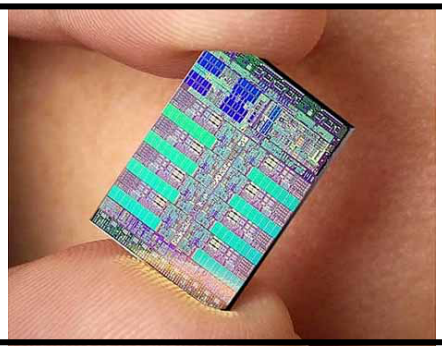
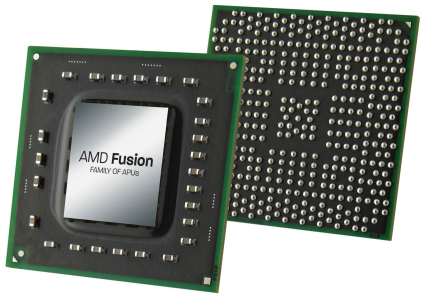
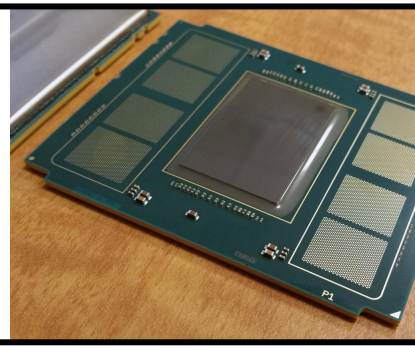
Exceptional service in the national interest

Sandia National Laboratories



KokkosKernels Overview

S. Rajamanickam,

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

srajama@sandia.gov

Center for Computing Research

Sandia National Laboratories, NM

Unclassified Unlimited Release

SAND2020-6470 TR

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

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-NA-0003525.

Presentation outline

Sandia National Laboratories

1. Introduction
2. Dense linear algebra: BLAS capabilities
3. Team level linear algebra: BatchedBLAS
4. Sparse linear algebra, preconditioners and graph kernels
5. Build system using Cmake

Presentation outline

Sandia National Laboratories

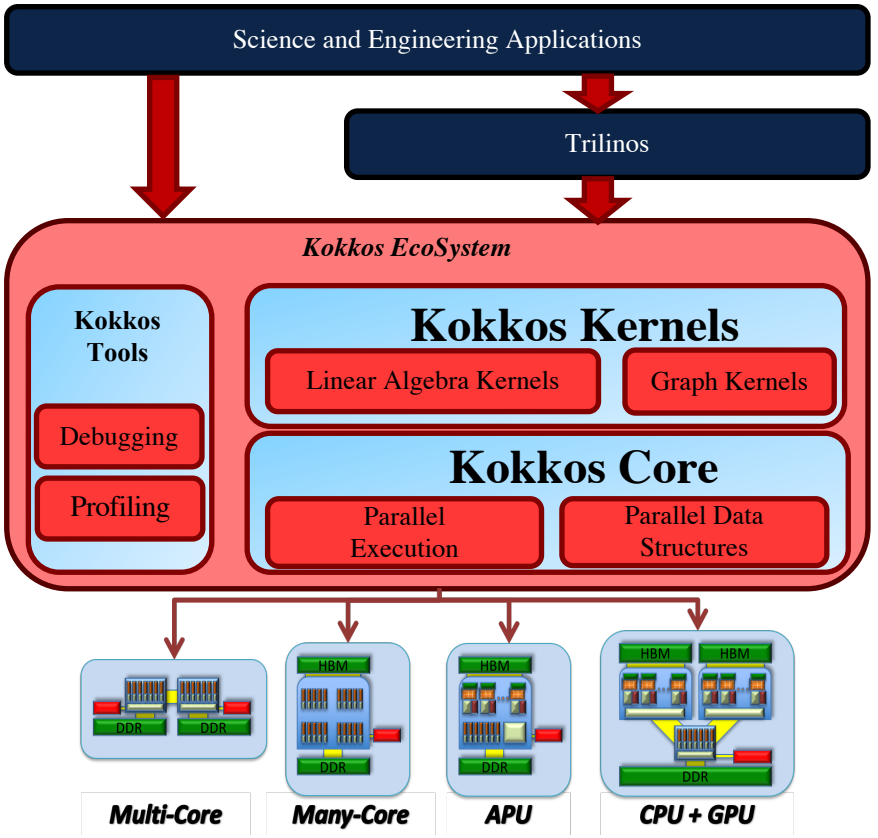
1. Introduction
2. Dense linear algebra: BLAS capabilities
3. Team level linear algebra: BatchedBLAS
4. Sparse linear algebra, preconditioners and graph kernels
5. Build system using Cmake

Goal of the Project

Sandia National Laboratories

KokkosKernels provides math kernels for dense and sparse linear algebra as well as graph computations. It has multiple aims:

- Portable BLAS, Sparse and Graph kernels
- Generic implementations for various scalar types and data layouts
- Access to major vendor optimized math libraries.
- Expand the scope of BLAS to hierarchical implementations.



KokkosKernels Tutorial and Exercises

Sandia National Laboratories

The hands-on exercises that follow demonstrate representative examples from the following categories of capabilities:

- BLAS and Team-based BLAS kernels
- Batched BLAS kernels
- Sparse linear-algebra kernels
- Sparse graph kernels
- Conjugate Gradient examples (Sparse matrix CG and Preconditioned CG)

Exercise location:
https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels

Preliminary step: run the **run_installlibs_cmake.sh** script

- This will install kokkos, kokkos-kernels, and run CMake to generate a Makefile

The KokkosKernels Wiki will be a useful API reference for the exercises:
https://github.com/kokkos/kokkos-kernels/wiki/APIReference

Presentation outline

Sandia National Laboratories

1. Introduction
2. Dense linear algebra: BLAS capabilities
3. Team level linear algebra: BatchedBLAS
4. Sparse linear algebra, preconditioners and graph kernels
5. Build system using Cmake

Capabilities I: BLAS

Sandia National Laboratories

BLAS

BLAS-1 functions are available as multi-vector variants.

– 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
– iamax(x)	i = min{i of MAX_i(x[i]) }
– 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]|)
– reciprocal(r,x)	r[i] = 1 / 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 * z[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])
– trmm(s,uplo,t,d,alpha,A,B)	B = alpha*op(A)*B or alpha*B*op(A)
– trsm(s,uplo,t,d,alpha,A,B)	X = op(A)\alpha*B or alpha*B/op(A)

Capabilities II: LAPACK and TeamBLAS

Sandia National Laboratories

BLAS/LAPACK

- GESV
- TRTRI

TEAM

- abs
- axpby
- dot
- mult
- norm2
- scal
- update

Capabilities I: BLAS

Sandia National Laboratories

BLAS

BLAS-1 functions are available as multi-vector variants.

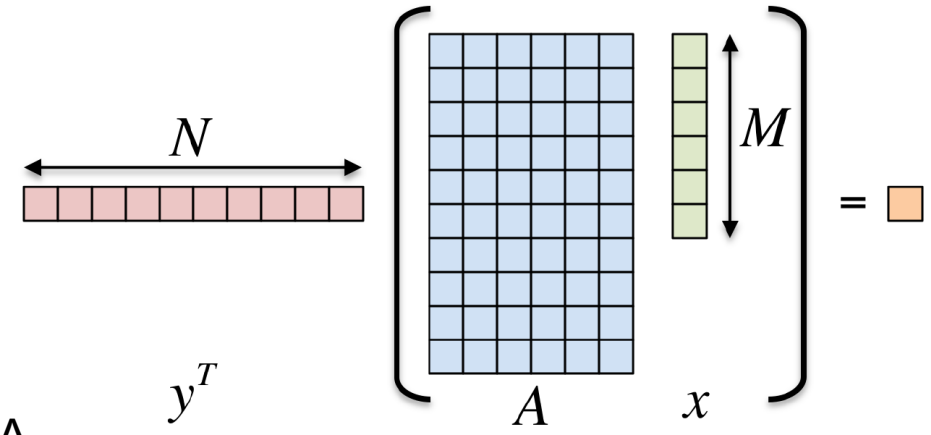
```
- abs(y,x)                    y[i]   = |x[i]|
- axpy(alpha,x,y)             y[i]  += alpha * x[i]
- axpby(alpha,x,beta,y)       y[i]   = beta * y + alpha * x[i]
- dot(x,y)                    dot    = SUM_i ( x[i] * y[i] )
- fill(x,alpha)               x[i]   = alpha
- iamax(x)                    i      = min{i of MAX_i(x[i]) }
- 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]| )
- reciprocal(r,x)             r[i]   = 1 / 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 * z[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])
- trmm(s,uplo,t,d,alpha,A,B)  B      = alpha*op(A)*B or alpha*B*op(A)
- trsm(s,uplo,t,d,alpha,A,B)  X      = op(A)\alpha*B or alpha*B/op(A)
```

KokkosKernels Interface: Inner Product Exercise

Sandia National Laboratories

BLAS and Team-based BLAS interface

- Goal: Re-implement the inner product exercise using
  1. KokkosKernels BLAS functions
  2. KokkosKernels team-based BLAS function
- Details: <**y**,**A*****x**>
  - **y** is *N*x1, **A** is *N*x*M,* **x** is *M*x1
  - Look for comments labeled with "EXERCISE"
  - Compile and run on OpenMP, CUDA backends
  - Compare performance of these two implementations
  - Try LayoutLeft and LayoutRight



KokkosKernels Interface: Inner Product Exercise

Sandia National Laboratories

- Exercise location:

  https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/InnerProduct/Begin

- Details:
  1. KokkosKernels BLAS functions
     - Convert from hierarchical parallel execution to using BLAS functions
     - **tmp** = **A*x** (tmp for **gemv** results)
     - result = <**y**,**tmp**>
  2. KokkosKernels team-based BLAS
     - Same as hierarchical parallel execution
     - Call team-based **dot** within each team to perform <A[teamId,:],x>

Hint: KokkosKernels Functions

```
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

Presentation outline

Sandia National Laboratories

1. Introduction
2. Dense linear algebra: BLAS capabilities
3. Team level linear algebra: BatchedBLAS
4. Sparse linear algebra, preconditioners and graph kernels
5. Build system using Cmake

Capabilities III: Batched BLAS/LAPACK

Sandia National Laboratories

Batched BLAS

```
parallel_for(TeamPolicy(N), KOKKOS_LAMBDA(member_type &member) {
  parallel_for(TeamThreadRange(member, nRows), [&](int &i) {
    parallel_for(ThreadVectorRange(member, nCols),[&](cint &j) {
      for (int k=0;k<len;++k)
        C(i,j) += A(i,k) * B(k,j);
}); }); });
```

SerialDot ← TeamVectorGemm

- Parallel batched operations are composed of using parallel-for with hierarchical interface of BLAS/LAPACK.
- Provides hierarchical interface matching to Kokkos hierarchical parallelism.
  - Serial – a single thread can invoke this layer of functions
  - Team – a single level of nested parallelism is used
  - TeamVector
    - Two nested parallel-for's are used TeamThread, ThreadVector, or
    - A single flattened nested parallel-for is used with TeamVectorRange.
- Function list
  - [S] Givens rotation, [S,TV] Householder transformation, [S] Hessenberg transformation, [S,T] LU, [S,T] Gemm, [S,T] Gemv, [S,T] Trsm, [S,T] Trsv, [S] Trmm, [S] Trtri, etc.
  - Implement new functions upon requests.

Capabilities III: Batched BLAS/LAPACK

Sandia National Laboratories

Batched BLAS

```
parallel_for(TeamPolicy(N), KOKKOS_LAMBDA(member_type &member) {
  parallel_for(TeamThreadRange(member, nRows), [&](int &i) {
    parallel_for(ThreadVectorRange(member, nCols),[&](cint &j) {
      for (int k=0;k<len;++k)
        C(i,j) += A(i,k) * B(k,j);
});  });  });
```

SerialDot ← TeamVectorGemm

- Parallel batched operations are composed of using parallel-for with hierarchical interface of BLAS/LAPACK.
- Provides hierarchical interface matching to Kokkos hierarchical parallelism.
  - Serial – a single thread can invoke this layer of functions
  - Team – a single level of nested parallelism is used
  - TeamVector
    - Two nested parallel-for's are used TeamThread, ThreadVector, or
    - A single flattened nested parallel-for is used with TeamVectorRange.
- Function list
  - [S] Givens rotation, [S,TV] Householder transformation, [S] Hessenberg transformation, [S,T] LU, [S,T] Gemm, [S,T] Gemv, [S,T] Trsm, [S,T] Trsv, [S] Trmm, [S] Trtri, etc.
  - Implement new functions upon requests.

Team-level Batched BLAS: Gemm

Sandia National Laboratories

- Objective
  - Learn how to use the Team-level interface for Gemm.
- Exercise location:

  https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/TeamGemm/Begin
- Details:
  1. Perform general matrix multiplies in parallel on a batch of matrices using the team-level Gemm function.
  2. Tune the team size on Cuda (via –A.m) and see how it impacts the performance.

KokkosBatched BLAS/LAPACK

Construction & Apply BlockJacobi

Sandia National Laboratories

- Objective
  - Compose a batched LU factorization on diagonal blocks and compute the inverse of the blocks.
  - Compare different parallelization strategies: a sequence of parallel_for calls vs a single parallel_for interface.

- Exercise location:

  https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/BlockJacobi/Begin

- Details:
  1. Simulate the standard batch functions running a sequence of parallel_for with LU, Lower Trsm, Upper Trsm to invert diagonal blocks.
  2. Compose a batched block matrix inversion using the so-called batched BLAS/LAPACK interface and a parallel_for with functor-level functions.
  3. On GPUs,
     1. Test the code with different team size with the run-different-teamsize.sh script.
     2. Run nvprof and see the difference (run-nvprof.sh script) and tune the team size for the problem size.

Presentation outline

Sandia National Laboratories

1. Introduction
2. Dense linear algebra: BLAS capabilities
3. Team level linear algebra: BatchedBLAS
4. Sparse linear algebra, preconditioners and graph kernels
5. Build system using Cmake

Capabilities IV: Sparse Kernels

Sandia National Laboratories

Sparse

- CSR-Sparse Matrix Class providing fundamental capabilities
- SPMV: Sparse Matrix Vector Multiply
- SPADD: Sparse Matrix Matrix Addition
- SPGEMM: Sparse Matrix Matrix Multiply; separate symbolic and numeric phase
- SPGEMM_Jacobi: Fused Sparse Matrix Matrix Multiply + Jacobi
- GS: Gauss-Seidel Method using graph coloring: symbolic, numeric, solve phases]
- SPILU(K): Incomplete LU Factorization
- SPTRSV: Sparse triangular solve; super-nodal variant

Graph

- Distance-1 and Distance-2 graph coloring
- Triangle enumeration for graph analytics
  - Using SpGEMM + Visitor Pattern: can be used to represent large problems

SPARSE: CrsMatrix

Sandia National Laboratories

- Example location:
- https://github.com/kokkos/kokkos-kernels/blob/develop/example/wiki/sparse/KokkosSparse_wiki_crsmatrix.cpp

- Container for compressed row sparse matrix (CrsMatrix)
  - API available in our wiki: https://github.com/kokkos/kokkos-kernels/wiki/CrsMatrix
  - Currently no deep_copy constructor or operator
  - Row pointers is const, you cannot re-order in place or increase storage space

- Interfaced with the following kernels:
  - SpMV
  - SpADD
  - SpGEMM

SPARSE: SPMV

Sandia National Laboratories

- Exercise location:
- https://github.com/kokkos/kokkos-kernels/blob/develop/example/wiki/sparse/KokkosSparse_wiki_spmv.cpp
- Usage:
  - `KokkosSparse::spmv(mode, alpha, A, x, beta, y);`
- Performs:
  - `y = beta*y + alpha*op(A)*x`
- Parameters:
  - `mode`: input "`N`" (non-transpose), "`T`" (transpose), "`C`" (conjugate-transpose)
  - `alpha, beta`: input scalars
  - `A`: input sparse matrix (KokkosSparse::CrsMatrix) of size n x m
  - `x`: input vector (rank-1 or rank-2 Kokkos::View) of length m
  - `y`: input/output vector (rank-1 or rank-2 Kokkos::View) of length n

SPARSE: SPADD

Sandia National Laboratories

- Exercise location:
- https://github.com/kokkos/kokkos-kernels/blob/develop/example/wiki/sparse/KokkosSparse_wiki_spadd.cpp
- Usage:
  - `KokkosSparse::spadd_symbolic(handle, A, B, C);`
  - `KokkosSparse::spadd_numeric(handle, alpha, A, beta, B, C);`
- Performs:
  - `C = beta*C + alpha*(A+B)`
- Parameters:
  - `alpha, beta`: input scalars
  - `A, B`: input sparse matrices (KokkosSparse::CrsMatrix) of size n x m
  - `C`: input/output sparse matrix (KokkosSparse::CrsMatrix) of size n x m

SPARSE: SpGEMM

Sandia National Laboratories

- Exercise location:
- https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/SpGEMM/Begin
- Usage:
  - `KokkosSparse::spgemm_symbolic(handle, A, isTransposed, B, isTransposed, C);`
  - `KokkosSparse::spgemm_numeric(handle, A, isTransposed, B, isTransposed, C);`
- Performs:
  - `C = op(A)*op(B)`
- Notes:
  - Create an SpGEMM handle: `handle.create_spgemm_handle(spgemm_algorithm);`
  - Create an empty matrix for output: `KokkosSparse::CrsMatrix<…> C;`
  - Use the same handle for both symbolic and numeric calls

SPARSE: Gauss-Seidel

Sandia National Laboratories

Given: **A**, a CrsMatrix that is numRows x numCols

x: unknown vector. **b**: right-hand side vector.

First, create `KokkosKernelsHandle` and call `create_gs_handle()`.

Structural (symbolic) setup.

- `KokkosSparse::Experimental::gauss_seidel_symbolic(&handle, numRows, numCols, A.graph.row_map, A.graph.entries, graphIsSymmetric);`

Numeric setup: anything depending on matrix values.

- `KokkosSparse::Experimental::gauss_seidel_numeric(&handle, numRows, numCols, A.graph.row_map, A.graph.entries, A.values, graphIsSymmetric);`

Forward apply (symmetric and backward have same interface):

- `KokkosSparse::Experimental::forward_sweep_gauss_seidel_apply(&handle, numRows, numCols, A.graph.row_map, A.graph.entries, A.values, x, b, initZeroX, updateCachedB, omega, numSweeps);`

SPARSE: Gauss-Seidel Exercise

Sandia National Laboratories

- Why both numRows and numCols?
  - For distributed Tpetra::CrsMatrix, numRows is number of locally owned rows, and numCols is locally owned + remote columns (column map). Matrix should still be globally square.
- `initZeroX`: whether to zero out x before apply. Only set to true on first apply on a given system.
- `updateCachedB`: if first apply with this b, or b has changed for any reason since last apply.

Exercise: https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/GaussSeidel/Begin

- First, fill in the code after `//EXERCISE` comments. Check that the solve converges in ~7 iterations.
- Then: try other algorithms when creating the GS sub-handle:

```
handle.create_gs_handle(KokkosSparse::GS_CLUSTER, 10);
```

or:

```
handle.create_gs_handle(KokkosSparse::GS_TWOSTAGE);
```

- In apply, try increasing number of sweeps from 1, or omega (damping factor) to something other than 1.0
- Try using `symmetric_gauss_seidel_apply` instead of `forward`

GRAPH: Coloring

Sandia National Laboratories

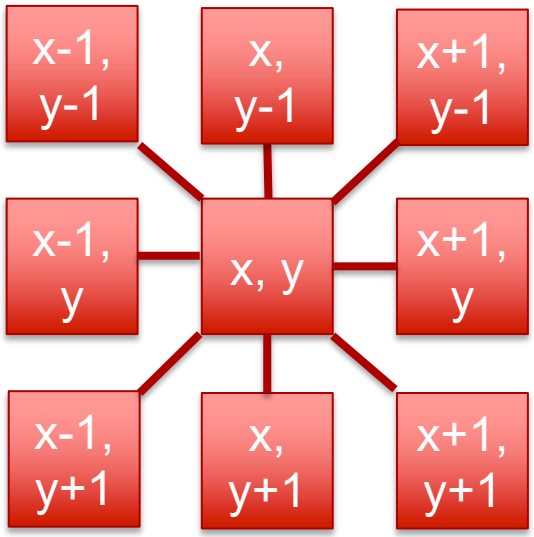
Given: **rowmap**, **entries** (forming a sparse CRS graph)

Compute distance-1 coloring: no pair of adjacent vertices can have same color. Example usage of bitwise vertex-based (VBBIT) algorithm:

```
// Set up
KernelHandle handle;
handle.create_graph_coloring_handle(KokkosGraph::COLORING_VB);
// Compute the coloring
KokkosGraph::Experimental::graph_color(&handle, numVertices,
numVertices, rowmap, entries);
// Get the colors array and number of colors used
auto colorHandle = handle.get_graph_coloring_handle();
auto colors = colorHandle->get_vertex_colors();
auto numColors = colorHandle->get_num_colors();
// Clean up
handle.destroy_graph_coloring_handle();
```

GRAPH: Coloring Exercise

- **Exercise**: https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/GraphColoring/Begin
- Creates graph from 2D rectangular grid and a 9-point stencil



- Distance-2 coloring: vertices that are within 1 or 2 edges of each other can't have same color.
- Exercise runs both D-1 and D-2 coloring, and prints out the colors in a grid.
- (If building with OpenMP) Try changing the number of threads and see how the number of colors changes:
- `export OMP_NUM_THREADS=<n>`

- **Some D1 algorithms:**
  `COLORING_VB`, `COLORING_SERIAL`, `COLORING_EB`
- **Some D2 algorithms:**
  `COLORING_D2_VB`, `COLORING_D2_SERIAL`, `COLORING_D2_NB_BIT`

SPARSE: SpILUK

Sandia National Laboratories

- Goal: get familiar with Kokkos Kernels interfaces of ILU(k) for CRS matrices
- Exercise location:

  https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/SpILUK/Begin

- Kokkos Kernels interfaces:

```
handle.create_spiluk_handle(SPILUKAlgorithm, nrows, nnzL, nnzU);
KokkosSparse::Experimental::spiluk_symbolic(handle, fill_level,
                                            A_rowmap, A_entries,
                                            L_rowmap, L_entries,
                                            U_rowmap, U_entries);
KokkosSparse::Experimental::spiluk_numeric(handle, fill_level,
                                           A_rowmap, A_entries, A_values,
                                           L_rowmap, L_entries, L_values,
                                           U_rowmap, U_entries, U_values);
```

SPARSE: SpILUK

Sandia National Laboratories

- Exercise Instructions:
  - ❑ Generate a simple Laplacian matrix on a cartesian grid as a KokkosSparse::CrsMatrix
  - ❑ Create an SpILUK handle
  - ❑ Call `spiluk_symbolic(…)` in the symbolic phase to construct the nonzero patterns of L and U, and to do level scheduling
  - ❑ Call `spiluk_numeric(…)` in the numeric phase to perform an ILU factorization based on the nonzero patterns found in the symbolic phase
  - ❑ As fill-level *k* changes, observe the changes of
    - ➢ ILU(k) fill-factor, and
    - ➢ row-sum difference (A*ones-L*U*ones)

KokkosKernels Interface: Conjugate Gradient Solver

Sandia National Laboratories

- 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

- Algorithm:

$$\mathbf{r_0} = \mathbf{b} - \mathbf{A} * \mathbf{x_0}$$

$$\mathbf{p_0} = \mathbf{r_0}$$

$$k = 0$$

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

Sandia National Laboratories

- Exercise location:

  https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/CGSolve/Begin

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

Hint: 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])`

ILU(k) Preconditioned Conjugate Gradient Solver with Kokkos Kernels

Sandia National Laboratories

- Goal: implement an ILU(k) preconditioned conjugate gradient solver for square, symmetric, positive-definite sparse matrix using Kokkos Kernels SpILUK and SpTRSV
- Exercise location:

  https://github.com/kokkos/kokkos-tutorials/tree/master/Intro-Full/Exercises/kokkoskernels/CGSolve_SpILUKprecond /Begin
- Kokkos Kernels interfaces:

```
handle.create_sptrsv_handle(SPTRSVAlgorithm, nrows, is_lower_tri);
KokkosSparse::Experimental::sptrsv_symbolic(handle,
                                            rowmap, entries);
KokkosSparse::Experimental::sptrsv_solve(handle,
                                         rowmap, entries, values,
                                         rhs, lhs);
```

ILU(k) Preconditioned Conjugate Gradient Solver with Kokkos Kernels

Sandia National Laboratories

- Exercise Instructions:
  - ❑ Generate a simple Laplacian matrix on a cartesian grid as a KokkosSparse::CrsMatrix
  - ❑ Create two SpTRSV handles (L and U)
  - ❑ Call `sptrsv_symbolic(…)` to do level scheduling
  - ❑ Call `sptrsv_solve(…)` to apply the preconditioner during the CGSolve

    ```
    1. tmp = L\r
    2. z = U\(tmp)
    ```

  - ❑ Observe the convergence behaviors:
    - ➢ without preconditioner
    - ➢ with preconditioner (as ILU(k) fill-level changes)

- Algorithm:

$\mathbf{r_0} = \mathbf{b} - \mathbf{A} * \mathbf{x_0}$

$\mathbf{z_0} = \mathbf{M}^{-1} * \mathbf{r_0}$

$\mathbf{p_0} = \mathbf{z_0}$

$k = 0$

while $\|\mathbf{r}_k\| > \varepsilon$ and $k < N$

$$\alpha = \frac{\mathbf{r}_k^T * z_k}{\mathbf{p}_k^T * \mathbf{A} * \mathbf{p}_k}$$

$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha * \mathbf{p}_k$

$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha * \mathbf{A} * \mathbf{p}_k$

$\mathbf{z}_{k+1} = \mathbf{M}^{-1} * \mathbf{r}_{k+1}$

$$\beta = \frac{\mathbf{r}_{k+1}^T * \mathbf{z}_{k+1}}{\mathbf{r}_k^T * \mathbf{z}_k}$$

$\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta * \mathbf{p}_k$

$k = k + 1$

Presentation outline

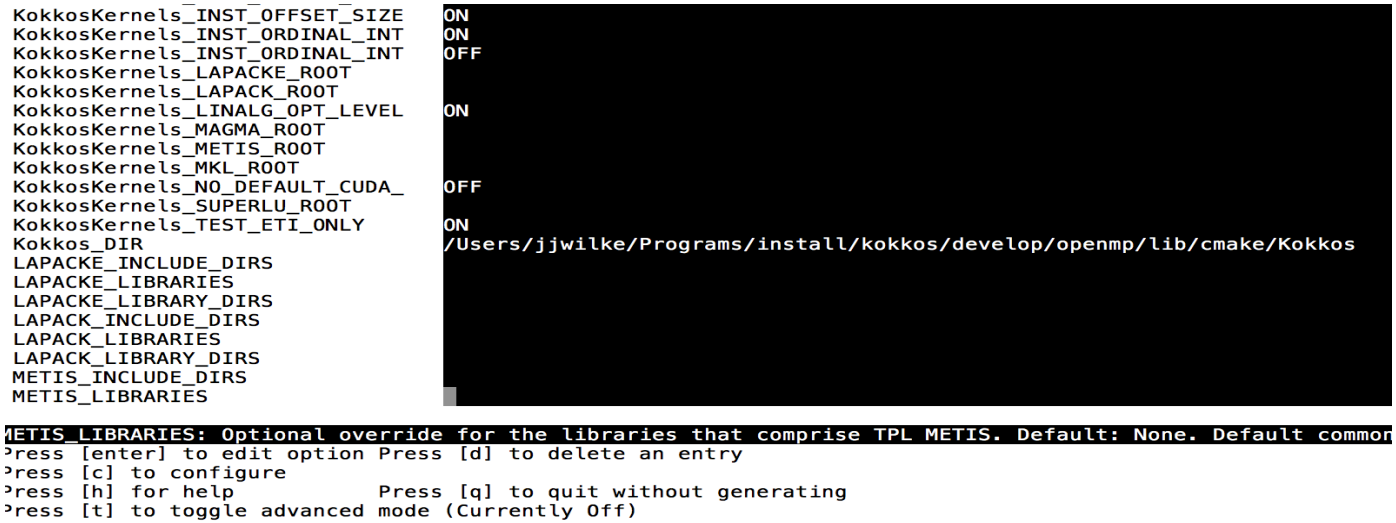
Sandia National Laboratories

1. Introduction
2. Dense linear algebra: BLAS capabilities
3. Team level linear algebra: BatchedBLAS
4. Sparse linear algebra, preconditioners and graph kernels
5. Build system using Cmake

Building Kokkos Kernels with CMake

Sandia National Laboratories

- cmake ${KERNELS_SOURCE} –D{OPTION}:BOOL=ON …
  Specify config variables on cmd line as –DOPT=Value
- To get a list of options in TUI, use ccmake:
  - ccmake ${KERNELS_SOURCE}
    –DCMAKE_CXX_COMPILER={}
    –DKokkos_ROOT=${PATH_TO_KOKKOS_INTALL}



  - Scroll over option and hit ‘h’ to see option details
  - Hit ‘enter’ to change option value

Most Important Build Options

Sandia National Laboratories

- -DKokkos_ROOT=${PATH_TO_KOKKOS}
  The underlying Kokkos. Many options (CUDA vs OpenMP, e.g.) automatically derived from Kokkos installation
- Template instantiation ahead of time to save compile-time
  - Scalars: e.g. –D KokkosKernels_INST_DOUBLE:BOOL=ON
  - Offsets: e.g. –D KokkosKernels_INST_OFFSET_SIZE_T:BOOL=ON
  - Mem/Exec Spaces: e.g.
    –D KokkosKernels_INST_EXECSPACE_OPENMP:BOOL=ON
- Third-party library (TPLs)
  - Enable: e.g. –D KokkosKernels_ENABLE_TPL_BLAS:BOOL=ON
    Kernels will look for BLAS in default locations
  - Customize paths:
    - e.g. –D BLAS_LIBRARIES="…"
    - e.g. –D BLAS_LIBRARY_DIRS="…"

A basic CMake project file creates a target and links to Kokkos

Sandia National Laboratories

Target linking in CMake automatically adds all include/link options.
Only need to link to KokkosKernels (Kokkos automatically comes with)

```
cmake_minimum_required(VERSION 3.12) #need >= 3.12
project(myProject CXX) #kokkos is always a C++ project

#must find installed kokkos kernels
find_package(KokkosKernels REQUIRED)

add_executable(myExe source.cpp) #add my program

#link to kokkos
target_link_libraries(myExe PRIVATE Kokkos::kokkoskernels)
```

Sandia National Laboratories

Exceptional service in the national interest

http://www.github.com/kokkos