Future Generation Computer Systems 169 (2025) 107802

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Effect of implementations of the N-body problem on the performance and portability across GPU vendors

Rodrigo A.C. Bartolomeu [a], René Halver [a], Jan H. Meinke [a,*], Godehard Sutmann [a,b]

[a] Jülich Supercomputing Centre (JSC), Institute for Advanced Simulation, Forschungszentrum Jülich, 52425, Jülich, Germany
[b] ICAMS, Ruhr-Universität Bochum, 44801, Bochum, Germany

ARTICLE INFO

Keywords:
GPU
High Performance Computing
N-body problem
Programming models
Performance portability

ABSTRACT

Since Aurora entered the TOP500 list in November 2023, the top ten systems saw some shifts in the ratio of GPU vendors represented. With each vendor supplying their own preferred programming models for their hardware, it becomes relevant to compare the portability of these models on other hardware platforms. For the present paper we implemented the N-body problem with different optimizations using native and portable programming frameworks. For each of those we determined the best performing optimized version on one target architecture and compared the performance achieved for each platform.

1. Introduction

About 20 years ago Nvidia introduced CUDA for programming their general-purpose graphics processing units (GPGPU). Since then many simulation codes have included support for offloading either parts or even all of their work to GPUs. For many years, Nvidia dominated the GPGPU and especially the high-performance computing (HPC) GPU market and a majority of developers used CUDA to write their GPU programs. Nvidia fostered this development with its tools, support, documentation and user community events. Those developers that wanted to use a single code base for their CPU and their GPU code were encouraged by Nvidia to use OpenACC (a pragma-based model) and more recently standard-based programming of GPUs using, e.g., Fortran's "**do** concurrent" or the parallel algorithms introduced with C++17. There were cross-vendor attempts to establish alternatives such as OpenCL, but they never became strong competitors.

At the same time, developers needed both flexible and portable, in particular performance portable, solutions [1] to avoid vendor lock-in. Several C++ frameworks, e.g., Alpaka [2], Kokkos [3], Phalanx [4], Raja [5], Thrust [6], and UPC [7] were developed to provide abstractions. They can be compiled for different GPUs and CPUs and promise not only functional portability but also performance portability across CPUs and GPUs from different vendors.

When the Department of Energy announced[1] that the first exascale system Frontier, hosted at Oak Ridge National Laboratory, would use AMD MI250 GPUs, vendor-portable solutions took on a new importance. AMD's GPUs need to be programmed with HIP or OpenMP. To take advantage of Frontier, codes that had previously been ported to GPUs using CUDA or OpenACC needed to be adapted. In November 2023, Argonne National Laboratory's new system, Aurora, entered the TOP500 list as number 2. It is based on Intel GPU Max and CPUs and uses SYCL as its preferred programming model. Again, developers that used CUDA or HIP before, needed to modify their code to run it on the new machine. Developers that used a portable programming model, on the other hand, had to wait until the model became available on the new machine, but ideally did not have to modify their own codes. With the TOP500 list published in November 2024, the majority of GPU-accelerated systems in the top 10 systems contain GPUs from AMD. This underlines the importance of portable code solutions to avoid a vendor lock-in and to be ready for changes in the HPC accelerator landscape in the future.

With the development of new GPUs from different vendors, it is an ongoing monitoring process, how well portable frameworks perform and how versatile they can be applied. The initial question is, whether a portable framework is already available and functional on a given architecture. While the notion of portability between different platforms is based on successful program execution (and therefore easy to define), the definition of performance portability and a metric to measure it, is more involved.

A common approach to assess the performance and capabilities of portable programming models is to use mini-apps or even full applications to gather relevant information [1,8] either by porting one

* Corresponding author.
E-mail addresses: r.bartolomeu@fz-juelich.de (R.A.C. Bartolomeu), r.halver@fz-juelich.de (R. Halver), j.meinke@fz-juelich.de (J.H. Meinke), g.sutmann@fz-juelich.de (G. Sutmann).

[1] U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL, May 7, 2019.

https://doi.org/10.1016/j.future.2025.107802
Received 6 December 2024; Received in revised form 13 February 2025; Accepted 1 March 2025
Available online 12 March 2025

into various performance portable programming models [9–12] or by porting many mini-apps into one performance portable programming model and compare the findings to other available versions [13]. In this paper we focus on the N-body problem.

The N-body problem has been popular on GPUs from the beginning [14]. It has been included with the examples in the CUDA SDK since version 1.1 and there are implementations available in other programming models as well emphasizing different features of the N-Body problem [15,16]. In the present paper we implement an N-body particle dynamics code with gravitational interactions with different portability solutions. In contrast to miniBUDE, e.g., used in [8,16] that focuses on docking and calculates the energy of the configuration, the codes used in this paper compute the force on each particle and perform an integration of the equation of motion. The programs are written in a straightforward manner as might be used by a scientific programmer, for example, a Ph.D. student in physics. Our goal is to provide scientist with some guidance and answer questions such as which programming model can be used on which platform? What does code written in this model look like? How much performance do I loose in comparison to the same code written in a less portable programming model?

The present work is a continuation of our work presented in [17]. We include results for additional programming models and GPU platform combinations. We added an implementation for HIP that uses shared memory and two implementations for SYCL that allow us to control the distribution of work (SYCL (nd)) and also let us take advantage of shared memory (SYCL (s)). Furthermore, we investigate the effect of typical algorithmic choices in the implementation of the N-body force calculation on performance and present some results running the programs on CPUs where possible. It presents a first step towards a more general portability benchmark framework.

2. The N-body problem

The N-body problem is an important problem class and has been listed as one of the seven original dwarfs in Ref. [18] each of which represents a compelling use case worth studying with its own challenges.

Many scientific fields require the computation of forces between pairs of interacting particles, e.g., gravitational forces in astrophysics, electrostatic forces in charged systems, van der Waals forces for modeling atomic systems, etc.

In this paper, we use the gravitational potential between particles,

$$U(d_{ij}) = -G\frac{m_i m_j}{d_{ij}}, \tag{1}$$

where $G = 6.6743 \cdot 10^{-11}\,\frac{\mathrm{m}^3}{\mathrm{kg\ s}^2}$ is the gravitational constant $d_{ij} = |\mathbf{r}_{ij}|$ is the distance between the two particles i and j, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ and m_i and m_j are the masses of the respective particles. Given a potential $U(d_{ij})$, the force $\mathbf{F}_{ij}$ between the particles i and j is given as

$$\mathbf{F}_{ij} = -\nabla U(d_{ij}) = -G\frac{m_i m_j}{d_{ij}^3}\mathbf{r}_{ij}. \tag{2}$$

The total force acting on particle i is then given by the sum over all other particles

$$\mathbf{F}_i = \sum_{j \neq i} \mathbf{F}_{ij}. \tag{3}$$

To obtain the trajectory of each particle, Newton's equations of motion

$$\dot{\mathbf{x}}_i = \mathbf{v}_i \quad , \quad \dot{\mathbf{v}}_i = \mathbf{a}_i = \frac{\mathbf{F}_i}{m_i}, \tag{4}$$

are integrated, where the dot-notation means the time derivative, $\mathbf{a}_i$ is the acceleration and m_i the mass of particle i.

The gravitational N-body problem (the 4th dwarf in Ref. [18]) is a representative of an algorithmic method which includes long-range interactions in open-boundaries and includes summation over all particle pairs in the system to compute the forces on each particle, which makes the natural computational complexity $\mathcal{O}(N^2)$. To avoid the quadratic complexity, more efficient methods, like the Barnes–Hut tree method or the fast multipole method (FMM), which reduce the complexity to $\mathcal{O}(N \log(N))$ or $\mathcal{O}(N)$ can be used. Since these methods have a high level of complexity and their performance strongly depends on the implementation and optimization, we focus here on a simple implementation, based on the all-pairs computation of energies and forces, which isolates the performance outcome from algorithmic issues.

3. Programming models

In this section we provide brief summaries of the programming models used in this study.

CUDA (*C*ompute *U*nified *D*evice *A*rchitecture) is an application programming interface (API) for programming GPU devices from Nvidia. CUDA is based on C++ with Fortran support provided by the NVHPC SDK. It provides functions for allocating data on the GPU and transferring data between the GPU and the host. This data can be processed with kernels, which are executed on the GPU.

For kernel execution, threads are organized in blocks, and blocks are organized in a grid. The maximum number of threads in a block and blocks in a grid is determined by the specific hardware. GPUs provide a hierarchical memory layout that can be utilized with CUDA, distinguishing between *global* and *shared* memory. Global memory is the main memory of the device, while shared memory is similar to a programmable L1 cache shared between threads of the same block.

CUDA kernels are `void` functions starting with the `__global__` keyword. Kernels are launched with a special syntax that passes the launch configuration (shape of the grid and shape of a thread block) within triple chevrons (`<<<...,...>>>`). The launch configuration determines the number of threads (total number of threads is equal to the number of blocks in a grid multiplied by the number of threads in a block). Threads are characterized by their lightness and a GPU can efficiently manage millions of threads.

Within a kernel the built-in variables `threadIdx`, `blockIdx` and `blockDim` allow each thread to determine its global and local (within a thread block) position in up to three dimensions. Each thread in a launch executes the same kernel, but may follow different paths depending on its position. Additionally, CUDA kernels can call functions having the `__device__` attribute in the function declaration. Often such functions are marked also with the `__host__` attribute to indicate that they can be called from the host program as well.

HIP is a C++ API closely mimicking the CUDA API, but replacing the cuda prefix with hip. Kernels are defined and called in the same way as in CUDA. A program using the HIP API can be compiled for Nvidia GPUs using nvcc and for AMD GPUs using hipcc. In addition, efforts are on the way to enable compilation of HIP programs for Intel GPUs using the Intel Level Zero API [19].

Kokkos [3,20] is a C++ based API for writing performance-portable code for HPC hardware architectures. It introduces abstraction layers for code execution and data management, enabling code execution on diverse hardware platforms. For this purpose, it utilizes backends, such as CUDA, HIP, OpenMP, or SYCL, which allow for the compilation and execution of the code on various architectures without the need for source code alterations.

The Kokkos programming model allows for multiple execution and memory spaces in a single program in addition to the default execution space and default host execution space that are defined during initialization based on the enabled and available spaces. This makes it possible to distribute work across different parallel devices, for example, CPUs and GPUs. It is fairly straightforward to use compile time switches to use different execution spaces, e.g.,

```
// definitions for Kokkos spaces
using HostSpace = Kokkos::HostSpace;

#ifdef PPBS_KOKKOS_CUDA
```

```
using ExecutionSpace = Kokkos::Cuda;
using DeviceSpace = Kokkos::CudaSpace;
const std::string KOKKOSTYPE = "Cuda";
#elif PPBS_KOKKOS_HIP
using ExecutionSpace = Kokkos::HIP;
using DeviceSpace = Kokkos::HIPSpace;
const std::string KOKKOSTYPE = "HIP";
...
```

Kokkos introduces a parallel_for loop. The loop takes a range including an execution space and a function object usually implemented using a KOKKOS_LAMBDA that contains the code to be executed for each element of the range as arguments:

```
Kokkos::parallel_for(
  Kokkos::RangePolicy<ExecutionSpace>
  (0, particles.n),
  KOKKOS_LAMBDA(const int &i) {
    // Perform calculation ...
});
```

In addition it provides specialized implementations for reductions (parallel_reduce) and scans (parallel_scan).

The macro KOKKOS_LAMBDA maps to a host-device lambda function for CUDA and HIP backends

```
__host__ __device__ [](){};
```

and to a regular C++ lambda function for other backends.

SYCL is a C++ API developed as an open standard by the Khronos Group. SYCL programs can run on CPUs, GPUs, FPGAs, and other accelerators. It can launch basic data parallel kernels similar to Kokkos or the pSTL or explicit nd-range kernels, where each item belongs to an execution group comparable to thread blocks in CUDA and HIP. Each kernel is submitted to a queue, and data dependencies between kernels determine the order of execution. If there are no dependencies, they may run in parallel. A program using SYCL can have multiple queues, e.g., for different devices. The call for a basic data-parallel kernel is very similar to the one in Kokkos:

```
q.submit([&](handler& h){
  // Request access to data
  accessor x(x_buf, h, read_only);
  ...
  h.parallel_for(N, [=](auto id){
    // Perform calculation
  });
});
```

The basic data-parallel kernel leaves the distribution of work items to the runtime. Work items can be executed in any order and no communication between work items is possible. A second version of the parallel_for statement takes an nd-range, which provides a work-groups size in addition to the problem size. The problem size must be an integer multiple of the work group size. A work group is equivalent to a thread block in CUDA. Threads within a work group can synchronize easily and make use of local memory (shared memory in CUDA).

OpenACC provides a set of compiler directives and an API to map parallelizable code regions to accelerators, such as GPUs or multi-core CPUs for C, C++, and Fortran. It was first released in 2011 combining ideas from CAPS, Cray, PGI, and NVIDIA to port existing CPU codes to GPUs with minimal changes to the code base. It provides high-level directives, for example, #pragma acc kernel and #pragma acc parallel loop, to indicate such regions:

```
#pragma acc kernels
for (auto i = 0; i < N; ++i){
    for (auto j = 0; j < N; ++j){
        if (i != j) {
        ...  // do some work
        }
    }
}
```

The compiler is free to generate GPU kernels from the two nested loops in any way it sees fit. A compiler that does not support OpenACC will just ignore the pragmas and compile the code as a regular C++ program.

OpenACC also provides directives and clauses to manage the data flow between the host and the device, which is often essential to achieve good performance. OpenACC's approach is considered more descriptive than prescriptive compared to OpenMP.

OpenMP was the inspiration for OpenACC. Since its first release, in 1997, OpenMP has been one of the most popular ways to provide parallelism for shared memory CPU codes. Just like OpenACC it provides compiler directives that can be ignored by a compiler that does not know about them, an API, and libraries to generate parallel regions of a code as well as directives for managing the data flow between host and device. It is available for C, C++, and Fortran. Support for accelerators was included with the 4.0 release in 2013 [21]. While it took a long time to implement all of the features in the 4.0 standard, OpenMP offload to target devices is by now fully supported by different compilers. In GCC this is achieved using *libgomp*, in LLVM, *libomptarget*, and NVHPC internally uses the Thrust library, which leads to the linking of OpenACC libraries. Using OpenMP, the above code can be written as

```
#pragma omp target teams distribute parallel for
for (auto i = 0; i < N; ++i){
    for (auto j = 0; j < N; ++j){
        if (i != j) {
        ...  // do some work
        }
    }
}
```

An OpenMP capable compiler can offload this code block to a device and distribute the work of the outer loop over multiple teams and multiple threads within a team.

pSTL. C++17 introduced execution policies and made them an optional argument for several standard library algorithms. Execution policies provide hints to the compiler and the runtime if an algorithm can run in parallel. The set of algorithms that supports execution policies comprises the parallel Standard Template Library (pSTL). As of C++23, there is no standard way to specify *where* the algorithm should run. The default assumption is that it runs on the same device as the rest of the program.

With its release of the HPC SDK in 2020, Nvidia introduced a flag –stdpar to its compiler, that allows it to take pSTL algorithms with a parallel execution policy and run them on a CUDA capable GPU. For many C++ containers, data management can be handled similarly to the way we describe for our CUDA implementation. If you use –stdpar the default allocator is changed to one that uses managed memory making the data of many standard containers accessible on the GPU. For systems that use an A100 or newer Nvidia GPU and a kernel with heterogeneous memory management (HMM) enabled, all memory allocated by the system becomes accessible on the GPU. This makes even more data structures and pSTL algorithms available for execution on a GPU. With Grace Hopper Nvidia enables this feature in hardware [22].

Intel chose a different way to bring pSTL algorithms to devices. In addition to its regular implementation of the STL, Intel supports a second implementation of the STL with oneDPL. A vendor implementing the STL is allowed to define additional execution policies. oneDPL introduces a way to create execution policies that incorporate SYCL queues. A parallel algorithm using such an execution policy will offload the work to the device associated with the queue. For convenience, oneDPL also provides dpcpp_default, which executes the algorithm using the default queue on the default device. The device chosen can be controlled using environment variables. The Intel compiler provides the

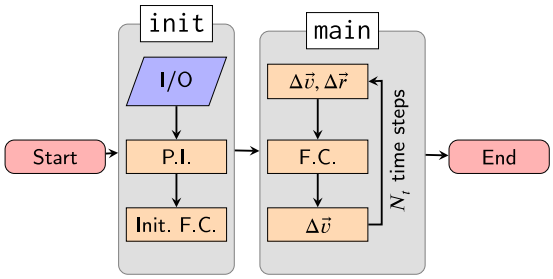

Figure 1. Sequence of operations and program flow within the N-body problem. The main phase of the program is repeated for N_t iterations, each computing a discrete time step of size dt. P.I.: particle initialization; F.C.: force computation; $\Delta\vec{r}$: propagation of positions in Velocity Verlet; $\Delta\vec{v}$: propagation of half step velocities in Velocity Verlet integrator.

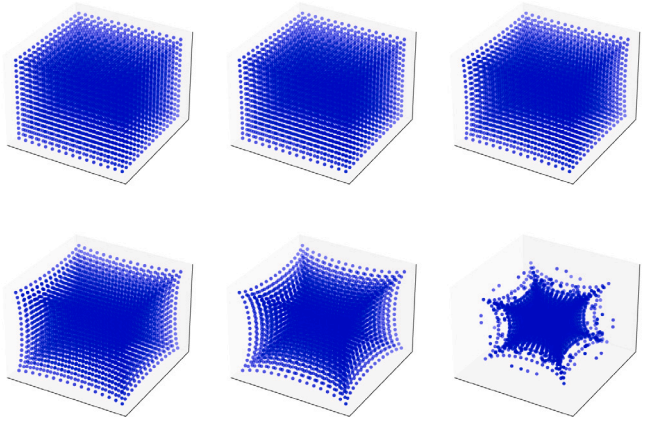

Figure 2. Snapshots from the simulation. The panels show six equally spaced (in time) snapshots of the system. We start by placing masses on the vertices of a cubic lattice with a lattice spacing of one meter. The cube contains $16 \times 16 \times 16$ point masses.

command line flag `-fsycl-pstl-offload`, which maps *all* algorithms using the execution policy `std::execution::par_unseq` to the corresponding oneDPL algorithms with the `dpcpp_default` execution policy.

Starting with ROCm 6.1 AMD added HIPSTDPAR (aka roc-stdpar) to its stack. Similarly to `-stdpar` for Nvidia and `-fsycl-pstl-offload`, the flag `--hipstdpar` offloads all algorithms with an execution policy set to `par_unseq` to the GPU. Unfortunately, ROCm 6.1 is not available on our MI250 test nodes, yet, and we were therefore unable to test it. Instead we used AdpativeCpp as a vendor independent solution.

AdaptiveCpp maps the pSTL algorithms to one or more calls to SYCL kernels with a tight integration between compiler and runtime [23]. It supports offloading to AMD, Intel, and Nvidia GPUs in a single binary.

4. Implementations

The studied implementations calculate explicitly the pair-wise interactions between N particles in the system, resulting in an $\mathcal{O}(N^2)$ complexity. Fig. 1 shows the functions that need to be implemented: *force calculation*, *position propagation*, and *velocity propagation*. Compact, self-contained functions like this are the first candidates for porting to GPU kernels.

The first computation of the force is no different from the rest but needs to be done before entering the loop since it is required by the Velocity Verlet method, which is used here, to integrate the equations of motion [24].

The simulation setup starts by arranging particles on a regular cubic grid of length L (c.f. Fig. 2). The total number of particles is $N = L^3$. Most of our implementations perform the initialization on the GPU. *All* of our implementations offload the force calculation, position propagation, and velocity propagation to the GPU. After the initial force calculation, the data usually resides in GPU memory, i.e., memory transfer does not play a critical role in our measurements.

Keeping the data on the GPU as long as possible is, in general, an important factor for an efficient GPU implementation. The force calculation includes a double-loop over all particles and is compute intensive with a very high arithmetic intensity (operations per loaded byte), while the two propagation kernels consist of single loops with a low arithmetic intensity. Nevertheless, there is still a benefit for the performance of the application when executing these functions on the GPU since data transfer is avoided.

We use C++ and various frameworks. Other language choices, e.g., Fortran, Julia, or Python, are possible, but beyond the scope of the present paper. For an overview of the support of various vendors of the most popular frameworks see [25].

In the following, we briefly describe the common and unique features of the different implementations.

Our **CUDA** implementation uses CUDA's managed memory and C++ allocators. They define the way memory is assigned to a data structure when it requests memory via `new`. Our allocator uses `cudaMallocManaged` to reserve memory that can be accessed from the CPU *and* the GPU. To use it, we define `CudaVector` with the `using` keyword:

```
template <typename T>
using CudaVector = std::vector<T,
                               CudaAllocator<T>>;
```

Now we can use `CudaVector` just like any other `std::vector`.

The particle initialization kernel computes the location of the particles and stores them in the respective buffers. Masses and velocities are directly initialized during instantiation using our `CudaVector`. The following line allocates memory for N elements and sets their value to MASS:

```
CudaVector<fp> m(N, MASS);
```

Using allocators with standard containers instead of the frequently encountered direct use of pointers with `cudaMalloc` and `cudaFree` frees the programmer from dealing with the memory management avoiding many common pitfalls and security issues.

Each of the kernels works on the data stored in the managed memory, which, since it is not accessed from the host, resides on the devices. The managed memory is only accessed from the host, whenever the total sum of forces is computed to check for correctness of the results.

Function calls require the passing of the launch configuration within triple chevrons (`<<<...,...>>>`). The call to compute the forces is as follows

```
forceComputation<<<forceGridSize,
                   FORCE_THREAD_SIZE>>>(
  N,
  x.data(), y.data(), z.data(), m.data(),
  fx.data(), fy.data(), fz.data());
```

We implemented two variants for the force computation kernel. Each thread calculates the force on a particle (c.f. Eq. (3)). In the *base implementation*, positions and masses of the interaction partners are loaded directly from global memory.

The *advanced implementation* uses shared memory. Instead of loading the data directly from global memory, each thread in a thread block copies data from the global memory into buffers residing in shared memory. This utilizes coalesced memory access and minimizes memory loads from global memory. Since shared memory size is limited, we split the loop depending on the size of the shared memory. For each loaded chunk of data, the force contributions of the particles stored in shared memory are added to the current total. This is repeated until all interactions are computed.

Our **HIP** implementation does not take advantage of the allocator implemented for the CUDA version, yet, but uses pointers directly.

The kernels of our HIP implementation are equivalent to the CUDA implementation.

Our **Kokkos** code uses a structure of arrays to store positions, velocities, masses, and forces containing for each parameter a Kokkos::View<T*, DeviceType> element. Depending on the DeviceType, defined as

```
using
DeviceType = Kokkos::Device<ExecutionSpace,
                            DeviceSpace>;
```

the data will be allocated using host or device memory. A Kokkos::View also adjusts its data layout to optimize data access for a particular type of device.

All operations on the data are performed in the same memory space, so that no data transfer is necessary.

For **SYCL** we have three different implementations: the first one uses a basic data-parallel kernel. The runtime determines how and in which order the operations are performed. The second implementation uses an nd-range kernel, where we specify the size of a work group instead of leaving it to the runtime. While this may hinder performance portability, it also gives us more control and allows us to tune the distribution of work in the same way we can do using CUDA or HIP. The third implementation uses local memory (called shared memory for CUDA) to improve data reuse.

Our **OpenACC** and **OpenMP** implementations are nearly identical. All compute-specific calculations reside inside a `data` directive-scope reducing communication between host and device. Parallel loop regions are marked for offloading with `#pragma acc parallel loop gang vector` for OpenACC and `#pragma omp target teams distribute parallel for` for OpenMP.

For our implementation using C++'s parallel STL (**pSTL**) we used std::for_each_n to implement the force calculation loop:

```
std::for_each_n(EP, iterator.begin(),
    iterator.size(),
  [=, particleList = particleList.get()](auto i)
  { /* Perform calculation */ });
```

5. Optimizations

To compute the total force acting on a single particle i, as shown in Eq. (3) the most basic way to implement is to use two nested loops with a condition, that the summation only takes place if i is not equal to j to avoid zero distances. A straightforward implementation in C++ could look like this:

```
for (auto i{0u}; i < N; ++i){
    for (auto j{0u}; j < N; ++j)
    {
        if (i != j)
        {
            // force computation here
        }
    }
}
```

This implementation performs a comparison for each pair of atoms i and j and causes branch divergence on massively parallel architectures. Therefore ways to avoid the branching should result in improved performance. In this paper we present two ways to tame the resulting division by zero without explicit if-condition, i.e., introducing a smoothing term to the necessary d_{ij}-division

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{d_{ij}^3 + \varepsilon}\mathbf{r}_{ij},\ \varepsilon \ll \min(d_{ij}),$$

where the value of the smoothing factor needs to be both sufficiently small for not impacting the correctness of the results and being large enough to be different from zero in the machine representation, i.e., close to the machine accuracy of the chosen data type. The second alternative is adding the Kronecker delta δ_{ij} defined as:

$$\delta_{ij} = \begin{cases} 1 & : \mathrm{i} = \mathrm{j} \\ 0 & : i \neq j \end{cases}$$

to the partial forces:

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{d_{ij}^3 + \delta_{ij}}\mathbf{r}_{ij}$$

This avoids the division by zero and is negated by the subsequent multiplication with the distance, being 0 in the case, where i equals j. While the latter alternative ensures the correctness of the results, there might be a runtime impact since it requires the comparison of i and j to compute δ_{ij}, even if directly cast into an integer type.

In physics texts the formula for the partial forces is often written as:

$$\mathbf{F}_{ij} = -G\frac{m_i m_j}{d_{ij}^2}\frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|_2},$$

to emphasize that the vector $\mathbf{r}_{ij}$ is divided by its length to create a unit vector pointing in the right direction. $\|\mathbf{r}_{ij}\|_2$ is the Euclidean norm of the distance vector between the particles, i.e., the distance d_{ij}. Thus a first implementation of the force computation in C++ might look like this:

```
for (auto i{0u}; i < N; ++i){
  double ftmp[i][0] = 0.0;
  double ftmp[i][0] = 0.0;
  double ftmp[i][0] = 0.0;
  for (auto j{0u}; j < N; ++j){
    if (i == j) continue;
    double d[0] = r[i][0] - r[j][0];
    double d[1] = r[i][1] - r[j][1];
    double d[2] = r[i][2] - r[j][2];
    double dist = sqrt(d[0] * d[0]
                     + d[1] * d[1]
                     + d[2] * d[2]);
    double F_ij = -G * m[i] * m[j]
                  / (dist * dist) / dist;
    ftmp[i][0] += d[0] * F_ij;
    ftmp[i][1] += d[1] * F_ij;
    ftmp[i][2] += d[2] * F_ij;
  }
  f[i][0] += ftmp[i][0];
  f[i][1] += ftmp[i][1];
  f[i][2] += ftmp[i][2];
}
```

Alternatively using a smoothing factor ε leads to the following code (only showing changes):

```
for (auto i{0u}; i < N; ++i){
  [...]
  for (auto j{0u}; j < N; ++j){
    // remove the if-statement
    [...]
    double dist = sqrt(d[0] * d[0]
                     + d[1] * d[1]
                     + d[2] * d[2] )
                     + epsilon;
    double F_ij = -G * m[i] * m[j]
                  / (dist * dist) / (dist);
    [...]
  }
  [...]
}
```

The use of the Kronecker delta δ_{ij} would lead to this code:

```
for (auto i{0u}; i < N; ++i){
  [...]
  for (auto j{0u}; j < N; ++j){
```

```
    // remove the if-statement
    [...]
    double dist = sqrt(d[0] * d[0]
                     + d[1] * d[1]
                     + d[2] * d[2] )
                     + (double)(i == j);
    double F_ij = -G * m[i] * m[j]
                  / (dist * dist) / (dist);
    [...]
  }
  [...]
}
```

This computation will be executed within the nested loop, which might use any of the previously mentioned ways to avoid the computation of the self-force (i equals j). It can be seen that there are simple ways to improve the performance of this code. The most direct one is to take the multiplication with $-G * m_i$ out of the inner loop, since that factor is invariant within the inner loop, e.g.

```
for (auto i{0u}; i < N; ++i){
  [...]
  for (auto j{0u}; j < N; ++j)  {
    [...]
    ftmp[i][0] += d[0] * F_ij;
    ftmp[i][1] += d[1] * F_ij;
    ftmp[i][2] += d[2] * F_ij;
  }
  f[i][0] += -G * m[i] * ftmp[i][0];
  f[i][1] += -G * m[i] * ftmp[i][1];
  f[i][2] += -G * m[i] * ftmp[i][2];
}
```

Since divisions are expensive instructions, a further optimization consists of reducing the number of required divisions as much as possible. Reformulating the problem leads to

$$\mathbf{F}_{ij} = -Gm_i \frac{m_j}{d^2 \cdot d}\mathbf{r}_{ij},$$

which can be formulated in the code as follows:

```
for (auto i{0u}; i < N; ++i){
  [...]
  for (auto j{0u}; j < N; ++j){
    [...]
    double d2 = d[0] * d[0]
              + d[1] * d[1]
              + d[2] * d[2];
    double F_ij = m[j] / (d2 * sqrt(d2));
    [...]
  }
  [...]
}
```

Some architectures offer implementations for a fast computation of an inverse square root ($1/\sqrt{x}$), which can make the following formulation of the code advantageous

```
for (auto i{0u}; i < N; ++i){
  [...]
  for (auto j{0}; j < N; ++j){
    [...]
    double d2 = d[0] * d[0]
              + d[1] * d[1]
              + d[2] * d[2];
    double invD = 1.0 / sqrt(d2);
    double F_ij = m[j] * invD * invD * invD;
    [...]
  }
  [...]
}
```

Both optimization, i.e., how to avoid division by zero and improving the calculation of the force contribution are independent and can be combined. The impact of these implementation details is dependent on both the compiler and the platform the code is executed on, since compiler optimizations and hardware instruction sets impact their benefit.

Further possibilities for improvement could be found in the way variables are declared or minimization of load/store operations by, e.g., not storing results of pre-computations. Since these are more influenced by compiler optimizations and hardware capabilities and would result in an increasing number of possible combinations of optimizations, they are not discussed in the present paper. Furthermore, hardware-specific optimizations are not considered here since one goal of the present work is to have a single code base for all available platforms.

6. Baseline for efficiency

To determine the efficiency of an implementation we need a baseline. Architectural efficiency is measured relative to the theoretical peak performance or the practical peak performance based on the architecture specific throughput of the required operations [26], e.g., multiplications, divisions, and square roots or alternatively a roofline analysis. The application efficiency on the other hand is relative to the problem specific performance measure of the fastest available implementation [27].

In our previous paper [17], we presented both architectural and application efficiencies, but since users are mostly interested in the time to solution, we focus on application efficiency in this paper based on the achieved computation rate of pairwise interactions per second (throughput), given as $\frac{\text{GInt}}{\text{s}} = 10^9 \frac{\text{Int}}{\text{s}}$. In contrast to runtime, throughput can be compared independently of system size. All measurements presented are calculated from the average runtime of a time step taken from 20 consecutive time steps for each system size. Standard deviations are omitted from the plots as they are smaller than the symbols.

We compare the best achieved throughput of each implementation across all system sizes to the best achieved throughput between all programming models capable of running on that platform. Since the Nvidia platforms are currently the only ones on which all programming models can run, we chose the GH200 as the reference point to decide which set of optimizations described in Section 5 are used in each programming model. The reasoning behind this is that it allows a choice of implementation for each programming model and then use the same code across all of the different platforms for comparison. By deciding to use the best achieved performance on a given platform the resulting relative performance is limited to 100%, i.e., the fastest measured performance.

Defining a good baseline for performance portability is not a trivial task and several metrics have been used in the literature [28–31].

To compare the portability quality of the programming models we use $\overline{\%}$ (c.f. (5)), which represents the mean application efficiency, with respect to the application throughput achieved by the best implementation performance measured in $\frac{\text{GInt}}{\text{s}}$. To account for the fact that two Nvidia platforms were tested, while only one platform by AMD and Intel were used, we introduced a weighted average to balance the vendor based results. If an implementation cannot be executed on a platform, its throughput is zero.

$$\overline{\%} = \frac{\frac{\%_{\text{A100}} + \%_{\text{GH200}}}{2} + \%_{\text{MI250}} + \%_{\text{Max1100}}}{3}. \tag{5}$$

As pointed out in Ref. [28] using $\overline{\%}$ as a portability metric has several issues. We therefore include $\bar{\Phi}$ introduced in [27] and first used in [32] defined as

$$\bar{\Phi}\,(a,p,H) = \begin{cases} \frac{\sum_{i\in H} e_i(a,p)}{|H|} & \text{if } i \text{ is supported } \forall i \in H \\ \text{NA} & \text{otherwise,} \end{cases} \tag{6}$$

where $e_i(a,p)$ is the efficiency of application a implementing problem p on platform i and H is the set of investigated platforms.

Table 1
Target architecture details. Cores are streaming multiprocessors (SMs) on Nvidia, Compute Units (CUs) on AMD, and X^e cores on Intel GPUs. The bandwidth (Bw) is given for the high-bandwidth memory internal to the GPUs. References for the values can be found in footnotes[2–5].

Architecture	Cores	Clock [GHz]	FP64 [TFLOP/s]	Bw [GB/s]	Memory [GB]
Nvidia A100	108	0.76 / 1.41	9.7	1555	40
Nvidia GH200	132	1.66 / 1.84	34	4000	96
AMD MI250	208	1.00 / 1.70	45.3	3200	128
Intel GPU Max 1100	56	1.00 / 1.55	22	1229	48

Table 2
Overview of implementations on investigated architectures. ✓ indicates that benchmarks were performed. ✗ indicates that the implementation was not supported on that architecture at the time of the benchmark.

Architecture	OpenMP	pSTL	Kokkos	CUDA	HIP	SYCL	OpenACC
Nvidia A100	✓	✓	✓	✓	✓	✓	✓
Nvidia GH200	✓	✓	✓	✓	✓	✓	✓
AMD MI250	✓	✓	✓	✗	✓	✓	✓
Intel GPU Max 1100	✓	✓	✓	✗	✗	✓	✗

7. Benchmark setup

The benchmarks presented in this work were conducted on four different platforms: Nvidia Ampere 100 (A100), Nvidia Grace Hopper (GH200), AMD MI 250, and Intel GPU MAX 1100 (formerly known as Ponte Vecchio (PVC)). Information about relevant aspects of each GPU used in this study can be found in Table 1.

All implementations were compiled with the optimization level set to `-O3`. Relevant tuning and platform parameters (e.g., `-march` and `-mtune` for GCC) were applied where needed. For each system and implementation, an appropriate compiler was used. Below are the details of the nodes used for this work:

Benchmarks of **Nvidia A100**[2] were conducted on a node featuring an AMD EPYC Rome 7402 CPU, 512 GB of memory, and four 40 GB Nvidia A100 GPUs. We used NVHPC SDK version 23.7 to compile the pSTL, openACC, and OpenMP offload implementations, *nvcc* version 12 for CUDA, Kokkos, and HIP implementations, and *icpx* version 2024.1 for the SYCL implementation. As for **GH200**,[3] benchmarks were executed on a node with a 72-core Nvidia Grace CPU, 480 GB LPDDR5x memory, and a 96 GB HBM3 Nvidia H100 GPU. The compiler versions are the same as those used on the A100, except that the NVHPC SDK version is 23.11. For SYCL we build oneAPI DPC++ from the sources available on GitHub.

MI250[4] benchmarks were run on an experimental test node, with an AMD EPYC 7443 CPU, 512 GB memory, and 4 MI250 GPUs with 128 GB. We used the Clang 17.0 compiler provided as part of ROCm for HIP, Kokkos with HIP backend, and the OpenMP offloading implementations. For SYCL and pSTL we used AdaptiveCpp [23] and for OpenACC we used clacc [33]. For OpenMP offloading, we included `-fopenmp-target-fast` to enable the generation of specialized kernels. The optimization level remains at `-O3`.

Our benchmarks for **Intel GPU Max 1100**[5] were run on the free partition of the Intel Development Cloud. Each node is equipped with an Intel Xeon Platinum 8480+ processor, 512 GB of RAM, and 4 Intel GPU Max 1100 with 48 GB. The programs were compiled using the Intel compilers version 2024.1. We used the experimental SYCL backend for Kokkos.

We used only one GPU for the benchmarks and each architecture used a custom installation of Kokkos 4.3.0 with the above-mentioned backends. Table 2 has an overview of the status of the implementations on each system. The code is available publicly at Zenodo [34] and includes a set of JUBE [35] scripts that can be used to reproduce our benchmarks.

8. Results and discussion

The implementations that showed the best performance on the GH200, picked as the target platform, were used as analysis cases. Underlying measurements can be found in Fig. 3. The GH200 was chosen as the more modern architecture on which all programming models could be executed.

To evaluate the utilization of the hardware architectures in our benchmarks, we compared the throughput of particle–particle computations, where the number of interactions scales as N^2. As shown in Fig. 4, more powerful GPUs require larger system sizes to saturate the processor, indicated by the flattening of the lines in the plots.

It can be observed that the variants using shared memory generally perform better than the ones not using it. Especially on the GH200 this can be seen, as the three shared-memory variants group together at the top of the graph, while the remaining variants are closely grouped together below. Considering this grouping, results show that the achieved performance within each group is not diverging a lot. There are some exceptions, e.g., OpenACC on MI250 or Kokkos on the MI250 and GPU Max. These need to be investigated further.

Notably, the only native programming model to achieve the best measured results on its platform was CUDA on the GH200 when employing shared memory. On the MI250 SYCL outperformed HIP, while being outperformed on the Intel platform by OpenMP and pSTL. Surprisingly OpenACC, OpenMP and pSTL achieved higher performance on the A100 card in comparison to the CUDA version using shared memory as shown in Table 3. These results are in agreement with the study by Davis et al. [8], where recent improvements of SYCL's software stack led its implementations to outperform native programming models on compute bound kernels.

Memory transfer analysis revealed that OpenMP offloading and pSTL models use different strategies on Nvidia GPUs: asynchronous transfers for OpenMP and page faults for pSTL. However, for larger systems, this impact is minimal since the compute kernel is not memory-bound. For a thorough assessment of pSTL's backend implementations the reader may refer to the work of Lin et al. [13].

When comparing some of the other implementations across the different platforms a special case was encountered: using the overall slowest optimization variant (the if-based one) on A100 produced expected performance results, i.e., slower than the other optimization

[2] https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
[3] https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper?ncid=no-ncid
[4] https://www.amd.com/en/support/downloads/drivers.html/accelerators/instinct/instinct-mi200-series/amd-instinct-mi250.html
[5] https://www.intel.com/content/www/us/en/products/sku/232876/intel-data-center-gpu-max-1100/specifications.html

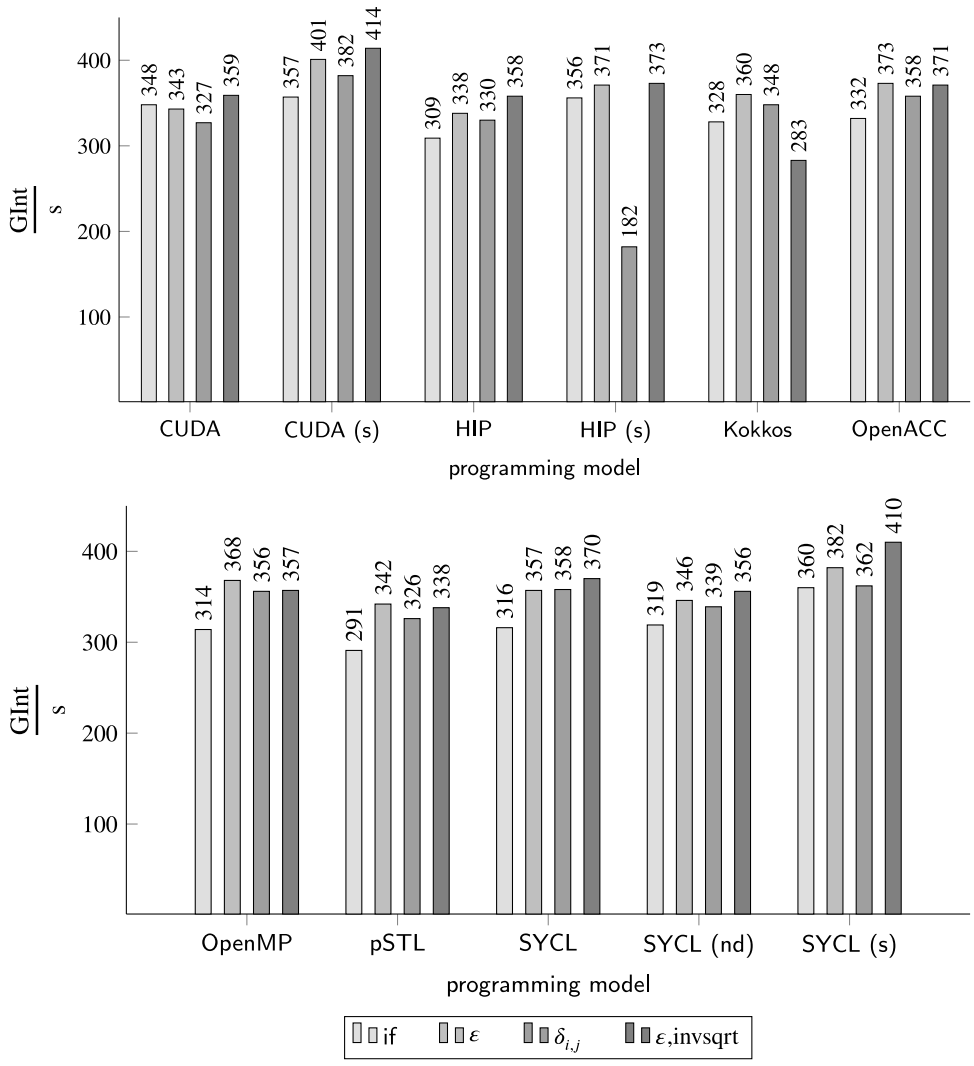

Figure 3. Benchmarks comparing the different types of optimization for the different implementations based on the GH200 architecture for all programming models. SYCL nd_range kernels are marked with (nd). Implementations using shared memory are indicated with (s). *if* denotes the basic if-based implementation, ϵ the variant using the smoothing factor, $\delta_{i,j}$ using the Kronecker delta and ϵ, invsqrt using the smoothing factor and a precomputed inverse square root.

Table 3
Fastest measured performance data across different system sizes, measured in Giga-interactions per second ($\frac{\text{GInt}}{\text{s}}$) on each platform for each programming model. The fastest overall measured performance is used as reference value for the application efficiency (%) and is indicated in bold. $\overline{\%}$ (cmp. Eq. (5)) is the weighted mean of % over all architectures, assuming % = 0, when benchmarks could not be performed. The underlined result is the best achieved value for $\overline{\%}$. (s) indicates variants using shared memory, while (nd) indicates the SYCL variant using nd-ranges. The last column shows the performance portability metric $\bar{\Phi}$.

	A100		GH200		MI250		GPU Max 1100			
	$[\frac{\text{GInt}}{\text{s}}]$	%	$[\frac{\text{GInt}}{\text{s}}]$	%	$[\frac{\text{GInt}}{\text{s}}]$	%	$[\frac{\text{GInt}}{\text{s}}]$	%	$\overline{\%}$	$\bar{\Phi}$
CUDA	165	93	369	87					30	NA
CUDA (s)	172	97	**423**	**100**					33	NA
HIP	164	92	370	87	182	98			63	NA
HIP (s)	169	95	398	94	179	97			64	NA
Kokkos	166	94	366	87	145	78	130	71	80	83
OpenACC	176	99	371	88	1.5	1			32	NA
OpenMP	**177**	**100**	371	88	142	78	**184**	**100**	91	92
pSTL	174	98	365	86	182	98	181	98	96	95
SYCL	159	90	370	87	176	95	171	93	92	91
SYCL (nd)	158	89	368	87	**185**	**100**	173	94	94	93
SYCL (s)	167	94	411	97	**185**	**100**	175	95	<u>97</u>	<u>97</u>

choices with one exception: the SYCL implementation showed 40% better performance on the A100 than the variants chosen based on the GH200, while showing a similar FLOP rate. This is one of the reasons, why we decided to focus on the application efficiency instead of architectural efficiency since users will be more often interested in application performance (i.e., runtime) rather than FLOP rates.

Analyzing the portability quality of the different programming models using the mean application efficiency $\overline{\%}$ (see Eq. (5)), it can be seen

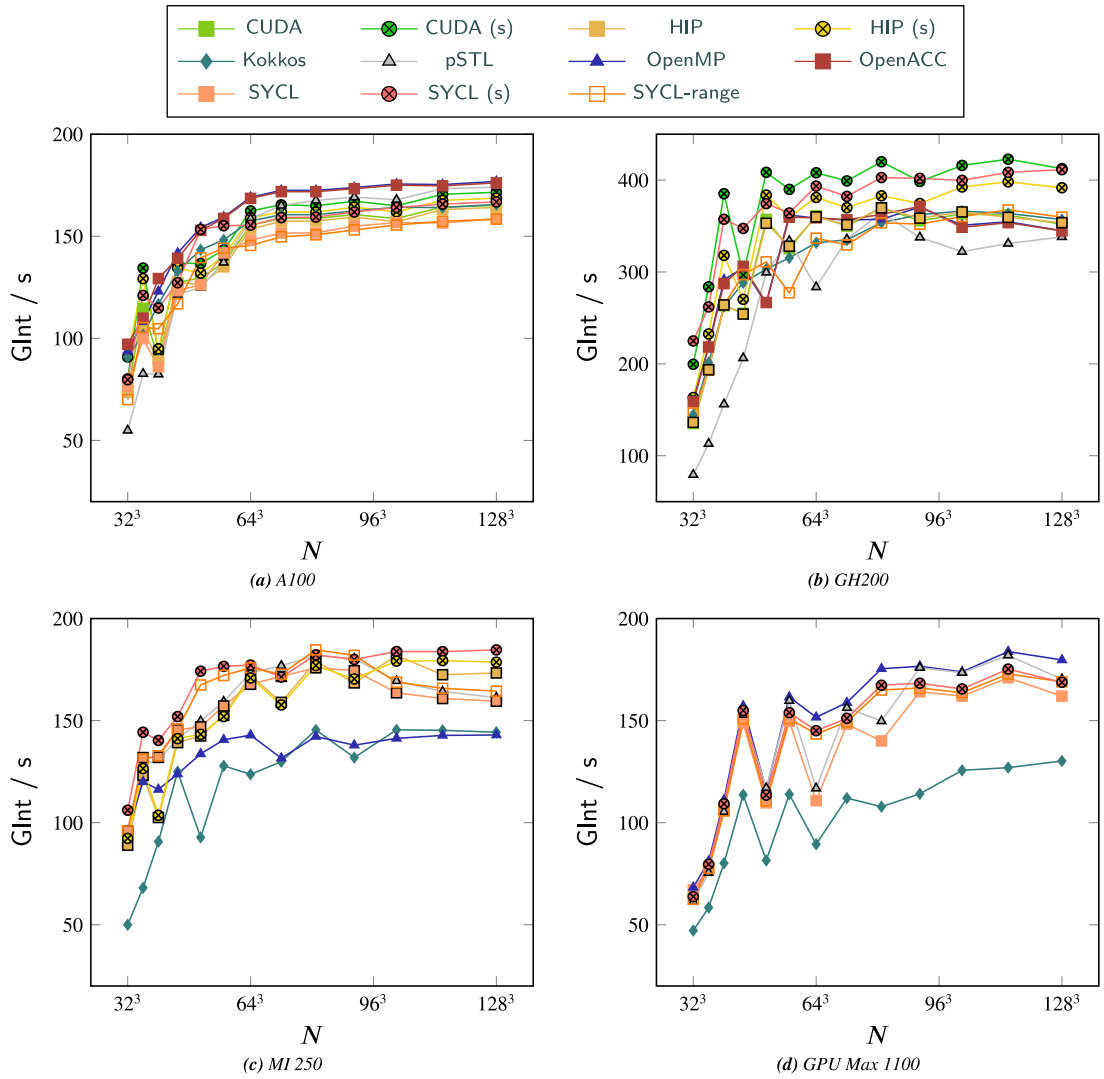

Figure 4. Interactions per second vs number of particles N on the investigated architectures by the different implementations. Note the different scale of the y-axis for GH200 (b). We do not include the results for OpenACC on the MI 250 since they are about a factor of 100 lower than those from the other programming models and nearly independent of system size.

that the models designed to be portable perform better than those targeted at specific platforms (HIP, CUDA). While this is to be expected, it is noteworthy that pSTL and SYCL (using shared memory) show the best portability quality. Kokkos shows comparatively lower performance on the AMD and Intel platforms. Regarding AMD, our result differs from recent studies, where Kokkos showed excellent performance on AMD-based systems [8,36]. This may be due to the variant chosen for its performance on the GH200.

The respective results for $\bar{\Phi}$ (Tables 3 and 5) show little deviation from the results for $\overline{\%}$ for the portable programming model, but make it clear that CUDA, HIP, and OpenACC are not portable since they are not applicable (NA) per definition for programming models that cannot be executed on one or more of the investigated architectures. The deviation in the percentages for the portable programming models comes from the different ways of averaging (see. Eq. (5) and (6)).

For a more detailed analysis of performance portability the number of taken measurements as well as the number of targeted platforms would need to be increased. Despite this, the performed analysis on the presented data already indicates that using an optimized version of a code on a given architecture (in this case the GH200) might not be optimally suited for other architectures but still perform satisfactory well.

8.1. Effect of block size

Some models, e.g., CUDA, HIP, and SYCL nd_range take an explicit thread block (work group) size as parameter. Threads (work items) in this block can take advantage of shared (local) memory if available and can be more easily synchronized. On GPUs a thread block is usually pinned to a processing unit. To fill a GPU, we therefore need at least as many blocks as processing units. Requested hardware resources, e.g., registers, can limit the number of blocks that can be kept in flight.

We use 512 threads per block for CUDA and HIP and 256 threads per block for SYCL (nd) and SYCL (s). In Fig. 5 we show the effect of changing the number of threads per block with respect to our benchmark results for each of the 4 GPUs. Choosing a block size that is less than the size of a warp (or wave front) is obviously detrimental to the performance of the kernel. Setting the block size to 16 threads, for example, leaves half of the hardware on an Nvidia GPU and 3/4 of the hardware of an AMD GPU without work. If we choose our block size in the interval [64, 512] the effect on performance is usually within 5% of the optimum.

Keeping the block size fixed across all platforms our choice is optimal except for CUDA without shared memory (see Table 4). A small gain of about 0.5% on average could have been achieved by using 64 or 128 threads per block.

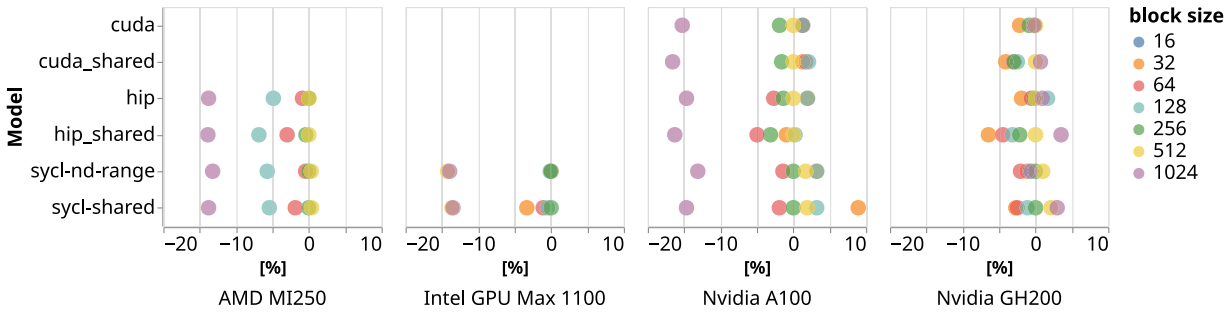

Figure 5. Performance impact of deviating from the block size used in the benchmark. We used a block size of 512 threads for HIP and CUDA and 256 threads for SYCL. In this figure, we show the impact of changing the block size for the different architectures. All platforms show reasonable elasticity for block sizes between the minimum thread number required by the hardware and 512 threads. In this range the loss in performance is limited to less than 10% with respect to the maximum.

Table 4
Average effect of choice of block size across all platforms with respect to the block size used in our benchmarks. The table shows that the block size used (512 for CUDA and HIP and 256 for SYCL) is near optimal for all models. It also indicates that choosing a different block size from the interval [64,512] would have had little effect on the performance.

Block size	CUDA	CUDA (s)	HIP	HIP (s)	SYCL (nd)	SYCL (s)
16	−46.44	−48.83	−54.18	−57.42	−46.14	−47.60
32	−0.47	−1.41	−15.04	−18.14	−10.75	−11.08
64	0.52	−0.64	−1.38	−4.15	−1.00	−1.84
128	0.51	−0.21	−0.43	−3.27	−0.80	−0.95
256	−1.40	−2.26	−0.52	−1.90	0.00	0.00
512	0.00	0.00	0.00	0.00	−2.81	−2.30
1024	−7.72	−7.96	−9.18	−8.89	−10.30	−9.75

For the pSTL and simple data-parallel kernels in SYCL the distribution of work is determined exclusively by the runtime. The pragma-based models OpenMP and OpenACC also allow the runtime to determine the block size, but contain clauses that let the developer influence the distribution of work over threads and blocks.

8.2. Consumer GPUs

While the focus of this paper is performance and portability across GPUs used in HPC environments, development and early simulations are often performed on consumer cards or even CPUs. We therefore decided to share our experience.

Any Nvidia consumer card since the GeForce 8 series released in 2006 has included support for CUDA. Available features are categorized by compute capability and usually cards with a higher compute capability include the features supported by a lower compute capability. This and the easy availability of the CUDA SDK have enabled scientists and hobbyists to learn about and develop CUDA code.

The list of consumer devices supported officially by ROCm, AMD GPU software stack, is fairly small,[6] but also deceiving. A much larger set of devices is supported by Clang[7] to compile HIP code for AMD devices. SYCL code can be compiled using, e.g., AdaptiveCpp or Intel's fork of LLVM.

Intel's support for SYCL on consumer devices includes integrated and discrete GPUs and can be used, for example, with AdaptiveCpp, Intel's fork of LLVM oneAPI DPC++, or Intel's icpx compiler. Intel's icpx compiler also includes support for OpenMP offloading.

Consumer devices usually have a much smaller ratio of double to single precision performance than devices meant for HPC applications and it can be especially beneficial to implement single-or mixed precision algorithms, which would benefit the performance on the HPC cards as well.

[6] https://rocm.docs.amd.com/projects/radeon/en/latest/docs/compatibility/native_linux/native_linux_compatibility.html#gpu-support-matrix

[7] https://llvm.org/docs/AMDGPUUsage.html#processors

8.3. CPUs

GPUs and CPUs have different characteristics and a program that has been written with a GPU in mind will not necessarily perform well on CPUs. Since several of the programming models provide CPU backends, we benchmarked the following selection of CPUs. As an AMD CPU we used an EPYC Rome 7743, for Intel a Xeon Platinum 8168 and the Nvidia Grace CPU that has Arm Neoverse-V2 cores. Compiler and options used for the CPU benchmark are the same as presented in Section 7 with the exception of pSTL, here GCC 13.3.0 with Intel's Thread Building Blocks (TBB) library and for SYCL on the Grace chip AdaptiveCPP are used. In Table 5 we show the results of this benchmark.

For brevity in the CPU results we present only the standard version of the SYCL implementation. All SYCL variants have shown similar results with deviations of less then 4%. In contrast to the GPU findings presented in Table 3 pSTL does not perform as well on CPU as on GPU. OpenMP and OpenACC show good portability across all CPU platforms, with OpenMP being one of the best performing programming models on all architectures.

9. Conclusion and outlook

There has never been a better time for using portable programming models for writing GPU code. Our results show that OpenMP, pSTL, and SYCL are well suited for portable implementations of the N-body problem across various GPU vendors, while Kokkos showed slightly reduced performance on the non-Nvidia platforms. Kokkos might show better performance, when using algorithms more dependent on complex memory access patterns, e.g., short-range interactions as described below. In addition, OpenMP shows close to perfect portability behavior in the performed CPU benchmarks. For Nvidia and AMD GPUs, HIP also provides a competitive framework for writing portable code.

Future analyses could also include other portable solutions (e.g., RAJA [5] or Alpaka [2]) to provide a more comprehensive comparison. Also, where applicable the impact of using different compilers and/or optimization flags on the used models should be analyzed in more detail.

Table 5
Fastest measured performance data across different system sizes, measured in Giga-interactions per second $\frac{\text{GInt}}{\text{s}}$ on different CPUs. The fastest overall measured performance is used as reference value for the application efficiency (%) and is indicated in bold. $\overline{\%}$ (cmp. Eq. (5)) is the weighted mean of % over all architectures, assuming % = 0, when benchmarks could not be performed. The underlined result is the best achieved value for $\overline{\%}$. For comparison in the last column the results for the performance metric $\bar{\Phi}$ is presented.

	AMD		Intel		Grace			
	$[\frac{\text{GInt}}{\text{s}}]$	%	$[\frac{\text{GInt}}{\text{s}}]$	%	$[\frac{\text{GInt}}{\text{s}}]$	%	$\overline{\%}$	$\bar{\Phi}$
Kokkos	25	42	9	68	26	77	62	62
OpenACC	**58**	**100**	**14**	**100**	32	94	98	98
OpenMP	**58**	**100**	**14**	**100**	**34**	**100**	<u>100</u>	<u>100</u>
pSTL	14	24	5	36	30	92	51	51
SYCL	55	97	**14**	**100**	23	68	88	88

A more in-depth analysis of the impact of using code optimized for one architecture on the performance on other platforms to determine the performance portability [37] would be beneficial to users and developers for choosing a programming model, depending on their individual aims (portability vs. high performance on a single architecture).

While the N-body problem and its reduction to the force calculation is a fundamental method for computing interactions between particles and is integral to more advanced algorithms, extending this analysis to more sophisticated algorithms is essential, which includes splitting methods (e.g., Ewald summation) or hierarchical methods (e.g., Barnes–Hut).

Another area to investigate is algorithms for the computation of short-range interactions. Unlike the gravitational interaction used in this paper, short-range interactions exclude some neighbor computations and introduce steps to determine neighbor pairs based on proximity. This additional sorting step significantly changes memory access patterns and impacts performance.

Finally, we plan to make these and implementations for other problem classes representing different algorithmic methods publicly available.

CRediT authorship contribution statement

Rodrigo A.C. Bartolomeu: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization. **René Halver:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Conceptualization. **Jan H. Meinke:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Conceptualization. **Godehard Sutmann:** Writing – review & editing, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank the Jülich Supercomputing Centre, Forschungszentrum Jülich for access to the JURECA-DC Evaluation Platform, JUWELS Booster and JEDI. This work has received funding from the European High Performance Computing Joint Undertaking under the grant agreement 101093169 and was supported by the German Federal Ministry of Education and Research (BMBF) and the Ministry of Culture and Science (MKW) of the state of North-Rhine-Westphalia through funding of the Gauss Centre for Supercomputing (GCS).

Data availability

Data will be made available on request.

References

[1] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, J. Salmon, Performance portability across diverse computer architectures, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2019, pp. 1–13, http://dx.doi.org/10.1109/P3HPC49587.2019.00006.

[2] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W.E. Nagel, M. Bussmann, Alpaka–an abstraction library for parallel kernel acceleration, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW, IEEE, 2016, pp. 631–640, http://dx.doi.org/10.1109/IPDPSW.2016.50.

[3] C.R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilke, Kokkos 3: Programming model extensions for the exascale era, IEEE Trans. Parallel Distrib. Syst. 33 (2022) 805–817, http://dx.doi.org/10.1109/TPDS.2021.3097283.

[4] M. Garland, M. Kudlur, Y. Zheng, Designing a unified programming model for heterogeneous machines, in: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, Salt Lake City, UT, 2012, pp. 1–11, http://dx.doi.org/10.1109/SC.2012.48.

[5] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R. Scogland, RAJA: Portable performance for large-scale scientific applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 71–81, http://dx.doi.org/10.1109/P3HPC49587.2019.00012.

[6] N. Bell, J. Hoberock, C. Rodrigues, THRUST: A productivity-oriented library for CUDA, in: Programming Massively Parallel Processors, Elsevier, 2017, pp. 475–491, http://dx.doi.org/10.1016/B978-0-12-811986-0.00033-9.

[7] L. Chen, L. Liu, S. Tang, L. Huang, Z. Jing, S. Xu, D. Zhang, B. Shou, Unified parallel C for GPU clusters: Language extensions and compiler implementation, in: K. Cooper, J. Mellor-Crummey, V. Sarkar (Eds.), Languages and Compilers for Parallel Computing, Springer, Berlin, Heidelberg, 2011, pp. 151–165, http://dx.doi.org/10.1007/978-3-642-19595-2_11.

[8] J.H. Davis, P. Sivaraman, I. Minn, K. Parasyris, H. Menon, G. Georgakoudis, A. Bhatele, Taking gpu programming models to task for performance portability, 2024, arXiv preprint arXiv:2402.08950.

[9] C. Phuong, N. Saied, C. Tanis, Assessing Kokkos performance on selected architectures, in: Latin American High Performance Computing Conference, Springer, 2019, pp. 170–184, http://dx.doi.org/10.1007/978-3-030-41005-6_12.

[10] A.S. Dufek, R. Gayatri, N. Mehta, D. Doerfler, B. Cook, Y. Ghadar, C. DeTar, Case study of using Kokkos and SYCL as performance-portable frameworks for Milc-Dslash benchmark on NVIDIA, AMD and Intel GPUs, in: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2021, 2021, pp. 57–67, http://dx.doi.org/10.1109/P3HPC54578.2021.00009.

[11] M. Breyer, A. Van Craen, D. Pflüger, A comparison of SYCL, OpenCL, CUDA, and OpenMP for massively parallel support vector machine classification on multi-vendor hardware, in: Proceedings of the 10th International Workshop on OpenCL, IWOCL '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 1–12, http://dx.doi.org/10.1145/3529538.3529980.

[12] Y. Ding, C. Xu, H. Qiu, Q. Wang, W. Dai, Y. Lin, Y. Che, Evaluating performance portability of SYCL and Kokkos: A case study on LBM simulations, in: 2023 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom), 2023, 2023, pp. 328–335, http://dx.doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom59178.2023.00075.

[13] W.-C. Lin, T. Deakin, S. McIntosh-Smith, Evaluating ISO C++ parallel algorithms on heterogeneous HPC systems, in: 2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS, IEEE, 2022, pp. 36–47, http://dx.doi.org/10.1109/PMBS56514.2022.00009.

[14] L. Nylons, Mark. Harris, Jan. Prins, Chapter 31. fast n-body simulation with CUDA, in: GPU Gems 3, 2007, pp. 62–66.

[15] T. Thüring, M. Breyer, D. Pflüger, Comparing a naive and a tree-based n-body algorithm using different standard SYCL implementations on various hardware, in: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1906–1917, http://dx.doi.org/10.1145/3624062.3624604.

[16] A. Poenaru, W.-C. Lin, S. McIntosh-Smith, A performance analysis of modern parallel programming models using a compute-bound application, in: International Conference on High Performance Computing, Springer, 2021, pp. 332–350.

[17] R.A.C. Bartolomeu, R. Halver, J.H. Meinke, G. Sutmann, Assessing performance of programming models across GPU vendors, in: Parallel Processing and Applied Mathematics, 2025, in press.

[18] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The landscape of parallel computing research: A view from Berkeley, in: Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, 2006.

[19] J. Zhao, C. Bertoni, J. Young, K. Harms, V. Sarkar, B. Videau, HIPLZ: Enabling performance portability for exascale systems, Concurr. Computation: Pr. Exp. 35 (2023) e7866, http://dx.doi.org/10.1002/cpe.7866.

[20] H.C. Edwards, C.R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, J. Parallel Distrib. Comput. 74 (2014) 3202–3216, http://dx.doi.org/10.1016/j.jpdc.2014.07.003.

[21] OpenMP Architecture Review Board, Openmp application program interface version 4.0, 2013, URL: https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[22] J. Hubbard, G. Brito, C. Garg, N. Sakharnykh, F. Oh, Simplifying GPU application development with heterogeneous memory management, 2023, URL: https://developer.nvidia.com/blog/simplifying-gpu-application-development-with-heterogeneous-memory-management/.

[23] A. Alpay, V. Heuveline, AdaptiveCpp stdpar: C++ standard parallelism integrated into a SYCL compiler, in: Proceedings of the 12th International Workshop on OpenCL and SYCL, IWOCL '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 1–12, http://dx.doi.org/10.1145/3648115.3648117.

[24] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Oxford University Press, 2017, http://dx.doi.org/10.1093/oso/9780198803195.001.0001.

[25] A. Herten, Many cores, many models: GPU programming model vs. vendor compatibility overview, in: Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1019–1026, http://dx.doi.org/10.1145/3624062.3624178.

[26] R. Halver, J.H. Meinke, G. Sutmann, Examining performance portability with Kokkos for an Ewald sum Coulomb solver, in: International Conference on Parallel Processing and Applied Mathematics, Springer, 2019, pp. 35–45, http://dx.doi.org/10.1007/978-3-030-43222-5_4.

[27] A. Marowka, On the performance portability of OpenACC, OpenMP, Kokkos and RAJA, in: HPCAsia '22, Association for Computing Machinery, 2022a, pp. 103–114, http://dx.doi.org/10.1145/3492805.3492806.

[28] A. Marowka, Reformulation of the performance portability metric, Software: Pr. Exp. 52 (2022) 154–171, http://dx.doi.org/10.1002/spe.3002.

[29] S.J. Pennycook, J.D. Sewall, Revisiting a metric for performance portability, in: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2021, pp. 1–9, http://dx.doi.org/10.1109/P3HPC54578.2021.00004.

[30] S.J. Pennycook, J.D. Sewall, V.W. Lee, A metric for performance portability, 2016, http://dx.doi.org/10.48550/arXiv.1611.07409, arXiv preprint arXiv:1611.07409.

[31] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, et al., An empirical roofline methodology for quantitatively assessing performance portability, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2018, pp. 14–23, http://dx.doi.org/10.1109/P3HPC.2018.00005.

[32] W.F. Godoy, P. Valero-Lara, T.E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R.G. Miller, M. Gonzalez-Tallada, J.S. Vetter, V. Churavy, Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes, in: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2023, pp. 373–382, http://dx.doi.org/10.1109/IPDPSW59300.2023.00068.

[33] J.E. Denny, S. Lee, P. Valero-Lara, M. Gonzalez-Tallada, K. Teranishi, J.S. Vetter, Clacc: Openacc for C/C++ in Clang, Int. J. High Perform. Comput. Appl. 38 (2024) 427–446, http://dx.doi.org/10.1177/10943420241261976.

[34] R.A.C. Bartolomeu, R. Halver, J.H. Meinke, G. Sutmann, [DATASET] Effect of implementations of the N-body problem on the performance and portability across GPU vendors, 2024, http://dx.doi.org/10.5281/zenodo.14287180.

[35] T. Breuer, J. Wellmann, F. Souza Mendes Guimarães, C. Himmels, S. Luehrs, JUBE, Zenodo, 2024, http://dx.doi.org/10.5281/zenodo.11235164.

[36] W.-C. Lin, S. McIntosh-Smith, T. Deakin, Preliminary report: Initial evaluation of StdPar implementations on AMD GPUs for HPC, 2024, http://dx.doi.org/10.48550/arXiv.2401.02680, arXiv:2401.02680.

[37] A. Marowka, Portability efficiency approach for calculating performance portability, 2024, http://dx.doi.org/10.48550/arXiv.2407.00232, arXiv:2407.00232.

Rodrigo A. C. Bartolomeu is a Post Doc staff scientist of the Simulation and Data Laboratory Complex Particle Systems at the Jülich Supercomputing Centre, Forschungszentrum Jülich. His research interests include electrostatics, algorithm optimization, research software engineering, performance portability, statistical mechanics, and molecular dynamics.

René Halver is a staff scientist of the Simulation and Data Laboratory Complex Particle Systems at the Jülich Supercomputing Centre, Forschungszentrum Jülich. His research interests include load balancing algorithms for particle simulations and performance portability to make efficient use of HPC hardware for simulations.

Jan H. Meinke is a senior staff scientist of the Simulation and Data Laboratory Biology at the Jülich Supercomputing Centre, Forschungszentrum Jülich. His research interests include protein folding and finding ways to make efficient use of HPC hardware for solving scientific problems.

Godehard Sutmann is head of the Simulation and Data Laboratory Complex Particle Systems at Jülich Supercomputing Centre, Forschungszentrum Jülich and is Professor at ICAMS, Ruhr-University Bochum. His research interests include parallel algorithms for particle simulations, load balancing, statistical physics and materials science simulations.