SAND2016-3136C

Gradually porting an in-use sparse matrix library to use CUDA

Mark Hoemmen
Center for Computing Research
Sandia National Laboratories

06 Apr 2016

Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

National Nuclear Security Administration

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. SAND NO. 2011-XXXXP

Outline

Sandia National Laboratories

- Our sparse matrix library: Tpetra (Trilinos project)
- Goal: Support MPI + X parallelism on current & future architectures. X: threads, not just CUDA.
- Genericity (“X”) via Kokkos programming model
- Past, current, & future gradual porting timeline
- Strategies for dealing with specific hardware features

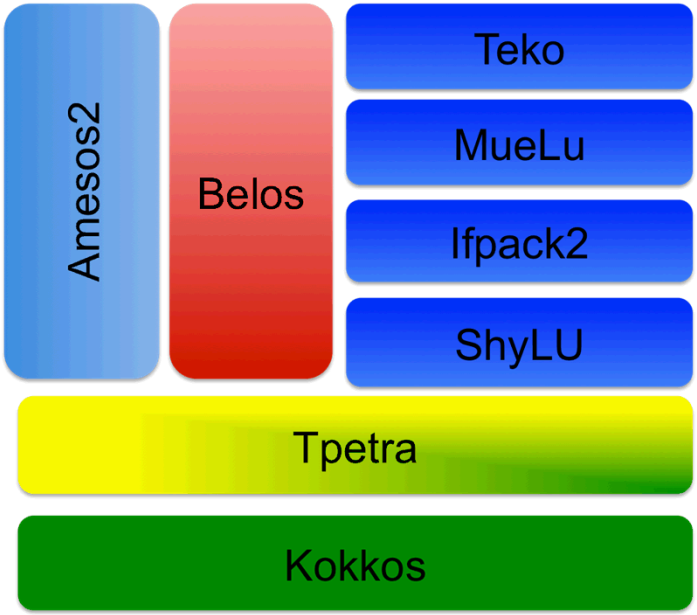
What is Trilinos?

Sandia National Laboratories

- Software project: trilinos.org, github.com/trilinos/Trilinos
- For solving large math problems using parallel computers
- Numerical simulations for science & engineering (esp. solving partial differential equations); graph & data analysis
  - Focus on creating & solving large, sparse linear systems
- For both research (algorithms, math, computer science) & development (applications both internal & external)
- Package architecture: dozens of packages, managed separately – can use as little or as many as you want
- Mostly C++; some C, Fortran, Python

Trilinos' linear solvers

Sandia National Laboratories

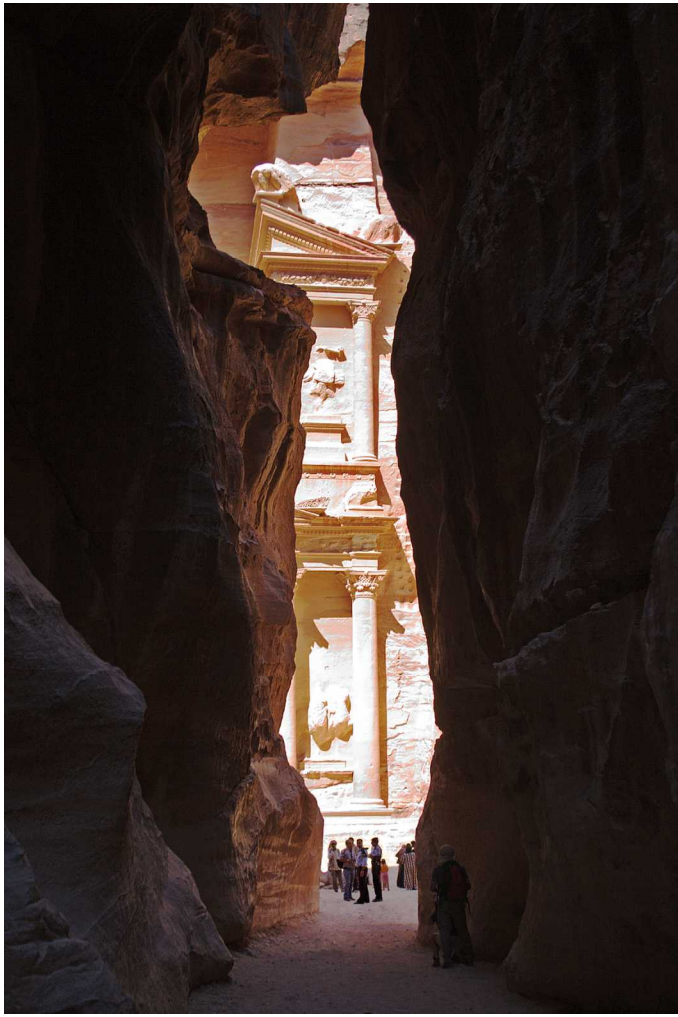


- Sparse linear algebra (Tpetra)
  - Sparse graphs, (block) sparse matrices, dense vectors, parallel solve kernels, parallel communication & redistribution
- Iterative (Krylov) solvers (Belos)
  - CG, GMRES, TFQMR, recycling methods
- Sparse direct solvers (Amesos2)
- Algebraic iterative methods (Ifpack2)
  - Jacobi, SOR, polynomial, incomplete factorizations, additive Schwarz
- Shared-memory factorizations (ShyLU)
  - LU, ILU(k), ILUt, IC(k), iterative ILU(k)
  - Direct+iterative preconditioners
- Segregated block solvers (Teko)
- Algebraic multigrid (MueLu)

What is Tpetra?

Sandia National Laboratories

- Tpetra implements
  - Sparse graphs, (block) sparse matrices, & dense (multi)vectors
  - Parallel kernels for solving Ax=b & Ax=λx
  - Parallel communication & (re)distribution
- Key Tpetra features
  - Can solve problems with over 2 billion (10^9) unknowns
  - Can pick the type of values:
    - Real, complex, extra precision
    - Automatic differentiation
    - Types for stochastic PDE discretizations
  - Center of growing support for MPI + X parallelism, for several X



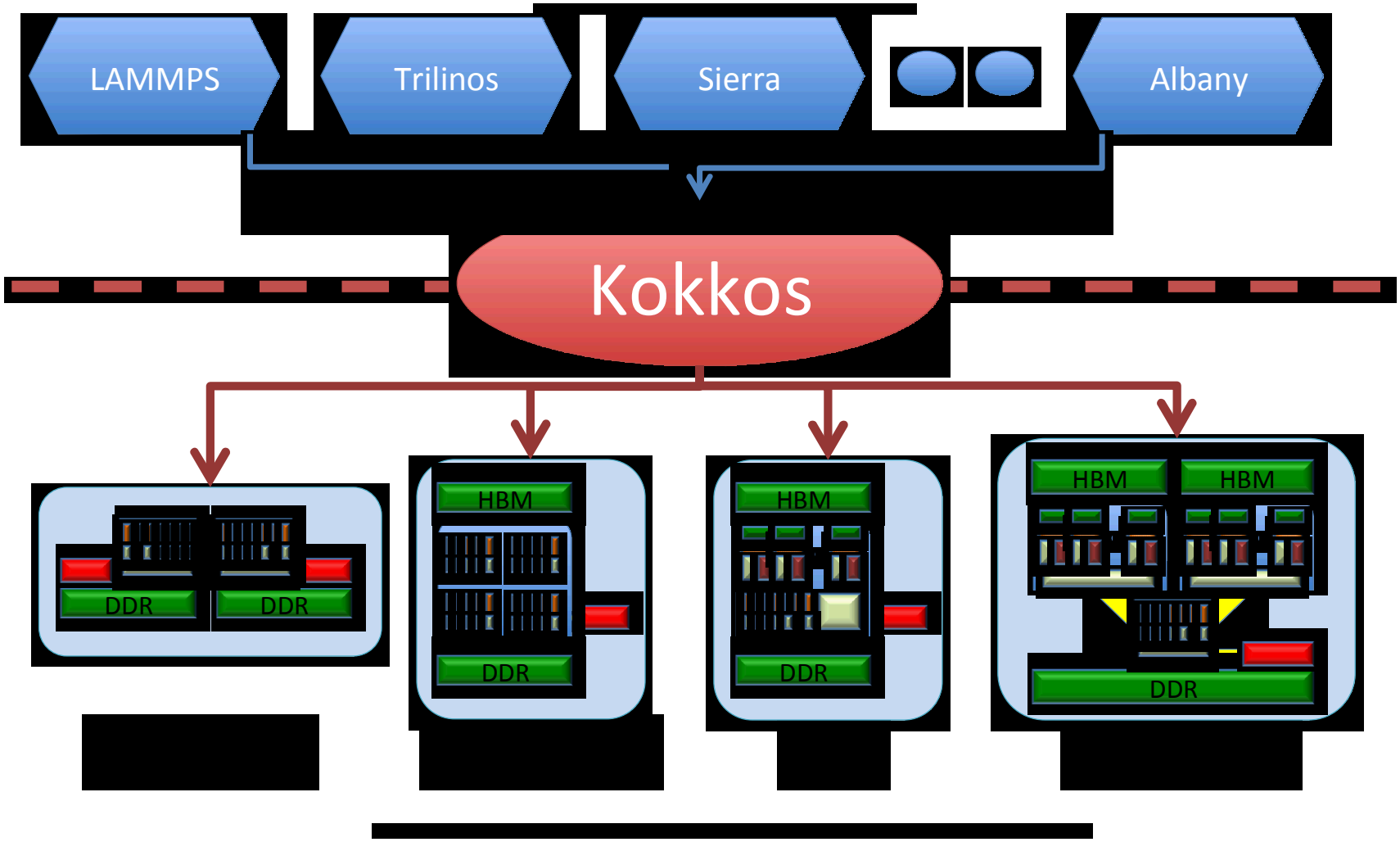
Tpetra development goals

Sandia National Laboratories

- 1 implementation for all platforms & parallelism options
  - Very limited developer time (1 full-time staff & some fractions)
  - Requirement: scale from laptop to full supercomputer
  - Easier to debug solver (convergence) & performance issues
- Maintain backwards compatibility
  - Trilinos only allows breaking it at major releases (every 1-2 years)
  - Must balance research, prep for future / oncoming hardware, & support today's apps (often running on old hardware & software)
  - Both apps & other Trilinos packages use Tpetra directly & heavily
  - Interfaces (create & fill) matter for performance & parallelism
- Exploit optimized kernels but minimize library dependencies
  - We need our own implementations that work everywhere
  - 3rd-party libraries often ignore features needed for MPI or apps

Kokkos: *Performance, Portability, & Productivity*

Sandia National Laboratories



Kokkos programming model

Sandia National Laboratories

- Parallel patterns: for, reduce, scan
  - Custom reduction ops & types (unlike OpenMP)
  - Reproducible reductions & scans (unlike OpenMP)
- Different kinds of parallelism
  - Simple [0,N) range
  - Hierarchical: thread teams (OpenMP 4 / CUDA) & vector
- Different memory & execution spaces
  - Control where data live & code executes
  - Manual “hybrid” parallelism (e.g., host + GPU)
  - Write code once, run on many different back-ends
- Multidimensional arrays
  - Default layout optimized for the memory space (SoA / AoS)
  - User-controlled layout (for library compatibility or performance)

Kokkos as hedge against…

Sandia National Laboratories

- Hardware heterogeneity
- A particular shared-memory programming model
  - OpenMP, OpenACC, CUDA, TBB, Pthreads, Qthreads, …
- Traditional shared memory (vs. PGAS / distributed shared)
  - No coherency requirements; could use 1-sided comm for atomics
  - Kokkos::View could wrap MPI_Win, Global Arrays, UPC shared, …
  - Permits async parallel execution ➔ doesn't require fork-join
- Threads at all
  - Kokkos::Serial (no threads) is a valid execution space
  - Kokkos' semantics require vectorizable (ivdep) loops
  - Kokkos::View can pad for alignment, & declare it
  - Many hooks for passing info (e.g., dimensions) to compiler

Keys to gradual CUDA porting

Sandia National Laboratories

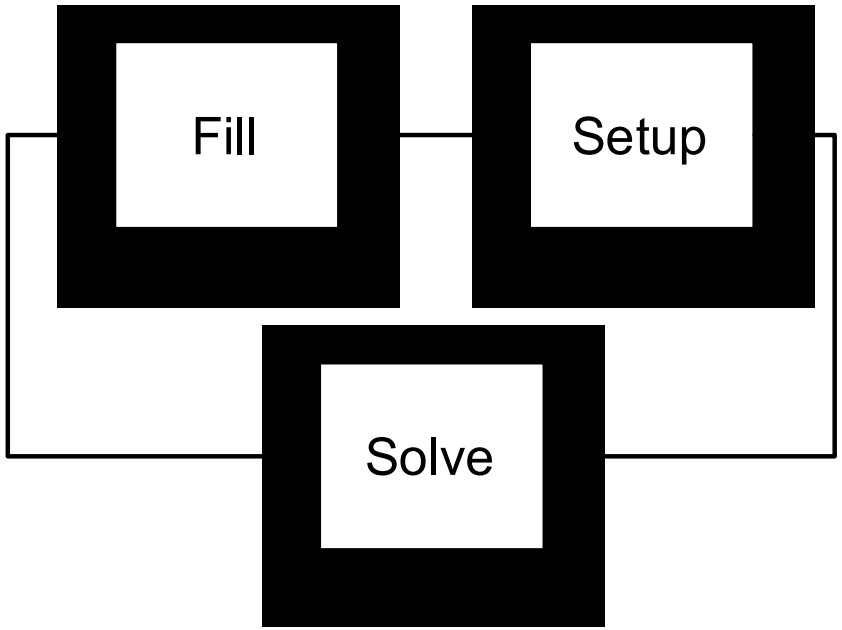
- Abstract away memory allocation & deallocation
  - Kokkos::View (multidimensional array) as building block
  - Rely on C++ inlining for performant array access
  - Automatic memory management avoids unnecessary deep copies
- Abstract away data-parallel loops & computational kernels
  - Loops: Kokkos::parallel_{for, reduce, scan}
    - I write loop body as functor or C++11 lambda (new CUDA feature)
    - Kokkos semantics force me to write vectorizable & parallelizable loops
  - Computational kernels: “KokkosKernels” Trilinos package
- Manage data movement between memory spaces
  - Kokkos’ abstractions (host mirrors, deep_copy, DualView) let me write as if I’m always on an NVIDIA GPU; w/ no performance cost elsewhere
  - CUDA UVM means I can port one kernel at a time

Sparse linear algebra use pattern

Sandia National Laboratories

- Fill: Create / modify matrix & vector data structures
  - As many ways to do this as there are applications
  - e.g., iterate over rows, entries, mesh points, elements (FEM), volumes (FVM), aggregates (AMG), …
  - Software interfaces affect performance A LOT
- Setup for solve (e.g., build preconditioner)
- Solve linear system(s), eigenvalue problems, etc.
  - Coarse-grained computational kernels (e.g., sparse mat-vec)
  - Software interfaces affect performance less
- Repeat (nonlinear iteration, time steps, parameter study, …)
  - Trilinos data structures & solvers optimized for reuse, e.g., of
    - Data structures (graph, basis vectors, allocations) &/or
    - Communication patterns

Need thread-parallel fill



- Fill & setup not free
- Some solves are cheap, so fill & setup time matter
- Most jobs use few nodes
- Amdahl's Law:
  - Solves get threaded first
  - But: 90% time w/ 1 thread ➔ 50% time w/ 10 threads
- Preconditioners create sparse matrices, so they also need fill

Thread-parallel fill options

Sandia National Laboratories

Coarse-grained (batched)

- Pass many finite elements, cliques, etc. into linear algebra interface
- Library parallelizes inside
- Need not be thread-safe
- Doesn't actually solve the problem: User still must set up input (e.g., do the finite element method) in parallel

Fine-grained

- 1 item (row, elt, …) at a time
- User parallelizes outside; library promises no thread scalability issues
- Limited parallelism possible inside (e.g., vectorization)
- Tpetra's choice

A brief history of Tpetra

Sandia National Laboratories

- 2008: Interface defined
- 2009: Initial thread parallelism (computational kernels only)
- 2009-2010: Initial efforts at adding preconditioners
- I started at Sandia in 2010
- "Productionization" (team effort): 2011-2013
  - Fix bugs & improve performance of (single-threaded) solvers & fill
  - We integrated into an internal engineering numerical simulation
  - Fruits of our effort in Nalu: https://github.com/spdomin/nalu
- "Kokkos refactor": Late 2013 – present
  - Stage 1 (FY14-15): Keep interface, replace data structures & kernels
  - Stage 2 (FY15-now): Change interface for thread-parallel fill; finish kernels

Fill in 2012 was not thread-scalable

Sandia National Laboratories

- Dynamic memory allocation ("dynamic profile")
  - Impossible in some parallel models; slow on others
  - Allocation implies synchronization (must agree on pointer)
  - Better: Count, Allocate (thread collective), Fill, Compute
- Throw C++ exceptions on error / when out of space
  - Either doesn't work (CUDA) or hinders compiler optimization
  - Prevents fruitful retry in (count, allocate, fill, compute)
  - Better: Return success / failed count; user reduces over counts
- Unscalable reference counting implementation
  - Teuchos::(Array)RCP: like std::shared_ptr but not thread safe
  - Not hard to make thread *safe*, but updating the ref count serializes!
  - Better: Use Kokkos::View's thread-safe count; prefer unmanaged View

Tpetra had no plan for parallel fill

Sandia National Laboratories

- Must fill Tpetra data structures on host, sequentially
  - Common way to access data: host copy ("generalized view")
    - Read-only: Host copy of device data
    - Read-write: Host copy, copies back to device at ref count 0
  - "Device view" was expert mode, never used outside Tpetra
  - Sparse matrix data vanished into an opaque data structure; repeated linear solves w/ different matrix values but same structure (common case) required keeping a host copy
- Teuchos::{RCP, ArrayRCP} ref count not thread safe
  - Tpetra stored & returned everything (e.g., Maps, CrsGraph) by RCP
  - In debug mode, even Teuchos::ArrayView ref-counts
  - Can't use device buffers (ArrayRCP) in parallel kernels

Kokkos refactor of Tpetra

Sandia National Laboratories

Goals of Kokkos refactor

Sandia National Laboratories

- Make thread-parallel fill correct & fast
- Adjust familiar fill interfaces; consider new ones
- Make it easier to add thread-parallel computation kernels
  - If you wanted just sparse mat-vec & AXPY, use Epetra w/ OpenMP
  - Christian's Aug 2013 sparse mat-vec performance plot encouraged me

Refactor plan: Stage 1 (FY14-15)

Sandia National Laboratories

- Replace all internal data structures & kernels w/ Kokkos
  - Sparse matrix-vector multiply & vector ops first
  - Later, we factored out local kernels into KokkosKernels
- Assume CUDA UVM but aim to remove UVM assumption
- Maintain interface backwards compatibility when possible
- For Stage 1, if you wanted thread-parallel fill, you had to fill into Kokkos data structures, & hand off to Tpetra
- View semantics (shallow copy), just like Kokkos::View
  - Makes thread-scalable operations w/ Tpetra objects easier
- Partial specialization let “Classic” & “Refactor” coexist
- “Classic” could build with older compilers (no need for C++11)

Thread-parallel fill still primitive

Sandia National Laboratories

- Stage 1 plan: “If you want thread-parallel fill, get the Kokkos widget out of the Tpetra object, & fill into that”
- Advantages
  - Preserve Tpetra interface backwards compatibility
  - Let Tpetra developers work gradually
- Disadvantages
  - Appears to defeat a major purpose of switching to Tpetra
  - “Kokkos widget” is yet another public interface to support
  - Users already complained that Tpetra was hard to use
    - 3 different namespaces (Tpetra, Kokkos, Teuchos)

Stage 2 (FY15-16): Thread-safe fill

Sandia National Laboratories

- Done for CrsMatrix & (Multi)Vector, for methods that
  - Don't change graph structure (no "insert" yet)
  - Don't cause MPI communication (+= values for off-process rows)
- Return error code / success count; don't throw on error
- No more internal temp array dynamic allocation
- sumInto, transform: Atomic update option
  - Default: Use atomic updates if not Serial
- sumInto, replace, transform: Take Kokkos::View or raw arrays
  - Avoid Teuchos::ArrayView debug mode reference count issues

Pattern for parallel dynamic allocation

Sandia National Laboratories

- Pattern:
  1. Count / estimate allocation size; may use Kokkos parallel_scan
  2. Allocate; use Kokkos::View for best data layout & first touch
  3. Fill: parallel_reduce over error codes; if you run out of space, keep going, count how much more you need, & return to (2)
  4. Compute (e.g., solve the linear system) using filled data structures
- Compare to typical sparse linear algebra use pattern:
  - Fill linear system
  - Setup for solve (e.g., preconditioners)
  - Solve linear system
- Fortran <= 77 coders should find this familiar
- Semantics change: Running out of memory not an error!
  - Generalizes to other kinds of failures, even fault tolerance

Under development: KokkosKernels

Sandia National Laboratories

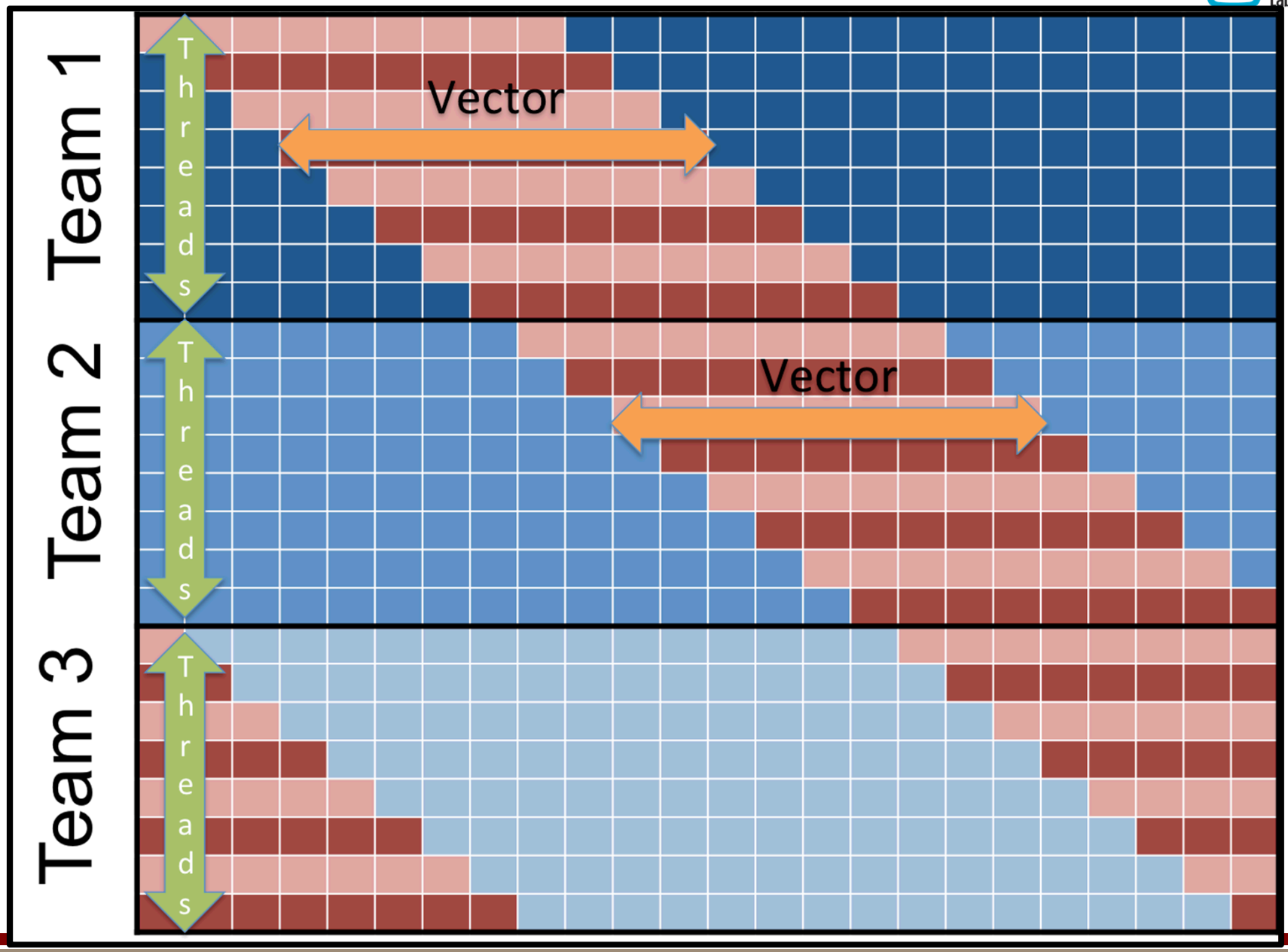
- Local computational kernels used by Trilinos, usable outside
  - Dense (BLAS 1,2,3), sparse, graph, & tensor kernels
  - Local (no MPI) – Trilinos / users responsible for MPI
  - No required software dependencies other than Kokkos
  - Hooks for 3rd-party libraries like cu{Blas,Sparse} if available
- Multi-year effort w/ many contributors, mostly Trilinos devs
- Provide kernels for all levels of hierarchical parallelism:
  - Global: all available execution resources (e.g., whole GPU)
  - Team-level: thread block / team, use shared memory
  - Thread-level: single “thread” / warp, vectorization inside
  - Serial: provide elemental functions (OpenMP declare SIMD)

Why kernels for different levels?

Sandia National Laboratories

- How often do real applications need a 10k x 10k DGEMM?
- Many apps do many small computations in parallel
  - e.g., Small dense matrix operations (BLAS & factorizations)
    - PDE discretizations w/ multiple unknowns per mesh point: ~10 x 10
    - Sparse matrix factorizations: ~100 x 100 (NOT square)
  - Users want to write code for a single element, point, particle, …
  - "Batch" interfaces lose locality when doing many ops to a single thing
  - Suits fine-grained parallel fill
- To use all hardware, need to exploit all levels of parallelism
- Need reusable kernels for those small operations

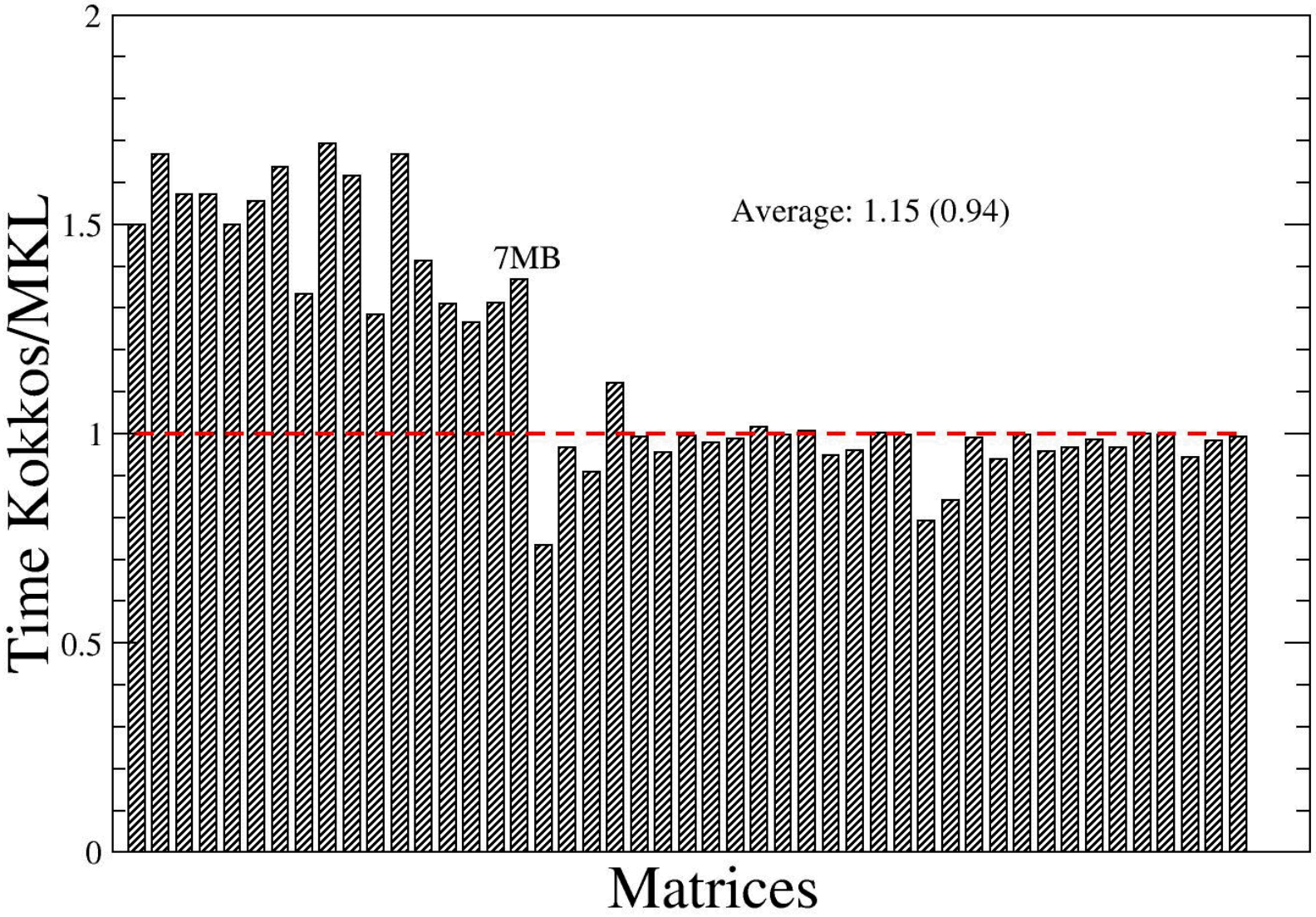
SPMV – Using Hierarchical Parallelism

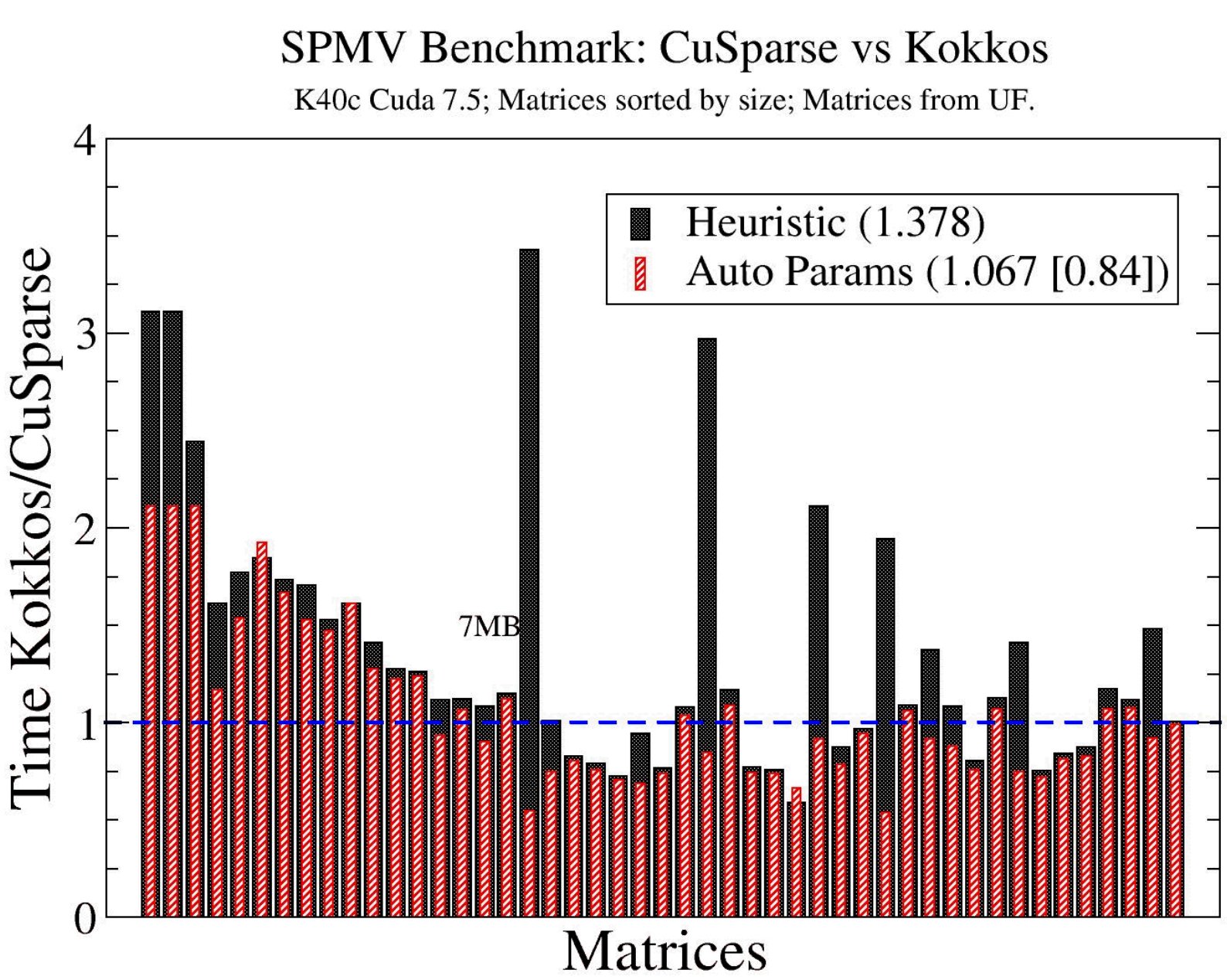


SPMV – Using Hierarchical Parallelism

Sandia National Laboratories

```
void spmv(Matrix A, Scalar alpha, XType x, Scalar beta, YType y) {
 int nnz_per_team = 2048;
 int conc = execution_space::concurrency();
 while((conc * nnz_per_team * 4> A.nnz())&&(nnz_per_team>256)) nnz_per_team/=2;

 int nnz_per_row = A.nnz()/A.numRows();
 int rows_per_team = (nnz_per_team+nnz_per_row-1)/nnz_per_row;
 int vector_length = GetVectorLength(A);
 const int nworkset = (y.dimension_0()+rows_per_team-1)/rows_per_team;

 parallel_for(TeamPolicy<Schedule<Dynamic> >(nworkset, AUTO(),vector_length),
  KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team)  {
  const int startRow = team.league_rank() * rows_per_team;
  const int endRow = startRow + rows_per_team < A.numRows() ?
                startRow + rows_per_team : A.numRows()

  parallel_for(TeamThreadRange(team,startRow,endRow), [&] (const int& loop) {
   const SparseRowViewConst<MatrixType,SizeType> row = A.template rowConst<SizeType>(iRow);
   const int row_length = row.length;
   Scalar sum = 0;

   parallel_reduce(ThreadVectorRange(team,row_length), [&] (const int& iEntry, Scalar& lsum) {
    const Scalar val = conjugate ?
          ATV::conj (row.value(iEntry)) :
          row.value(iEntry);
    lsum += val * x(row.colidx(iEntry));
   },sum);

   single(PerThread(team), [&] () {
    sum *= alpha;
    y(iRow) = beta * y(iRow) + sum;
   });
  });
 }
```


SPMV Benchmark: MKL vs Kokkos
1S HSW 24 Threads, Matrices sorted by size, Matrices obtained from UF
Time Kokkos/MKL
2
1.5
1
0.5
0
Average: 1.15 (0.94)
7MB
Matrices


SPMV Benchmark: CuSparse vs Kokkos
K40c Cuda 7.5; Matrices sorted by size; Matrices from UF.
Heuristic (1.378)
Auto Params (1.067 [0.84])
7MB
Time Kokkos/CuSparse
Matrices
0
1
2
3
4

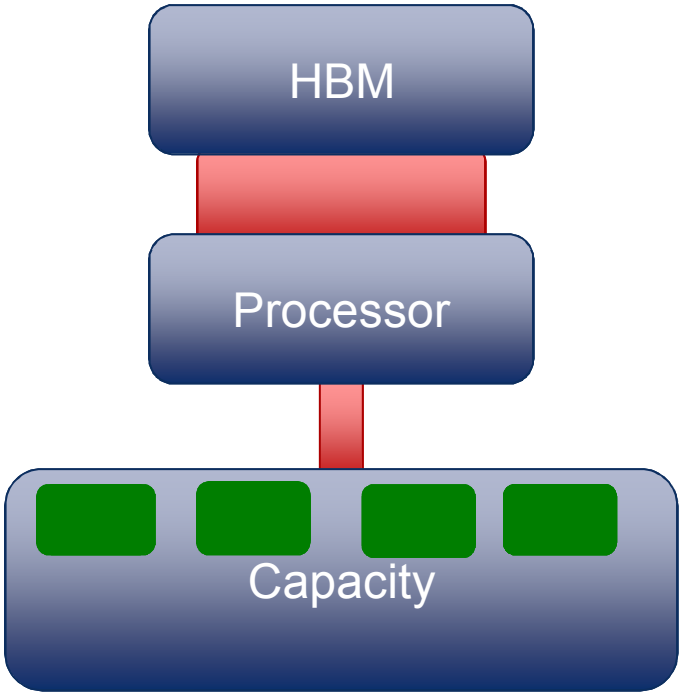
>1 memory or execution spaces

Sandia National Laboratories

- Upcoming NNSA platforms
  - Trinity (KNL): 2 memory spaces (HBM, DDR4)
  - CORAL (Sierra): 2 execution & memory spaces (NVIDIA GPUs, IBM multicore CPUs)
- Common hardware features
  - 2 memory spaces: “fast & small” vs. “slow & big”
  - Can access each memory space from each exec space (acts like NUMA)
  - “Fast” memory limited (<1GB/core); use as temp workspace

- Support via some comb of {Kokkos, Tpetra, solvers, app}
- Use cases to support
  - Gradual port (mix new & legacy)
  - Concurrently use 2 exec spaces (e.g., MPI pack & compute)

GPU / High-Bandwidth Memory



Cost Estimate (Bandwidth Bound):

Run From Main (capacity) Memory

Time = N_{iter} * Size / $BW_{Capacity}$

Run From HBM

Time = N_{iter} * Size / BW_{HBM} + Size / $BW_{Capacity}$

Expect

$BW_{HBM}/BW_{Capacity}$ ~ 5-20

Question: Generally need higher parallelism to achieve BW_{HBM} vs $BW_{Capacity}$

=> What about Direct Solvers?

Strategies for limited device memory

Sandia National Laboratories

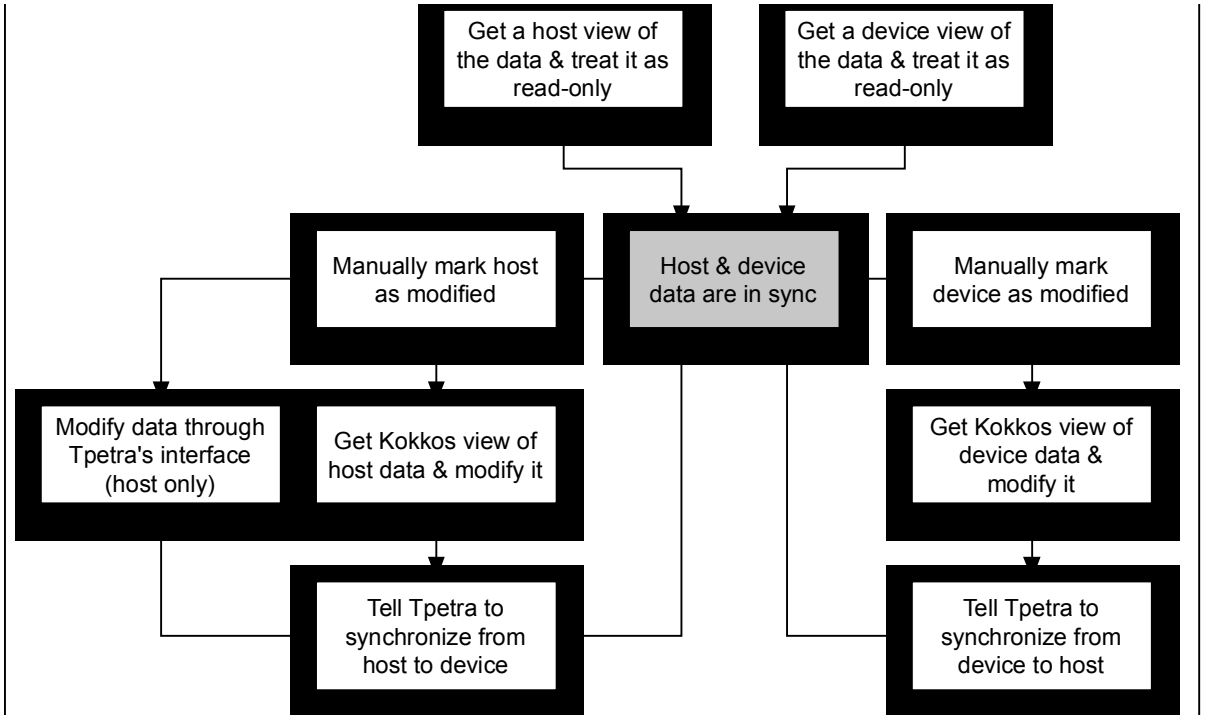
- All app data, even whole linear system, may not fit on GPU
- Prefer algorithmic solutions over auto-magic
  - Don’t want different libraries to need to arbitrate limited resource
  - Painful run-dependent debugging & performance variation
- Strategy 1: Stage in individual linear systems temporarily
  - Real physics is multiphysics ➔ solve multiple linear systems at same time
  - Works with block preconditioners or nonlinear (loose) coupling
  - Tpetra’s current interface could support this, w/ more impl work
- Strategy 2: Domain decomposition (divide up single solves)
  - Affects convergence; doesn’t work well for all linear systems
  - Subdomain solvers need enough reuse to amortize data transfer
  - Would take more software work to avoid e.g., data reformatting

Tpetra objects are “DualViews”

Sandia National Laboratories

- 1 *preferred* execution space, 2 memory spaces (“Host” & “Device”)
- Tpetra *may* execute in another space (e.g., overlap pack & compute)
- User sets “modified” flags & “syncs” explicitly between spaces
- Successful use in LAMMPS (interactions btw user vs. GPU modules)

“DualView” example: Vector



Tpetra objects act just like Kokkos::DualView.
Tpetra’s evolution of legacy fill interface is host only.
To fill on (CUDA) device, must use Kokkos interface.

If you only have one memory space, you can ignore all of this; it turns to no-ops.

Preferred use with two memory spaces:

1. Assume unsync’d
2. Sync to memory space where you want to modify it (free if in sync)
3. Get & modify view in that memory space
4. Leave the Tpetra object unsync’d

Next steps

Sandia National Laboratories

Next steps

Sandia National Laboratories

- Finish integrating new KokkosKernels into Trilinos' solvers
- Finish refactoring BlockCrsMatrix; new fill interface
- Standardize small dense / batched BLAS / LAPACK interface
- Thread-parallel graph construction
- Solidify Tpetra's handling of >1 memory / execution spaces
- Experiment w/ higher-level fill interfaces

Thanks!

Sandia National Laboratories

- Trilinos' thread parallelism has been / still is a HUGE effort
- Dozens of colleagues & collaborators have contributed