

# The Kokkos Lectures

Module 2: Views and Spaces

April 25, 2024

## Kokkos View

What are Views? How to create them? Why should you use it?

## Memory and Execution Spaces

How to control where data lives and code executes.

## Memory Access Patterns

The importance of access patterns for performance portability and how to control it.

# Views

## Learning objectives:

- ▶ Motivation behind the `View` abstraction.
- ▶ Key `View` concepts and template parameters.
- ▶ The `View` life cycle.

## Example: running daxpy on the GPU:

Lambda

```
double * x = new double[N]; // also y
parallel_for("DAXPY",N, [=] (const int64_t i) {
    y[i] = a * x[i] + y[i];
});
```

Functor

```
struct Functor {
    double *_x, *_y, a;
    void operator()(const int64_t i) const {
        _y[i] = _a * _x[i] + _y[i];
    }
};
```

## Example: running daxpy on the GPU:

Lambda

```
double * x = new double[N]; // also y
parallel_for("DAXPY",N, [=] (const int64_t i) {
    y[i] = a * x[i] + y[i];
});
```

Functor

```
struct Functor {
    double *_x, *_y, a;
    void operator()(const int64_t i) const {
        _y[i] = _a * _x[i] + _y[i];
    }
};
```

**Problem:** x and y reside in CPU memory.

## Example: running daxpy on the GPU:

Lambda

```
double * x = new double[N]; // also y
parallel_for("DAXPY",N, [=] (const int64_t i) {
    y[i] = a * x[i] + y[i];
});
```

Functor

```
struct Functor {
    double *_x, *_y, a;
    void operator()(const int64_t i) const {
        _y[i] = _a * _x[i] + _y[i];
    }
};
```

**Problem:** x and y reside in CPU memory.

**Solution:** We need a way of storing data (multidimensional arrays) which can be communicated to an accelerator (GPU).

⇒ **Views**

## View abstraction

- ▶ A *lightweight* C++ class with a pointer to array data and a little meta-data,
- ▶ that is *templated* on the data type (and other things).

## High-level example of Views for daxpy using lambda:

```
View<double*, ...> x(...), y(...);  
...populate x, y...  
  
parallel_for("DAXPY",N, [=] (const int64_t i) {  
    // Views x and y are captured by value (shallow copy)  
    y(i) = a * x(i) + y(i);  
});
```

## View abstraction

- ▶ A *lightweight* C++ class with a pointer to array data and a little meta-data,
- ▶ that is *templated* on the data type (and other things).

## High-level example of Views for daxpy using lambda:

```
View<double*, ...> x(...), y(...);
...populate x, y...

parallel_for("DAXPY",N, [=] (const int64_t i) {
    // Views x and y are captured by value (shallow copy)
    y(i) = a * x(i) + y(i);
});
```

## Important point

Views are **like pointers**, so copy them in your functors.



## 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.
- ▶ Access elements via "`(...)`" operator.

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.
- ▶ Access elements via "(...)" operator.

**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
//Access
data(i,j,k) = 5.3;
```

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**.
- ▶ They behave like `std::shared_ptr`

## 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**.
- ▶ They behave like `std::shared_ptr`

## Example:

```
View<double*[5]> a("a", N), b("b", K);  
a = b;  
View<double**> c(b);  
a(0,2) = 1;  
b(0,2) = 2;  
c(0,2) = 3;  
print_value( a(0,2) );
```

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**.
- ▶ They behave like `std::shared_ptr`

## Example:

```
View<double*[5]> a("a", N), b("b", K);  
a = b;  
View<double**> c(b);  
a(0,2) = 1;  
b(0,2) = 2;  
c(0,2) = 3;  
print_value( a(0,2) );
```

What gets printed?

3.0

## View Properties:

- ▶ Accessing a View's sizes is done via its `extent(dim)` function.
  - ▶ Static extents can *additionally* be accessed via `static_extent(dim)`.
- ▶ You can retrieve a raw pointer via its `data()` function.
- ▶ The label can be accessed via `label()`.

## Example:

```
View<double*[5]> a("A",N0);
assert(a.extent(0) == N0);
assert(a.extent(1) == 5);
static_assert(a.static_extent(1) == 5);
assert(a.data() != nullptr);
assert(a.label() == "A");
```

## Exercise #2: Inner Product, Flat Parallelism on the CPU, with Views

- ▶ Location: Exercises/02/Begin/
- ▶ Assignment: Change data storage from arrays to Views.
- ▶ Compile and run on CPU, and then on GPU with UVM

```
# CPU-only using OpenMP
cmake -B build-openmp -DKokkos_ENABLE_OPENMP=ON
cmake --build build-openmp
# GPU - note UVM in Makefile
cmake -B build-cuda -DKokkos_ENABLE_CUDA=ON
cmake --build build-cuda
# Run exercise
./build-openmp/02_Exercise -S 26
./build-cuda/02_Exercise -S 26
# Note the warnings, set appropriate environment variables
```

- ▶ Vary problem size: **-S #**
- ▶ Vary number of rows: **-N #**
- ▶ Vary repeats: **-nrepeat #**
- ▶ Compare performance of CPU vs GPU

- ▶ **Memory space** in which view's data resides; *covered next*.
- ▶ **deep\_copy** view's data; *covered later*.  
Note: Kokkos *never* hides a deep\_copy of data.
- ▶ **Layout** of multidimensional array; *covered later*.
- ▶ **Memory traits**; *covered later*.
- ▶ **Subview**: Generating a view that is a “slice” of other multidimensional array view; *covered later*.



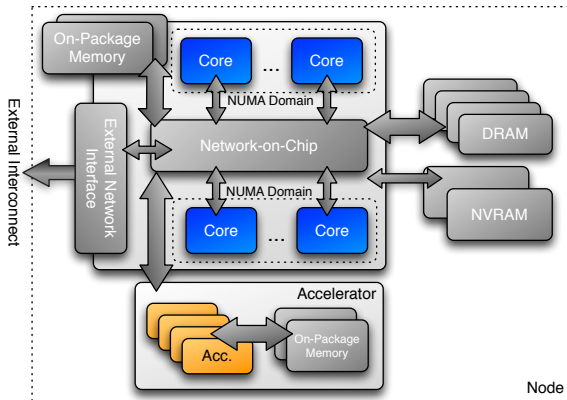
# Execution and Memory Spaces

## Learning objectives:

- ▶ Heterogeneous nodes and the **space** abstractions.
- ▶ How to control where parallel bodies are run, **execution space**.
- ▶ How to control where view data resides, **memory space**.
- ▶ How to avoid illegal memory accesses and manage data movement.
- ▶ The need for `Kokkos::initialize` and `finalize`.
- ▶ Where to use Kokkos annotation macros for portability.

## Execution Space

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



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

```
Host
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);

Parallel
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                    [=] (const int64_t somethingIndex) {
                        const double y = ...;
                        // do something interesting
                    }
                    );
```

```
Host
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);

Parallel
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                    [=] (const int64_t somethingIndex) {
                        const double y = ...;
                        // do something interesting
                    }
                    );
```

- ▶ Where will **Host** code be run? CPU? GPU?  
⇒ Always in the **host process**

```
Host
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);

Parallel
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                    [=] (const int64_t somethingIndex) {
                        const double y = ...;
                        // do something interesting
                    }
                    );
```

- ▶ Where will **Host** code be run? CPU? GPU?  
⇒ Always in the **host process**
- ▶ Where will **Parallel** code be run? CPU? GPU?  
⇒ The **default execution space**

```

Host
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);

Parallel
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                    [=] (const int64_t somethingIndex) {
                        const double y = ...;
                        // do something interesting
                    }
                    );

```

- ▶ Where will **Host** code be run? CPU? GPU?  
⇒ Always in the **host process**
- ▶ Where will **Parallel** code be run? CPU? GPU?  
⇒ The **default execution space**
- ▶ How do I **control** where the **Parallel** body is executed?  
Changing the default execution space (*at compilation*),  
or specifying an execution space in the **policy**.

## Changing the parallel execution space:

Custom

```
parallel_for("Label",  
    RangePolicy< ExecutionSpace >(0, numberOfIntervals),  
    [=] (const int64_t i) {  
        /* ... body ... */  
    });
```

Default

```
parallel_for("Label",  
    numberOfIntervals, // => RangePolicy<>(0, numberOfIntervals)  
    [=] (const int64_t i) {  
        /* ... body ... */  
    });
```

## Changing the parallel execution space:

Custom

```
parallel_for("Label",  
    RangePolicy< ExecutionSpace >(0, numberOfIntervals),  
    [=] (const int64_t i) {  
        /* ... body ... */  
    });
```

Default

```
parallel_for("Label",  
    numberOfIntervals, // => RangePolicy<>(0, numberOfIntervals)  
    [=] (const int64_t i) {  
        /* ... body ... */  
    });
```

Requirements for enabling execution spaces:

- ▶ Kokkos must be **compiled** with the execution spaces enabled.
- ▶ Execution spaces must be **initialized** (and **finalized**).
- ▶ **Functions** must be marked with a **macro** for non-CPU spaces.
- ▶ **Lambdas** must be marked with a **macro** for non-CPU spaces.



## Kokkos function and lambda portability annotation macros:

### Function annotation with KOKKOS\_INLINE\_FUNCTION macro

```
struct ParallelFunctor {
  KOKKOS_INLINE_FUNCTION
  double helperFunction(const int64_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const int64_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 int64_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const int64_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

```

Kokkos::parallel_for("Label", numberOfIterations,
  KOKKOS_LAMBDA (const int64_t index) {...});
// Where Kokkos defines:
#define KOKKOS_LAMBDA [=] /* #if CPU-only */
#define KOKKOS_LAMBDA [=] __device__ __host__ /* #if CPU+Cuda */

```

## Memory space motivating example: summing an array

```
View<double*> data("data", size);
for (int64_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy<SomeExampleExecutionSpace>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

## Memory space motivating example: summing an array

```
View<double*> data("data", size);
for (int64_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy<SomeExampleExecutionSpace>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

Question: Where is the data stored? GPU memory? CPU memory? Both?

## Memory space motivating example: summing an array

```
View<double*> data("data", size);
for (int64_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy<SomeExampleExecutionSpace>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

Question: Where is the data stored? GPU memory? CPU memory? Both?

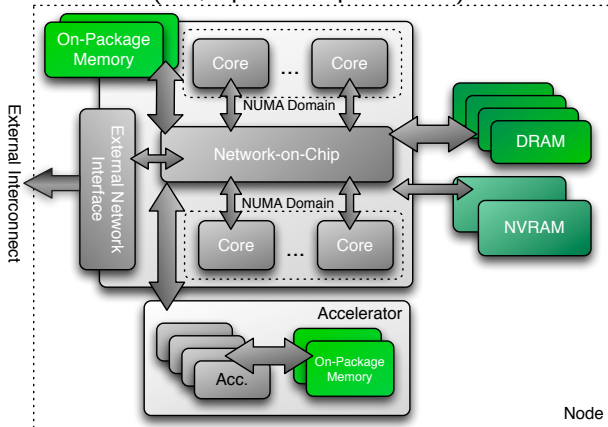
## Memory space motivating example: summing an array

```
View<double*> data("data", size);  
for (int64_t i = 0; i < size; ++i) {  
    data(i) = ...read from file...  
}  
  
double sum = 0;  
Kokkos::parallel_reduce("Label",  
    RangePolicy<SomeExampleExecutionSpace>(0, size),  
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {  
        valueToUpdate += data(index);  
    },  
    sum);
```

Question: Where is the data stored? GPU memory? CPU memory? Both?

⇒ **Memory Spaces**

**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
  - Portable: SharedSpace, SharedHostPinnedSpace

## 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
  - Portable: SharedSpace, SharedHostPinnedSpace
- ▶ 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
  - Portable: SharedSpace, SharedHostPinnedSpace
- ▶ 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**.

## Important concept: Memory spaces

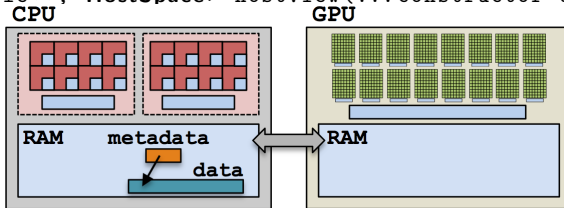
Every view stores its data in a **memory space** set at compile time.

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

```
// Equivalent:  
View<double*> a("A",N);  
View<double*,DefaultExecutionSpace::memory_space> b("B",N);
```

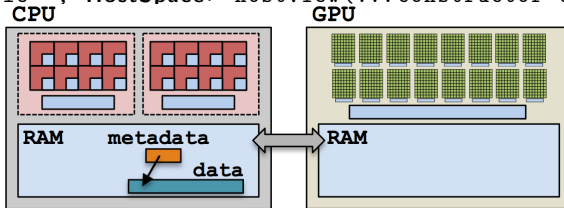
## Example: HostSpace

```
View<double**, HostSpace> hostView(...constructor arguments...);
```



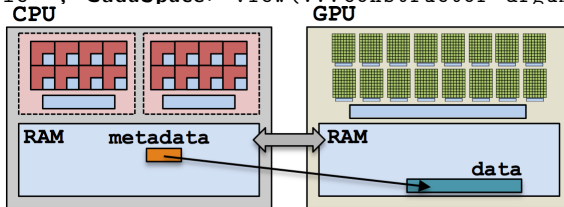
## Example: HostSpace

```
View<double**, HostSpace> hostView(...constructor arguments...);
```



## Example: CudaSpace

```
View<double**, CudaSpace> view(...constructor arguments...);
```



## Anatomy of a kernel launch:

1. User declares views, allocating.
2. User instantiates a functor with views.
3. User launches `parallel_something`:
  - ▶ Functor is copied to the device.
  - ▶ Kernel is run.
  - ▶ Copy of functor on the device is released.

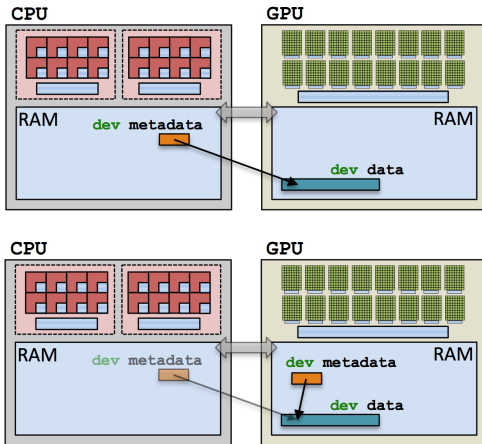
```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev(...);
parallel_for("Label",N,
  KL (int i) {
    dev(i) = ...;
  });
```

Note: **no deep copies** of array data are performed;  
*views are like pointers.*



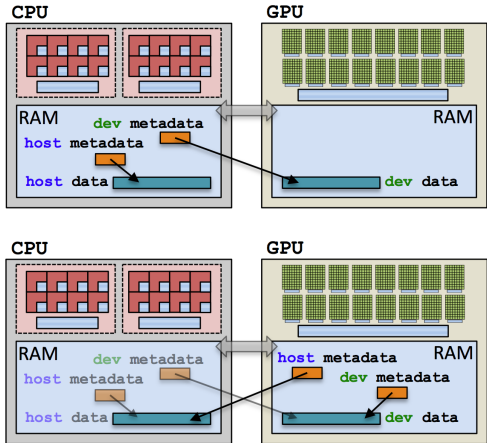
## Example: one view

```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
parallel_for("Label",N,
  KL (int i) {
    dev(i) = ...;
  });
```



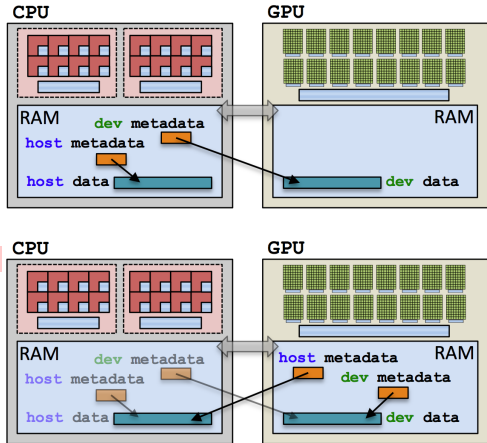
## Example: two views

```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for("Label",N,
  KL (int i) {
    dev(i) = ...;
    host(i) = ...;
  });
```



## Example: two views

```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for("Label", N,
  KL (int i) {
    dev(i) = ...;
    host(i) = ...;
  });
```



## Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

```
View<double*, CudaSpace> array("array", size);
for (int64_t i = 0; i < size; ++i) {
    array(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += array(index);
    },
    sum);
```

## Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

```
View<double*, CudaSpace> array("array", size);
for (int64_t i = 0; i < size; ++i) {
    array(i) = ...read from file...          fault
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += array(index);
    },
    sum);
```

## Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

```
View<double*, HostSpace> array("array", size);
for (int64_t i = 0; i < size; ++i) {
    array(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += array(index);
    },
    sum);
```

## Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

```
View<double*, HostSpace> array("array", size);
for (int64_t i = 0; i < size; ++i) {
    array(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += array(index);      illegal access
    },
    sum);
```

## Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

```
View<double*, HostSpace> array("array", size);
for (int64_t i = 0; i < size; ++i) {
    array(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += array(index);          illegal access
    },
    sum);
```

What's the solution?

- ▶ CudaUVMSpace
- ▶ CudaHostPinnedSpace (skipping)
- ▶ Mirroring

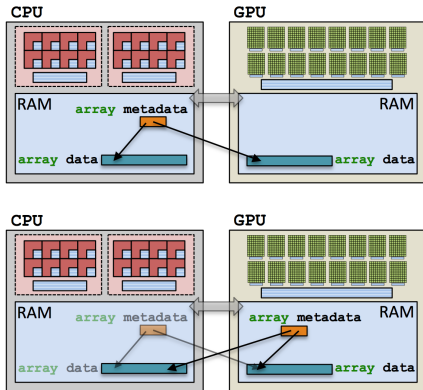


## CudaUVMSpace

```

#define KL KOKKOS_LAMBDA
View<double*,
    CudaUVMSpace> array;
array = ...from file...
double sum = 0;
parallel_reduce("Label", N,
    KL (int i, double & d) {
        d += array(i);
    },
    sum);

```



Cuda runtime automatically handles data movement,  
at a **performance hit**.

### Important concept: Mirrors

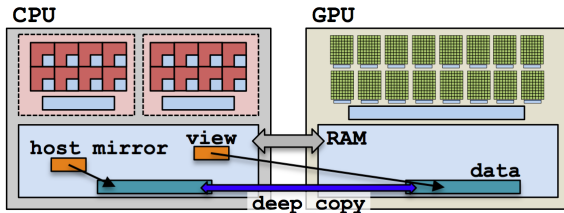
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

## Important concept: Mirrors

Mirrors are views of equivalent arrays residing in possibly different memory spaces.

### Mirroring schematic

```
using view_type = Kokkos::View<double**, Space>;
view_type view(...);
view_type::HostMirror hostView =
    Kokkos::create_mirror_view(view);
```



1. Create a `view`'s array in some memory space.

```
using view_type = Kokkos::View<double*, Space>;  
view_type view(...);
```

1. Create a `view`'s array in some memory space.

```
using view_type = Kokkos::View<double*, Space>;  
view_type view(...);
```

2. Create `hostView`, a *mirror* of the `view`'s array residing in the host memory space.

```
view_type::HostMirror hostView =  
    Kokkos::create_mirror_view(view);
```

1. Create a `view`'s array in some memory space.

```
using view_type = Kokkos::View<double*, Space>;  
view_type view(...);
```

2. Create `hostView`, a *mirror* of the `view`'s array residing in the host memory space.

```
view_type::HostMirror hostView =  
    Kokkos::create_mirror_view(view);
```

3. Populate `hostView` on the host (from file, etc.).

1. **Create** a **view**'s array in some memory space.  

```
using view_type = Kokkos::View<double*, Space>;  
view_type view(...);
```
2. **Create** **hostView**, a *mirror* of the **view**'s array residing in the host memory space.  

```
view_type::HostMirror hostView =  
    Kokkos::create_mirror_view(view);
```
3. **Populate** **hostView** on the host (from file, etc.).
4. **Deep copy** **hostView**'s array to **view**'s array.  

```
Kokkos::deep_copy(view, hostView);
```

1. **Create** a `view`'s array in some memory space.  
`using view_type = Kokkos::View<double*, Space>;`  
`view_type view(...);`
2. **Create** `hostView`, a *mirror* of the `view`'s array residing in the host memory space.

```
view_type::HostMirror hostView =  
    Kokkos::create_mirror_view(view);
```

3. **Populate** `hostView` on the host (from file, etc.).
4. **Deep copy** `hostView`'s array to `view`'s array.  
`Kokkos::deep_copy(view, hostView);`

5. **Launch** a kernel processing the `view`'s array.

```
Kokkos::parallel_for("Label",  
    RangePolicy< Space>(0, size),  
    KOKKOS_LAMBDA (...) { use and change view });
```



1. **Create** a **view's** array in some memory space.  

```
using view_type = Kokkos::View<double*, Space>;  
view_type view(...);
```
2. **Create** **hostView**, a *mirror* of the **view's** array residing in the host memory space.

```
view_type::HostMirror hostView =  
    Kokkos::create_mirror_view(view);
```

3. **Populate** **hostView** on the host (from file, etc.).
4. **Deep copy** **hostView's** array to **view's** array.  

```
Kokkos::deep_copy(view, hostView);
```

5. **Launch** a kernel processing the **view's** array.

```
Kokkos::parallel_for("Label",  
    RangePolicy< Space>(0, size),  
    KOKKOS_LAMBDA (...) { use and change view });
```

6. If needed, **deep copy** the **view's** updated array back to the **hostView's** array to write file, etc.

```
Kokkos::deep_copy(hostView, view);
```

What if the View is in HostSpace too? Does it make a copy?

```
using ViewType = Kokkos::View<double*, Space>;  
ViewType view("test", 10);  
ViewType::HostMirror hostView =  
    Kokkos::create_mirror_view(view);
```

- ▶ `create_mirror_view` allocates data only if the host process cannot access `view`'s data, otherwise `hostView` references the same data.
- ▶ `create_mirror` **always** allocates data.
- ▶ Reminder: Kokkos *never* performs a **hidden deep copy**.

## Exercise #3: Flat Parallelism on the GPU, Views and Host Mirrors

### Details:

- ▶ Location: Exercises/03/Begin/
- ▶ Add HostMirror Views and deep copy
- ▶ Make sure you use the correct view in initialization and Kernel

```
# Compile for CPU
cmake -B build-openmp -DKokkos_ENABLE_OPENMP=ON
cmake --build build-openmp
# Compile for GPU (we do not need UVM anymore)
cmake -B build-cuda -DKokkos_ENABLE_CUDA=ON
cmake --build build-cuda
# Run on GPU
./build-cuda/03_Exercise -S 26
```

### Things to try:

- ▶ Vary problem size and number of rows (-S ...; -N ...)
- ▶ Change number of repeats (-nrepeat ...)
- ▶ Compare behavior of CPU vs GPU

- ▶ Data is stored in Views that are “pointers” to **multi-dimensional arrays** residing in **memory spaces**.
- ▶ Views **abstract away** platform-dependent allocation, (automatic) deallocation, and access.
- ▶ **Heterogeneous nodes** have one or more memory spaces.
- ▶ **Mirroring** is used for performant access to views in host and device memory.
- ▶ Heterogeneous nodes have one or more **execution spaces**.
- ▶ You **control where** parallel code is run by a template parameter on the execution policy, or by compile-time selection of the default execution space.

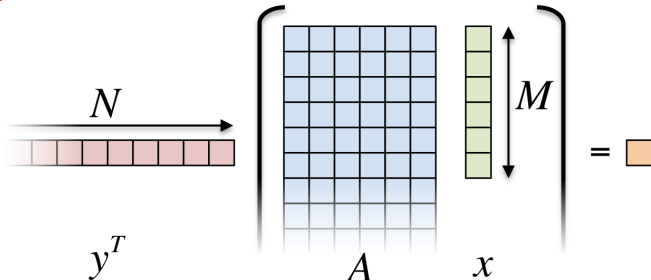
# Managing memory access patterns for performance portability

## Learning objectives:

- ▶ How the View's `Layout` parameter controls data layout.
- ▶ How memory access patterns result from Kokkos mapping parallel work indices **and** layout of multidimensional array data
- ▶ Why memory access patterns and layouts have such a performance impact (caching and coalescing).
- ▶ See a concrete example of the performance of various memory configurations.

## Example: inner product (0)

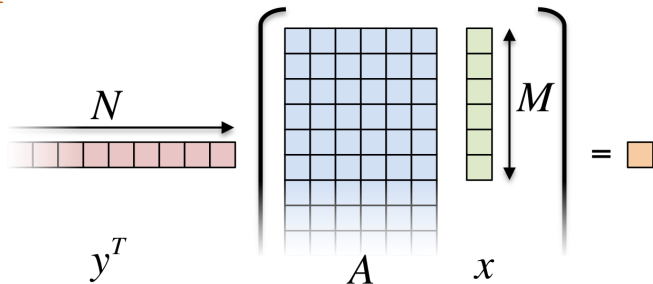
```
Kokkos::parallel_reduce("Label",  
  RangePolicy<ExecutionSpace>(0, N),  
  KOKKOS_LAMBDA (const size_t row, double & valueToUpdate) {  
    double thisRowsSum = 0;  
    for (size_t entry = 0; entry < M; ++entry) {  
      thisRowsSum += A(row, entry) * x(entry);  
    }  
    valueToUpdate += y(row) * thisRowsSum;  
  }, result+).
```



```

Kokkos::parallel_reduce("Label",
  RangePolicy<ExecutionSpace>(0, N),
  KOKKOS_LAMBDA (const size_t row, double & valueToUpdate) {
    double thisRowsSum = 0;
    for (size_t entry = 0; entry < M; ++entry) {
      thisRowsSum += A(row, entry) * x(entry);
    }
    valueToUpdate += y(row) * thisRowsSum;
  }, result);

```

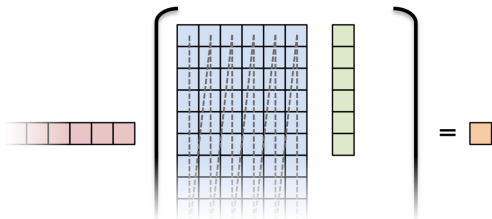


**Driving question:** How should  $A$  be laid out in memory?

Layout is the mapping of multi-index to memory:

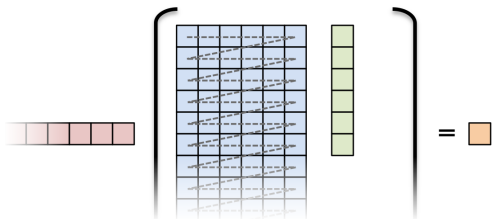
### LayoutLeft

in 2D, "column-major"



### LayoutRight

in 2D, "row-major"





## Important concept: Layout

Every View has a multidimensional array Layout set at compile-time.

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

## Important concept: Layout

Every View has a multidimensional array 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:  $\approx 50$  lines
- ▶ Advanced layouts: `LayoutStride`, `LayoutTiled`, ...

### Details:

- ▶ Location: Exercises/04/Begin/
- ▶ Replace ‘‘N’’ in parallel dispatch with `RangePolicy<ExecSpace>`
- ▶ Add `MemSpace` to all Views and Layout to A
- ▶ Experiment with the combinations of `ExecSpace`, Layout to view performance

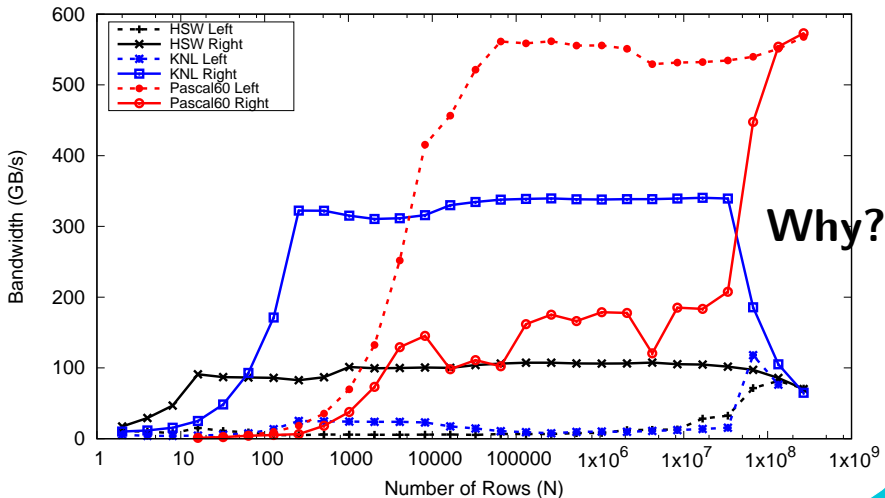
### Things to try:

- ▶ Vary problem size and number of rows (-S ...; -N ...)
- ▶ Change number of repeats (-nrepeat ...)
- ▶ Compare behavior of CPU vs GPU
- ▶ Compare using UVM vs not using UVM on GPUs
- ▶ Check what happens if `MemSpace` and `ExecSpace` do not match.

## Exercise #4: Inner Product, Flat Parallelism

### $\langle y | Ax \rangle$ Exercise 04 (Layout) Fixed Size

KNL: Xeon Phi 68c HSW: Dual Xeon Haswell 2x16c Pascal60: Nvidia GPU



## Thread independence:

```
operator()(int index, double & valueToUpdate) const {  
    const double d = _data(index);  
    valueToUpdate += d;  
}
```

Question: once a thread reads d, does it need to wait?

## Thread independence:

```
operator()(int index, double & valueToUpdate) const {  
    const double d = _data(index);  
    valueToUpdate += d;  
}
```

Question: once a thread reads `d`, does it need to wait?

- ▶ **CPU** threads are independent.
  - ▶ i.e., threads may execute at any rate.

## Thread independence:

```
operator()(int index, double & valueToUpdate) const {  
    const double d = _data(index);  
    valueToUpdate += d;  
}
```

Question: once a thread reads `d`, does it need to wait?

- ▶ **CPU** threads are independent.
  - ▶ i.e., threads may execute at any rate.
- ▶ **GPU** threads execute synchronized.
  - ▶ i.e., threads in groups can/must execute instructions together.

## Thread independence:

```
operator()(int index, double & valueToUpdate) const {  
    const double d = _data(index);  
    valueToUpdate += d;  
}
```

Question: once a thread reads *d*, does it need to wait?

- ▶ **CPU** threads are independent.
  - ▶ i.e., threads may execute at any rate.
- ▶ **GPU** threads execute synchronized.
  - ▶ i.e., threads in groups can/must execute instructions together.

In particular, all threads in a group (*warp* or *wavefront*) must finish their loads before *any* thread can move on.



## Thread independence:

```
operator()(int index, double & valueToUpdate) const {  
    const double d = _data(index);  
    valueToUpdate += d;  
}
```

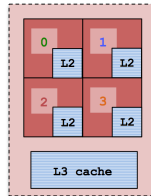
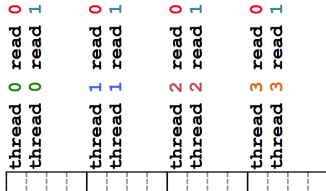
Question: once a thread reads *d*, does it need to wait?

- ▶ **CPU** threads are independent.
  - ▶ i.e., threads may execute at any rate.
- ▶ **GPU** threads execute synchronized.
  - ▶ i.e., threads in groups can/must execute instructions together.

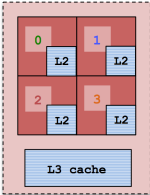
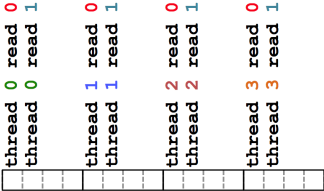
In particular, all threads in a group (*warp* or *wavefront*) must finish their loads before *any* thread can move on.

So, **how many cache lines** must be fetched before threads can move on?

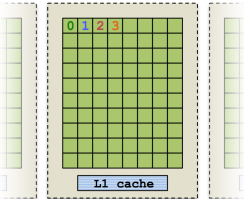
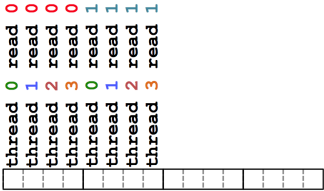
CPU: few (independent) cores with separate caches:



**CPU:** few (independent) cores with separate caches:



**GPU:** many (synchronized) cores with a shared cache:



### Important point

For performance, accesses to views in HostSpace must be **cached**, while access to views in CudaSpace must be **coalesced**.

**Caching:** if thread  $t$ 's current access is at position  $i$ , thread  $t$ 's next access should be at position  $i+1$ .

**Coalescing:** if thread  $t$ 's current access is at position  $i$ , thread  $t+1$ 's current access should be at position  $i+1$ .

### Important point

For performance, accesses to views in HostSpace must be **cached**, while access to views in CudaSpace must be **coalesced**.

**Caching:** if thread  $t$ 's current access is at position  $i$ , thread  $t$ 's next access should be at position  $i+1$ .

**Coalescing:** if thread  $t$ 's current access is at position  $i$ , thread  $t+1$ 's current access should be at position  $i+1$ .

### Warning

Uncoalesced access on GPUs and non-cached loads on CPUs *greatly* reduces performance (can be 10X)

Consider the array summation example:

```
View<double*, Space> data("data", size);  
...populate data...  
  
double sum = 0;  
Kokkos::parallel_reduce("Label",  
    RangePolicy< Space>(0, size),  
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {  
        valueToUpdate += data(index);  
    },  
    sum);
```

Question: is this cached (for OpenMP) and coalesced (for Cuda)?

Consider the array summation example:

```
View<double*, Space> data("data", size);
...populate data...

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Space>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

Question: is this cached (for OpenMP) and coalesced (for Cuda)?

Given P threads, **which indices** do we want thread 0 to handle?

Contiguous:

0, 1, 2, ..., N/P

Strided:

0, N/P, 2\*N/P, ...

Consider the array summation example:

```
View<double*, Space> data("data", size);
...populate data...

double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Space>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

Question: is this cached (for OpenMP) and coalesced (for Cuda)?

Given P threads, **which indices** do we want thread 0 to handle?

Contiguous:

0, 1, 2, ..., N/P

**CPU**

Strided:

0, N/P, 2\*N/P, ...

**GPU**

**Why?**



## Iterating for the execution space:

```
operator()(int index, double & valueToUpdate) const {  
    const double d = _data(index);  
    valueToUpdate += d;  
}
```

As users we don't control how indices are mapped to threads, so how do we achieve good memory access?

## Iterating for the execution space:

```
operator()(int index, double & valueToUpdate) const {  
    const double d = _data(index);  
    valueToUpdate += d;  
}
```

As users we don't control how indices are mapped to threads, so how do we achieve good memory access?

### Important point

Kokkos maps indices to cores in **contiguous chunks** on CPU execution spaces, and **strided** for Cuda.

## Rule of Thumb

Kokkos index mapping and default layouts provide efficient access if **iteration indices** correspond to the **first index** of array.

### Example:

```
View<double***, ...> view(...);  
...  
Kokkos::parallel_for("Label", ... ,  
  KOKKOS_LAMBDA (int workIndex) {  
  ...  
  view(..., ... , workIndex ) = ...;  
  view(... , workIndex, ... ) = ...;  
  view(workIndex, ... , ... ) = ...;  
  });  
...
```

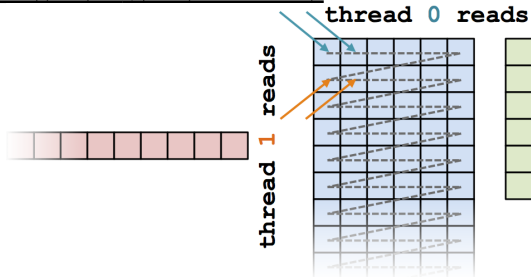
## Important point

Performant memory access is achieved by Kokkos mapping parallel work indices **and** multidimensional array layout *appropriately for the architecture*.

## Important point

Performant memory access is achieved by Kokkos mapping parallel work indices **and** multidimensional array layout *appropriately for the architecture*.

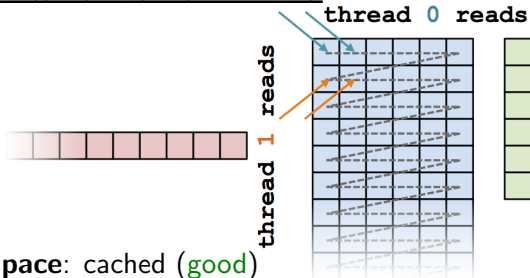
### Analysis: row-major (LayoutRight)



## Important point

Performant memory access is achieved by Kokkos mapping parallel work indices **and** multidimensional array layout *appropriately for the architecture*.

### Analysis: row-major (LayoutRight)

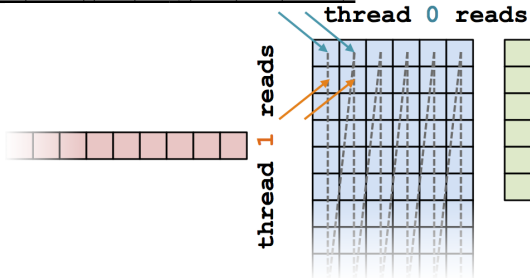


- ▶ **HostSpace**: cached (good)
- ▶ **CudaSpace**: uncoalesced (bad)

## Important point

Performant memory access is achieved by Kokkos mapping parallel work indices **and** multidimensional array layout *optimally for the architecture*.

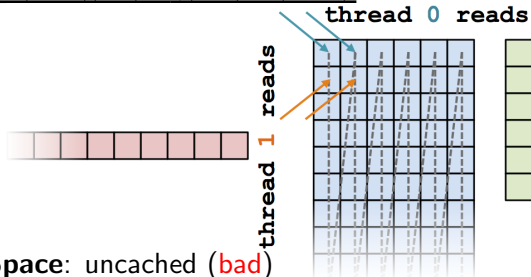
### Analysis: column-major (LayoutLeft)



## Important point

Performant memory access is achieved by Kokkos mapping parallel work indices **and** multidimensional array layout *optimally for the architecture*.

### Analysis: column-major (LayoutLeft)

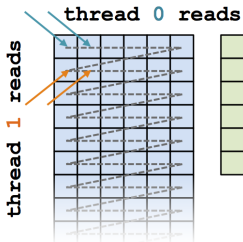


- ▶ **HostSpace**: uncached (**bad**)
- ▶ **CudaSpace**: coalesced (**good**)

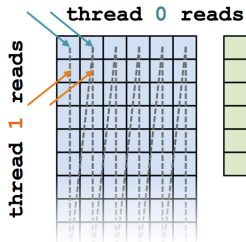


## Analysis: Kokkos architecture-dependent

```
View<double**, ExecutionSpace> A(N, M);
parallel_for(RangePolicy< ExecutionSpace>(0, N),
  ... thisRowsSum += A(j, i) * x(i);
```



(a) OpenMP

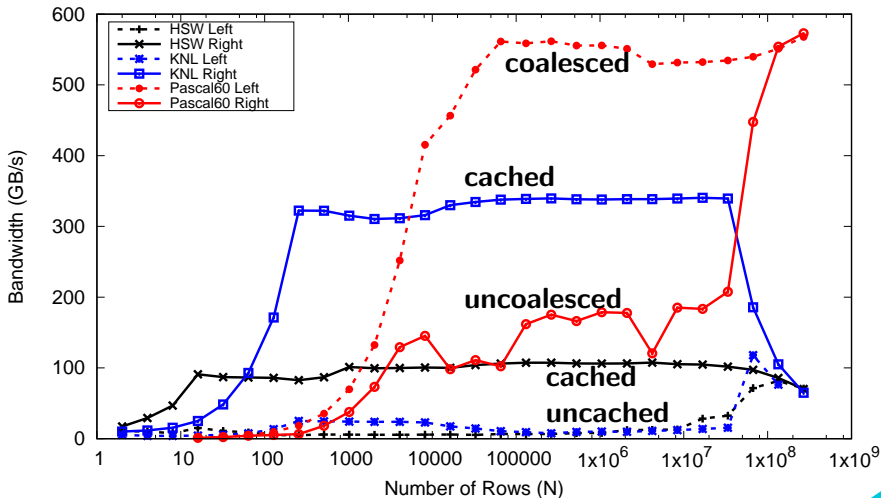


(b) Cuda

- ▶ **HostSpace**: cached (good)
- ▶ **CudaSpace**: coalesced (good)

# <y|Ax> Exercise 04 (Layout) Fixed Size

KNL: Xeon Phi 68c HSW: Dual Xeon Haswell 2x16c Pascal60: Nvidia GPU



- ▶ Every View has a Layout set at compile-time through a **template parameter**.
- ▶ LayoutRight and LayoutLeft are **most common**.
- ▶ Views in HostSpace default to LayoutRight and Views in CudaSpace default to LayoutLeft.
- ▶ Layouts are **extensible** and **flexible**.
- ▶ For performance, memory access patterns must result in **caching** on a CPU and **coalescing** on a GPU.
- ▶ 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 View

- ▶ Multi Dimensional Array.
- ▶ Compile and Runtime Dimensions.
- ▶ Reference counted like a `std::shared_ptr` to an array.

```
Kokkos::View<int*[5]> a("A", N);  
a(3,2) = 7;
```

## Execution Spaces

- ▶ Parallel operations execute in a specified **Execution Space**
- ▶ Can be controlled via template argument to **Execution Policy**
- ▶ If no Execution Space is provided use  
`DefaultExecutionSpace`

```
// Equivalent:  
parallel_for("L", N, functor);  
parallel_for("L",  
    RangePolicy<DefaultExecutionSpace>(0, N), functor);
```

## Memory Spaces

- ▶ Kokkos Views store data in **Memory Spaces**.
- ▶ Provided as template parameter.
- ▶ If no Memory Space is given, use `Kokkos::DefaultExecutionSpace::memory_space`.
- ▶ `deep_copy` is used to transfer data: no hidden memory copies by Kokkos.

```
View<int*, CudaSpace> a("A", M);  
// View in host memory to load from file  
auto h_a = create_mirror_view(a);  
load_from_file(h_a);  
// Copy  
deep_copy(a, h_a);
```

## Layouts

- ▶ Kokkos Views use an index mapping to memory determined by a **Layout**.
- ▶ Provided as template parameter.
- ▶ If no **Layout** is given, derived from the execution space associated with the memory space.
- ▶ Defaults are good if you parallelize over left most index!

```
View<int**, LayoutLeft> a("A", N, M);
View<int**, LayoutRight> b("B", N, M);

parallel_for("Fill", N, KOKKOS_LAMBDA(int i) {
    for(int j = 0; j < M; j++) {
        a(i,j) = i * 1000 + j; // coalesced
        b(i,j) = i * 1000 + j; // cached
    }
});
```