Kokkos: The C++ Performance Portability Programming Model

Christian Trott (crtrott@sandia.gov), Carter Edwards


Center for Computing Research, Sandia National Laboratories, NM

SAND2017-4935 C
New Programming Models

- HPC is at a Crossroads
  - Diversifying Hardware Architectures
  - More parallelism necessitates paradigm shift from MPI-only

- Need for New Programming Models
  - Performance Portability: OpenMP 4.5, OpenACC, Kokkos, RAJA, SyCL, C++20, ...
  - Resilience and Load Balancing: Legion, HPX, UPC++, ...

- Vendor decoupling drives external development
New Programming Models

- HPC is at a Crossroads
  - Diversifying Hardware Architectures
  - More parallelism necessitates paradigm shift from MPI-only

- Need for New Programming Models
  - Performance Portability: OpenMP 4.5, OpenACC, Kokkos, RAJA, SyCL, C++20?, ...
  - Resilience and Load Balancing: Legion, HPX, UPC++, ...

- Vendor decoupling drives external development

What is Kokkos?
What is new?
Why should you trust us?
Kokkos: Performance, Portability and Productivity

https://github.com/kokkos
Performance Portability through Abstraction

Separating of Concerns for Future Systems...

Kokkos

Data Structures
- Memory Spaces ("Where")
  - Multiple-Levels
  - Logical Space (think UVM vs explicit)
- Memory Layouts ("How")
  - Architecture dependent index-maps
  - Also needed for subviews
- Memory Traits
  - Access Intent: Stream, Random, …
  - Access Behavior: Atomic
  - Enables special load paths: i.e. texture

Parallel Execution
- Execution Spaces ("Where")
  - N-Level
  - Support Heterogeneous Execution
- Execution Patterns ("What")
  - parallel_for/reduce/scan, task spawn
  - Enable nesting
- Execution Policies ("How")
  - Range, Team, Task-Dag
  - Dynamic / Static Scheduling
  - Support non-persistent scratch-pads

Parallel Execution
- Support nesting
- Range, Team, Task-Dag
- Dynamic / Static Scheduling
- Support non-persistent scratch-pads
## Capability Matrix

<table>
<thead>
<tr>
<th>Implementation Technique</th>
<th>Parallel Loops</th>
<th>Reduction Parallel Loops</th>
<th>Tightly Nested Loops</th>
<th>Non-tightly Nested Loops</th>
<th>Task Parallelism</th>
<th>Data Allocations</th>
<th>Data Transfers</th>
<th>Advanced Data Abstractions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kokkos</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>OpenMP</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>OpenACC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>CUDA</td>
<td>(X)</td>
<td>-</td>
<td>(X)</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>OpenCL</td>
<td>(X)</td>
<td>-</td>
<td>(X)</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>C++AMP</td>
<td>X</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>Raja</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>(X)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TBB</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C++17</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>(X)</td>
<td>X</td>
<td>(X)</td>
<td>(X)</td>
</tr>
<tr>
<td>Fortran2008</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>(X)</td>
<td>-</td>
</tr>
</tbody>
</table>
Example: Conjugent Gradient Solver

- Simple Iterative Linear Solver
- For example used in MiniFE
- Uses only three math operations:
  - Vector addition (AXPBY)
  - Dot product (DOT)
  - Sparse Matrix Vector multiply (SPMV)
- Data management with Kokkos Views:

```c
View<double*, HostSpace, MemoryTraits<Unmanaged>> h_x(x_in, nrows);
View<double*> x("x", nrows);
deep_copy(x, h_x);
```
CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound
- Serial Implementation:

```c
void axpby(int n, double* z, double alpha, const double* x, double beta, const double* y) {
    for(int i=0; i<n; i++)
        z[i] = alpha*x[i] + beta*y[i];
}
```

- Kokkos Implementation:

```c
void axpby(int n, View<double*> z, double alpha, View<const double*> x, double beta, View<const double*> y) {
    parallel_for("AXpBY", n, KOKKOS_LAMBDA ( const int& i) {
        z(i) = alpha*x(i) + beta*y(i);
    });
}
```
CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound
- Serial Implementation:

```c
void axpby(int n, double* z, double alpha, const double* x,
           double beta, const double* y) {
    for(int i=0; i<n; i++)
        z[i] = alpha*x[i] + beta*y[i];
}
```

- Kokkos Implementation:

```c
void axpby(int n, View<double>* z, double alpha, View<const double>* x,
           double beta, View<const double>* y) {
    parallel_for("AXpBY", n, KOKKOS_LAMBDA ( const int& i) {
        z(i) = alpha*x(i) + beta*y(i);
    });
}
```

Parallel Pattern: for loop
CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound
- Serial Implementation:

```c
void axpby(int n, double* z, double alpha, const double* x,
           double beta, const double* y) {
    for(int i=0; i<n; i++)
        z[i] = alpha*x[i] + beta*y[i];
}
```

- Kokkos Implementation:

```c
void axpby(int n, View<double*> z, double alpha, View<const double*> x,
           double beta, View<const double*> y) {
    parallel_for("AXpBY", n, KOKKOS_LAMBDA ( const int& i) {
        z(i) = alpha*x(i) + beta*y(i);
    });
}
```
CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound
- Serial Implementation:

  ```
  void axpby(int n, double* z, double alpha, const double* x, 
            double beta, const double* y) {
    for(int i=0; i<n; i++)
      z[i] = alpha*x[i] + beta*y[i];
  }
  ```

- Kokkos Implementation:

  ```
  void axpby(int n, View<double> z, double alpha, View<const double*> x, 
             double beta, View<const double*> y) {
    parallel_for("AXpBY", n, KOKKOS_LAMBDA (const int& i) {
      z(i) = alpha*x(i) + beta*y(i);
    });
  }
  ```

Execution Policy: do n iterations
CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound
- Serial Implementation:
  
  ```c
  void axpby(int n, double* z, double alpha, const double* x,
              double beta, const double* y) {
    for(int i=0; i<n; i++)
      z[i] = alpha*x[i] + beta*y[i];
  }
  ```

- Kokkos Implementation:
  
  ```c
  void axpby(int n, View<double*> z, double alpha, View<const double*> x,
              double beta, View<const double*> y) {
    parallel_for("AXpBY", n, KOKKOS_LAMBDA (const int& i) {
      z(i) = alpha*x(i) + beta*y(i);
    });
  }
  ```

  Iteration handle: integer index
CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound

Serial Implementation:

```c
void axpby(int n, double* z, double alpha, const double* x,
            double beta, const double* y) {
    for(int i=0; i<n; i++)
        z[i] = alpha*x[i] + beta*y[i];
}
```

Kokkos Implementation:

```c
void axpby(int n, View<double>* z, double alpha, View<const double*> x,
            double beta, View<const double*> y) {
    parallel_for("AXpBY", n, KOKKOS_LAMBDA (const int& i) {
        z(i) = alpha*x(i) + beta*y(i);
    });
}
```
CG Solve: The AXPBY

- Simple data parallel loop: Kokkos::parallel_for
- Easy to express in most programming models
- Bandwidth bound
- Serial Implementation:

  ```
  void axpby(int n, double* z, double alpha, const double* x, 
             double beta, const double* y) {
    for(int i=0; i<n; i++)
      z[i] = alpha*x[i] + beta*y[i];
  }
  ```

- Kokkos Implementation:

  ```
  void axpby(int n, View<double*> z, double alpha, View<const double*> x, 
             double beta, View<const double*> y) {
    parallel_for("AXpBY", n, KOKKOS_LAMBDA (const int& i) {
      z(i) = alpha*x(i) + beta*y(i);
    });
  }
  ```
CG Solve: The Dot Product

- Simple data parallel loop with reduction: Kokkos::parallel_reduce
- Non trivial in CUDA due to lack of built-in reduction support
- Bandwidth bound

Serial Implementation:

```c
double dot(int n, const double* x, const double* y) {
  double sum = 0.0;
  for(int i=0; i<n; i++)
    sum += x[i]*y[i];
  return sum;
}
```

Kokkos Implementation:

```c
double dot(int n, View<const double*> x, View<const double*> y) {
  double x_dot_y = 0.0;
  parallel_reduce("Dot",n, KOKKOS_LAMBDA (const int& i,double& sum) {
    sum += x[i]*y[i];
  }, x_dot_y);
  return x_dot_y;
}
```
CG Solve: The Dot Product

- Simple data parallel loop with reduction: Kokkos::parallel_reduce
- Non trivial in CUDA due to lack of built-in reduction support
- Bandwidth bound
- Serial Implementation:
  ```cpp
double dot(int n, const double* x, const double* y) {
  double sum = 0.0;
  for(int i=0; i<n; i++)
    sum += x[i]*y[i];
  return sum;
}
```
- Kokkos Implementation:
  ```cpp
double dot(int n, View<const double*> x, View<const double*> y) {
  double x_dot_y = 0.0;
  parallel_reduce("Dot",n, KOKKOS_LAMBDA (const int& i,double& sum) {
    sum += x[i]*y[i];
  }, x_dot_y);
  return x_dot_y;
}
```
CG Solve: The Dot Product

- Simple data parallel loop with reduction: Kokkos::parallel_reduce
- Non trivial in CUDA due to lack of built-in reduction support
- Bandwidth bound
- Serial Implementation:
  
  ```cpp
  double dot(int n, const double* x, const double* y) {
      double sum = 0.0;
      for(int i=0; i<n; i++)
          sum += x[i]*y[i];
      return sum;
  }
  ```

- Kokkos Implementation:
  
  ```cpp
  double dot(int n, View<const double*> x, View<const double*> y) {
      double x_dot_y = 0.0;
      parallel_reduce("Dot", n, KOKKOS_LAMBDA (const int& i, double& sum) {
          sum += x[i]*y[i];
      }, x_dot_y);
      return x_dot_y;
  }
  ```
CG Solve: The Dot Product

- Simple data parallel loop with reduction: Kokkos::parallel_reduce
- Non trivial in CUDA due to lack of built-in reduction support
- Bandwidth bound

Serial Implementation:

```cpp
def dot(int n, const double* x, const double* y) {
    double sum = 0.0;
    for(int i=0; i<n; i++)
        sum += x[i]*y[i];
    return sum;
}
```

Kokkos Implementation:

```cpp
def dot(int n, View<const double*> x, View<const double*> y) {
    double x_dot_y = 0.0;
    parallel_reduce("Dot", n, KOKKOS_LAMBDA (const int& i, double& sum) {
        sum += x[i]*y[i];
    }, x_dot_y);
    return x_dot_y;
}
```
CG Solve: The SPMV

- Loop over rows
- Dot product of matrix row with a vector
- Example of Non-Tightly nested loops
- Random access on the vector (Texture fetch on GPUs)

```c
void SPMV(int nrows, const int* A_rowOffsets, const int* A_cols,
           const double* A_vals, double* y, const double* x) {
    for(int row=0; row<nrows; ++row) {
        double sum = 0.0;
        int row_start=A_rowOffsets[row];
        int row_end=A_rowOffsets[row+1];
        for(int i=row_start; i<row_end; ++i) {
            sum += A_vals[i]*x[A_cols[i]];
        }
        y[row] = sum;
    }
}
```
CG Solve: The SPMV

- Loop over rows
- Dot product of matrix row with a vector
- Example of Non-Tightly nested loops
- Random access on the vector (Texture fetch on GPUs)

```c
void SPMV(int nrows, const int* A_row_offsets, const int* A_cols,
          const double* A_vals, double* y, const double* x) {
    for(int row=0; row<nrows; ++row) {
        double sum = 0.0;
        int row_start=A_row_offsets[row];
        int row_end=A_row_offsets[row+1];
        for(int i=row_start; i<row_end; ++i) {
            sum += A_vals[i]*x[A_cols[i]];
        }
        y[row] = sum;
    }
}
```
CG Solve: The SPMV

- Loop over rows
- Dot product of matrix row with a vector
- Example of Non-Tightly nested loops
- Random access on the vector (Texture fetch on GPUs)

```c
void SPMV(int nrows, const int* A_row_offsets, const int* A_cols,
          const double* A_vals, double* y, const double* x) {
    for(int row=0; row<nrows; ++row) {
        double sum = 0.0;
        int row_start=A_row_offsets[row];
        int row_end=A_row_offsets[row+1];
        for(int i=row_start; i<row_end; ++i) {
            sum += A_vals[i]*x[A_cols[i]];
        }
        y[row] = sum;
    }
}
```
CG Solve: The SPMV

- Loop over rows
- Dot product of matrix row with a vector
- Example of Non-Tightly nested loops
- Random access on the vector (Texture fetch on GPUs)

```c
void SPMV(int nrows, const int* A_row_offsets, const int* A_cols,
           const double* A_vals, double* y, const double* x) {
    for(int row=0; row<nrows; ++row) {
        double sum = 0.0;
        int row_start=A_row_offsets[row];
        int row_end=A_row_offsets[row+1];
        for(int i=row_start; i<row_end; ++i) {
            sum += A_vals[i]*x[A_cols[i]];
        }
        y[row] = sum;
    }
}
```
void SPMV(int nrows, View<const int*> A_rowOffsets,
        View<const int*> A_cols, View<const double*> A_vals,
        View<double*> y,
        View<const double*, MemoryTraits<RandomAccess>> x) {
    #ifdef KOKKOS_ENABLE_CUDA
        int rows_per_team = 64; int team_size = 64;
    #else
        int rows_per_team = 512; int team_size = 1;
    #endif

    parallel_for("SPMV:Hierarchy", TeamPolicy< Schedule< Static >>((nrows+rows_per_team-1)/rows_per_team,team_size,8),
            KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team) {

        const int first_row = team.league_rank()*rows_per_team;
        const int last_row = first_row+rows_per_team<nrows? first_row+rows_per_team : nrows;

        parallel_for(TeamThreadRange(team,first_row,last_row),[&] (const int row) {
            const int row_start=A_rowOffsets[row];
            const int row_length=A_rowOffsets[row+1]-row_start;

            double y_row;
            parallel_reduce(ThreadVectorRange(team,row_length),[=] (const int i,double& sum) {
                sum += A_vals(i+row_start)*x(A_cols(i+row_start));
            }, y_row);
            y(row) = y_row;
        });
    });
void SPMV(int nrows, View<const int*> A_row_offsets,
    View<const int*> A_cols, View<const double*> A_vals,
    View<double*> y,
    View<const double*, MemoryTraits< RandomAccess>> x) {
    #ifdef KOKKOS_ENABLE_CUDA
        int rows_per_team = 64; int team_size = 64;
    #else
        int rows_per_team = 512; int team_size = 1;
    #endif

    parallel_for("SPMV:Hierarchy", TeamPolicy< Schedule< Static > >
        ((nrows+rows_per_team-1)/rows_per_team,team_size,8),
        KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team) {

            const int first_row = team.league_rank() *rows_per_team;
            const int last_row = first_row +rows_per_team<nrows? first_row+rows_per_team : nrows;

            parallel_for(TeamThreadRange(team,first_row,last_row),[&] (const int row) {
                const int row_start = A_row_offsets[row];
                const int row_length = A_row_offsets[row+1] - row_start;

                double y_row;
                parallel_reduce(ThreadVectorRange(team,row_length), [=] (const int i,double& sum) {
                    sum += A_vals[i+row_start] * x(A_cols[i+row_start]);
                }, y_row);
                y(row) = y_row;

            });
        });
}
CG Solve: The SPMV

```c
void SPMV(int nrows, View<const int*> A_row_offsets,
          View<const int*> A_cols, View<const double*> A_vals,
          View<double*> y,
          View<const double*, MemoryTraits< RandomAccess>> x) {
  #ifdef KOKKOS_ENABLE_CUDA
    int rows_per_team = 64; int team_size = 64;
  #else
    int rows_per_team = 512; int team_size = 1;
  #endif

  parallel_for("SPMV:Hierarchy", TeamPolicy< Schedule< Static >> ((nrows+rows_per_team-1)/rows_per_team,team_size,8),
                    KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team) {

          const int first_row = team.league_rank() * rows_per_team;
          const int last_row = first_row + rows_per_team < nrows ? first_row + rows_per_team : nrows;

          parallel_for(TeamThreadRange(team,first_row,last_row),[&] (const int row) {
            const int row_start = A_rowOffsets[row];
            const int row_length = A_rowOffsets[row+1] - row_start;

            double y_row;
            parallel_reduce(ThreadVectorRange(team, row_length), [=] (const int i,double& sum) {
              sum += A_vals(i+row_start)*x(A_cols(i+row_start));
            }, y_row);
            y(row) = y_row;
          });
} );
```

Distribute rows in workset over team-threads
void SPMV(int nrows, View<const int*> A_row_offsets, View<const int*> A_cols, View<const double*> A_vals, View<double> y, View<const double*, MemoryTraits< RandomAccess>> x) {
    #ifdef KOKKOS_ENABLE_CUDA
        int rows_per_team = 64; int team_size = 64;
    #else
        int rows_per_team = 512; int team_size = 1;
    #endif

    parallel_for("SPMV:Hierarchy", TeamPolicy< Schedule< Static >>((nrows+rows_per_team-1)/rows_per_team,team_size,8),
        KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team) {

            const int first_row = team.league_rank()*rows_per_team;
            const int last_row = first_row+rows_per_team<nrows? first_row+rows_per_team : nrows;

            parallel_for(TeamThreadRange(team,first_row,last_row),[&] (const int row) {
                const int row_start=A_row_offsets[row];
                const int row_length=A_row_offsets[row+1]-row_start;

                double y_row;

                parallel_reduce(ThreadVectorRange(team,row_length),[=] (const int i,double& sum) {
                        sum += A_vals(i+row_start)*x(A_cols(i+row_start));
                    }, y_row);

                y(row) = y_row;
            });
        });
    }

Row x Vector dot product
void SPMV(int nrows, View<const int*> A_row_offsets, View<const int*> A_cols, View<const double*> A_vals, View<double*> y, View<const double*, MemoryTraits<RandomAccess>> x) {
    #ifdef KOKKOS_ENABLE_CUDA
        int rows_per_team = 64; int team_size = 64;
    #else
        int rows_per_team = 512; int team_size = 1;
    #endif
    parallel_for("SPMV:Hierarchy", TeamPolicy< Schedule< Static >> ((nrows+rows_per_team-1)/rows_per_team,team_size,8),
        KOKKOS_LAMBDA (const TeamPolicy<>::member_type& team) {
            const int first_row = team.league_rank() * rows_per_team;
            const int last_row = first_row + rows_per_team < nrows? first_row + rows_per_team : nrows;
            parallel_for(TeamThreadRange(team,first_row,last_row),[&] (const int row) {
                const int row_start = A_row_offsets[row];
                const int row_length = A_rowOffsets[row+1] - row_start;

                double y_row;
                parallel_reduce(ThreadVectorRange(team,row_length),[=] (const int i,double& sum) {
                    sum += A_vals(i+row_start)*x(A_cols(i+row_start));
                }, y_row);
                y(row) = y_row;
            });
        });
    }
}
CG Solve: Performance

- Comparison with other Programming Models
- Straight forward implementation of kernels
- OpenMP 4.5 is immature at this point
- Two problem sizes: 100x100x100 and 200x200x200 elements
Custom Reductions With Lambdas

- Added Capability to do Custom Reductions with Lambdas
- Provide built-in reducers for common operations
  - Add, Min, Max, Prod, MinLoc, MaxLoc, MinMaxLoc, And, Or, Xor,
- Users can implement their own reducers
- Example Max reduction:

```cpp
double result
parallel_reduce(N,
    KOKKOS_LAMBDA(const int& i,
        double& lmax) {  
    if(lmax < a(i)) lmax = a(i);
    },Max<double>(result));
```
New Features: MDRangePolicy

- Many people perform structured grid calculations
  - Sandia’s codes are predominantly unstructured though
- MDRangePolicy introduced for tightly nested loops
- Usecase corresponding to OpenMP collapse clause

```c
void launch (int N0, int N1, [ARGS]) {
    parallel_for(MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
                 KOKKOS_LAMBDA (int i0, int i1)
                 {/**<...*/});
}
```

- Optionally set iteration order and tiling:

```c
MDRangePolicy<Rank<3,Iterate::Right,Iterate::Left>>(
    {{0,0,0},{N0,N1,N2},{T0,T1,T2}})
```
New Features: Task Graphs

- Task dags are an important class of algorithmic structures
- Used for algorithms with complex data dependencies
  - For example triangular solve
- Kokkos tasking is on-node
- Future based, not explicit data centric (as for example Legion)
  - Tasks return futures
  - Tasks can depend on futures
- Respawn of tasks possible
- Tasks can spawn other tasks
- Tasks can have data parallel loops
  - I.e. a task can utilize multiple threads like the hyper threads on a core or a CUDA block

Carter Edwards S7253 “Task Data Parallelism”, Right after this talk.
New Features: Pascal Support

- Pascal GPUs Provide a set of new Capabilities
  - Much better memory subsystem
  - NVLink (next slide)
  - Hardware support for double precision atomic add
- Generally significant speedup 3-5x over K80 for Sandia Apps

![Graph showing Lammmps Tersoff Potential and Atomic Bandwidth performance](image)
New Features: HBM Support

- New architectures with HBM: Intel KNL, NVIDIA P100
- Generally three types of allocations:
  - Page pinned in DDR
  - Page pinned in HBM
  - Page migratable by OS or hardware caching
- Kokkos supports all three on both architectures
  - For Cuda backend: CudaHostPinnedSpace, CudaSpace and CudaUVMSpace
  - E.g.: Kokkos::View<double*, CudaUVMSpace> a(“A”, N);
- P100 Bandwidth with and without data Reuse
Upcoming Features

- Support for OpenMP 4.5+ Target Backend
  - Experimentally available on github
  - CUDA will stay preferred backend
  - Maybe support for FPGAs in the future?
  - Help maturing OpenMP 4.5+ compilers

- Support for AMD ROCm Backend
  - Experimentally available on github
  - Mainly developed by AMD
  - Support for APUs and discrete GPUs
  - Expect maturation fall 2017
Beyond Capabilities

- Using Kokkos is invasive, you can’t just swap it out
  - Significant part of data structures need to be taken over
  - Function markings everywhere

- It is not enough to have initial Capabilities

- Robustness comes from utilizations and experience
  - Different types of application and coding styles will expose different corner cases

- Applications need libraries
  - Interaction with TPLs such as BLAS must work
  - Many library capabilities must be reimplemented in the programming model

- Applications need tools
  - Profiling and Debugging capabilities are required for serious work
Timeline

2008

Initial Kokkos: Linear Algebra for Trilinos

2011

Restart of Kokkos: Scope now Programming Model

Mantevo MiniApps: Compare Kokkos to other Models

2012

LAMMPS: Demonstrate Legacy App Transition

2013

Trilinos: Move Tpetra over to use Kokkos Views

Multiple Apps start exploring (Albany, Uintah, …)

2014

Github Release of Kokkos 2.0

2015

Sandia Multiday Tutorial (~80 attendees)

Sandia Decision to prefer Kokkos over other models

2016

DOE Exascale Computing Project starts

Kokkos-Kernels and Kokkos-Tools Release

2017
Initial Demonstrations (2012-2015)

- Demonstrate Feasibility of Performance Portability
  - Development of a number of MiniApps from different science domains

- Demonstrate Low Performance Loss versus Native Models
  - MiniApps are implemented in various programming models

- DOE TriLab Collaboration
  - Show Kokkos works for other labs app
  - *Note this is historical data:* Improvements were found, RAJA implemented similar optimization etc.

---

**LULESH Figure of Merit Results (Problem 60)**

<table>
<thead>
<tr>
<th>Platform</th>
<th>Benchmarked Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW 1x16</td>
<td></td>
</tr>
<tr>
<td>HSW 1x32</td>
<td></td>
</tr>
<tr>
<td>P8 1x40 XL</td>
<td></td>
</tr>
<tr>
<td>KNC 1x224</td>
<td></td>
</tr>
<tr>
<td>ARM64 1x8</td>
<td></td>
</tr>
<tr>
<td>NV K40</td>
<td></td>
</tr>
</tbody>
</table>

Higher is Better
Training the User-Base

- Typical Legacy Application Developer
  - Science Background
  - Mostly Serial Coding (MPI apps usually have communication layer few people touch)
  - Little hardware background, little parallel programming experience
- Not sufficient to teach Programming Model Syntax
  - Need training in parallel programming techniques
  - Teach fundamental hardware knowledge (how does CPU, MIC and GPU differ, and what does it mean for my code)
  - Need training in performance profiling
- Regular Kokkos Tutorials
  - ~200 slides, 9 hands-on exercises to teach parallel programming techniques, performance considerations and Kokkos
  - Now dedicated ECP Kokkos support project: develop online support community
  - ~200 HPC developers (mostly from DOE labs) had Kokkos training so far
Keeping Applications Happy

- Never underestimate developers ability to find new corner cases!!
  - Having a Programming Model deployed in MiniApps or a single big app is very different from having half a dozen multi-million line code customers.
  - 538 Issues in 24 months
  - 28% are small enhancements
  - 18% bigger feature requests
  - 24% are bugs: often corner cases

- Example: Subviews
  - Initially data type needed to match including compile time dimensions
  - Allow compile/runtime conversion
  - Allow Layout conversion if possible
  - Automatically find best layout
  - Add subview patterns
Testing and Software Quality

New Features are developed on forks, and branches. Limited number of developers can push to develop branch. Pull requests are reviewed/tested.

**Compilers**
- GCC (4.8-6.3), Clang (3.6-4.0), Intel (15.0-18.0), IBM (13.1.5, 14.0), PGI (17.3), NVIDIA (7.0-8.0)

**Hardware**
- Intel Haswell, Intel KNL, ARM v8, IBM Power8, NVIDIA K80, NVIDIA P100

**Backends**
- OpenMP, Pthreads, Serial, CUDA

Each merge into master is minor release. Extensive integration test suite ensures backward compatibility, and catching of unit-test coverage gaps.
Building an EcoSystem

MiniApps

Applications

Trilinos
(Linear Solvers, Load Balancing, Discretization, Distributed Linear Algebra)

Kokkos – Kernels
(Sparse/Dense BLAS, Graph Kernels, Tensor Kernels)

Algorithms
(Random, Sort)

Containers
(Map, CrsGraph, Mem Pool)

Kokkos
(Parallel Execution, Data Allocation, Data Transfer)

std::thread
OpenMP
CUDA
ROCm

Kokkos – Tools
(Kokkos aware Profiling and Debugging Tools)

Kokkos – Support Community
(Application Support, Developer Training)
Kokkos Tools

https://github.com/kokkos/kokkos-tools

- Utilities
  - **KernelFilter**: Enable/Disable Profiling for a selection of Kernels
- Kernel Inspection
  - **KernelLogger**: Runtime information about entering/leaving Kernels and Regions
  - **KernelTimer**: Postprocessing information about Kernel and Region Times
- Memory Analysis
  - **MemoryHighWater**: Maximum Memory Footprint over whole run
  - **MemoryUsage**: Per Memory Space Utilization Timeline
  - **MemoryEvents**: Per Memory Space Allocation and Deallocations
- Third Party Connector
  - **VTune Connector**: Mark Kernels as Frames inside of Vtune
  - **VTune Focused Connector**: Mark Kernels as Frames + start/stop profiling
Kokkos-Tools: Example MemUsage

- Tools are loaded at runtime
  - Profile actual release builds of applications
  - Set via: `export KOKKOS_PROFILE_LIBRARY=[PATH_TO_PROFILING_LIB]`
- Output depends on tool
  - Often per process file
- MemoryUsage provides per MemorySpace utilization timelines
  - Time starts with Kokkos::initialize
  - `HOSTNAME-PROCESSID-CudaUVM.memspace_usage`

<table>
<thead>
<tr>
<th>#</th>
<th>Space CudaUVM</th>
<th>Time(s)</th>
<th>Size(MB)</th>
<th>HighWater(MB)</th>
<th>HighWater-Process(MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.317260</td>
<td>38.1   38.1   81.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.377285</td>
<td>0.0    38.1   158.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.384785</td>
<td>38.1   38.1   158.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.441988</td>
<td>0.0    38.1   158.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kokkos-Kernels

- Provide BLAS (1,2,3); Sparse; Graph and Tensor Kernels
- No required dependencies other than Kokkos
- Local kernels (no MPI)
- Hooks in TPLs such as MKL or cuBLAS/cuSparse where applicable
- Provide kernels for all levels of hierarchical parallelism:
  - Global Kernels: use all execution resources available
  - Team Level Kernels: use a subset of threads for execution
  - Thread Level Kernels: utilize vectorization inside the kernel
  - Serial Kernels: provide elemental functions (OpenMP declare SIMD)
- Work started based on customer priorities; expect multi-year effort for broad coverage
- People: Many developers from Trilinos contribute
  - Consolidate node level reusable kernels previously distributed over multiple packages
Kokkos-Kernels: Dense Blas Example

- Batched small matrices using an interleaved memory layout
- Matrix sizes based on common physics problems: 3, 5, 10, 15
- 32k small matrices
- Vendor libraries get better for more and larger matrices

**Batched DGEMM 32k blocks**

**Batched TRSM 32k blocks**
Kokkos Users Spread

- Users from a dozen major institutions
- More than two dozen applications/libraries
  - Including many multi-million-line projects
Further Material

- **https://github.com/kokkos** Kokkos Github Organization
  - **Kokkos**: Core library, Containers, Algorithms
  - **Kokkos-Kernels**: Sparse and Dense BLAS, Graph, Tensor (under development)
  - **Kokkos-Tools**: Profiling and Debugging
  - **Kokkos-MiniApps**: MiniApp repository and links
  - **Kokkos-Tutorials**: Extensive Tutorials with Hands-On Exercises

- **https://cs.sandia.gov** Publications (search for ’Kokkos’)
  - Many Presentations on Kokkos and its use in libraries and apps

- Talks at this GTC:
  - Carter Edwards S7253 “Task Data Parallelism”, Today 10:00, 211B
  - Pierre Kestener, S7166 “High Res. Fluid Dynamics”, Today 14:30, 212B
  - Michael Carilli, S7148 “Liquid Rocket Simulations”, Today 16:00, 212B
  - Panel, S7564 “Accelerator Programming Ecosystems”, Tuesday 16:00, Ball3
  - Training Lab, L7107 “Kokkos, Manycore PP”, Wednesday 16:00, LL21E