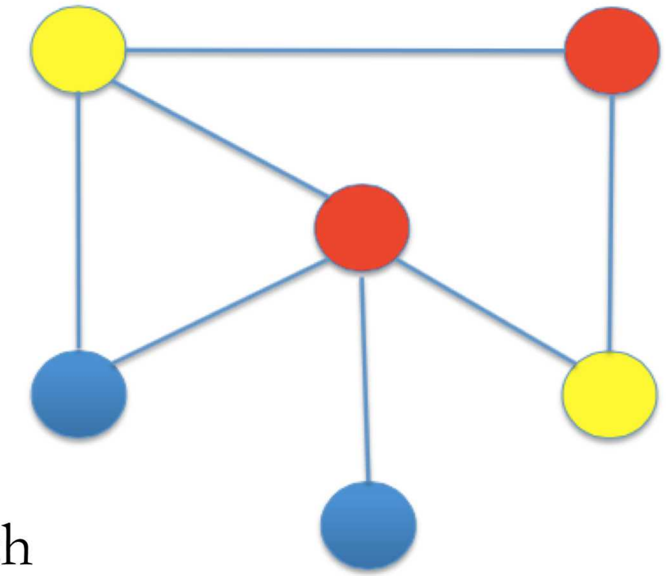
- Production software capabilities that give high performance, portable and turn-key
- Tested on number of configurations nightly (architectures, compilers, debug/optimized, programming model backend, complex/real, ordinal types…)
- Larger release/integration testing with Trilinos and applications
- Kokkos Support, github issues, tutorials, hackathons, user group meetings (planned)

Kokkos Kernels delivers portable, high-performance kernels in a robust software ecosystem to support ECP applications

Overview of the Tutorial

- Use **three kernels** to demonstrate the use of a library based approach for performance portability
  - **Distance-1 graph coloring**: Identify independent rows that can be processed in parallel for a parallel preconditioner.
  - **Sparse matrix-matrix multiplication**: Compute the result of C = A * B where A and B are sparse matrices
  - **Team level batched linear algebra**: Compute a block tridiagonal factorization using team-level linear algebra

Graph Kernel: Distance 1 Graph Coloring



Given a graph G = (V , E),

- With vertices $v \in V$
- Edges $(v_1, v_2) \in E \quad v_1, v_2 \in V$

- Distance-1 graph coloring: assigns colors to vertices so that each vertex have different color from all of its neighbors
  - $C : V \rightarrow N \quad C(v_1) \neq C(v_2) \quad for\ all\ (v_1, v_2) \in E$

The distinct number of colors assigned to vertices: $|C|$

- Graph coloring problem that minimizes $|C|$ is NP-Hard [Zuckerman, 2006]

Applications:

- Parallel computation, Jacobian computation, Register allocations …

Image courtesy of Sariyuce, Saule, Catalyurek, SIAM PP, 2012

Graph Coloring Heuristics

Simple greedy heuristics often obtain near optimal solutions

- First-fit [Matula, 1972], with $O(|V|+|E|)$
- Keeps forbidden array to store the colors of neighbors
- Obtains $|C| \leq \delta + 1$ where δ max degree in the graph

Parallel Implementations

- Speculative Method [Gebremedhin and Manne,2000], [Bozdag, 2008]
- [Jones and Plassmann, 1993] parallelization of [Luby, 1986]
- Distributed Implementations: [Catalyurek,2012]
- Hybrid MPI+OpenMP Implementations: [Sariyuce, 2012]

Manycore Coloring Heuristics Before Kokkos Kernels approach

Xeon Phi: Speculative Method (IPGC) [Saule, 2012]

- Speculatively color vertices in each threads
- Detect conflicts due to race conditions and recolor them

GPUs: Nvidia cuSPARSE: [Naumov, 2015]

- Relaxation of Jones and Plassmann (JP) based on the independent sets
- Highly parallel, runs fast
- But the number of colors found are usually very high

Vertex-Based Coloring on GPUs

Minimum atomic work are vertices, therefore 1 vertex is owned by a single thread: IPGC, cuSPARSE

Some implementation details are often ignored

- e.g. the requirement of thread private Forbidden array O(δ)
- can be a problem on highly irregular graphs, or when number of threads are high

Optimization:

- Limit the size of Forbidden array e.g. with constant size 32 **(called VB)**
  - Traverse the adjacency multiple times
    - first for the vertices with colors 1-32, then 33-64 …
  - On GPUs this array can be stored in slow local memory
- Use the bits of single int **(called VBBIT)**
  - Conversion to back and forth to bit representation
  - Stored in registers on GPU rather than slow memory

Edge-Based Coloring on GPUs

Minimum atomic work are edges, therefore 1 edge is owned by a single thread

- Requires more complex synchronizations

Three Phase algorithm

- Assign Colors (Vertex Based)
- Detect Conflict (Edge Based)
- Forbid Colors (Edge Based)

Optimization:

- Bit-based forbidden arrays
- Use Edge Filtering to minimize number of times an edge is seen
- Convergence improvements with tentative coloring
- Parallel-Prefix sums vs Atomics

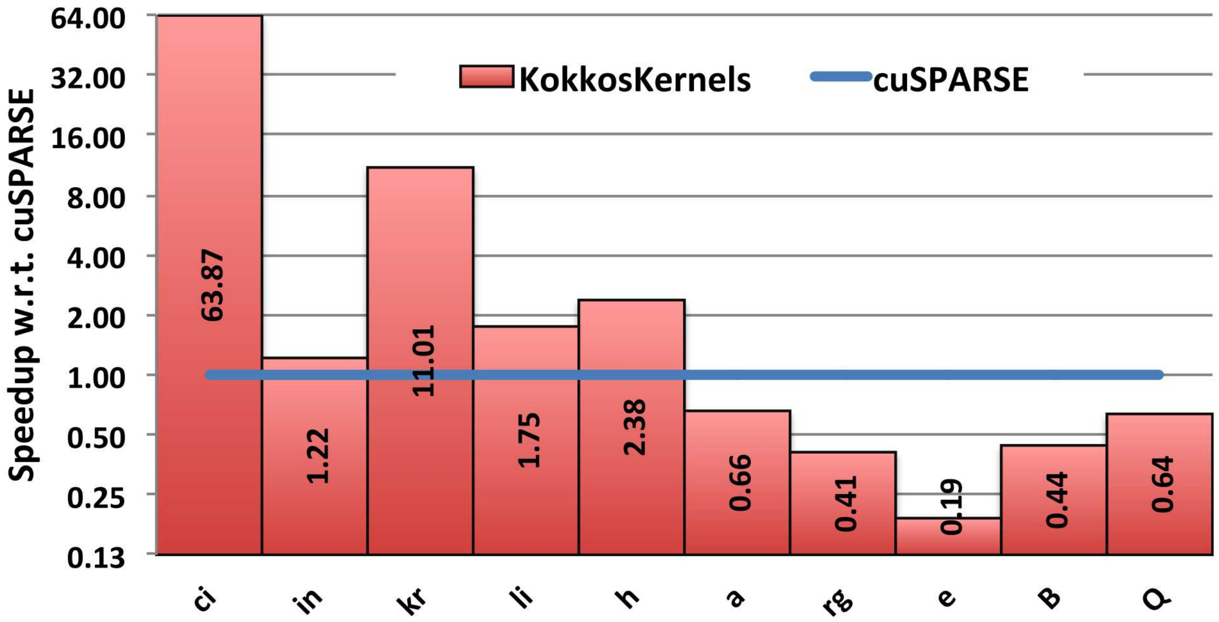
“Simple” Distance-1 graph coloring on GPUs can become complex very quickly due to the massive parallelism.

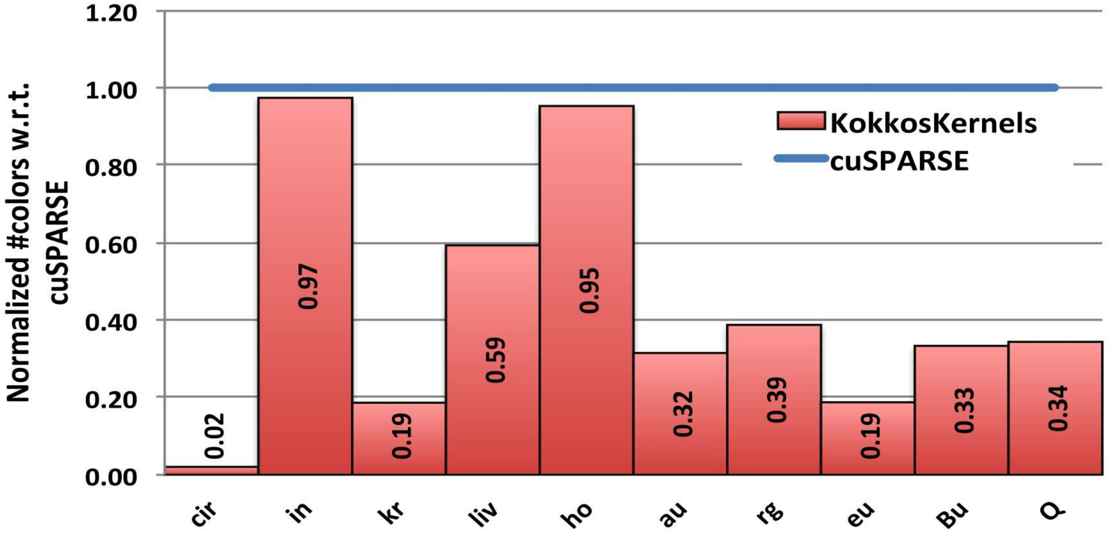
Graph Coloring and Multithreaded Gauss Seidel

- Conjugated Gradient Algorithm in Kokkos Kernels
  - Preconditioner: multi-threaded Gauss-Seidel
    - Very sequential algorithm
      - Coloring to find independent rows
      - Then operations can be done in parallel for independent rows
      - More colors → more synchronization, less work in parallel regions
  - Other Approaches possible for Gauss-Seidel preconditioning:
    - Level-set based Gauss-Seidel (similar to a triangular solve)
      - Dynamic parallelism is difficult on GPUs
    - Block Gauss-Seidel similar to MPI
    - Can become (block) Jacobi preconditioner as #threads increase as in GPUs

Details of the algorithm are in the paper *Parallel Graph Coloring for Manycore Architectures, M. Deveci, E. Boman, K. Devine and S. Rajamanickam*, IPDPS 2016.

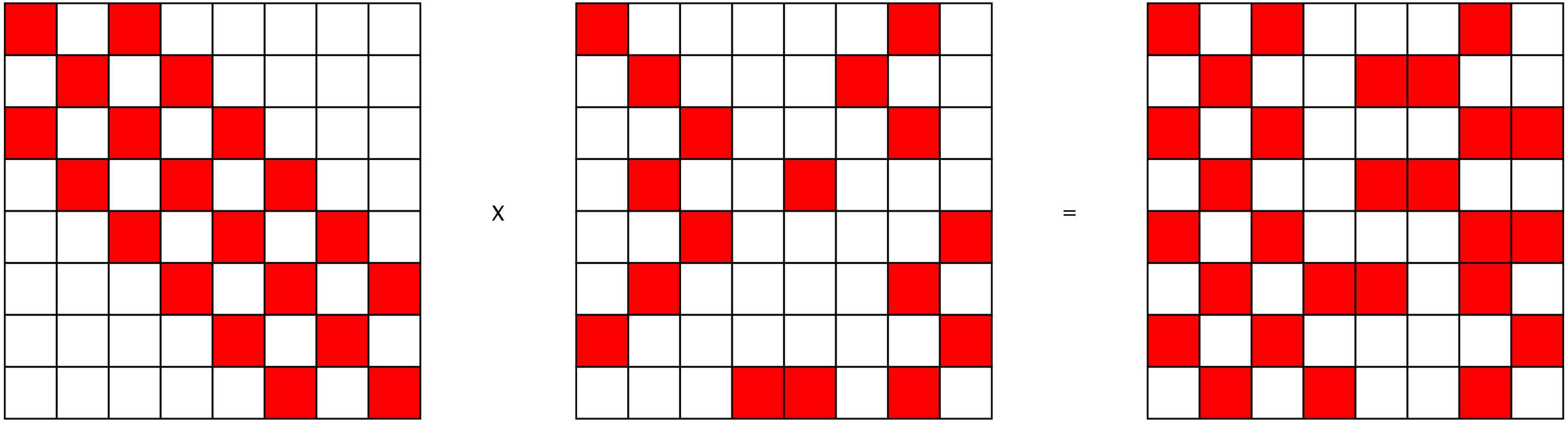
Graph Coloring and Multithreaded Gauss Seidel





- **Performance**: Better quality (4x on average) and run time (1.5x speedup) w.r.t cuSPARSE.
- Performance portable implementation allows better results on the KNL as well.
- Enables parallelization of preconditioners: Gauss Seidel: 136x on K20 GPUs w.r.t. serial Sandy Bridge (significant for a triangular solve like kernel)
- Application Integration
  - Integrated in Trilinos preconditioners (IFPACK2 package)
  - Evaluated in the Exascale Computing Project Wind Energy application Nalu

Sparse Kernel : Sparse Matrix-Matrix Multiplication



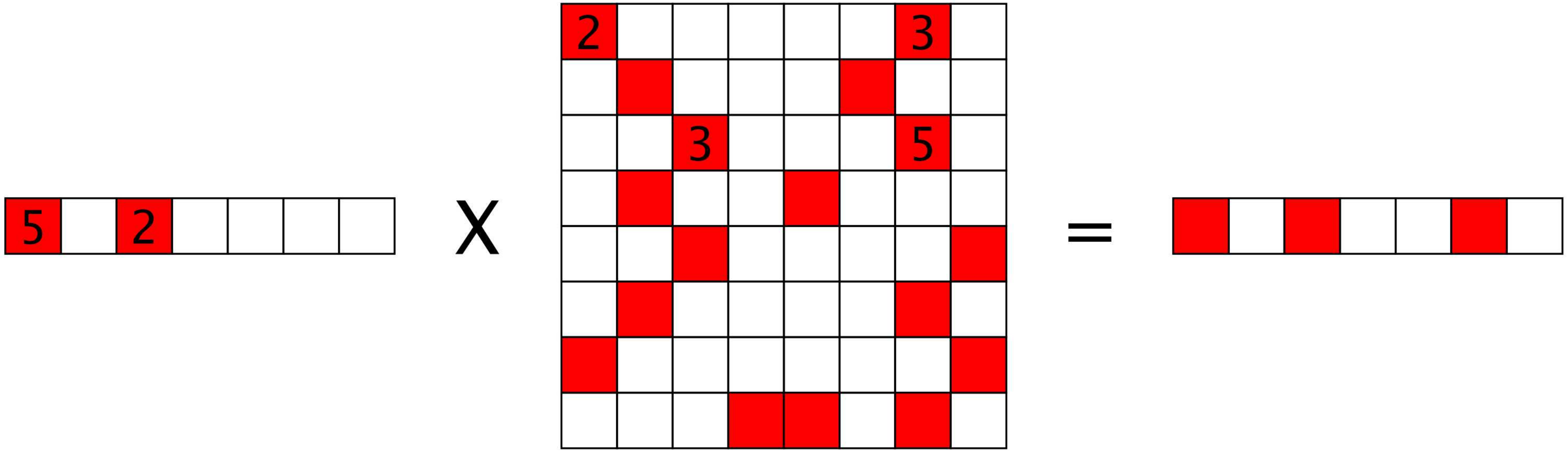
Sparse Matrix-Matrix Multiplication (SpGEMM): fundamental block for

- Algebraic multigrid RxAxP
- Various graph analytics problems: triangle counting, clustering, betweenness centrality…

More complex than most of the other sparse BLAS and graph problems:

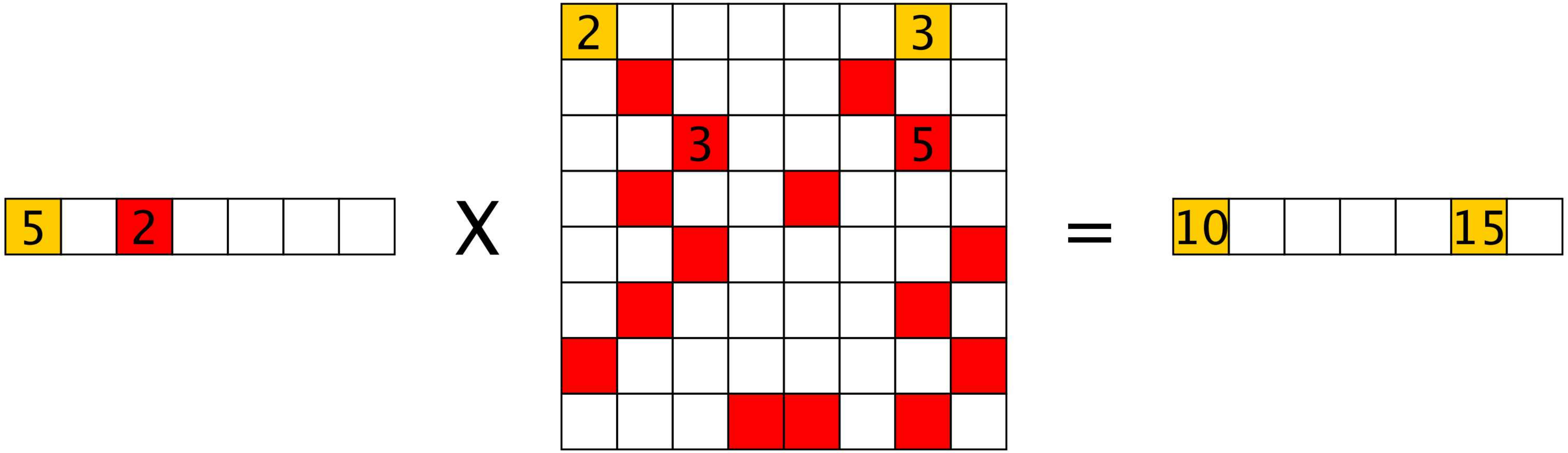
- Extra irregularity: nnz of C is unknown beforehand.
- Requirement of thread private data structures

Sparse Matrix-Matrix Multiplication Background



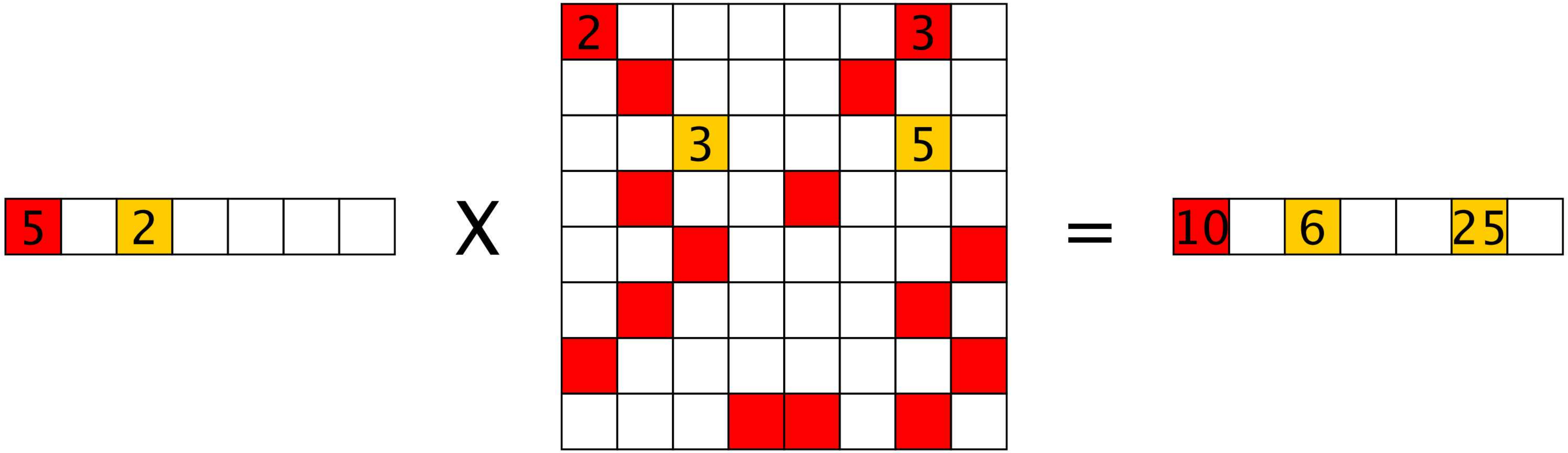
Sequential Algorithms: 1D [Gustavson 78]

Sparse Matrix-Matrix Multiplication Background



Sequential Algorithms: 1D [Gustavson 78]

Sparse Matrix-Matrix Multiplication Background



Distributed algorithms:

- 1D Trilinos
- 2D Combinatorial Blas [Buluç 12] – 2D Decomposition of C
- 3D [Azad 15]
- Hypergraph-based: [Akbudak 14], [Ballard 16]

Sparse Matrix-Matrix Multiplication Background

Most of the shared algorithms bases on 1D Gustavson algorithm

- Differ in the data structure they use for accumulation

Multi-threaded algorithms:

- Dense Accumulator [Patwary 15]
- Sparse Heap accumulators: ViennaCL, CommBlass
- Sparse accumulators: MKL

GPUs:

- CUSP: Expand – Sort – Collapse
- Hierarchical: cuSPARSE, bhSparse [Liu 14]

Portable Sparse Matrix-Matrix Multiplication

Variety in architectures

- Tens/Hundreds/thousands of threads
- CPUs/lightweight-cores/streaming multi-processors
- Shared / high bandwidth / DDR memory

Native multi-threaded algorithms

- Fewer threads, more memory & more work per thread

GPU algorithms

- Thousands of threads, less memory & less work per thread

Design decisions

- Work distribution to threads
- Scalable data structures
- Limitations of specific architectures

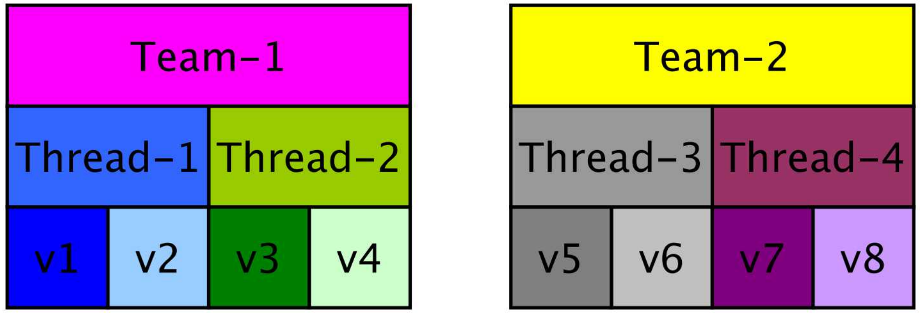
Thread Mapping

Each team works on a bunch of rows of C (or A)

- **Team**: Thread block (GPU) group of hyper-threads in a core (CPU)

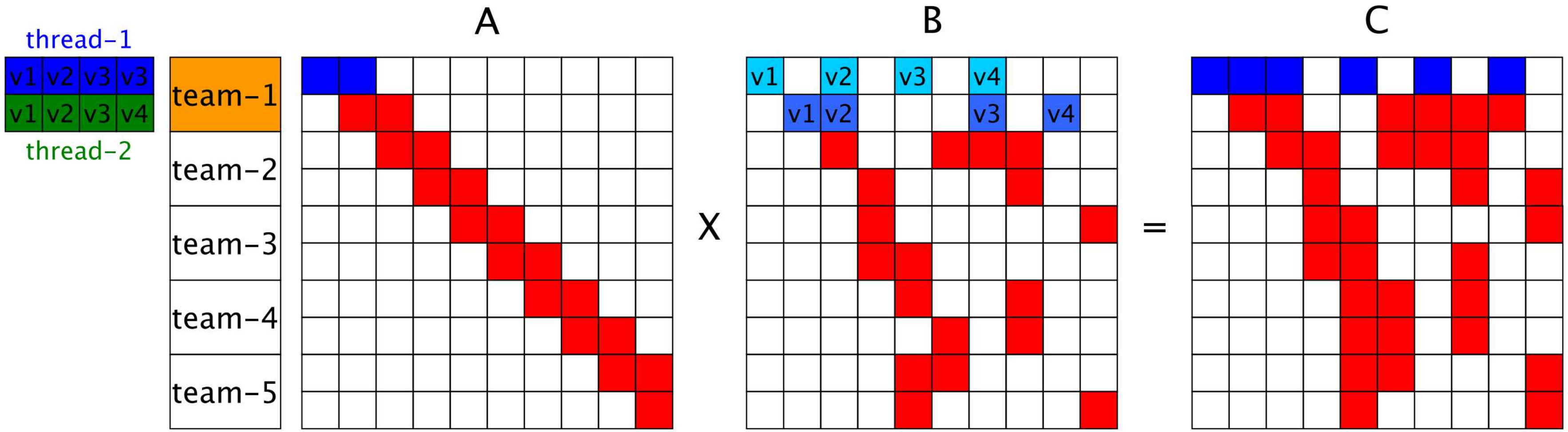
Each worker in team works on consecutive rows of C

- **Worker**: Warp (GPUs), hyperthread (CPU)
- More coalesced access on GPUs,
- Better L1-cache usage on CPUs.

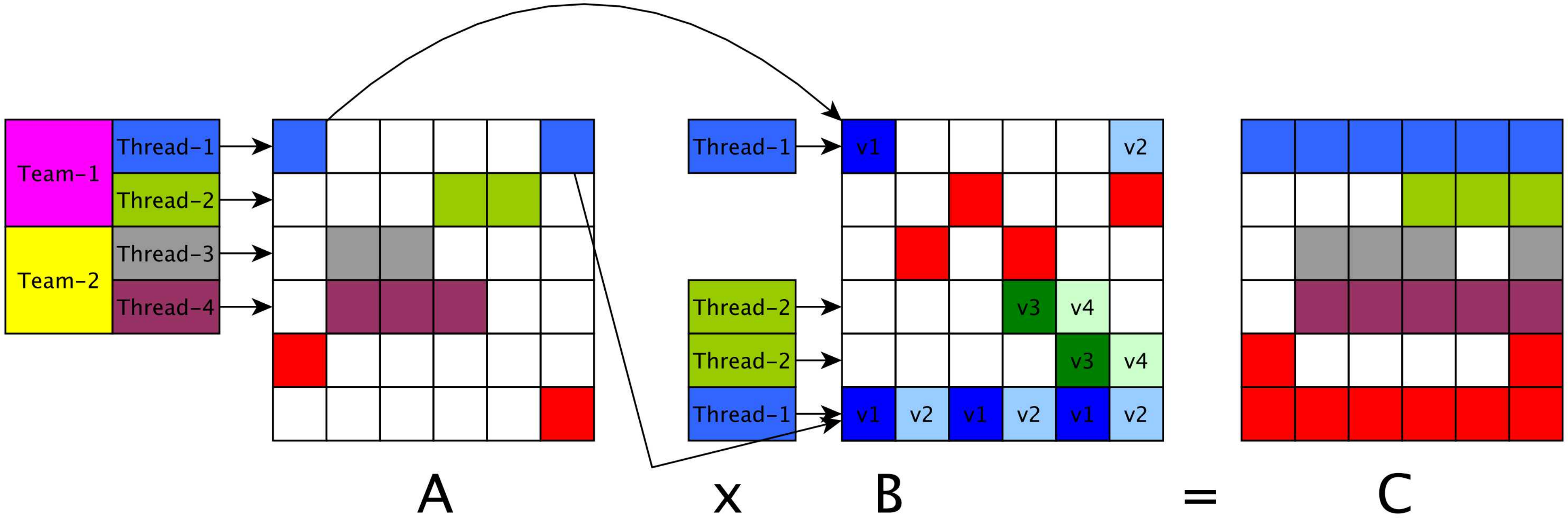


Each vectorlane in a worker works on a different multiplications within a row:

- **Vectorlane**: Threads in a Warp (GPUs), vector units (CPU)

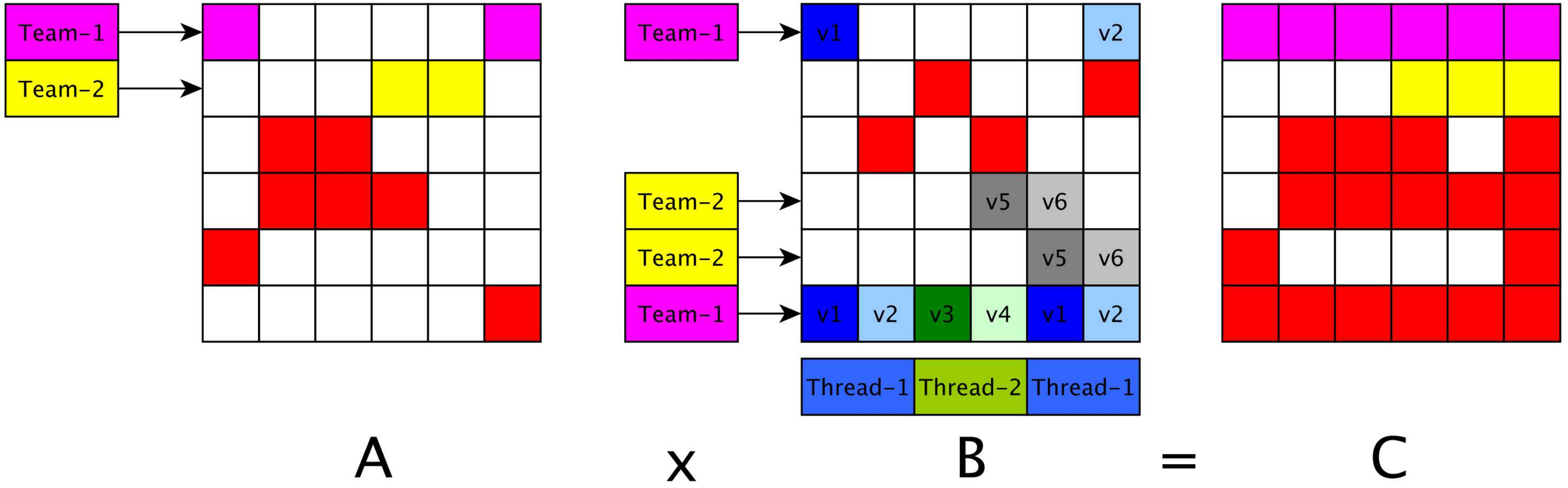


Thread Mapping (continued)



Thread to rows of A

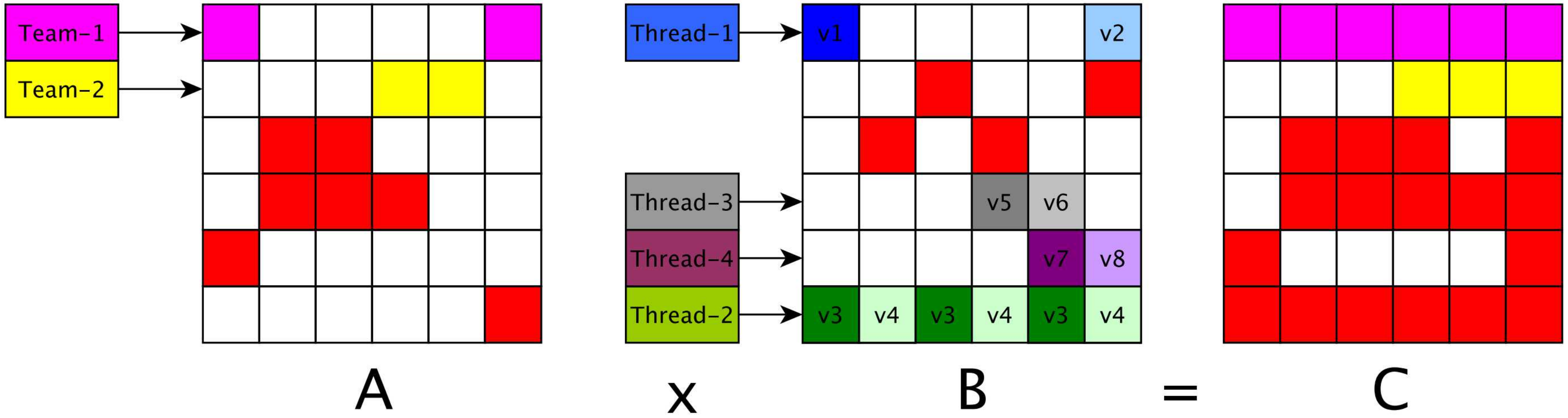
- No atomics needed in data structures
- Load balancing could be a problem



Team to row of B

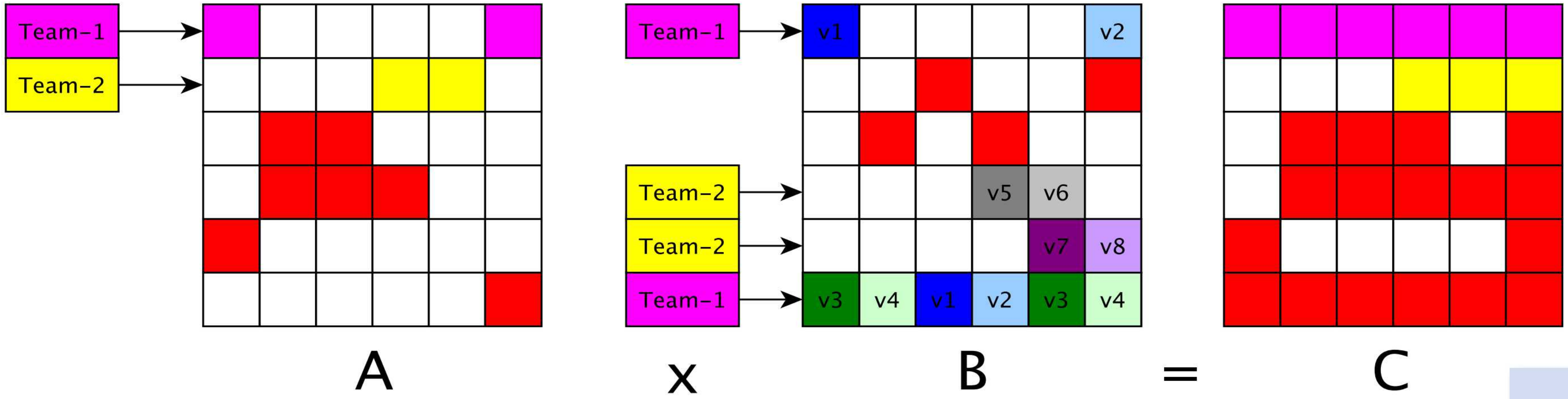
- No atomics needed
- Load balancing could be a problem between teams

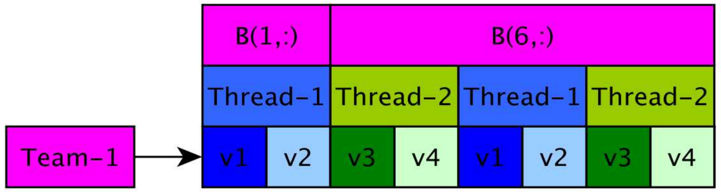
Thread Mapping (continued)



Threads to rows of B

- Team level synchronization needed
- Load balancing could be a problem





Teams to rows of A, unroll all computation to threads

- Team level synchronization needed

Kokkos allows exploring different styles of hierarchical parallelism

Data Structures for SpGEMM

Two-level Hashmap Accumulator:

- 1st level accumulator: GPUs shared memory or a small memory that will fit in L1 cache
- 2nd level goes to global memory

Memory Pool: Only some of the workers need 2nd level hash map. They request memory from memory pool.

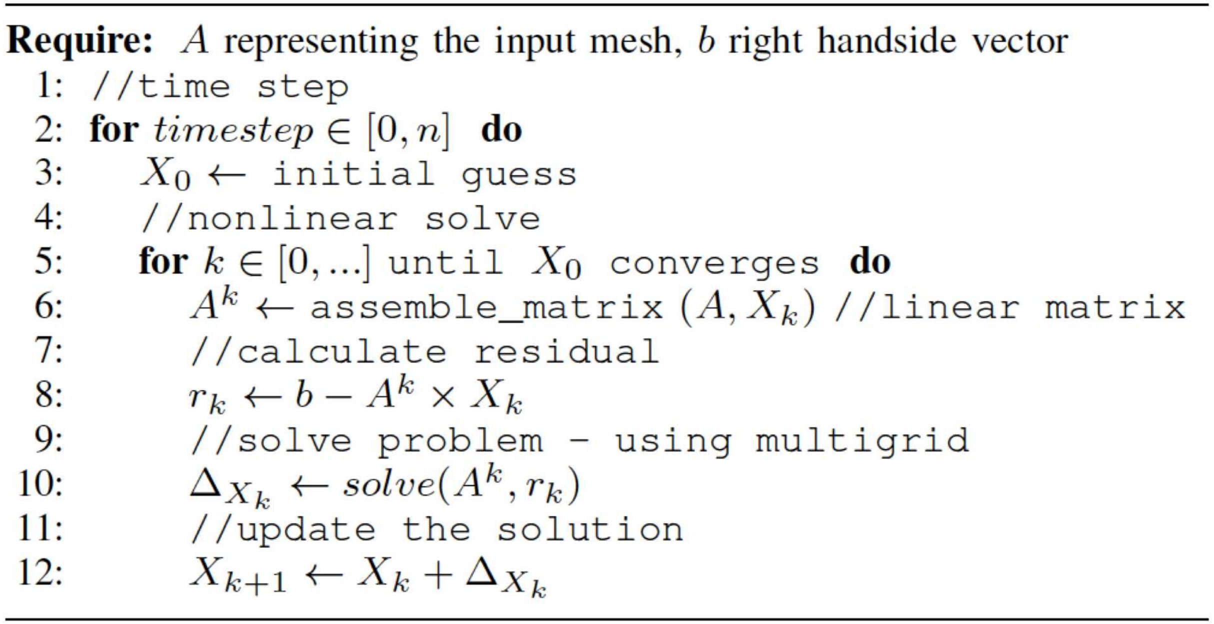
- Fixed size, fixed alignment

```
#pragma omp parallel
{
  data_type *my_data = new data_type[n];
  //initialize my_data ---> O(n)
  //once O(n) per thread
#pragma omp for
  for (i = 1...m){
    //work on my_data ---> O(k) and k << n
    //re-initialize my_data ----> O(k)
  }
}
```

Limitations of Accelerators require two phase SpGEMM

Size and structure of rows are unknown at the beginning

- over-allocation: expensive
- dynamically increase: not suitable to GPUs
- Estimation methods: not cheaper than calculating the actual size in practice



```
Require: $A$ representing the input mesh, $b$ right handside vector
//time step
for $timestep \in [0, n]$ do
    $X_0 \leftarrow$ initial guess
    //nonlinear solve
    for $k \in [0, ...]$ until $X_0$ converges do
        $A^k \leftarrow$ assemble_matrix $(A, X_k)$ //linear matrix
        //calculate residual
        $r_k \leftarrow b - A^k \times X_k$
        //solve problem - using multigrid
        $\Delta_{X_k} \leftarrow solve(A^k, r_k)$
        //update the solution
        $X_{k+1} \leftarrow X_k + \Delta_{X_k}$
```

Two-phase:

symbolic - calculate #nnz

then numeric - actual flops

Repetitive multiplications for different numeric values with same symbolic structure

Kokkos Kernels Two phase SpGEMM

Doubles the amount of work performed

Symbolic phase: works on the symbolic structure – no floating values

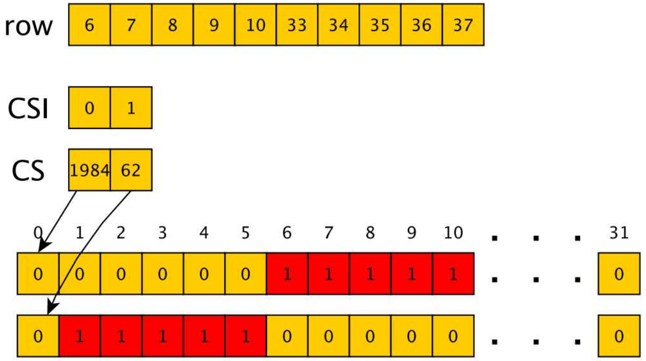
- performs unions on rows to find the structure/size of the output row
- compression method to speedup first phase and reduce its memory requirements

Compression: Compress the rows of B: O(nnz(B)) using 2 integers.

- Column Set Index (CSI): represents column set index
- Column Set (CS): the bits represent the existence of a column

Advantages:

- Symbolic complexity: O(FLOPS) -> on average ~O(avgdeg(A)x nnz(B))
- How much memory we need is unknown
- and overestimated as max row flops



row: 6 7 8 9 10 33 34 35 36 37

CSI: 0 1

CS: 1984 62

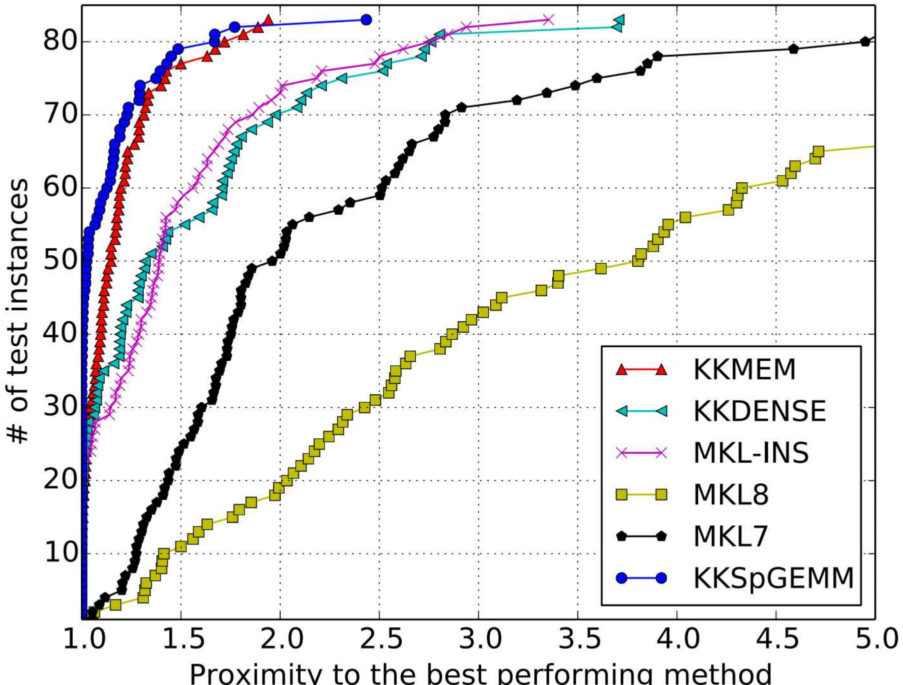
0 1 2 3 4 5 6 7 8 9 10 . . . 31

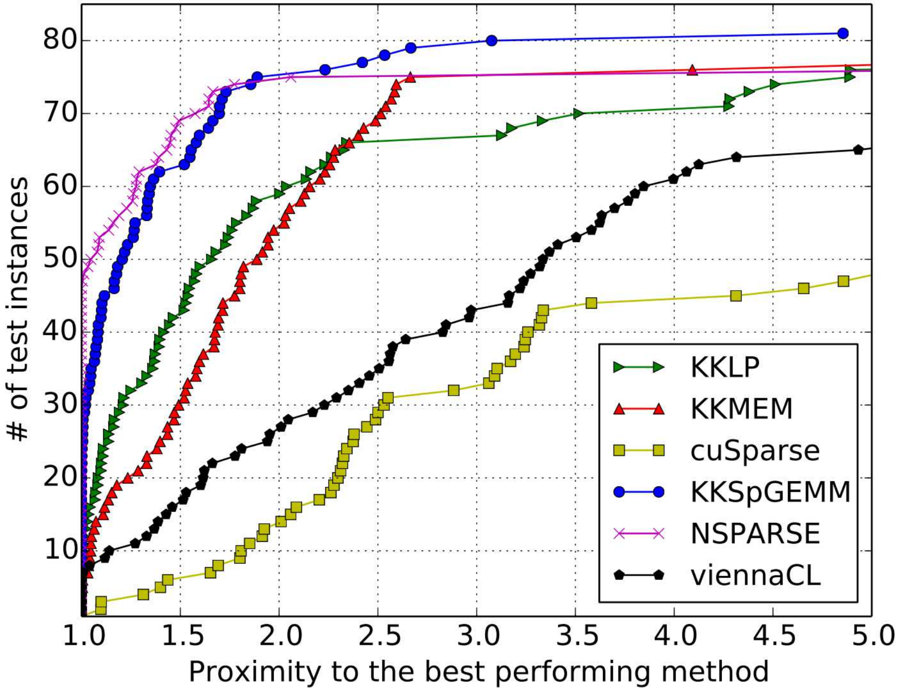
0 0 0 0 0 0 1 1 1 1 1 . . . 0

0 1 1 1 1 1 0 0 0 0 0 . . . 0

Kokkos Kernels Two phase SpGEMM Performance

- Integration:
  - Integrated into the Tpetra package of Trilinos and used by the multigrid solver MueLU within a distributed memory sparse matrix-matrix multiply
  - Integration with several Exascale Computing Project applications (ExaWind, EMPIRE)
- Performance Profile – 82 different instances of SpGEMM
  - Quicker and higher better. KNL DDR mode (left): KK-SpGEMM is the best for ~50 test instances, within 1.5x of the best instance for all but 3 instances. GPU (right): KKSpGEMM and NSPARSE are the two best methods.





Kernel 3: Team Level BLAS in Kokkos Kernels

Common hierarchical-parallel pattern in Kokkos based applications

- *parallel_for* over some entities (rows, elements, vertices, particles)
- BLAS/LAPACK call is team-level / serial call within the *parallel_for* with vectorization
- The device level loop is much larger than the BLAS call underneath. It is **not** one BLAS call, it is a set of BLAS calls and other operations together in a larger kernel.
- "Standard" Batched BLAS as proposed by the reference implementation introduces synchronization and data movement for these use cases
- Kokkos Kernels provides BLAS / LAPACK functionality at the team-level / serial level

Some exceptions

- Machine learning use cases where a layer could be fully expressed and utilize the device level concurrency
- "Standard" batched BLAS interface provides the required functionality here
- Kokkos Kernels can provide interfaces to vendor kernels and libraries like MAGMA

Kokkos Kernels provides team-level and serial BLAS / LAPACK functionality to be used within a parallel application context.

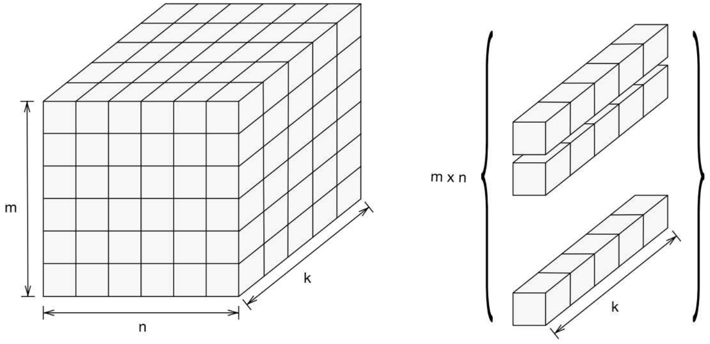
Kokkos Kernels interface to BLAS/LAPACK – Other considerations

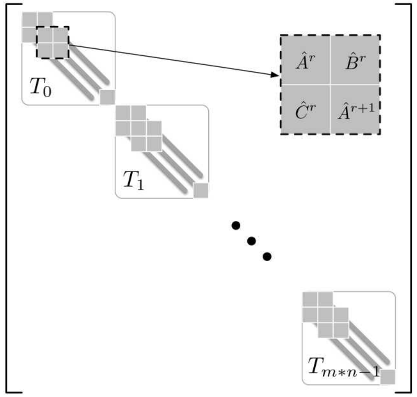
Compact Layout

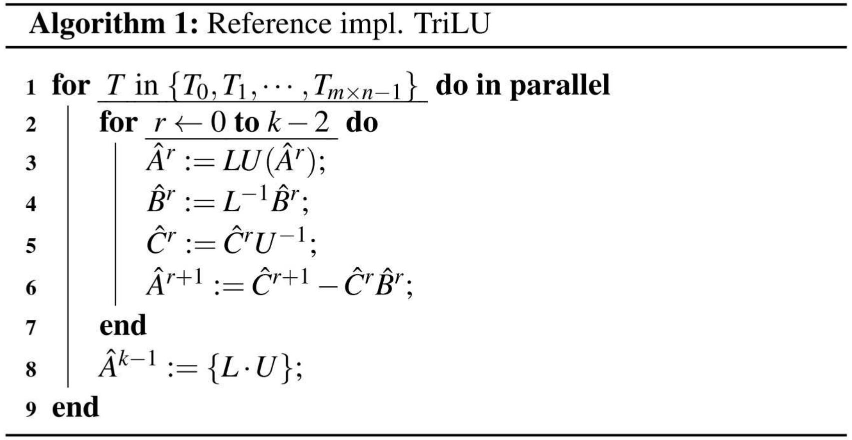
- Repack the matrices to be stored in compact layout for small matrix sizes in the batch
- Use traditional layouts for large matrix sizes or based on architectures
- Purely a memory access issue
  - Vectorized reads/writes
  - Coalesced memory access on a GPU
- See work by Kokkos Kernels and Intel MKL team (SC'17) for the usefulness of compact layout
  - Also implemented in Intel MKL

Usage of different memory layouts based on problem sizes and architecture needs is critical for performance

Application Use Case : Tridiagonal solvers







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
```

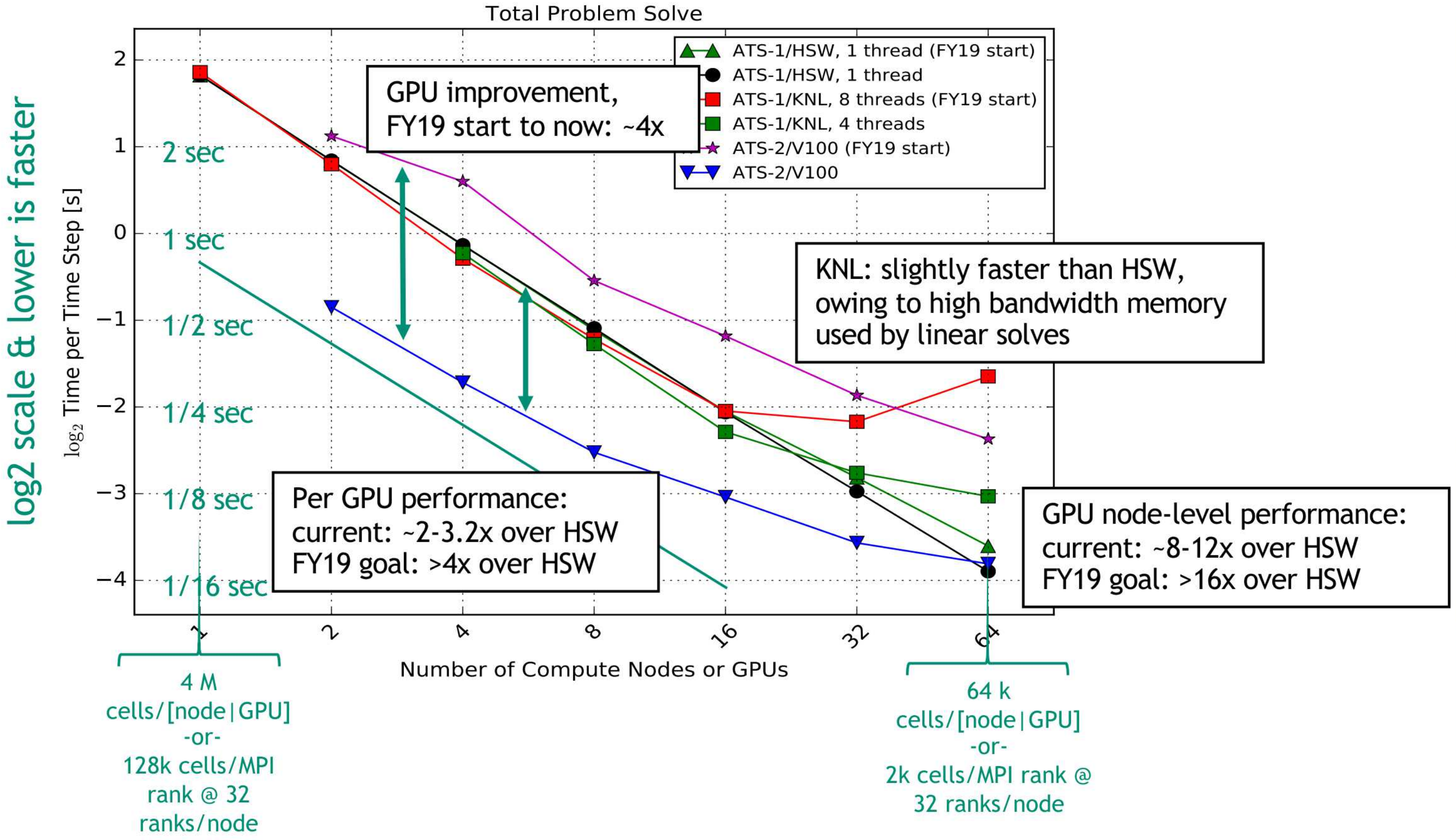
- Application characteristics
  - One dimension of the mesh more important than the others when preconditioning
  - Multiple degrees of freedom per element gives rise to tiny blocks

- 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

It is important to define BLAS and LAPACK kernels within the parallel regions

Developing line solvers for a CFD code

- Team Level BLAS Kernels
- Different from community developed standards for Batched BLAS at the device level
- Proposed as part of the community standard and being developed as part of multiple implementations



GPU improvement, FY19 start to now: ~4x

KNL: slightly faster than HSW, owing to high bandwidth memory used by linear solves

Per GPU performance: current: ~2-3.2x over HSW FY19 goal: >4x over HSW

GPU node-level performance: current: ~8-12x over HSW FY19 goal: >16x over HSW

4 M cells/[node|GPU] -or- 128k cells/MPI rank @ 32 ranks/node

64 k cells/[node|GPU] -or- 2k cells/MPI rank @ 32 ranks/node

- Optimized, vectorized implementation in Kokkos Kernels for Intel CPU, KNL and GPU platforms
- Line solvers based on compact kernels and integrated the compact BLAS based preconditioners in CFD code

Kokkos Kernels Capabilities

Dense Linear Algebra

- Good coverage of BLAS + LAPACK
- Team level kernels – coverage based on application requirement
- Complex support
- Tuned for problem sizes

Sparse Linear Algebra

- Sparse matrix-vector multiplication, Sparse matrix-matrix multiplication,
- Sparse Triangular solves,
- Preconditioners – Gauss-Seidel Preconditioner, ILU(k) preconditioner

Graph Algorithms

- D-1 coloring, D-2 coloring
- Triangle Counting

Data Structures

- Hash Map, Memory Pool
- Team Level Sorting

Machine Learning Kernels

- 2D and 3D Convolutions

Capabilities : BLAS

abs(y,x)	y[i] = |x[i]|
axpy(alpha,x,y)	y[i] += alpha * x[i]
axpby(alpha,x,beta,y)	y[i] = beta * y + alpha * x[i]
dot(x,y)	dot = SUM_i (x[i] * y[i])
fill(x,alpha)	x[i] = alpha
mult(gamma,y,alpha,A,x)	y[i] = gamma * y[i] + alpha * A[i] * x[i]
nrm1(x)	nrm1 = SUM_i(|x[i]|)
nrm2(x)	nrm2 = sqrt (SUM_i(|x[i]| * |x[i]|))
nrm2w(x,w)	nrm2w = sqrt (SUM_i((|x[i]|/|w[i]|)^2))
nrminf(x)	nrminf = MAX_i(|x[i]|)
scal(y,alpha,x)	y[i] = alpha * x[i]
sum(x)	sum = SUM_i(x[i])
update(a,x,b,y,g,z)	y[i] = g * y[i] + b * y[i] + a * x[i]
gemv(t,alph,A,x,bet,y)	y[i] = bet*y[i] + alph*SUM_j(A[i,j]*x[j])
gemm(tA,tB,alph,A,B,bet,C)	C[i,j]=bet*C[i,j]+alph*SUM_k(A[i,k]*B[k,j])

KokkosKernels Interface

1. KokkosKernels BLAS functions

- Convert from hierarchical parallel execution to using BLAS functions
- **tmp** = **A***x** (extra view, holds **gemv** results)
- result = <**y**,**tmp**>

2. KokkosKernels team-based BLAS function

- Same as hierarchical parallel execution
- Call team-based **dot** within each team to perform <A[teamId,:],x>

```
result = KokkosBlas::dot(x,y)
```

performs `result = SUM_i(y[i]*x[i])`

```
KokkosBlas::gemv(“N”,alpha,A,x,beta,y)
```

performs matrix-vector multiplication
`y[i] = beta*y[i] + alpha*SUM_j(A[i,j]*x[j])`

```
KokkosBlas::Experimental::dot(teamId,x,y)
```

performs `result = SUM_i(y[i]*x[i])` within each thread team

KokkosKernels Interface : Conjugate Gradient Solver

- Exercise can be found as part of the Kokkos Tutorials
- Goal: implement conjugate gradient solver for square, symmetric, positive-definite sparse matrix
- Details: **A*x=b**
  - **b** is *Nx*1
  - **A** is *NxN* symmetric, positive-definite sparse matrix
  - **x** is *Nx*1
  - Look for comments labeled with "EXERCISE"
  - Use KokkosKernels BLAS and KokkosKernels Sparse BLAS

```
$\mathbf{r_0} = \mathbf{b} - \mathbf{A} * \mathbf{x_0}$
$\mathbf{p_0} = \mathbf{r_0}$
$k = 0$
while $\|\mathbf{r}_k\| > \varepsilon$ and $k<N$
    $\alpha = \frac{\mathbf{r}_k^T * \mathbf{r}_k}{\mathbf{p}_k^T * \mathbf{A} * \mathbf{p}_k}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha * \mathbf{p}_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha * \mathbf{A} * \mathbf{p}_k$
    $\beta = \frac{\mathbf{r}_{k+1}^T * \mathbf{r}_{k+1}}{\mathbf{r}_k^T * \mathbf{r}_k}$
    $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta * \mathbf{p}_k$
    $k = k + 1$
```

KokkosKernels Interface : Conjugate Gradient Solver

- Sparse matrix generation is provided
- Compile and run on OpenMP, CUDA backends
- Vary problem size: -N #
- Compare performance of CPU vs GPU

KokkosKernels Functions

```
result = KokkosBlas::dot(x,y)
```

performs `result = SUM_i(y[i]*x[i])`

```
KokkosBlas::axpy(alpha,x,y)
```

performs `y[i] = y[i] + alpha*x[i]`

```
KokkosBlas::axpby(alpha,x,beta,y)
```

performs `y[i] = beta*y[i] + alpha*x[i]`

```
KokkosSparse::spmv("N",alpha,A,x,beta,y)
```

performs sparse matrix-vector multiplication
`y[i] = beta*y[i] + alpha*SUM_j(A[i,j]*x[j])`

Collaborations with Vendors

NVDIA

- Summit on Summit meetings
- Biweekly work stream meetings to guide NVIDIA's math libraries plans
- DOE wide effort, Kokkos Kernels requirements prioritized along with other labs
- Kernel requirements prioritized by application needs and milestones
- Long history of interaction as part of COE

AMD

- Just started the interactions on sparse, dense, batched linear algebra kernels, and sparse solvers
- Kokkos backend under-development
- Kokkos Kernels will be the performance test case

Intel

- Long history of interaction as part of COE, Aurora plans
- Kokkos backend under development
- Kokkos Kernels will be the performance test case

ARM

- Working with the math libraries team both on algorithms and prioritization

Kokkos Kernels team working with hardware vendors to support application needs on current and exascale platforms

Collaborations with ECP Applications

SPARC: state-of-the-art hypersonic unsteady hybrid structured/unstructured finite volume CFD code

- **High performance line solvers; batched BLAS on CPUs and GPUs**
- **Performance-portable programming models**

EMPIRE: next-gen unstructured-mesh FEM PIC/multifluid plasma simulation code

- Scalable solvers for electrostatic and electromagnetic systems for Trinity and Sierra architectures
- **Thread-scalable, performance-portable, on-node linear algebra kernels to support multigrid methods**
- **Performance-portable programming models**
- Non-linear solvers, discretization, and automatic differentiation approaches

Exawind: next-gen wind simulation code

- **Scalable solvers for Trinity and Sierra architectures**
- **Thread-scalable, performance-portable, on-node linear algebra kernels to support multigrid methods**
- **Performance-portable programming models**

QMCPACK: Electronic structure code with Quantum Monte Carlo Algorithms

- Team level BLAS and LAPACK within the Kokkos ecssytem

Kokkos Kernels integrated into several applications in an agile manner at all stages from understanding requirements, designing kernels and integrating them.

Rules of Thumb for library based approach to accelerator programming with performance portability

Identify performance critical kernels

- Call library based option when possible
  - Allows library developers and optimize the kernels to the best extent possibl

Develop portable algorithms when library based option not available

- Use a portable programming model or directive based approach
- Use architecture independent abstractions
- Pay attention to memory layouts, hierarchical parallelism, synchronization costs

Use team level data structures and linear algebra kernels when possible

- Optimize performance at all the hierarchical levels