PARALLEL PROGRAMMING...

By Patrick Lemoine 2025.

Kokkos

Κόκκος

SESSION 1

Kokkos?

Basic Concepts

- Views,
- Memory spaces,
- Memory access patterns,
- Mirrors

Advanced Concepts

- Asynchronicity, Streams and Task Parallelism
- Multi-level hierarchical parallelism
- Multidimensional Loops and Data Structures
- Advanced Reductions
- MPI (Message Passing Interface) and PGAS (Partitioned Global Address Spaces)
- SIM (Single Instruction Multiple Data) Task Flow

Introduction to Kokkos

Kokkos is a C++ library designed for parallel programming, allowing developers to write performance-portable code across various hardware architectures, including CPUs, GPUs, and multi-core processors.

Developed at Sandia National Laboratories (SNL) in the United States

An open source project with a growing community

The main goal of Kokkos is to facilitate the development of high-performance applications that can efficiently utilize the capabilities of various computing resources.

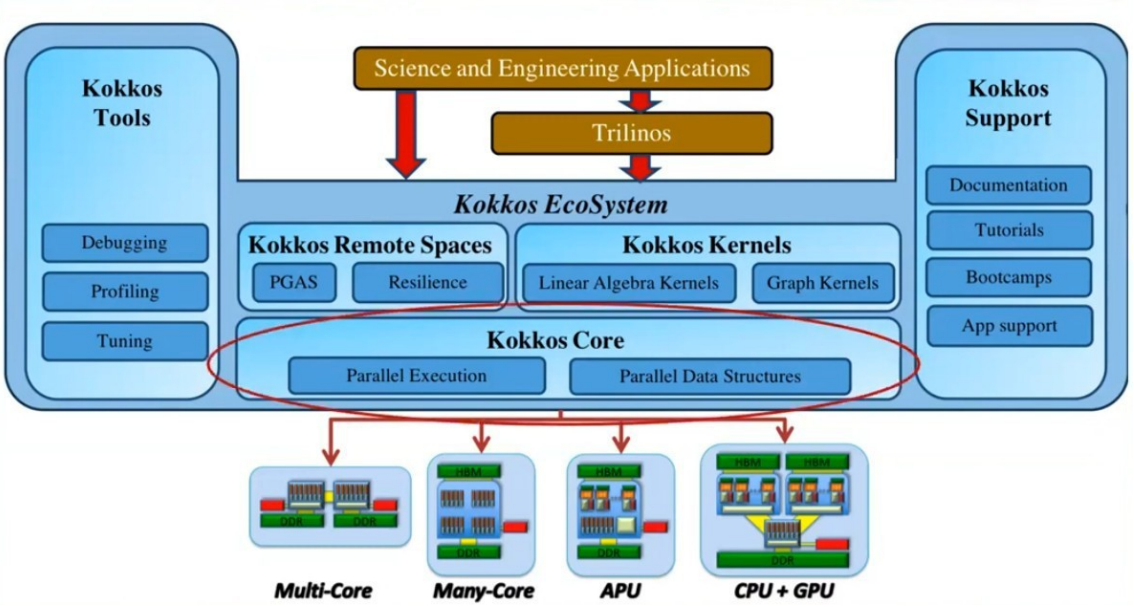
Kokkos-Core Github Repository:https://github.com/kokkos/kokkos

Kokkos Ecosystem

- **Kokkos Core:**
  - Programming model for parallel execution.
  - Efficient data management across different architectures.

- **Kokkos Kernels:**
  - Optimized math library.
  - Provides basic routines for increased performance.

- **Kokkos Tools:**
  - Profiling tools to analyze application performance.
  - Debugging tools to facilitate development.



- **Portability:**
  - Enables development of HPC applications on various architectures without rewriting code.

- **Performance :**
  - Optimized to maximize efficiency on a variety of systems, from CPUs to GPUs.

This ecosystem is designed to simplify the development of scientific applications while ensuring high performance.

Why use Kokkos?

Benefits

1. Unified code base: write once, run on CPU, GPU or other accelerators without rewriting.
2. High performance: optimizes resource usage (e.g. multi-core CPU, GPU).
3. Scalability: Supports single-node and multi-node systems with complex memory hierarchies.

Adoption

- Used by hundreds of HPC developers worldwide

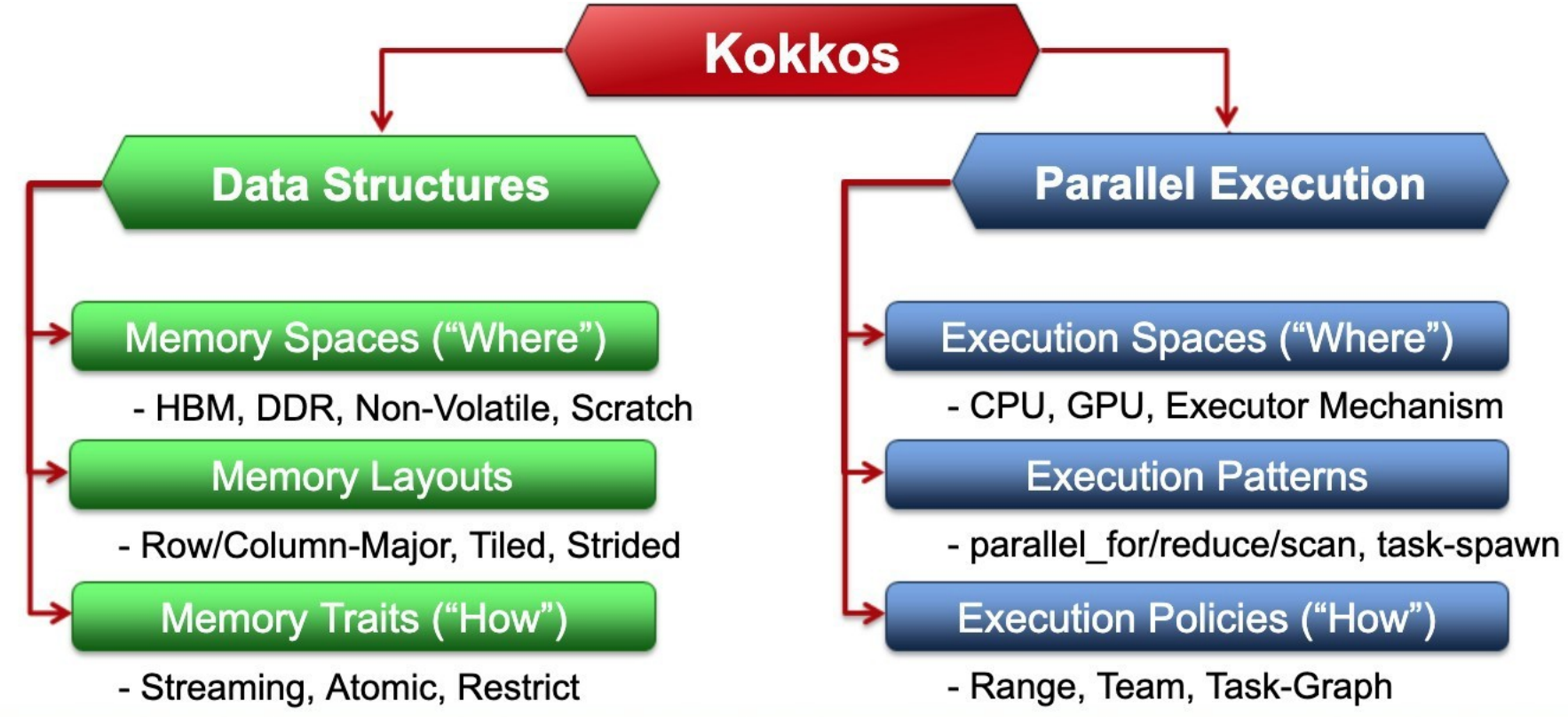
Examples of supported backends

- CUDA (NVIDIA graphics processor)
- HIP (AMD graphics processor)
- OpenMP (multi-core processor)
- SYCL (GPU and Intel accelerators)

Key components

- Kokkos Core: Basic programming model
- Kokkos Kernels: portable math library
- Kokkos Tools: Debugging and performance analysis tools

Abstractions of the Kokkos kernel



Kokkos provides a powerful abstraction for the development of parallel algorithms, enabling performance portability across diverse computing architectures through its execution and data management models.

Fundamental Concepts of Kokkos

Memory spaces: Determine where and how data is allocated in memory

- Main options:
  - Kokkos::CudaSpace (NVIDIA GPU memory)
  - Kokkos::HostSpace (CPU memory)
  - Kokkos::CudaUVMSpace (CPU-GPU unified memory)
  - Kokkos::CudaHostPinnedSpace (pinned CPU memory)
  - Kokkos::DefaultMemorySpace (default memory space)

HostMemory:: standard memory accessible by the CPU. Used for data that does not require high-speed access.

DeviceMemory:memory allocated on GPUs. Provide faster access to calculations that take advantage of GPU capabilities.

PinnedMemory:a special type of host memory that allows faster data transfer between the host and device. Improved performance in data-intensive applications.

Execution spaces: Define where parallel builds are executed

- Main options:
  - Kokkos::Cuda (NVIDIA GPU)
  - Kokkos::HIP (AMD graphics processor)
  - Kokkos::OpenMP (multi-core processor)
  - Kokkos::Threads (multithreaded processor)
  - Kokkos::Serial (single-threaded CPU)
- Kokkos::DefaultExecutionSpace (default execution space))

Parallel models: Parallel models are high-level abstractions that simplify the implementation of parallel algorithms in Kokkos.

ParallelFor: a model that allows a loop to be executed in parallel. Distributes iterations across available threads.

ParallelReduThis: combines the results of multiple threads into a single output, useful for operations such as adding an array.

ParallelScan: computes prefix sums in parallel, allowing efficient data processing in algorithms that require cumulative results.

Kokkos “View” Concept

Multidimensional array with management of compilation provisions and memory space.

- Type of entries
- Number of dimensions
- Arrangement (**LayoutLeft, LayoutRight**)

```
Kokkos::View<double, Kokkos::LayoutRight> cpuView("cpuData", N, M);
Kokkos::View<double, Kokkos::LayoutLeft> gpuView("gpuData", N, M);
```

Views: Work like pointers, but carry much more information:

- DataType: type of data contained.
- LayoutType: layout of data in memory.
- MemorySpace: data storage location.
- MemoryTraits: data access characteristics.

- Only the data type must be specified.

```
// Allocate x, y vectors and Matix A on device using
vector = kokkos::view<double *>; using matrix =
kokkos::view<double **>;

vector y(“y”,N);
vector x(“x”,M);
matrix A(“A”,N,M);

// Initialize y vector For (int i=0; i<N; ++i) {
    y(i) = 1;//can potentially lead to illegal access
if DefaultSpace is not accessible by the CPU!!!
    }
```

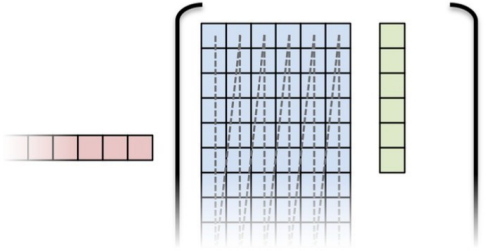
```
View<double***> data("label", N, M, K);//3 at runtime, 0 at compile time

View<double**[K]> data("label", N, M);//2 at runtime, 1 at compile time

View<double[N][M][K]> data("label");//3 at compilation
```

LayoutLeft, LayoutRight?

The **adaptable memory layouts** determine how data is organized in memory in Kokkos, thereby influencing memory access efficiency and overall application performance.

LayoutLeft

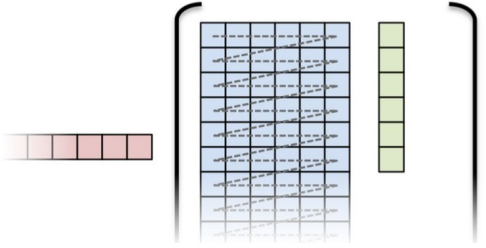
in 2D, "column-major"



LayoutLeft: Ideal for GPU architectures where memory access coalescing is crucial, as this allows for faster data access.

LayoutRight

in 2D, "row-major"



LayoutRight: Preferred for traditional processors because it allows better cache utilization and increased performance when accessing data by rows.

The choice between **LayoutLeft** And **LayoutRight** is essential to maximize access efficiency memory and adapt the code to the different hardware architectures used.

Kokkos Concept Mirrors

Mirror:These are views of equivalent tables that may reside in different memory spaces.

View Builder: It allows to create a view in a specific memory space.

Main functions:

- **create_mirror_view**:Creates a mirror view in the host's memory space.
- **deep_copy**:Copies data between compatible views.
- **create_mirror_view_and_copy**:Creates a mirror view and copies the data at the same time.

Syntax:**Kokkos::View<type, MemorySpace> data("label", dimensions);**

```
// create host mirrors of device views vector::HostMirror
h_y
= Kokkos::create_mirror_view(y); vector::HostMirror h_x =
Kokkos::create_mirror_view(x); matrix::HostMirror h_A =
Kokkos::create_mirror_view(A);

// initialize y vector on host
for (int i=0; i<N; ++i) { h_y(i) = 1;} // initialize x vector on
host
for (int i=0; i<M; ++i) { h_x(i) = 1;}

// initialize A matrix on host
for (int j=0; j<N; ++j) {
   for (int i=0; i<M; ++i) {
      h_A(j, i) = 1;
   }
}

// Deep copy views to device views

Kokkos::deep_copy(y, h_y); Kokkos::deep_copy(x, h_x);
Kokkos::deep_copy(A, h_A);
```

Kokkos Memory Spaces

Representing the “where” and “how” of memory allocation and access in Kokkos, this concept is fundamental to ensuring performance portability.

Kokkos automatically adapts these models according to the target hardware.

Common memory spaces-Kokkos::HostSpace: Traditional CPU memory - Kokkos::CudaSpace: GPU memory for NVIDIA - Kokkos::CudaUVMSpace: Unified CPU-GPU memory...

Key Features

- **Portability**:Optimization of memory accesses on CPU and GPU.
- **Flexibility**:Adaptive memory layouts, such as LayoutLeft and LayoutRight.
- **Performances**:Contiguous access on CPU and coalescing on GPU.

Implementation

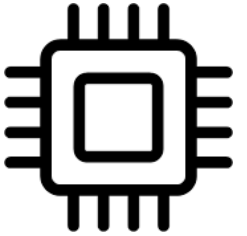
- **CPU**(eg: OpenMP): Use LayoutRight by default to optimize cache usage.
- **GPU**(eg: CUDA): Use LayoutLeft by default to promote memory merging.

-**Layout polymorphism**:Ability to change the schema without modifying the function code. This allows developers to easily adapt data structures to different hardware architectures (such as CPUs and GPUs) while maintaining a consistent interface.

Kokkos: *Managing memory access patterns and impact on performance*

Importance of memory access patterns

- Essential for optimizing performance on CPU (cache) and GPU (coalescing).
- Kokkos automatically adjusts these models according to the target architecture.



Memory arrangements in Kokkos

- LayoutLeft: Main column order, ideal for GPU.
- LayoutRight: Order of main lines, optimal for the processor.
- Flexibility: Customizable layouts according to needs.

Optimization strategies

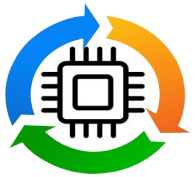
-**CPU**:Optimized cache and memory usage**SIMD**. A technique that allows a single instruction to be executed on multiple data elements simultaneously. This contrasts with scalar operations which process one element at a time.

- **GPU**:Ensure memory fusion for efficient access.

Impact on performance

- Efficient memory access of up to **10 times more efficient**.
- Portability ensured thanks to a single code, allowing optimal performance on various architectures.

Kokkos Execution Spaces



Representing the “where” and “how” of parallel execution in Kokkos, this abstraction integrates available computing resources such as CPUs and GPUs.

Execution spaces

Execution models
- **Simple loops**:Using parallel_for .
- **Discounts**:Using parallel_reduce`.
- **Scans**:Using parallel_scan .

Execution Policy:are fundamental abstractions that define how parallel tasks are executed.
- **Range policies**:Sets of indexes on which operations are performed.
- **Team policies**:Grouping threads into teams, forming a subset of the execution space to enable hierarchical parallelism.

```
Kokkos::parallel_for( Kokkos::RangePolicy<Kokkos::DefaultExecutionSpace>(
        0,   10),    KOKKOS_LAMBDA(int n) { /* ... */ }
) ;

Kokkos::TeamPolicy<ExecutionSpace>  team_policy(league_size,  team_size);
```

Memory layouts: LayoutRight` vs LayoutLeft , with automatic conversion between the two depending on runspaces.

Memory traits
- **Types of access**:Atomic access, random access (shader memory), and streaming storage.

Thank you for your attention !