

Kokkos: Enabling Performance Portability

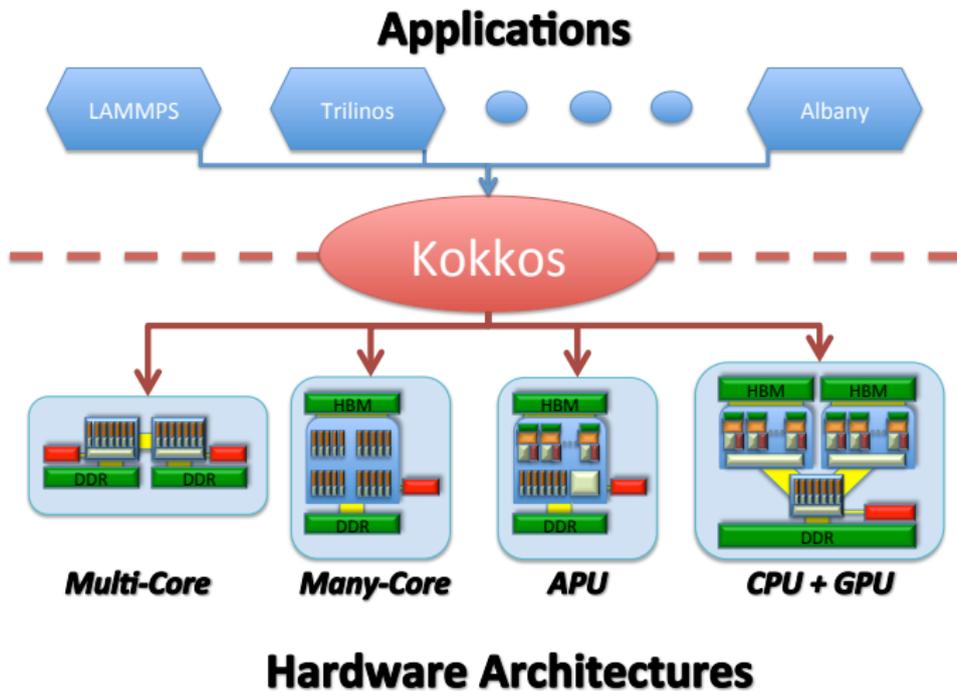
Christian R. Trott ¹, H. Carter Edwards ¹

¹Sandia National Laboratories

PACT15, San Francisco, Oct. 18th 2015

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.

SAND2015-9078 C



▶ Machine model

- ▶ N execution spaces \times M memory spaces
- ▶ $N \times M$ matrix for memory access performance/possibility
- ▶ Asynchronous execution allowed

▶ Implementation Approach

- ▶ A C++ template library
- ▶ Application focused: each feature is requested by application and used right now
- ▶ Performance focused: very high bar for acceptance if a feature impeders performance
- ▶ C++11 required
- ▶ Target different back-ends for different hardware architectures

▶ Distribution

- ▶ Open Source library
- ▶ Available on Github: github.com/kokkos/kokkos
- ▶ Extensive tutorial: github.com/kokkos/kokkos-tutorials

Execution Pattern: `parallel_for`, `parallel_reduce`, `parallel_scan`, `task`, ...

Execution Policy: how (and where) a user function is executed

- ▶ E.g., data parallel range : concurrently call `function(i)` for $i = [0..N)$
- ▶ User's function is a C++ functor or C++11 lambda

Execution Space: where functions execute

- ▶ Encapsulates hardware resources; e.g., cores, GPU, vector units, ...

Memory Space: where data resides

- ▶ AND what execution space can access that data
- ▶ Also differentiated by access performance; e.g., latency & bandwidth

Memory Layout: how data structures are ordered in memory

- ▶ provide mapping from logical to physical index space

Memory Traits: how data shall be accessed

- ▶ allow specialisation for different usage scenarios (read only, random, atomic, ...)

Pattern

Policy

```

for (size_t i = 0; i < N; ++i) {
  double y_i = 0;
  for (int j = 0; j < M; ++j) {
    y_i += A[i][j] * x[j];
  }
  y[i] = y_i;
}
    
```

Body

Terminology:

- ▶ **Pattern:** structure of the computations
for, reduction, scan, task-graph, ...
- ▶ **Execution Policy:** how computations are executed
static scheduling, dynamic scheduling, thread teams, ...
- ▶ **Computational Body:** code which performs each unit of
work; e.g., the loop body

⇒ The **pattern** and **policy** drive the computational **body**.

Example: $y = Ax$

OpenMP

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    double y_i = 0;
    for (int j = 0; j < M; ++j) {
        y_i += A[i][j] * x[j];
    }
    y[i] = y_i;
}
```

Kokkos

```
parallel_for(N, [=] (const size_t i) {
    double y_i = 0;
    for (int j = 0; j < M; ++j) {
        y_i += A[i][j] * x[j];
    }
    y[i] = y_i;
});
```

Example: $\langle y^T | Ax \rangle$ OpenMP
Kokkos

```

double yAx = 0;
#pragma omp parallel for reduction(+:yAx)
for (int i = 0; i < N; ++i) {
    double Ax_i = 0;
    for (int j = 0; j < M; ++j) {
        Ax_i += A[i][j] * x[j];
    }
    yAx += y[i] * Ax_i;
}

```

```

double yAx = 0;
parallel_reduce(N, [=] (const size_t i, double& yAx_thread) {
    double Ax_i = 0;
    for (int j = 0; j < M; ++j) {
        Ax_i += A[i][j] * x[j];
    }
    yAx_thread += y[i] * Ax_i;
}, yAx);

```

View overview:

- ▶ **Multi-dimensional array** of 0 or more dimensions
scalar (0), vector (1), matrix (2), etc.
- ▶ **Number of dimensions (rank)** is fixed at compile-time.
- ▶ Arrays are **rectangular**, not ragged.
- ▶ **Sizes of dimensions** set at compile-time or runtime.
e.g., 2x20, 50x50, etc.

View overview:

- ▶ **Multi-dimensional array** of 0 or more dimensions
scalar (0), vector (1), matrix (2), etc.
- ▶ **Number of dimensions (rank)** is fixed at compile-time.
- ▶ Arrays are **rectangular**, not ragged.
- ▶ **Sizes of dimensions** set at compile-time or runtime.
e.g., 2x20, 50x50, etc.

Example:

```
View<double***> data("label", N0, N1, N2); 3 run, 0 compile
View<double**[N2]> data("label", N0, N1); 2 run, 1 compile
View<double*[N1][N2]> data("label", N0); 1 run, 2 compile
View<double[N0][N1][N2]> data("label"); 0 run, 3 compile
```

Note: runtime-sized dimensions must come first.

View life cycle:

- ▶ Allocations only happen when *explicitly* specified.
i.e., there are **no hidden allocations**.
- ▶ Copy construction and assignment are **shallow** (like pointers).
so, you pass Views by value, *not* by reference
- ▶ Reference counting is used for **automatic deallocation**.

View life cycle:

- ▶ Allocations only happen when *explicitly* specified.
i.e., there are **no hidden allocations**.
- ▶ Copy construction and assignment are **shallow** (like pointers).
so, you pass Views by value, *not* by reference
- ▶ Reference counting is used for **automatic deallocation**.

Example:

```
void assignValueInView(View<double*> data) { data(0) = 3; }  
  
View<double*> a("a", NO), b("b", NO);  
a(0) = 1;  
b(0) = 2;  
a = b;  
View<double*> c(b);  
assignValueInView(c);  
print a(0)
```

What gets printed?

View life cycle:

- ▶ Allocations only happen when *explicitly* specified.
i.e., there are **no hidden allocations**.
- ▶ Copy construction and assignment are **shallow** (like pointers).
so, you pass Views by value, *not* by reference
- ▶ Reference counting is used for **automatic deallocation**.

Example:

```
void assignValueInView(View<double*> data) { data(0) = 3; }
```

```
View<double*> a("a", NO), b("b", NO);
```

```
a(0) = 1;
```

```
b(0) = 2;
```

```
a = b;
```

```
View<double*> c(b);
```

```
assignValueInView(c);
```

```
print a(0)
```

What gets printed?

3.0

Example: $\langle y^T | Ax \rangle$

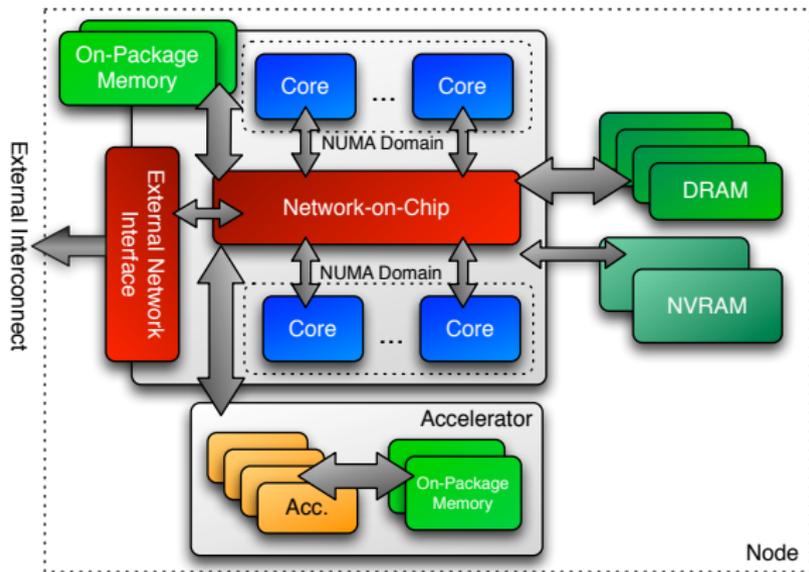
```

#include <Kokkos_Core.hpp>

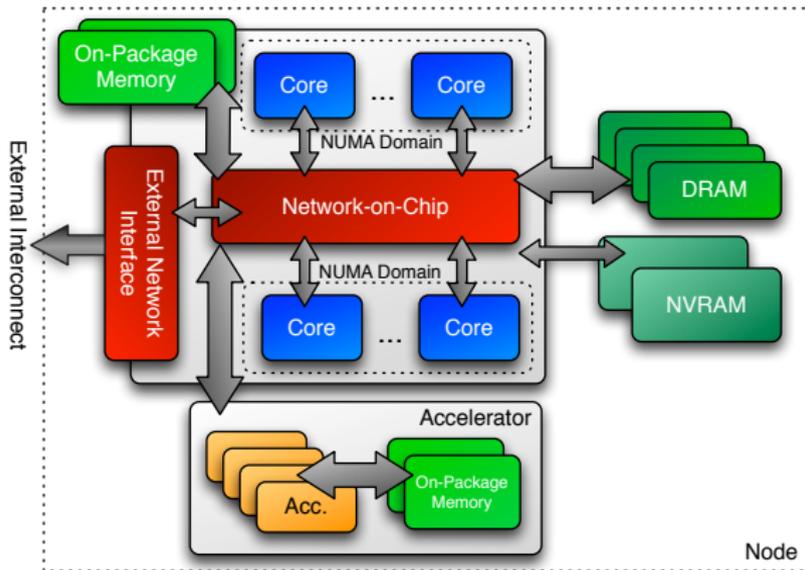
int main(int argc, char* argv[]) {
    // Initialize Kokkos analogous to MPI_Init()
    Kokkos::initialize(argc, argv);
    ...
    Kokkos::View<double**> A ("A", N,M); // Allocate matrix "A"
    Kokkos::View<double*> x("X",M), y("Y",N); // Allocate vector
    ...
    double yAx = 0;
    Kokkos::parallel_reduce(N, [=] (const size_t i,
                                   double& yAx_thread) {
        double Ax_i = 0;
        for (int j = 0; j < M; ++j) {
            Ax_i += A(i,j) * x(j);
        }
        yAx_thread += y(i) * Ax_i;
    }, yAx);
    ...
    Kokkos::finalize();
}

```

Compute nodes will be **heterogeneous** in cores *and* memory:

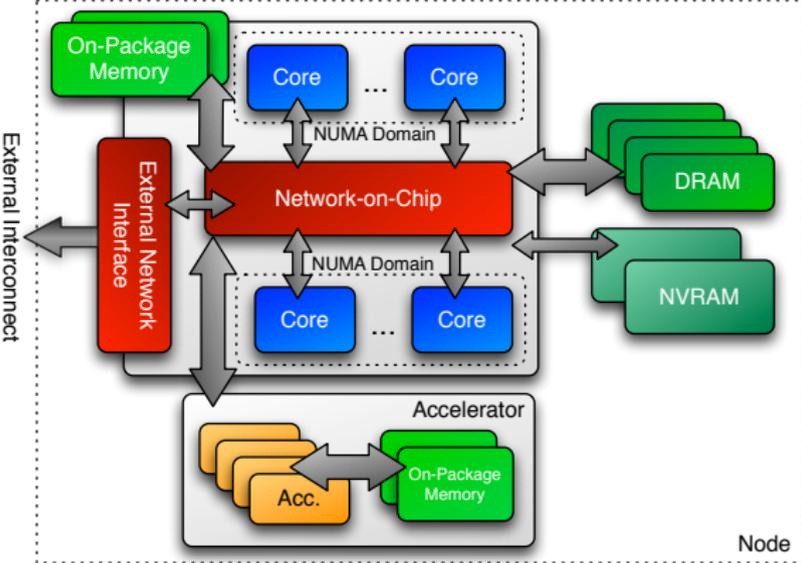


Compute nodes will be **heterogeneous** in cores *and* memory:



Many-core revolution: 20-year “just recompile” free ride is over.

Compute nodes will be **heterogeneous** in cores *and* memory:

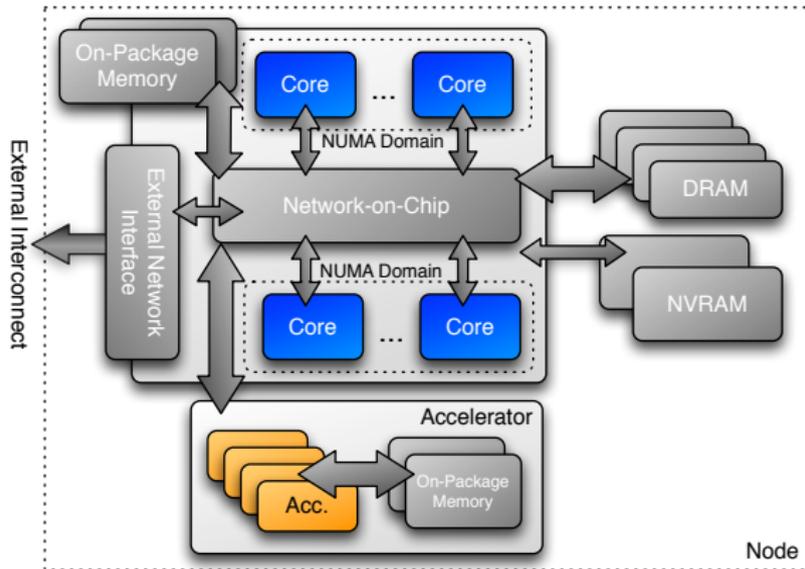


Many-core revolution: 20-year “just recompile” free ride is over.

How much do I have to **learn and change** to use these nodes?

Execution Space

a homogeneous set of cores and an execution mechanism
(i.e., “place to run code”)



Execution spaces: Serial, Threads, OpenMP, Cuda, ...

Important concept: Execution spaces

Every parallel operation is executed in an **execution space** set at compile time as part of an **execution policy**.

Important concept: Execution spaces

Every parallel operation is executed in an **execution space** set at compile time as part of an **execution policy**.

▶ `ExecutionPolicy<ExecutionSpace>(…)`

Important concept: Execution spaces

Every parallel operation is executed in an **execution space** set at compile time as part of an **execution policy**.

- ▶ `ExecutionPolicy<ExecutionSpace>(…)`
- ▶ Available **execution spaces**:
Serial, Pthread, OpenMP, Cuda, ... more

Important concept: Execution spaces

Every parallel operation is executed in an **execution space** set at compile time as part of an **execution policy**.

- ▶ `ExecutionPolicy<ExecutionSpace>(…)`
- ▶ Available **execution spaces**:
Serial, Pthread, OpenMP, Cuda, ... more
- ▶ If no `ExecutionSpace` is provided to an execution policy the **default execution space** is used.

Important concept: Execution spaces

Every parallel operation is executed in an **execution space** set at compile time as part of an **execution policy**.

- ▶ `ExecutionPolicy<ExecutionSpace>(…)`
- ▶ Available **execution spaces**:
Serial, Pthread, OpenMP, Cuda, ... more
- ▶ If no `ExecutionSpace` is provided to an execution policy the **default execution space** is used.
- ▶ Giving an integer `N` as policy is equivalent to `RangePolicy<>(N)`

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```
struct ParallelFunctor {
  KOKKOS_INLINE_FUNCTION
  double helperFunction(const size_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const size_t index) const {
    helperFunction(index);
  }
}
// Where kokkos defines:
#define KOKKOS_INLINE_FUNCTION inline /* #if CPU-only */
#define KOKKOS_INLINE_FUNCTION inline __device__ __host__ /* #if CPU+Cuda */
```

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```

struct ParallelFunctor {
  KOKKOS_INLINE_FUNCTION
  double helperFunction(const size_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const size_t index) const {
    helperFunction(index);
  }
}
// Where kokkos defines:
#define KOKKOS_INLINE_FUNCTION inline /* #if CPU-only */
#define KOKKOS_INLINE_FUNCTION inline __device__ __host__ /* #if CPU+Cuda */

```

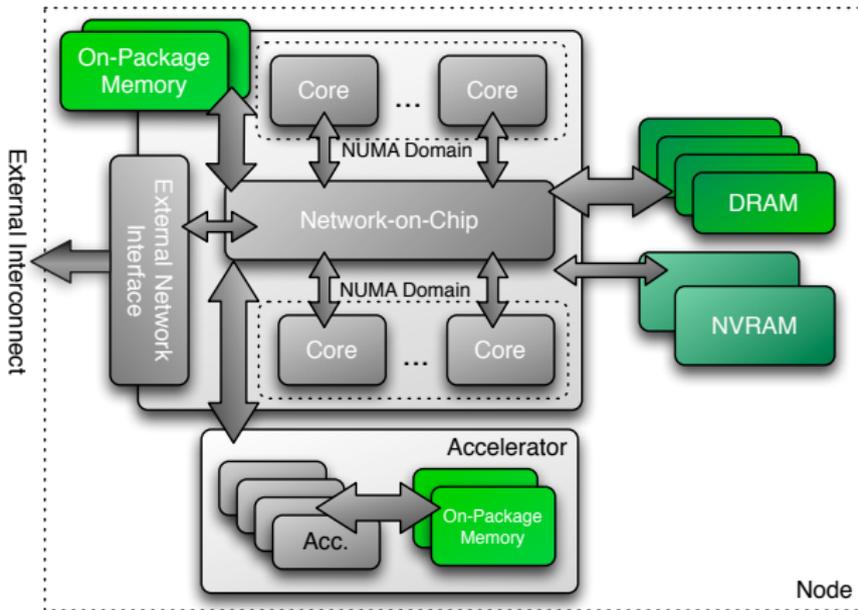
Lambda annotation with KOKKOS_LAMBDA macro (CUDA requires v 7.5)

```

Kokkos::parallel_for(numberOfIterations,
  KOKKOS_LAMBDA (const size_t index) {...});
// Where kokkos defines:
#define KOKKOS_LAMBDA [=] /* #if CPU-only */
#define KOKKOS_LAMBDA [=] __device__ /* #if CPU+Cuda */

```

Memory space:
explicitly-manageable memory resource
(i.e., “place to put data”)



Important concept: Memory spaces

Every view stores its data in a **memory space** set at compile time.

Important concept: Memory spaces

Every view stores its data in a **memory space** set at compile time.

▶ `View<double***, MemorySpace> data(...);`

Important concept: Memory spaces

Every view stores its data in a **memory space** set at compile time.

- ▶ `View<double***, MemorySpace> data(...);`
- ▶ Available **memory spaces**:
 `HostSpace, CudaSpace, CudaUVMSpace, ... more`

Important concept: Memory spaces

Every view stores its data in a **memory space** set at compile time.

- ▶ `View<double***, MemorySpace> data(...);`
- ▶ Available **memory spaces**:
 `HostSpace, CudaSpace, CudaUVMSpace, ... more`
- ▶ Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space

Important concept: Memory spaces

Every view stores its data in a **memory space** set at compile time.

- ▶ `View<double***, MemorySpace> data(...);`
- ▶ Available **memory spaces**:
 `HostSpace, CudaSpace, CudaUVMSpace, ...` more
- ▶ Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space
- ▶ If no `Space` is provided, the view's data resides in the **default memory space** of the **default execution space**.

Example: $\langle y^T | Ax \rangle$

```

...
// Allocate explicitly in CudaSpace
Kokkos::View<double**, Kokkos::CudaSpace> A ("A", N,M);
Kokkos::View<double*, Kokkos::CudaSpace> x("X",M), y("Y",N);
...
double yAx = 0;
// Run explicitly in the Cuda execution space
Kokkos::parallel_reduce(Kokkos::RangePolicy<Kokkos::Cuda>(N),
  KOKKOS_LAMBDA (const size_t i, double& yAx_thread) {
  double Ax_i = 0;
  for (int j = 0; j < M; ++j) {
    Ax_i += A(i,j) * x(j);
  }
  yAx_thread += y(i) * Ax_i;
}, yAx);
...

```

Important concept: Layouts

Every View has a Layout set at compile-time.

```
View<double***, Layout, Space> name(...);
```

Important concept: Layouts

Every View has a Layout set at compile-time.

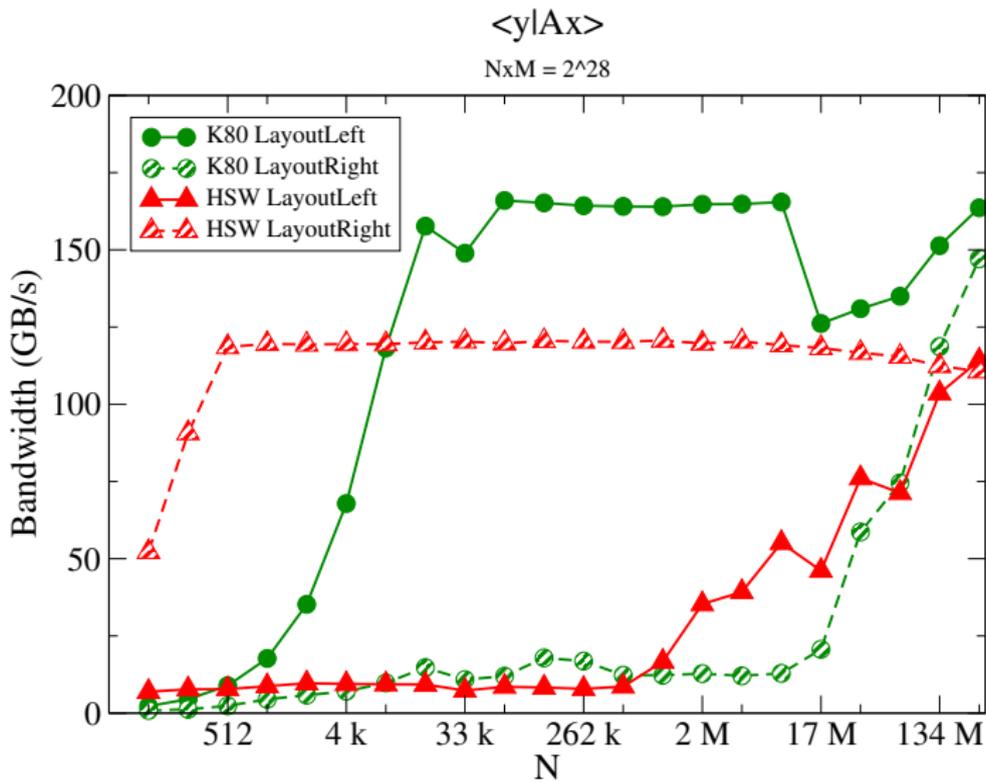
```
View<double***, Layout, Space> name(...);
```

- ▶ Most-common layouts are `LayoutLeft` and `LayoutRight`.
 - `LayoutLeft`: left-most index is stride 1.
 - `LayoutRight`: right-most index is stride 1.
- ▶ If no layout specified, default for that memory space is used.
 - `LayoutLeft` for `CudaSpace`, `LayoutRight` for `HostSpace`.
- ▶ Layouts are extensible: ~50 lines
- ▶ Advanced layouts: `LayoutStride`, `LayoutTiled`, ... extensible

Example: $\langle y^T | Ax \rangle$

```
...
// Allocate explicitly with LayoutRight
Kokkos::View<double**, Kokkos::LayoutRight> A ("A", N,M);
Kokkos::View<double*> x("X",M), y("Y",N);
...
double yAx = 0;
// Run explicitly in the Cuda execution space
Kokkos::parallel_reduce(N,
    KOKKOS_LAMBDA (const size_t i, double& yAx_thread) {
    double Ax_i = 0;
    for (int j = 0; j < M; ++j) {
        Ax_i += A(i,j) * x(j);
    }
    yAx_thread += y(i) * Ax_i;
}, yAx);
...

```



Important concept: Hierarchical Parallelism

Parallel execution patterns can be **nested** by using a **TeamPolicy**.

Important concept: Hierarchical Parallelism

Parallel execution patterns can be **nested** by using a **TeamPolicy**.

▶ Team:

```
parallel_xx(TeamPolicy<>(WorkSets,TeamSize[,VectorLength]), ... );
```

Important concept: Hierarchical Parallelism

Parallel execution patterns can be **nested** by using a **TeamPolicy**.

▶ Team:

```
parallel_xx(TeamPolicy<>(WorkSets,TeamSize[,VectorLength]), ... );
```

▶ Thread: `parallel_xx(TeamThreadRange(team_handle,Begin,End),...);`

Important concept: Hierarchical Parallelism

Parallel execution patterns can be **nested** by using a **TeamPolicy**.

▶ Team:

```
parallel_xx(TeamPolicy<>(WorkSets,TeamSize[,VectorLength]), ... );
```

▶ Thread: `parallel_xx(TeamThreadRange(team_handle,Begin,End),...);`

▶ Vector: `parallel_xx(ThreadVectorRange(team_handle,Begin,End),...);`

Important concept: Hierarchical Parallelism

Parallel execution patterns can be **nested** by using a **TeamPolicy**.

▶ Team:

```
parallel_xx(TeamPolicy<>(WorkSets,TeamSize[,VectorLength]), ... );
```

▶ Thread: `parallel_xx(TeamThreadRange(team_handle,Begin,End),...);`

▶ Vector: `parallel_xx(ThreadVectorRange(team_handle,Begin,End),...);`

▶ The **Vector Level** is optional, and the provided vector length has not on all platforms meaning.

Important concept: Hierarchical Parallelism

Parallel execution patterns can be **nested** by using a **TeamPolicy**.

▶ Team:

```
parallel_xx(TeamPolicy<>(WorkSets,TeamSize[,VectorLength]), ... );
```

▶ Thread: `parallel_xx(TeamThreadRange(team_handle,Begin,End),...);`

▶ Vector: `parallel_xx(ThreadVectorRange(team_handle,Begin,End),...);`

▶ The **Vector Level** is optional, and the provided vector length has not on all platforms meaning.

▶ The **body** of a **TeamPolicy** kernel is executed as a **parallel region** with respect to each team.

Important concept: Hierarchical Parallelism

Parallel execution patterns can be **nested** by using a **TeamPolicy**.

▶ Team:

```
parallel_xx(TeamPolicy<>(WorkSets,TeamSize[,VectorLength]), ... );
```

▶ Thread: `parallel_xx(TeamThreadRange(team_handle,Begin,End),...);`

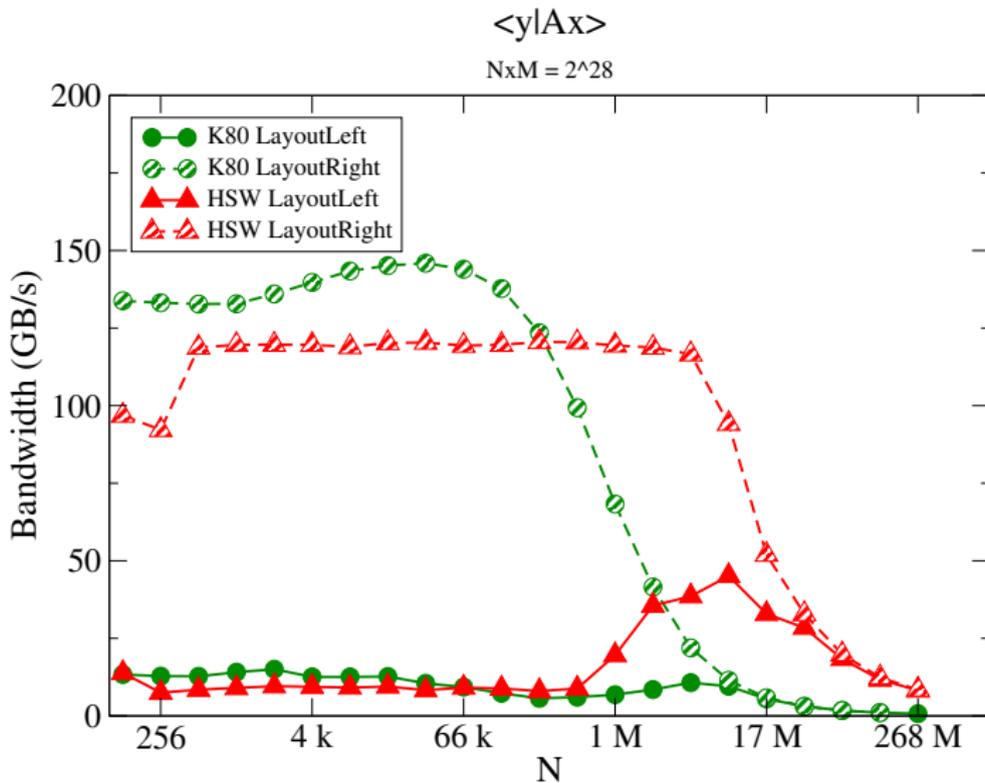
▶ Vector: `parallel_xx(ThreadVectorRange(team_handle,Begin,End),...);`

▶ The **Vector Level** is optional, and the provided vector length has not on all platforms meaning.

▶ The **body** of a **TeamPolicy** kernel is executed as a **parallel region** with respect to each team.

▶ Threads within a team are guaranteed to run concurrent, teams are not.

```
...
// Execution policies use a 'member type' as argument
typedef Kokkos::TeamPolicy<>::member_type team_type;
double yAx = 0;
// Split rows over teams, with Kokkos choosing team size
Kokkos::parallel_reduce(Kokkos::TeamPolicy<>(N, Kokkos::AUTO),
    KOKKOS_LAMBDA (const team_type& team, double& yAx_team) {
    double Ax_i = 0;
    // Do nested dot product with the team
    Kokkos::parallel_reduce(Kokkos::TeamThreadRange(team, M),
        [&] (const int& j) {
            Ax_i += A(i,j) * x(j);
        }, Ax_i);
    // Only one thread per team adds to the result
    Kokkos::single(Kokkos::PerTeam(team), [&] () {
        yAx_team += y(i) * Ax_i;
    });
}, yAx);
...
```



Features which were not discussed:

- ▶ Atomics: Support of arbitrary sized atomics
- ▶ Team Scratch Pads: Exposes Cuda shared memory functionality
- ▶ Algorithms: Sort and Random Numbers
- ▶ Containers: DualView, `std::vector` replacement, unordered map
- ▶ ExecutionTags: have classes act as functors with multiple tagged operators
- ▶ Custom Reductions/Scans: use functors with join, init and final functions
- ▶ Profiling support: simple inbuild capabilities + hooks for third party tools

Whats next (next couple of years and subject to finding people):

- ▶ Kernels package in Trilinos: BLAS, Sparse LA, Graph algorithms
- ▶ Task support: under development, prototype on CPUs
- ▶ Remote memory spaces: incorporate shmem like capabilities
- ▶ More debugging features: e.g. runtime identification of potential write conflicts