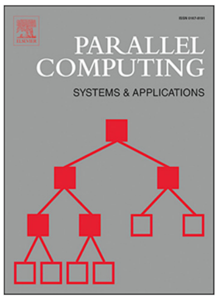


Using heterogeneous GPU nodes with a Cabana-based implementation of MPCD

Rene Halver, Christoph Junghans, Godehard Sutmann

PII: S0167-8191(23)00039-X
DOI: https://doi.org/10.1016/j.parco.2023.103033
Reference: PARCO 103033

To appear in: *Parallel Computing*

Received date : 14 November 2022
Revised date : 10 May 2023
Accepted date : 3 June 2023

Using Heterogeneous GPU Nodes with a Cabana-Based Implementation of MPCD

Rene Halver[a,**], Christoph Junghans[c], Godehard Sutmann[a,b,*]

[a]*Forschungszentrum Jülich, Jülich Supercomputing Centre, Wilhelm-Johnen-Strasse, 52425 Jülich, Germany*
[b]*ICAMS, Ruhr-University Bochum, 44801 Bochum, Germany*
[c]*Los Alamos National Laboratory, CCS-7, 87545 Los Alamos, New Mexico, USA*

Abstract

The Kokkos based library Cabana, which has been developed in the Co-design Center for Particle Applications (CoPA), is used for the implementation of Multi-Particle Collision Dynamics (MPCD), a particle-based description of hydrodynamic interactions. Cabana allows for a function portable implementation, which has been used to study the interplay between CPU and GPU usage on a multi-node system as well as analysis of said interplay with performance analysis tools. As a result, we see most advantages in a homogeneous GPU usage, but we also discuss the extent to which heterogeneous applications might be more performant, using both CPU and GPU concurrently.

Keywords: MPCD, Kokkos, modular computing

1. Introduction

The recent development of high-end parallel architectures shows a clear trend to a heterogeneity of compute components, pointing towards a dominance of General Purpose Graphics Processing Units (GPU) as accelerator components, compared to the Central Processing Units (CPU). According to the Top 500 list [1], more than 25% of the machines have GPU support while the overall performance share is more than 40%, i.e., heterogeneous cluster architectures have a large impact for high compute performance. Often these nodes consist of only a few multicore CPUs, while supporting 2-6 GPUs. In many applications one can observe a trend that the most powerful component of the nodes, i.e. the GPUs, is addressed, while the CPUs are used as administrating or data management components. A reason might be the additional overhead in writing/maintaining two different code versions for each architecture, as usually a CPU code cannot simply run on a GPU or vice versa.

With the advent of function and performance portable programming models, such as Kokkos [2] or Raja [3] it has become possible to use the same code base for different architectures, most prominently including CPUs or GPUs. It might be tempting to use the full capacity of a compute-node concurrently, i.e. not wasting compute resources because of the disparate character of the architecture and programming model. In this case one encounters both different performance characteristics of components and possibly a non-negligible data transfer between components. This discrepancy might be targeted by load balancing strategies which would need to take into account hardware and software specific characteristics to achieve an overall performance gain. But regardless of the possible improvements, it is expected that if there is a benefit for heterogeneous execution, this might apply only to smaller system sizes, where due to the under-utilization of the GPU, it might be advantageous to avoid data transfer to the GPU, while the CPU can compute the necessary calculations in a similar timeframe as the GPU. This is of interest for scientific use cases where not the spatial size of the system is important, but rather the length for which the system can be simulated, due to required equilibration or minimization effects.

In the present paper we consider a stochastic particle based method for the simulation of hydrodynamic phenomena, i.e. the Multi-Particle Collision Dynamics (MPCD) [4] algorithm and its implementation with Cabana [5, 6, 7, 8], a Kokkos based library. We first introduce the underlying MPCD method and then describe the Cabana library. Furthermore, we present benchmarking results, present analyze results with performance analysis tools and draw conclusions from our findings before giving some outlook for further research.

2. Multi-Particle Collision Dynamics

MPCD is a particle-based description for hydrodynamic interactions in an incompressible fluid. The method is based on a stochastic collision scheme in which particles, that describe the simulated fluid are rotated in velocity space while conserving linear momentum and energy (variants exist which also conserve angular momentum [4]). The method proceeds by sorting particles into a regular mesh with grid cells of size of a characteristic length scale. In order to transport momentum and energy across the system, the mesh is randomly shifted in each time step, changing the local environment of each particle stochastically. For each particle in a cell its relative velocity

[*]Principal Corresponding Author
[**]Corresponding Author

Email addresses: r.halver@fz-juelich.de (Rene Halver), junghans@lanl.gov (Christoph Junghans), g.sutmann@fz-juelich.de (Godehard Sutmann)

with respect to the center-of-mass (*com*) velocity

$$v_{cm}^{(c)} = \frac{1}{M} \sum_{n=1}^{N^{(c)}} v_i \cdot m_i \quad , \quad M = \sum_{n=1}^{N^{(c)}} m_i \qquad (1)$$

of the cell is computed. The *com* velocity v_{cm} is computed by summing up the momenta of each particle in cell, with the cell containing $N^{(c)}$ particles, and dividing this cell momentum by the total particle mass of the cell. This velocity is split into a parallel and perpendicular component with respect to a randomly oriented axis in the cell. Consequently, the perpendicular component is rotated around that axis by a fixed angle, which determines together with the particle mass and density, the time step and the cell length the diffusion and viscosity of the fluid under consideration. This procedure can be shown to mimic hydrodynamic behaviour and, in a limiting case, enters into the Navier Stokes equations [4]. Using this procedure the conservation of linear momentum and energy is guaranteed and can also be coupled to embedded particles, simulated by other methods, e.g. molecular dynamics, thereby coupling particle dynamics to a hydrodynamic medium [4, 9].

From an algorithmic point of view, three main parts can be identified, these are:

i) the local identification of particles in the underlying cell structure and the computation of *com* velocities of cells

ii) the computation of the relative velocities of particles with respect to the *com* velocity of a cell

iii) rotation of perpendicular velocity component of particles around a random axis

The implementation in Cabana of these three parts will be discussed separately in Sec. 3 in more detail. Since the performance portability framework Kokkos is the basis of the Cabana library, one could study two related questions: (i) what is the overhead, i.e. performance loss of using a library and (ii) what is the efficiency in terms of performance portability, compared with native optimised implementations, e.g. OpenMP + CUDA on NVIDIA based GPU nodes or OpenMP + rocM on AMD based GPU nodes. In this paper, however, the focus is put on the investigation of function portability, i.e. ease of use of running the code on different architectures. The analysis of benchmarks in this paper is driven from the motivation of using Kokkos for the provision of a base implementation, which can be easily executed on different types of architectures without the need of rewriting the code. I.e. we do not specifically consider a given metric for performance portability but show that a variety of architectures can be addressed by using the same code base. Nevertheless, we will show and compare performance of executions on the different architectures. The important question of performance comparison between Kokkos-based and optimized implementations.

3. Implementation with Cabana

The aim of the implementation was to write a code, that is function portable between clusters consisting of CPU and clusters with GPU nodes, which often consist of one or two CPUs

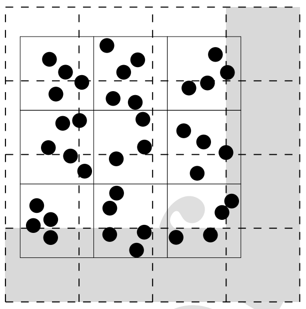


Figure 1: Illustration of the shifted collision cell grid (black, dashed) in comparison to the static logical cell grid (black, solid). The grey cells mark the periodic images of the shifted grid.

and a number of GPUs ranging from two to six. Also, it might be that different clusters use architecture from different vendors, which might require different types of programming languages to target these differing architectures. Maintaining two or more codebases for all targeted architectures increases the overhead time of, e.g., design or maintenance time, and calls for solutions which allow a unified approach for various architectures. In addition, maintaining different codebases can lead to introduction of unintended differences in those, making comparisons between each other difficult.

For this reason performance portable programming models are attractive for reducing time spent with porting codes to various architectures. One of the more popular programming models in this regard is Kokkos [2], which provides an abstraction layer for data structures, called *Views*, while providing different *ExecutionSpaces* which can either be on the host (usually the CPU) or on devices, i.e. GPUs or other accelerator cards, e.g. Intel KNLs. Kokkos uses different backends to provide this performance portability, e.g. CUDA for the use of NVIDIA GPUs or ROCm / HIP for the use of AMD GPUs. Furthermore, OpenMP or PThread backends can be used among others to utilise multicore architectures of CPUs. An important detail here is, that only one kernel needs to be written, which can then be passed to different (compiled) backends.
Within the Exascale Computing Project (ECP) [10] funded by the Department of Energy (DoE) in the USA, the Co-Design Center for Particle Applications (CoPA) [11] developed a performance portable library, based on Kokkos, with the main focus of supporting the development of particle and grid based codes on HPC systems. Cabana not only provides data structures based on Kokkos *Views* but also provides routines in order to facilitate data transfer between different processes in a distributed-memory environment, based on MPI. Additionally, the Cajita sub-library provides tools for the management of grid in parallel environments, providing routines for data transfer and distribution.
Since the MPCD method is a mixture of a particle and a grid based method (due to the requirement to sort the particles into cells), the implementation of the MPCD code using Cabana was considered reasonable. In the rest of the section the main points of the implementation will be presented, based on the previ-

ously identified main points of the MPCD algorithm.

3.1. Collection of Particles in Cells

Before the *com* velocity for a cell can be calculated, it is necessary to identify the particles that reside in each collision cell. One technique to achieve this is the linked-cell list. Accordingly, all particles are checked and flagged with a cell identifier to which they belong to. In addition, a (linked) list of particles belonging to the cell is created. Listing 1 shows how such a list is created in Cabana. The use of Cabana simplifies the creation of such a linked cell list, as Cabana deals with the issues of creating a linked cell list in a multithreaded environment, as described e.g. in [12] or [13].

Listing 1: Creation of the linked cell list of the shifted collision cell grid

```
// boundaries of spacial domains
double gridMin[3], gridMax[3];
for (int d = 0; d < 3; ++d)
{
  gridMin[d] = domBorders(2*d) -
               (double)haloWidth
               * cellSize(d) +
               offset(d);
  gridMax[d] = domBorders(2*d+1) +
               (double)haloWidth
               * cellSize(d) +
               offset(d);
}
// creating the linked cell list
// r = list of particle positions
// cellSize = size of linked
//            cells (3d)
Cabana::LinkedCellList<DeviceType>
  linkedList( r, cellSize, gridMin,
              gridMax );
// permute the particle AoSoA to
// correspond to the cells
Cabana::permute( linkedList,
                 particles );
```

3.2. Communication of Required Information

As described in section 2, it is necessary to compute the *com* velocity, i.e. the velocity in a zero momentum frame with regard to the local collision cell [14], in order to calculate the collisions within each mesh cell, which requires all velocities and masses of particles that reside within the given collision cell. The underlying parallel algorithm is based on a domain decomposition, where compute resources administrate geometrical spatial regions which are connected. Since the underlying mesh is shifted in each time step cells might be split among several domains. To compute a unique value for the *com* velocity, one can either collect all particles together with their properties on a local domain or one can compute the partial *com* velocities on each local domain and then reduce this value among those processes which share the given cell.

The first of these methods has the advantage that since all particles are collected on a single domain, the computation of the *com* velocity and the following rotation of velocities can be executed without the need of additional communication steps in between. The disadvantage is that it requires the communication of particle data in each time step, since the collision cell mesh needs to be shifted in each time step to avoid artefacts in the computation of the hydrodynamic interactions. Listing 2 shows the necessary steps to prepare the particle migration between domains. Shown here is a way to try to avoid unnecessary branching while determining the target processes for particles. This is done by masking the target processes with a base-3 number, where each 'bit' indicates either a shift down(0) or up (1) or residing in the domain's boundary concerning that Cartesian direction. As an example a base-3 number of $(201)_3$ would be assigned to a particle leaving the local domain in positive x-direction and negative y-direction, while stay in the same z-region, as the local domain. This way to determine target processes should improve execution on GPU, with the tertiary operator being removed, in case that true is cast to integer one and false to integer zero.

Listing 2: Particle based communication with Cabana

```
Kokkos::parallel_for(
  Kokkos::RangePolicy<ExecutionSpace>
        (0, nParticles),
  KOKKOS_LAMBDA (const size_t i)
  {
    int dims = 1, index = 0;
    // compute the direction of the
    // neighbour the particle needs
    // to be moved into and use dims
    // to compute a base 3 mask:
    // (xyz)_3 with 0 (left),
    //              1 (remains),
    //              2 (right)
    // r = list of particle
    //     positions
    for (int d = 2; d >= 0; --d)
    {
        index += dims *
          ( 1 -
            ((r(i,d) <
            domBorders(2*d))?1:0) +
            ((r(i,d) >=
            domBorders(2*d+1))?1:0)
          );
        dims *= 3;
    }
    // tag the particle with the
    // target neighbour rank
    export_ranks(i) = neigs(index);
  });
Kokkos::fence();

// create particles distribution object and
// migrate particles to targets
Cabana::Distributor<DeviceType> dist( mpiCart,
                        export_ranks, neighbours );
Cabana::migrate(dist, particles);
```

In contrast, the second method allows the use of a stable, halo-based communication scheme, where particles are not necessarily communicated in each time step, but only when leaving a halo region around the local domain, allowing the distributed computation of partial *com* velocities, that are reduced with a static communication scheme. The result is then sent back to the domains sharing the same cell. Listing 3 shows the required function calls to Cabana to do the halo exchange. This work, related to mesh administration, is implemented in *Cajita*, which is part of Cabana. In addition, it provides methods for particle-grid interactions, e.g. interpolation of particle properties to a grid, which is, however, not used in this work. Furthermore, Cajita provides a domain-based load balancing based on a tensor decomposition scheme, provided by the ALL library [15].

Listing 3: Grid based halo communication with Cabana

```
// create the halo communication object based
// on the Cajita grid
auto arrHalo = Cajita::createHalo( *arrNode,
               Cajita::NodeHaloPattern<3>());
// [...] computation of com velocities
// bring the data to the halo cells
arrHalo->gather(ExecutionSpace(), *arrNode);
// collect the data from the halo cells
arrHalo->scatter(ExecutionSpace(),
     Cajita::ScatterReduce::Sum(), *arrNode);
```

For the implementation of the two different communication schemes two different kinds of communication in Cabana were used. For the former method, the particle-based one, Cabana provides a *Distributor* class, which allows the transfer of particle data between processes. This requires that particles are tagged with the target process, so that the *Distributor* object can generate a communication topology for this specific transfer. As a consequence this object needs to be recreated in every time step, since the communication pattern in each time step changes due the random shift of the collision cell grid and particle movements across domain borders. Analysis of this part of the communication scheme showed that due to implementation of the particle transfer routine in Cabana in each communication step the required buffers had to be newly allocated and the communication pattern was established anew everytime. This led to a sizeable overhead, when particle transfers between different processes were conducted. Implementing a manual particle transfer, in which particle buffers were reused and resized if necessary, improved the performance in that part. Discussions with developers of the Cabana library led to the plan to include this feature into a future release.

For the second communication pattern, reducing the partial results and redistributing them, a halo-based communication on a grid is used. For this purpose, two different grids are combined, i.e. a logical collision grid which is used for communication and a linked-cell list, which sorts the particles into the shifted collision cell grid. Since the number and size of mesh cells in each grid is identical, both grids can be perfectly matched onto each other. The particles are sorted into the linked-cell list (Sec. 3.1) from where the *com* momentum of each cell is computed. For collision cells, overlapping with domain borders (Fig. 1), a halo-based communication reduces the partial results on the process which administrates the logical cell. This process redistributes the reduced sum back to each participating neighbour, where the rotations of velocities are computed for residing particles. Since the number of cells is usually far smaller than the number of particles, this leads to (i) a static communication scheme (for each iteration step the same operations on the same amount of data) and (ii) a reduced and constant amount of data that needs to be communicated.

During the development, it became apparent that the second communication scheme leads to a better performance due to the reduced amount of transferred data and the strongly reduced necessity to recreate communication patterns, due to the stable communication scheme of the halo exchange (this needs to be done only once in the beginning or after possible load balancing steps, after which the communication pattern is static). In addition, the transfer of particles can be reduced to cases, where particles left the halo region surrounding the local domain, instead of being required in every time step.

3.3. Rotation of Velocities

To simplify the computation of the velocity rotation, the linked cell list mentioned in section 3.1 is used to sort particles into the correct cell of the collision cell grid. Using the *com* velocity, gathered by one of the two previously described methods, the linked-cell list provides the particles which belong to the given cell and their velocity vector rotated.

Listing 4: Using the linked cell list from listing 1 to compute the com velocity

```
// Kokkos parallel_for iterates over
// all cells on local domain
// vcm = Kokkos::View containing the
//       center of mass velocites
//       for each collision cell
// v   = Cabana::slice containing
//       particle velocities
// m   = Cabana::slice containing
//       particles masses
Kokkos::parallel_for(
  Kokkos::RangePolicy<ExecutionSpace>
  (0, linkedList.totalBins()),
  KOKKOS_LAMBDA( const size_t i)
  {
    int ix, iy, iz;
    // computing the cartesian
    // coordinates of the cell
    linkedList.ijkBinIndex(i,
                         ix, iy, iz);
    int binOff =
      linkedList.binOffset(ix, iy, iz);
    // compute com velocity
    for (int d = 0; d < 4; ++d)
      vcm(ix,iy,iz,d) = 0.0;
    // computing com momentum and sum of mass
    for (int n = 0;
         n < linkedList.binSize(ix,iy,iz);
         ++n)
    {
      for (int d = 0; d < 3; ++d)
        vcm(ix,iy,iz,d) += v(binOff + n, d) *
                           m(binOff + n);
      vcm(ix,iy,iz,3) += m(binOff + n);
    }
  });
Kokkos::fence();
```

3.4. First Analysis and Improvements

During the development and implementation of the code and performing first benchmarks, it was noticed that the performance was not showing expected behaviour. When using small system sizes the runtimes for runs using GPUs were massively slower than expected.

Using performance analysis tools, such as NVIDIA Nsight Compute [16], it was discovered, that one problem of the first implementation was the use of unified memory for data storage, especially for the particle data. This led to problems during the transfer of data, as according to the analysis results, data was transferred between host and device a lot more than expected. After restructuring the code to mainly use device based memory, these unintended transfers vanished and where applicable device-to-device communication between different GPUs was used. For upcoming versions of Kokkos we will reinvestigate, whether the discrepancy between the different memory models is still present or if the issue has been solved.

In Fig. 2 the results for one of these benchmarks is shown. It can be seen that for the implementation using unified memory, the majority of time is spent in the halo exchange, which takes place in every time step. The shown example is a quasi-static fluid, so no particles are transferred in the scope of the simulation between different processes. It is assumed that this would increase the discrepancies between the two implementations further.

4. Benchmarks and Discussion

For the benchmark runs basic fluid systems were used, i.e. a pure MPCD fluid in 3d periodic boundary conditions. Each cubic collision cell has an edge length of one length unit, while containing $\langle N_c \rangle = 10$ particles on average. Each system in the benchmarks is cubic with side length L (the edge length L given as the system size in the following graphs, i.e. Fig. 3, from where the total number of particles in a system is computed as $N = L^3 \langle N_c \rangle$.
The Juwels booster module [17] at Jülich Supercomputing Centre consists of GPU nodes with four NVIDIA A100 cards and two AMD EPYC 7402 processors, with 24 cores each. To maintain comparability of the benchmarks the pure CPU runs were also performed on these nodes. In addition testbed nodes on Jureca-DC were used, which provided access to two additional hardware architectures, first the AMD MI 250 GPU, second to an ARM platform consisting of Ampere Altra Q80-30 CPUs.
The code was compiled with the GNU compiler suite version 11.2, OpenMPI v. 4.1.2, Kokkos 3.5.00 and Cabana v. 1.0-dev. Kokkos and Cabana were configured with the ”RelWithDebugInfo” settings in order to be able to backtrace possible errors during execution. No further compiler flag based optimisation has been taken into account since our main focus was the comparison of runtime and scalability of the function portable implementation of the Kokkos / Cabana code. In future we plan a more in depth comparison of the Kokkos / Cabana implementation against machine optimised versions of the code, e.g. using OpenMP + CUDA, OpenACC + GPU offloading and vendor specific compilers. This will provide further information of a possible performance loss of a Kokkos / Cabana implementation in comparison to an optimised one.
As backends, the `AMD` and `Ampere70` flags were used in Kokkos since these fit to the architecture of the Juwels booster nodes. For the benchmark runs on the Jureca-DC testbed nodes, the `VEGA90A` backend was used for the runs on the AMD MI 250 card, while for the ARM node the `ARMv81` backend flag was employed.
Due to the limited availability of AMD profiling features on the benchmark systems, roofline model measurements were undertaken with Intel Advisor on the cluster part of Juwels, using the Intel Xeon Platinum 8168 processors. The resulting roofline model is reported in Fig. 6. Results show that the code is mainly memory bound due to many accesses to DRAM while on the same time showing only a low algorithmic intensity (compute operations per transferred byte). This can be traced back to the main characteristic of the method and algorithm. Since particles have to be sorted in every timestep into a shifted mesh (cmp. Sec. 2) data reusage is minimal. This could be improved by increasing data locality and to combine different compute kernels, which, on the other side would require a reformula-

Figure 2: Benchmarks for different memory models used in the implementation for a system of base length 32.

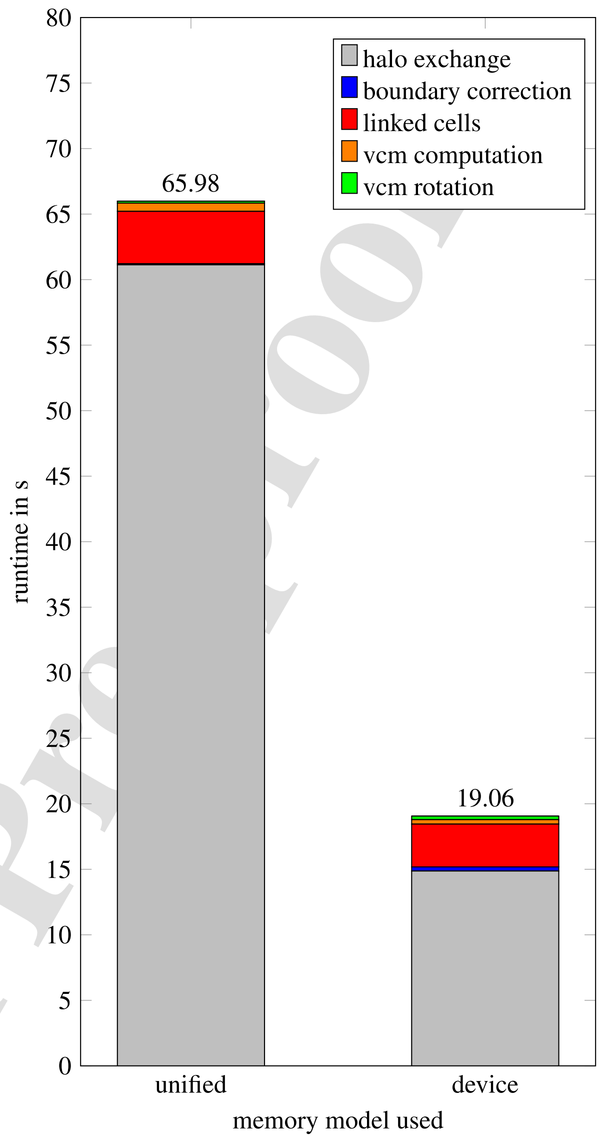


Figure 3: Benchmarks comparing different modes of execution and different system sizes to each other. All benchmarks are run on a single node.

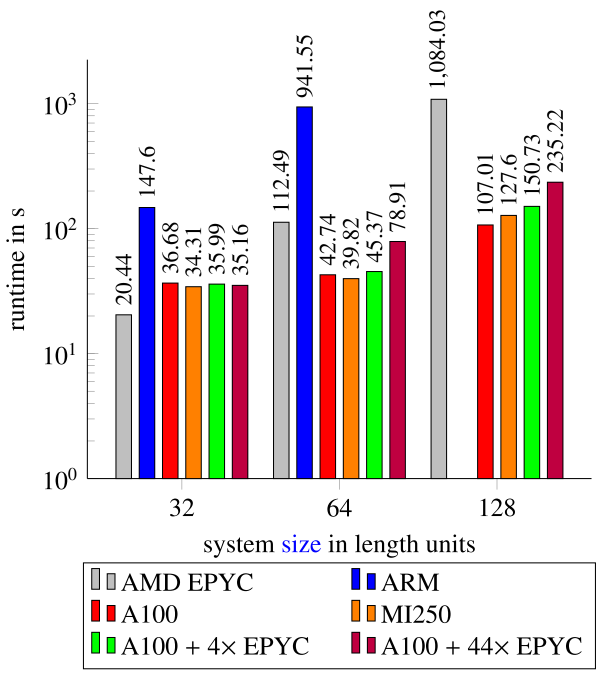
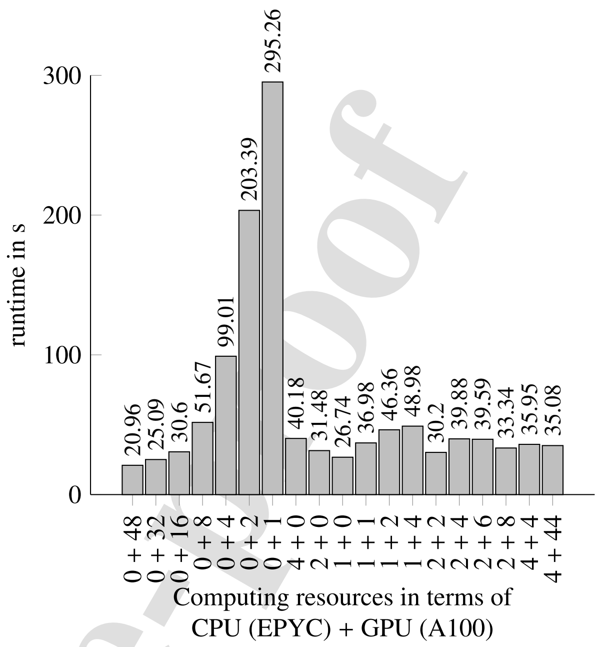


Figure 4: Benchmarks comparing different modes of execution one a single node for the system with system length 32, using different subsets of CPUs and GPUs, as well as combinations.



tion of the underlying algorithm, i.e. a rewrite of the code. An additional performance analysis with Nsight Compute on the A100 of the booster partition of Juwels has been conducted, which showed a comparable result, i.e. performance limited by DRAM bandwidth. In Fig. 7 the roofline model for the A100 is presented.
In order to improve code performance in future releases, changes to the algorithm are required, which enable vectorisation and improve memory access strategies. The best performing kernels in the roofline models are those, where particles are accessed linearly in memory and then processed independently of each other, e.g. the integration of particle velocities or checks if periodic boundary corrections need to be applied.

Additional benchmarks were conducted on a single node with a constant number of processes used:

ARM	4 processes with 20 threads
MI250	8 processes using 4 AMD MI 250 GPUs
A100	4 processes using a single A100 GPU
EPYC	48 processes with two threads each
A100 + x	4 processes each employing an A100 and $x \in \{4, 44\}$ processes on AMD EPYC CPUs running with two threads each

Results of the benchmarks are reported in Fig. 3. In the figure, six different bars each represent a different execution method (cmp. text above). All bars are based on runtimes measured with the Cabana implementation. MI250 and A100 are runtimes measured for cases where code is executed only on the respective GPUs. It can be seen that for the system sizes under consideration, both types of GPUs are not fully utilised. This can be concluded from Fig. 5, where it can be seen that for the small system sizes, runtimes for multiple GPUs (A100s) are not decreasing, but, in contrast, even increase. This observation can be related to an increase in communication. For system sizes larger than 128^3 it can be seen that a single A100 can be utilized until hitting the bandwidth limitation, discussed before. I.e. using additional A100s now can decrease the runtime. For sufficiently large systems good scalability is found for strong scaling on the A100 cards. Due to the fact that only one node with MI250s was available at the time of writing this article no strong scaling results for this architecture could be gathered.

Executing the code on a combination of A100s and processes on EPYC CPUs resulted in an execution time, that is oriented towards the execution time of the slower of both architectures, since this is the bottleneck for this kind of combined execution. Benchmarks for small system sizes show execution times, which are on par with executions on single hardware architectures. On the other side, for large system sizes, the runtime increases such that a combined execution is not competitive with respect to the execution on the faster architecture alone, usually the GPU. Also, it can be seen that using only four processes on EPYC CPUs results in shorter runtimes than using all remaining available cores on the node (i.e. 44 cores).
The reason for that is the increased communication overhead and restricted process grid topology. For larger system sizes the number of CPU processes gets irrelevant. The huge difference in computing power between the different architectures becomes dominant. In order to balance the compute load be-

Figure 5: Strong scaling behaviour of the Cabana code using multiple NVIDIA A100 GPUs for different systems sizes (edge length 32 to 512). Up to 4 GPUs using a single node, then 4 GPUs per node. The largest test case was not run on a single node due insufficient memory.

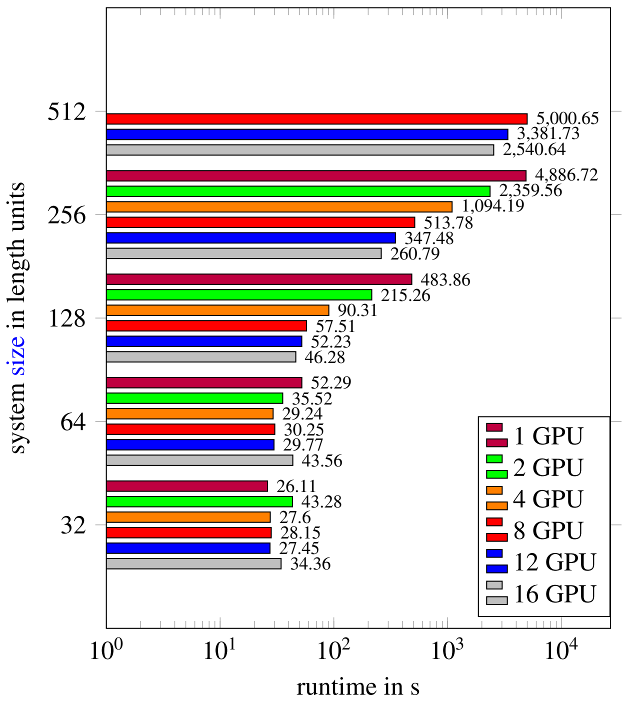


Figure 6: Roofline model from Intel Advisor for Xeon nodes on Juwels cluster part. The measured kernel orient around the lowest roofline, indicating the DRAM bandwith of 103.8 GB/s.

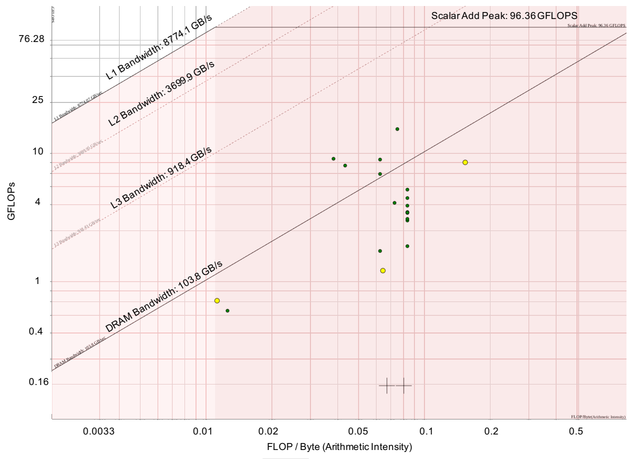
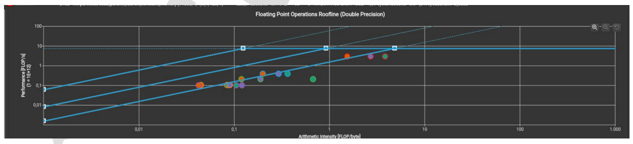


Figure 7: Roofline model from Nsight Compute for NVIDIA A100 on Juwels Booster. In this case the measured kernels are also bounded by the lowest memory bandwidth roofline.



tween CPU and GPU, compute domains on the devices have to be adjusted in size according to relative compute power. As a result, balancing can only nearly be achieved by assigning domains of the smallest possible size to the CPU processes. Accordingly the remaining share is distributed to the GPU processes. Due to the topological restriction of the process grid, this forms a single plane of GPU processes in which the majority of the system is divided into a $2 \times 2 \times 1$ grid. As a consequence, the processes on the EPYC CPUs are also put into a $2 \times 2 \times n_z$ grid ($n_z = 1$ for four, $n_z = 11$ for 44 EPYC processes). The larger the number of used processes on the EPYC CPU the more disadvantageous the distribution becomes. Since the size of the CPU processes in the last dimension is defined as the minimum possible size which a domain can have without necessitating halo exchanges with over-next neighbours, the CPU processes become really thin domains with a huge surface over which particle information needs to be exchanged. This is true for both the halo exchanges as well as the particle exchanges due to particle movement. This results in unnecessary overhead and should therefore be avoided.

A possible solution to this issue is to subdivide the two parts of the system in a different manner. E.g. a subdivision of the GPU domains into a $2 \times 2 \times 1$ grid, while using 40 processes on the EPYC CPU in, e.g., a $8 \times 5 \times 1$ distribution which results in a better communication surface between the CPUs. Also this allows to allocate a smaller share of the system to the CPUs, in order to better balance the load between the two partitions, since the stacking of CPU domains requires a larger share of the system to be assigned to the CPU part. This improvement could not yet be implemented, since Cabana does not yet support non-uniform process grids.

5. Conclusion and Outlook

The target of this work was to investigate if combined execution of the MPCD code on heterogeneous can improve the performance that can be achieved on such a node. Furthermore, the parallel scaling behaviour of the Cabana implementation was analyzed. While it was seen that the code scales reasonably well, given a sufficiently large test case, both on CPU and GPU, it was also shown that in the current implementation it was not possible to improve the performance of the code by combining GPU and CPU execution on heterogeneous nodes. It remains to be seen if this can be improved by implementing the possibility to provide CPU and GPU partitions of the code with different topologies for domain decomposition, which will be an interesting point of research.

Another future point of investigation will be the addition of Molecular Dynamics (MD) simulation to the MPCD implementation, so that embedded systems can be simulated. Since MD and MPCD show different scaling behaviour it might be useful to distribute the computation of MD and MPCD between GPU and CPU. This can either be done with the model shown in this paper, running the code on the same node and using the GPU and CPU resources, or could be achieved by modular execution on different computing system partitions.

Acknowledgements

This work was performed as part of the Co-design Center for Particle Applications, supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. DOE Office of Science and the NNSA. Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy (Contract No. 89233218NCA000001). G.S. and R.H. thank LANL for hospitality in the summers of 2018&2019. We also acknowledge Jülich Supercomputing Centre for providing access to the JURECA DC Evaluation Platform, including AMD Instinct MI250 and NVIDIA Arm HPC Devkits used in this work [18]. Furthermore, we thank the developers of the Cabana library for their support and consultations during the time of implementation and providing assistance during the analysis of the benchmarking results.

References

[1] J. Dongarra, P. Luszczek, TOP500, Springer US, Boston, MA, 2011, Ch. 755, pp. 2055–2057. doi:10.1007/978-0-387-09766-4_157.

[2] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202 – 3216, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. doi:https://doi.org/10.1016/j.jpdc.2014.07.003.

[3] RAJA Performance Portability Layer, website.
URL https://github.com/LLNL/RAJA

[4] G. Gompper, T. Ihle, D. M. Kroll, R. G. Winkler, Multi-Particle Collision Dynamics: A Particle-Based Mesoscale Simulation Approach to the Hydrodynamics of Complex Fluids, in: Advanced Computer Simulation Approaches for Soft Matter Sciences III, Springer Berlin Heidelberg, 2008, pp. 1–87. doi:10.1007/978-3-540-87706-6_1.

[5] Cabana, website.
URL https://github.com/ECP-copa/Cabana

[6] S. M. Mniszewski, J. Belak, J.-L. Fattebert, C. F. Negre, S. R. Slattery, A. A. Adedoyin, R. F. Bird, C. Chang, G. Chen, S. Ethier, S. Fogerty, S. Habib, C. Junghans, D. Lebrun-Grandié, J. Mohd-Yusof, S. G. Moore, D. Osei-Kuffuor, S. J. Plimpton, A. Pope, S. T. Reeve, L. Ricketson, A. Scheinberg, A. Y. Sharma, M. E. Wall, Enabling particle applications for exascale computing platforms, The International Journal of High Performance Computing Applications 35 (6) (2021) 572–597. doi:10.1177/10943420211022829.

[7] S. Slattery, S. Reeve, C. Junghans, D. Lebrun-Grandie, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, S. Mniszewski, Cabana: A Performance Portable Library for Particle-Based Simulations, J. Open Source Software 7 (72) (2022) 4115. doi:10.21105/joss.04115.

[8] R. Halver, C. Junghans, G. Sutmann, Kokkos-based implementation of mpcd on heterogeneous nodes, in: R. Wyrzykowski, J. Dongarra, E. Deelman, K. Karczewski (Eds.), Parallel Processing and Applied Mathematics, Springer International Publishing, Cham, 2023, pp. 3–13.

[9] C. Huang, R. Winkler, G. Sutmann, G. Gompper, Semidilute Polymer Solutions at Equilibrium and under Shear Flow, Macromol. 43 (2010) 10107–10116.

[10] Exascale Computing Project, website.
URL https://www.exascaleproject.org/

[11] Co-Design Center for Particle Applications, website.
URL https://www.exascaleproject.org/research-project/particle-based-applications/

[12] R. Halver, G. Sutmann, Multi-Threaded Construction of Neighbour Lists for Particle Systems in OpenMP, in: Parallel Processing and Applied Mathematics 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part II, 11th International Conference on Parallel Processing and Applied Mathematics, Krakow (Poland), 6 Sep 2015 - 9 Sep 2015, 2015, pp. 153–165.
URL https://juser.fz-juelich.de/record/279249

[13] K. Ohno, T. Nitta, H. Nakai, SPH-based Fluid Simulation on GPU Using Verlet List and Subdivided Cell-Linked List, in: 2017 Fifth International Symposium on Computing and Networking (CANDAR), 2017, pp. 132–138. doi:10.1109/CANDAR.2017.104.

[14] H. Goldstein, C. Poole, J. Safko, Classical Mechanics, Addison Wesley, San Francisco, CA, 2002.

[15] R. Halver, S. Schulz, G. Sutmann, ALL - A loadbalancing library, C++ / Fortran library, website.
URL https://gitlab.version.fz-juelich.de/SLMS/loadbalancing/-/releases

[16] Nvidia nsight compute documentation, website.
URL https://docs.nvidia.com/nsight-compute/NsightCompute/index.html

[17] Juwels, website.
URL https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/JUWELS_node.html

[18] P. Thörnig, B. von St. Vieth, JURECA: Data Centric and Booster Modules implementing the Modular Supercomputing Architecture at Jülich Supercomputing Centre, Journal of large-scale research facilities 7 (2021) A182. doi:10.17815/jlsrf-7-182.
URL https://juser.fz-juelich.de/record/902211

Highlights of the present work

1. The implementation of a particle simulation code is shown to profit significantly by employing the performance portability framework Kokkos via including implicitly OpenMP shared memory parallelism without any programming overhead.
2. The transition between different CPU and GPU architectures and their hybrid combination is enabled with very low programming overhead.
3. The Cabana library provides an easy way to bridge CPU and GPU architectures in the same program execution and enables hybrid and modular computing.

Declaration of interests

☒ 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.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: