New methods in computational mechanics and physics

High performance computing using Kokkos and
its application to N-Body simulations

Arnaud Remi
Victor Mangeleer
University of Liège 2022 – 2023

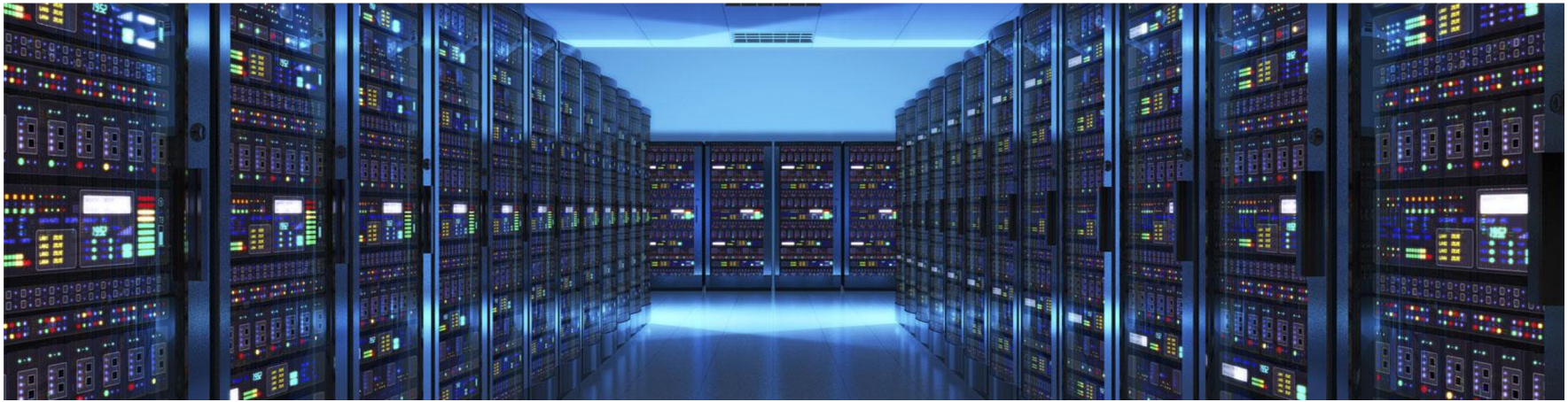
OVERVIEW

1. Introduction
2. Kokkos for parallelization
3. Memory spaces
4. Polymorphic memory access
5. Kokkos graphs
6. N-Body simulations

INTRODUCTION (1/5)

What is HPC ?

High-Performance Computing, refers to the utilization of **advanced computing systems and techniques** to tackle intricate problems and execute computationally intensive tasks with remarkable **speed** and **efficiency**.



@ vertiv

INTRODUCTION (2/5)

How to do HPC ?

High-Performance Computing can be implemented through various approaches, including **multithreading** on a **CPU**, **multi-core** on a **CPU**, or utilizing a **GPU**.

CPU

Threads

Multithreading on a CPU involves **running multiple threads concurrently** on a **single processing unit**, allowing for parallel processing.

Cores

Multi-core CPUs have **multiple independent processing units** within a single chip, enabling parallel execution of tasks across different cores. **Each of these cores** can also perform **multithreading**.

GPU

GPUs, or Graphics Processing Units, **have a highly parallel architecture** that can handle massive amounts of data simultaneously

INTRODUCTION (3/5)

Pros

High-Performance Computing can be implemented through various approaches, including **multithreading** on a **CPU**, **multi-core** on a **CPU**, or utilizing a **GPU**.

CPU

Threads

- **Efficient utilization of CPU** resources by running multiple threads concurrently.

- Enables parallel processing, **enhancing performance** for certain types of tasks.

Cores

- **Efficient utilization of CPU** resources, enabling better workload distribution.

- Provides **multiple independent processing units within a single CPU chip**, allowing for parallel execution of tasks.

GPU

- **Highly parallel architecture**, capable of **processing large amounts of data** simultaneously.

- Well-suited for tasks that can be parallelized, such as graphics rendering and **scientific simulations.**

- Can achieve **significant higher speedup compared to CPUs** for certain types of computations.

INTRODUCTION (4/5)

Cons

High-Performance Computing can be implemented through various approaches, including **multithreading** on a **CPU**, **multi-core** on a **CPU**, or utilizing a **GPU**.

CPU

Threads

- **Limited number** of available CPU **threads**.

- **Increased complexity** in managing shared resources and ensuring thread safety.

- **May face synchronization overhead** when multiple threads access shared data.

Cores

- **Limited** by the **number of cores** available on the CPU.

- **May require software optimization** to fully exploit the potential parallelism.

- **Increased power consumption** and **heat generation** due to multiple cores operating simultaneously.

GPU

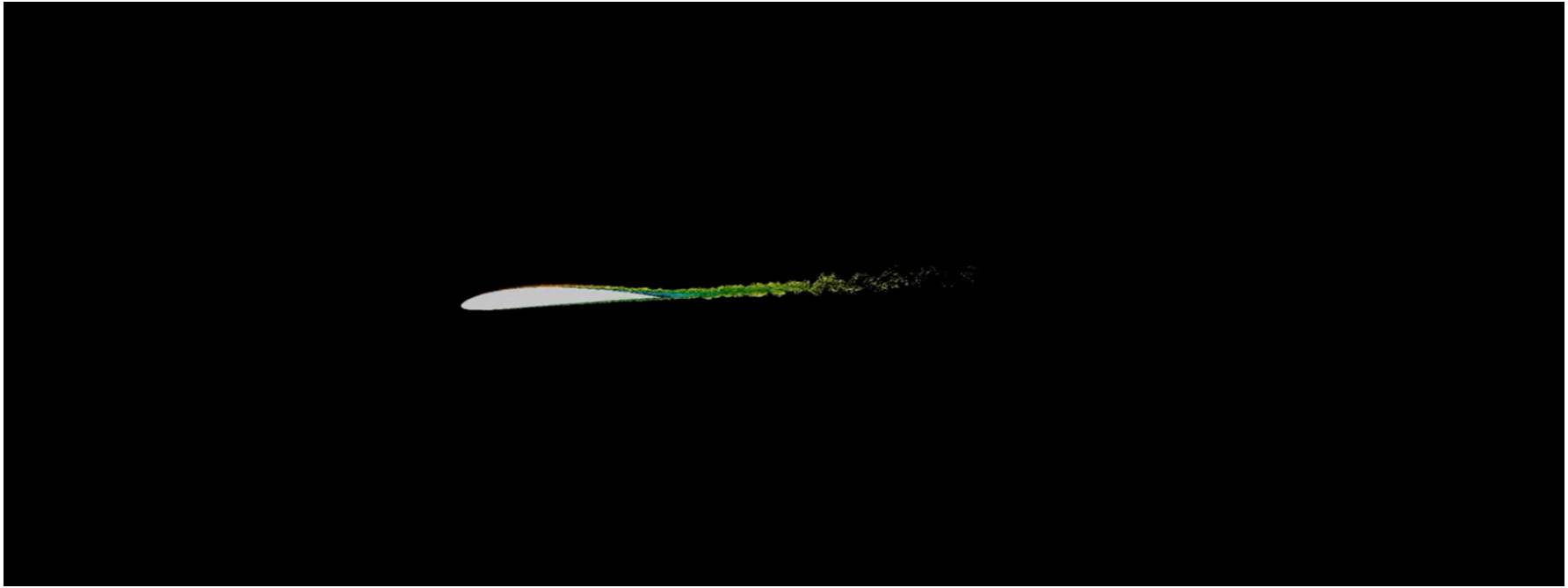
- **Not all applications** can effectively **harness the parallelism** offered by GPUs.

- **Requires specialized programming** techniques (e.g., using CUDA or OpenCL) to utilize the GPU's full potential.

- **Additional data transfer overhead between the CPU and GPU** can impact performance in certain scenarios.

INTRODUCTION (5/5)

Illustrations



@ Linné FLOW center

KOKKOS FOR PARALLELIZATION (1/6)

Context

Imagine spending years **developing a high-performance** code on your **NVIDIA** machine, **then giving it** to your friend which uses an **AMD** graphics card.

Then, you realize that the **performance** of your code is **10x worse** than on your personal computer….

What can you do ?

KOKKOS FOR PARALLELIZATION (2/6)

Introduction

Kokkos is a **programming library in C++** that allows developers to write **performance portable code** for parallel computing **across different architectures**, including CPUs and GPUs.

PROS

- **Performance Portability**: Enables writing code that can efficiently run on various architectures, including CPUs and GPUs.

- **Abstraction Layer**: Abstracts away hardware-specific details, simplifying the development process.

- **Unified Programming Interface**: Provides a consistent API for parallel computing, reducing the need for architecture-specific coding.

- **Productivity and Code Reusability**: Allows developers to write code once and execute it on multiple platforms, saving time and effort.

- **Ecosystem and Community**: Benefits from an active community and ecosystem with ongoing development, support, and documentation.

CONS

- **Learning Curve**: Requires understanding the Kokkos programming model and API, which may have a learning curve for developers new to the library.

- **Overhead**: The abstraction layer may introduce some overhead compared to writing highly optimized, architecture-specific code directly.

- **Advanced Features**: Certain advanced or specialized features may have limited or less mature support within the Kokkos library.

KOKKOS FOR PARALLELIZATION (3/6)

Basics - Kernels

A **kernel** refers to a function or a code block that represents a **computational task** to be **executed in parallel**. It is designed to **operate** on a specific **subset of data elements**, such as an array or a range of values. Kernels are the building blocks of parallel computations and **can be executed concurrently by multiple threads or processing units**.

```
// Computing forces
for (int i = 0; i < n_bodies; i++)
{
    // Piece of code indexed by i
}
```

KOKKOS FOR PARALLELIZATION (4/6)

Basics - Pattern

Patterns are predefined parallel algorithms or operations that **simplify the development of parallel computations**. They provide a **high-level of abstraction**, allowing developers to express computations without dealing with low-level details. Patterns like ***parallel for*** automate parallel execution, making code **concise** and **readable** while **handling thread creation and data distribution behind the scenes**.

```
// Computing forces
for (int i = 0; i < n_bodies; i++)
{
    // Piece of code indexed by i
}
```

KOKKOS FOR PARALLELIZATION (5/6)

Basics - Policy

A **policy** specifies **how a computation should be executed** in terms of parallelism and data layout. It defines the **characteristics of parallel execution**, such as the number of threads or processing units to use, and how the data should be partitioned.

```
// Computing forces
for (int i = 0; i < n_bodies; i++)
{
    // Piece of code indexed by i
}
```

KOKKOS FOR PARALLELIZATION (6/6)

Basics - Body

The **body** refers to the function or **functor** that **contains the computational logic** for each iteration of a parallel operation. It **represents the code that will be executed in parallel** by multiple threads or processing units. The **body** is typically defined as a **lambda function** or a **functor object** and is invoked for each element or iteration within the parallel execution range.

```
// Computing forces
for (int i = 0; i < n_bodies; i++)
{
    // Piece of code indexed by i
}
```

MEMORY SPACES (1/3)

Introduction

- **Kernels** are executed in an **execution space**
  - → Serial `typedef Kokkos::Serial ExecSpace;`
  - → OpenMP `typedef Kokkos::OpenMP ExecSpace;`
  - → Cuda `typedef Kokkos::Cuda ExecSpace;`

- **Data** reside in a certain **memory space**
  - → HostSpace `typedef Kokkos::HostSpace MemSpace;`
  - → CudaSpace `typedef Kokkos::CudaSpace Memspace;`

- There is **a need to manage** where **data** are stored
  - → Kokkos **View**

MEMORY SPACES (2/3)

Kokkos view

Kokkos **View** $X = (\{x_t\}, X_{\mathrm{S}}, X_{\mathrm{L}})$

- $\{x_t\}$: set of data
- X_{S} : index space
- X_{L} : layout (mapping) between $\{x_t\}$ and X_{S}

In **code**, one obtains:

```
typedef Kokkos::View<double**, Layout, MemSpace>  ViewPositions;

ViewPositions  x("x", dim1, dim2);
ViewPositions  y("y", dim1, dim2);
```

MEMORY SPACES (3/3)

Transferring data

Need to transfer informations between **CPU** and **GPU** execution spaces ?

- **Mirror** view

```
ViewPositions::HostMirror h_x = Kokkos::create_mirror_view( x );
ViewPositions::HostMirror h_y = Kokkos::create_mirror_view( y );
```

→ `h_x` and `h_y` can be filled with data read by the CPU

- **Transferring** data between the different execution spaces

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

POLYMORPHIC MEMORY ACCESS (1/3)

Introduction

- The optimal way to map data and indices (layout) is **device-dependent**.

- Kokkos allows to choose the layout of any view:

  → **Polymorphic** data layout !

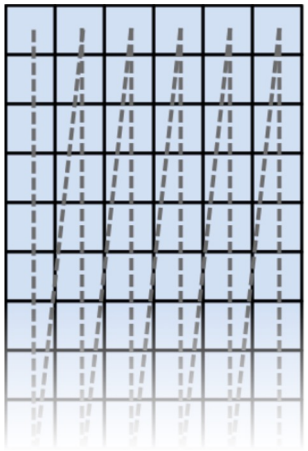
- Most common layouts:

  → **LayoutLeft** and **LayoutRight**

POLYMORPHIC MEMORY ACCESS (2/3)

Layouts

- Layout left (**column** major)



@ Kokkos Tutorials

→ Suited for **CudaSpace**

- Layout right (**row** major)

@ Kokkos Tutorials

→ Suited for **HostSpace**

POLYMORPHIC MEMORY ACCESS (3/3)

Layouts

- An **illustration** of the use of a layout:

```
// Define type for specific layout
typedef Kokkos::LayoutLeft Layout;
typedef Kokkos::View<double**, Layout, MemSpace>  ViewPositions;
ViewPositions x( "x", dim1, dim2);
ViewPositions y( "y", dim1, dim2);
```

- **The default** layouts if not specified are:

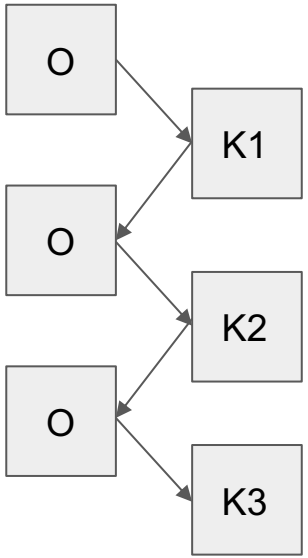
  → LayoutRight for **HostSpace**

  → LayoutLeft for **CudaSpace**

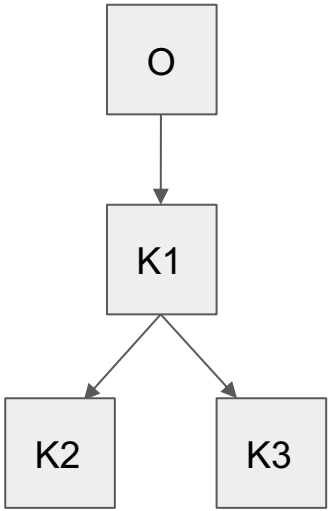
KOKKOS GRAPHS (1/3)

What if **multiple** kernels have to be executed?

Without graph, kernels are executed one after the others



Creating a graph allows efficient path



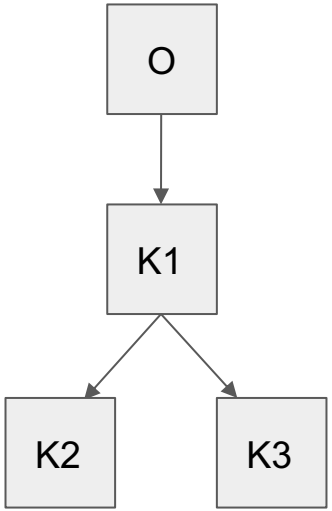
KOKKOS GRAPHS (2/3)

Advantages

- Do not come back to root between each kernel:

  → **Reduce overhead**

- If only one kernel K1 is prerequisite, others can be executed **simultaneously** after K1 !



KOKKOS GRAPHS (3/3)

Example

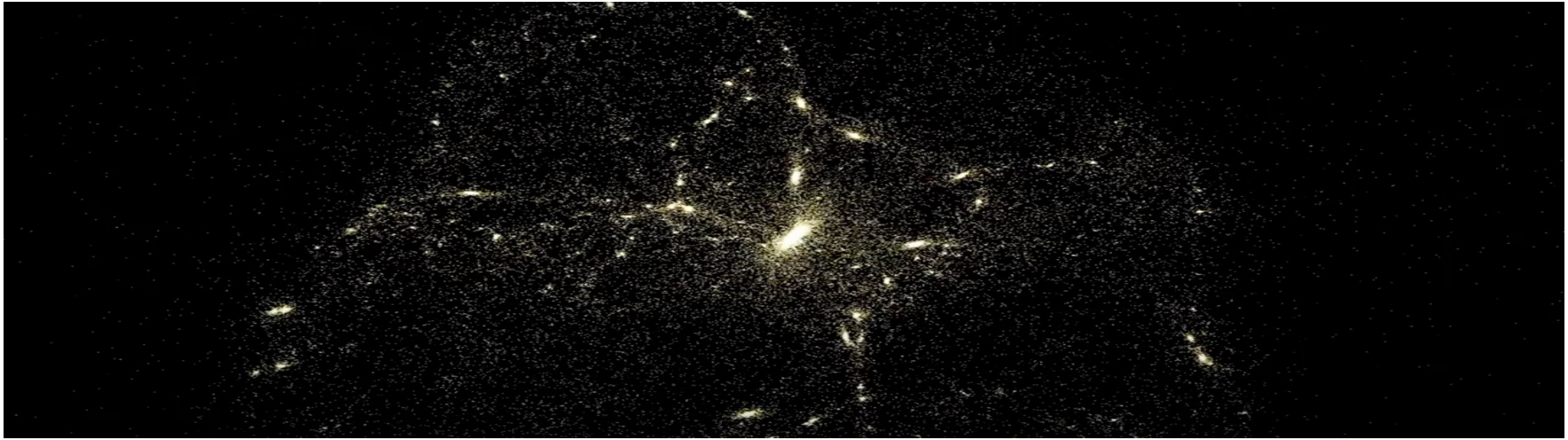
```
auto loop_K1 = KOKKOS_LAMBDA (int i)
    {
        // body code indexed by i
    };

auto loop_K2 = KOKKOS_LAMBDA (int i)
    {
        // body code indexed by i
    };

auto loop_K3 = KOKKOS_LAMBDA (int i)
    {
        // body code indexed by i
    };
```

```
// Kokkos graph
auto graph = Kokkos::Experimental::create_graph(
    [=](auto root){
        auto K1 = root.then_parallel_for("K1", N, loop_K1);
        auto K2 =   K1.then_parallel_for("K2", N, loop_K2);
        auto K3 =   K1.then_parallel_for("K3", N, loop_K3);
    }
);

// Graph execution
graph.submit();
```

N-BODY SIMULATIONS (1/9)

Context

A **n-body simulation** is a **computational model** used to **simulate the interactions and movements of multiple bodies or particles**. It finds applications in *astrophysics*, *molecular dynamics*, and computer graphics to study complex systems and their behavior.

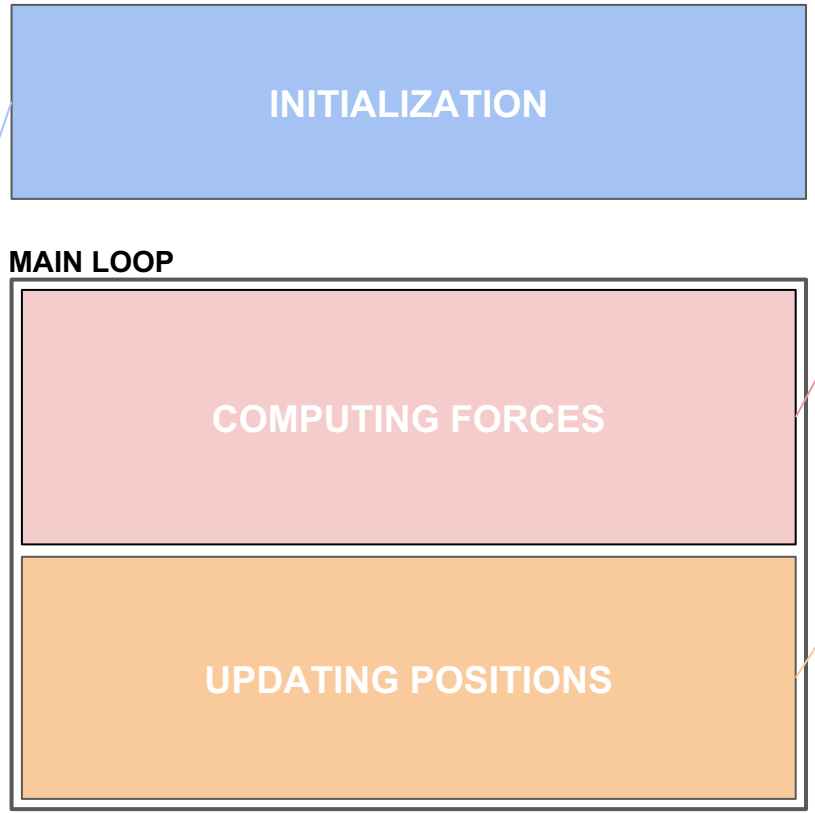


@ a2flow

N-BODY SIMULATIONS (2/9)

Structure

- **Loading** the data
- **Reading** the data
- **Allocating** memory
- …



Looping over all the bodies to **compute the gravitational force**.

Updating the position of each particle **knowing** the **total force** applied on them.

N-BODY SIMULATIONS (3/9)

Transformation

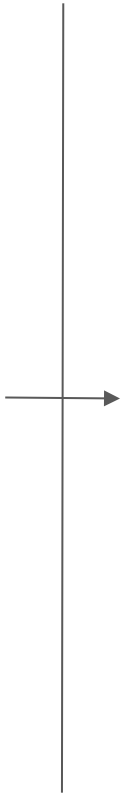
```
// Allocation (1)
Body *bodies = new Body[n_bodies];
double **x   = new double *[n_bodies];
double **y   = new double *[n_bodies];
double **vx  = new double *[n_bodies];
double **vy  = new double *[n_bodies];

// Allocation (2)
for (int i = 0; i < n_bodies; i++)
{
    x[i]  = new double[nb_iterations];
    y[i]  = new double[nb_iterations];
    vx[i] = new double[nb_iterations];
    vy[i] = new double[nb_iterations];
}
```

```
// Used to store bodies, x and y positions
typedef Kokkos::View<Body*,             MemSpace>     ViewBodies;
typedef Kokkos::View<double**, Layout, MemSpace>  ViewPositions;
typedef Kokkos::View<double**, Layout, MemSpace>      ViewSpeed;

// Allocation
ViewBodies    Bodies("Bodies", n_bodies);
ViewPositions      x( "x",     n_bodies, nb_iterations);
ViewPositions      y( "y",     n_bodies, nb_iterations);
ViewSpeed         vx("vx",     n_bodies, nb_iterations);
ViewSpeed         vy("vy",     n_bodies, nb_iterations);
```

N-BODY SIMULATIONS (4/9)

Transformation

```
//--------------------------
//       Simulation
//--------------------------
for (int t = 1; t < nb_iterations; t++)
{
       // Computing forces
      for (int i = 0; i < n_bodies; i++)
      {
            // Piece of code indexed by i
      }

      // Updating positions and velocities
      for (int i = 0; i < n_bodies; i++)
      {
            // Piece of code indexed by i
      }
}
```



```
//--------------------------
//       Simulation
//--------------------------
for(int t = 1; t < nb_iterations ; t++)
{
      Kokkos::parallel_for("Bodies - Force Loop",
      range_policy(0, n_bodies), KOKKOS_LAMBDA (int i)
      {
            // Piece of code indexed by i
      }


      Kokkos::parallel_for("Bodies - Update Loop",
      range_policy(0, n_bodies), KOKKOS_LAMBDA (int i)
      {
            // Piece of code indexed by i
      }
}
```

N-BODY SIMULATIONS (5/9)

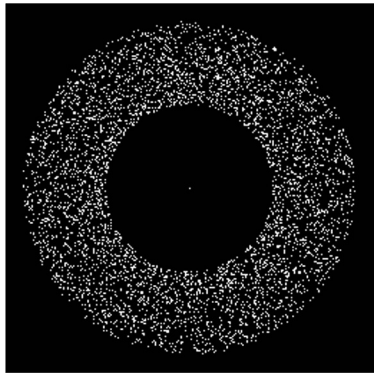
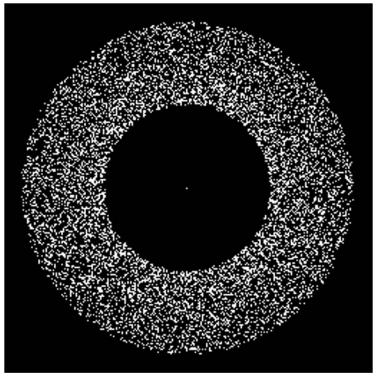
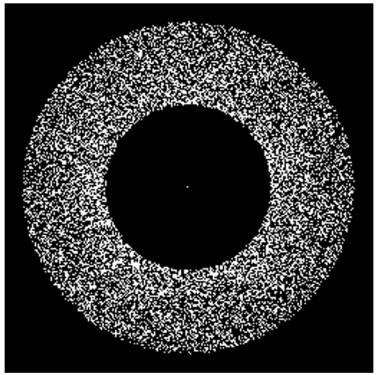
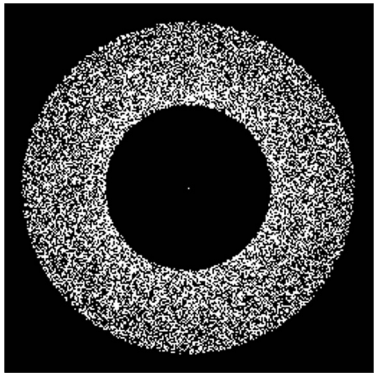
Metrics

- The **bandwidth** [GB/s] represents the rate of transferred data.
- The **speed up** [-] represents the ratio between a reference time (sequential here) and the new execution time (improved by parallelization for example)

N-BODY SIMULATIONS (5/9)

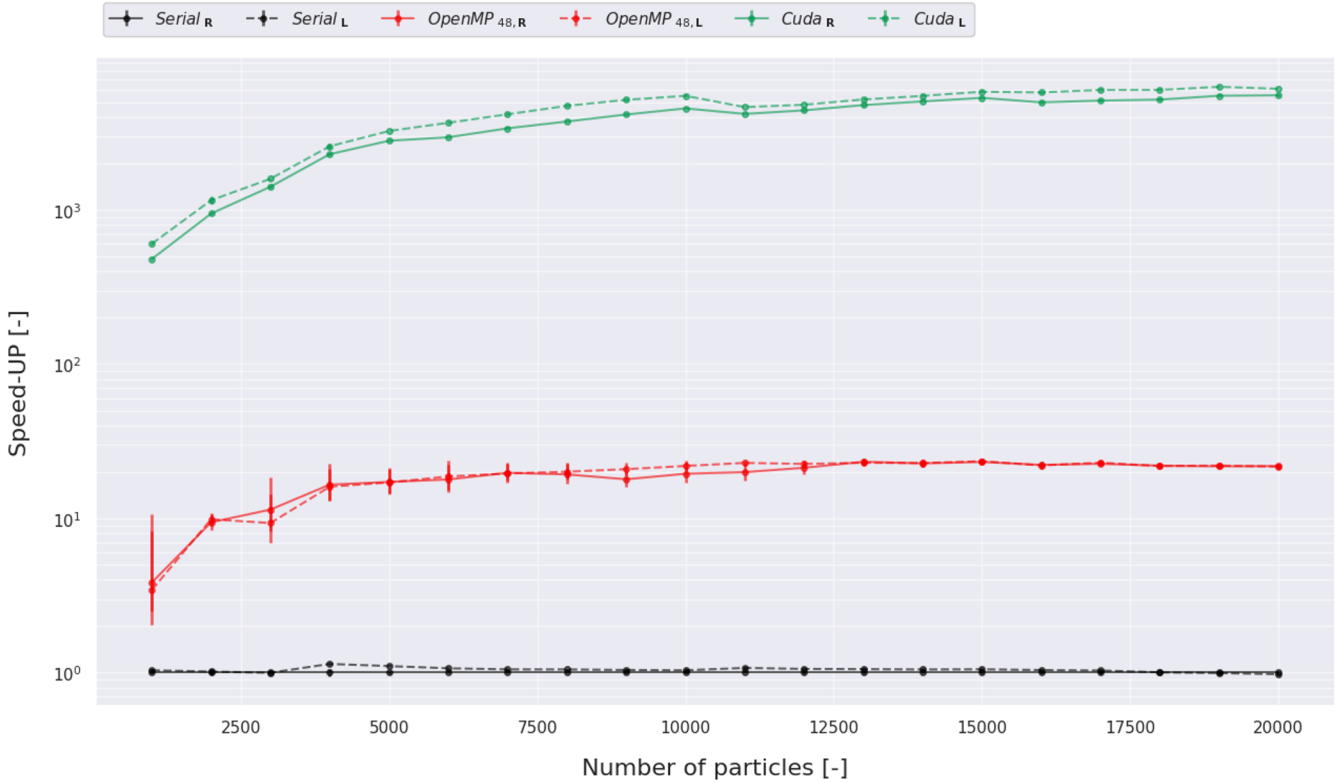
Metrics

- The **bandwidth** [GB/s] represents the rate of transferred data.
- The **speed up** [-] represents the ratio between a reference time (sequential here) and the new execution time (improved by parallelization for example)

#bodies	5000	10000	15000	20000
				

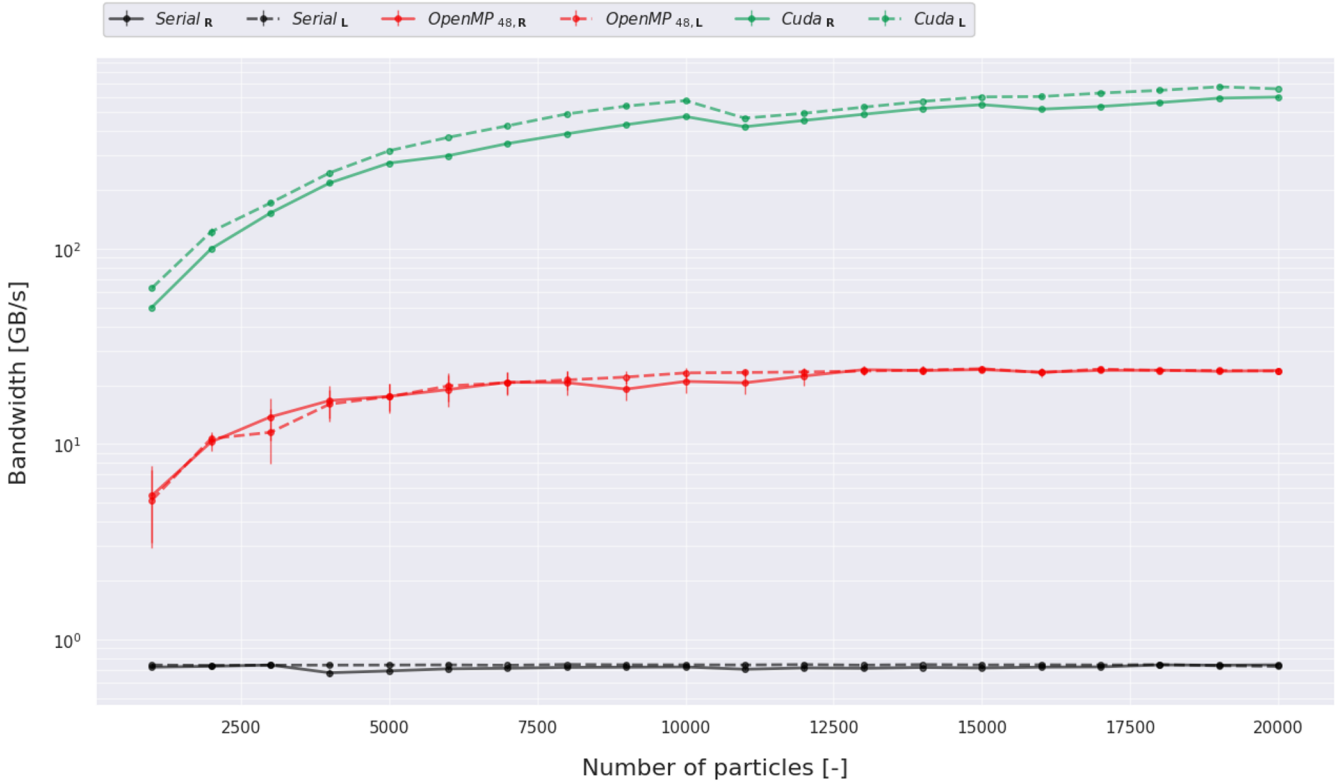
N-BODY SIMULATIONS (6/9)

Results (1/4)



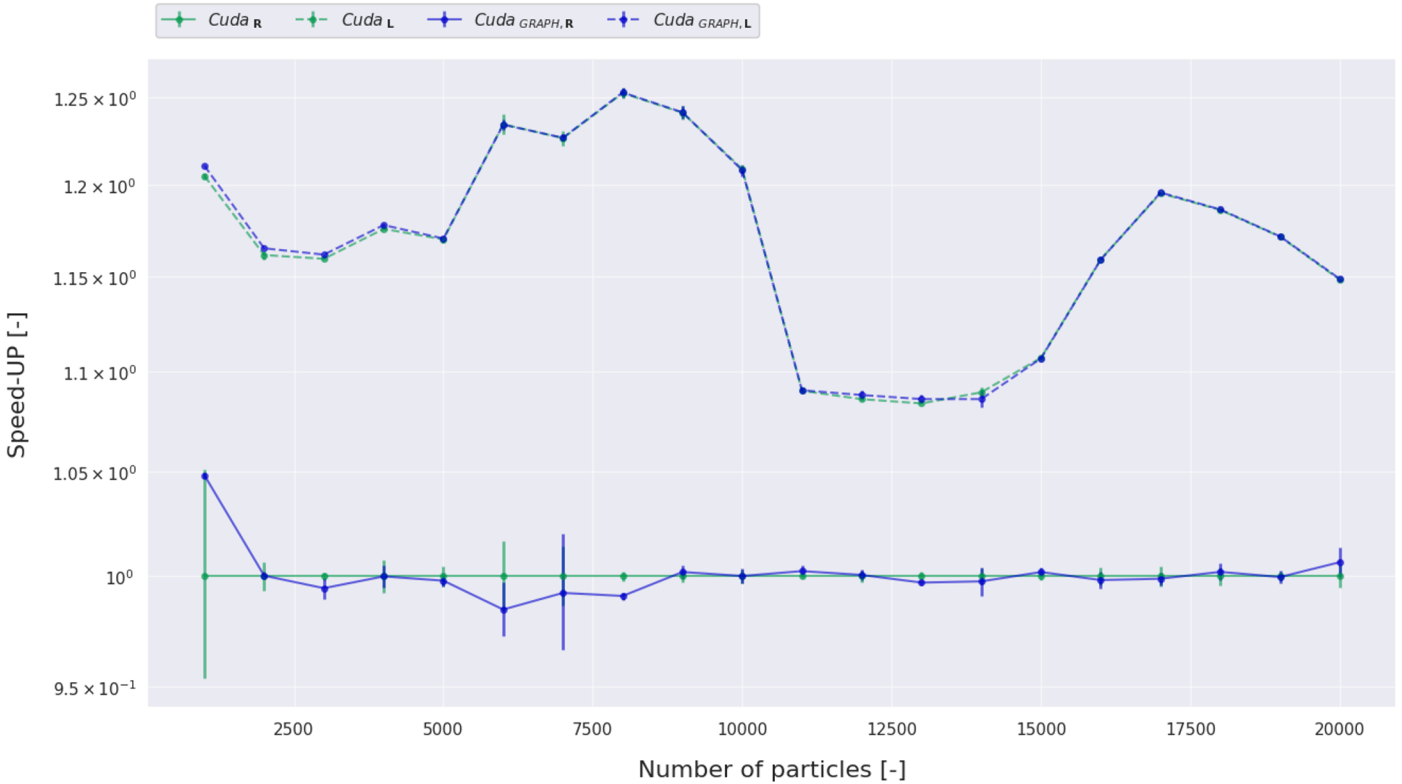
N-BODY SIMULATIONS (7/9)

Results (2/4)



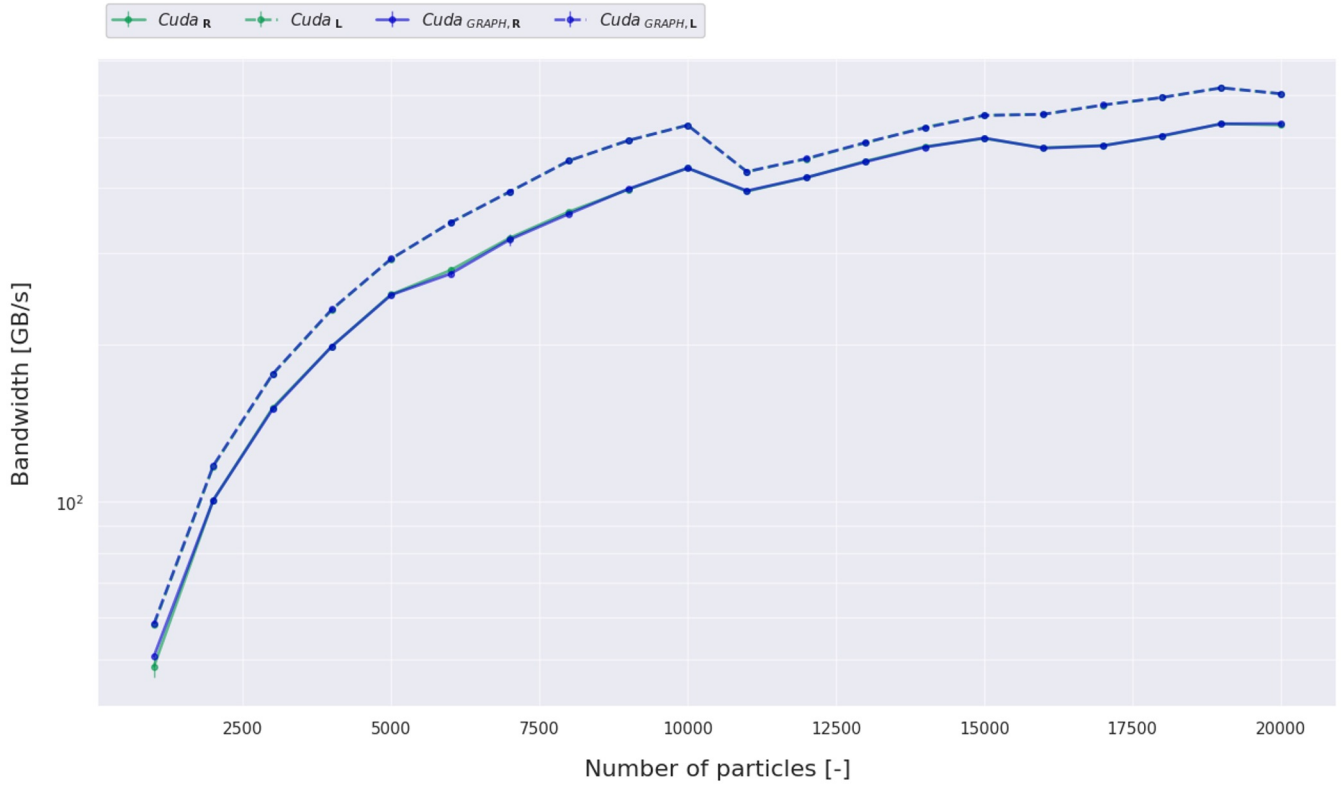
N-BODY SIMULATIONS (8/9)

Results (3/4)



N-BODY SIMULATIONS (9/9)

Results (4/4)



CONCLUSION

In conclusion, **Kokkos** is an **excellent tool** for **developing performance portable code** for parallel computing, providing **high-level abstractions** and a **unified programming interface** for efficient code reuse across different platforms.