Kokkos: Performance, Portability and Productivity

H. Carter Edwards, Christian R Trott, Daniel Sunderland

*Sandia National Laboratories*

Slides prepared by Vikram Sharma Mailthody for CS598APK course at UIUC.

The paper goes over abstractions, APIs and some unit test results on different applications
Timeline

2008

 Initial Kokkos: Linear Algebra for Trilinos

2011

 Restart of Kokkos: Scope now Programming Model

2012

 Mantevo MiniApps: Compare Kokkos to other Models

2013

 LAMMPS: Demonstrate Legacy App Transition

2014

 Trilinos: Move Tpetra over to use Kokkos Views

 Paper coverage

 Multiple Apps start exploring (Albany, Uintah, …)

2015

 Github Release of Kokkos 2.0

2016

 Sandia Multiday Tutorial (~80 attendees)

 Sandia Decision to prefer Kokkos over other models

2017

 DOE Exascale Computing Project starts

 Kokkos-Kernels and Kokkos-Tools Release
**Motivation**

<table>
<thead>
<tr>
<th>System</th>
<th>CPU</th>
<th>Memory</th>
<th>GPU</th>
<th>Interconnect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicore</td>
<td>Power9</td>
<td>DDR4 (256GB)</td>
<td>Volta V100</td>
<td>X-bus</td>
</tr>
<tr>
<td>Manycore</td>
<td>Power9</td>
<td>DDR4 (256GB)</td>
<td>Volta V100</td>
<td>X-bus</td>
</tr>
<tr>
<td>GPU</td>
<td></td>
<td></td>
<td>Volta V100</td>
<td></td>
</tr>
<tr>
<td>CPU + GPU</td>
<td></td>
<td></td>
<td>Volta V100</td>
<td></td>
</tr>
</tbody>
</table>
Motivation

• Many core –threads ↑ Memory per thread ↓ and Heterogeneity ↑
• To meet HPC requirement and maintain scalability (load balance, resilience)
  • Need to exploit fine-grain parallelism offered by diverse architecture
  • Shift from MPI-Only model

• Major obstacle to performance portability is the diverse and conflicting set of constraints on memory access patterns across devices.
Motivation

• Contemporary programming models – Good for manycore parallelism

• But fail to address memory access patterns
Goals

• To build a C++ library that is efficient
  • Template based library

• Obtain same performance as a variant of code that is written to the specific device
  • Support best data layout strategies specific to architecture

• That provides a programming model with performance portability across diverse devices
  • By unifying abstractions for fine-grain parallelism and memory access patterns

• Targeted to scientific and engineering codes.
  • Supports from sparse linear algebra to broadly usable libraries (mini – apps specific)
Kokkos: Programming model abstractions

Data structures
- Memory space (where)
  - Multiple levels, logical space to support UVMs
- Memory layouts (How)
  - Arch dependent index maps, Views, subviews, user specific
- Memory Traits (what)
  - Access intent (stream, random)
  - Access behavior (atomic)
  - Specialized load primitives - textures

Parallel Execution
- Execution space (where)
- Execution pattern (How)
  - Parallel_for/reduce/scan, nesting, task spawn
- Execution Policy (what)
  - Range, team, etc
  - Dynamic/static scheduling
  - Non-persistent scratch-pads, etc

Slide taken (and modified) from Kokkos Tutorial deck 2018
Kokkos: PM abstractions - Fundamentals

1. Execute computation kernels are in fine-grain data parallel within an execution spaces.
2. Computational kernels operate on multidimensional arrays resides in memory spaces.

In other words:

- Execution space is just where the program executes. (Threads in execute space, data in memory space.)
- Execution space has accessibility and performance relationship with memory space. (CPU-GPU – UVM)
Polymorphic data layout

Let's say we have a base abstract class with implementation derived1, derived2, and derived3. Then, memory layout of std::vector<base*> shown above.

Adjacent elements are not allocated continuously. Issues with dynamic polymorphism.

Even though the derived types are unknown, they typically become available during compile time. Build a container with internal data structure pointing to as many vectors or segments as there are derived class.
Kokkos: Multidimensional arrays

1. Consists of
   1. a set of datums $\{x_i\}$ of the same value type
   2. an index $X_S$ defined by the Cartesian product of integer ranges
   3. A layout $X_L$ - a bijective map between the previous two.

2. A function contains
   1. A sequence of nested loops over dimensions of an array $X_S$
   2. Access array datums via the layout $X_L$

To change the memory access pattern – one either needs to change the memory layout or the dimension ordering of $X_S$

Kokkos array layouts are chosen at compile time!

Using “Views”
Kokkos: Arrays – Declaration, allocation and access

Views

- C++ class with a pointer to array data and metadata
- Semantics similar to std::shared_ptr
- Supports multi-dimensions - scalar, vector, matrix ...
- Size and dimension is fixed at the compile time
- Copy constructs and assignments are shallow
- Automatic deallocation when last view of the memory is destroyed or reassigned. (ref counting)
- Supports deep_copy. Layout, memory traits and memory space.

```cpp
// The View constructor allocates an array
// in Device memory space with dimensions
// N*M*8*3, where each '*' token denotes a
// dimension to be supplied at runtime.
// The label "A" is used in error messages
// which may occur in regard to this array.
View<dtype**, [8]*[3], Device> a("A", N, M);

// The parentheses operator implements the
// layout map.
a(i,j,k,l) = value;
```
Kokkos: Arrays – Declaration, allocation and access

Other Available memory spaces: HostSpace, CudaSpace, CudaUVMSpace, HostMirror, etc.

Other Available Behavioral traits: StreamingLoad, StreamingStore, RandomRead, RandomWrite

Memory Traits: Atomics, ReadOnce, ReadWrite, RandomAccess
Kokkos: Memory Access – quick look

• Every View has a Layout set at compile-time through a template parameter.

• LayoutRight (row major) and LayoutLeft (column major) are common.

• Views in HostSpace default to LayoutRight and Views in CudaSpace default to LayoutLeft. (caching vs coalesced)

• Layouts are extensible and flexible.

• Kokkos maps parallel work indices and multidimensional array layout for performance portable memory access patterns.

• There is nothing in OpenMP, OpenACC, or OpenCL to manage layouts. You'll need multiple versions of code or pay the performance penalty
Kokkos: Parallel Execution

```c
for( idx = 0; idx < Max; ++idx){
    sum[idx] = calSum(..data..);
}
```

Kokkos maps work to the cores:
- Each iteration of computation body is a unit of work
- An iteration index identifies particular unit of work
- Iteration range identifies the amount of work

Kokkos maps each iteration indices to cores and then run them in parallel.
Kokkos: Parallel Execution

Computation bodies or kernels are given as functors or function objects.

```
ParallFunctor functor;
Kokkos::parallel_for(numIteration, functor);
```

Work items are assigned to functors one-by-one

```
struct Functor{
    void operator()(const size_t idx) const { ... }
}
```
Kokkos: Parallel Execution

Computation bodies or kernels are given as functors or function objects.

```cpp
ParallFunctor functor;
Kokkos::parallel_for(numIteration, functor);
```

Work items are assigned to functors one-by-one

```cpp
struct Functor{
    void operator()(const size_t idx) const { ... }
}
```
Kokkos: Parallel Execution

Computation bodies or kernels are given as functors or function objects.

```
struct Functor{
    void operator()(const size_t idx) const {...}
}
```

Kokkos runtime does not guarantee concurrency and ordering

Parallel functor body must have access to all the data it needs through functor’s data members.
## Kokkos: What more?

<table>
<thead>
<tr>
<th>Implementation Technique</th>
<th>Parallel Loops</th>
<th>Reduction</th>
<th>Parallel Loops</th>
<th>Tightly Nested</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>
<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>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>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>X</td>
<td>X</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>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>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>
<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>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>
<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>
<td>-</td>
</tr>
</tbody>
</table>
Kokkos: Parallel Execution - AXPBY

```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];
}

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

Parallel pattern: for loop
Kokkos: Parallel Execution - AXPBY

```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];
}
```

```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_LAMDA (const int& i){
        z[i] = alpha*x[i] + beta*y[i];
    });
}
```
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];
}

void axpby(int n, View<double*> z, double alpha, View<const double*> x, double beta, View<const double*y){
    parallel_for("AXpBY", n, KOKKOS_LAMDA (const int& i){
        z[i] = alpha*x[i] + beta*y[i];
    });
}
Kokkos: Parallel Execution - AXPBY

```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];
}

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

Its handle: integer index
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];
}

void axpby(int n, View<double*> z, double alpha, View<const double*> x, double beta, View<const double*y>){
    parallel_for("AXpBY", n, KOKKOS_LAMDA (const int& i){
        z[i] = alpha*x[i] + beta*y[i];
    });
}
Kokkos: Parallel Execution – Dot product

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

def dot(int n, View<const double*> n, View<const double*> y){
    double x_dot_y = 0.0;
    parallel_reduce("Dot", n, KOKKOS_LAMDA( const int&i, double& sum){
        sum += x[i] * y[i];
    }, x_dot_y);
    return x_dot_y;
}
Kokkos: Parallel Execution – Dot product

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

double dot(int n, View<const double*> n, View<const double*> y){
    double x_dot_y = 0.0;
    parallel_reduce("Dot", n, KOKKOS_LAMDA(const int& i, double& sum){
        sum += x[i] * y[i];
    }, x_dot_y);
    return x_dot_y;
}
Kokkos: Parallel Execution – Dot product

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

double dot(int n, View<const double*> n, View<const double*> y){
    double x_dot_y = 0.0;
    parallel_reduce("Dot", n, KOKKOS_LAMDA(const int&i, double& sum){
        sum += x[i] * y[i];
    }, x_dot_y);
    return x_dot_y;
}
Kokkos: Parallel Execution – Inner product

Kokkos :: parallel_reduce (N,
       KOKKOS_LAMBDA ( const int row , double & valueToUpdate ) {
            double thisRowsSum = 0;
            for (int col = 0; col < M; ++ col ) {
                  thisRowsSum += A(row ,col) * x( col );
            }
            valueToUpdate += y( row ) * thisRowsSum ;
       }, result );

What if we don't have enough rows to saturate the GPU?

1. Atomics
2. Thread Teams

Poor performance
Doing each individual row with atomics is like doing scalar integration with atomics.
Kokkos: Parallel Execution – Inner product

High-level strategy:
1. Do one parallel launch of N teams of M threads.
2. Each thread performs one entry in the row.
3. The threads within teams perform a reduction.
4. The thread teams perform a reduction

The final hierarchical parallel kernel:

```cpp
parallel_reduce(  
    team_policy(N, Kokkos::AUTO),

    KOKKOS_LAMBDA (member_type & teamMember, double & )  
    int row = teamMember league_rank();

    double thisRowsSum = 0;
    parallel_reduce(TeamThreadRange(teamMember, M),  
        [=] (int col, double & innerUpdate) {
            innerUpdate += A(row, col) * x(col);
        }, thisRowsSum);

    if (teamMember team_rank() == 0) {
        update += y(row) * thisRowsSum;
    }
}, result);
```

How many threads in a team
Kokkos: Another example

Listing 1: Kokkos SPMV skeleton code

```c++
int team_size;
int vector_length;

// instantiate SPMV functor
SPMV_Functor<...> func ( ... );

// registration of the team_size and vector_length parameters
Kokkos::Profiling::autoTune(&team_size,&vector_length);

// These parameters are used by the Kokkos TeamPolicy to map to hardware
Kokkos::parallel_for("SPMV",
    Kokkos::TeamPolicy <Kokkos::Schedule<Kokkos::Dynamic>> >
    (league_size,team_size,vector_length,func);

// ...
struct SPMV_Functor {
// ...
    operator () ( const team_member& dev ) const{
        Kokkos::parallel_for(Kokkos::TeamThreadRange(dev,0,rows_per_team),[=](...){
            // ...
            // perform vector reduction
            Kokkos::parallel_reduce(Kokkos::ThreadVectorRange(dev,row_length),[=](...){
                // ...
                lsum += ...;
            },sum);

            // Add the results once per thread
            Kokkos::single(Kokkos::PerThread(dev),[&](){
                // ...
                m_y(iRow) = sum;
            });
        };
    }
// ...
```

How many lanes in a vector (SIMD abstraction)
Kokkos: Evaluation

miniMD

miniFE
Kokkos: Evaluation
Kokkos: Evaluation
7.5. Specialize kernels for specific architectures

In some cases, it is not possible to design a single algorithm which is nearly optimal on all architectures. For example, a device specific feature can be leveraged for a more performant implementation of an algorithm on that device. In other situations, optimal algorithms might be different due to the large difference in the number of concurrent threads (e.g., $O(10^5)$ for GPU, $O(10^2)$ for Xeon Phi, and $O(10^1)$ for CPU). In these situations, it can be necessary to introduce a device-specialized version of a computation.

This situation occurred in miniFE for the sparse matrix-vector multiplication shown in Figure 24.
Kokkos: Pros and cons

• Pros:
  • is a C++ library, not a new language or language extension.
  • It is widely used and backbone of Trilinos – Big package for multi-package apps
  • supports clear, concise, thread-scalable parallel patterns.
  • lets you write algorithms once and run on many architectures
  • minimizes the amount of architecture-specific implementation details users must know. (if you need perf – better understand the system)
  • solves the data layout problem by using multi-dimensional arrays with architecture-dependent layouts

• Cons:
  • Rewrite of complete application, need to worry about abstraction, arch, etc
  • kokkos kernels are used by scientist and they don’t deal with how to write kokkos kernels! (that’s the complexity!)
  • Not a compile time transformation
  • Its TEMPLATE – unreadable.
  • Surprise – some application require to device specific optimization using Kokkos to get performance… Ex: MiniFE.
More details/References

• https://www.sciencedirect.com/science/article/pii/S0743731514001257
• https://github.com/kokkos/kokkos/wiki/The-Kokkos-Programming-Guide (only few contents written)
• https://github.com/kokkos/kokkos
• https://github.com/kokkos/kokkos-tutorials
Kokkos: Questions and answer

2. Views: https://github.com/kokkos/kokkos/wiki/View
5. Sorry – Kokkos View is a potentially reference counted multi dimensional array with compile time layouts and memory space. (not runtime as I mentioned)