Check for updates

Received: 22 December 2022 | Revised: 3 July 2023 | Accepted: 5 July 2023
DOI: 10.1002/cpe.7870

RESEARCH ARTICLE

WILEY

# A portable C++ library for memory and compute abstraction on multi-core CPUs and GPUs

Pietro Incardona[1,2,3,4] | Aryaman Gupta[1,2,3] | Serhii Yaskovets[1,2,3] | Ivo F. Sbalzarini[1,2,3]

[1]Faculty of Computer Science, Technische Universität Dresden, Dresden, Germany
[2]Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany
[3]Center for Systems Biology Dresden, Dresden, Germany
[4]Center for Scalable Data Analytics and Artificial Intelligence ScaDS.AI, Leipzig, Germany

**Correspondence**
Ivo F. Sbalzarini, CSBD, Pfotenhauerstr. 108, D-01307 Dresden, Germany.
Email: sbalzarini@mpi-cbg.de

**Present address**
Ivo F. Sbalzarini, IGSB, University Hospital Bonn, Bonn, Germany

**Funding information**
Bundesministerium für Bildung und Forschung, Grant/Award Numbers: 01/S18026A-F, 031L0160

## Abstract
We present a C++ library for transparent memory and compute abstraction across CPU and GPU architectures. Our library combines generic data structures like vectors, multi-dimensional arrays, maps, graphs, and sparse grids with basic generic algorithms like arbitrary-dimensional convolutions, copying, merging, sorting, prefix sum, reductions, neighbor search, and filtering. The memory layout of the data structures is adapted at compile time using C++ tuples with optional memory double-mapping between host and device and the capability of using memory managed by external libraries with no data copying. We combine this transparent memory layout with generic thread-parallel algorithms under two alternative common interfaces: a CUDA-like kernel interface and a lambda-function interface. We quantify the memory and compute performance and portability of our implementation using micro-benchmarks, showing that the abstractions introduce negligible performance overhead, and we compare performance against the current state of the art in a real-world scientific application from computational fluid mechanics.

**KEYWORDS**
C++ tuples, generic algorithms, GPU, memory layout, multi-core, performance portability

## 1 | INTRODUCTION

Performance portability and maintainability of thread-parallel codes are rapidly gaining importance as hardware is becoming more heterogeneous. With Graphics Processing Units (GPU) now commonplace, new accelerator architectures from Nvidia, AMD, and Intel are about to enter the market. Additionally, the landscape of multi-core Central Processing Units (CPU) is diversifying with x86_64/amd64 increasingly joined by ARM and POWER. Porting code to new hardware costs valuable developer time due to the large semantic gap between different hardware-specific programming models. There is thus an urgent need for code that runs across a variety of hardware platforms with minimal or no changes, while still achieving state-of-the-art performance. Going forward, this will be crucial for maintainability of software.

Typically, portability and maintainability are achieved in software engineering by abstraction.[1] This has been successfully demonstrated also for performance portability, for example by libraries like Kokkos,[2,3] Alpaka,[4] and RAJA,[5] as well as Intel's OneAPI built on top of SYCL.[6] These libraries provide abstractions to execute code across hardware architectures. While providing a good variety of data structures as containers, these libraries are, however, limited in having containers that fit the requirements of a particular hardware, in particular if the object is a not a primitive type (memory layout restructuring). Libraries like LLAMA,[7] used in Alpaka,[4] therefore provide more complex memory layout restructuring across hardware platforms, but are in turn limited to multi-dimensional arrays as the only container type they support. The data structures created by such libraries also typically use a single memory type, for example, they are stored either on the CPU or on the GPU. Moreover, most of the existing

This is an open access article under the terms of the Creative Commons Attribution-NonCommercial License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited and is not used for commercial purposes.
© 2023 The Authors. *Concurrency and Computation: Practice and Experience* published by John Wiley & Sons Ltd.

libraries currently lack the capability of combining data structures with tuples of types, where the tuples are parsed to generate a specific memory layout for the container (i.e., for tuple-based layout switching). Additionally all of them lack support for sparse data structures or the possibility to automatically serialize/deserialize arbitrarily nested levels of containers. These limitations currently prevent their use in applications involving complex, high-dimensional, or sparse data, as is increasingly the case in computer simulations, data science, and machine learning.

Here, we address this gap by providing an open-source memory- and compute-abstraction library that supports arbitrarily nested and sparse tuple data structures mapped to different memory layouts, as well as commonly used basic algorithms tuned for performance on a variety of hardware targets. Our library is implemented using C++14 tuples (see Section 2) for compile-time code generation of generic scalar, vector, and multi-dimensional tensor arrays, in addition to more complex data structures like compressed-sparse-row graphs and arbitrary-dimensional sparse block grids.[7] Our library uses double-mapping to support data structures that simultaneously exist on both device and host, enabling user codes to, for example, have CPU and GPU sections share an abstract data structure simultaneously mapped to both memories. Along with the abstract data structures, we provide optimized generic algorithms, for example, for arbitrary-dimensional convolutions, neighborhood search, copying, merging, sorting, prefix sum, reduction, and filtering (Section 3).

The presented library, `openfpm_data`, is available as part of the OpenFPM scalable computing project.[9] It provides the shared-memory layer of OpenFPM, but can also be used as a stand-alone library. It provides two interfaces for user-implemented algorithms over abstract data structures: CUDA-like compute kernels and lambda functions. Since `openfpm_data` is able to use any provided external memory to construct compile-time data structures, it seamlessly interfaces with other libraries (Section 4) that provide algorithms or shape memory, like Kokkos[2,3] or LLAMA.[7] As such, we intend `openfpm_data` to integrate between existing solutions, rather than to replace them, supplementing them with functionality like sparse grids, graphs, and generic neighborhood search.

We show in micro-benchmarks and in a real-world application that the flexibility and portability afforded by `openfpm_data` does not impact performance (Section 5). Indeed, we find that combining memory layout restructuring of complex data structures with generic algorithms under the same interface can benefit the performance optimizations of modern C++ compilers on multiple CPU and GPU architectures. We conclude the paper in Section 6.

## 2 | FROM C++ TUPLES TO COMPILE-TIME DATA STRUCTURES

We construct memory-layout reconfigurable data structures with a common abstract programming interface by exploiting two features of the C++ programming language: The first is the existence of three types of brackets — `<>`, `()`, and `[]`. We use them to cleanly separate the semantics of data structures. Angle braces are used to specify which property of a tuple/composite data structure one wants to access. Round parentheses are used to specify an element of a discrete set. Square brackets are used to access individual components of a vector or array. This three-brackets access semantic is common across all `openfpm_data` data structures and independent of the physical memory layout used.

The second C++ feature we use are tuples (and, consequently, variadic templates). We use the tuple data structure provided by the Boost library[*] to define properties or elements of an `openfpm_data` data structure. Using tuples instead of structs enables content parsing at compile time using template meta-programming[10,11] to define the implementation of a data structure. Template meta-programming uses the C++ template pre-processor to implement code-generation logic for the compiler in so-called meta-algorithms.[11] We use meta-algorithms to determine the memory address of each tuple element of a container (i.e., the memory mapping) at compile time, enabling memory layout restructuring. We then construct an object that stores the information about a container with the specified layout and injects the appropriate data-access methods with layout-specific code required to overload the three parenthesis operators for memory mapping.

### 2.1 | Data structures and memory layouts

The data structures and memory layouts available in `openfpm_data` are summarized in Figure 1. The UML diagram on the left shows the composition of the available containers, starting from the base class "multi-dimensional array". A vector is a one-dimensional array, a Compressed Sparse Row (CSR) graph is stored in an encapsulated vector of vertices and edges, a map is a sorted vector, and a sparse grid is an $n$-dimensional map.[8] All sub-classes inherit the layout reconfigurability of the base class as defined by the four template parameters (distinguished by different colors) shown in the right box. Every container in the hierarchy can override every layout parameter, leading to a combinatorial diversity of possible implementations. At the time of writing, there are two different mappings for the `<>` operator (red in Figure 1) and five that control the linearization for the `()` operator (violet, two examples shown). Since the component access operator `[]` is uniquely defined, this makes a total of 10 memory mappings.

The best layout for a given data structure depends on both the hardware backend and the algorithm to be used on that data structure. By default, `openfpm_data` selects an Array-of-Struct (AoS) layout for data structures on the CPU, while for GPU data structures the default is Struct-of-Arrays (SoA) as they generally improve memory coalescence. The default can be overridden and fine-tuned by the user passing any combination of template parameters to select the implementations of the `<>` and `()` operators. This can be used to account for additional knowledge about the algorithm or the structure of the input data.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

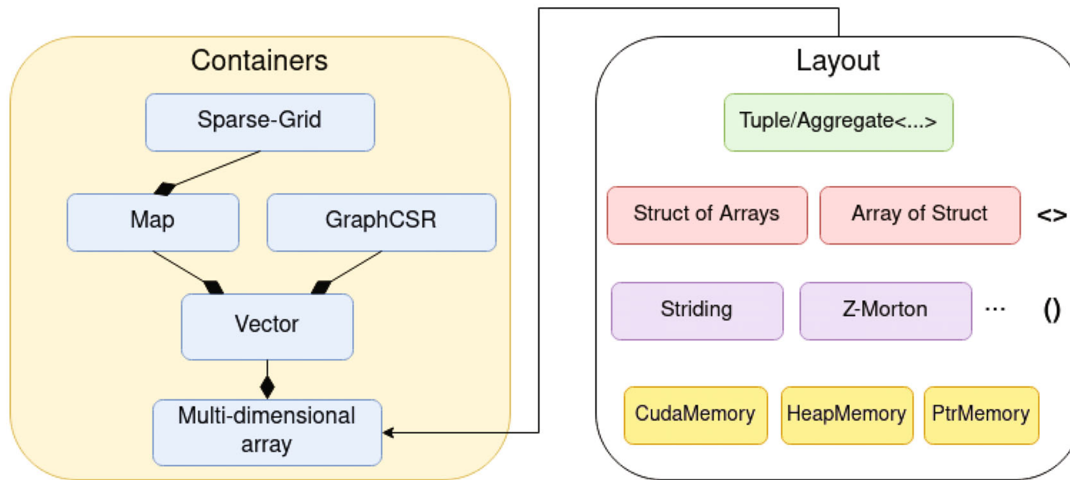


**FIGURE 1** Summary of the `openfpm_data` library: The UML diagram on the left lists the implemented containers and their composition, starting from multi-dimensional arrays, with template parameters as listed in the right box. The first template parameter (green) is the tuple defining the data type of the container. The memory layout is defined in the second parameter (red) for the <> bracket with two current implementations. The linearization of multi-dimensional indices () is defined by the third template parameter (violet), where two of the currently available five implementations are shown exemplarily. The fourth template argument (yellow) defines the type of memory to be allocated: GPU device (Nvidia or AMD), heap memory, or external memory.

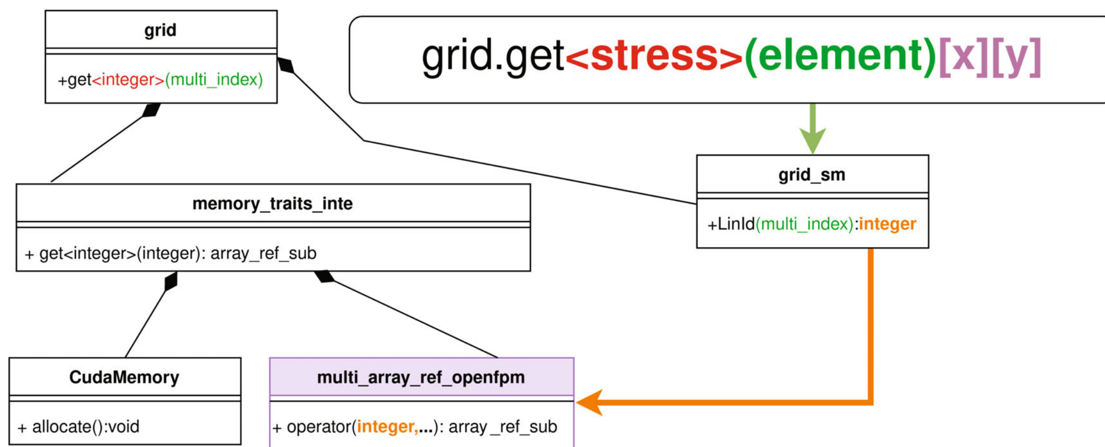


**FIGURE 2** Example to illustrate the classes involved in accessing an element of a Struct-of-Arrays (SoA) container in GPU memory with standard C++ striding linearization for the () operator. The figure illustrates how the method `grid.get<stress>(element)[x][y]` is implemented across classes using the three bracket types of C++. Colors of arrows and parameters match the parenthesis and in-parenthesis parameter colors. In the example of the figure, the component `[x][y]` (two-dimensional tensor index) of the element `(element)` of a named property `<stress>` is accessed. This is how one would access the components of a stress tensor field in a fluid mechanics simulation. The operator `()` is overloaded by `grid_sm` (green arrow), which converts the multi-index to an integer (orange) using standard C++ striding. This integer is passed to `multi_array_ref_openfpm`, which overloads the `[]` operator. The class `memory_traits_inte` implements the interleaved memory layout for SoA with memory allocated on the GPU in the `CudaMemory` object, which is in turn used to store the grid object.

Figure 2 illustrates by example the mechanism used for resolving memory addresses so as to render data access independent from the memory mapping (abstract layout switching). In the example of the figure, the object `memory_traits_inte` implements the meta-algorithm to transform a tuple into a multi-dimensional container object with interleaved (i.e., Struct of Arrays) memory layout, and it contains the code for the parentheses functions. The figure also shows how the parentheses are used to calculate a memory address once the property in `<>`, the element in `()`, and the components in `[]` have been specified. In the figure, this is shown for the example of a `get` method on a multi-dimensional array named `grid` to access tensor component $[x, y]$ of a certain element of a container called `stress` (e.g., the stress tensor field of a fluid mechanics simulation). All layout-specific code is encapsulated in the objects that overload the parenthesis operators, as indicated by the colors.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

### 2.2 | Double-mapped data structures

We distinguish single-map data structures, which are mapped to a memory layout on one device, and double-map data structures, which are simultaneously mapped (possibly using different layouts) to two physically separate memories. Thus, all `openfpm_data` data structures can use host memory, device memory, or both simultaneously. Double-mapped data structures can simplify code where some sections run, for example, on a CPU and others on a GPU. A single double-mapped data structure then replaces two separate single-mapped data structures for the host and the device. Multi-socket devices are supported using a kernel that copies the data from the host to the devices. This guarantees that the memory pages allocated by any given socket will be used by that same socket in all subsequent kernels, reducing NUMA accesses across sockets.

However, `openfpm_data` does not provide any memory consistency model. This means that `openfpm_data` does not attempt any automatic or implicit communication or transfer of data. Data transfer between the host and device memory of a double-mapped data structure needs to be explicitly triggered by the user program when needed (synchronization of a double-mapped data structure). Functions to conveniently move data from host to device, and vice versa, are provided. If the two maps of a double-mapped data structure use different memory layouts, these functions also automatically and transparently convert the data from one layout to the other.

## 3 | GENERIC ALGORITHMS FOR PERFORMANCE PORTABILITY

We complement the hardware-independent data structures and memory layout capabilities of `openfpm_data` with generic algorithms optimized for massively parallel architectures. This includes commonly used primitives of parallel computing[12,13] as listed in Table 1. All of these are translated to optimized hardware-specific implementations at compile time using the `openfpm_data` hardware backends. Any program that can be written as a combination of parallel kernels with these primitives becomes performance portable and scalable. This set of "stock" algorithms can be extended by user-implemented algorithms.

### 3.1 | User-implemented algorithms

We expose two different interfaces for user-implemented algorithms: a CUDA-like kernel interface and a lambda function interface. Like in CUDA, `openfpm_data` kernels are labeled with the attribute `__global__` and device functions are labeled with the attribute `__device__`. Also like in CUDA, computation is divided into a grid of blocks, where each block contains a user-defined number of threads. Within a kernel, `openfpm_data` provides the local variables `blockIdx`, `blockDim`, `threadIdx`, and `gridDim` that contain the thread block index, the block dimension, the thread index within the block, and the number of blocks in the grid. Static shared memory is marked with `__shared__`.

To illustrate the similarity of the `openfpm_data` kernel programming interface with CUDA, and to provide an example of how user-defined algorithms can be implemented, List 1 shows the first part (defining the shared memory and loading the fields) of the miniBUDE benchmark[14] implemented as an `openfpm_data` kernel that can run on both CPUs and GPUs, along with the code required to launch the kernel using the CUDA-like interface of `openfpm_data` (Lines 20–24).

**TABLE 1** Intrinsic generic algorithms provided by `openfpm_data` at the time of writing.

| |
|---|
| Atomic add |
| Prefix sum |
| In-warp exclusive prefix sum (`blockScan`) |
| Segmented reduce |
| In-warp reduce |
| Adding and removing elements from maps |
| Data structure copying and merging |
| Neighbor search using cell-lists |
| Sorting |
| Multi-dimensional stencils operations |
| Multi-dimensional convolutions |

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
////// CUDA-LIKE KERNEL INTERFACE

template<typename vector_atom, ... >
__global__ void fasten_main( ...
    const vector_atom protein_molecule,
    const vector_atom ligand_molecule, ...) {
  // Compute first index
  int ix = blockIdx.x*blockDim.x*N_TD_PER_THR + threadIdx.x;
  int tid = threadIdx.x;
  ix = ix < numTransforms ? ix : numTransforms - N_TD_PER_THR

#ifdef USE_SHARED
  __shared__ FFParams forcefield[N_ATOM_TYPES];
  if(tid < num_atom_types) {
    forcefield[tid].hbtype = ...; forcefield[tid].radius = ...;
  }
#endif
 ...
}

CUDA_LAUNCH_DIM3(fasten_main,global,local,
        ...
        d_protein.toKernel(),
        d_ligand.toKernel(),
        ...);
```

Listing 1: Example of an `openfpm_data` compute kernel able to run on both GPU and CPU. The listing shows the first part of the miniBUDE benchmark 13 and the code to launch the kernel using the CUDA-like interface, with the backend-specific kernel launch syntax abstracted by the macro `CUDA_LAUNCH_DIM3`.

```
////// LAMBDA FUNCTION INTERFACE

auto lamb = [... d_protein.toKernel(),d_ligand.toKernel(), ...] __device__ \
                                    (dim3 & blockIdx, dim3 & threadIdx)
{
 ...
};

CUDA_LAUNCH_LAMBDA(ite,lamb);
```

Listing 2: Example of how to launch the kernel from List 1 on using the lambda interface of `openfpm_data`.

For lambda-based computation, `openfpm_data` supports directly launching a lambda function similar to libraries like Kokkos,[2,3] RAJA,[5] and SYCL.[6] The `blockIdx` and `threadIdx` constants are passed to the function as arguments, as illustrated in List 2). This implies that TLS for the OpenMP backend is not required, because `blockIdx` and `threadIdx` are local function arguments rather than global variables.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

## 3.2 Hardware backends

In order for openfpm_data kernels to run on different hardware, openfpm_devices provide hardware-native implementations of every algorithmic primitive from Table 1. These hardware-specific implementations can be selected at compile time without changes to the user code. All changes are encapsulated in C++ objects that determine the specific implementation of an algorithm for a given hardware backend (switchable backends). At the time of writing, the following backends are available in openfpm_data: CUDA (Nvidia GPU), HIP (AMD GPU), SEQUENTIAL (CPU), and OpenMP (CPU). The backend is chosen by the user at compile time.

For the **CUDA backend**, openfpm_data uses the optimized algorithm implementations from the header-only C++ CUDA library moderngpu[15] and from the Nvidia header library CUB. RAJA also uses CUB as its backend for CUDA, while Kokkos has its own implementations. The moderngpu library provides traditional bulk synchronous parallel (BSP) general-purpose functions in addition to templated pattern functions. These kernel primitives support argument passing with lambda capture or using variadic arguments with automatic restrict tagging of pointers. The most important algorithmic primitives provided by moderngpu are listed in Table 2.

For the **HIP backend**, the openfpm_data algorithms directly wrap the corresponding implementations from AMD's hipCUB API and RadeonOpenCompute (ROCm).

The **SEQUENTIAL backend** executes each block sequentially on the CPU. Then, `__global__` and `__device__` map at preprocessor level to an empty string and an inline, respectively, and blockIdx, blockDim, threadIdx, and gridDim are global variables. The global variables blockDim and gridDim are recomputed every time the kernel launches before looping over the blocks. The variables blockIdx and threadIdx are set in each iteration of the loop. `__syncthreads()` is implemented with lightweight threads (number of threads = size of the thread block). Each thread has 8 KB of stack memory by default, adjustable via a compile-time parameter, and supports fast context switching. Every time `__syncthreads()` is encountered, execution is stopped and a fast context switch is performed, moving to the next lightweight thread. While this leads to sub-optimal performance, it provides a direct mapping for user-defined kernels where no backend-native implementation is available to at least run (e.g., for debugging). When reaching the end of a block, the first lightweight thread in the block is resumed in a cyclic way. The threads are created internally in the SEQUENTIAL backend, while fast context switching is performed using the Boost library's boost::context. Because lightweight threads are not concurrent, atomicAdd reduces to a regular addition operation. A block scan is implemented as a `__syncthreads()` followed by the calculation of the exclusive prefix sum for thread zero in the block and a final `__syncthreads()`. The use of lightweight threads in the SEQUENTIAL backend is necessary to support the thread-block programming model. Unnecessarily using this model, however, impedes performance. It prevents the compiler from using vectorization and optimization across iterations because a context switch happens at every iteration. This problem is avoided in openfpm_data by always forcing the compilation of two versions of each kernel: one with lightweight threads and one without. Then, openfpm_data starts executing one block using the lightweight threads implementation. If after one block the library did not detect any context switch, it changes to the other implementation, where the compiler was able to apply the optimizations.

In the **OpenMP backend**, blockIdx and threadIdx are marked thread_local and use thread-local storage (TLS) in order to have an independent copy for each thread. Blocks are distributed across OpenMP workgroups, with each thread of a block executed by one OpenMP thread. Again, if blocks do not use `__syncthreads()`, the backend automatically switches to non-lightweight threads to enable vectorization and facilitate compiler optimizations. The TLS mechanism incurs an overhead for small kernels, but the benchmarks of Section 5 show that the effect of this overhead becomes negligible for memory-bound applications. This is because having more cores than memory channels compensates for a thread being slower due to TLS. For compute-bound applications, however, performance degradation will depend on the timing ratio between the TLS mechanism and the compute kernel.

**TABLE 2** Parallel computing primitives provided by the moderngpu library for the CUDA backend.

| Templated pattern functions | BSP functions |
|---|---|
| transform | reduce |
| transform_reduce | scan |
| transform_scan | merge |
| transform_lbs | bulk_remove |
| lbs_segreduce | bulk_insert |
| | mergesort |
| | segmented_sort |
| | sorted_search |

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

## 4 | INTEROPERABILITY

The openfpm_data library is intended to seamlessly integrate with existing abstraction libraries and frameworks, supplementing them with functionality like sparse grids, graphs, and fast neighborhood search not otherwise available. This renders interoperability of data structures a primary design goal. The interoperability of openfpm_data rests on the concept of "absorbing" external memory, that is, transparently using memory allocated by other software. This memory absorbing is, for example, useful to transfer coded functions from openfpm_data to other frameworks, or to benefit from memory layout capabilities of other frameworks. Conceptually, this amounts to constructing data structures around memory that has not been allocated by that data structure itself, but is "externally" managed. This is achieved in openfpm_data by declaring the data structure with the special memory allocator PtrMemory (yellow template parameter in Figure 1), which accepts a pointer to external memory along with a description of the layout used by the external memory.

We illustrate this in an example showing how to construct an openfpm_data cell-list neighbor search data structure[16] on Kokkos array views using a SoA layout with zero copying. The corresponding C++ code is shown in List 3. In line 1, we create an openfpm_data vector wrapper with SoA layout. The wrapper does not allocate any memory, but accepts external memory that it is going to wrap, as indicated by the allocator PtrMemory. The container w_pos contains the particle position for which we build the cell-list. In line 2, we create the memory object (PtrMemory) providing the address of the beginning of the Kokkos view (&pos(0,0)) followed by the size of the chunk of memory (N*sizeof(float)*3), where N is the total number of particles. Line 3 sets the memory for the wrapper, and line 4 resizes the vector to the number of particles. openfpm_data data structures internally ensure that the external memory does not overflow during resizing. In debug mode, they additionally perform range checks and notify of errors with a complete stack trace.

Lines 23–25 construct the openfpm_data cell-list data structure based on the wrapped Kokkos memories. For this, line 23 first declares a box-shaped computational domain as the three-dimensional unit cube containing all particles. Line 24 declares an openfpm_data cell-list object consisting of $10 \times 10 \times 10$ cells with extra padding of two cells at each border to handle boundary conditions ("ghost layer", "halo layer"). Thanks to the dynamic GPU context created in line 21, the CudaMemory type automatically switches between host and GPU memory depending on whether a GPU is available in the system and the code has been configured to make use of it. The option no_print_props suppresses all diagnostic output from the automatic detection. In line 25, we call the openfpm_data generic algorithm to build the cell-list and sort the particles and their properties into the newly created cell-list.

While the sorting of the particles in the cell list is optional, it improves memory-access patterns for particle-based computation. It does, however, require a second memory buffer to store the sorted vectors. This is also done using native Kokkos memory, illustrating write access to Kokkos memory with no need for copying data. The declaration of the sorted vector (w_pos_ord) follows the same logic as was already used for the input vector (lines 6–9). It is also possible to correspondingly reorder any vector of particle properties stored in Kokkos memory by constructing additional openfpm_data wrappers, as shown here for a particle mass (w_mass and w_mass_ord, lines 11–19).

Taken together, this example shows that openfpm_data can transparently use memory allocated and mapped by third-party libraries, such as Kokkos, for zero-copy read and write operations, and how this can effectively be used to extend other frameworks with openfpm_data-specific functionality, like cell-lists.

```
1  openfpm::vector<aggregate<float[3]>,PtrMemory,memory_traits_inte> w_pos;
2  {PtrMemory & ptr = *(new PtrMemory(&pos(0,0),N*sizeof(float)*3));
3  w_pos.setMemory(ptr);
4  w_pos.resize(N);}
5
6  openfpm::vector<aggregate<float[3]>,PtrMemory,memory_traits_inte> w_pos_ord;
7  {PtrMemory & ptr = *(new PtrMemory(&pos_ord(0,0),N*sizeof(float)*3));
8  w_pos_ord.setMemory(ptr);
9  w_pos_ord.resize(N);}
10
11 openfpm::vector<aggregate<float>,PtrMemory,memory_traits_inte> w_mass;
12 {PtrMemory & ptr = *(new PtrMemory(&mass(0),N*sizeof(float)));
13 w_mass.setMemory(ptr);
14 w_mass.resize(N);}
15
16 openfpm::vector<aggregate<float>,PtrMemory,memory_traits_inte> w_mass_ord;
17 {PtrMemory & ptr = *(new PtrMemory(&mass_ord(0),N*sizeof(float)));
```

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
18  w_mass_ord.setMemory(ptr);
19  w_mass_ord.resize(N);}
20
21  mgpu::ofp_context_t context(mgpu::gpu_context_opt::no_print_props);
22
23  SpaceBox<3,float> box({0.0f,0.0f,0.0f},{1.0f,1.0f,1.0f});
24  CellList_gpu<3,float,CudaMemory> cl2(box,{10,10,10},2);
25  cl2.construct(w_pos,w_pos_ord,w_mass,w_mass_ord,context,N);
```

Listing 3: Zero-copy `openfpm_data` cell-list construction on Kokkos memory.

## 5 | BENCHMARKS

We profile the memory and compute performance of `openfpm_data` in micro-benchmarks, and we showcase the resulting performance portability in a real-world application from computational fluid dynamics. All benchmarks are performed on the hardware and using the compilers listed in Table 3. Benchmarks for sparse data structures are available elsewhere.[8] We only benchmark the OpenMP, CUDA, and HIP backends of `openfpm_data`; SEQUENTIAL is always slower and only intended for debugging or porting purposes.

In certain types of computation, kernels require blocks of threads. This is typical of GPU programming, for example. In CUDA and `openfpm_data`, blocks of threads are created during kernel launch. OpenMP and Kokkos provide similar functionality in the form of workgroups and teams, respectively. In the first benchmark, we therefore compare the performance of `openfpm_data __syncthreads` on CPUs with Kokkos teams. This benchmark does not actually compute anything, but only measures the latency of thread synchronization. It does so by executing 24 context switches for every thread in 262,144 workgroups of 64 threads each and computing the average time per switch.

Table 4 shows the measured latency on each tested CPU, defined as the mean wall-clock time to complete a context switch (averaged over about 400 million context switches), in comparison with a single CPU clock tick. The `openfpm_data` code uses the OpenMP backend with a workgroup size of 64 threads. The team size in Kokkos is limited by the number of CPU cores available and was chosen as large as possible on each tested CPU. On the GPUs, the kernel primitives simply wrap the equivalent CUDA or HIP functions, respectively, so we do not benchmark them here.

Compared to `openfpm_data __syncthreads`, Kokkos teams are not only slower, but also less flexible. Their main limitation is that performance sharply deteriorates when using team sizes that exceed the number of CPU cores available to OpenMP. For example, using team sizes ranging from 16 to 64 on the 64-core AMD EPYC 7702 CPU, synchronization latency is between 88 and 287 ns. Using team sizes larger than 64 significantly increases latency to 2165 ns for a team of size 128. This is because the only way to run Kokkos teams larger than the number of cores is to oversubscribe the cores in a way that multiple threads run on a single core. In `openfpm_data`, the performance of `__syncthreads` is independent of the number of physical cores and of the block size.

### 5.1 | Memory performance

We next analyze the memory performance of `openfpm_data`. We do so using a micro-benchmark that moves data between structures containing scalars, vectors, and rank-two tensors. Because this benchmark is memory-bound, it assesses the memory performance portability of the

**TABLE 3** Hardware/compiler combinations considered for the benchmarks.

| Hardware | Type | Vendor | Compiler |
|---|---|---|---|
| A100 | GPU | Nvidia | NVCC 11.01 |
| RTX 3090 | GPU | Nvidia | NVCC 11.01 |
| M1 | CPU | Apple | clang 12.05 |
| POWER 9 | CPU | IBM | GCC 10.2 |
| Ryzen 3990X | CPU | AMD | GCC 9.3 |
| EPYC 7702 | CPU | AMD | GCC 10.2 |
| Xeon 8276 | CPU | Intel | GCC 10.2 |
| RXVega 64 | GPU | AMD | clang 13 |

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

**TABLE 4** Measured latency (in nanoseconds, ns) for a single thread synchronization operation.

| CPU | `__syncthreads` | Kokkos teams | Clock tick |
|---|---|---|---|
| M1 | 41.1 | 227 | 0.313 |
| POWER 9 | 55.3 | 109 | 0.250 |
| Ryzen 3990X | 13.4 | 65 | 0.233 … 0.345 |
| EPYC 7702 | 16.6 | 407 | 0.291 … 0.500 |
| Xeon 8276 | 9.0 | 168 | 0.250 … 0.455 |

*Note*: We compare the `openfpm_data __syncthreads` primitive on different CPUs using the OpenMP backend with a workgroup size of 64 threads with Kokkos teams with team sizes equal to the number of cores available on the respective CPU. The duration of a single CPU clock tick is given in the last column for scale; dynamically clocked CPUs provide a range. The standard deviations over 20 repetitions of the Kokkos timings are: M1 (2.3%), POWER 9 (0.3%), Ryzen 3990X (11.9%), EPYC 7702 (3.8%), Xeon 8276 (3.0%).

`openfpm_data` aggregates/tuple data abstractions. We evaluate the results both absolutely and relatively. For the relative evaluation, we compare against a hand-tuned implementation in Kokkos[2] and a C++ plain-array implementation. For the absolute evaluation, we compare the memory bandwidth achieved by `openfpm_data` with the synthetic benchmarks babel-STREAM (for Power 9, ARM, and dual-socket x86_64), pmbw (for single-socket x86_64 — an optimized parallel memory bandwidth benchmark written in assembly), and vendor-specific memory copy functions for the GPUs, as well as with the theoretical peak memory bandwidth reported in the data sheets.

We perform the benchmark on 67.1 million elements, each containing a scalar, two 2-vectors, and a tensor of rank two and size $2 \times 2$. As evident from Figure 2, this tests all abstraction levels of `openfpm_data`. We repeat each benchmark both for reading and for writing. The **write benchmark** reads one element from component 0 of the first vector and copies it into component 1 of the first vector, the scalar, all four components of the $2 \times 2$ tensor, and all components of the second 2-vector. This requires a total of nine memory accesses (counted from the generated assembly code): 8 write and 1 read. The **read benchmark** reads the values from the first 2-vector, the scalar, the tensor, and component 0 of the second vector, sums them, and writes the sum into component 1 of the second vector. This results in a total of 8 reads and 1 write. In this benchmark, we use lambda-based `openfpm_data` implementations compiled for the OpenMP backend on CPUs and for CUDA/HIP backends on GPUs. Memory bandwidth is calculated as the number of access operations divided by the runtime to complete all of them. The results are shown in Table 5.

On the x86_64 CPUs, the measured memory bandwidth when reading is significantly larger than when writing. This suggests the use of a cache policy of type `write_allocate` rather than `write_around`. In `write_allocate`, a write to a memory location out of cache generates a cache line that is filled from memory. Eventually the line is written back, causing double transfer of data compared to a read. The GPUs appear to implement a `write_through` cache policies. On all platforms, the memory performance of `openfpm_data` is comparable to that of plain C++ arrays (Table 5). With the exception of the M1 and the POWER 9, the numbers also match the synthetic benchmarks, confirming that the double-map tuple abstraction of `openfpm_data` incur low performance overhead. Further analysis shows that the difference between `openfpm_data`/Kokkos/C++ and the synthetic benchmark on the M1 is mainly due to the performance of the thread-local storage (TLS). In particular, reading the Software

**TABLE 5** Memory performance (read/write) on different hardware in Gigabytes/second (GB/s) for the same memory transfer micro-benchmark (see main text) implemented in `openfpm_data`, Kokkos, and plain C++ arrays, compared with the synthetic memory benchmarks described in the text and the vendor-provided memory bandwidth from the data sheet, where available.

| Hardware | `openfpm_data` | Kokkos | Plain C++ | Synthetic | Data sheet |
|---|---|---|---|---|---|
| A100 | (1390/1212) | (1375/1131) | (1394/1226) | 1297 | 1555 |
| RTX 3090 | (868/818) | (869/819) | (868/818) | 835 | 936 |
| M1 | (47.5/27.5) | (43.1/28.6) | (47.8/26.1) | 61.8 | N/A |
| POWER 9 | (120.2/109.8) | (143.0/112.8) | (121.6/111.8) | 250.0 | 340 |
| Ryzen 3990X | (70.8/37.7) | (54.0/32.8) | (70.6/37.7) | (77.1/37.7) | 96 |
| EPYC 7702 | (242.5/135.3) | (243.6/134.7) | (243.9/133.2) | 214.0 | 384 |
| Xeon 8276 | (137.1/87.7) | (142.9/89.6) | (144.3/89.6) | 150.0 | 216.8 |
| RXVega 64 | (359/358) | (323/293) | (359/360) | 378 | 484 |

*Note*: All synthetic benchmarks except pmbw (for Ryzen 3990X) and data sheets only report composite read/write bandwidth. For each measurement, the standard deviation over 100 repetitions (after 10 warm-up repetitions) is <3%.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Thread ID Register `TPIDRRO_EL0` on the M1 seems to be slow, slowing down codes using private variables as are used here to store `blockIdx` and `threadIdx` for each kernel.

## 5.2 Compute performance

In order to benchmark the compute performance of `openfpm_data`, we use the miniBUDE performance benchmark,[14] which has previously been used to compare compute performance of programming models including OpenCL, Kokkos, CUDA, SYCL, OpenMP, and OpenACC. While this benchmark does not over-stress the data structures, it quantifies the performance portability of the algorithms provided by `openfpm_data`. We do so by running the miniBUDE CUDA benchmark kernel through `openfpm_data`'s kernel interface. The `openfpm_data` compute kernel remains the same across all benchmarks, but is compiled using different backends. On Nvidia GPUs we use the CUDA backend of `openfpm_data`, on CPUs we use the OpenMP backend, and on AMD GPUs we use the HIP backend.

To render the results reproducible and comparable across compilers, we manually enable DAZ (denormals are zero) and FTZ (flush to zero) on all hardware. This does not affect significantly the values computed, but prevents compilers from using different SIMD mask flags with different compilation options when subnormal numbers are computed.

Table 6 reports the relative performance of the same `openfpm_data` code on different hardware compared with the respective best performer from the miniBUDE test suite, as indicated in the last column. Despite the fact that the `openfpm_data` kernel was not manually changed or tuned for the different hardware targets, it mostly performs on par with the specialized CUDA or OpenMP implementations of miniBude, demonstrating performance portability of the algorithm kernels. The only exception is the RXVega 64, where OpenCL is faster than `openfpm_data` with HIP backend. Code inspection shows that this is because the two compilers produce different code: HIP produces code with fewer registers and higher occupancy, while OpenCL does the opposite. While it is counter-intuitive that this explains the performance difference, it is what the measurements show, and it possibly hints at latency or GPU stalling as the problem for `openfpm_data` on the RXVega 64.

We confirm that the `openfpm_data` data structures do not interfere with the vectorization capabilities of the compiler on the CPU backends. For this, we consider the $N$-body code in List 4, where we compute the pairwise forces between all $N^2$ combinations of $N$ particles. This code uses `openfpm_data` slice-like views supporting complex memory-access patterns. The listing shows the main loop calculating the force and resulting velocity change on each particle `j` due to interactions with all other particles.

Lines 1 and 5 use an `openfpm_data` directive to specify that the loops can be vectorized. This directive is mapped to the appropriate pragmas in a portable way across compilers. Lines 2–6 loop through all pairwise particle interactions. In line 8, we get a view on the current particle `j`. A view is used in this case in order to delay the memory address computation until all indices are known. This view `pj` does not access any data. In line 10, we use this generic view to access the position property `POS` of the particle using the `<>` access operator (see Section 2). The result `posj` is another view on which we can use the operator `[]` to access individual spatial coordinate components like in lines 13–15. Lines 17–24 contain the force calculation, while lines 27–29 perform the time integration to compute the resulting change in particle velocity using explicit Euler time stepping.

We show that despite the view abstractions, the clang 13 compiler is able to understand the contiguity of the memory access and vectorize the code for an x86_64 CPU with 256 bit AVX extensions. Generation of AXV-512 instructions requires clang 15 with proper compiler options[†]. An excerpt from the generated assembly code from List 4 using 256 bit AVX is shown in List 5. The full file, as well as the file for AVX-512 are provided

**TABLE 6** Performance of the same miniBUDE-like `openfpm_data` kernel on different hardware compared with the respective best performer of the miniBude benchmark[14] as given in the last column.

| Hardware | `openfpm_data`/miniBude | Best miniBude |
|---|---|---|
| A100 | 1.00 ± 0.07 | CUDA |
| RTX 3090 | 1.00 ± 0.04 | CUDA |
| M1 | **1.05 ± 0.01** | OpenMP |
| POWER 9 | 0.80 ± 0.09 | OpenMP |
| Ryzen 3990X | **1.08 ± 0.04** | OpenMP |
| EPYC 7702 | 1.01 ± 0.03 | OpenMP |
| Xeon 8276 | 0.97 ± 0.03 | OpenMP |
| RXVega 64 | 0.54 ± 0.01 | OpenCL |

*Note*: Values are given as relative performance (GFlops `openfpm_data`)/(GFlops best miniBude) with mean ± standard deviation over 30 independent repetitions. Values >1 (in bold) mean that `openfpm_data` was statistically significantly faster than the fastest miniBude implementation.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

in the github repository[‡]. In particular, lines 2, 5, 7 load the $x$, $y$, and $z$ components of the position of particle $j$ (indexed by `%rbp`) into AVX registers `ymm0`, `ymm1`, and `ymm2`, respectively. Lines 4, 6, 8 subtract `posj[0]`, `posj[1]`, `posj[2]` for 8 particles at once in one AVX instruction. This is possible because we chose a data structure layout where `posj[0]` of particle $n$ is contiguous with `posj[0]` of particle $n-1$, and `posj[0]` is not contiguous with `posj[1]` like it would be in standard C ordering. Lines 9–25 contain the force computation as vectorized AVX instructions, while lines 26, 28, 30 have the vectorized store operations for the resulting velocity.

```
1  IVDEP
2  for (int i = 0; i < PROBLEM_SIZE; i++) {
3      const auto& pi = particles.get(i);
4      const auto& posi = pi.get<POS>();
5      IVDEP
6      for (int j = 0 ; j < PROBLEM_SIZE ; j++) {
7          // Construct a view on particle j
8          const auto& pj = particles.get(j);
9          // Construct a view on the position of particle j
10         const auto& posj = pj.get<POS>();
11
12         // Calculate force on the particle
13         const float distanceX = posi[0] − posj[0];
14         const float distanceY = posi[1] − posj[1];
15         const float distanceZ = posi[2] − posj[2];
16
17         const float distanceSqrX = distanceX ∗ distanceX;
18         const float distanceSqrY = distanceY ∗ distanceY;
19         const float distanceSqrZ = distanceZ ∗ distanceZ;
20
21         const float distSqr = EPS2 + distanceSqrX + distanceSqrY + distanceSqrZ;
22         const float distSixth = distSqr ∗ distSqr ∗ distSqr;
23         const float invDistCube = 1.0f / sqrtf(distSixth);
24         const float sts = pj.get<2>() ∗ invDistCube ∗ TIMESTEP;
25
26         // Store the change in particle velocity
27         particles.get<VEL>(j)[0] += distanceSqrX ∗ invDistCube ∗ sts;
28         particles.get<VEL>(j)[1] += distanceSqrY ∗ invDistCube ∗ sts;
29         particles.get<VEL>(j)[2] += distanceSqrZ ∗ invDistCube ∗ sts;
30     }
31 }
```

Listing 4: $N$-body force calculation in `openfpm_data`.

```
1  // load vectorized particle j
2  vbroadcastss 0x0(%rbp),%ymm0
3
4  vsubps (%r11),%ymm0,%ymm0
5  vbroadcastss 0x0(%rbp,%rsi,4),%ymm1
6  vsubps (%r11,%rsi,4),%ymm1,%ymm1
```

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
7  vbroadcastss 0x0(%rbp,%r9,4),%ymm2
8  vsubps (%r11,%rsi,8),%ymm2,%ymm2
9  vmulps %ymm0,%ymm0,%ymm0
10 vmulps %ymm1,%ymm1,%ymm1
11 vmulps %ymm2,%ymm2,%ymm2
12 vaddps %ymm6,%ymm0,%ymm3
13 vaddps %ymm2,%ymm1,%ymm4
14 vaddps %ymm4,%ymm3,%ymm3
15 vmulps %ymm3,%ymm3,%ymm4
16 vmulps %ymm3,%ymm4,%ymm3
17 vrsqrtps %ymm3,%ymm4
18 vmulps %ymm4,%ymm3,%ymm3
19 vfmadd213ps %ymm7,%ymm4,%ymm3
20 vmulps %ymm4,%ymm8,%ymm4
21 vmulps %ymm3,%ymm4,%ymm3
22 vmulps (%rax,%rcx,4),%ymm9,%ymm4
23 vmulps %ymm3,%ymm3,%ymm3
24 vmulps %ymm4,%ymm3,%ymm3
25 vfmadd213ps (%rbx),%ymm3,%ymm0
26 vmovups %ymm0,(%rbx)                  // vectorized store for veli[x]
27 vfmadd213ps (%rbx,%rdx,4),%ymm3,%ymm1
28 vmovups %ymm1,(%rbx,%rdx,4)           // vectorized store for veli[y]
29 vfmadd213ps (%rbx,%rdx,8),%ymm3,%ymm2
30 vmovups %ymm2,(%rbx,%rdx,8)           // vectorized store for veli[z]
```

Listing 5: Assembly code generated by C++; compiler from List[4]

## 5.3 | Application example: Smoothed particle hydrodynamics

We demonstrate the use of openfpm_data in a typical real-world application from scientific computing: a computational fluid dynamics simulation using the numerical method of Smoothed Particle Hydrodynamics (SPH).[17] As a baseline, we use the CPU-only implementation of SPH from the original OpenFPM paper,[9] which is freely available in the OpenFPM repository, albeit without the CPU-specific manual optimizations (like Verlet lists and symmetric interactions). This MPI implementation was shown in the original paper to be almost a factor of two faster than the state-of-the-art specialized SPH code "DualSPHysics",[18] therefore providing a good baseline for the present comparison. We derive from this code a version implemented using the CUDA-like interface of openfpm_data along with the built-in algorithmic primitives cell-list, sort, and prefix sum.

We use both codes—the original MPI-only CPU code[9] and the code using openfpm_data kernels—to simulate the same "dam break" SPH test case,[9] solving for the dynamics of a fluid sloshing around a square pillar in a rectangular tank (see Figure 3).

Table 7 shows the measured relative performances of these two codes on different CPUs. Performance is reported as the runtime ratio (original code)/ (openfpm_data code) in percent for the OpenMP backend of openfpm_data. Therefore, numbers >100% (in bold) indicate speedup. The most expensive part of the simulation, the force calculation step, is also profiled separately.

The results show that the openfpm_data abstraction layer adds no detectable performance penalty in this complex real-world application. It actually being a few percent faster than the original MPI code is likely because the OpenMP backend has a lower communication overhead than MPI.

Unlike the original MPI version, however, the openfpm_data code can also run on GPUs. On an Nvidia A100, for example, it runs 36 times faster than on all cores of an EPYC 7702 CPU, and on a RXVega 64, the speedup is 2.7. This difference in speedups is expected, as profiling shows the bottleneck for this application to be memory access and L2 cache. The Vega has slower memory than the A100 (484 GB/s vs. 1.5 TB/s) and 10x less L2 cache (4 MB vs. 40 MB). In addition, the Vega uses the old GCN architecture, known to be less efficient than AMD's new CDNA architecture. The performance difference between the A100 and the EPYC 7702 CPU is justified by the difference in memory bandwidths and the lack of CPU vectorization due to the scattered memory access pattern of the SPH particles, although there could be additional reasons, too.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

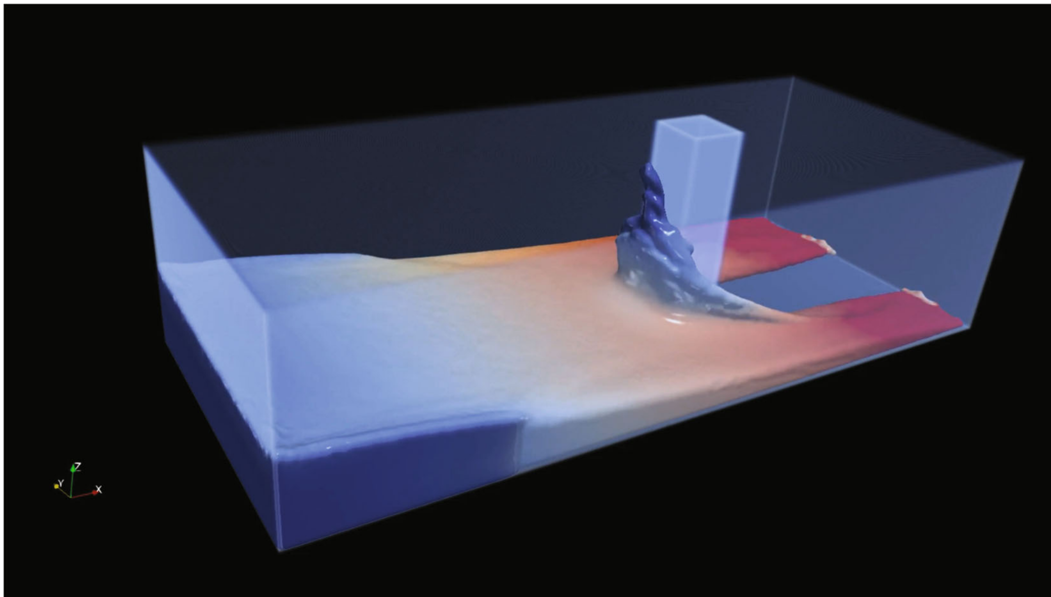

**FIGURE 3** Visualization (using ParaView) of a smoothed particle hydrodynamics (SPH) simulation of the "dam break" test case at simulated time $t = 0.6$.

**TABLE 7** Performance of the `openfpm_data` SPH "dam break" simulation on different CPUs using all available cores, relative to the performance of the original MPI code[9] on the same CPUs (=100%).

| Hardware | Overall | Force calculation |
|---|---|---|
| M1 | **109% ± 2.5%** | **113% ± 2.5%** |
| Ryzen 3990X | **105% ± 2.4%** | 98% ± 2.4% |
| EPYC 7702 | **115% ± 2.5%** | **121% ± 2.6%** |
| Xeon 8276 | **122% ± 2.5%** | 97% ± 2.4% |

*Note*: Numbers >100% (in bold) indicate statistically significant speedups.

## 6 | CONCLUSIONS

We have presented and benchmarked a C++ library for memory and compute abstraction across different CPU and GPU architectures. The presented library, called `openfpm_data`, combines hardware-independent abstract data structures with generic algorithmic building blocks. This places `openfpm_data` between libraries that focus on algorithm portability, like Kokkos[2,3] or Alpaka,[4] and libraries that focus on memory abstraction, like LLAMA.[7] Compared to the state of the art, `openfpm_data` provides more flexible memory layouts with tuples, memory double-mapping and absorbing, and advanced data structures like cell list, sparse grids, and graphs.

The present combination of abstract algorithms and data structures[1] allows accounting for their interdependence (e.g., reordering a data structure may require changing an algorithm for better performance, as shown here in the cell-list example of Section 4). We have shown the benefits this brings for performance portability in both micro-benchmarks and a typical real-world numerical simulation application, comparing to the respective state of the art. The presented benchmarks have also shown that memory layout switching using double-mapped C++ tuples and views do not interfere with performance and do not distract compiler optimizations. Finally, we have demonstrated how the memory absorbing capability of `openfpm_data` can be used to transparently wrap data structures that are allocated and managed by other software and enable their native use in `openfpm_data`. This allows extending and complementing existing frameworks by `openfpm_data`-specific functionality, and it enables `openfpm_data` to benefit from other memory-layout capabilities of external libraries.

The algorithmic primitives provided by `openfpm_data` include arbitrary-dimensional convolution, copying, merging, sorting, prefix sum, reductions, neighbor search, and filtering. They are available in optimized implementations for CUDA, HIP, SEQUENTIAL, and OpenMP backends and can be used and extended in either a CUDA-like kernel programming interface or a lambda-function interface. This allows the same code to run on different hardware platforms without losing performance, as demonstrated in the SPH fluid-flow simulation example.

The abstract data structures provided by `openfpm_data` are composable and can be used as building blocks for more complex data structures, such as distributed sparse block grids,[8] and for domain-specific data structures.[9] The memory layout capabilities are inherited, as well as the memory double-mapping and absorbing capabilities, allowing the same data structure to simultaneously be mapped to host and device. Moreover, third-party libraries can be interfaced via external memory. This is used in the scalable *distributed* scientific computing project OpenFPM.[9] The distributed data structures of OpenFPM are implemented on top of the `openfpm_data` abstraction layer presented here, enabling multi-node and multi-GPU applications with transparent network communication.

While the current version of `openfpm_data` at the time of writing is fully usable for practical applications, it has several limitations. One limitation is that no convenient user interface is available for layout overriding. Each parenthesis operator can be overridden with a user-defined memory layout without requiring changes to code using the data structure in question. At the moment, however, `openfpm_data` does not provide a simple way to generate custom layouts, and they need to be written by hand. A second limitation is that while `openfpm_data` improves performance portability, some manual fine-tuning may still be required, for example, to optimize thread block sizes. Also, hardware-specific optimization of user-implemented kernels is still required, albeit the algorithmic primitives provided by `openfpm_data` help. Future work could also include the addition of more backends, for example for OpenCL or OpenACC, in order to support more hardware and/or further improve performance portability.

Taken together, the hardware-portable data structures, generic algorithms, and the CUDA-like and lambda kernel interfaces provided by `openfpm_data` enable C++ codes to transparently run across multiple CPU and GPU architectures upon recompiling with a different backend enabled. We believe this has the potential to significantly reduce developer overhead in porting codes and enable more applications to harness the power of GPU computing and accelerator hardware.

## ACKNOWLEDGMENTS

We thank Christian Trott from the Kokkos project for his help and advise in tuning the Kokkos benchmarks for optimal performance. The authors are grateful to the Center for Information Services and High Performance Computing (ZIH) of TU Dresden and to the Scientific Computing Facility of MPI-CBG for providing their facilities for the benchmarks. This work was supported by the Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grants 01/S18026A-F (competence center for Big Data and AI “ScaDS.AI Dresden/Leipzig”) and 031L0160 (project “SPlaT-DM – computer simulation platform for topology-driven morphogenesis”). Open Access funding enabled and organized by Projekt DEAL.

## DATA AVAILABILITY STATEMENT

The source code of the presented library is available under the GPLv3 license as part of the OpenFPM project for scalable scientific computing (http://openfpm.mpi-cbg.de/) at: https://github.com/mosaic-group/openfpm_data. The repository also contains all benchmark codes used to generate the results in this paper:

- `__syncthreads` and Kokkos teams: https://github.com/mosaic-group/openfpm_pdata/tree/master/example/Performance/Syncthreads_kokkos_benchmark,
- memory bandwidth: https://github.com/mosaic-group/openfpm_pdata/tree/master/example/Performance/memBW/,
- miniBUDE https://github.com/mosaic-group/openfpm_pdata/tree/master/example/Performance/miniBUDE,
- *N*-body: https://github.com/mosaic-group/openfpm_pdata/tree/master/example/Performance/Nbody_benchmark, and
- SPH: https://github.com/mosaic-group/openfpm_pdata/tree/master/example/Vector/7_SPH_dlb_gpu_opt.

## ENDNOTES

[*] https://www.boost.org/

[†] `-march=skylake-avx512-mavx2-mtune=skylake-avx512-mprefer-vector-width=512`

[‡] https://github.com/mosaic-group/openfpm_pdata/blob/master/example/Performance/Nbody_benchmark as “nbody.s” (AVX) and “nbody_avx512.s” (AVX-512).

## REFERENCES

1. Sbalzarini IF. Abstractions and middleware for petascale computing and beyond. *Int J Distrib Syst Technol*. 2010;1(2):40-56.
2. Trott C, Lebrun-Grandié D, Arndt D, et al. Kokkos 3: programming model extensions for the exascale era. *IEEE Trans Parallel Distrib Syst*. 2022;33:805-817.
3. Edwards HC, Trott C, Sunderland D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J Parallel Distrib Comput*. 2014;74(12):3202-3216.
4. Zenker E, Worpitz B, Widera R, et al. Alpaka–an abstraction library for parallel kernel acceleration. *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016:631-640.

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5. Beckingsale D, Burmark J, Hornung R, et al. RAJA: portable performance for large-scale scientific applications. *IEEE/ACM International Workshop on Performance, Portability and Productivity In HPC (P3HPC)*. 2019:71-81.
6. Reyes R, Lomüller V. SYCL: single-source C++ accelerator programming. *Parallel Computing: on the Road to Exascale*. IOS Press; 2016:673-682.
7. Gruber B, Amadio G, Blomer J, Matthes A, Widera R, Bussmann M. LLAMA: The low-level abstraction for memory access. *Softw: Pract Exp*. 2023;53(1):115-141.
8. Incardona P, Bianucci T, Sbalzarini IF. Distributed sparse block grids on GPUs. *High Perform Comput. ISC High Perform*. 2021;2021(12728):272-290.
9. Incardona P, Leo A, Zaluzhnyi Y, Ramaswamy R, Sbalzarini IF. OpenFPM: a scalable open framework for particle and particle-mesh codes on parallel computers. *Comput Phys Commun*. 2019;241:155-177.
10. Alexandrescu A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison Wesley; 2001.
11. Bancila M. *Template Metaprogramming with C++: Learn Everything about C++ Templates and Unlock the Power of Template Metaprogramming*. Packt Publishing; 2022.
12. Mattson TG, Sanders B, Massingill B. Patterns for parallel programming. Pearson Education; 2004.
13. Green O, McColl R, Bader D. GPU merge path: a GPU merging algorithm. *Proceedings of the 26th ACM International Conference on Supercomputing*. 2012:331-340. 10.1145/2304576.2304621.
14. Poenaru A, Lin W, McIntosh-Smith S. A performance analysis of modern parallel programming models using a compute-bound application. *High Perform Comput ISC High Perform*. 2021;2021(12728):332-350.
15. Baxter S. moderngpu 2.0. 2016. https://github.com/moderngpu/moderngpu/wiki
16. Hockney R, Eastwood J. *Computer Simulation Using Particles*. Institute of Physics Publishing; 1988.
17. Monaghan J. Smoothed particle hydrodynamics. *Ann Rev Astron Astrophys*. 1992;30:543-574.
18. Crespo AJC, Domínguez JM, Rogers BD, et al. DualSPHysics: open-source parallel CFD solver based on smoothed particle hydrodynamics (SPH). *Computer Physics Communications*. Elsevier; 2015:204-216. doi:10.1016/j.cpc.2014.10.004

**How to cite this article:** Incardona P, Gupta A, Yaskovets S, Sbalzarini IF. A portable C++ library for memory and compute abstraction on multi-core CPUs and GPUs. *Concurrency Computat Pract Exper*. 2023;e7870. doi: 10.1002/cpe.7870

15320634, 0, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/cpe.7870 by Schweizerische Akademie Der, Wiley Online Library on [25/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License