Monday, March 20 at 4:30 – 6:00 PM

Kokkos

A Parallel, Portable Programming Model for CPUs and GPUs

Rohit Kakodkar

Research Software Engineer

PRINCETON GEOSCIENCES

PRINCETON Research Computing

What is Kokkos?

- Kokkos is a C++* performance portability library
  - Write a single source implementation
  - Descriptive programming model
  - Compile for CPUs or GPUs
  - Kokkos is a shared memory programming model (works in conjunction with MPI)

- Major buy-in by DOE and national labs
  - LAAMPS, Trilinos, PETSc
  - Over 100 projects using kokkos
  - Contributing to the C++ standard
  - Active kokkos developers community via slack (invite only)

*Python and Fortran bindings are available

Goals

The goal here is to motivate a use case for Kokkos. Given time limit this is not an in-depth tutorial of Kokkos (probably sometime in the future).

- Outline of the talk:
    - Need for performance portability
    - Understanding performance
        - Caching (CPUs) vs Coalescing(GPUs)
    - Kokkos views
    - Kokkos parallel construct
        - Hands-on example
        - Parallel policies
    - Packaging Kokkos projects
    - Why use kokkos?

Need for performance portability

Modern high-performance computing applications need to run efficiently across multiple architectures:

	GPU			CPU
Architecture				
Programming model	CUDA	HIP	DPC++ (SYCL)	OpenMP pthreads

Need for performance portability

Modern high-performance computing applications need to run efficiently across multiple architectures:

Pre-Exascale

LANL Trinity
Intel Haswell/ Intel KNL
OpenMP 3

ORNL Summit
NVIDIA Volta100
OpenMP/CUDA

Exascale

ORNL Frontier
AMD GPUs
OpenMP/HIP

ANL Aurora
Intel GPUs
DPC++

Moving into the exascale era will require adapting your applications to utilize modern architectures

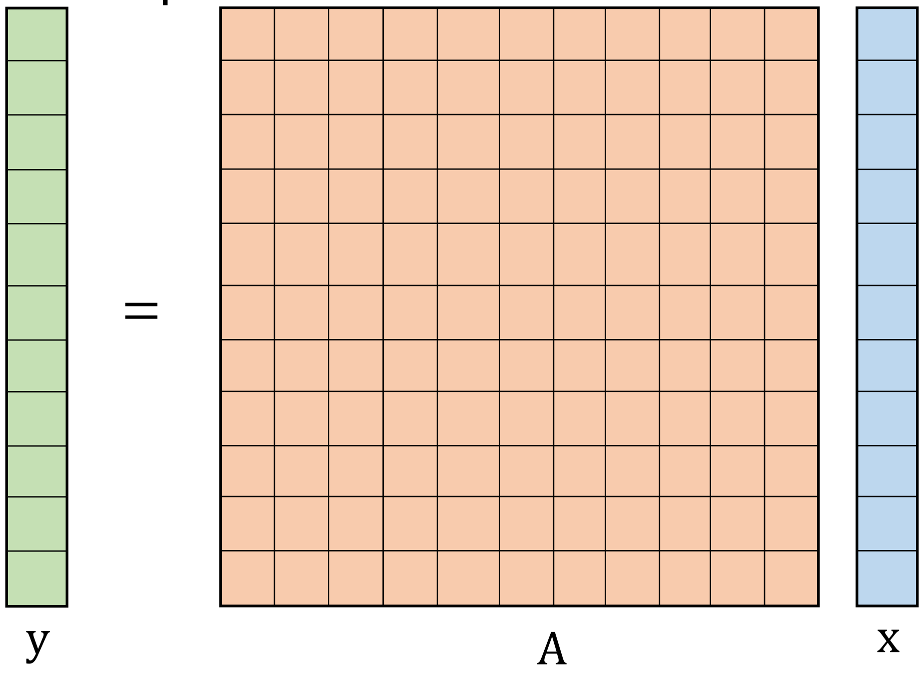
Need for performance portability

- Problem: Porting applications for various architectures is time consuming
  - Typical HPC application : 300k – 600k lines of code
    - Smaller scale applications : SpecFEM2D + SpecFEM3D ~ 100k lines of code (conservative estimate)
    - Porting requires ~10% rewrite of the application
    - Typical software engineer writes about 20k LOC/year

- Potential portability options: OpenMP 5 or OpenACC

What about performance??

- Example: Matrix – vector multiplication

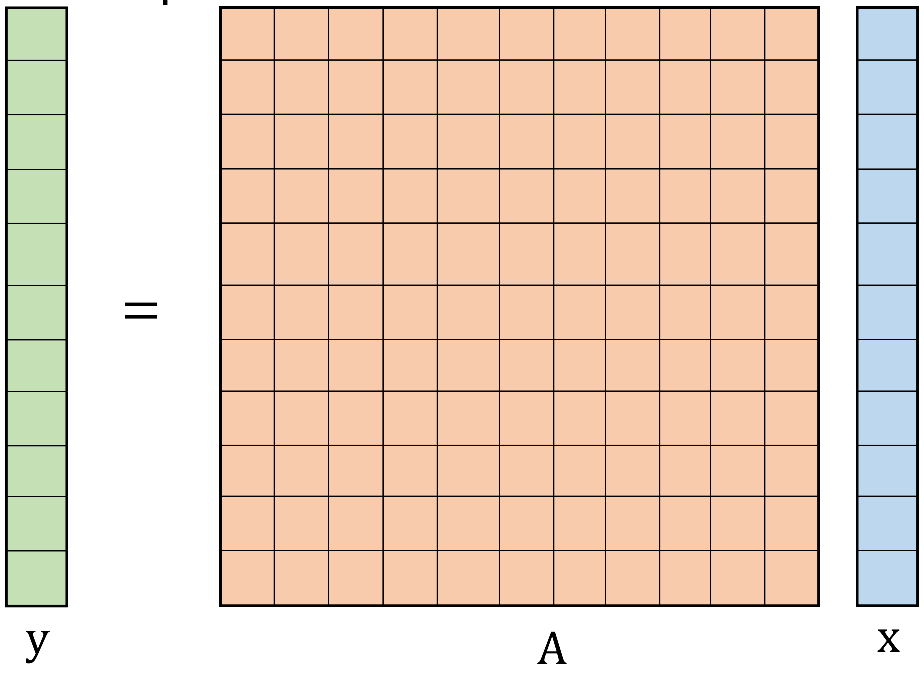


Serial Implementation

```
for (int j = 0; j < N; j++){
        for (int i = 0; i < M; i++){
                y(j) += A(j,i)*x(i);
        }
}
```

Understanding performance

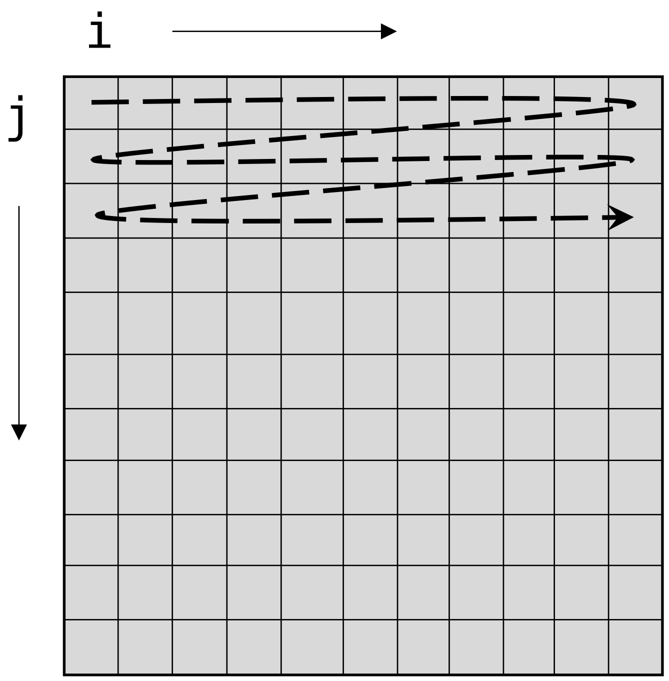
- Example: Matrix – vector multiplication



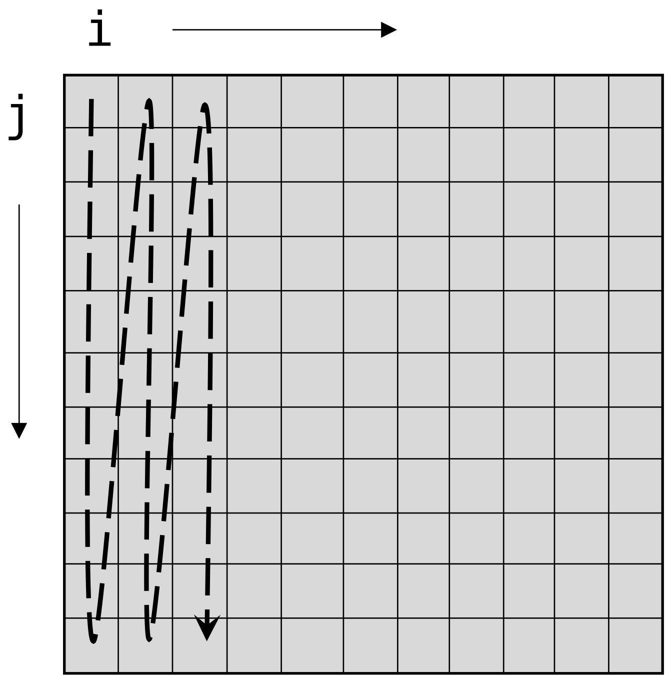
Parallel Implementation

```
parallel_for (int j = 0; j < N; j++){
// distribute different rows to different threads
        for (int i = 0; i < M; i++){
                y(j) += A(j,i)*x(i);
        }
}
```

- Performance is dependent on matrix layout



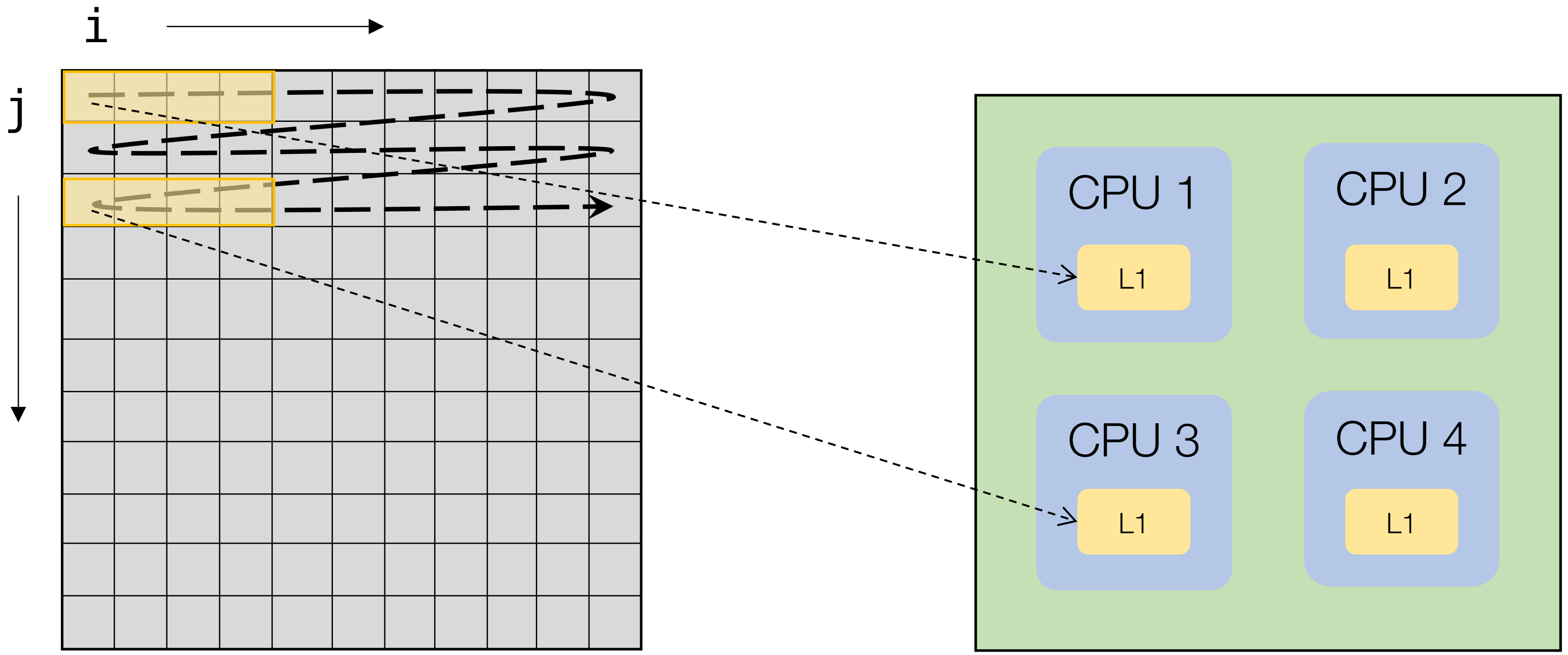
Row major format



Column major format

CPU performance

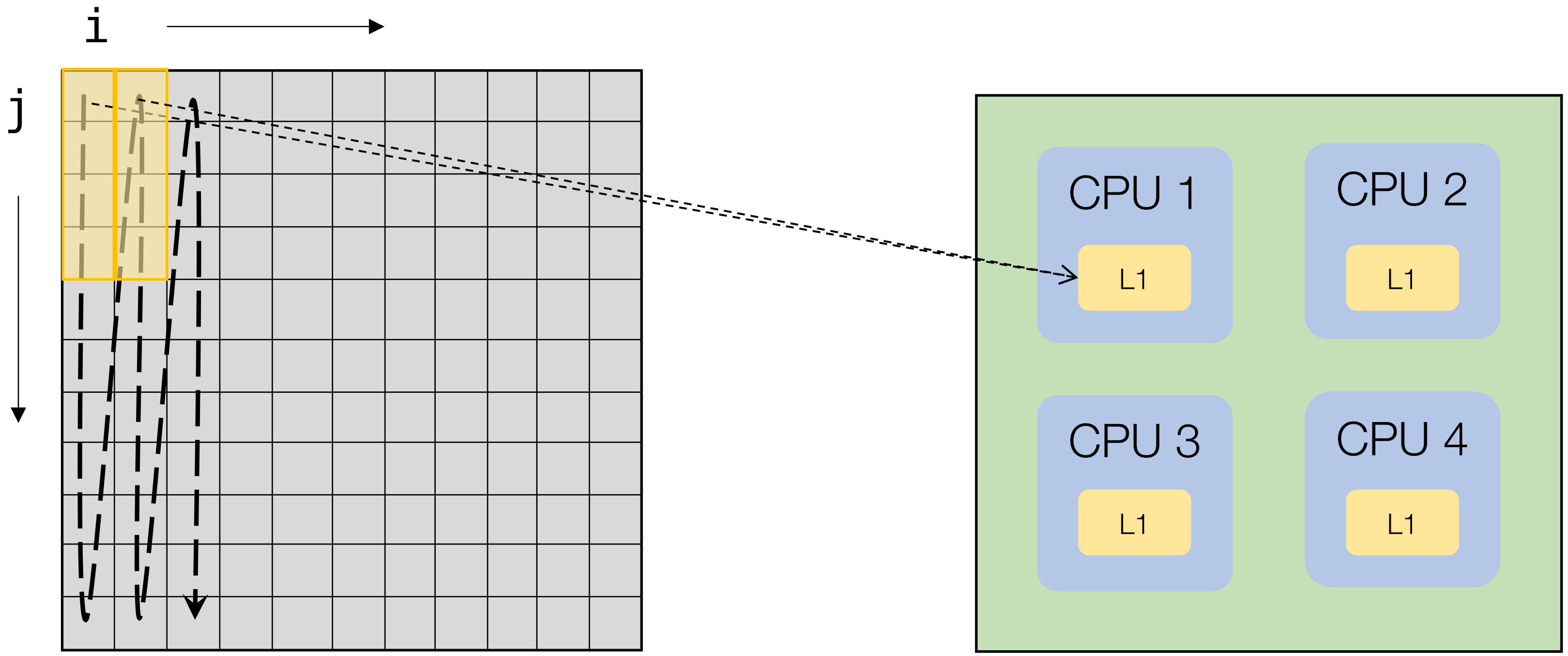
- Good performance



Row major format

```
parallel_for j: N
// every thread gets a specific j to execute
   for i: M
      A(j,i)
```

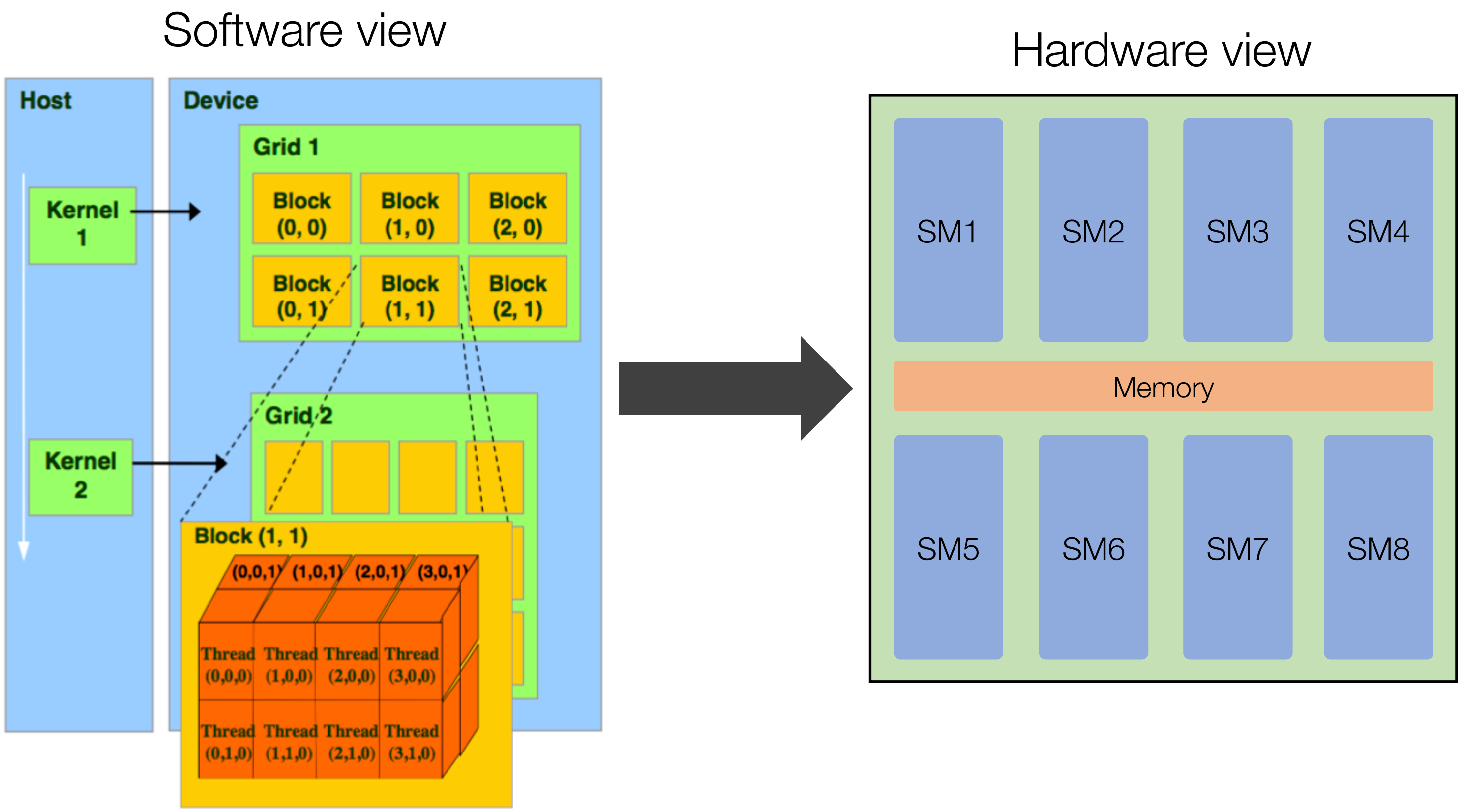
- Bad performance

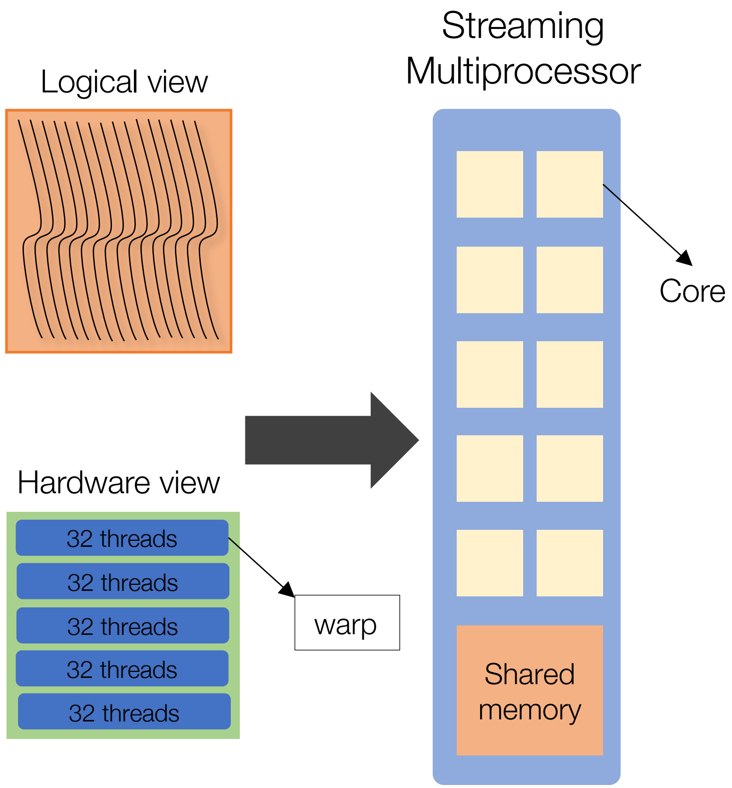


Column major format

```
parallel_for j: N
// every thread gets a specific j to execute
   for i: M
      A(j,i)
```

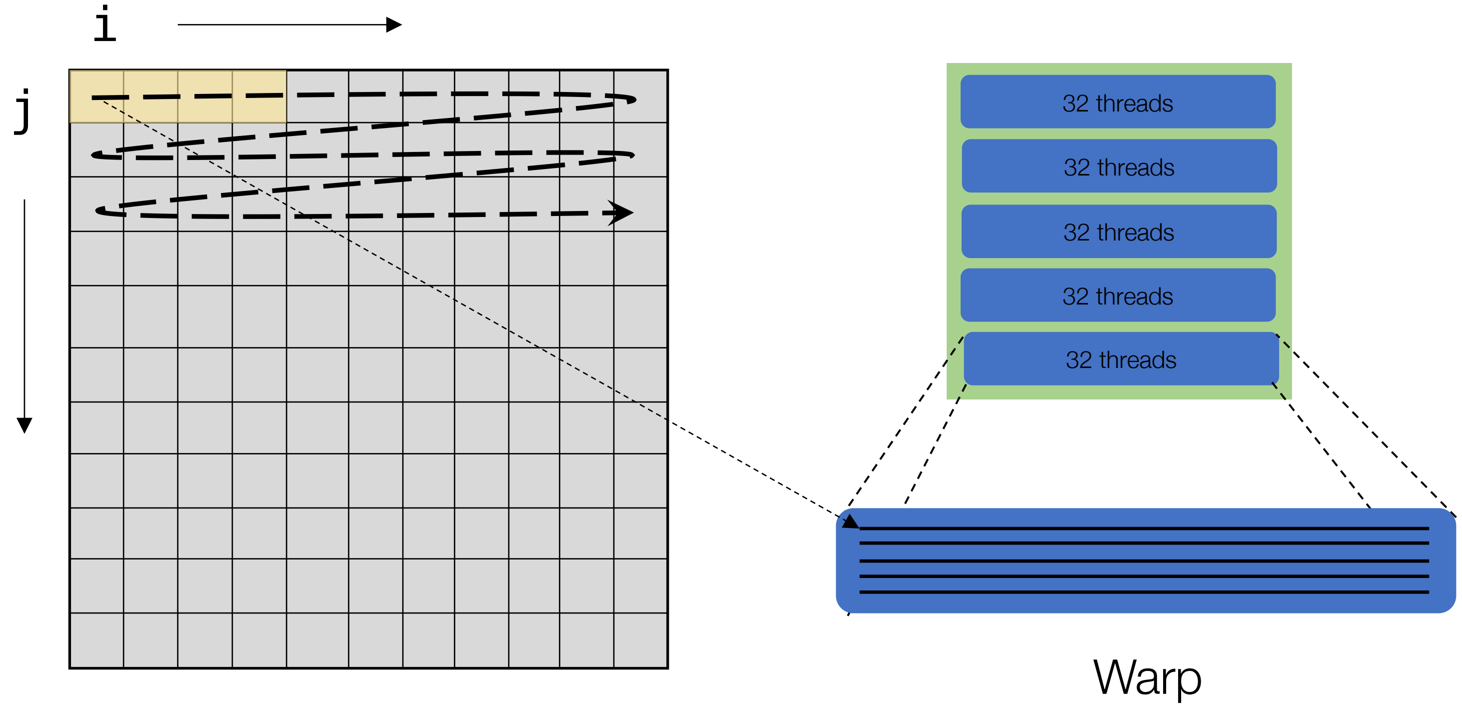
- GPU execution model





- Takeaway: Threads in a warp execute the same instruction

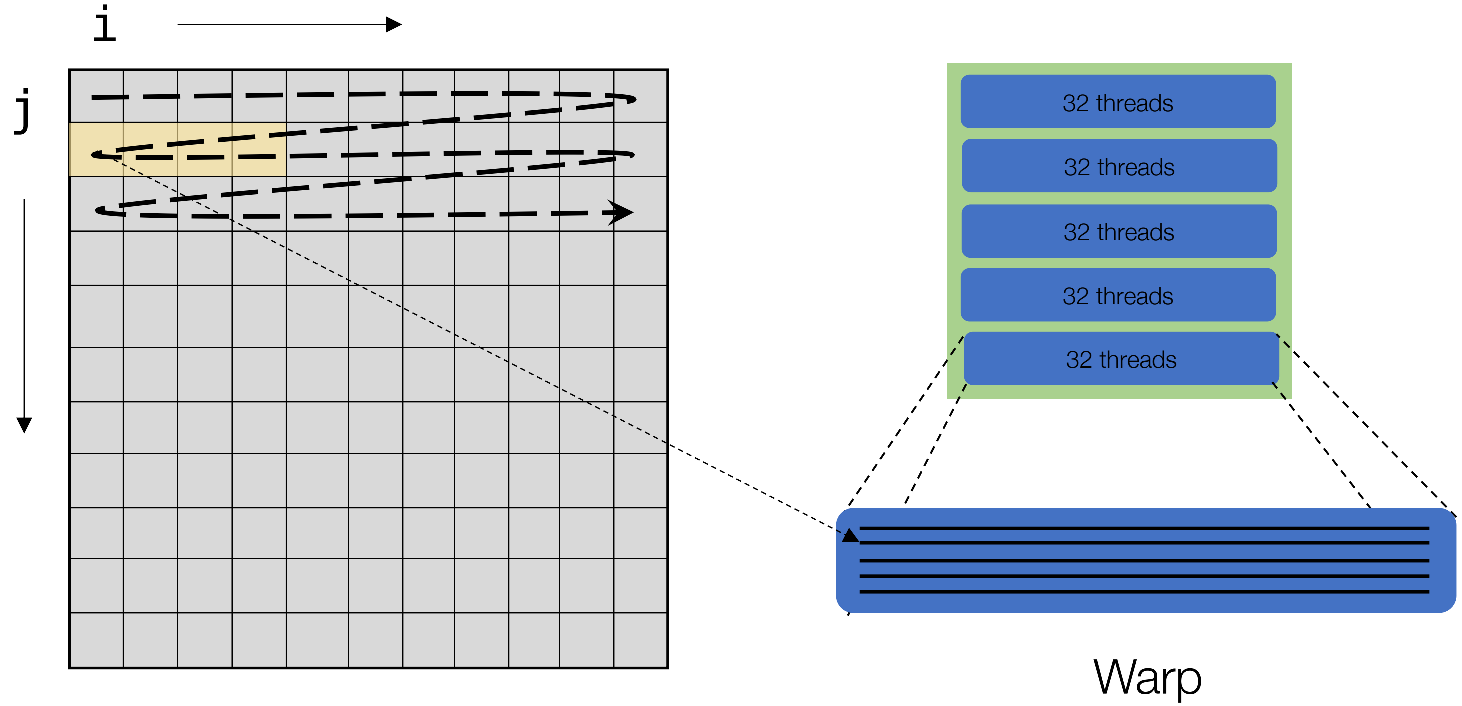
- Bad performance




```
parallel_for j: N
// every thread gets a specific j to execute
   for i: M
      A(j,i)
```

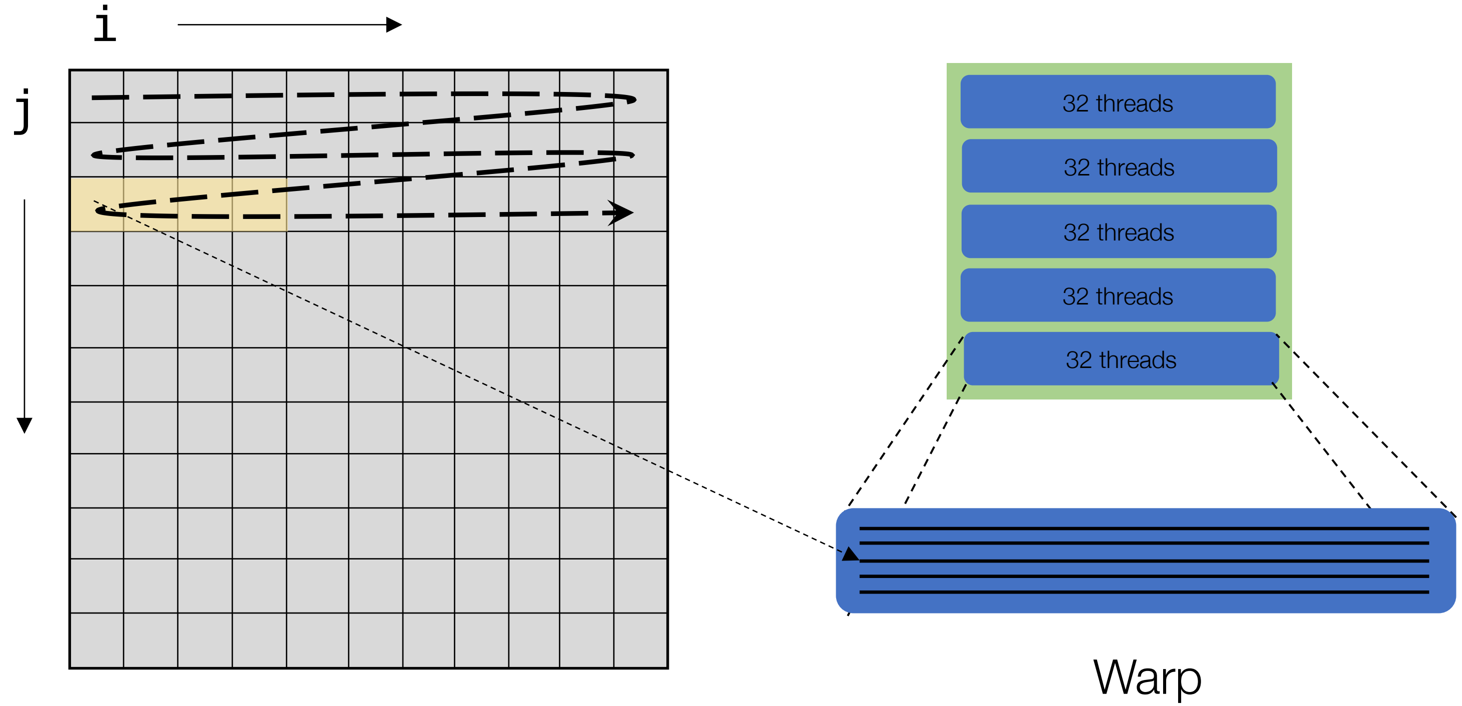
GPU performance

- Bad performance




```
parallel_for j: N
// every thread gets a specific j to execute
   for i: M
      A(j,i)
```

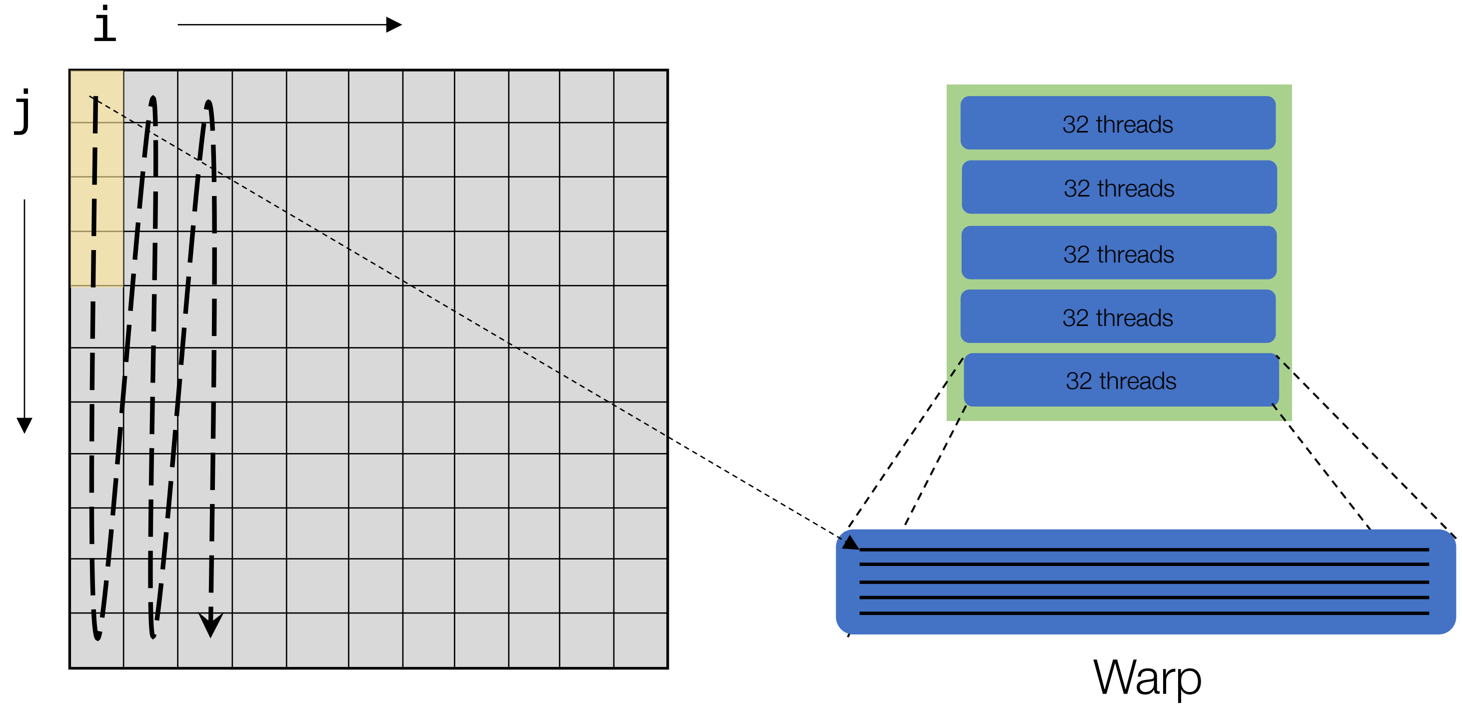
- Bad performance




```
parallel_for j: N
// every thread gets a specific j to execute
   for i: M
      A(j,i)
```

GPU performance

- Good performance



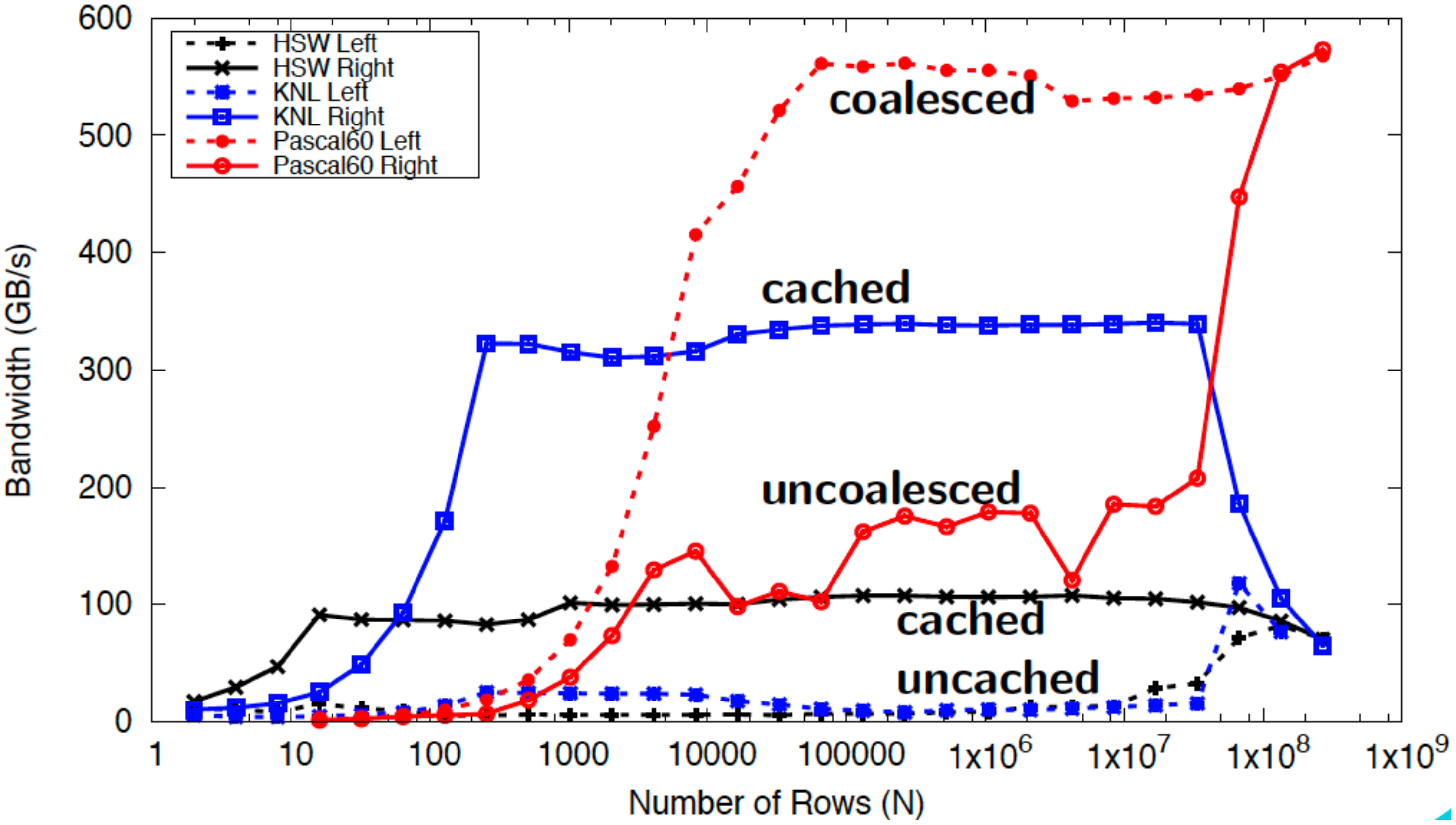

```
parallel_for j: N
// every thread gets a specific j to execute
   for i: M
      A(j,i)
```

- CPUs require row major format
  - $A(i_1,\ i_2\ ,\ \ldots,\ i_n) \rightarrow i_n$ needs to be continuous in memory (Layout Right)
  - Results in data caching

- GPUs require a column major format
  - $A(i_1,\ i_2\ ,\ \ldots,\ i_n) \rightarrow i_1$ needs to be continuous in memory (Layout Left)
  - Results in data coalescing

Performance benchmarks

- vector - matrix – vector multiplication
- Total number of elements in matrix is a constant (M*N = constant)
- Bandwidth ~ amount of data processed per second



Understanding performance

- CPUs require row major format
  - `A(`i_1`, `i_2` , …, `i_n`)` → i_n needs to be continuous in memory (Layout Right)
  - Results in data caching

- GPUs require a column major format
  - `A(`i_1`, `i_2` , …, `i_n`)` → i_1 needs to be continuous in memory (Layout Left)
  - Results in data coalescing

Data layout needs to be determined at compile time based on target architecture

Solution: Kokkos views

- Kokkos views are multidimensional arrays
- Can reside on CPUs or GPUs
  - Use “mirror” of your arrays to sync between CPUs and GPUs
  - deep_copy can be initialized between original and mirror views
- View layout can be specified at compile time
  - Default behavior is to optimized layout based on execution space
    - GPUs = Layout Left
    - CPUs = Layout Right
- Views are reference counted
  - Easy memory management
- Copy construction leads to both views pointing to the same data
  - deep copy requires use of “deep_copy” routine rather than a copy construction

Kokkos view

```
Kokkos::View<float **, Kokkos::LayoutLeft, Kokkos::CudaSpace>
A("A_matrix", N, M);
```

Policy Concept

Policy

Pattern

```
for (int element = 0; element < nelements; element++){
    total = 0;
    for (int qp = 0; qp < numQP; qp++){
        // reduction over every qp in element
        total += foo(variable, element, qp);
    }

    elementValue[element] = total;
}
```

Body

OpenMP – An analogy

Policy

Pattern

```
#pragma omp parallel for schedule(<static, dynamic>)
for (int element = 0; element < nelements; element++){
    total = 0;
    for (int qp = 0; qp < numQP; qp++){
        // reduction over every qp in element
        total += foo(variable, element, qp);
    }

    elementValue[element] = total;
}
```

Body

Kokkos - Example

Pattern

Policy

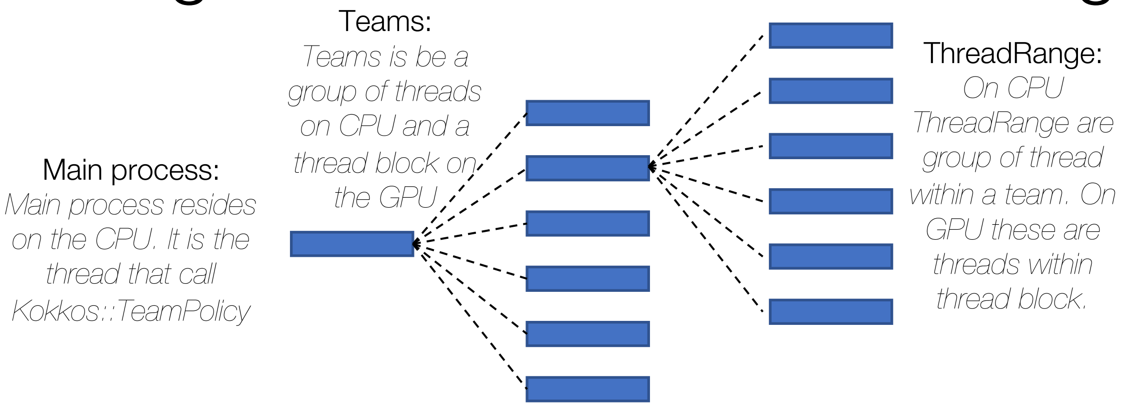
```
Kokkos::parallel_for("LoopName", Kokkos::RangePolicy(nelements),
      [=](const int element){
      total = 0;
      for (int qp = 0; qp < numQP; qp++){
            // reduction over every qp in element
            total += foo(variable, element, qp);
      }

      elementValue[element] = total;
});
```

Body

Example

Parallel policies

- `Kokkos::RangePolicy` – Each iterate is an integer in a contiguous range
- `Kokkos::MDRangePolicy` – Each iterate for each rank is an integer in a contiguous range
- `Kokkos::TeamPolicy` – Assigns to each iterate in a contiguous range a team of threads



Packaging Kokkos projects

Packaging Kokkos projects

- CMAKE - Recommended
  - Provide Kokkos as a dependency package

```
include(FetchContent)
FetchContent_Declare(
kokkos
URL
https://github.com/kokkos/kokkos/archive/refs/tags/3.6.01.zip
)
FetchContent_MakeAvailable(kokkos)

add_executable(question question.cpp)
target_link_libraries(question Kokkos::kokkos)
```

Packaging Kokkos projects

- CMAKE – Recommended
  - In house installations

```
set(KOKKOS_PATH <PATH TO KOKKOS ROOT>)

add_subdirectory(KOKKOS_PATH)
add_executable(question question.cpp)
target_link_libraries(question Kokkos::kokkos)
```

- Makefiles – maybe suitable for individual private projects
  - Check this example out: https://github.com/kokkos/kokkos/blob/master/benchmarks/bytes_and_flops/Makefile

Documentation Overview

Why use Kokkos?

	CUDA/HIP/DPC++	OpenMP 5/ OpenACC	Kokkos
Portability across architectures	No. Need to write separate kernels for every architecture	Yes. Single source code with pragma-based approach	Yes. Single source code implemented using Kokkos functions
Performance	Optimized performance	Tough to optimize for performance	Very good performance
Cost of portability	Very high	[illegible]	High
Cost of maintenance	Very high. Newer architectures might require tuning of kernels	Low. Assuming compilers do a good job of implementing the standard	Low. Assuming Kokkos backend is always optimized
Compiler dependence	N/A	These are standards, vendors have a flexibility on implementation	N/A
Fortran support	No. Could use bindings	Yes	No. Could use bindings

Why use Kokkos?

Summary

- Kokkos is a ***performance portability*** library
- Performance is dependent on data layout
  - Kokkos Views are easy way to manage data layout
- Compile time definition of data using C++ template meta-programming
- Low maintenance overhead for future architectures

Useful resources:

- Documentation
- Tutorials
- Slack (invite only) : ctrott [a] sandia.gov
- Local help - cses@princeton.edu

Thank you