Kokkos

Lucas Frérot, Lars Pastewka

Table of contents

Hardware abstraction

Kokkos is a hardware abstraction layer that uses templated C++. It can address NVIDIA, AMD and Intel hardware that all propagate their own programming models, in particular CUDA, HIP and SYCL, respectively. An overview of programming models can be found in Herten (2023). For an overview of the underlying GPU hardware – threads, warps, memory hierarchy – see the GPU architecture lecture.

After working through this lecture, you should be able to:

- allocate and access multidimensional arrays with Kokkos `Views` and understand their reference semantics,
- express parallel loops and reductions with `parallel_for` and `parallel_reduce`,
- explain how `RangePolicy` and `TeamPolicy` map onto GPU threads, blocks and shared memory,
- implement a simple explicit time-stepping scheme (the wave equation) with Kokkos.

Further Kokkos links:

- Documentation
- Tutorials
- Video lectures

Initialization

Kokkos needs to be initialized and finalized. Your main code should look like this:

```
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  // Initialize Kokkos
  Kokkos::initialize(argc, argv);

  // Run the simulation
  run_simulation();

  // Finalize Kokkos
  Kokkos::finalize();
  return 0;
}
```

Multidimensional arrays

A *memory space* defines where data resides on your machine (GPU, CPU, etc.). The *layout* defines how the data is organized in that memory space. Kokkos `View` is a reference-counted multi-dimensional array that specifies both the memory location and layout at compile time. Its semantics are similar to `std::shared_ptr`. The `View` API is described here.

As an example, let us consider a one-dimensional field $u(x)$ discretized on N equally-spaced points x_i. We represent this using a `View`:

```
// Field is a one-dimensional array
using Field_t = Kokkos::View<double*>;

// Allocate memory
const int N{1000};
Field_t u("u", N);
```

Note that all `Views` have names (to ease debugging).

Multidimensional arrays implement `operator()` for accessing elements. For example, we can initialize the field with:

```
double dx = 0.1; // Grid spacing
for (int i{0}; i < N; ++i) {
    u(i) = 0.1 * i * dx;
}
```

For multidimensional fields (for example, the density field in the Lattice Boltzmann method), we define a two-dimensional array:

```
using Density_t = Kokkos::View<double**>;

Density_t density("density", 10, 15);
density(2, 3) = 1.5;
```

The Kokkos documentation has more information on constructing `Views`.

> **Note**
>
> Kokkos `Views` are automatically reference counted, making them similar to a `std::shared_ptr`. This means assigning a `View` to another one does not actually copy the data. For example:
>
> ```
> Density_t a("a", 10, 15);
> Density_t b("b", 10, 15);
> b = a;
> ```
>
> does not copy anything. Both views `a` and `b` point to the same memory location after the assignment. Use `deep_copy` to actually copy the data. `deep_copy` can also be used to copy data from host to device and back.

Parallel loops

A companion to the *memory space* is the *execution space* which determines where calculations are carried out. Kokkos supports three execution patterns: Loops (`parallel_for`), reductions `parallel_reduce` and prefix sums `parallel_scan`. We here first discuss the loops.

The parallel loop executes part of the loop in parallel, either by vectorizing or distributing among processing units. The above initialization loop can be written with the Kokkos `parallel_for` pattern in the form

```
Kokkos::parallel_for(
    "initialize_u", N, KOKKOS_LAMBDA(const int i) {
      u(i) = 0.1 * i * dx;
    });
```

The macro `KOKKOS_LAMBDA` wraps a C++ lambda expression, allowing it to run on different execution spaces (CPU, GPU, etc.).

> **Warning**
>
> `KOKKOS_LAMBDA` expands to a lambda that captures *by copy* (`[=]`), because the lambda may execute on a device with its own memory. The classic pitfall is capturing `this` (implicitly, by referring to a member variable inside the lambda) or other host-side data such as a `std::vector` – the device then receives a copy of a host pointer that it cannot dereference. Capturing a `View` by copy is cheap and safe: a `View` is only a handle (pointer plus metadata) to data that already resides in the correct memory space. Inside class methods, copy members you need into local variables before the `parallel_for` and capture those.

Reductions

Loops that accumulate a result – a sum, a maximum, a dot product – must not simply write to a shared variable from many threads. Kokkos provides the `parallel_reduce` pattern for this purpose. Each thread accumulates into a private contribution, and Kokkos combines the contributions safely and efficiently. For example, the sum over all elements of the field `u`:

```
double sum = 0.0;
Kokkos::parallel_reduce(
    "sum_u", N,
    KOKKOS_LAMBDA(const int i, double& local_sum) {
      local_sum += u(i);
    },
    sum);
```

The second lambda argument `local_sum` is the thread-private accumulator; the final argument receives the reduced result. Other reductions (e.g. `Kokkos::Max`, `Kokkos::Min`, products) are available as built-in reducers. We will use `parallel_reduce` later, for example to compute norms and observables in the lecture on memory performance.

Multidimensional loops

Specifying the number of iterates `N` implicitly defines what Kokkos calls a range execution policy. For multidimensional loops, e.g. over a two-dimension grid, we need to explicitly define this policy:

```
Kokkos::parallel_for(
    "initialize_u", Kokkos::MDRangePolicy({0, 0}, {Nx, Ny}),
    KOKKOS_LAMBDA(const int i, const int j) {
        u(i, j) = 0.1 * i * dx + 0.2 * j * dy;
    });
```

The first and second argument of `MDRangePolicy` give the lower and upper bounds of the iteration.

How Kokkos maps to GPU hardware

Understanding the mapping between Kokkos abstractions and GPU hardware (see the GPU architecture lecture) helps you reason about performance without writing vendor-specific code.

`RangePolicy` – simple parallel loops

When you write a `parallel_for` with a simple integer range (or equivalently a `RangePolicy`), Kokkos maps this to a GPU kernel launch where each iteration `i` corresponds to one GPU thread. The runtime automatically chooses a block size (typically 128 or 256 threads) and launches enough blocks to cover all iterations. This is the simplest and most common pattern.

`TeamPolicy` – hierarchical parallelism

For more control, Kokkos provides `TeamPolicy` which exposes the two-level thread hierarchy of GPUs:

```
Kokkos::TeamPolicy<> policy(num_teams, team_size);
Kokkos::parallel_for("label", policy,
    KOKKOS_LAMBDA(const Kokkos::TeamPolicy<>::member_type& team) {
        // team.league_rank() = block index
        // team.team_rank()   = thread index within the block
        Kokkos::parallel_for(Kokkos::TeamThreadRange(team, work_per_team),
            [&](const int j) {
                // ...
            });
    });
```

Here `num_teams` corresponds to the number of GPU **thread blocks** (the *league size*) and `team_size` corresponds to the number of **threads per block**. `TeamThreadRange` distributes work among threads within a block. This policy also gives access to `ScratchMemorySpace`, Kokkos' abstraction for GPU shared memory:

```
using team_policy = Kokkos::TeamPolicy<>;
using member_type = team_policy::member_type;

int scratch_size = Kokkos::View<double*, Kokkos::DefaultExecutionSpace::scratch_memory_space
    shmem_size(tile_size);

Kokkos::parallel_for(
    team_policy(num_teams, team_size).set_scratch_size(0, Kokkos::PerTeam(scratch_size)),
    KOKKOS_LAMBDA(const member_type& team) {
        // Allocate a scratch view in shared memory
        Kokkos::View<double*, Kokkos::DefaultExecutionSpace::scratch_memory_space>
            tile(team.team_scratch(0), tile_size);

        // Cooperatively load data into shared memory
        Kokkos::parallel_for(Kokkos::TeamThreadRange(team, tile_size),
            [&](const int i) {
                tile(i) = global_data(team.league_rank() * tile_size + i);
            });

        // Synchronize -- ensure all threads have finished loading
        team.team_barrier();

        // Now all threads can read from tile() with low latency
        // ...
    });
```

`TeamPolicy` is needed when threads within a block must cooperate – for example, in tiled algorithms or reductions over shared data.

`View` memory spaces

Kokkos memory space	Maps to
`Kokkos::HostSpace`	CPU RAM
`Kokkos::CudaSpace` / `Kokkos::HIPSpace`	GPU global memory
`Kokkos::CudaUVMSpace` / `Kokkos::HIPManagedSpace`	Unified (managed) memory
`ScratchMemorySpace` (via `TeamPolicy`)	Shared memory / LDS

Data must reside in the correct memory space for the execution space. Use `Kokkos::deep_copy` to transfer data between host and device:

```
// Allocate on GPU
Kokkos::View<double*> d_data("device_data", N);
// Allocate on CPU
Kokkos::View<double*, Kokkos::HostSpace> h_data("host_data", N);

// Copy host → device
Kokkos::deep_copy(d_data, h_data);

// ... run GPU kernel on d_data ...

// Copy device → host
Kokkos::deep_copy(h_data, d_data);
```

Figure 1 summarizes the rules: each `View` lives in exactly one memory space, and only `deep_copy` moves data across the host–device boundary.

Nomenclature mapping

GPU concept	CUDA (NVIDIA)	ROCm/HIP (AMD)	Kokkos
Thread group	Thread block	Workgroup	Team
All groups	Grid	Grid	League
Shared memory	Shared memory	LDS	`ScratchMemorySpace`
Block size	`blockDim`	`workgroup_size`	`team_size`
Number of blocks	`gridDim`	`grid_size`	`league_size`

Example: The wave equation

Kokkos `Views` and the `parallel_for` pattern are already sufficient to implement simple numerical methods. We will here illustrate the numerical solution of the wave equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \equiv a(x,t)$$

as an example of how to use Kokkos. Here u is a displacement, e.g. of a string on which the wave propagates, and c is the wave speed. The left hand side is the acceleration of each point x on the string; we abbreviate the right hand side, which determines this acceleration, as $a(x,t)$.

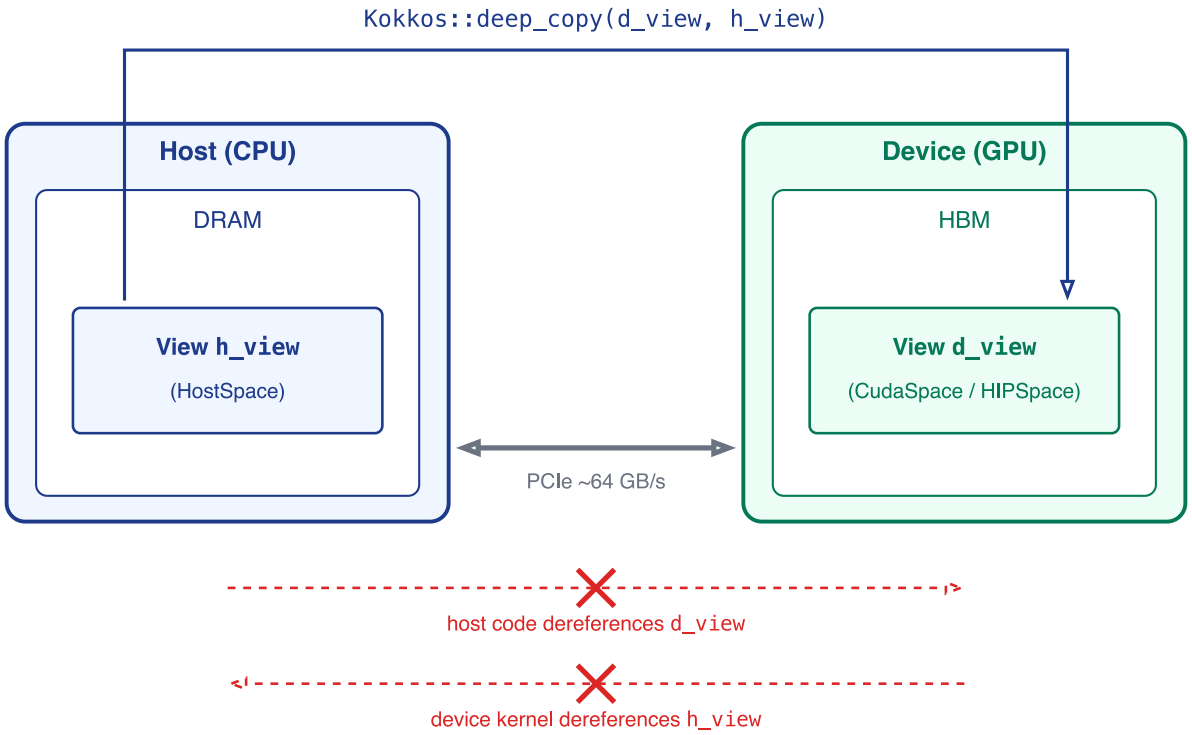

Figure 1: Host and device are separate memory spaces: a `View` allocates its data in one of them, `deep_copy` is the only way to move data across the PCIe/NVLink boundary, and dereferencing a host pointer in device code (or vice versa) is illegal.

We discretize the x-positions on an equidistant lattice with grid spacing Δx, $x_i = i\Delta x$ and $u_i = u(x_i)$. The second derivative can then be approximated through the second-order central-differences approximation

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x_{i-1}) - 2u(x_i) + u(x_{i+1})}{\Delta x^2}.$$

A straightforward implementation of this expression in Kokkos looks as follows:

```
/// Compute the second derivative using finite differences
void second_derivative_fd(Field_t& u_second, Field_t& u, double dx) {
  // Select everything except edges
  auto n{u.extent(0) - 2};

  // Centered finite differences
  Kokkos::parallel_for(
      "second_derivative_fd", n, KOKKOS_LAMBDA(const int i) {
        u_second(i + 1) = (u(i) - 2 * u(i + 1) + u(i + 2)) / (dx * dx);
      });
}
```

Note that the boundary nodes u_0 and u_{N-1} remain undefined.

For the propagation in time we need a time-stepping algorithm. We will here use the velocity Verlet algorithm. Since we are solving a second-order equation in time, we need two initial

conditions: the displacements $u(x, t = 0)$ and the velocities $v(x, t = 0)$. Here $v(x, t) = \dot{u}(x, t)$, where the dot indicates the derivative with respect to time t. The velocity Verlet algorithm is

$$u(x, t + \Delta t) = u(x, t) + v(x, t)\Delta t + \frac{a(x, t)}{2}\Delta t^2 \tag{1}$$

$$v(x, t + \Delta t) = v(x, t) + \frac{a(x, t) + a(x, t + \Delta t)}{2}\Delta t \tag{2}$$

In order to avoid storing both the previous $a(x, t)$ and current $a(x, t + \Delta t)$ accelerations, the velocity Verlet algorithm is often formulated as a predictor-corrector algorithm. The predictor step is then

$$v'(x, t + \Delta t/2) = v(x, t) + \frac{a(x, t)}{2}\Delta t \tag{3}$$

$$u(x, t + \Delta t) = u(x, t) + v'(x, t + \Delta t/2)\Delta t, \tag{4}$$

which is followed by a recalculation of the accelerations $a(x, t+\Delta t)$ from the updated displacements $u(x, t + \Delta t)$. The corrector steps then updates the velocities from the "predicted" velocities $v'(x, t + \Delta t/2)$:

$$v(x, t + \Delta t) = v'(x, t + \Delta t/2) + \frac{a(x, t + \Delta t)}{2}\Delta t$$

An implementation of the predictor and corrector steps of this algorithm in Kokkos looks like:

```
/// Verlet predictor step
void verlet_predictor(Field_t& u_second, Field_t& u_first, Field_t& u,
                      double dt) {
  auto n{u.extent(0)};
  Kokkos::parallel_for(
      "verlet_predictor", n, KOKKOS_LAMBDA(const int i) {
        u(i) += dt * u_first(i) + (0.5 * dt * dt) * u_second(i);
        u_first(i) += (dt * 0.5) * u_second(i);
      });
}

/// Verlet corrector step
void verlet_corrector(Field_t& u_second, Field_t& u_first, Field_t& u,
                      double dt) {
  auto n{u.extent(0)};
  Kokkos::parallel_for(
      "verlet_corrector", n, KOKKOS_LAMBDA(const int i) {
        u_first(i) += (dt * 0.5) * u_second(i);
      });
}
```

The full integration can then be carried out via

```
/// Verlet full step
void verlet(Field_t& u_second, Field_t& u_first, Field_t& u, double dt,
            double dx) {
  verlet_predictor(u_second, u_first, u, dt);
  second_derivative_fd(u_second, u, dx);
  verlet_corrector(u_second, u_first, u, dt);
}
```

The full code for the wave equation can be found in wave.cpp.

Running Kokkos code

Kokkos will run in parallel even if you have no accelerator. An example execution space is `Kokkos::Threads`. To enable threads, pass the appropriate CMake option when configuring Kokkos (see notes/kokkos_gpu_compilation.qmd).

When executing the code, you need to specify the number of threads. To run our wave-equation code on 4 parallel threads,

```
./wave --kokkos-num-threads=4
```

You can get a help page by running the code with the `--kokkos-help` command line option.

Files

Here are the helper files for this section:

- CMakeLists.txt
- wave.cpp
- animate.py
- plot.py

> **Note**
>
> Kokkos is one of several hardware abstraction layers for GPU programming. For a survey of alternatives – including OpenMP target offloading, RAJA, JAX, PyTorch, Taichi, and Julia – see the other abstraction layers lecture.

Herten, Andreas. 2023. "Many Cores, Many Models: GPU Programming Model Vs. Vendor Compatibility Overview." *arXiv [Cs.DC]*.