Contents

1 Source code

1.1 Baseline implementation

We begin by introducing the **naive** sequential implementation of the **P2P inner** kernel. This version serves as a baseline for the experiments and is only used to verify the numerical results. The implementation is quite basic, leaving significant room for optimization. Indeed, we can notice for example that it does not take advantage of the computation's inherent symmetry. In this approach, we simply iterate over the *target particles* and update their potential by calculating interactions with all *source particles*, while skipping self-interactions (we remind the reader that, in that case, the *target* and *source* particles are actually the same).

```
#pragma once

#include "utils.hpp"

#include <concepts>
#include <iostream>
#include <vector>

/**
 * @brief Naive version of the P2P inner kernel.
 *
 * This function calculates pairwise interactions (potential
 ↪  in this case) between particles based on
```

```
 * their positions and charges. It iterates over all target particles and computes their interaction
 * with all source particles, except themselves, and stores the result in the `potential` array.
 *
 * @tparam ContainerType The type of container used for the positions, charges, and potentials. This container must
 *         support random access (e.g., `std::vector`, `std::array`) and store floating-point values.
 *
 * @param[in] position_x Container of x-coordinates of the particles.
 * @param[in] position_y Container of y-coordinates of the particles.
 * @param[in] charge Container of charge values associated with each particle.
 * @param[out] potential Output container where the computed potential for each particle is stored.
 *
 */
template<Container ContainerType>
auto simplified_p2p_inner(ContainerType const& position_x, ContainerType const& position_y, ContainerType const& charge,
                          ContainerType& potential) -> void
{
    // set-up (aliases)
    using container_type = ContainerType;
    using value_type = typename container_type::value_type;

    const std::size_t size = position_x.size();
    value_type result;

    // loop through the target data
    for(std::size_t i = 0; i < size; ++i)
    {
        const value_type local_x_target = position_x[i], local_y_target = position_y[i];
        result = 0.;
```

```
        // loop through the source data (we skip the case
        ↪  where 'i == j')
        for(std::size_t j = 0; j < i; ++j)
        {
            const value_type diff_y = position_y[j] -
            ↪  local_y_target, diff_x = position_x[j] -
            ↪  local_x_target;
            result += charge[j] / std::sqrt(diff_y * diff_y +
            ↪  diff_x * diff_x);
        }
        for(std::size_t j = i + 1; j < size; ++j)
        {
            const value_type diff_y = position_y[j] -
            ↪  local_y_target, diff_x = position_x[j] -
            ↪  local_x_target;
            result += charge[j] / std::sqrt(diff_y * diff_y +
            ↪  diff_x * diff_x);
        }

        // update potential
        potential[i] = result;
    }
}
```

1.2 Existing implementations

Prior to the internship, there were existing implementations of the **P2P inner** kernel, which were originally more generic but had been adapted to the simplified two-dimensional case. These included a **CPU vectorized** version using xsimd and a CUDA version. We begin by presenting these two versions, as they will serve as a baseline for comparison with the Kokkos-based implementations.

1.2.1 Implementation on CPU using xsimd

This implementation is a simplified version of a kernel extracted from ScalFMM 3.0 and was adapted for this simplified context (the two-dimensional case). It leverages the xsimd C++ library which provides wrappers for SIMD intrinsics. Roughly speaking, the xsimd library automatically detects and uses the available SIMD instruction sets on the target machine, for instance

SSE, **AVX**, **AVX2**, and **AVX512** on **x86 architectures**, and **NEON** on **ARM architectures** (an exhaustive list of the supported instruction sets is given in their documentation). One of the key benefits of using xsimd is its portability; it offers a high-level API that can be used across different CPU architectures, simplifying the process of writing SIMD-optimized code without having to handle specific low-level instruction sets (have a look at the Intel instrinsics guide to see how cumbersome it can be to use it manually, especially for beginners). This library relies on SIMD instruction detection at compile-time, allowing the same code to adapt to different target architectures. Another consequence is that these wrappers don't introduce any runtime overhead compared to the use of raw instrinsics. Additionally, xsimd is tightly integrated with xtensor, a **Numpy-like** multidimensional array library for C++, which provides excellent tools for scientific computing. In the implementation below, we vectorize the first part of the outermost loop (over the *target* data) by using SIMD batches to process multiple particles simultaneously, while the remaining part of the loop is treated in the "standard scalar manner". To remove self-interactions, we use SIMD batch logical masks, which act as filters to exclude them efficiently.

```
#pragma once

#include "utils.hpp"

#include <concepts>
#include <iostream>
#include <type_traits>

#include <xsimd/xsimd.hpp>

// Helper function to print the SIMD instruction set being used
void print_simd_instruction_set()
{
    std::cout << "- SIMD instruction set: ";
#if XSIMD_WITH_AVX512F
    std::cout << "AVX-512" << '\n';
#elif XSIMD_WITH_AVX2
    std::cout << "AVX2" << '\n';
#elif XSIMD_WITH_AVX
    std::cout << "AVX" << '\n';
#elif XSIMD_WITH_SSE4_2
```

```
    std::cout << "SSE4.2" << '\n';
#elif XSIMD_WITH_SSE4_1
    std::cout << "SSE4.1" << '\n';
#elif XSIMD_WITH_SSE2
    std::cout << "SSE2" << '\n';
#elif XSIMD_WITH_NEON
    std::cout << "NEON" << '\n';
#else
    std::cout << "Unknown SIMD instruction set \n";
#endif
}

/**
* @brief P2P inner kernel using SIMD vectorization via xsimd.
*
* This function calculates pairwise interactions (potential in
↪  this case) between particles based
* on their positions and charges. It leverages SIMD
↪  vectorization via the **xsimd** library to
* process multiple particles simultaneously. The function uses
↪  SIMD to handle large portions of the
* data in parallel, while a scalar fallback handles any
↪  remaining particles that don't fit into a full
* SIMD batch. We use SIMD batch logical masks to remove
↪  self-interactions (acting as a filter).
*
* @tparam ContainerType The type of container used for the
↪  positions, charges, and potentials. This container must
*         support random access (e.g., `std::vector`,
↪  `std::array`) and store floating-point values.
*
* @param[in] position_x Container of x-coordinates of the
↪  particles.
* @param[in] position_y Container of y-coordinates of the
↪  particles.
* @param[in] charge Container of charge values associated with
↪  each particle.
* @param[out] potential Output container where the computed
↪  potential for each particle is stored.
*
```

```
*/
template<Container ContainerType>
auto p2p_inner_xsimd(ContainerType const& position_x,
↪  ContainerType const& position_y, ContainerType const&
↪  charge,
                     ContainerType& potential) -> void
{
    // Set-up (aliases)
    using container_type = ContainerType;
    using value_type = typename container_type::value_type;
    using batch_type = xsimd::batch<value_type>;
    static constexpr std::size_t batch_size = batch_type::size;

    const std::size_t size = position_x.size();
    const std::size_t vec_size = size - size % batch_size;

    const value_type local_tolerance{std::numeric_limits<value
    ↪  _type>::epsilon()};

    batch_type vec_contribution;

    // Initialize logical mask
    std::array<value_type, batch_size> next_target_indices;
    std::iota(next_target_indices.begin(),
    ↪  next_target_indices.end(), value_type(0.));
    auto vec_next_target_indices =
    ↪  xsimd::load_unaligned(next_target_indices.data());

    // Loop through target data (vectorized part)
    for(std::size_t i = 0; i < vec_size; i += batch_size,
    ↪  vec_next_target_indices += batch_size)
    {
        const batch_type vec_local_target_x =
        ↪  xsimd::load_unaligned(&position_x[i]);
        const batch_type vec_local_target_y =
        ↪  xsimd::load_unaligned(&position_y[i]);

        vec_contribution = 0.;

        // Loop through source data
```

```
    for(std::size_t j = 0; j < size; ++j)
    {
        const batch_type vec_diff_x = vec_local_target_x -
        ↪  position_x[j],
                         vec_diff_y = vec_local_target_y -
                         ↪  position_y[j];

        // Apply logical mask to skip the case 'i = j'
        const auto batch_mask =
        ↪  xsimd::abs(vec_next_target_indices - j) >
        ↪  local_tolerance;
        vec_contribution += xsimd::select(
          batch_mask, charge[j] / xsimd::sqrt(vec_diff_x *
          ↪  vec_diff_x + vec_diff_y * vec_diff_y),
          ↪  batch_type(0.));
    }

    vec_contribution.store_unaligned(&potential[i]);
}

value_type result{0.};

// Loop through target data (remaining scalar part)
for(std::size_t i = vec_size; i < size; ++i)
{
    const value_type local_target_x = position_x[i],
    ↪  local_target_y = position_y[i];
    result = 0.;

    // Loop through source data
    for(std::size_t j = 0; j < size; ++j)
    {
        if(i != j)
        {
            const value_type diff_x = local_target_x -
            ↪  position_x[j], diff_y = local_target_y -
            ↪  position_y[j];
            result += charge[j] / std::sqrt(diff_x *
            ↪  diff_x + diff_y * diff_y);
        }
```

```
        }

        potential[i] = result;
    }
}
```

1.2.2 Implementation on GPU using CUDA

The CUDA kernel presented here was originally extracted from ScalFMM 2.0 and was rewritten to fit the current simplified context. Before moving on, we provide the reader for brief yet comprehensive introductions to (old-fashioned) CUDA for beginners:

- An easy introduction to CUDA C and C++
- CUDA Refresher: The CUDA Programming Model

1. Useful macro for error handling For convenience, we define the following macro named `CHECK_CUDA` to handle errors in CUDA API calls.

```
#pragma once

#include "utils.hpp"

#include <concepts>
#include <functional>
#include <iostream>
#include <type_traits>

#include <cuda_runtime.h>

// Error handling macro
#define CHECK_CUDA(call)
↪
↪           \
    {
    ↪
    ↪               \
        cudaError_t err = call;
        ↪
        ↪                   \
```

```
        if(err != cudaSuccess)
        ↪
        ↪                   \
        {
        ↪
        ↪                   \
            throw std::runtime_error(std::string("CUDA
            ↪  error at ") + __FILE__ + ":" +
            ↪  std::to_string(__LINE__) +       \
                                        " - " + cudaGetError
                                        ↪  String(err));
                                        ↪
                                        ↪
                                        ↪               \
        }
        ↪
        ↪                   \
}
```

2. CUDA kernel using shared memory Below is the code for the CUDA version of the simplified **P2P inner** kernel [**nylons2007fast**]. This implementation leverages *shared memory* to reduce *global memory* access and improve performance. Blocks of *source data* (positions and charges) are cached into *shared memory* to minimize repeated access to *global memory*. This is achieved using CUDA synchronization primitives like `__syncthreads` to make sure that all the required data is loaded into *shared memory* before proceeding with the computation. Additionally, the kernel takes advantage of CUDA's **thread hierarchy**: each thread is responsible for computing the potential for a specific *target* particle by interacting with all *source* particles.

```
#pragma once

#include "cuda-common.hpp"
#include "utils.hpp"

#include <concepts>
#include <functional>
#include <iostream>
#include <type_traits>
```

```cpp
#include <cuda_runtime.h>

#define LOCAL_SHARED_MEMORY_SIZE 128

/**
 * @brief CUDA version of the P2P inner kernel.
 *
 * This function calculates pairwise interactions
 ↪  (potential in this case) between particles based on
 * their positions and charges. Each thread processes a
 ↪  "target" particle, computes its interaction with all
 * "source" particles, and accumulates the result into
 ↪  the potential array. Shared memory is used to cache
 * blocks of source data (positions and charges) to
 ↪  reduce global memory access, enhancing performance.
 *
 * @tparam ValueType Floating-point type for the position
 ↪  and charge arrays (e.g., float or double).
 *
 * @param[in] position_x Global memory array holding the
 ↪  x-coordinates of the particles.
 * @param[in] position_y Global memory array holding the
 ↪  y-coordinates of the particles.
 * @param[in] charge Global memory array holding the
 ↪  charge of each particles.
 * @param[out] potential Global memory array where the
 ↪  computed potential for each particle is stored.
 * @param[in] size The total number of particles.
 */
template<std::floating_point ValueType>
__global__ auto p2p_inner_gpu_shared_kernel(const
↪  ValueType* const position_x, const ValueType* const
↪  position_y,
                                            const ValueTy
                                            ↪  pe* const
                                            ↪  charge, V
                                            ↪  alueType*
                                            ↪  const
                                            ↪  potential,
```

```
                                                const std::si
                                                ↪  ze_t
                                                ↪  size) ->
                                                ↪  void
{
    using value_type = ValueType;

    static constexpr std::size_t shared_mem_size =
    ↪  LOCAL_SHARED_MEMORY_SIZE;
    const std::size_t total_nb_threads = blockDim.x *
    ↪  gridDim.x;
    const std::size_t unique_thread_id = threadIdx.x +
    ↪  blockIdx.x * blockDim.x;
    const std::size_t total_nb_iterations = ((size +
    ↪  blockDim.x - 1) / blockDim.x) * blockDim.x;

    // loop through all the target data
    for(std::size_t target_idx = unique_thread_id;
    ↪  target_idx < total_nb_iterations; target_idx +=
    ↪  total_nb_threads)
    {
        bool thread_compute = (target_idx < size);

        value_type target_position_x{},
        ↪  target_position_y{}, target_potential{};

        if(thread_compute)
        {
            // load current target data from global
            ↪  memory to register
            target_position_x = position_x[target_idx];
            target_position_y = position_y[target_idx];
        }
        // loop through all the source data, processing
        ↪  in blocks cached to shared memory
        for(std::size_t shared_start_idx = 0;
        ↪  shared_start_idx < size; shared_start_idx +=
        ↪  shared_mem_size)
        {
```

```
__shared__ value_type
↪  shared_source_position_x[shared_mem_size];
__shared__ value_type
↪  shared_source_position_y[shared_mem_size];
__shared__ value_type
↪  shared_source_charge[shared_mem_size];
// compute the number of entries to cache in
↪  shared memory
const std::size_t nb_copies =
  (shared_mem_size < (size -
  ↪  shared_start_idx)) ? shared_mem_size :
  ↪  (size - shared_start_idx);

// load source data from global memory to
↪  shared memory
for(std::size_t shared_local_idx =
↪  threadIdx.x; shared_local_idx < nb_copies;
    shared_local_idx += blockDim.x)
{
    const std::size_t global_source_idx =
    ↪  shared_local_idx + shared_start_idx;
    shared_source_position_x[shared_local_idx
    ↪  ] =
    ↪  position_x[global_source_idx];
    shared_source_position_y[shared_local_idx
    ↪  ] =
    ↪  position_y[global_source_idx];
    shared_source_charge[shared_local_idx] =
    ↪  charge[global_source_idx];
}

// synchronize all threads within the block
__syncthreads();

if(thread_compute)
{
    // skip the case 'target idx == source
    ↪  idx' in the summation
    // note that 'source_idx =
    ↪  shared_start_idx + local_source_idx'
```

```
std::size_t left_copies = nb_copies;
if(shared_start_idx <= target_idx &&
↪  target_idx < shared_start_idx +
↪  nb_copies)
{
    left_copies = target_idx -
    ↪  shared_start_idx;
}

// left part: loop for 'source idx <
↪  target_idx'
for(std::size_t local_source_idx = 0;
↪  local_source_idx < left_copies;
↪  ++local_source_idx)
{
    const value_type diff_x = shared_sour
    ↪  ce_position_x[local_source_idx] -
    ↪  target_position_x;
    const value_type diff_y = shared_sour
    ↪  ce_position_y[local_source_idx] -
    ↪  target_position_y;
    target_potential +=
      shared_source_charge[local_source_i
      ↪  dx] / std::sqrt(diff_x * diff_x
      ↪  + diff_y * diff_y);
}
// right part: loop for 'source idx >
↪  target_idx'
for(std::size_t local_source_idx =
↪  left_copies + 1; local_source_idx <
↪  nb_copies; ++local_source_idx)
{
    const value_type diff_x = shared_sour
    ↪  ce_position_x[local_source_idx] -
    ↪  target_position_x;
    const value_type diff_y = shared_sour
    ↪  ce_position_y[local_source_idx] -
    ↪  target_position_y;
    target_potential +=
```

```
                        shared_source_charge[local_source_i
                        ↪  dx] / std::sqrt(diff_x * diff_x
                        ↪  + diff_y * diff_y);
                    }
                }

                // synchronize all threads within the block
                __syncthreads();
            }

            // update outputs of the current target
            if(thread_compute)
            {
                potential[target_idx] = target_potential;
            }

            // synchronize all threads within the block
            __syncthreads();
        }
    }
```

To run the previous kernel, all necessary data must be properly transferred to the *device* (GPU), and once the computation is complete, the required results are copied back to the *host* (CPU). This process is handled by the function shown below. We opted for "C-style" **explicit memory copies**, using low-level instructions like `cudaMalloc` and `cudaMemcpy` (see here). Another interesting alternative would have been to use the more modern *high-level* library Thrust. We could also have considered using CUDA Unified memory, which simplifies memory management. By the way, it is worth noting that Kokkos supports both approaches: **explicit transfers** and **unified memory**. Finally, as it commonly done in CUDA, we run this kernel by setting up a **grid** of **blocks of threads**, with a tunable block size.

```
/**
 * @brief Executes the CUDA version of the P2P inner
 ↪  kernel.
 *
```

```
 * This function sets up and manages the execution of the
 ↪  CUDA kernel `p2p_inner_gpu_shared_kernel` that
 ↪  computes
 * pairwise interactions (here the potential) between
 ↪  particles. The function transfers data from
 * the host (CPU) to the device (GPU), allocates device
 ↪  memory, runs the kernel, synchronizes the GPU,
 ↪  retrieves
 * the results, and measures the total elapsed time of
 ↪  the kernel execution.
 *
 * @tparam ContainerType The type of container used for
 ↪  the positions, charges, and potentials. This
 ↪  container must
 *         support random access (e.g., `std::vector`,
 ↪  `std::array`) and store floating-point values.
 *
 * @param[in] position_x Container of x-coordinates of
 ↪  the particles on the host.
 * @param[in] position_y Container of y-coordinates of
 ↪  the particles on the host.
 * @param[in] charge Container of charge values
 ↪  associated with each particle on the host.
 * @param[out] potential Output container where the
 ↪  computed potential for each particle is stored on
 ↪  the host.
 * @param[out] elapsed_time The elapsed time (in seconds)
 ↪  for the kernel execution.
 * @param[in] threads_per_block The number of threads per
 ↪  block to use when launching the CUDA kernel.
 */
template<Container ContainerType>
auto p2p_inner_gpu_shared(ContainerType const&
↪  position_x, ContainerType const& position_y,
↪  ContainerType const& charge,
                          ContainerType& potential,
                          ↪  float& elapsed_time, int
                          ↪  threads_per_block) -> void
{
    using container_type = ContainerType;
```

```
using value_type = typename
↪  container_type::value_type;

const std::size_t size = position_x.size();
value_type *d_position_x, *d_position_y, *d_charge,
↪  *d_potential;
size_t mem_size = size * sizeof(value_type);

// Create CUDA events for timestamps
cudaEvent_t start, stop;

CHECK_CUDA(cudaEventCreate(&start));
CHECK_CUDA(cudaEventCreate(&stop));

// Allocate memory on the device
CHECK_CUDA(cudaMalloc(&d_position_x, mem_size));
CHECK_CUDA(cudaMalloc(&d_position_y, mem_size));
CHECK_CUDA(cudaMalloc(&d_charge, mem_size));
CHECK_CUDA(cudaMalloc(&d_potential, mem_size));

// Copy data from host to device
CHECK_CUDA(cudaMemcpy(d_position_x,
↪  position_x.data(), mem_size,
↪  cudaMemcpyHostToDevice));
CHECK_CUDA(cudaMemcpy(d_position_y,
↪  position_y.data(), mem_size,
↪  cudaMemcpyHostToDevice));
CHECK_CUDA(cudaMemcpy(d_charge, charge.data(),
↪  mem_size, cudaMemcpyHostToDevice));

// Define grid dimension
const std::size_t blocks_per_grid = (size +
↪  threads_per_block - 1) / threads_per_block;

// Run kernel function
CHECK_CUDA(cudaEventRecord(start));

p2p_inner_gpu_shared_kernel<<<blocks_per_grid,
↪  threads_per_block>>>(d_position_x, d_position_y,
↪  d_charge,
```

```
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ ⌋
↪ d⌋
↪ _⌋
↪ p⌋
↪ o⌋
↪ t⌋
↪ e⌋
↪ n⌋
↪ t⌋
↪ i⌋
↪ a⌋
↪ l⌋
↪ ⌋
↪ ,⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ s⌋
```

```
    CHECK_CUDA(cudaGetLastError());
    CHECK_CUDA(cudaDeviceSynchronize());

    CHECK_CUDA(cudaEventRecord(stop));
    CHECK_CUDA(cudaEventSynchronize(stop));
    CHECK_CUDA(cudaEventElapsedTime(&elapsed_time, start,
    ↪  stop));
    elapsed_time /= 1.e3;   // conversion ms -> s

    // Copy the result matrix from device to host
    CHECK_CUDA(cudaMemcpy(potential.data(), d_potential,
    ↪  mem_size, cudaMemcpyDeviceToHost));

    // Free device memory
    CHECK_CUDA(cudaFree(d_position_x));
    CHECK_CUDA(cudaFree(d_position_y));
    CHECK_CUDA(cudaFree(d_charge));
    CHECK_CUDA(cudaFree(d_potential));

    // Destroy CUDA events
    CHECK_CUDA(cudaEventDestroy(start));
    CHECK_CUDA(cudaEventDestroy(stop));
}
```

1.3 Implementations using Kokkos

In this section, we present the implementations that were obtained using Kokkos. To give a general idea, Kokkos offers **built-in instructions** that can be used to parallelize common patterns. We focused particularly on the `Kokkos::parallel_for` and `Kokkos::parallel_reduce` instructions, which allow for quick parallelization of for loops. The `Kokkos::parallel_for` instruction is used to parallelize a for loop with independent iterations, while `Kokkos::parallel_reduce` is used to perform a reduction operation. We explored two approaches to parallelize the **P2P inner** with Kokkos. The first involves **flat parallelism**, where only one of the two loops is parallelized. The second approach employs **nested** and **hierarchical parallelism**, allowing both loops to be parallelized simultaneously.

1.3.1 Flat parallelism

1. Using `parallel for` Let's dive into the code provided below. We begin by defining a set of **aliases** via the `using` keyword (this is completely optional but it helps to simplify the code by avoiding to deal directly with lengthy type names). The memory space (where the data is stored) and the execution space (where the code runs) are determined at compile time, based on the chosen **backend**. This is controlled by a set of **macros** defined before the function, although Kokkos can often deduce the appropriate memory and execution spaces automatically, depending on the selected **backend**. Additionally, we can specify the memory layout (for example **layout right** or **layout left**, more details here), which is useful for handling **multi-dimensional arrays**. For optimal performance, the most efficient layout depends on the **backend** in use.

   Now let's shift our focus on the "computation part" of the kernel. This implementation uses the `Kokkos::parallel_for` instruction to parallelize the *outermost* loop. Note the similarity with the traditional (and notorious) `#pragma omp parallel_for` OpenMP directive. This instruction is used in combination with a 1D range policy used to iterate over the *source* particle indices (specified as the first argument of the `Kokkos::parallel_for` instruction). Note that the parallelized instructions (the *body*) are contained within a lambda function (roughly speaking, an **anonymous function**), although a functor could also be used (which is a **class** or **struct** that overloads the `operator()` allowing instances of that class to be called as if they were functions).

   Kokkos uses custom containers called views to manage memory on both the *host* and the *device*. Unlike standard containers, only views can be used within a `Kokkos::parallel_for` section. Data is populated into views on the *host* via mirrors, and if needed, a deep copy transfers the data from the *host* to the *device*. However, if the code is set to run directly on the *host*, no data transfer occurs. Thus, this implementation is portable, it can run on CPUs (with OpenMP or C++ threads as backends) as well as on GPUs (using CUDA as the backend for example).

```
#pragma once

#include "utils.hpp"
```

```
#include <Kokkos_Core.hpp>

#include <concepts>
#include <iostream>
#include <type_traits>

#ifdef KOKKOS_ENABLE_CUDA
#define MemSpace Kokkos::CudaSpace
#define ExecSpace Kokkos::Cuda
#define Layout Kokkos::LayoutLeft
#endif

#ifdef KOKKOS_ENABLE_OPENMP
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::OpenMP
#define Layout Kokkos::LayoutRight
#endif

#ifdef KOKKOS_ENABLE_THREADS
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::Threads
#define Layout Kokkos::LayoutRight
#endif

#ifndef MemSpace
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::DefaultHostExecutionSpace
#define Layout Kokkos::LayoutRight
#endif

/**
 * @brief Portable P2P inner kernel using Kokkos
 ↪  parallelism (only the 'Kokkos::parallel_for'
 ↪  instruction).
 *
 * This function calculates pairwise interactions
 ↪  (potential in this case) between particles based on
 * their positions and charges using the
 ↪  'Kokkos::parallel_for' instruction to parallelize
 ↪  the outermost
```

```
 * loop of the kernel. It enables parallel execution
 ↪  across different hardware architectures (e.g., CPU,
 * GPU), making it portable. It sets up device views,
 ↪  copies data between host and device,
 * runs the parallel computation, and retrieves the
 ↪  results back to the host (note that no data transfer
 * occurs when the code is executed directly on the host).
 *
 * @tparam ContainerType The type of container used for
 ↪  the positions, charges, and potentials. This
 ↪  container must
 *         support random access (e.g., `std::vector`,
 ↪  `std::array`) and store floating-point values.
 *
 * @param[in] position_x Container of x-coordinates of
 ↪  the particles on the host.
 * @param[in] position_y Container of y-coordinates of
 ↪  the particles on the host.
 * @param[in] charge Container of charge values
 ↪  associated with each particle on the host.
 * @param[out] potential Output container where the
 ↪  computed potential for each particle is stored on
 ↪  the host.
 * @param[out] elapsed_time The elapsed time (in seconds)
 ↪  for the parallel computation.
 *
*/
template<Container ContainerType>
auto p2p_inner_flat_parallel_for(ContainerType const&
↪  position_x, ContainerType const& position_y,
                                 ContainerType const&
                                 ↪  charge,
                                 ↪  ContainerType&
                                 ↪  potential, float&
                                 ↪  elapsed_time) -> void
{
    // Set-up (aliases)
    using container_type = ContainerType;
    using value_type = typename
    ↪  container_type::value_type;
```

```
    using execution_space_type = ExecSpace;
    using memory_space_type = MemSpace;
    using layout_type = Layout;

    using policy_type =
    ↪  Kokkos::RangePolicy<execution_space_type>;

    using view_type = Kokkos::View<value_type*,
    ↪  layout_type, memory_space_type>;
    using host_mirror_type = typename
    ↪  view_type::HostMirror;

    using timer_type = Kokkos::Timer;

#ifdef LOCAL_VERBOSITY
    // Print execution space details
    std::cout << std::string(40, '-') << '\n';
    execution_space_type ex{};
    std::cout << " - Execution Space = " << ex.name() <<
    ↪  '\n';

    // Print memory space details
    memory_space_type mem{};
    std::cout << " - Memory Space = " << mem.name() <<
    ↪  '\n';
    std::cout << std::string(40, '-') << '\n';
#endif

    const std::size_t size = position_x.size();

    // Declaration of the views
    view_type view_position_x("view_position_x", size);
    view_type view_position_y("view_position_y", size);
    view_type view_charge("view_charge", size);
    view_type view_potential("view_potential", size);

    // Create host mirrors of device views
    host_mirror_type host_position_x =
    ↪  Kokkos::create_mirror_view(view_position_x);
```

```
host_mirror_type host_position_y =
↪  Kokkos::create_mirror_view(view_position_y);
host_mirror_type host_charge =
↪  Kokkos::create_mirror_view(view_charge);
host_mirror_type host_potential =
↪  Kokkos::create_mirror_view(view_potential);

// Initialize vectors on host (via mirrors)
for(std::size_t i = 0; i < size; ++i)
{
    host_position_x[i] = position_x[i];
    host_position_y[i] = position_y[i];
    host_charge[i] = charge[i];
}

// Deep copy from host mirrors to device views
Kokkos::deep_copy(view_position_x, host_position_x);
Kokkos::deep_copy(view_position_y, host_position_y);
Kokkos::deep_copy(view_charge, host_charge);

timer_type timer{};

// Actual computation
Kokkos::parallel_for(
  "Extern for", policy_type(0, size),
  ↪  KOKKOS_LAMBDA(std::size_t i) {
      const value_type local_x_target =
      ↪  view_position_x(i), local_y_target =
      ↪  view_position_y(i);
      value_type result{0.};
      for(std::size_t j = 0; j < i; ++j)
      {
          const value_type diff_x =
          ↪  view_position_x(j) - local_x_target,
                           diff_y =
                           ↪  view_position_y(j) -
                           ↪  local_y_target;
          result += view_charge[j] / std::sqrt(diff_y
          ↪  * diff_y + diff_x * diff_x);
      }
```

```
            for(std::size_t j = i + 1; j < size; ++j)
            {
                const value_type diff_x =
                ↪  view_position_x(j) - local_x_target,
                                diff_y =
                                ↪  view_position_y(j) -
                                ↪  local_y_target;
                result += view_charge[j] / std::sqrt(diff_y
                ↪  * diff_y + diff_x * diff_x);
            }
            view_potential(i) = result;
        });

    // We make sure that everything is completed before
    ↪  moving on...
    Kokkos::fence();

    elapsed_time = timer.seconds();

    // Deep copy device view to host view
    Kokkos::deep_copy(host_potential, view_potential);

    // Retrieve values
    for(std::size_t i = 0; i < size; ++i)
    {
        potential[i] = host_potential[i];
    }
}
```

2. Using `parallel reduce` We now present a second approach. Unlike the previous implementation, this version uses the `Kokkos::parallel_reduce` instruction to parallelize the *innermost* loop, performing a reduction on the *target* potential being updated. While this alternative produces correct **numerical results**, it is expected to have lower performance due to the reduced level of parallelism compared to the previous version (this is confirmed with the results shown in the result section). Similarly to the first approach, we use a 1D range policy to iterate over the *source* particle indices. Additionally, we still use views and mirrors to manage potential data transfers between the *host* and *device* as needed.

```
#pragma once

#include "utils.hpp"

#include <Kokkos_Core.hpp>

#include <concepts>
#include <iostream>
#include <type_traits>

#ifdef KOKKOS_ENABLE_CUDA
#define MemSpace Kokkos::CudaSpace
#define ExecSpace Kokkos::Cuda
#define Layout Kokkos::LayoutLeft
#endif

#ifdef KOKKOS_ENABLE_OPENMP
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::OpenMP
#define Layout Kokkos::LayoutRight
#endif

#ifdef KOKKOS_ENABLE_THREADS
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::Threads
#define Layout Kokkos::LayoutRight
#endif

#ifndef MemSpace
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::DefaultHostExecutionSpace
#define Layout Kokkos::LayoutRight
#endif

/**
,* @brief Portable P2P inner kernel using Kokkos
↪  parallelism (only the 'Kokkos::parallel_reduce'
↪  instruction).
,*
```

```
,* This function calculates pairwise interactions
↪  (potential in this case) between particles based on
,* their positions and charges using the
↪  'Kokkos::parallel_reduce' instruction to parallelize
↪  the innermost
,* loop of the kernel. It enables parallel execution
↪  across different hardware architectures (e.g., CPU,
,* GPU), making it portable. It sets up device views,
↪  copies data between host and device,
,* runs the parallel computation, and retrieves the
↪  results back to the host (note that no data transfer
,* occurs when the code is executed directly on the host).
,*
,* @tparam ContainerType The type of container used for
↪  the positions, charges, and potentials. This
↪  container must
,*         support random access (e.g., `std::vector`,
↪  `std::array`) and store floating-point values.
,*
,* @param[in] position_x Container of x-coordinates of
↪  the particles on the host.
,* @param[in] position_y Container of y-coordinates of
↪  the particles on the host.
,* @param[in] charge Container of charge values
↪  associated with each particle on the host.
,* @param[out] potential Output container where the
↪  computed potential for each particle is stored on the
↪  host.
,* @param[out] elapsed_time The elapsed time (in seconds)
↪  for the parallel computation.
,*
,*/
template<Container ContainerType>
auto p2p_inner_flat_parallel_reduce(ContainerType const&
↪  position_x, ContainerType const& position_y,
↪  ContainerType const& charge,
                                    ContainerType&
                                    ↪  potential, float&
                                    ↪  elapsed_time) ->
                                    ↪  void
```

```
{
    // Set-up (aliases)
    using container_type = ContainerType;
    using value_type = typename
    ↪  container_type::value_type;

    using execution_space_type = ExecSpace;
    using memory_space_type = MemSpace;
    using layout_type = Layout;

    using policy_type =
    ↪  Kokkos::RangePolicy<execution_space_type>;

    using view_type = Kokkos::View<value_type*,
    ↪  layout_type, memory_space_type>;
    using host_mirror_type = typename
    ↪  view_type::HostMirror;

    using timer_type = Kokkos::Timer;

#ifdef LOCAL_VERBOSITY
    // Print execution space details
    std::cout << std::string(40, '-') << '\n';
    execution_space_type ex{};
    std::cout << " - Execution Space = " << ex.name() <<
    ↪  '\n';

    // Print memory space details
    memory_space_type mem{};
    std::cout << " - Memory Space = " << mem.name() <<
    ↪  '\n';
    std::cout << std::string(40, '-') << '\n';
#endif

    const std::size_t size = position_x.size();

    // Declaration of the views
    view_type view_position_x("view_position_x", size);
    view_type view_position_y("view_position_y", size);
    view_type view_charge("view_charge", size);
```

```
view_type view_potential("view_potential", size);

// Create host mirrors of device views
host_mirror_type host_position_x =
↪  Kokkos::create_mirror_view(view_position_x);
host_mirror_type host_position_y =
↪  Kokkos::create_mirror_view(view_position_y);
host_mirror_type host_charge =
↪  Kokkos::create_mirror_view(view_charge);
host_mirror_type host_potential =
↪  Kokkos::create_mirror_view(view_potential);

// Initialize vectors on host (via mirrors)
for(std::size_t i = 0; i < size; ++i)
{
    host_position_x[i] = position_x[i];
    host_position_y[i] = position_y[i];
    host_charge[i] = charge[i];
}

// Deep copy host mirrors to device views
Kokkos::deep_copy(view_position_x, host_position_x);
Kokkos::deep_copy(view_position_y, host_position_y);
Kokkos::deep_copy(view_charge, host_charge);

timer_type timer{};

// Actual computation
for(std::size_t i = 0; i < size; ++i)
{
    const value_type local_x_target =
    ↪  host_position_x(i), local_y_target =
    ↪  host_position_y(i);

    Kokkos::parallel_reduce(
      "Intern reduce", policy_type(0, size),
      KOKKOS_LAMBDA(std::size_t j, value_type&
      ↪  result) {
          if(i != j)
          {
```

```
                    const value_type diff_y =
                    ↪  view_position_y(j) -
                    ↪  local_y_target, diff_x =
                    ↪  view_position_x(j) - local_x_target;
                    result += view_charge[j] /
                    ↪  std::sqrt(diff_y * diff_y + diff_x
                    ↪  * diff_x);
                }
            },
            host_potential(i));
    }

    // We make sure that everything is completed before
    ↪  moving on...
    Kokkos::fence();

    elapsed_time = timer.seconds();

    // No deep copy needed here as host_potential was
    ↪  directly updated
    // Kokkos::deep_copy( host_potential, view_potential
    ↪  );

    // Retrieve values
    for(std::size_t i = 0; i < size; ++i)
    {
        potential[i] = host_potential[i];
    }
}
```

1.3.2 Nested parallelism

The two previous approaches can be categorized as **flat parallelism**. However, this new version of the **P2P inner** kernel below uses nested parallelism, which involves multiple layers of **parallel execution**. In practice, we combine several Kokkos instructions, such as `Kokkos::parallel_for` and `Kokkos::parallel_reduce`, to achieve this (most of the time, we expect this to lead to better results than with simple **flat parallelism**). While the handling of views, mirrors, and deep copies remains unchanged from the previous implementations, the "computation part" differs as both loops are now par-

allelized. Instead of using a simple 1D range policy, we employ a team policy that distributes the workload across the two loops. This approach distributes the threads into **teams** that form a **league** (more details here), drawing an analogy to CUDA, where all the threads are organized in a **grid** made up of **blocks**.

```
#pragma once

#include "utils.hpp"

#include <Kokkos_Core.hpp>

#include <concepts>
#include <iostream>
#include <type_traits>

#ifdef KOKKOS_ENABLE_CUDA
#define MemSpace Kokkos::CudaSpace
#define ExecSpace Kokkos::Cuda
#define Layout Kokkos::LayoutLeft
#endif

#ifdef KOKKOS_ENABLE_OPENMP
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::OpenMP
#define Layout Kokkos::LayoutRight
#endif

#ifdef KOKKOS_ENABLE_THREADS
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::Threads
#define Layout Kokkos::LayoutRight
#endif

#ifndef MemSpace
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::DefaultHostExecutionSpace
#define Layout Kokkos::LayoutRight
#endif
```

```cpp
/**
* @brief Portable P2P inner kernel using Kokkos parallelism
↪  (both 'Kokkos::parallel_for' and 'Kokkos::parallel_reduce'
↪  instructions).
*
* This function calculates pairwise interactions (potential in
↪  this case) between particles based on
* their positions and charges using nested parallelism via
↪  Kokkos. It combine the 'Kokkos::parallel_for'
* instruction to parallelize the outermost loop and the
↪  'Kokkos::parallel_reduce' instruction to parallelize
* the innermost loop.
*
* @tparam ContainerType The type of container used for the
↪  positions, charges, and potentials. This container must
*         support random access (e.g., `std::vector`,
↪  `std::array`) and store floating-point values.
*
* @param[in] position_x Container of x-coordinates of the
↪  particles on the host.
* @param[in] position_y Container of y-coordinates of the
↪  particles on the host.
* @param[in] charge Container of charge values associated with
↪  each particle on the host.
* @param[out] potential Output container where the computed
↪  potential for each particle is stored on the host.
* @param[out] elapsed_time The elapsed time (in seconds) for
↪  the parallel computation.
*/
template<Container ContainerType>
auto p2p_inner_nested_parallel(ContainerType const&
↪  position_x, ContainerType const& position_y,
                               ContainerType const& charge,
                               ↪  ContainerType& potential,
                               ↪  float& elapsed_time) -> void
{
    // Set-up (aliases)
    using container_type = ContainerType;
    using value_type = typename container_type::value_type;
```

```
    using execution_space_type = ExecSpace;
    using memory_space_type = MemSpace;
    using layout_type = Layout;

    using view_type = Kokkos::View<value_type*, layout_type,
    ↪  memory_space_type>;
    using host_mirror_type = typename view_type::HostMirror;

    using range_policy_type =
    ↪  Kokkos::TeamPolicy<execution_space_type>;
    using member_type = typename
    ↪  range_policy_type::member_type;

    using timer_type = Kokkos::Timer;

#ifdef LOCAL_VERBOSITY
    // Print execution space details
    std::cout << std::string(40, '-') << '\n';
    execution_space_type ex{};
    std::cout << " - Execution Space = " << ex.name() << '\n';

    // Print memory space details
    memory_space_type mem{};
    std::cout << " - Memory Space = " << mem.name() << '\n';
    std::cout << std::string(40, '-') << '\n';
#endif

    const std::size_t size = position_x.size();

    // Declaration of the views
    view_type view_position_x("view_position_x", size);
    view_type view_position_y("view_position_y", size);
    view_type view_charge("view_charge", size);
    view_type view_potential("view_potential", size);

    // Create host mirrors of device views
    host_mirror_type host_position_x =
    ↪  Kokkos::create_mirror_view(view_position_x);
    host_mirror_type host_position_y =
    ↪  Kokkos::create_mirror_view(view_position_y);
```

```
host_mirror_type host_charge =
↪  Kokkos::create_mirror_view(view_charge);
host_mirror_type host_potential =
↪  Kokkos::create_mirror_view(view_potential);

// Initialize vectors on host (via mirrors)
for(std::size_t i = 0; i < size; ++i)
{
    host_position_x[i] = position_x[i];
    host_position_y[i] = position_y[i];
    host_charge[i] = charge[i];
}

// Deep copy host mirrors to device views
Kokkos::deep_copy(view_position_x, host_position_x);
Kokkos::deep_copy(view_position_y, host_position_y);
Kokkos::deep_copy(view_charge, host_charge);

timer_type timer{};

// Actual computation
Kokkos::parallel_for(
  "Extern for", range_policy_type(size, Kokkos::AUTO),
  ↪  KOKKOS_LAMBDA(const member_type& teamMember) {
      const std::size_t i = teamMember.league_rank();
      const value_type local_x_target =
      ↪  view_position_x(i), local_y_target =
      ↪  view_position_y(i);
      value_type result{0.};

      Kokkos::parallel_reduce(
        Kokkos::TeamThreadRange(teamMember, size),
        [&](std::size_t j, value_type& innerUpdate)
        {
            if(i != j)
            {
                const value_type diff_y =
                ↪  view_position_y(j) - local_y_target,
                ↪  diff_x = view_position_x(j) -
                ↪  local_x_target;
```

```
                    innerUpdate += view_charge[j] /
                    ↪  std::sqrt(diff_y * diff_y + diff_x *
                    ↪  diff_x);
                }
            },
            result);

        Kokkos::single(Kokkos::PerTeam(teamMember), [&]() {
        ↪  view_potential(i) += result; });
    });

    // We make sure that everything is completed before moving
    ↪  on...
    Kokkos::fence();

    elapsed_time = timer.seconds();

    // Deep copy from device view to host mirror
    Kokkos::deep_copy(host_potential, view_potential);

    // Retrieve values
    for(std::size_t i = 0; i < size; ++i)
    {
        potential[i] = host_potential[i];
    }
}
```

1.3.3 Hierarchical parallelism

Finally, an attempt was made to implement what is known in the Kokkos ecosystem as hierarchical parallelism. As in the nested approach, this one combines several instructions, such as `Kokkos::parallel_for` and `Kokkos::parallel_reduce`. Likewise, instead of using a simple 1D range policy, we use a team policy to distribute the workload across several loops. As in the previous section, we use **teams of threads** arranged into a **league**, allowing for more structured parallel execution. The difference this time is that we use **a shared memory space** for the threads within a **team**, known as the team scratch pad memory, which is definitely analogous to CUDA's **shared memory**. We use this scratch pad memory to cache data that can be reused during computations. Below is an example of how this feature can be used. We reorganized the

computation to distribute the workload in a way that leverages the scratch pad memory. Although this version produced accurate numerical results, it did not yield a significant performance improvement over the previous implementations (see the result section), indicating that further investigation is needed, and this remains a work in progress at the time of writing.

```
#pragma once

#include "utils.hpp"

#include <Kokkos_Core.hpp>

#include <concepts>
#include <iostream>
#include <type_traits>

#ifdef KOKKOS_ENABLE_CUDA
#define MemSpace Kokkos::CudaSpace
#define ExecSpace Kokkos::Cuda
#define Layout Kokkos::LayoutLeft
#endif

#ifdef KOKKOS_ENABLE_OPENMP
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::OpenMP
#define Layout Kokkos::LayoutRight
#endif

#ifdef KOKKOS_ENABLE_THREADS
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::Threads
#define Layout Kokkos::LayoutRight
#endif

#ifndef MemSpace
#define MemSpace Kokkos::HostSpace
#define ExecSpace Kokkos::DefaultHostExecutionSpace
#define Layout Kokkos::LayoutRight
#endif
```

```cpp
/**
* @brief Portable P2P inner kernel using Kokkos parallelism
↪  (hierarchical parallelism).
*
* This function calculates pairwise interactions (potential in
↪  this case) between particles based on
* their positions and charges using a hierarchical parallel
↪  execution model provided by the Kokkos library.
* The computation is divided into teams of threads that
↪  process chunks of data, making use of shared
* scratch memory to minimize global memory access and improve
↪  performance. The function handles data
* transfer between host and device, manages synchronization,
↪  and measures the total elapsed time
* for the computation.
*
* @tparam ContainerType The type of container used for the
↪  positions, charges, and potentials. This container must
*         support random access (e.g., `std::vector`,
↪  `std::array`) and store floating-point values.
*
* @param[in] position_x Container of x-coordinates of the
↪  particles on the host.
* @param[in] position_y Container of y-coordinates of the
↪  particles on the host.
* @param[in] charge Container of charge values associated with
↪  each particle on the host.
* @param[out] potential Output container where the computed
↪  potential for each particle is stored on the host.
* @param[in] chunk_size The size of each chunk to be processed
↪  by a team of threads.
* @param[in] vec_size The vector length (to use in the
↪  hierarchical parallel execution).
* @param[out] elapsed_time The elapsed time (in seconds) for
↪  the parallel computation.
*/
template<Container ContainerType>
auto p2p_inner_hierarchical_parallel(ContainerType const&
↪  position_x, ContainerType const& position_y,
```

```
                                        ContainerType const&
                                        ↪  charge,
                                        ↪  ContainerType&
                                        ↪  potential,
                                        ↪  std::size_t
                                        ↪  chunk_size,
                                        std::size_t vec_size,
                                        ↪  float& elapsed_time)
                                        ↪  -> void
{
    // Set-up (aliases)
    using container_type = ContainerType;
    using value_type = typename container_type::value_type;

    using execution_space_type = ExecSpace;
    using memory_space_type = MemSpace;
    using layout_type = Layout;

    using view_vector_type = Kokkos::View<value_type*,
    ↪  layout_type, memory_space_type>;
    using view_matrix_type = Kokkos::View<value_type**,
    ↪  layout_type, memory_space_type>;

    using host_mirror_type = typename
    ↪  view_vector_type::HostMirror;
    using host_mirror_matrix_type = typename
    ↪  view_matrix_type::HostMirror;

    using range_policy_type =
    ↪  Kokkos::TeamPolicy<execution_space_type>;
    using member_type = typename
    ↪  range_policy_type::member_type;

    using scratch_memory_space_type = typename
    ↪  execution_space_type::scratch_memory_space;
    using scratch_pad_view_type = Kokkos::View<value_type*,
    ↪  scratch_memory_space_type>;

    using timer_type = Kokkos::Timer;
```

```
#ifdef LOCAL_VERBOSITY
    // Print execution space details
    std::cout << std::string(40, '-') << '\n';
    execution_space_type ex{};
    std::cout << " - Execution Space = " << ex.name() << '\n';

    // Print memory space details
    memory_space_type mem{};
    std::cout << " - Memory Space = " << mem.name() << '\n';
    std::cout << std::string(40, '-') << '\n';
#endif

    const std::size_t size = position_x.size();
    const std::size_t nb_chunks = (size + chunk_size - 1) /
    ↪  chunk_size;

    // Declaration of the views
    view_matrix_type view_position_x("view_position_x",
    ↪  chunk_size, nb_chunks);
    view_matrix_type view_position_y("view_position_y",
    ↪  chunk_size, nb_chunks);
    view_matrix_type view_charge("view_charge", chunk_size,
    ↪  nb_chunks);
    view_matrix_type view_potential("view_potential", size,
    ↪  nb_chunks);

    // Create host mirrors of device views
    host_mirror_matrix_type host_position_x =
    ↪  Kokkos::create_mirror_view(view_position_x);
    host_mirror_matrix_type host_position_y =
    ↪  Kokkos::create_mirror_view(view_position_y);
    host_mirror_matrix_type host_charge =
    ↪  Kokkos::create_mirror_view(view_charge);
    host_mirror_matrix_type host_potential =
    ↪  Kokkos::create_mirror_view(view_potential);

    // Initialize vectors on host (via mirrors)
    std::size_t local_index{};
    for(std::size_t e = 0; e < nb_chunks; ++e)
    {
```

```
    for(std::size_t i = 0; i < chunk_size; ++i)
    {
        local_index = e * chunk_size + i;
        if(local_index < size)
        {
            host_position_x(i, e) =
            ↪  position_x[local_index];
            host_position_y(i, e) =
            ↪  position_y[local_index];
            host_charge(i, e) = charge[local_index];
        }
        else
        {
            // Zero-padding
            host_position_x(i, e) = 0.;
            host_position_y(i, e) = 0.;
            host_charge(i, e) = 0.;
        }
    }
}

// Deep copy from host mirrors to device views
Kokkos::deep_copy(view_position_x, host_position_x);
Kokkos::deep_copy(view_position_y, host_position_y);
Kokkos::deep_copy(view_charge, host_charge);

std::size_t scratch_mem_size{3 *
↪  scratch_pad_view_type::shmem_size(chunk_size)};

timer_type timer{};

// Actual computation
int level{0};
Kokkos::parallel_for(
  "Extern for",
  range_policy_type(nb_chunks, Kokkos::AUTO,
  ↪  vec_size).set_scratch_size(level,
  ↪  Kokkos::PerTeam(scratch_mem_size)),
  KOKKOS_LAMBDA(const member_type& team_member) {
      const std::size_t e = team_member.league_rank();
```

```
// Cache source data into scratch pad memory
scratch_pad_view_type shared_position_x(team_member.
↪  team_scratch(level),
↪  chunk_size);
scratch_pad_view_type shared_position_y(team_member.
↪  team_scratch(level),
↪  chunk_size);
scratch_pad_view_type
↪  shared_charge(team_member.team_scratch(level),
↪  chunk_size);
if(team_member.team_rank() == 0)
{
    Kokkos::parallel_for(Kokkos::ThreadVectorRange(t
    ↪  eam_member,
    ↪  chunk_size),
                         [&](std::size_t i)
                         {
                             shared_position_x(i) =
                             ↪  view_position_x(i,
                             ↪  e);
                             shared_position_y(i) =
                             ↪  view_position_y(i,
                             ↪  e);
                             shared_charge(i) =
                             ↪  view_charge(i, e);
                         });
}

// Synchronization (to make sure that all the values
↪  are cached)
team_member.team_barrier();

Kokkos::parallel_for(Kokkos::TeamThreadRange(team_me
↪  mber,
↪  size),
                     [&](std::size_t j)
                     {
```

```
const std::size_t local_i =
↪  j % chunk_size, local_e
↪  = j / chunk_size;
const value_type
↪  local_x_target = view_p⌋
↪  osition_x(local_i,
↪  local_e),
                local_y_ta⌋
                ↪  rget =
                ↪  view_p⌋
                ↪  ositio⌋
                ↪  n_y(lo⌋
                ↪  cal_i,
                ↪  local_⌋
                ↪  e);

value_type local_result{0.};

Kokkos::parallel_reduce(
  Kokkos::ThreadVectorRange⌋
  ↪  (team_member,
  ↪  chunk_size),
  [&](std::size_t i,
  ↪  value_type&
  ↪  update_local_result)
  {
      if(j != e *
      ↪  chunk_size + i &&
      ↪  e * chunk_size +
      ↪  i < size)
      {
          const value_type
          ↪  diff_y =
          ↪  shared_positi⌋
          ↪  on_y(i) -
          ↪  local_y_targe⌋
          ↪  t,
```

```
⌋
↪ d⌋
↪ i⌋
↪ f⌋
↪ f⌋
↪ -⌋
↪ x⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ =⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ s⌋
↪ h⌋
↪ a⌋
↪ r⌋
↪ e⌋
↪ d⌋
↪ -⌋
↪ p⌋
↪ o⌋
↪ s⌋
↪ i⌋
↪ t⌋
↪ i⌋
↪ o⌋
↪ n⌋
↪ -⌋
↪ x⌋
↪ ⌋
↪ (⌋
↪ ⌋
↪ i⌋
↪ ⌋
↪ )⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ -⌋
↪ ⌋
↪
↪ ⌋
↪ ⌋
↪ l⌋
```

```
                                        update_local_resu⌋
                                        ↪ lt
                                        ↪ +=
                                          shared_charge(i⌋
                                          ↪ ) /
                                          ↪ std::sqrt(d⌋
                                          ↪ iff_y *
                                          ↪ diff_y +
                                          ↪ diff_x *
                                          ↪ diff_x);
                                    }
                                },
                                local_result);

                            view_potential(j, e) =
                            ↪ local_result;
                        });
  });

// Final reduction (over all the chunks)
Kokkos::parallel_for(
  "Extern for", range_policy_type(size, Kokkos::AUTO),
  ↪ KOKKOS_LAMBDA(const member_type& team_member) {
      const std::size_t j = team_member.league_rank();
      value_type result{0.};

      Kokkos::parallel_reduce(
        Kokkos::TeamVectorRange(team_member, nb_chunks),
        [&](std::size_t e, value_type& inner_update) {
        ↪ inner_update += view_potential(j, e); },
        ↪ result);

      Kokkos::single(Kokkos::PerTeam(team_member), [&]() {
      ↪ view_potential(j, 0) = result; });
  });

// We make sure that everything is completed before moving
↪ on...
Kokkos::fence();
```

```
    elapsed_time = timer.seconds();

    // Deep copy device view to host mirror
    Kokkos::deep_copy(host_potential, view_potential);

    // Retrieve values
    for(std::size_t j = 0; j < size; ++j)
    {
        potential[j] = host_potential(j, 0);
    }
}
```

1.3.4 Vectorization with Kokkos via Kokkos SIMD

Additionally, we implemented a version that uses the Kokkos SIMD API (still experimental at the time of writing), which provides tools similar to those found in xsimd, acting as a set of wrappers for **SIMD intrinsics**. Unlike previous approaches, this version is designed to run **only on the CPU** but it is portable across **different CPU architectures**, though it supports fewer instruction sets compared to xsimd. This feature of Kokkos allows us to write **portable vectorized** code specifically for the CPU. However, during the internship, we did not manage to develop a version that could run on both the GPU and the CPU with vectorization (again this is still a work in progress). The versions presented in previous sections can run on the GPU via CUDA and on the CPU via OpenMP, for example, but not with **explicit vectorization** like xsimd. Nonetheless, in the next section, we demonstrate that this implementation achieved performance comparable to that of xsimd (the code is also very similar).

```
#pragma once

#include "utils.hpp"

#include <Kokkos_Core.hpp>
#include <Kokkos_SIMD.hpp>

#include <concepts>
#include <iostream>
#include <type_traits>
```

```
/**
* @brief P2P inner kernel using SIMD vectorization via
↪  Kokkos::simd API (still experimental
* at the time of writing).
*
* This function calculates pairwise interactions (potential in
↪  this case) between particles based
* on their positions and charges. It leverages SIMD
↪  vectorization via the SIMD API offered by Kokkos
* to process multiple particles simultaneously. The function
↪  uses SIMD to handle large portions of the
* data in parallel, while a scalar fallback handles any
↪  remaining particles that don't fit into a full
* SIMD batch. We use SIMD batch logical masks to remove
↪  self-interactions (acting as a filter).
*
* @tparam ContainerType The type of container used for the
↪  positions, charges, and potentials. This container must
*         support random access (e.g., `std::vector`,
↪  `std::array`) and store floating-point values.
*
* @param[in] position_x Container of x-coordinates of the
↪  particles.
* @param[in] position_y Container of y-coordinates of the
↪  particles.
* @param[in] charge Container of charge values associated with
↪  each particle.
* @param[out] potential Output container where the computed
↪  potential for each particle is stored.
*
*/
template<Container ContainerType>
 auto p2p_inner_kokkos_simd(ContainerType const& position_x,
 ↪  ContainerType const& position_y,
                            ContainerType const& charge,
                            ↪  ContainerType& potential) ->
                            ↪  void
 {
     // Set-up (aliases)
     using container_type = ContainerType;
```

```
using value_type = typename container_type::value_type;
using ksimd_type =
↪  Kokkos::Experimental::native_simd<value_type>;
static constexpr std::size_t batch_size =
↪  ksimd_type::size();

const std::size_t size = position_x.size();
const std::size_t vec_size = size - size % batch_size;

ksimd_type vec_local_x_target, vec_local_y_target,
↪  vec_contribution, vec_local_result;

// Initialize logical mask
ksimd_type vec_next_target_indices([](std::size_t i) {
↪  return i; }), vec_source_index;

// Loop through target data (vectorized part)
for(std::size_t i = 0; i < vec_size; i += batch_size,
↪  vec_next_target_indices += batch_size)
{
    vec_local_x_target.copy_from(position_x.data() + i,
    ↪  Kokkos::Experimental::simd_flag_default);
    vec_local_y_target.copy_from(position_y.data() + i,
    ↪  Kokkos::Experimental::simd_flag_default);

    vec_contribution = 0.;

    // Loop through source data
    for(std::size_t j = 0; j < size; ++j)
    {
        const ksimd_type vec_diff_x = vec_local_x_target
        ↪  - position_x[j];
        const ksimd_type vec_diff_y = vec_local_y_target
        ↪  - position_y[j];

        // Apply logical mask to skip the case 'i = j'
        vec_source_index = j;
        vec_local_result = charge[j] /
        ↪  Kokkos::sqrt(vec_diff_x * vec_diff_x +
        ↪  vec_diff_y * vec_diff_y);
```

```
            where(vec_next_target_indices ==
            ↪  vec_source_index, vec_local_result) = 0.0;

            vec_contribution += vec_local_result;
        }

        vec_contribution.copy_to(potential.data() + i,
        ↪  Kokkos::Experimental::simd_flag_default);
    }

    value_type result{0.};

    // Loop through target data (remaining scalar part)
    for(std::size_t i = vec_size; i < size; ++i)
    {
        const value_type local_target_x = position_x[i],
        ↪  local_target_y = position_y[i];
        result = 0.;

        // Loop through source data
        for(std::size_t j = 0; j < size; ++j)
        {
            if(i != j)
            {
                const value_type diff_x = local_target_x -
                ↪  position_x[j], diff_y = local_target_y -
                ↪  position_y[j];
                result += charge[j] / std::sqrt(diff_x *
                ↪  diff_x + diff_y * diff_y);
            }
        }

        potential[i] = result;
    }
}
```