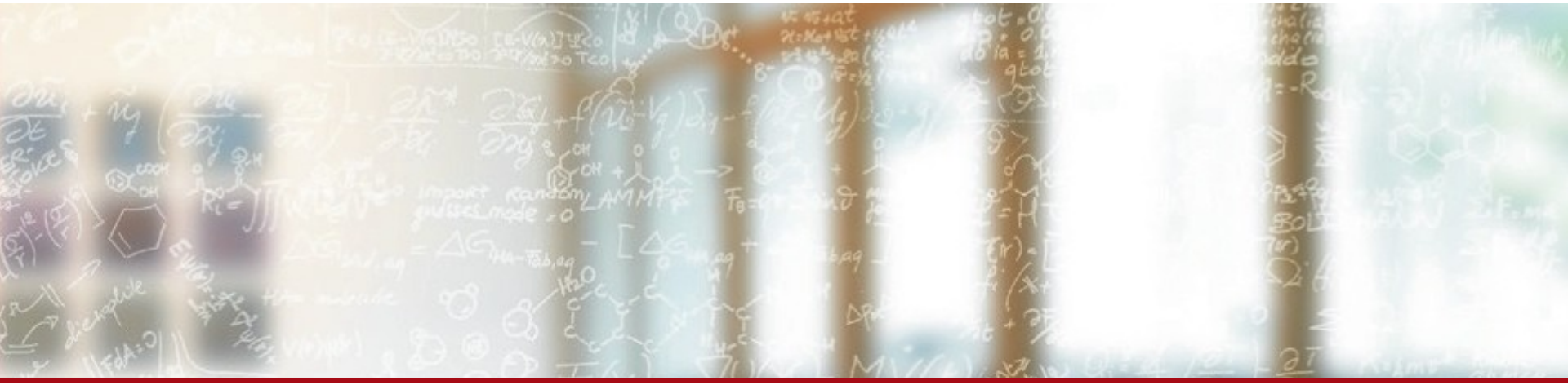
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Introduction to Kokkos

Advanced C++ course

Nur A. Fadel, CSCS

October 13th, 2021

Table of Contents

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Motivation

Introduction

- Kokkos is C++ Performance Portability
  - Write a **single** source implementation using C++
  - Use a descriptive Programming Model
  - Compile code for CPU and GPU

- Kokkos is Ready for Use
  - Developed in 5 National Labs
  - Estabished at Sandia National Lab in 2012
  - Currently +100 projects using Kokkos
  - Compiles with GCC 5+, Clang4+, NVCC9+, XL16

The Kokkos Team

C.R.Trott, J. Ciesko, V. Dang, N. Ellingwood, D.S. Hollman, D. Ibanez, J. Miles, J. Wilke, , H. Finkel, N. Liber, D. Lebrun-Grandie, D. Arndt, B. Turcksin, J. Madsen, R. Gayatri, S. Rajamanickam, L. Berger, V. Dang, N. Ellingwood, E. Harvey, B. Kelley, K. Kim, C.R. Trott, J. Wilke, S. Acer, D. Poliakoff,

former: H.C. Edwards, D. Labreche, G. Mackey, S. Bova, D. Sunderland

Why Kokkos? - The Hardware Landscape

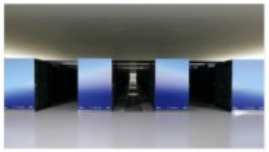
Current Generation: Programming Models OpenMP 3, CUDA and OpenACC depending on machine

LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3

LLNL SIERRA
IBM Power9 / NVIDIA Volta
CUDA / *OpenMP*[a]

ORNL Summit
IBM Power9 / NVIDIA Volta
CUDA / *OpenACC / OpenMP*[a]

SNL Astra
ARM CPUs
OpenMP 3



Riken Fugaku
ARM CPUs with SVE
OpenMP 3 / *OpenACC*[b]

Upcoming Generation: Programming Models OpenMP 5, CUDA, HIP and DPC++ depending on machine

NERSC Perlmutter
AMD CPU / NVIDIA GPU
CUDA / *OpenMP 5*[c]

ORNL Frontier
AMD CPU / AMD GPU
HIP / *OpenMP 5*[d]

ANL Aurora
Xeon CPUs / Intel GPUs
DPC++ / *OpenMP 5*[e]

LLNL El Capitan
AMD CPU / AMD GPU
HIP / *OpenMP 5*[d]

- OpenMP 5 available only on new machine
- Not clear how OpenMP 5 will be interoperable among vendors
- CUDA
- HIP
- SYCL

Why Kokkos? - The Cost of Coding

Industry Estimate: A full time software engineer writes 10 lines of production code per hour: 20k LOC/year.

- Typical HPC production app: 300k-600k lines
- Large Scientific Libraries:
  - E3SM: 1,000k lines
  - Trilinos: 4,000k lines

Conservative estimate: need to rewrite 10% of an app to switch Programming Model

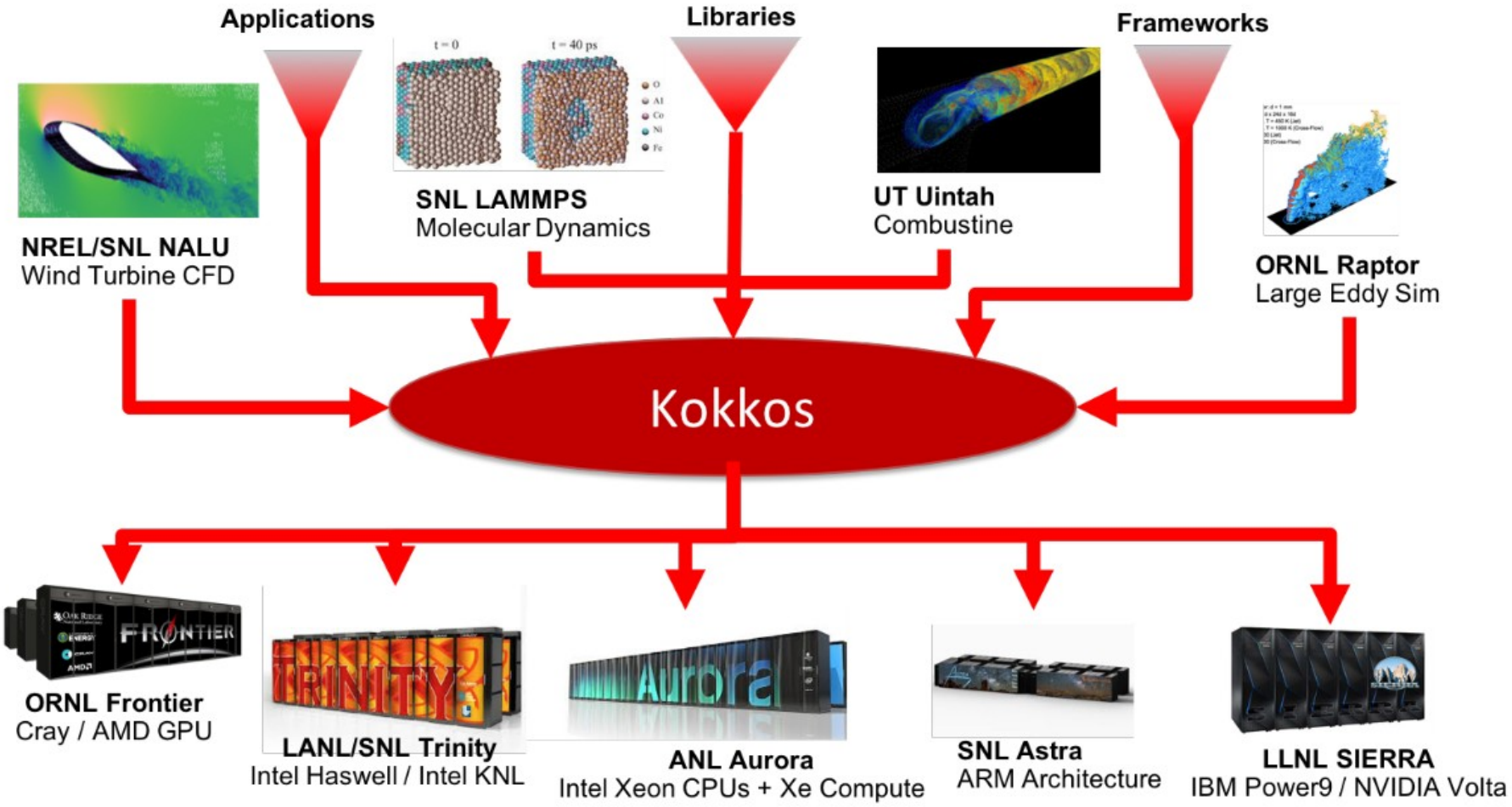
Just switching Programming Models costs

multiple person-years per app!

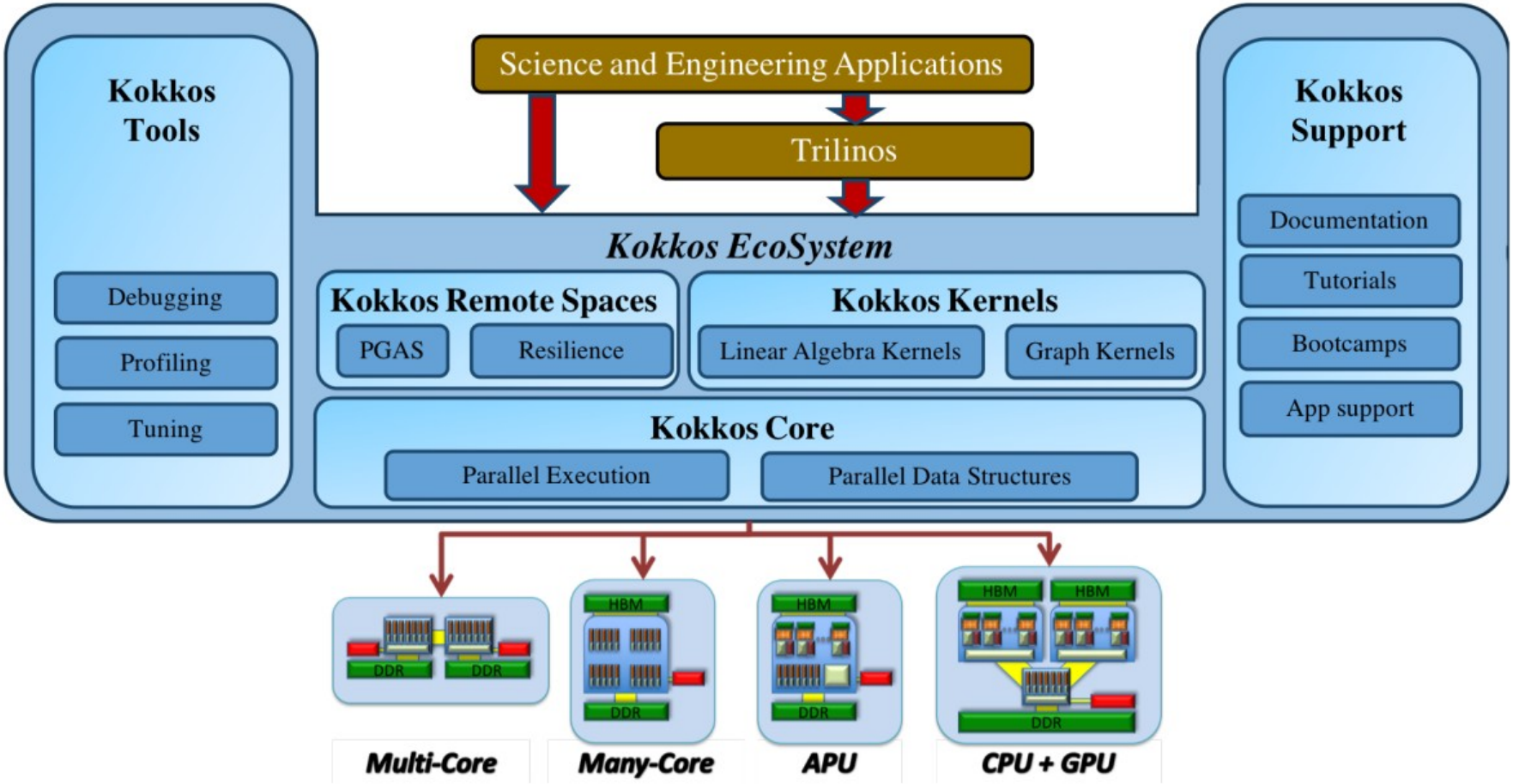
What is Kokkos?

- A C++ programming model for **performance** portability
  - C++ Templated library on top CUDA, HIP, SyCL
  - It is descriptive not prescriptive
  - Aligned with develpoments of C++ standard
- Address common needs of modern science:
  - Math library based on Kokkos
  - Tools for Profiling and debugging
  - Utilities for interfacing also Python and Fortran
- Open source
  - https://github.com/kokkos

Kokkos at the Center



The Kokkos EcoSystem

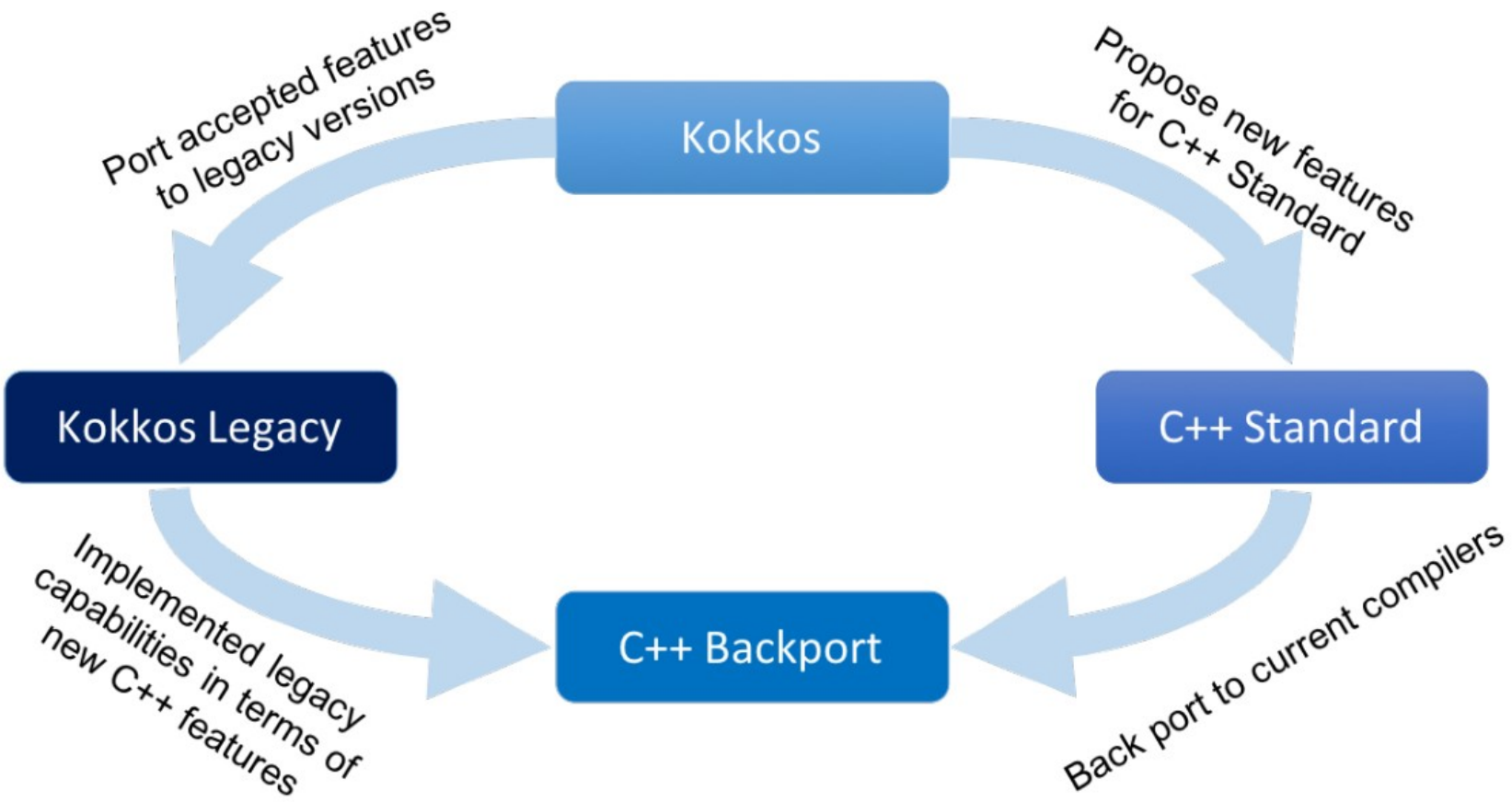


Additional Tools

- KokkosKernels
  - A BLAS/Laapack library – tools for Sparse and Linear Algebra
- Kokkos Remote Spaces
  - Internode parallelization – still experimental
- KokkosProfiler
  - Tools to debug and to profile your Kokkos code
- KokkosTutorial
  - A set of self contained example for each concept of Kokkos

Kokkos helps improve C++

- Kokkos developers push to merge Kokkos parts into C++ standard
- 10 people from Kokkos team are part of C++ std committee



Example: C++23 `std::mdspan`

- C++ does not have multi-dimensional arrays
  - Fortran, Matlab, Python have them
- C++23 `std::mdspan` adds `Kokkos::View` like arrays
  - Data layouts allow adapting to hardware specific access patterns
  - Subviews
  - Reference Semantics
  - Compile time and Runtime extents

```
View <int **[5],LayoutLeft> a("A",10 ,12); a(3,5,1) = 5;
```

How to build

- `git clone https://github.com/kokkos/kokkos`
- Configure with cmake
- `make & make install`

OR

- `spack install kokkos +openmp`

  (it takes a while from scracth)

Online Resources:

- Primary Kokkos GitHub Organization
  - https://github.com/kokkos
- Wiki including API reference
  - https://github.com/kokkos/kokkos/wiki
- Slack channel for Kokkos:
  - https://kokkosteam.slack.com
- Additional tutorials
  - https://github.com/kokkos/kokkos-tutorials/wiki/

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Basic Concepts

Takeaways

Kokkos' basic capabilities:

- Simple 1D data parallel computational patterns
- Deciding where code is run and where data is placed
- Managing data access patterns for performance portability

Performance Portability

Example:

implementations may target particular architectures and may not be thread scalable. (e.g., locks on CPU won't scale to 100,000 threads on GPU)

Goal: write **one implementation** which:

- compiles and **runs on multiple architectures**,
- obtains **performant memory access patterns** across architectures,
- can leverage **architecture-specific features** where possible

The prerequisites – body, pattern, policy

```
for ( element = 0; element < numElements ; ++ element ) {
  total = 0;
  for ( qp = 0; qp < numQPs ; ++ qp ) {
   total += dot ( left [ element ][ qp ] , right [ element ][ qp ]);
  }
  elementValues [ element ] = total;
}
```

- Pattern – type of pattern
- Execution policy – controls how things are executed
- Computational Body – the unit of work, what you want to do
- pattern and policy drive the computational body

What if we want to thread the loop?

```
#pragma omp parallel for
for ( element = 0; element < numElements ; ++ element ) {
 total = 0;
 for ( qp = 0; qp < numQPs ; ++ qp ) {
  total += dot ( left [ element ][ qp ] , right [ element ][ qp ]);
 }
 elementValues [ element ] = total;
}
```

- OpenMP is simple for parallelizing loops on multi-core CPUs
- but what if we then want to do this on other architectures?
  - OpenACC, OpenMP Target, OpenCL

What if we want to thread the loop?

A standard thread parallel programming model **may** give you portable parallel execution if it is supported on the target architecture.

- But what about performance?

- Performance depends upon the computation's **memory access pattern.**

Data parallel patterns and work

```
for ( atomIndex = 0; atomIndex < numberOfAtoms ; ++ atomIndex )
{
atomForces [ atomIndex ] = c alculate Force (... data ...);
}
```

Kokkos maps work to execution resources

- each iteration of a computational body is a unit of work.
- an **iteration index** identifies a particular unit of work.
- an **iteration range** identifies a total amount of work

You give an **iteration range** and **computational body** (kernel) to Kokkos, and Kokkos decides how to map that work to execution resources.

Data parallel patterns and work - Functors

How are computational bodies given to Kokkos?

- As functors or function objects, a common pattern in C++

```
struct ParallelFunctor {
...
void operator ()( a work assignment ) const {
/* ... computational body ... */
...
};
```

A total amount of work items is given to a Kokkos pattern:

```
ParallelFunctor functor ;
Kokkos::parallel_for (numberOfIterations, functor );
```

- A parallel functor body **must have access** to all the data it needs through the functor's data members.

Data parallel patterns and work - Lambdas

Functors are tedious ⇒ C++11 Lambdas are concise!

```
atomForces already exists
data already exists
Kokkos::parallel_for ( numberOfAtoms ,
    [=] ( const int64_t atomIndex ) {
    atomForces [ atomIndex ] = calculateForce( data );
}
);
```

The compiler is auto-generating a functor for you from the lamba

For portability to GPU a lambda must **capture by value** [=].

- Don’t capture containers (e.g., std::vector) by value because it will copy the container’s entire contents.

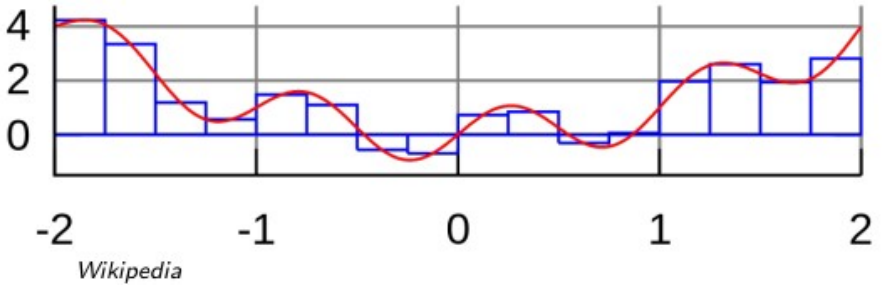
How does this compare to OpenMP?

```
// SERIAL
for ( int64_t i = 0; i < N ; ++ i ) {
/* loop body */
}
//OpenMP
# pragma omp parallel for
for ( int64_t i = 0; i < N ; ++ i ) {
/* loop body */
}
// KOKKOS
parallel_for ( N , [=] ( const int64_t i ) {
/* loop body */
});
```

Simple Kokkos usage is no more conceptually difficult than OpenMP, the annotations just go in different places.

Example: Riemann-sum-style numerical integration

$$y = \int_{lower}^{upper} function(x)\, dx$$




```
double totalIntegral = 0;
for ( int64_t i = 0; i < numberOfIntervals ; ++ i ) {
    const double x =
     lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function( x );
    totalIntegral += thisIntervalsContribution ;
}
totalIntegral *= dx ;
```

How do we parallelize it? Correctly?

Example: Riemann-sum-style numerical integration

```
double totalIntegral = 0;
Kokkos :: parallel_for ( numberOfIntervals,
[=] ( const int64_t index ) {
    const double x =
    lower + ( index/numberOfIntervals) * ( upper - lower );
    totalIntegral += function ( x );} ,
);
totalIntegral *= dx ;
```

compiler error: cannot increment *totalIntegral*!

(lambdas capture by value and are treated as *const*)

Root Problem: wrong pattern

We're using the wrong pattern, **for** instead of **reduction.**

Reductions combine the results contributed by parallel work.

```
//OpenMP
double finalReducedValue=0;
#pragma omp parallel for reduction (+:finalReducedValue)
for ( int64_t i = 0; i < N ; ++ i ) {
   finalReducedValue += ...
}
//Kokkos
double finalReducedValue = 0;
parallel_reduce(N, functor, finalReducedValue);
```

Reduction

- The operator takes two arguments:
  - a work index and
  - a value to update.
- The second argument is a thread-private value that is managed by Kokkos; it is not the final reduced value

```
double totalIntegral = 0;
parallel_reduce ( numberOfIntervals,
    [=] ( const int64_t i , double & valueToUpdate ) {
     valueToUpdate += function (...);
    },
    totalIntegral );
```

Views

Views

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

- 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).
- Are like smart pointers, so copy them in your functors.

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

Views (2)

- Multi-dimensional array of 0 or more dimensions
- Number of dimensions (rank) is fixed at compile-time
- Sizes of dimensions set at compile-time or runtime.
- Access elements via ”(...)” operator
- 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 shared ptr

Example - Views

```
View<double ***> data ("label",N0,N1,N2);
View<double **[ N2 ]> data ("label",N0,N1);
View<double *[N1][N2]> data ("label" , N0);
View<double [N0][N1][N2]> data ("label");
// Access
data (i ,j , k ) = 5.3;
```

Runtime-sized dimensions must come first.

Example - Views

```
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

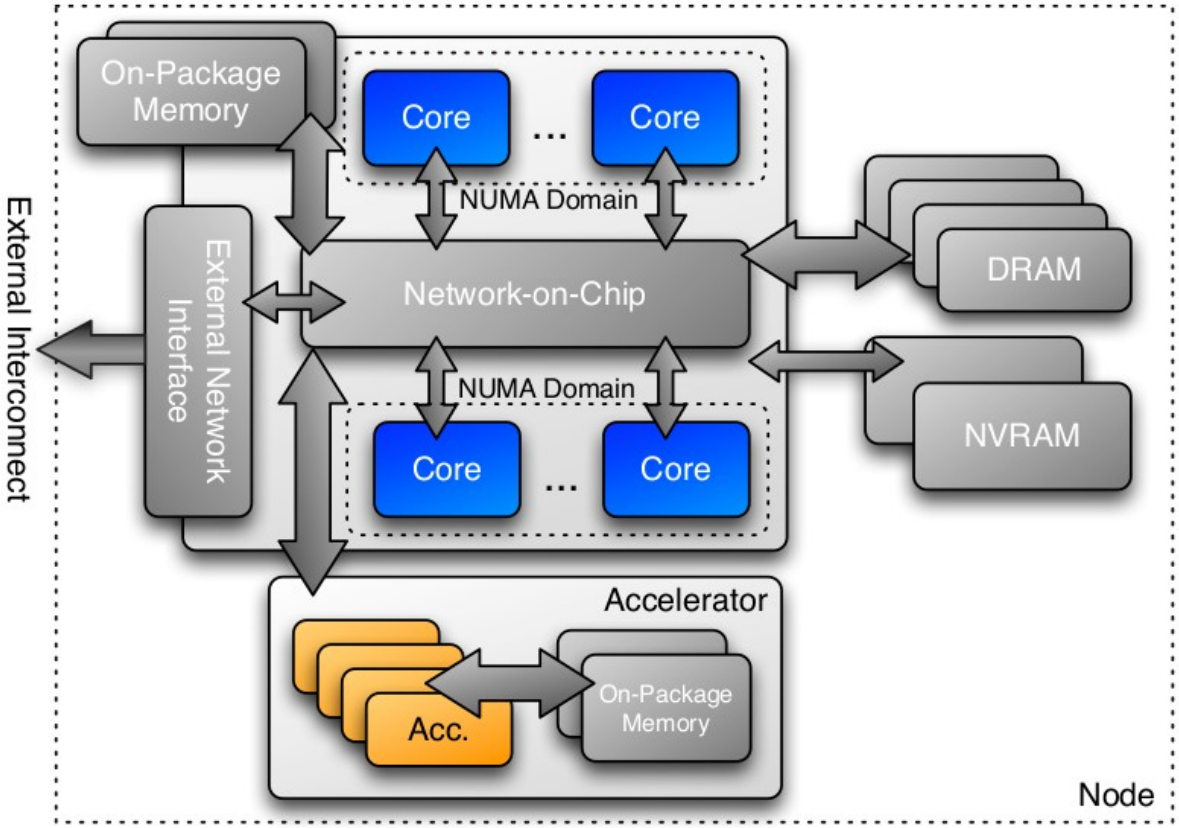
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Execution and Memory Spaces

Execution Spaces

a homogeneous set of cores and an execution mechanism



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

Execution Spaces

```
MPI_Reduce (...);
FILE * file = fopen(...);                    HOST
RunANormalFunction(... data ...);

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

- Host code run always in the host process
- Parallel code be run in the **default execution space**
- How do I control where the Parallel body is executed?
  1) Changing the default execution space (*at compilation*),
  2) specifying an execution space in the **policy**

Execution Spaces

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

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

- *Range Policy* takes an template argument – the **execution space**
- It s a policy that says I am parallelizing over a set of intervals
- *Execution space* is **only** compile time

Requirements for enabling execution spaces:

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

Execution Spaces

The compiler needs to know which function/lambda run on the device

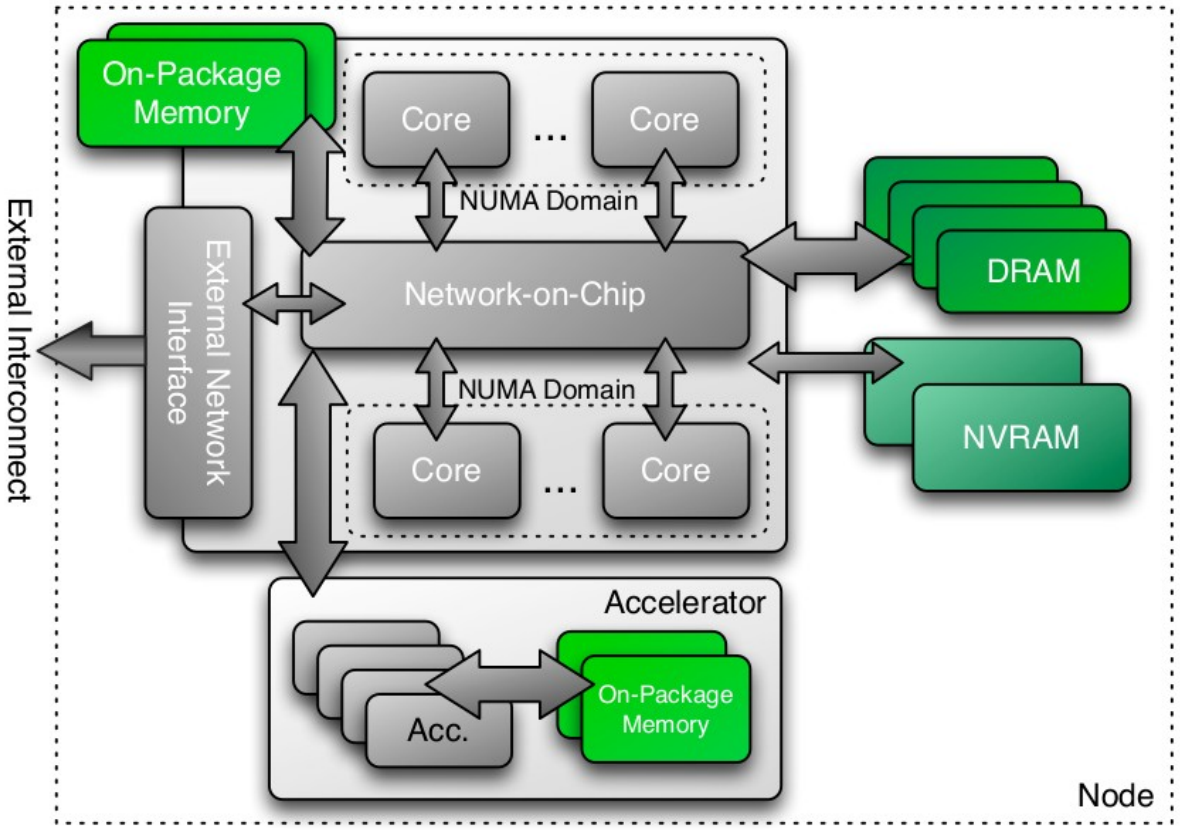
- Kokkos defines two macros:
  - **KOKKOS_LAMBDA** and
  - **KOKKOS_INLINE_FUNCTION**

```
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*/
```

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

Memory Spaces

explicitly-manageable memory resource
aka where we put the data



Memory Spaces

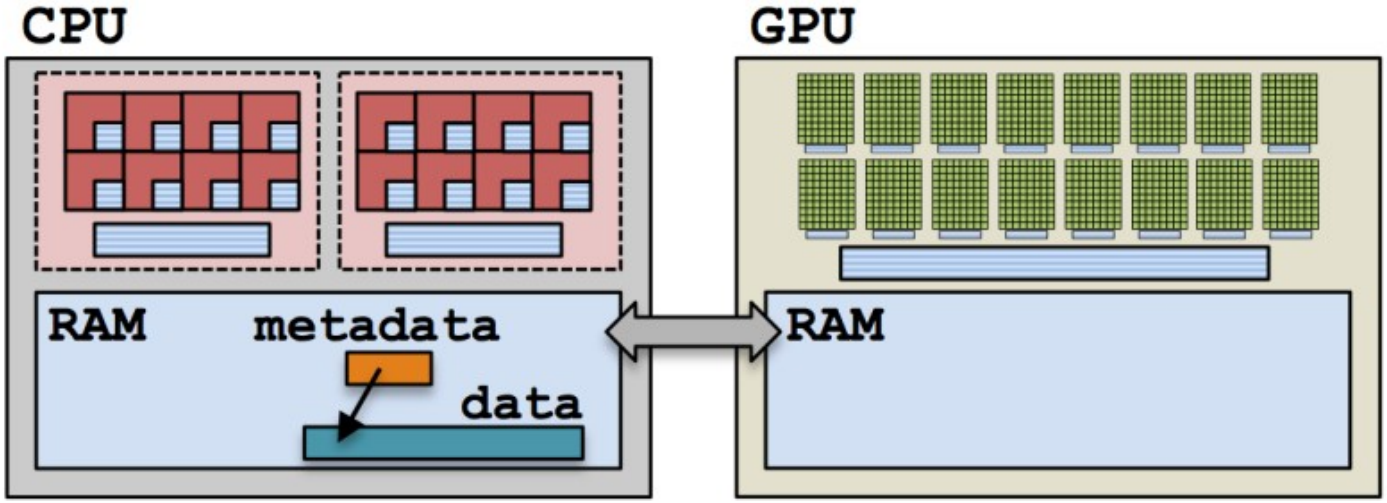
Every view stores its data in a memory space set at compile time.

- Available memory spaces:
  - HostSpace, CudaSpace, CudaUVMSpace, …
- 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

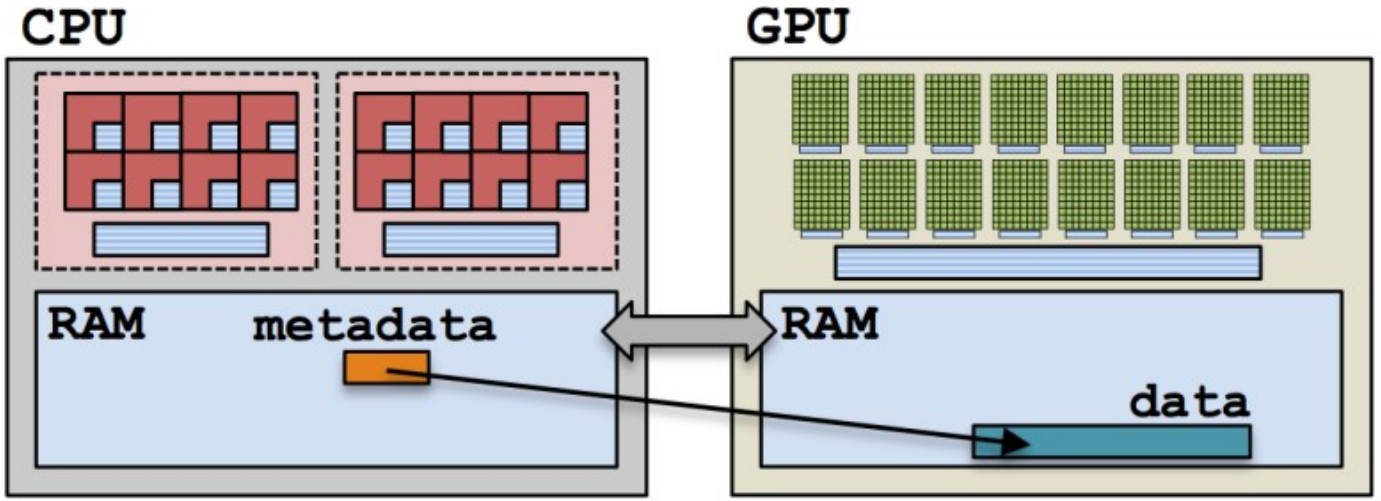
```
//Equivalent:
View <double* > a ("A",N );
View <double* , DefaultExecutionSpace::memory_space> b ("B",N);
```

Example

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




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



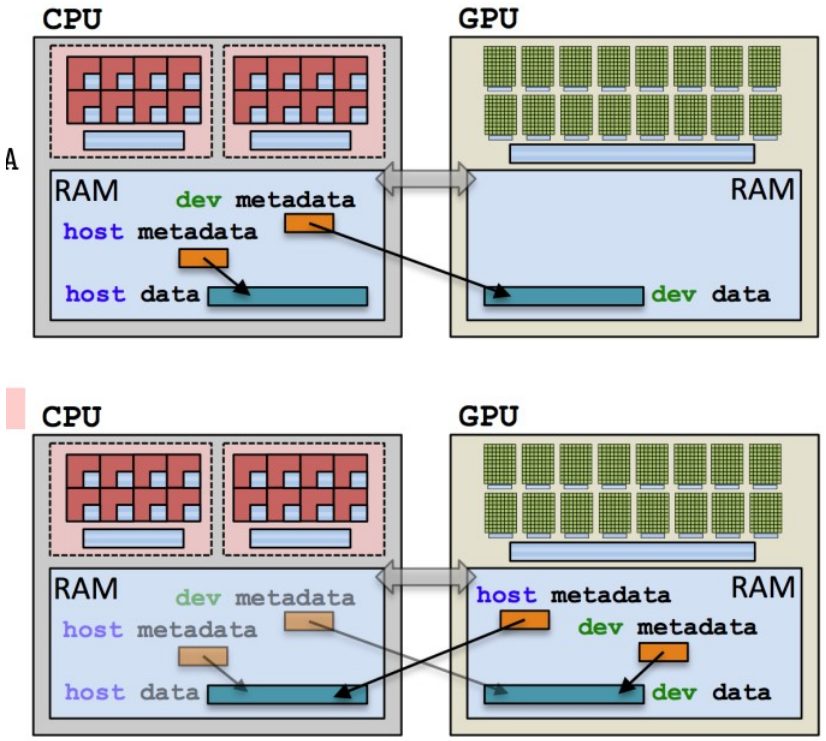
Anatomy of a kernel launch

1) User declares views, allocating
2) User instantiates a functor with views
3) User launches `parallel something`
    1) Functor is copied to the device.
    2) Kernel is run
    3) Copy of functor on the device is released

1) no deep copies of array data are performed
2) views are like pointers

Example: two views

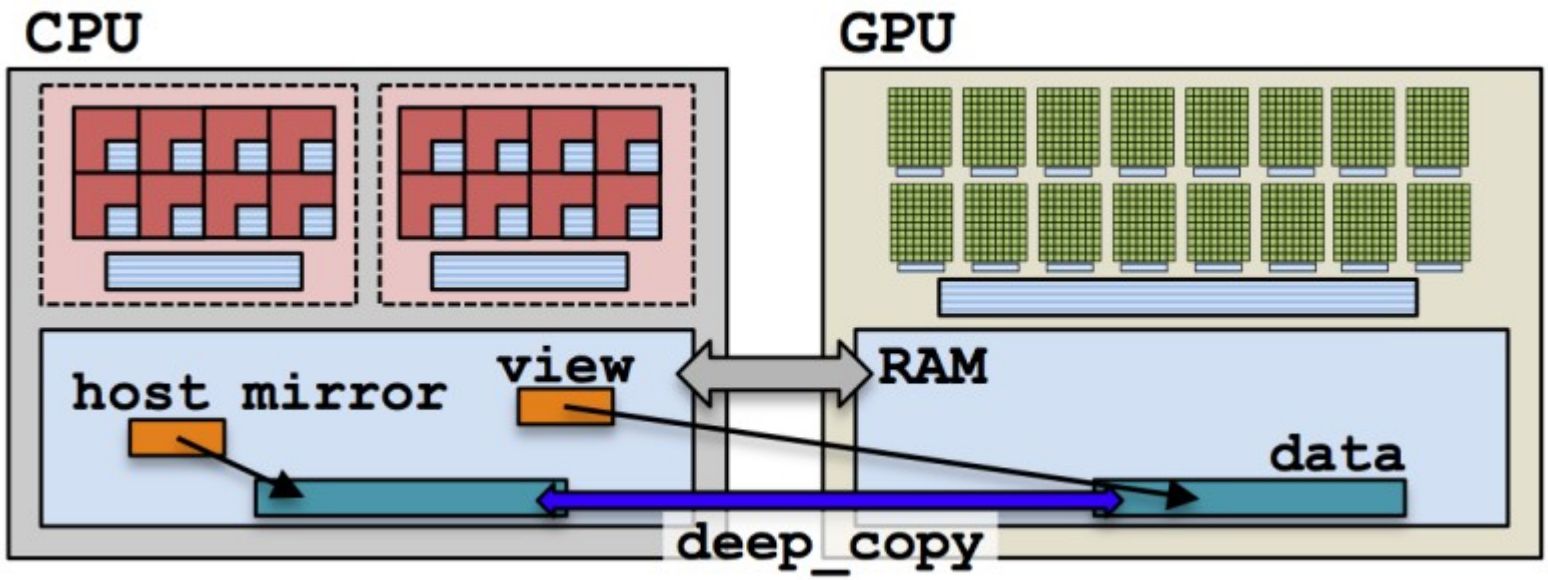
```
#define KL KOKKOS_LAMBDA
View <int*, Cuda> dev;
View <int*, Host> host;
parallel_for ( " Label ",N,
KL (int i) {
    dev(i)=...;
    host(i)=...;
});
```



Mirrors View

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

```
using view_type = Kokkos::View <double**, Space>;

view_type view (...);

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



Mirrors

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::HostMirrorhostView =

    Kokkos::createmirrorview(view);
```

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

4) Deep copy hostView's array to view's array.

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

Mirrors (2)

What if the View is in HostSpace too?

Does it make a copy?

- `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.

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

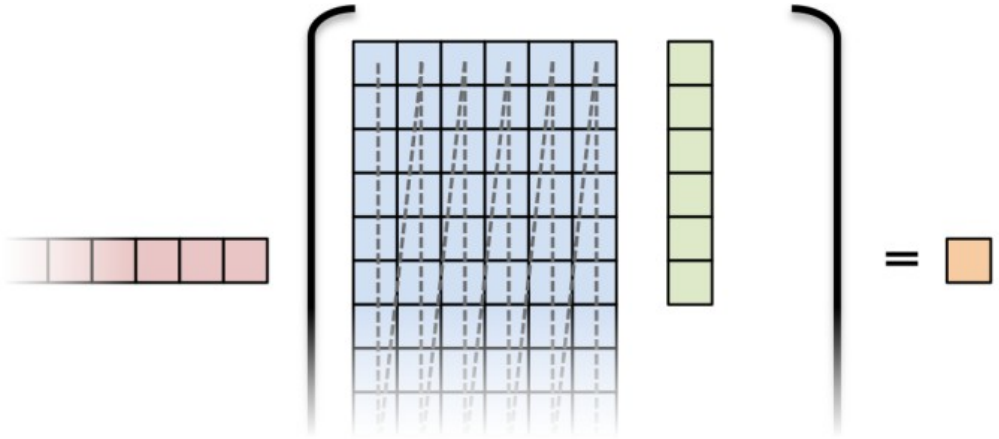
Layout

Layout

Layout is the mapping of multi-index to memory:

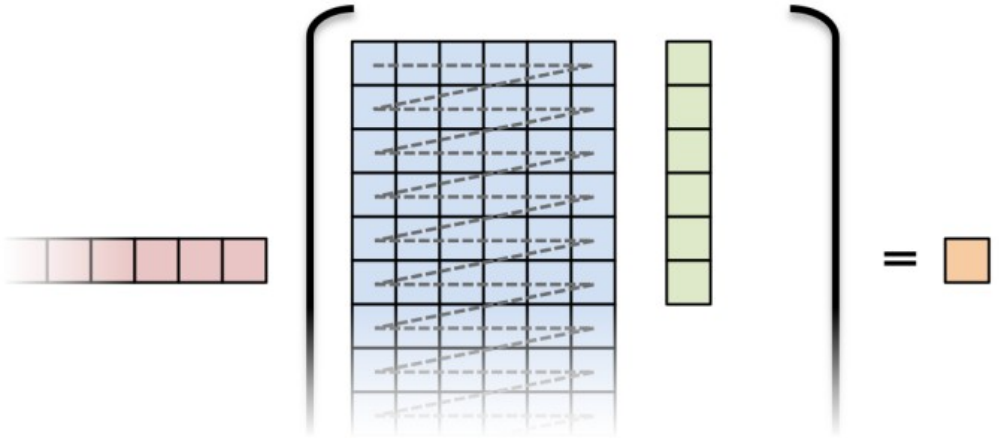
LayoutLeft

in 2D, "column-major"



LayoutRight

in 2D, "row-major"



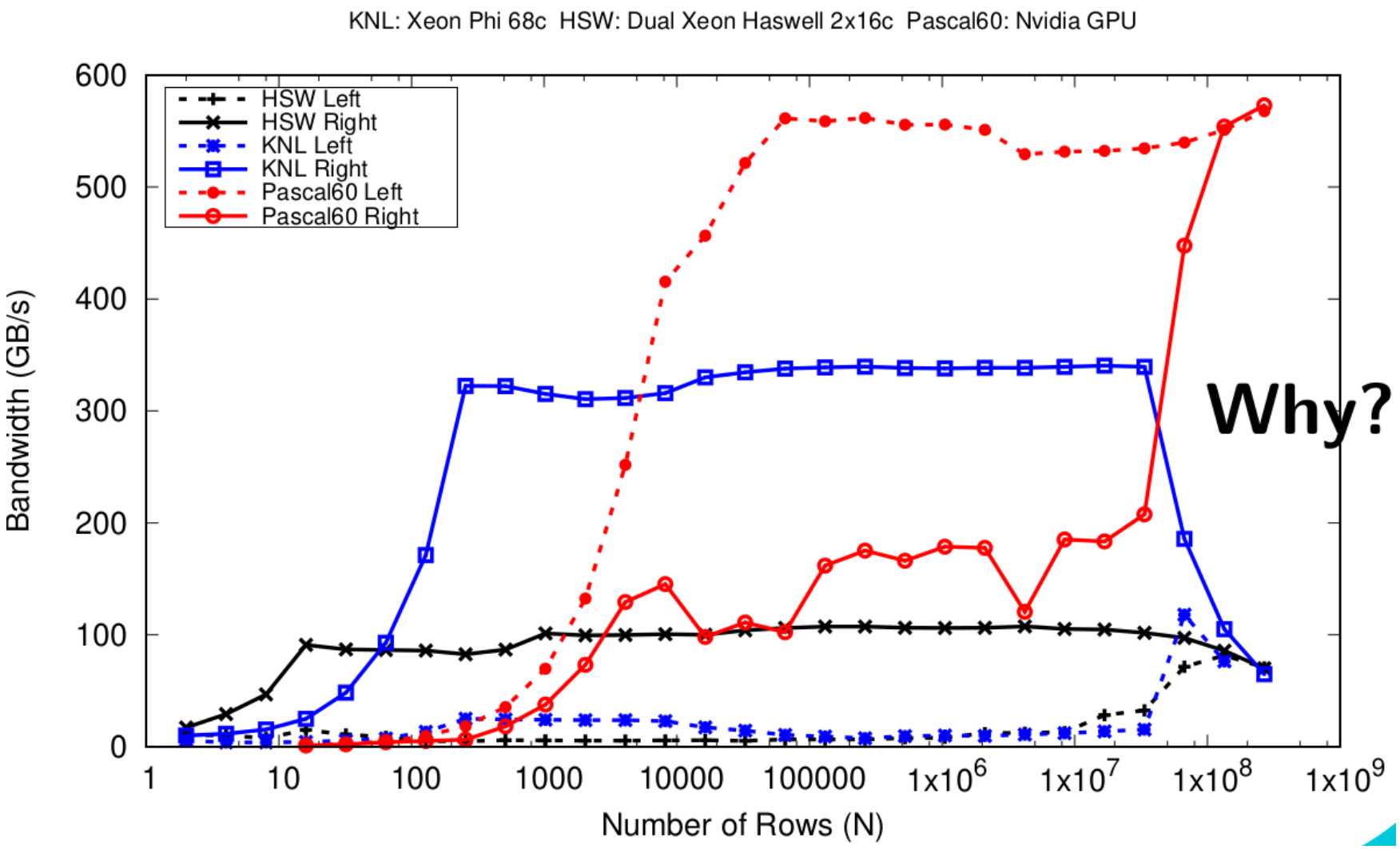
Layout

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

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

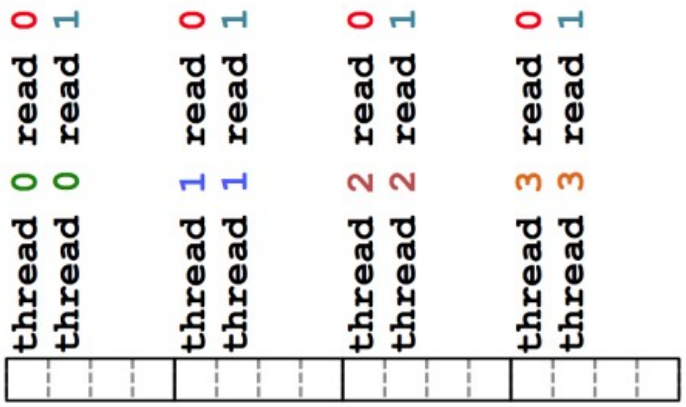
- Most-common layouts are LayoutLeft and LayoutRight.
  - *LayoutLeft*: left-most index is stride 1.
  - *LayoutRight*: right-most index is stride 1.
- If no layout specified, **default** for that memory space is used.
  - *LayoutLeft* for *CudaSpace*, *LayoutRight* for *HostSpace*.
- Layouts are extensible: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

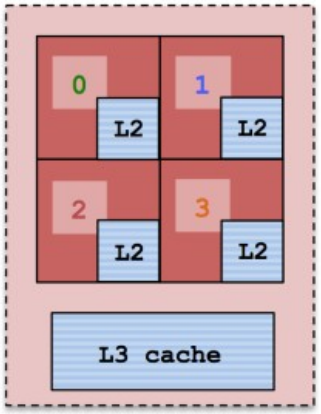
Layout


KNL: Xeon Phi 68c HSW: Dual Xeon Haswell 2x16c Pascal60: Nvidia GPU
HSW Left
HSW Right
KNL Left
KNL Right
Pascal60 Left
Pascal60 Right
Bandwidth (GB/s)
0
100
200
300
400
500
600
1
10
100
1000
10000
100000
1x10^6
1x10^7
1x10^8
1x10^9
Number of Rows (N)
Why?

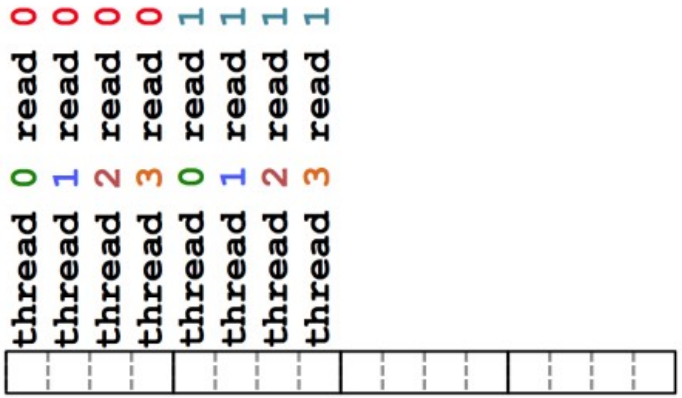
Layout

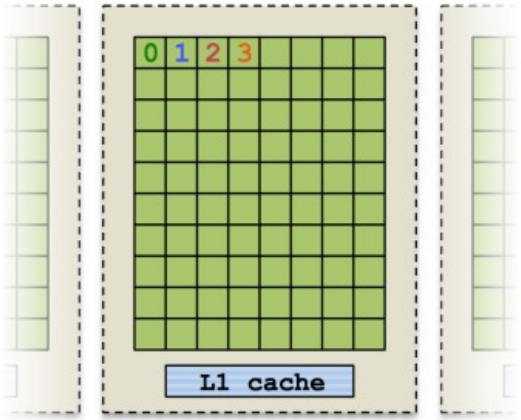
CPUs: few (independent) cores with separate caches:





GPUs: many (synchronized) cores with a shared cache:





Layout

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

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

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

Caching and Coalescing

- **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)!!

Question:

- is this cached (for OpenMP) and coalesced (for Cuda)?
- Given P threads, which indices do we want thread 0 to handle?

Caching and Coalescing

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 - good for CPU

- Strided:
  - 0, N/P, 2*N/P, … - good for GPU

Caching and Coalescing

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

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

Caching and Coalescing

- 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.

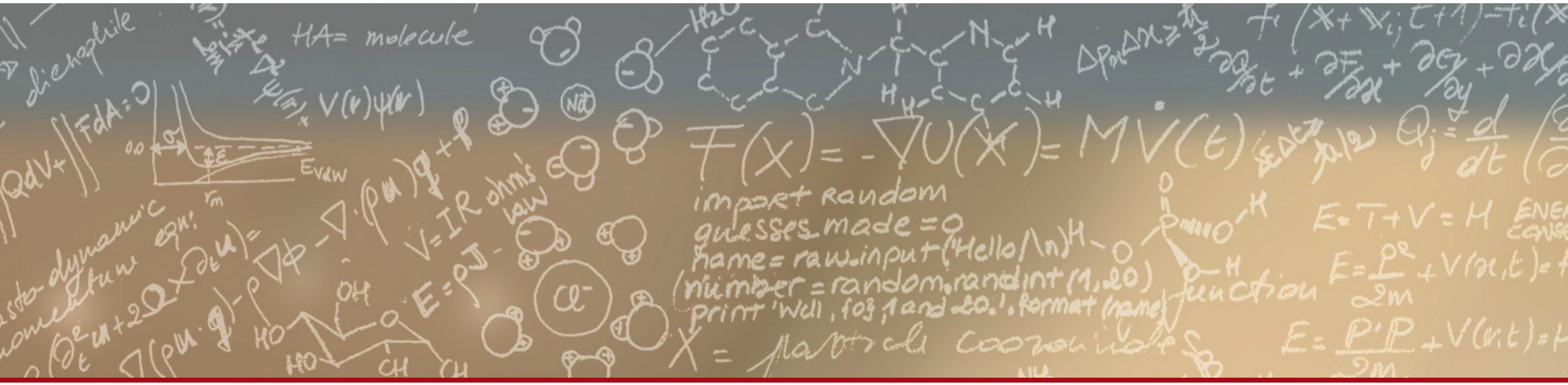
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Conclusion

Conclusion

- Importance of Performace Portability
- Reason why to use Kokkos
- Basic parallel patterns
- Kokkos' main concepts
  - RangePolicy, Body of Work and Paralle Pattern
  - Execution Space
  - Memory Space
  - Mirro View
  - Layout



Thank you for your attention.