Parallel Computing 39 (2013) 94–111

Contents lists available at SciVerse ScienceDirect

Parallel Computing

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

A portable OpenCL implementation of generic particle–mesh and mesh–particle interpolation in 2D and 3D

Ferit Büyükkeçeci, Omar Awile, Ivo F. Sbalzarini *

MOSAIC Group, Institute of Theoretical Computer Science, ETH Zurich, CH-8092 Zurich, Switzerland
Swiss Institute of Bioinformatics, ETH Zurich, CH-8092 Zurich, Switzerland

ARTICLE INFO

Article history:
Received 2 February 2012
Received in revised form 26 August 2012
Accepted 4 December 2012
Available online 13 December 2012

Keywords:
OpenCL
GPGPU
Particle–mesh method
Interpolation
PIC method
PPM library

ABSTRACT

Hybrid particle–mesh methods provide a versatile framework for simulating discrete and continuous systems. A key component is the forward and backward interpolation of particle data to mesh nodes. These interpolations typically account for a significant portion of the computational cost of a simulation. Due to its regular compute structure, interpolation admits SIMD parallelism, and several GPU-accelerated implementations have been presented in the literature. We build on these works to develop a streaming-parallel algorithm for interpolation in hybrid particle–mesh methods that works in both 2D and 3D and is free of assumptions about the particle density, the number of particle properties to be interpolated, and the particle indexing scheme. We provide a portable OpenCL implementation of the algorithm and benchmark its accuracy and performance. We show that with such a generic algorithm speedups of up to 15× over an 8-core multi-thread CPU implementation are possible if the data are already available on the GPU. The maximum speedup reduces to about 7× if the data first have to be transferred to the GPU. The benchmarks also expose several limitations of GPU acceleration, in particular for low-order and 2D interpolation schemes. The present algorithm is integrated and available in the open-source Parallel Particle Mesh (PPM) library as a hybrid MPI-OpenCL implementation.

1. Introduction

Particle methods provide a versatile framework for the simulation of continuous and discrete systems. When simulating continuous systems, the particles are the collocation points on which the governing equations are discretized, leading to a set of coupled ODEs for the particle positions and properties. The dynamics of N particles is determined by particle–particle interactions. Interactions that only involve a local neighborhood can efficiently be computed in $O(N)$ time using fast neighbor lists, such as cell lists [1] or Verlet lists [2]. Recently, these have also been extended to the case where the interaction radius is a function of space [3]. Long-range interactions, where every particle has to interact with all other particles, however, incur a computational cost of $O(N^2)$, rendering their direct computation infeasible for practical simulations. At the expense of computing an approximate solution, this computational cost can be reduced to $O(N\log N)$ or even $O(N)$ using fast N-body solvers such as the Barnes–Hut algorithm [4] or Fast Multipole Methods [5], respectively. Hybrid particle–mesh methods reduce the computational cost of long-range interactions by computing them on a Cartesian mesh via the associated PDE [1]. On M mesh nodes, this can be done in $O(M\log M)$ or $O(M)$ time using fast Fourier transforms or multi-grid solvers, respectively. Hybrid particle–mesh methods are in practice often more efficient than fast N-body solvers. A large part of the

* Corresponding author. Present address: Max Planck Institute of Molecular Cell Biology and Genetics, Center of Systems Biology, Pfotenhauerstr. 108, D-01307 Dresden, Germany.
E-mail address: ivos@mpi-cbg.de (I.F. Sbalzarini).

0167-8191/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2012.12.001

computational cost of a hybrid particle–mesh method is accounted for by interpolating the particle properties to the mesh nodes ("particle–mesh interpolation") and interpolating the result of the mesh solver back to the particles ("mesh–particle interpolation"). In plasma physics Particle-In-Cell (PIC) codes, for example, interpolation accounts for 85–98% of the overall computer time, in vortex methods for incompressible fluid dynamics the fraction is 20–40%. Accelerating the computation of this part is hence expected to provide a significant speedup to the entire simulation.

Due to their regular computational structure and fine granularity, particle–mesh and mesh–particle interpolations admit SIMD (Single Instruction Multiple Data [6]) parallelism on streaming multiprocessors, such as Graphics Processing Units (GPU). Different approaches to particle–mesh interpolation on GPUs were discussed and compared by Stantchev et al. in the context of a 2D plasma physics PIC code using CUDA [7]. A very efficient implementation of 2D particle–mesh interpolation on a GPU was presented by Rossinelli and Koumoutsakos [8]. Using OpenGL, they reported real-time incompressible fluid mechanics simulations using vortex methods [9,10] at 25 frames per second on a 1024×1024 mesh, representing a 26-fold speedup over the highly optimized reference CPU code. The same authors later extended their work to simulate flows in bounded, complex geometries in 2D [11] and reported a 100-fold speedup of OpenGL over CUDA when using the OpenGL point-sprite primitives and blending techniques. A GPU implementation of 2D mesh–particle interpolation has also been provided by Rossinelli et al. [12] in both OpenCL and CUDA. For double-precision arithmetics, they reported 2 to 3-fold speedups of the GPU implementation over a 16-threaded CPU implementation and a 20 to 45-fold speedup over a single-thread CPU implementation. Madduri et al. presented another 2D implementation of particle–mesh interpolation in a PIC code [14]. Their implementation uses CUDA, particle binning, grid replication, and texture memory, but barely provides any speedup over an optimized multi-threaded CPU Implementation, demonstrating the limitations of GPU-accelerated interpolation. Recently, Conti et al. presented a complete OpenCL implementation of a 2D finite-time Lyapunov exponent computation with an up to 30-fold speedup over their single-thread CPU reference code; they also compared the performance of a GPU and an APU (Accelerated Processing Unit) [15].

While purpose-made GPU implementations can offer impressive speedups, they typically suffer from low programmer productivity and poor performance portability. Libraries and generic algorithms are hence a recent trend in the field [16,17], even though they normally entail a performance toll when compared with specialized solutions. Here we follow this trend and build on the above-mentioned prior works in GPU-accelerated particle–mesh interpolation in order to present a portable OpenCL [13,17] implementation of a generic algorithm for particle–mesh and mesh–particle interpolation in both 2D and 3D. Our algorithm is generic in the following ways, without implying superior performance in all cases: (1) It is free of assumptions about the (typical or maximal) number of particles per mesh cell. (2) It does not expect the input data to be stored or sorted in any particular way. (3) It works with arbitrary numbers of particle properties and vector-valued fields. (4) It works in 2D and 3D, single precision and double precision. (5) It works with arbitrary mesh sizes that do not have to be powers of two. Moreover, the OpenCL implementation is portable across hardware platforms (multi-core CPUs and GPUs from different vendors). Parallelism is achieved by first reordering the particle data according to access patterns of threads in workgroups. Moreover, particle and mesh values are stored so as to distribute particles in the same mesh cell among different buffer frames. Together with a mesh-padding technique, this avoids race conditions and provides coalesced global memory access in strides. The mesh is additionally decomposed into blocks to achieve data locality for high cache hit rates. Our approach avoids atomic operations, which have been reported to harm performance on the GPU [14]. We show that a common parallelization strategy can be used for both particle–mesh and mesh–particle interpolation, albeit not matching the performances of the respective specialized implementations by Rossinelli et al. [11,12]. Our approach, however, naturally extends to 3D and we present and benchmark a full 3D implementation of particle–mesh and mesh–particle interpolation on the GPU. The present implementation is integrated and available in the open-source Parallel Particle Mesh (PPM) library [18,19].

The PPM library is a state-of-the-art middleware for distributed-memory parallel particle–mesh simulations. It is based on a set of abstract data types and operators for hybrid particle–mesh methods [20], providing an abstraction layer that hides message passing between distributed-memory nodes from the application programmer. PPM is based on adaptive geometric domain decomposition, communication through ghost (halo) layers, and approximate graph coloring for communication scheduling [18]. This has enabled transparently parallelized particle–mesh simulations that frequently outperformed hand-parallelized reference codes [20]. The present GPU implementation of particle–mesh/mesh–particle interpolation runs independently on each sub-domain of a distributed-memory PPM simulation, provided the corresponding node has a GPU. This extends GPU acceleration of particle–mesh simulations to distributed-memory settings, potentially enabling very large simulations in multi-GPU environments.

We benchmark the present OpenCL implementation against the highly optimized reference CPU implementation already existing in the PPM library, as well as a shared-memory parallel OpenMP version of the reference implementation. The results show that significant speedups on the GPU can only be achieved for higher-order interpolation schemes (here, we use the third-order M'_4 scheme [21]), especially in 3D and when using single-precision arithmetic. For mesh–particle interpolation in this case, we observe an at most 7-fold speedup of the GPU over the 8-thread CPU reference implementation in PPM (case with 863k particles, including data transfer times). The speedup with respect to a single-thread CPU is about 22-fold. A convergence analysis shows that when using single precision, numerical truncation errors around 10^{-6} to 10^{-5} are to be tolerated.

Our results indicate that higher-order interpolation schemes benefit more from GPU acceleration. For linear interpolation, the GPU implementation is in all cases slower than the 8-thread CPU implementation when accounting for communication

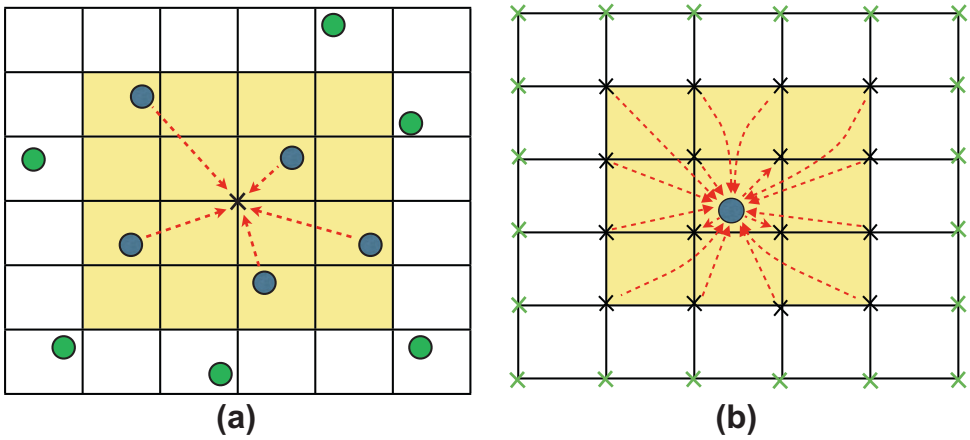


Fig. 1. Schematic of (a) particle–mesh and (b) mesh–particle interpolation in 2D using the M'_4 interpolation function, which leads to a support of $\pm 2\mathbf{h}$ (support region is shaded in yellow). Blue particles and mesh nodes are within the support region of the center node/particle and hence assign onto it. Green particles and nodes lie outside the support and are not considered. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

overhead. Interpolation schemes with a higher compute intensity (FLOP/byte ratio) generally show larger speedups. This confirms previous reports [14] and illustrates the limitations of GPU-accelerated interpolation.

2. Interpolation schemes

Hybrid particle–mesh methods rely on accurate and efficient interpolation of particle properties or *strengths* to the nodes of a *(block-wise) uniform Cartesian* mesh[1] and back [1]. We refer to the former as "particle–mesh interpolation" and the latter as "mesh–particle interpolation". In particle–mesh interpolation, one aims at exact conservation of the first few moments of the interpolated function. This ensures conservation of mass, momentum, angular momentum, etc., depending on the order of the interpolation scheme. In mesh–particle interpolation, conservation of moments is generally not possible due to non-uniform spacing of the target particles. In both cases, the interpolation error decreases as a power of the mesh spacing h. This power is called the *order of convergence* of the interpolation scheme.

We consider moment-conserving interpolation schemes in two and three dimensions that are Cartesian products of one-dimensional interpolants. The weights W of the one-dimensional interpolants are computed independently for each dimension i using the *mesh distance* $\mathbf{s}$ between the mesh node m and the particle p, $\mathbf{s}(m,p)$:

$$W_i(m,p) = f(s_i(m,p)) \tag{1}$$

$$= f\left(\frac{|x_i(m) - x_i(p)|}{h_i}\right), \tag{2}$$

where f is the one-dimensional interpolation function, $x_i(m)$ is the ith component of the position of mesh node m, and $x_i(p)$ the ith component of the position of particle p.

The final interpolation weight in d dimensions is computed as:

$$W(m,p) = \prod_{i=1}^{d} W_i(m,p). \tag{3}$$

Particle–mesh interpolation is then formulated as follows:

$$\boldsymbol{\omega}(m) = \sum_{p \in \mathcal{P}(m)} W(m,p)\boldsymbol{\omega}(p), \tag{4}$$

where $\mathcal{P}(m)$ is the set of particles that contribute to mesh node m, i.e., are within the support of the interpolation weights W from the mesh node m (see Fig. 1(a)), and $\boldsymbol{\omega}$ is the function (property) that is being interpolated.

Likewise, mesh–particle interpolation is formulated as:

$$\boldsymbol{\omega}(p) = \sum_{m \in \mathcal{M}(p)} W(m,p)\boldsymbol{\omega}(m), \tag{5}$$

where $\mathcal{M}(p)$ is the set of mesh nodes that contribute to particle p (see Fig. 1(b)).

[1] Non-uniform and unstructured meshes are not considered in hybrid particle–mesh methods, as sub-grid scales can be represented on the particles.

The choice of interpolation function f determines the order of convergence and the number of conserved moments. Here, we consider the M'_4 (introduced by Monaghan as the W_4 function [21]) and the linear interpolation schemes:

$$f_{M'_4}(s) = \begin{cases} \frac{3}{2}s^3 - \frac{5}{2}s^2 + 1, & 0 \leqslant s \leqslant 1, \\ -\frac{1}{2}s^3 + \frac{5}{2}s^2 - 4s + 2, & 1 < s \leqslant 2, \\ 0, & \text{else}. \end{cases} \tag{6}$$

$$f_{\text{linear}}(s) = \begin{cases} 1 - s, & 0 \leqslant s \leqslant 1. \\ 0, & \text{else}. \end{cases} \tag{7}$$

The order of convergence is 3 for M'_4 and 2 for linear interpolation. M'_4 conserves moments up to and including the second moment, whereas the linear interpolation scheme conserves moments up to and including the first moment. Linear interpolation is sometimes also termed "Cloud-in-Cell" (CIC) interpolation [1]. In addition to its lower numerical accuracy, it also has a lower compute intensity (FLOP/byte). SIMD-efficient evaluation of the interpolation polynomials is done using the Horner scheme.

3. GPU programming with OpenCL

We provide a portable OpenCL implementation of particle–mesh and mesh–particle interpolation in 2D and 3D. The implementation is integrated and available in the open-source PPM library [18] for parallel hybrid particle–mesh simulations. The Open Computing Language (OpenCL) is a programming framework for heterogeneous multi-core computing devices, including GPUs, APUs, and CPUs [13]. It provides a higher level of hardware abstraction than OpenGL, while still providing enough hardware control to allow for efficient implementations. OpenCL does not expose low-level graphics primitives, but instead provides data structures and operations suitable for general-purpose programs. OpenCL can be used on a wide variety of operating systems and hardware platforms, also extending beyond GPUs. OpenCL achieves portability through a hierarchy of abstraction layers: the platform model, the execution model, the programming model, and the memory model. The execution model describes the computing platform as a host and a collection of devices. In our case, the host is the CPU and the devices are the streaming multiprocessors of the GPU. The host executes the *host program*, which creates a context for the devices and manages the execution of *kernels* on the devices. When a kernel is launched by the host program, the OpenCL devices execute many instances of this kernel, called *work items*. Each work item performs a set of instructions specified by the kernel at one point in index space, thus processing different data items in parallel. Work items that execute on the same set of processing elements, *the compute unit*, form a *workgroup*. The work items of a workgroup share resources, such as the on-chip memory of the compute unit.

Like most parallel computing environments, OpenCL provides synchronization support in the form of *barriers*. There are two types of barriers: the command-queue barrier and the workgroup barrier. Here, we only use workgroup barriers in order to synchronize the work items within a workgroup. A workgroup barrier dictates that no work item must leave the barrier before all work items have entered it. Synchronization of work items belonging to different workgroups is not possible using workgroup barriers. This global synchronization of work items can be achieved by atomic operations or semaphores, or by orchestration of the workgroups from the host code. Such global synchronization, however, usually leads to significant performance losses.

GPUs are many-core architectures consisting of collections of identical Streaming Multi-Processors (SMP) that execute many instances of the same kernel in a Single Instruction Multiple Data (SIMD) control structure [6]. They are massively parallel, but with limited resources per core [22]. Mapping the OpenCL execution model onto a GPU, each work item is executed in a separate *thread*, and all threads hosting work items from the same workgroup execute on the same SMP. Therefore, work items must perform simple tasks on small amounts of data in order to provide sufficient granularity. Threads are executed in *warps* (called *wave fronts* for ATI GPUs), which are collections of threads that are executed simultaneously. Control flow divergence among threads within a warp results in serialization [23] and ideally should be avoided. Hence, work items should be homogeneous, i.e., perform the same tasks. Homogeneous, fine-grained work items enable the GPU to apply fast *context switching*. GPUs with hardware support for context switching change between threads in a single clock cycle in order to hide data-access latency of a thread by computation of another. One of the most important performance determinants on GPUs is the memory access pattern of the threads within a warp. A memory transaction to or from the global device memory is a multi-word burst transaction. Offsets and non-unit stride access of a warp result in additional transactions and degrade the memory bandwidth. Thus, work items and the data they access must be aligned and coalesced.

4. Method

In order for an algorithm to perform well on a GPU, it has to meet a number of requirements. First, it should rely on fine-grained kernels that can be parallelized over a large number of small work items. Second, it has to avoid race conditions that would require synchronization. Third, it has to avoid conditional statements that lead to control flow divergence. Fourth, the algorithm and the data structures must guarantee coalesced and aligned global memory access and use local memory for frequent read/write operations within the same memory region.

In the following, we outline the design of a generic streaming-parallel particle–mesh/mesh–particle interpolation algorithm in 2D and 3D as guided by these design principles. Particles are assigned to work items that loop over mesh nodes within the support of the interpolation function. In particle–mesh interpolation, work items scatter the particles' contributions onto the mesh nodes, whereas in mesh–particle interpolation they gather contributions from the mesh nodes. As work items iterate over mesh nodes, the interpolation weights are only recomputed along dimensions that do change. Different parallelization strategies are discussed in the following subsection.

4.1. Strategies for interpolation on the GPU

Stantchev et al. distinguish two strategies for particle–mesh interpolation on the GPU: the *particle-push* and *particle-pull* strategies [7]. In the particle-push strategy, particles are assigned to work items that scatter the particle contributions onto the mesh nodes. In the particle-pull strategy, mesh nodes are assigned to work items that gather particle contributions. Both strategies have advantages and disadvantages: Particle-push allows the work item assigned to a particle p, to compute dynamically $\mathcal{M}(p)$ from the particle's coordinates. The set $\mathcal{M}(p)$ has fixed length for all p. Moreover, the work item can reuse some interpolation weights that were already computed earlier when traversing mesh nodes along the same dimension. However, the particle-push strategy causes memory collisions as concurrent work items may attempt to write to the same mesh node simultaneously. This is avoided in the particle-pull strategy, where each mesh node can be updated by only one work item. The disadvantage of the particle-pull strategy is that it is costly to compute $\mathcal{P}(m)$ unless the particles are arranged in a regular spatial pattern (e.g., on a grid). Furthermore, the interpolation weights always have to be recomputed for each particle.

Similar trade-offs also exist in mesh–particle interpolation. This can be seen by analogously defining *mesh-push* and *mesh-pull* strategies: In the mesh-push strategy, each work item scatters the contributions of mesh nodes onto the particles within the support of the interpolation weights. A clear disadvantage of this strategy is the high cost of computing $\mathcal{P}(m)$ if particles are not organized in a regular spatial pattern. More importantly, a particle p might simultaneously be updated by concurrent work items from m and $\tilde{m}$ if p resides both in $\mathcal{P}(m)$ and $\mathcal{P}(\tilde{m})$. In the mesh-pull strategy, all work items are launched over particles and visit the mesh nodes in $\mathcal{M}(p)$, which is easily computed thanks to the regular geometry of the mesh. Each work item then accumulates the contributions of the mesh nodes in $\mathcal{M}(p)$ to $\boldsymbol{\omega}(p)$, which is free of memory collisions.

The advantages and disadvantages of these four parallelization strategies are qualitatively summarized in Table 1. From this, we conclude that the particle-push and mesh-pull strategies are preferable for particle–mesh and mesh–particle interpolation, respectively. A common feature of both strategies is that work items are defined over particles. This allows using the same algorithm for both interpolations by storing the particle positions and properties in the private memories of the work items, reserving the shared memory for the mesh data. Since mesh node positions do not need to be stored, this is an additional advantage given the limited size of the shared memory. In the following, we present data structures and algorithms for generic particle-push/mesh-pull interpolation on the GPU in 2D and 3D.

4.2. Data structures

The main goal in designing the data structures is to guarantee coalesced and aligned global memory access and data locality. At the same time, race conditions and atomic operations are to be avoided.

4.2.1. Particle data

The particle positions and strengths are stored in two linear buffers, named `particle_pos` and `particle_str`, respectively. Particles in the same mesh cell are distributed across different frames of these buffers, which we call *domain copies* (see Fig. 2). We then decompose each domain copy into blocks (red lines in Fig. 2) and store the particle data of each block consecutively with an ordering as indicated by the dashed arrows in Fig. 2. Taking this decomposition approach one step further, we also store the particle data along different dimensions in separate segments of the buffer, called *main segments*. This renders our strategy dimension-oblivious and guarantees memory access in unit stride.

The buffers are constructed as follows: We first count the number of particles in each mesh cell and store it in an `np_cell` buffer. Counting is done by atomically incrementing the elements of `np_cell` using the `atom_inc` instruction of OpenCL, which ensures sequential access to `np_cell`. This is the only time we use atomic operations in the whole workflow, and it enables us to directly assign indices to the particles according to the new ordering. Using these indices, we calculate

Table 1
Qualitative comparison of parallelization strategies for particle–mesh (columns 2 and 3) and mesh–particle (columns 4 and 5) interpolation.

	Particle-		Mesh-	
	pull	*push*	*pull*	*push*
No memory collisions	√	×	√	×
Fast computation of $\mathcal{P}(m)$ and $\mathcal{M}(p)$	×	√	√	×
Re-using interpolation weights	×	√	√	×

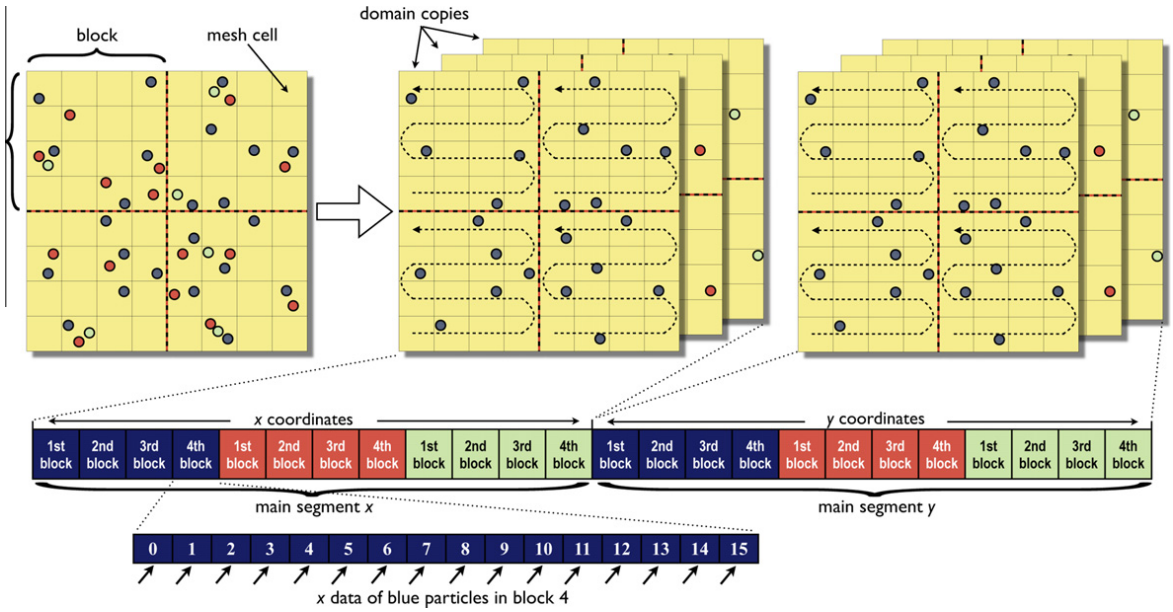


Fig. 2. Particle data are stored in buffer segments, blocks, and domain copies according to their dimension, mesh cell index, and number of fellow particles within the mesh cell. Particles within the same mesh cell are stored in different domain copies as highlighted by color. Each domain copy is subdivided into blocks (red dashed lines) that are stored consecutively in the buffer. Within each block, particles are numbered in the order shown by the dashed arrows. Data along different space dimensions are stored in separate main segments of the buffer. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

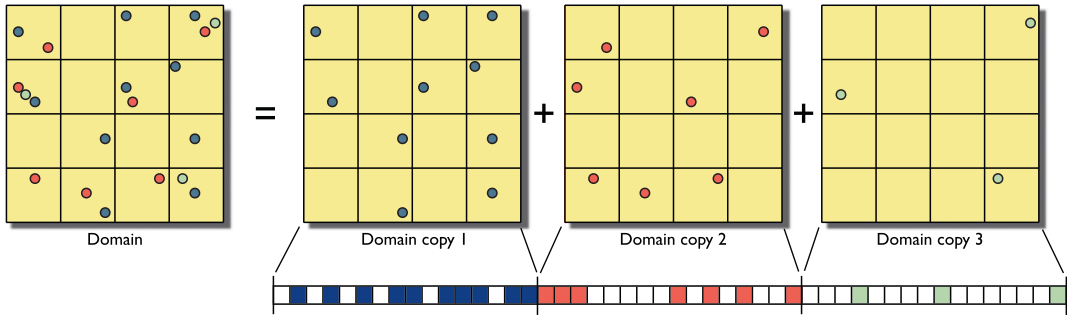


Fig. 3. Illustration of the use of domain copies and dummy particles in order to avoid conditional statements and guarantee coalesced memory access. See main text for details.

and store the index of the mesh cell within which each particle resides in the *cell list* `p_cell`. Then, the `np_cell` buffer is reduced in parallel to find the maximum number of particles in any mesh cell, $p_{\max}$. The required size of the particle data buffers is then given by $p_{\max} \times d \times n_{\text{cell}}$ where d is the number of dimensions and n_{cell} is the total number of mesh cells in the domain. The buffers then contain d main segments, each holding the particle data along one dimension. Each main segment further contains $p_{\max}$ domain copies (see Fig. 3), which are in turn decomposed into blocks. Each mesh cell corresponds to one memory location in the buffer; empty mesh cells are represented as "dummy particles" of zero strength, as shown in Fig. 3. This guarantees coalesced memory access in unit strides since the particles within each block are consecutively numbered. In this scheme, inhomogeneities in the particle distribution lead to memory and compute overhead. In our experience, however, this overhead is amortized by the performance gain from the resulting regular, coalesced memory access pattern. Moreover, adaptive meshes, along with dynamic remeshing, are used in practical simulations to limit particle inhomogeneity [24].

4.2.2. Mesh data

Mesh data are stored in a linear buffer `mesh_prop`, which is again decomposed into *main segments* by dimension and mesh cell *blocks*, as described above. In order to avoid the memory collisions that are possible in a particle-push strategy, and in order not to introduce sequential parts into the algorithm, we replicate mesh nodes that are closer to any block boundary than the support radius of the interpolation function. This is illustrated in 2D in Fig. 4. A block including its ghost

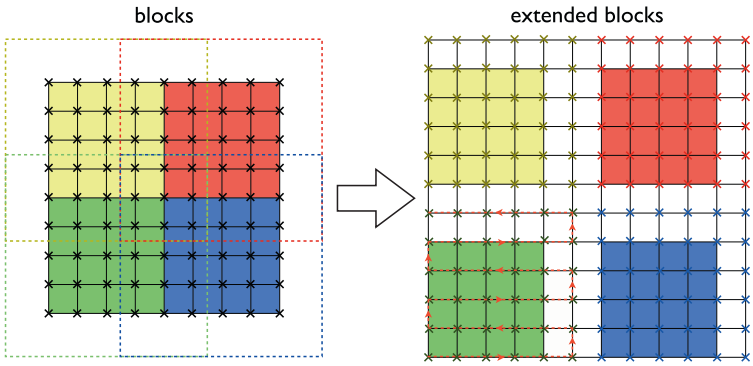


Fig. 4. Mesh nodes close to the boundary of a block are replicated in order to avoid concurrent writes and race conditions. **Left:** the dashed squares indicate the areas within which mesh nodes are influenced by particles in the block of the same color. **Right:** extended blocks are defined by replicating mesh nodes in overlapping influence regions. The red dashed arrow shows the storage order of the mesh nodes inside the green extended block. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

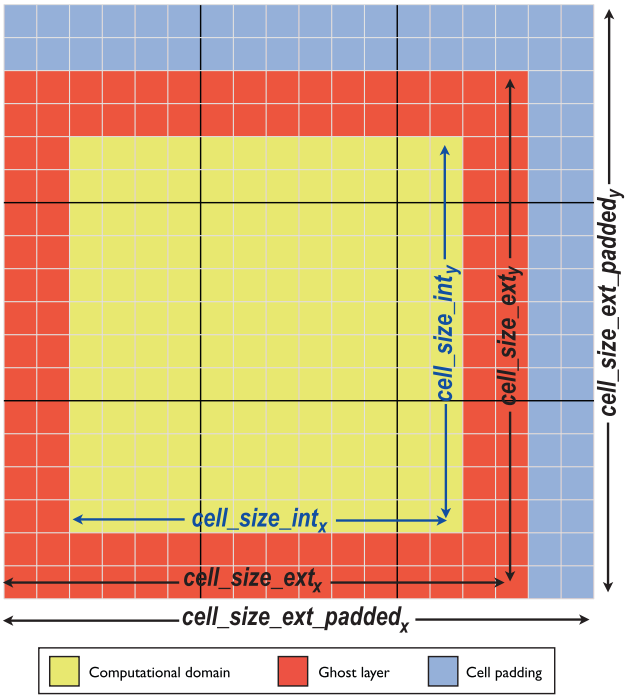


Fig. 5. Sizes of the computational domain (`cell_size_int`), including ghost layers (`cell_size_ext`), and after padding with extra cells (`cell_size_ext_padded`) as necessary for consecutive memory access. Extended blocks as defined in the main text are shown by black lines.

layer of replicated nodes is called an *extended block*. Mesh nodes are traversed in the same order as mesh cells, as shown by the red arrow in the green block in Fig. 4. After a completed particle–mesh interpolation, the contributions on replicated mesh nodes are aggregated by stitching the mesh back together in a post-processing step.

The total numbers of mesh cells in the computational domain without, and with, ghost layers, are termed `cell_size_int` and `cell_size_ext`, respectively. The total problem size is given by `cell_size_ext`. This size can be arbitrary, as it depends on the domain decomposition done by the PPM library. OpenCL, however, requires that the domain size be an integer multiple of the extended block size. We hence pad the domain as illustrated in Fig. 5.

4.3. Mapping of the data structures into OpenCL

The above-defined data structures are mapped onto the OpenCL execution model by assigning one mesh cell per work item. The data of all p_{max} particles (some of them being possibly dummy) inside that mesh cell are stored in the private memory of the work item; the mesh data are kept in shared memory. A workgroup is then defined as the collection of all work

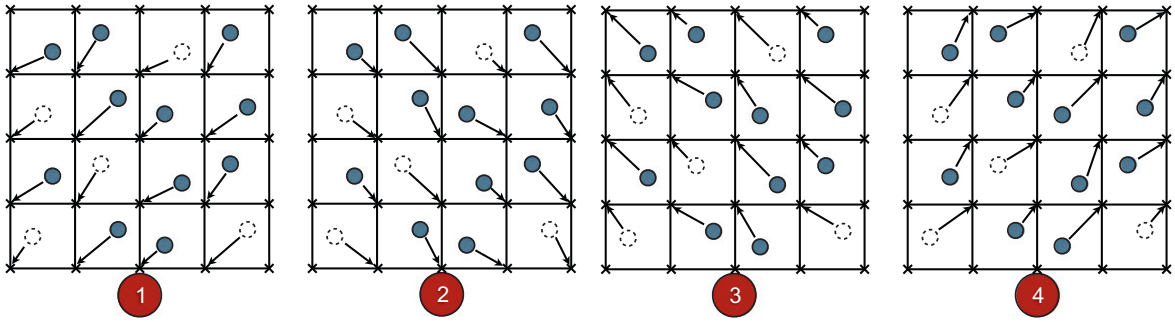


Fig. 6. Particle–mesh interpolation example in 2D with a linear interpolation function (support size 4). In each of the four iterations, the work items in the same workgroup (block) update the mesh nodes along the same direction in order to avoid concurrent writes. Dummy particles (dashed circles) avoid conditional statements. The four stages are repeated $p_{\max}$ times until all particles have been assigned.

items having mesh cells that belong to the same block. The data structure and storage order introduced in Fig. 2 then guarantee that work items always access the global memory in a coalesced fashion. The mesh padding shown in Fig. 4 avoids race conditions across workgroups and dispenses with the need for inter-workgroup synchronization and atomic operations.

The block size is chosen according to the GPU hardware. It depends on the number of shared memory banks and SMPs the GPU has, and it represents a memory trade-off. On the one hand, the blocks should be small enough for the data to fit into the shared memory of each workgroup. On the other hand, they should not be too small in order to limit the memory overhead stemming from the ghost layers around the extended blocks (see Fig. 4). To avoid shared memory bank conflicts, the blocks should moreover contain at least as many cells along the x-direction (i.e., the leading dimension of the loop) as the GPU has shared memory banks. The specific settings for the hardware used here are described in Section 6.

4.4. Interpolation algorithms for the GPU

In particle–mesh interpolation, each workgroup is assigned a block of mesh cells with one work item per cell. Every work item then loops over the $p_{\max}$ particles (i.e., the domain copies) in its cell and scatters their strength onto the mesh nodes within the support radius of the interpolation function, i.e., within the extended block. The redundant computation of dummy particles turns out to be more efficient than conditional statements on the GPU. Mesh–particle interpolation uses an analogous parallelization strategy, where dummy particles are purged from the `particle_str` buffer before reading the data back from the device.

4.4.1. Particle–mesh interpolation

In particle–mesh interpolation, each work item loops over the particles in one mesh cell and scatters their strengths onto the mesh nodes around in a nested loop over dimensions. Since domain copies and main segments are stored in different frames of the buffer, memory conflicts are avoided. The number of inner-loop iterations required is given by the size of the support of the interpolation weights W, $\mathrm{supp}(W)$, i.e., the number of mesh nodes where $W \neq 0$. With each iteration, all work items within a workgroup assign particle contributions onto mesh nodes along the same direction. A workgroup barrier is then used to synchronize all work items before assigning into the next direction. This synchronization ensures that no concurrent writes onto the same mesh node occur. Fig. 6 illustrates this "synchronized swimming" of work items for the example of a linear interpolation function (with local support of 4 mesh nodes). This is repeated $p_{\max}$ times until all particles have been assigned. The general kernel is given in Algorithm 1, the complete workflow for particle–mesh interpolation in Algorithm 2.

Algorithm 1. Particle–mesh interpolation kernel

1: $w \leftarrow$ work item ID
2: $\chi \leftarrow$ indices of mesh cells assigned to w
3: **for** i from 1 to $p_{\max}$
  (a) $p \leftarrow$ index of the ith particle in χ
  (b) $\mathbf{x} \leftarrow$ coordinates of the ith particle in χ
  (c) **for** $\kappa \in \mathrm{supp}(W)$
    i. $\mu \leftarrow \chi + \kappa$ (shift of mesh node in target)
    ii. $M \leftarrow$ index of the mesh node pointed to by μ
    iii. **for** each particle property ω_i
      A. $\omega_i(M) \leftarrow \omega_i(M) + W(\mu, \mathbf{x})\omega_i(p)$
    iv. workgroup barrier

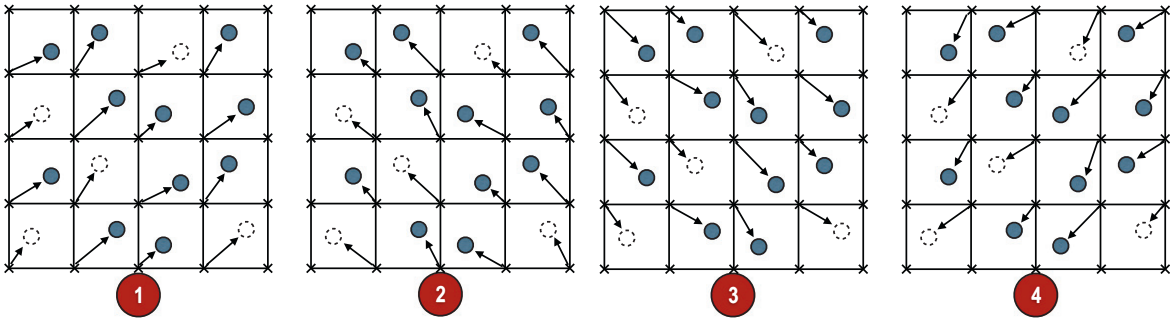


Fig. 7. Mesh–particle interpolation example in 2D with a linear interpolation function (support size 4). In each of the four iterations, the work items in the same workgroup (block) accumulate on their particles the contributions from the mesh nodes along the same direction. Dummy particles (dashed circles) avoid conditional statements. The four stages are repeated $p_{\max}$ times until all particles have been considered.

4.4.2. Mesh–particle interpolation

Mesh–particle interpolation follows an analogous parallelization strategy with workgroup being assigned mesh cell blocks, and work items particles within single mesh cells. The work items then gather in parallel the contributions from the mesh nodes within the support of the interpolation weights W. Unlike in particle–mesh interpolation, however, we do not need to synchronize the work items since particle strengths are stored in the private memory of the respective work item, and mesh nodes have been replicated as outlined in Section 4.2.2. Algorithm 3 shows the mesh–particle interpolation kernel. With each iteration, the particles receive contributions from mesh nodes along the same direction. This is illustrated in Fig. 7 for the example of a linear interpolation function with four mesh nodes in its support. It is repeated $p_{\max}$ times until all particles have been considered. The complete workflow of the mesh–particle interpolation is given in Algorithm 4.

Algorithm 2. Overall particle–mesh interpolation workflow

1: Copy particle positions and strengths from host to device
2: Initialize `np_cell`
3: Determine cell indices of particles and store them in the cell list `p_cell`
4: Reduce `np_cell` to compute $p_{\max}$
5: Store particle coordinates in `particle_pos`
6: Store particle strengths in `particle_str`
7: Allocate and initialize extended mesh with replicated mesh nodes
8: Launch particle–mesh interpolation kernel
9: Stitch duplicated mesh nodes
10: Read back mesh nodes from device

Algorithm 3. Mesh–particle interpolation kernel

1: $w \leftarrow$ work item ID
2: $\chi \leftarrow$ indices of mesh cells assigned to w
3: **for** i from 1 to $p_{\max}$
  (a) $p \leftarrow$ index of the ith particle in χ
  (b) $\mathbf{x} \leftarrow$ coordinates of the ith particle in χ
  (c) **for** $\kappa \in \mathrm{supp}(W)$
    i. $\mu \leftarrow \chi + \kappa$ (shift of mesh node in target)
    ii. $M \leftarrow$ index of the mesh node pointed to by μ
    iii. **for** each particle property ω_i
      A. $\omega_i(p) \leftarrow \omega_i(p) + W(\mu, \mathbf{x})\omega_i(M)$

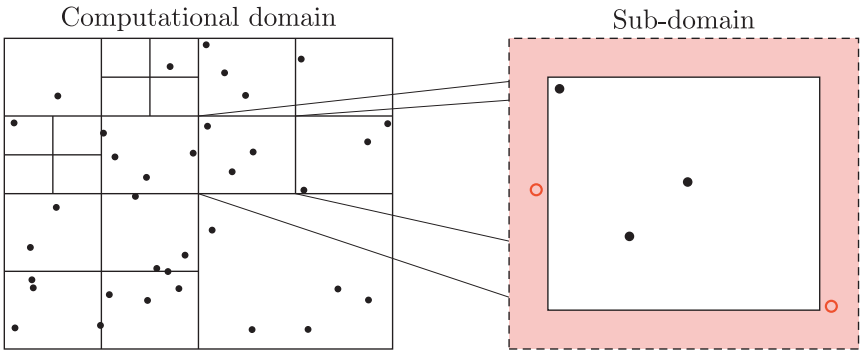


Fig. 8. In PPM, the computational domain (left) is adaptively subdivided and each sub-domain (right) is extended by a ghost layer (shaded red). The ghost layer is populated with ghost particles (red circles) that are copies of "real" particles (black dots) from neighboring processors. The present algorithm is applied locally per sub-domain and does not incur any additional inter-processor communication overhead. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

5. Integration in the PPM library

The above generic algorithms for GPU-accelerated interpolation are implemented in the PPM library [18,19]. PPM is a processor-independent programming middleware for parallel hybrid particle–mesh methods on distributed-memory computers. It implements a set of data and operation abstractions on an intermediate level of granularity, such as to expose network communication overhead, but hide the distribution of particle and mesh data across compute nodes (i.e., the data topology) [20]. Distributed-memory parallelism in PPM is based on adaptive geometric domain decomposition and ghost (halo) layers around the sub-domains, as shown in Fig. 8. After distributing the sub-domains onto the compute nodes, such that each node receives one or several sub-domains and good load balance is achieved, MPI (Message Passing Interface) communication is used to populate the ghost layers of all sub-domains with copies of the mesh nodes and particles from the respective neighboring processes. All solvers and numerical routines, including the present interpolation algorithms, are then applied locally per sub-domain, without requiring any further network communication.

Algorithm 4. Overall mesh–particle interpolation workflow

1: Copy particle positions and mesh node values from host to device
2: Initialize `np_cell`
3: Determine cell indices of particles and store them in the cell list `p_cell`
4: Reduce `np_cell` to compute p_{max}
5: Store particle coordinates in `particle_pos`
6: Construct extended mesh with replicated mesh nodes (for memory stride and coalesced access)
7: Launch mesh–particle interpolation kernel
8: Collect strengths of actual (non-dummy) particles
9: Read back particle strengths from device

Using the present OpenCL implementation of particle–mesh and mesh–particle interpolation, we extend the PPM library to multiple levels of parallelism. On the coarsest level, sub-domains are assigned to MPI processes that operate in separate memory address spaces. On the finest level, the light-weight GPU threads parallelize over the individual mesh cells within a sub-domain. Due to the presence of ghost particles (red circles in Fig. 8), all information is locally available and no additional MPI communication is incurred by using the present OpenCL implementation. This extends the present implementation to multi-GPU settings.

Since PPM is a general-purpose library, the particle data cannot be assumed to be sorted or arranged in any specific way when entering the interpolation routines. Also, PPM internally uses object-oriented data structures [19] that do not directly map onto the OpenCL memory model. This requires a number of "wrapper" or routines, as illustrated in Figs. 9 and 10 in yellow,[2] and several pre-and post-processing kernels (blue boxes *other than* the actual interpolation kernels in Figs. 9 and 10).

Particle–mesh interpolation (Fig. 9) starts by re-numbering the particles such that consecutively indexed particles are located in the same sub-domain. This is necessary because a process can be assigned multiple sub-domains. The re-numbering allows applying the GPU kernels locally per sub-domain on contiguous chunks of memory. The second step consists of allocating and populating the particle and mesh data buffers as described in Section 4.2. For each sub-domain in an MPI process, the OpenCL kernels are then run (potentially in parallel if multiple sub-domains and multiple GPUs are available on a com-

[2] For interpretation of color in Figs. 9 and 10, the reader is referred to the web version of this article.

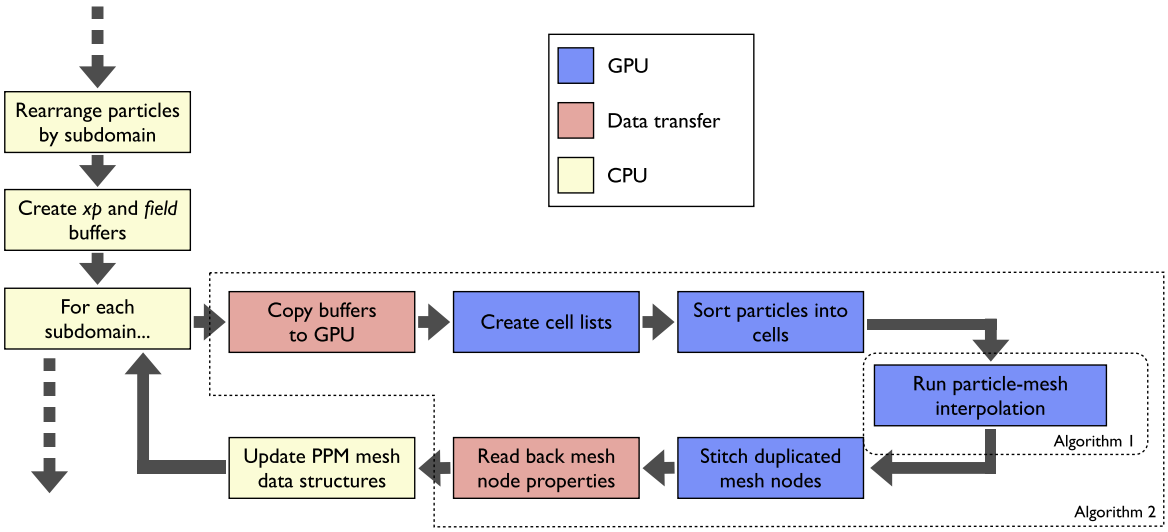


Fig. 9. Workflow scheme for GPU-accelerated particle–mesh interpolation in the PPM library.

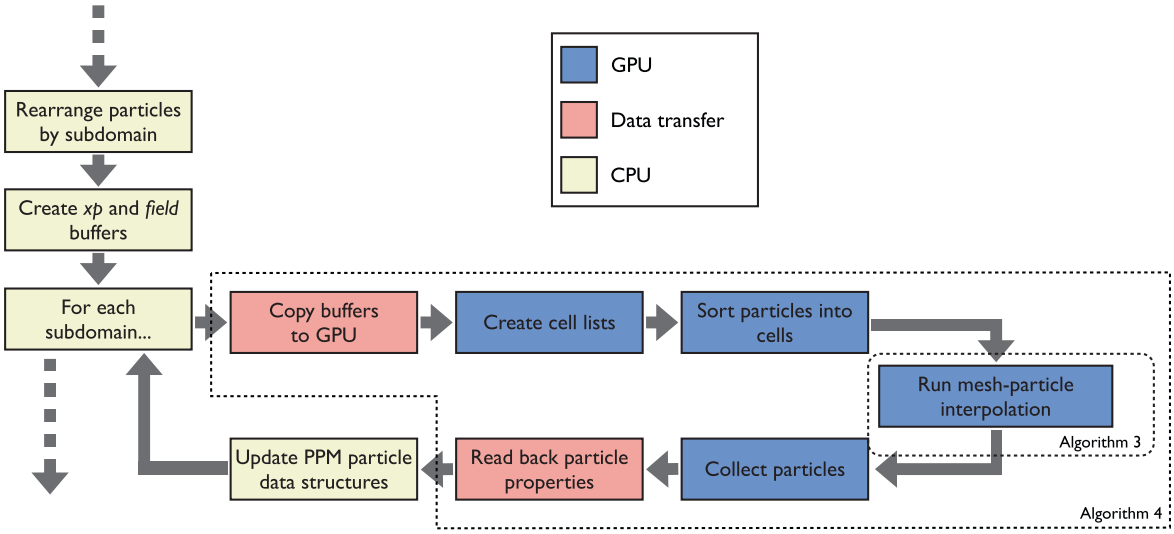


Fig. 10. Workflow scheme for GPU-accelerated mesh–particle interpolation in the PPM library.

pute node). This entails first copying the buffer data from the host memory to the device memory. Once on the GPU, the particles are sorted into mesh cells as described in Section 4.2.1. Then, Algorithm 1 is used before the replicated mesh nodes are stitched back together and the results copied back to the host memory. Finally, the results are copied from the flat buffers into the PPM mesh objects.

Mesh–particle interpolation (Fig. 10) proceeds analogously. The main difference is that the duplicated mesh nodes do not need to be stitched together after interpolation. Instead, the dummy particles need to be removed from the GPU buffer before the results are transferred back to the host memory.

6. Benchmarks

We benchmark the accuracy and runtime of the presented algorithms in 2D and 3D using both single and double-precision floating-point numbers. For the benchmarks we initialize one particle on each mesh node in the unit square and perturb the particle positions by uniform random numbers in $[-2\mathbf{h}, 2\mathbf{h}]$. This leads to a quasi-random particle distribution with 0 to 16 particles per mesh cell. For particle–mesh interpolation we sample the function $g(\mathbf{x}) = \exp\left(-\|\mathbf{x} - \mathbf{1}/\mathbf{2}\|_2^2/15\right)$ at the particle positions and interpolate the resulting particle strengths to the mesh nodes. The error is defined at each mesh node as the difference between the interpolation result and the exact value of g at the location of that mesh node. For mesh–particle

interpolation the same function is sampled at the mesh nodes and the error after interpolation is analogously defined on the particles.

We benchmark the OpenCL implementation on a NVIDIA Tesla C2050 GPU consisting of 448 CUDA cores organized into 14 SMPs with 1030 GFLOP/s single-precision and 515 GFLOP/s double-precision peak performance and 3 GB GDDR5 memory with ECC disabled for the benchmarks. The peak memory bandwidth of this GPGPU card is 144 GB/s. For comparison, and to demonstrate the portability of the OpenCL implementation, we also benchmark it on an ATI "Cayman" Radeon HD 6970 GPU featuring 1536 stream processors with 2.7 TFLOP/s single-precision and 683 GFLOP/s double-precision peak performance and 1 GB GDDR5 memory with a peak bandwidth of 176 GB/s. As a baseline we use both the highly optimized sequential Fortran 90 implementation available in the PPM library [18], as well as an OpenMP-parallelized version of the same PPM routines. All CPU code is compiled with the GCC Fortran compiler version 4.6.2 using the `-O3` optimization flag. Both CPU versions (sequential and multi-threaded) are run on an 8-core AMD FX 8150 at 4.2 GHz with 16 GB DDR3 SDRAM.

In order to minimize bank conflicts in the GPU's shared memory, and hence maximize the effective bandwidth, the block sizes (cf. Section 4.2) are set according to the number of shared memory banks and the maximum number of work items per compute unit. Both GPU devices tested have 32 shared memory banks. We hence always use 32 mesh cells along the x-direction.

The NVIDIA Tesla C2050 GPU can process up to 1024 work items in a workgroup, but can launch 1536 threads per compute unit. We hence choose 512 work items per workgroup, in order to launch three workgroups with 100% thread utilization. Thus, the block size is set to 32×16 in 2D and $32 \times 4 \times 4$ in 3D. The maximum workgroup size for the ATI "Cayman" Radeon HD 6970 GPU is 256 and we can launch 1536 threads per compute unit. Since 256 is a divisor of 1536, 100% occupancy is always guaranteed. Setting the number of work items in the x-direction to 32, we use block sizes of 32×8 in 2D and $32 \times 4 \times 2$ in 3D.

For the timings, we measure the runtime of GPU-based interpolation, consisting of all GPU (blue) stages of the workflows shown in Figs. 9 and 10. Additionally we measure the time to copy the data to the device memory (red). The sum of these times is compared with the runtime of the sequential and multi-threaded PPM implementations running on the CPU, where sorting of particles and re-ordering of data is not necessary. The *speedup* is defined as the ratio between the CPU wall-clock time and the GPU wall-clock time. For each interpolation kernel and GPU platform we also measure the sustained performance in GFLOP/s, only counting floating-point multiply and add operations, and use this to evaluate the *efficiency* of the implementations, defined as the fraction of the theoretical peak performance of the respective GPU that is actually sustained by the interpolation kernel. We only provide GFLOP/s rates for the actual interpolation kernels (i.e., Algorithms 1 and 3), but

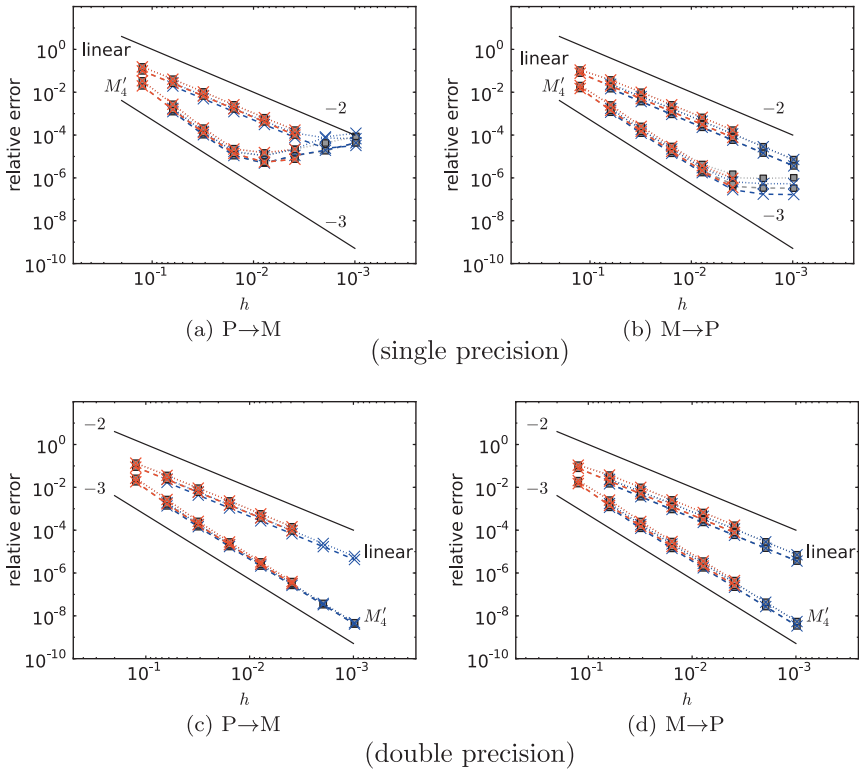


Fig. 11. Convergence plots for M_4' and linear interpolation in single precision (a/b) and double precision (c/d). We plot the ℓ^2 (dashed lines) and ℓ^∞ (dotted lines) norms of the relative interpolation errors for both the GPU (crosses) and CPU (gray squares) implementations in 2D (blue) and 3D (red). The solid black lines indicate the slopes of orders 2 and 3, respectively. (For interpretation of reference to color in this figure legend, the reader is referred to the web version of this article.)

not for the other modules run on the GPU (i.e., the pre-and post-processing kernels). The reason is that the latter perform virtually no floating-point multiply or add operations.

6.1. Accuracy

We benchmark the accuracy and correctness of the OpenCL implementations using both the M_4' and the linear interpolation functions on the above-described test case on the NVIDIA Tesla C2050 GPGPU. Fig. 11 shows the convergence plots for the 2D (blue) and 3D (red) case. When using single precision, the implementations do not converge below machine precision of 10^{-6} (Fig. 11(a) and (b)). The convergence plots also show the expected third-order convergence when using the M_4' scheme and second-order convergence for the linear interpolation scheme, in both the ℓ^2 and the ℓ^∞ norms of the error. In all cases, the errors on the CPU (gray squares) and on the GPU (crosses) are identical, demonstrating the correctness of the GPU implementations.

Table 2
Summary of the overall speedups (with and without the time required for data transfer to the device memory) measured for the OpenCL implementation on the **NVIDIA Tesla C2050** over the OpenMP reference implementation in the PPM library on the 8-core AMD FX 8150 CPU. The last column reports the sustained GFLOP/s rate for the interpolation kernels alone, i.e. Algorithms 1 and 3, and in parentheses the efficiency as the fraction of the theoretical peak performance (1030 GFLOP/s) sustained. Only the efficiency results for single precision are shown. For the double-precision kernels the GFLOP/s rates are half of those for the single-precision kernels, but the efficiencies remain the same (double-precision peak performance is 515 GFLOP/s). We only show the largest problems tested. Notice that speedups mentioned in the main text may correspond to other problem sizes.

Type	f	Dim.	Prec.	#Part.	Speedup		GFLOP/s
					w/o comm.	w/comm.	
P → M	Linear	2D	32 bit	4096k	3.0	0.6	124.2 (12.1%)
			64 bit	4096k	1.8	0.3	
		3D	32 bit	2048k	2.3	0.6	112.6 (10.9%)
			64 bit	2048k	1.8	0.4	
	M_4'	2D	32 bit	4096k	4.3	1.7	180.8 (17.6%)
			64 bit	4096k	2.9	0.6	
		3D	32 bit	2048k	4.8	1.7	432.6 (42.0%)
			64 bit	2048k	3.1	1.0	
M → P	Linear	2D	32 bit	4096k	1.8	0.4	64.3 (6.2%)
			64 bit	4096k	1.8	0.3	
		3D	32 bit	2048k	1.9	0.6	199.9 (19.4%)
			64 bit	2048k	1.5	0.4	
	M_4'	2D	32 bit	4096k	4.8	1.3	273.1 (26.5%)
			64 bit	4096k	3.6	0.8	
		3D	32 bit	2048k	13.0	5.1	648.1 (62.9%)
			64 bit	2048k	8.5	2.8	

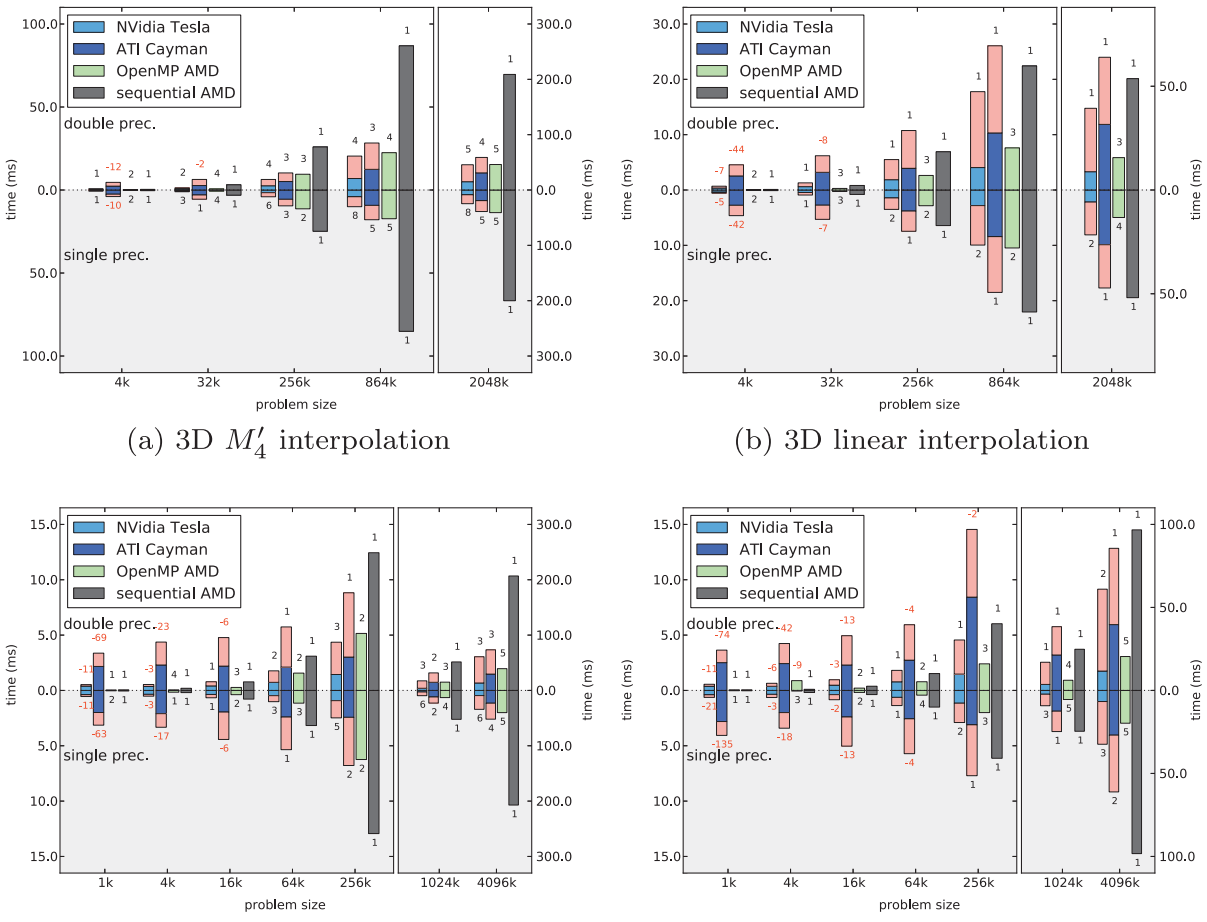
Table 3
Summary of the overall speedups (with and without the time required for data transfer to the device memory) measured for the OpenCL implementation on the **ATI Cayman Radeon HD 6970** over the OpenMP reference implementation in the PPM library on the 8-core AMD FX 8150 CPU. We only show the largest problems tested. Notice that speedups mentioned in the main text may correspond to other problem sizes.

Type	f	Dim.	Prec.	#Part.	Speedup		GFLOP/s
					w/o comm.	w/comm.	
P → M	Linear	2D	32 bit	4096k	0.7	0.3	95.9 (3.5%)
			64 bit	4096k	0.5	0.2	
		3D	32 bit	2048k	0.5	0.3	83.8 (3.1%)
			64 bit	2048k	0.5	0.2	
	M_4'	2D	32 bit	4096k	1.7	0.8	154.1 (5.7%)
			64 bit	4096k	1.3	0.5	
		3D	32 bit	2048k	2.1	1.1	184.2 (6.8%)
			64 bit	2048k	1.5	0.8	
M → P	Linear	2D	32 bit	4096k	0.4	0.2	57.0 (2.1%)
			64 bit	4096k	0.4	0.2	
		3D	32 bit	2048k	0.4	0.2	242.8 (9.0%)
			64 bit	2048k	0.4	0.2	
	M_4'	2D	32 bit	4096k	2.0	0.8	414.5 (15.3%)
			64 bit	4096k	1.6	0.7	
		3D	32 bit	2048k	5.7	2.8	889.0 (32.9%)
			64 bit	2048k	3.9	1.8	

6.2. Runtime

We compare the runtimes of the OpenCL implementations on the NVIDIA and ATI GPUs with the baselines provided by the sequential and multi-threaded OpenMP codes in the PPM library [18] run on the 8-core AMD CPU. For the OpenCL implementations we also take into account the time needed to sort the particles into the data structures outlined in Section 4.2.1 and to transfer the data to and from the device memory. Running both implementations for various problem sizes (i.e., numbers of particles and mesh nodes), we check how the implementations scale with problem size and how the speedups evolve. All OpenMP benchmarks were performed using eight threads distributed over the eight cores of the benchmark CPU. The speedups (OpenCL GPU vs. multi-threaded CPU) and sustained performances (interpolation kernels only) are measured in all cases. The values obtained for the NVIDIA Tesla are summarized in Table 2, those for the ATI Cayman in Table 3.

Fig. 12 shows the wall-clock times for particle–mesh interpolation on all four benchmark platforms. Bars for single-precision arithmetic point downward, for double-precision arithmetics upward. The numbers above the bars give the speedups with respect to the single-thread CPU implementation when taking communication overhead into account. The speedup of the GPU over the CPU grows with increasing problem size. The speedups obtained with single-precision arithmetic are about twice as large as those obtained with double-precision arithmetic. This is due to the fact that the GPUs used in our benchmarks require several clock cycles per double-precision instruction. On the CPU, however, single and double precision operations require approximately the same amount of time. The speedup increases with dimensionality of the space and with order (FLOP/byte) of the interpolation scheme, which can be ascribed to the increasing number of operations per particle. While the speedup of the GPU over the sequential CPU code, excluding transfer times, reaches up to 23, the speedups over



(a) 3D M_4' interpolation (b) 3D linear interpolation

(c) 2D M_4' interpolation (d) 2D linear interpolation

Fig. 12. Timings for particle–mesh interpolation. We compare the wall-clock times of the OpenCL implementation run on the NVIDIA Tesla GPU (light-blue bars) and on the ATI Cayman GPU (dark-blue bars) with those of the Fortran90-OpenMP implementation running on an 8-core AMD FX 8150 CPU at 4.2 GHz (8-thread: green bars, sequential: gray bars). The times needed to transfer the data to and from the GPU device memory are added in red. Bars pointing downwards refer to single-precision runs, bars pointing upwards to double-precision runs. The speedups with respect to the sequential CPU implementation are given above/below the bars. Speedups $1/x < 1$ are given as $-x$ in red. Notice the different time axes for the two parts of each plot. (For interpretation of reference to color in this figure legend, the reader is referred to the web version of this article.)

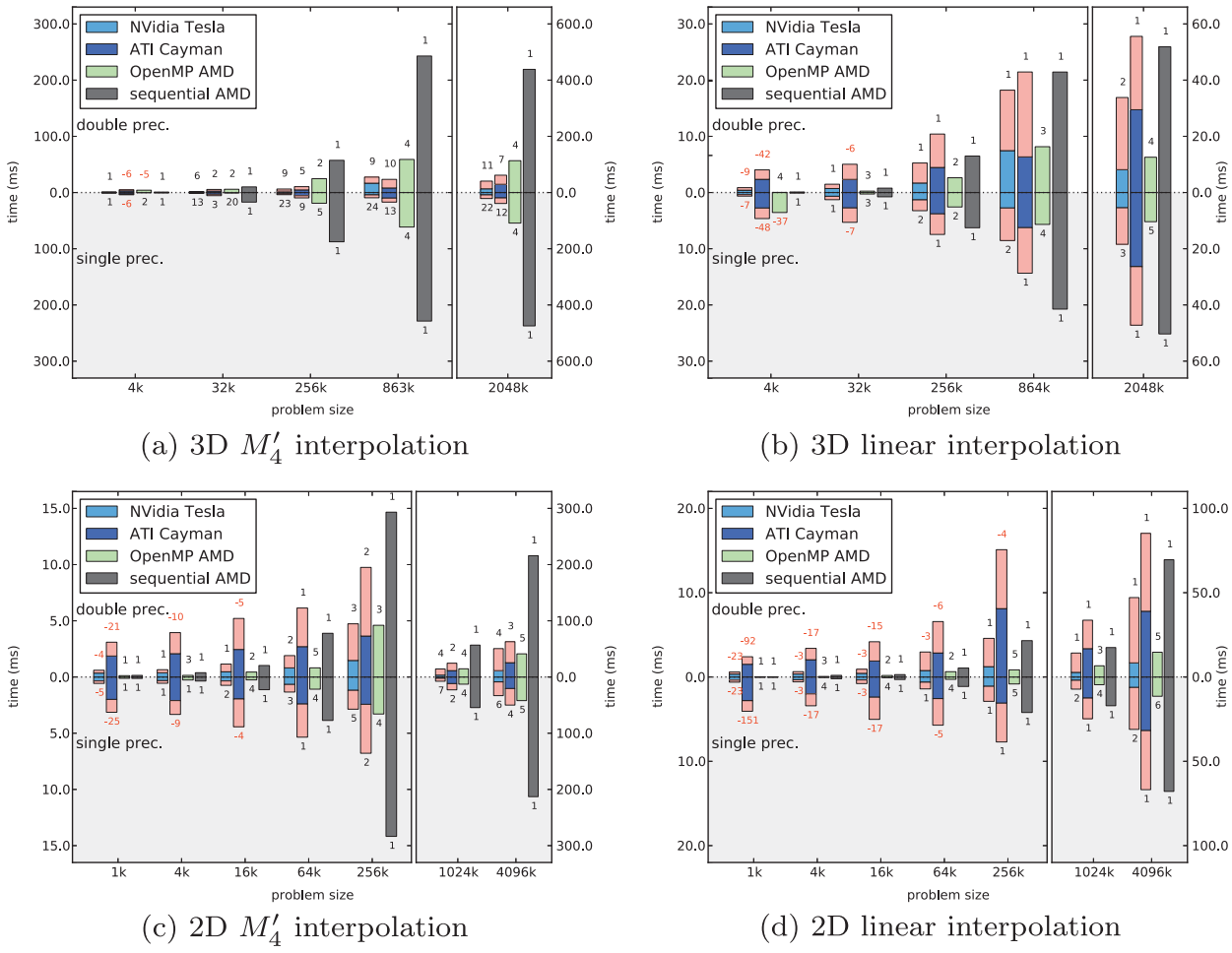


Fig. 13. Breakdown of the computational time on the NVIDIA Tesla C2050 GPU by kernel. We show the case (a) that delivered the largest speedup, i.e., 3D mesh–particle interpolation using the M_4' scheme in single precision and (b) the case that delivered the smallest speedup, i.e., 2D particle–mesh interpolation using the linear scheme in double precision.

the OpenMP-parallel CPU code show mixed results. The largest speedup over the parallel CPU code is observed when using M_4' interpolation in 3D with single-precision arithmetic and 256k particles. In this case, the NVIDIA Tesla C2050 solves the problem up to seven times faster than the CPU employing all 8 cores. This speedup reduces to 3-fold when also accounting for the data-transfer time. The OpenCL code on the ATI Cayman only outperformed the parallel CPU code when using M_4' interpolation. In all other cases, the 8-thread CPU code was faster. In the worst case (linear interpolation, 2D, double precision), both GPUs perform significantly worse than the 8-core CPU. This suggests that the most efficient implementation should be chosen based on the order of the interpolation scheme (support size and compute intensity), problem dimensionality, and problem size. The efficiency on the NVIDIA Tesla ranges from 10% for 3D linear interpolation to 42% for M_4' interpolation in 3D. As expected, computationally more intense kernels (3D and M_4') lead to higher efficiencies. Comparing the sustained performance on the GPU to the sustained performance on the CPU reveals the overhead imparted by building the auxiliary data structures and sorting the particles.

Fig. 14 shows the results for mesh–particle interpolation. On the GPU, mesh–particle interpolation is faster than particle–mesh interpolation, since the former does not use any synchronization barriers and also has a higher compute intensity (FLOP/byte). This is also reflected in the generally higher efficiencies in this case. As in the particle–mesh case, the speedups become smaller with decreasing dimensionality, decreasing problem size, increasing numerical precision, and decreasing order of accuracy of the interpolation scheme. The largest observed speedup over the parallel CPU code is 15-fold (7-fold when accounting for data-transfer time) when using M_4' interpolation in 3D with single precision on the NVIDIA Tesla. On the same device, the speedups with respect to the sequential CPU code are 57-fold and 22-fold without and with data-transfer time, respectively. A breakdown of the computational time by kernels for this case is shown in Fig. 13(a). The actual interpolation (Algorithm 3) only amounts to 14% of the total time; the rest is consumed by pre-and post-processing kernels and by data transfer. The case that delivered the smallest speedup is shown in Fig. 13(b). There, the actual interpolation kernel only accounts for 8% of the computer time, and 80% of the time are used for data transfer. The ATI Cayman only marginally outper-

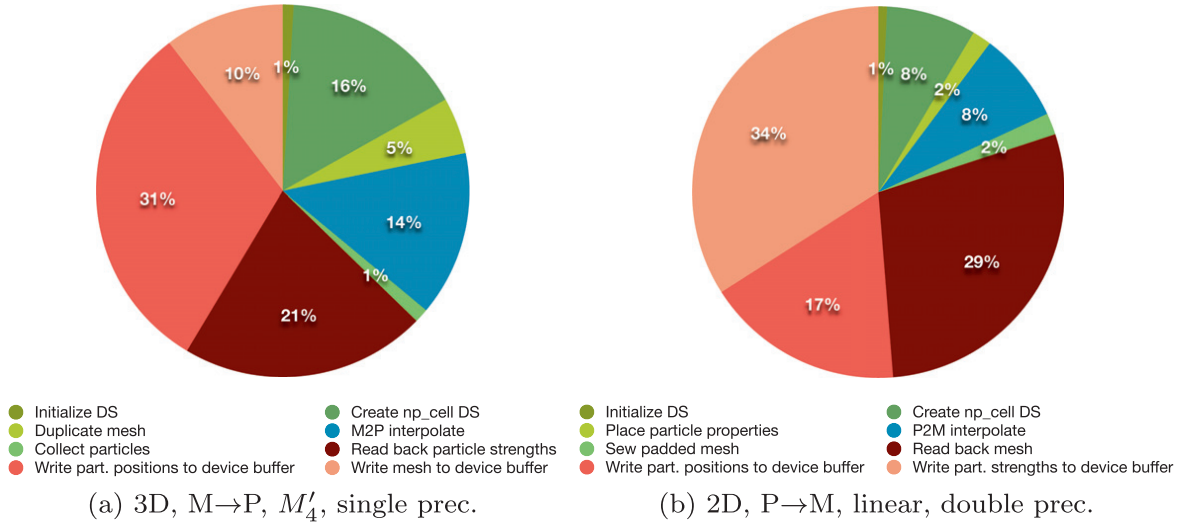


Fig. 14. Timings for mesh–particle interpolation. We compare the wall-clock times of the OpenCL implementation run on the NVIDIA Tesla GPU (light-blue bars) and on the ATI Cayman GPU (dark-blue bars) with those of the Fortran90-OpenMP implementation running on an 8-core AMD FX 8150 CPU at 4.2 GHz (8-thread: green bars, sequential: gray bars). The times needed to transfer the data to and from the GPU device memory are added in red. Bars pointing downwards refer to single-precision runs, bars pointing upwards to double-precision runs. The speedups with respect to the sequential CPU implementation are given above/below the bars. Speedups $1/x < 1$ are given as $-x$ in red. Notice the different time axes for the two parts of each plot. (For interpretation of reference to color in this figure legend, the reader is referred to the web version of this article.)

forms the 8-core CPU when using M'_4 interpolation. Its best case, M'_4 interpolation in 3D with single precision, shows a 6-fold speedup (4-fold when account for data-transfer time). In the worst case (linear interpolation, 2D, double precision) both GPUs perform significantly worse than the 8-core CPU (the NVIDIA Tesla is 3.2 times slower than the 8-thread CPU, the ATI Cayman 5.8 times). The efficiency again grows for more compute-intense kernels and reaches 63% for M'_4 interpolation in 3D on the NVIDIA Tesla. The fact that the speedup in the same case is not very large can again be attributed to the overhead of building the auxiliary data structures and parallel-reducing the results. The linear kernels are memory limited, and in 3D 70% of the theoretical peak memory bandwidth are sustainably utilized.

7. Conclusions and disussion

We have presented a portable OpenCL implementation of generic algorithms for SIMD-parallel particle–mesh and mesh–particle interpolation on GPUs. The same parallelization strategy has been used for both mesh–particle and particle–mesh interpolation, and it also readily generalized to 3D. Our algorithm is generic with respect to the mesh size, the input representation, the number of particles per mesh cell, and the number of particle properties that are to be interpolated. Large inhomogeneities in the particle density, however, lead to load imbalance and lower performance. In these cases, Adaptive Mesh Refinement (AMR) strategies are available to restore a more uniform particle distribution across mesh cells [24]. AMR codes have been parallelized on GPUs with about 10-fold speedup [25].

The present parallelization strategy uses threads (work items) over mesh cells and a particle-push, mesh-pull paradigm [7]. Particle data are reordered according to the memory access patterns of the work items in a workgroup. Additionally distributing multiple particles in a mesh cell across domain copies, and decomposing domain copies into blocks, leads to efficient use of the global memory bandwidth. Particle reordering also allows us to benefit from local memory and high cache hit rates. We avoid race conditions in particle–mesh interpolation by workgroup barriers and by replicating mesh nodes in ghost layers between neighboring blocks (workgroups).

We benchmarked both the accuracy and the computational cost of the presented OpenCL implementation on two GPU platforms against the existing, highly optimized single-thread CPU implementation in the PPM library [18] and a shared-memory-parallel OpenMP version respectively. The benchmarks have shown that speedups of up to 7-fold over an 8-thread CPU code and 22-fold over a sequential CPU code are possible with a generic OpenCL algorithm running on a GPU. This, however, depends on using an interpolation kernel with a high computation-to-memory ratio and rapidly diminishes when using double-precision arithmetics (due to the emulated double-precision support on the GPUs used). Also, the speedups observed for the present general-purpose implementation are below those reported for specialized codes in 2D [11,12]. This is mainly due to the time needed in the present implementation to sort the particle data into GPU-suited data structures. Without these pre-and post-processing kernels, i.e., if the calling program would already store the particle and mesh data in a GPU-friendly ordering, higher overall speedups could be realized, as can be extrapolated from the pie charts in Fig. 13. Rossinelli et al. reported a 35-fold speedup of their OpenCL implementation over their single-thread CPU implementation of mesh–particle interpolation using the M'_4 function in 2D with double precision [12]. For the same case, the present OpenCL

implementation shows a 19-fold speedup over the single-thread CPU implementation of the PPM library. Rossinelli et al.'s OpenGL implementation of 2D particle–mesh interpolation displayed a 26-fold speedup [8]. In single precision, Rossinelli et al.'s mesh–particle interpolation showed a 145-fold speedup [12], which is significantly larger than the 24-fold speedup of the present implementation in the same case.

From our efficiency measurements we also see that more compute-intense kernels with a larger number of floating-point operations per load-store operation allow higher efficiencies (up to 63% for M'_4 interpolation in 3D). Hence, 3D interpolation kernels use the hardware more efficiently than 2D ones, and M'_4 kernels have a higher efficiency than linear ones. The linear kernels were memory-limited in all cases.

Our results also confirm previous reports that linear interpolation schemes achieve smaller speedups on a GPU than higher-order schemes [14,12]. We believe that this is due to the lower FLOP/byte ratio of the linear kernels. The present results also show that on the hardware used, double-precision computations take about twice longer than single-precision ones. When using higher-order interpolation schemes, such as M'_4, this could be an issue since the interpolation error quickly drops below the single-precision machine epsilon. However, if solution accuracies around 10^{-5} to 10^{-6} are sufficient, single-precision arithmetic provides better speedups, especially for higher-order interpolation schemes with more compute-intense kernels. With the advent of APUs and better double-precision support on GPUs, this could be a temporary limitation.

Compared with the multi-threaded OpenMP reference implementation running on an 8-core CPU, the OpenCL code was in most cases slower (always slower when using linear interpolation). This is in agreement with previous reports [12,14] and illustrates the limitations of GPU acceleration, in particular when accounting for code development times (several weeks for the OpenCL implementation, less than a day for the OpenMP implementation). For particle–mesh interpolation, the OpenMP implementation does not incur any thread synchronization and hence does not require atomic operations. This leads to an almost perfect scaling of the OpenMP-Fortran90 code with the number of CPU cores. When taking data transfer times into account, significant speedups over the 8-core CPU could only be achieved for 3D mesh–particle interpolation using the M'_4 scheme in single precision.

Despite the modest speedups of the GPU implementation, however, outsourcing interpolation to the GPU may free CPU resources to work simultaneously on other tasks. Furthermore, GPUs may provide interesting options for real-time *in-situ* visualization of running simulations [26–28]. A bit more speed could probably be gained on the NVIDIA GPU when using CUDA instead of OpenCL [17,12], albeit at the expense of reduced portability. Aiming at a generic, portable code that could be integrated into the general-purpose PPM library [18], we chose not to do so.

OpenCL is a portable parallel computing language, and OpenCL code can also run on multi-core CPUs. For the present implementation, however, running the OpenCL code on the 8-core AMD CPU was slower than using the OpenMP code on the same CPU. We hence chose the OpenMP code as the baseline for the benchmarks presented here.

The presented particle–mesh and mesh–particle interpolations are available in the PPM library both in 2D and 3D. This extends the PPM library with OpenMP parallelism and transparent GPU support on distributed-memory parallel computers. In PPM, the presented algorithms are applied locally per sub-domain (i.e., per processor) of a domain decomposition. They thus have no influence on the network communication overhead of a distributed parallel simulation, assuming that the ghost (halo) layers are populated beforehand. If several nodes of a compute cluster are equipped with GPUs, this enables distributed-memory multi-GPU interpolation, offering a simple hybrid MPI-OpenCL platform for large hybrid particle–mesh simulations. The PPM library provides an application-independent middleware and is available free of charge and as open source from http://www.ppm-library.org.

Acknowledgments

F.B. and O.A. were funded by Grant #200021-132064 from the Swiss National Science Foundation (to I.F.S.). We thank Dr. Diego Rossinelli (CSE lab, ETH Zurich) and all members of the MOSAIC Group (ETH Zurich) for many fruitful discussions. We also thank Dr. Diego Rossinelli and Prof. Petros Koumoutsakos for their valuable comments on the final manuscript and for providing access to their ATI/AMD benchmark systems.

References

[1] R.W. Hockney, J.W. Eastwood, Computer Simulation Using Particles, Hilger, Bristol, 1988.
[2] L. Verlet, Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, Phys. Rev. 159 (1) (1967) 98–103.
[3] O. Awile, F. Büyükkeçeci, S. Reboux, I.F. Sbalzarini, Fast neighbor lists for adaptive-resolution particle simulations, Comput. Phys. Commun. (2012).
[4] Josh Barnes, Piet Hut, A hierarchical O($N \log N$) force-calculation algorithm, Nature 324 (1986) 446–449.
[5] L. Greengard, W.D. Gropp, A parallel version of the fast multipole method, Comput. Math. Appl. 20 (7) (1990) 63–71.
[6] R.W. Hockney, C.R. Jesshope, Parallel Computers, Hilger, Bristol, 1981.
[7] G. Stantchev, W. Dorland, N. Gumerov, Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU, J. Parallel Distrib. Comput. 68 (10) (2008) 1339–1349.
[8] D. Rossinelli, P. Koumoutsakos, Vortex methods for incompressible flow simulations on the GPU, Visual Comput. 24 (7) (2008) 699–708.
[9] P. Koumoutsakos, Inviscid axisymmetrization of an elliptical vortex, J. Comput. Phys. 138 (1997) 821–857.
[10] G.-H. Cottet, P. Koumoutsakos, Vortex Methods–Theory and Practice, Cambridge University Press, New York, 2000.
[11] D. Rossinelli, M. Bergdorf, G.H. Cottet, P. Koumoutsakos, GPU accelerated simulations of bluff body flows using vortex particle methods, J. Comput. Phys. 229 (9) (2010) 3316–3333.
[12] D. Rossinelli, C. Conti, P. Koumoutsakos, Mesh–particle interpolations on graphics processing units and multicore central processing units, Philos. Trans. Roy. Soc. A 369 (1944) (2011) 2164.

[13] Khronos OpenCL Working Group, The OpenCL Specification, Version 1.0, June 2009.
[14] K. Madduri, E.-J. Im, K.Z. Ibrahim, S. Williams, S. Ethier, L. Oliker, Gyrokinetic particle-in-cell optimization on emerging multi-and manycore platforms, Parallel Comput. 37 (9) (2011) 501–520.
[15] C. Conti, D. Rossinelli, P. Koumoutsakos, GPU and APU computations of finite time Lyapunov exponent fields, J. Comput. Phys. 231 (2012) 2229–2244.
[16] W. Hönig, F. Schmitt, R. Widera, H. Burau, G. Juckeland, M.S. Müller, M. Bussmann, A Generic Approach for Developing Highly Scalable Particle–mesh Codes for GPUs, Technical Report, Forschungszentrum Dresden-Rossendorf & Technical University of Dresden, Dresden, Germany, 2010.
[17] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, Jack Dongarra, From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming, Parallel Comput. (2011).
[18] I.F. Sbalzarini, J.H. Walther, M. Bergdorf, S.E. Hieber, E.M. Kotsalis, P. Koumoutsakos, PPM – a highly efficient parallel particle–mesh library for the simulation of continuum systems, J. Comput. Phys. 215 (2) (2006) 566–588.
[19] O. Awile, Ö. Demirel, I.F. Sbalzarini, Toward an object-oriented core of the PPM library, in: Proc. ICNAAM, Numerical Analysis and Applied Mathematics, International Conference, 2010, pp. 1313–1316.
[20] I.F. Sbalzarini, Abstractions and middleware for petascale computing and beyond, Int. J. Distrib. Syst. Technol. 1 (2) (2010) 40–56.
[21] J.J. Monaghan, Extrapolating B splines for interpolation, J. Comput. Phys. 60 (2) (1985) 253–262.
[22] D.B. Kirk, W.M.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann, 2010.
[23] NVIDIA, OpenCL Best Practices Guide, May 2010.
[24] M. Bergdorf, G.-H. Cottet, P. Koumoutakos, Multilevel adaptive particle methods for convection–diffusion equations, Multiscale Model. Simul. 4 (1) (2005) 328–357.
[25] H.Y. Schive, Y.C. Tsai, T. Chiueh, GAMER: a graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics, Astrophys. J. Suppl. Ser. 186 (2010) 457.
[26] X. Mei, P. Decaudin, B.G. Hu, Fast hydraulic erosion simulation and visualization on GPU, in: 15th Pacific Conference on Computer Graphics and Applications, PG'07, IEEE, 2007, pp. 47–56.
[27] R. Fraedrich, S. Auer, R. Westermann, Efficient high-quality volume rendering of SPH data, IEEE Trans. Visualiz. Comput. Graphics 16 (6) (2010) 1533–1540.
[28] P. Goswami, P. Schlegel, B. Solenthaler, R. Pajarola, Interactive SPH simulation and rendering on the GPU, in: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Eurographics Association, 2010, pp. 55–64.