PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A
MATHEMATICAL, PHYSICAL & ENGINEERING SCIENCES

Mesh−particle interpolations on graphics processing units and multicore central processing units

Diego Rossinelli, Christian Conti and Petros Koumoutsakos

Phil. Trans. R. Soc. A 2011 **369**, 2164-2175
doi: 10.1098/rsta.2011.0074

PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

Phil. Trans. R. Soc. A (2011) **369**, 2164–2175
doi:10.1098/rsta.2011.0074

Mesh–particle interpolations on graphics processing units and multicore central processing units

By Diego Rossinelli, Christian Conti and Petros Koumoutsakos*

Institute of Computational Science, ETH, Zurich 8092, Switzerland

Particle–mesh interpolations are fundamental operations for particle-in-cell codes, as implemented in vortex methods, plasma dynamics and electrostatics simulations. In these simulations, the mesh is used to solve the field equations and the gradients of the fields are used in order to advance the particles. The time integration of particle trajectories is performed through an extensive resampling of the flow field at the particle locations. The computational performance of this resampling turns out to be limited by the memory bandwidth of the underlying computer architecture. We investigate how mesh–particle interpolation can be efficiently performed on graphics processing units (GPUs) and multicore central processing units (CPUs), and we present two implementation techniques. The single-precision results for the multicore CPU implementation show an acceleration of 45–70×, depending on system size, and an acceleration of 85–155× for the GPU implementation over an efficient single-threaded C++ implementation. In double precision, we observe a performance improvement of 30–40× for the multicore CPU implementation and 20–45× for the GPU implementation. With respect to the 16-threaded standard C++ implementation, the present CPU technique leads to a performance increase of roughly 2.8–3.7× in single precision and 1.7–2.4× in double precision, whereas the GPU technique leads to an improvement of 9× in single precision and 2.2–2.8× in double precision.

Keywords: central processing units; graphics processing units; high-performance computing; mesh–particle

1. Introduction

Interpolations between particles and grids are key computational operations that are frequently performed in particle-in-cell codes [1,2] as they may pertain to plasma simulations, molecular dynamics with electrostatic interactions and vortex methods. In these codes, information has to be exchanged between particles that carry physical properties and grid nodes that are used to solve the field equations, through an interpolation. We consider two types of interpolation: the sampling of particle quantities at the grid point locations (particle–grid interpolations) and the sampling of grid quantities at the particle locations

*Author for correspondence (petros@ethz.ch).

One contribution of 26 to a Theme Issue ‘Discrete simulation of fluid dynamics: methods’.

(grid–particle interpolations). Their parallel implementations present substantial differences, as particle–grid interpolations involve a global data-scattering and grid–particle interpolations involve a global data-gathering.

Several studies have addressed the issues of particle–grid interpolations, converting global data-scattering into local data-gathering operations with some intermediate processing (sorting) [3], or taking advantage of special features of the underlying computing hardware [4]. Less effort has been put into finding how grid–particle interpolations can be efficiently implemented on multicore central processing units (CPUs) and graphics processing units (GPUs).

In this paper, we consider the one-dimensional grid–particle moment-conserving interpolation [1] that can be used for higher dimensions as tensorial products. The efficiency of grid–particle sampling hinges on the memory bandwidth of the underlying computer architecture and GPUs are well suited to accelerate this process owing to their high internal bandwidth. We investigate how grid–particle interpolation can be efficiently performed on GPUs and multicore CPUs, and we present two implementation techniques to map effectively the necessary computation to these architectures.

The text is organized as follows. In §2, we introduce the formulation for grid–particle interpolation and a unified CPU–GPU performance model. In §3, we present the CPU and GPU techniques; in §4, we report and discuss the observed performance; and we conclude with a discussion of the results in §5.

2. Method

(*a*) *Grid–particle interpolations*

We consider a set of particles whose two-dimensional trajectories $\mathbf{x}_p$ yield a set of ordinary differential equations involving a velocity field $\mathbf{u}$,

$$\frac{\mathrm{d}\mathbf{x}_p}{\mathrm{d}t} = \mathbf{u}(\mathbf{x}_p, t). \tag{2.1}$$

We assume that $\mathbf{u}$ is represented on a uniform grid with discrete points $\mathbf{u}_{i,j}$ and grid spacing h, and we use grid–particle interpolation to sample $\mathbf{u}(\mathbf{x}_p, t)$. This operation consists in convolving the grid points with the reconstruction kernel W,

$$\mathbf{u}_p = \sum_i \mathbf{u}_i^{\text{grid}} \cdot W\left(\frac{1}{h}(\mathbf{x}_p - \mathbf{x}_i^{\text{grid}})\right). \tag{2.2}$$

In this work, we build W as a tensor product of the M_4' kernel [5], which can be seen as a piecewise function of four cubic polynomials,

$$M_4'(x) = \begin{cases} w_0 & \text{if } x \in (-2,-1], \quad w_0 = \dfrac{1}{2}x^3 + x^2 + \dfrac{1}{2}x, \\[2ex] w_1 & \text{if } x \in (-1,0], \quad w_1 = -\dfrac{3}{2}x^3 - \dfrac{5}{2}x^2 + 1, \\[2ex] w_2 & \text{if } x \in (0,1], \quad w_2 = \dfrac{3}{2}x^3 + 2x^2 - \dfrac{1}{2}x, \\[2ex] w_3 & \text{if } x \in (1,2], \quad w_3 = -\dfrac{1}{2}x^3 - \dfrac{1}{2}x^2. \end{cases} \tag{2.3}$$

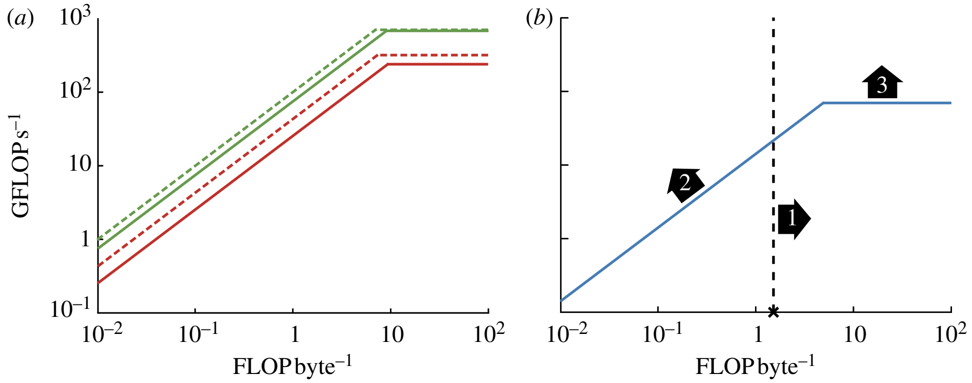


Figure 1. (*a*) Theoretical (dashed lines) and observed (solid lines) rooflines for the multicore AMD Opteron CPU (red) and the NVIDIA Tesla GPU (green), and (*b*) optimization steps for an implementation with operational intensity of 1.5. (Online version in colour.)

We note that the evaluation of w_0, w_1, w_2, w_3 can be efficiently performed with the Horner scheme. In one dimension, we can rewrite the grid–particle interpolation at the location x as

$$u(x) = w_3(\alpha - 1)u_{i-1} + w_2(\alpha)u_i + w_1(\alpha + 1)u_{i+1} + w_0(\alpha + 2)u_{i+2}, \tag{2.4}$$

where $i = \lfloor x/h \rfloor$ and $\alpha = x/h - i$. Analogously to equation (2.4), we perform a filtering of 4×4 grid points to sample the velocity field in two dimensions.

(*b*) *Performance model*

In order to compare different implementations of grid–particle interpolations, we use the roofline model [6], which can be used for both CPUs and GPUs. The model assumes that off-chip memory bandwidth is the constraining resource and for multicore CPUs and GPUs, in the context of particle-based flow simulations, this assumption is realistic.

The rooflines consist of the log–log plots shown in figure 1, where the maximum computing performance, measured in GFLOP s^{-1}, is reported against the operational intensity, measured in FLOP byte^{-1}. For an algorithm implementation (which we call the *kernel*), the operational intensity r is defined as the number of effective floating-point operations per byte of off-chip memory traffic. Two characteristic lines form a roofline: a left line that covers the values of r whose performance is bounded by the memory bandwidth and a right line that denotes the values of r for which the performance is bounded by the computing capacity of the underlying architecture. The point of minimum operational intensity for which the performance of a kernel is bounded by the computing power is called the ‘ridge point’. This point localizes the transition between the two regimes represented by the two lines. As the roofline is specific to the underlying computer architecture hardware and is kernel independent, it needs to be measured only once per computing hardware (and not once per kernel). Figure 1*a* shows the theoretical and observed rooflines of the computing hardware considered in this work: a multicore AMD CPU and an NVIDIA

GPU. As shown in figure 1*b*, by estimating the operational intensity of a computing kernel, we are now able to retrieve the maximum attainable computing performance. Any observed performance should lie somewhere below that point, as the roofline represents the upper bound. When the performance of a kernel is bounded by the memory bandwidth, three optimization steps can be applied, as depicted in figure 1*b*. Step 1 consists in increasing the operational intensity of the kernel by decreasing the memory traffic. The theoretical upper bound of the operational intensity is dictated by the compulsory cache misses. Step 2 consists in maximizing the observed bandwidth by considering the non-uniform memory access (NUMA) affinity, using wide registers to load and store data and performing unit–stride memory accesses. Step 3 improves the performance of the floating-point computation by increasing data-level parallelism, instruction-level parallelism, thread-level parallelism and the balance between instructions that are dispatched to different execution units.

(*c*) *Operational intensity*

We compute the operational intensity of grid–particle interpolation, assuming that the memory representation of the velocity field consists of two scalar grids, u and v, stored as two-dimensional contiguous arrays in single precision. Furthermore, we assume that the memory size of these arrays does not fit in the CPU cache but does fit within the CPU system memory and within the GPU global memory. To sample a vector field with c components, we estimate the following amount of computation:

$$F(c) = \underbrace{2\cdot 4}_{\text{localize grid points}} + \underbrace{2\cdot 4}_{\text{set-up } \alpha_0,\alpha_1} + \underbrace{2\cdot 4\cdot 3}_{\text{Horner eval.}} + \underbrace{4^2\cdot 3\cdot c}_{\text{filtering}} = 48\cdot c + 40 \text{ [FLOP]}.$$

The memory traffic depends on the spatial distribution of the particles. On CPUs, we observe a reduced memory traffic if the grid–particle interpolations of particles with close indices need almost the same grid points. When this happens, we benefit from spatial and temporal locality of the particles and therefore we maximize the data cache hits. We provide upper and lower bounds of the memory traffic by considering a good scenario and a bad scenario (actually the worst scenario). The good scenario happens when two particles with close indices employ 12 common grid points (out of 16) in grid–particle interpolation. If we assume that the 12 grid points are in-cache, then the memory traffic includes only four grid points per particle. The memory traffic per particle M reads

$$M^{\mathrm{CPU}}(c) = (\underbrace{2}_{\text{particle location}} + \underbrace{4\cdot c}_{\text{not in-cache}} + \underbrace{2\cdot c}_{\text{write-allocate}})\cdot 4 = 24\cdot c + 8 \text{ [bytes]}.$$

Multicore CPUs offer special instructions to avoid write–allocate write misses, the so-called non-temporal write instructions, which are accessible through the Streaming Single Instruction, Multiple Data (SIMD) Extension (SSE) intrinsics (such as the `_mm_stream_ps` instruction). If one makes use of such instructions, the memory traffic per particle reads $M^{\mathrm{CPU/NT}}(c) = 20\cdot c + 8$.

The worst scenario emerges when two particles with close indices share no grid point in grid–particle interpolation, which results in $M^{\mathrm{CPU}}_{\mathrm{worst}} = 72\cdot c + 8$ and $M^{\mathrm{CPU/NT}}_{\mathrm{worst}} = 68\cdot c + 8$ bytes if one uses non-temporal write instructions.

On GPUs, the good scenario appears when we represent u and v with OpenCL image objects, as images benefit from texture caches. We assume that texture caches show a similar (or better) spatial locality with respect to CPU caches. The memory traffic in this case is $M^{\mathrm{GPU}} = 20 \cdot c + 8$ bytes. The worst scenario on GPUs happens when particles do not show spatial locality, and is estimated by representing u and v as buffer objects that are read from global memory. In this case, we have $M^{\mathrm{GPU}}_{\mathrm{worst}} = 68 \cdot c + 8$ bytes. From these considerations, we estimate the bounds of the operational intensity for grid–particle interpolation (with $c = 2$) to be $0.9 < r_{\mathrm{CPU}} < 2.5$ on the CPU and $0.9 < r_{\mathrm{GPU}} < 2.9$ on the GPU. Because the average operational intensity could lie anywhere between the lowest and the highest possible values, we assume $r_{\mathrm{CPU}} = 1.5$ ($r_{\mathrm{CPU/NT}} = 1.88$) and $r_{\mathrm{GPU}} = 1.9$.

For double-precision computation, on multicore CPUs we assume an operational intensity of $r_{\mathrm{CPU}} = 0.75$ and $r_{\mathrm{CPU/NT}} = 0.94$. The estimation of the operational intensity on the GPU is problematic, as currently GPUs do not natively support double-precision texture formats. We therefore assume that particle positions, resulting samples and processing are in double precision, whereas the grid data are represented as a single-precision texture. The memory traffic in a good scenario thus reads $M^{\mathrm{GPU}} = 24 \cdot c + 16$ bytes; in a bad scenario, we have $M^{\mathrm{GPU}}_{\mathrm{worst}} = 72 \cdot c + 16$ bytes. Based on these considerations, we estimate an operational intensity of $r_{\mathrm{GPU}} = 1.49$.

3. Implementation

We use SSE intrinsics and OpenMP to increase the parallelism on multicore CPUs. OpenMP is also used to obtain maximum memory bandwidth through memory affinity. For GPU computing, we use the emerging OpenCL framework[1] and compare it with Compute Unified Device Architecture (CUDA),[2] a popular framework for GPU computing.

The OpenCL execution model is split into two parts: *kernels* that execute on one or more *OpenCL devices* (in our case the GPUs), and a *host program* that executes on the CPU (the *host*). When a kernel is submitted for execution by the host, the OpenCL devices are capable of running multiple instances of the same kernel in parallel. One kernel instance is called a *work item*. Work items are executed in parallel and can process different data.

In OpenCL, there are two different types of memory objects: buffers and images. Buffers are plain one-dimensional arrays, whereas image objects have special hardware support and store two-dimensional/three-dimensional arrays in a format optimized for data reads. Image elements (pixels) are accessed only with built-in functions and consist of four-component vectors (red, green, blue, alpha (RGBA)). The performance gain achieved by GPUs is very fragile: drastic performance degradation can be observed for different reasons. Examples are low kernel occupancy (high per work-item register usage), uncoalesced memory transactions, diverging control flow instructions and memory indirections. Optimal performance is achieved when work items are *homogeneous*, i.e. when the small tasks perform the same operations.

[1] See http://www.khronos.org/opencl/.

[2] See http://www.nvidia.com/object/cuda_home_new.html.

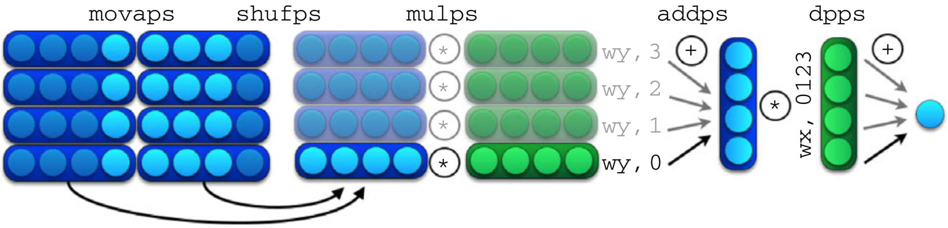


Figure 2. The grid data (blue) and the filter weights (green) in the SSE-based filtering of a grid–particle interpolation. (Online version in colour.)

(*a*) *Implementation on multicore central processing units*

The CPU implementation makes full use of SSE intrinsics and performs the sampling of all particles, considering only one scalar field at a time; therefore, the grid–particle interpolations of u and v are done in two independent passes. The input arguments are the two data streams, representing the particle positions, and the scalar grid, representing one component of the velocity field. The output is a one-dimensional data stream that contains the sampled values. All the streams are 16 byte aligned in memory.

The implementation consists of a main for-loop where all the particles are processed. To achieve maximum write bandwidth, we store the output as 16 byte transactions with non-temporal write instructions, meaning that we need to process four particles per iteration. As depicted in figure 2, the first step of the filtering consists in loading eight 16 byte words of grid data corresponding to the set of grid points in the proximity of the currently processed particle.

In a second step, we perform a 2 : 1 data reduction by discarding the unnecessary data by shuffling and using control flow instructions. The data are further 4 : 1 reduced by a vertical filtering with the y-weights (`wy,0123`), using four vector multiplications and three vector additions. Finally, the data are reduced by a horizontal filtering with the x-weights, using the dot product instruction (`dpps`). Alternatively, we replace the ‘dpps’ instructions by transposing the partially filtered data of the four independent particles, using `_MM_TRANSPOSE4_PS` and performing vertical additions. This implementation requires 130 lines of the C++/SSE intrinsics code.

Analogously, for the version in double precision, we process two particles per iteration and perform 16 byte non-temporal write transactions, after the 2 × 2 matrix transposition and vertical addition. Because of the 2 : 1 vector width ratio of SSE registers between single and double precision, the kernel in double precision has roughly twice the amount of assembly instructions with respect to those of the single-precision kernel and thus is more prone to instruction cache misses. The double-precision kernel requires 240 lines of C++/SSE intrinsics code.

(*b*) *Implementation on graphics processing units*

Grid–particle interpolation is performed independently on u and v. Because every work item is mapped to a scalar grid–particle interpolation, the amount of parallel work items is twice the number of particles.

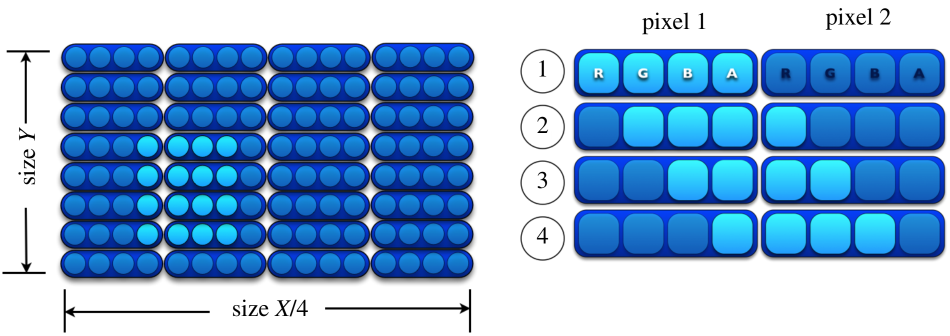


Figure 3. Uniform grids are represented as image objects, where each pixel contains four grid points (left). The light blue squares depict relevant grid points for grid–particle interpolation. (Online version in colour.)

The sampling of the velocity field takes place at arbitrary locations; therefore, the kernel cannot perform coalesced reads and so we cannot afford to use buffer objects to represent u and v. Instead, the representation of the velocity field as image objects is ideal, as there are chances that nearby particles show spatial locality in the resampling process. Within a pixel, all the image channels are used. The way we access the image is designed to minimize the number of fetched pixels. Figure 3 shows the storage format of the velocities. The light blue channels contain the velocities needed for one grid–particle interpolation. An efficient data extraction is challenging, as it should avoid memory indirections and control flow instructions. We perform this operation by using a superposition of Boolean weights. Velocities are accessed row by row, and for each row we always read two pixels. The first pixel contains the leftmost relevant grid points in that row, and the second element is simply the next pixel. There are four possible extractions of the relevant grid points, as depicted in figure 3. With the data read from the image, we create all four possibilities (by shuffling and swizzling the components) and store them into registers. We then multiply those registers with Boolean weights (obtained with work-item indices and by using ‘logical-AND’ and equivalence operators), which are used to determine the correct configuration for the current set of velocities. These vectors are then added up with vector operations, resulting in the correct extraction of the data. This method is more efficient than using if–else statements since the latter would cause diverging branches while executing the kernel, which in turn deteriorates the performance.

In OpenCL, the implementation consists of 55 and 57 lines, for single and double precision, respectively. The similar CUDA kernel consists of 63 lines, for both single and double precision.

In single precision, both OpenCL and CUDA generate a program that takes 32 registers per work item and leads to an occupancy of 0.5 on the GPU considered in this paper. The OpenCL runtime generates a program of 212 PTX instructions (excluding the declarative instructions), 178 of which are floating-point instructions. CUDA generates a program of 301 PTX instructions, 264 of which are floating-point instructions.

In double precision, OpenCL generates a program that takes 74 registers per work item and leads to an occupancy of 0.18, generating 246 PTX instructions, 210 of which are floating-point instructions. CUDA generates a program that takes 50 registers per work item and leads to an occupancy of 0.25. The program contains 330 PTX instructions, 293 of which are floating-point instructions.

4. Observed performance

CPU performance results were obtained on a 16-core CPU: an AMD 4P system of quad-core 'Shanghai' Opteron 8380 at 2.5 GHz. On this CPU, we observed a peak bandwidth of 24.5 GB s^{-1} of the theoretical 42.6 GB s^{-1}, and a peak computing performance of 237 FLOP s^{-1} of the theoretical 320 GFLOP s^{-1}. To measure the peak bandwidth, we have optimized the STREAM benchmark code [7] with the SSE intrinsics in order to guarantee proper memory accesses. To measure the peak computing performance, we have written an SSE code that evaluates a sequence of 64th-order polynomials in parallel. The code is written so as to balance SIMD multiplications and additions.

GPU performance results were obtained on a Tesla T10 GPU (model S1070), linked to the above-mentioned system through the PCIe bridge. We observed a peak internal GPU bandwidth of 73 GB s^{-1} of the theoretical 102 GB s^{-1} and a peak computing performance of 680 GFLOP s^{-1} of the theoretical 690 GFLOP s^{-1}. To measure the peak internal bandwidth, we used the 'CUDA bandwidth test', which exclusively makes use of the 'cudaMemcpy' function to copy the content of a buffer object into another one. To measure the computing performance on the GPU, we have written an OpenCL code that evaluates a sequence of 64th-order polynomials in parallel.

For both the CPU and the GPU, the measured peak bandwidth and peak computing performance were used to build the corresponding rooflines.

The reference CPU implementations are written in 'plain' C++ with OpenMP and compiled with the Intel C/C++ compiler and the flags `icc-O3-funroll-loops-ipo-fno-alias`.

To evaluate the different implementations, we have written a benchmark that considers $N \times M$ particles and a velocity field represented by a grid of $N \times M$ points. The benchmark initializes the particle locations on the grid points with a random displacement of a few grid spacings and performs the grid–particle interpolations of the velocity field.

(*a*) *Best single-precision performances*

Figure 4 shows the best performance of the single-precision grid–particle interpolation kernels for both the multicore CPU and the GPU. On the CPU (figure 4*a*), for the single-threaded C++ implementation and a system size of 512×256 we observe 0.8 GFLOP s^{-1}, corresponding to 27 per cent of the peak performance achievable with one core and exclusively using plain ×87 instructions. We note that the performance of this implementation is considered computationally bounded as the operational intensity at the right of the ridge point (blue roofline). We consider that the main reason for the suboptimal performance is the lack of overlap between computation and memory accesses and the excessive amount of capacity misses (in data cache). For a system size

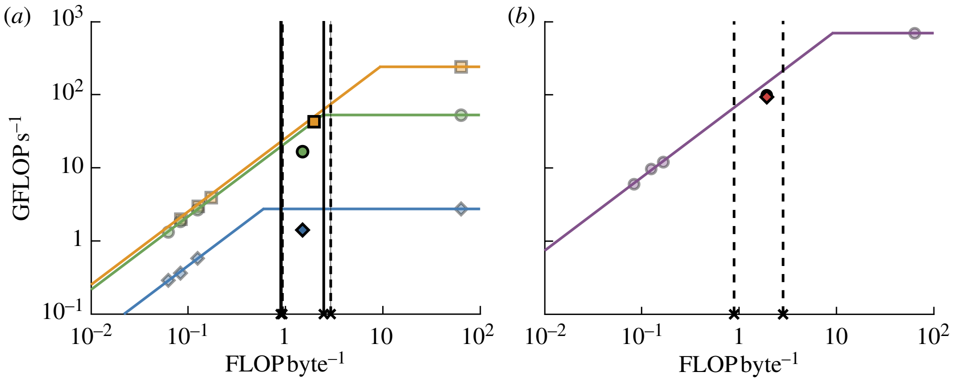


Figure 4. Rooflines and performances of grid–particle interpolations in single precision on (*a*) CPU and (*b*) GPU, C++ single-threaded (blue diamond), C++ multi-threaded (green circle), SSE multi-threaded (orange square), GPU-OpenCL implementation (red diamond) and GPU-CUDA implementation (violet circle). The black lines denote the ranges of the operational intensities of the CPU (black lines, solid), CPU/NT (black lines, dashed) and GPU (black lines, dashed). (Online version in colour.)

of 1024 × 512 particles and corresponding grid points, the 'plain' 16-threaded C++ implementation reaches 12 GFLOP s^{-1}, which corresponds to 38 per cent of the attainable peak performance using all the cores with ×87 instructions. We note that the performance of the multi-threaded implementations is bounded by the memory bandwidth (green and orange rooflines). Also, the multi-threaded implementations face some NUMA-related issues: particles allocated in one node are not guaranteed to exclusively use the grid points allocated on that node. With a system size of 1024 × 512 particles and corresponding grid points, the present CPU technique reaches a maximum performance of 43 GFLOP s^{-1}, which corresponds to 95 per cent of the attainable peak performance. This increase is explained by the reduced amount of instructions and a consequent better overlap between computation and memory traffic, as SSE instructions reduce the number of FLOP instructions roughly by a factor of 4 (the SIMD register width). This technique also shows a better bandwidth as it performs exclusively 16 byte aligned memory accesses and avoids write–allocate write misses. Since we access the grid data exclusively as 16 byte words, more data are loaded than the quantity that is effectively needed. This translates into a major performance bottleneck inherent to this technique.

On GPU, the OpenCL implementation shows a maximum performance of 94 GFLOP s^{-1}, corresponding to 68 per cent of the achievable peak performance, whereas the CUDA implementation is able to reach 99 GFLOP s^{-1}, corresponding to 72 per cent of the maximum achievable performance. We note that the difference between the best performance in OpenCL and that in CUDA is roughly 5 per cent.

A drawback of the GPU technique is that the quantity of data loaded is twice as much as the data that are used for the computations. This fact and the suboptimal occupancy of GPU implementations (0.5 instead of 1) are the major bottlenecks that prevent higher efficiency rates.

(*b*) *Performance versus system size for single and double precision*

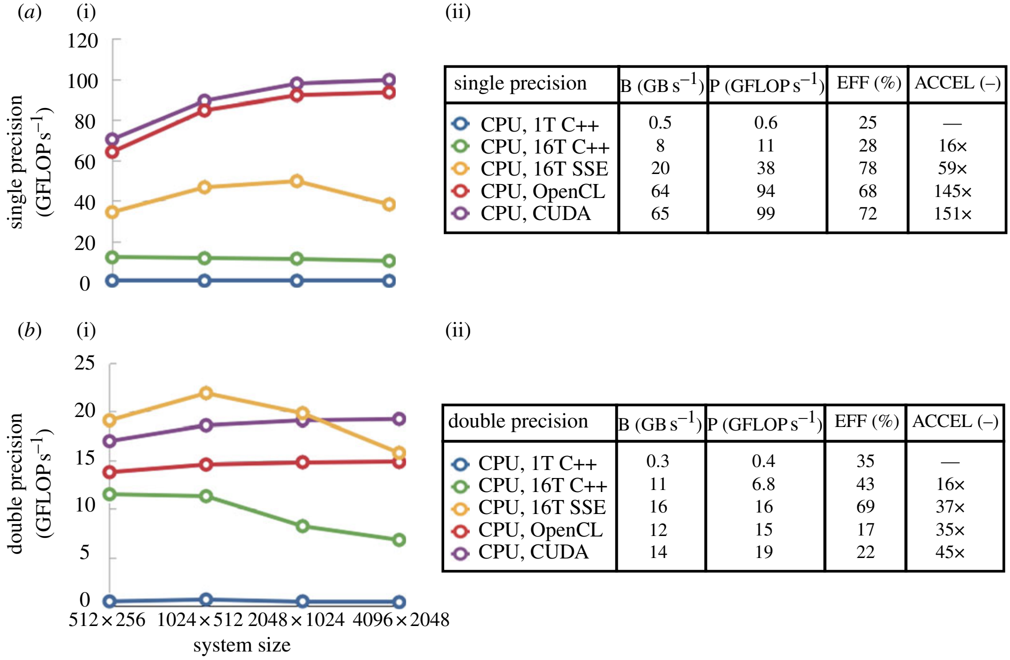
Figure 5*a*(i), *b*(i) shows the observed performance against the system size (number of particles and grid points) in single and double precision. The tables in figure 5 show the 'effective bandwidth' (B), performance (P), efficiency and acceleration factor of each implementation for a system size of 4096 × 2048 particles and corresponding grid points. In single precision, we observe that the performance of the C++ implementation deteriorates by increasing the system size: from 0.8 to 0.7 GFLOP s^{-1} for single-threaded execution, and from 12.3 to 10.6 GFLOP s^{-1} for multi-threaded execution.

The performance of the SSE implementation increases until the system size of 1024 × 512 particles and corresponding grid points, where the acceleration is 60× over the single-threaded C++ implementation. Since the SSE implementation makes use of non-temporal write instructions, it minimizes cache pollutions and at the same time maximizes the output bandwidth. This explains why performance deterioration starts only at the system size 2048 × 1024 considered, reaching 41 GFLOP s^{-1}. For the system size of 4096 × 2048, the speed-up of the SSE implementation over the C++ multi-threaded implementation is 3.6× and shows an efficiency of 78 per cent. As illustrated in figure 5*a*(i), GPU implementations constantly show superior performances compared with the CPU technique. Both GPU implementations show an increasing performance by increasing system size. Also, the CUDA kernel performance is always higher than the OpenCL by roughly 5 per cent.

In double precision, we observe that the performance of the single-threaded C++ implementation decreases faster than in single precision, dropping from 0.7 to 0.4 GFLOP s^{-1}, which corresponds to 35 per cent of the maximum achievable single-threaded non-SIMD performance. The multi-threaded C++ implementation shows a similar performance degradation, from 11.5 to 6.8 GFLOP s^{-1}, which corresponds to 43 per cent of the maximum achievable performance. The best performance of the SSE implementation is observed for a system size of 1024 × 512, reaching 22 GFLOP s^{-1}, which corresponds to 95 per cent of the maximum achievable performance. For system sizes larger than the optimum, the performance decreases owing to capacity cache misses. The phenomenon appears at a lower system size than in the case of single precision as the data are twice as large. Again, the use of non-temporal write instructions delays performance deterioration with respect to the system size. For a system size of 4096 × 2048, the SSE code in double precision shows an efficiency of 69 per cent and a speed-up of 2.4× over the multi-threaded C++ code.

Both GPU implementations are four times slower than their single-precision counterparts, indicating a disparity between the computing resources for double and single precision on the GPU. This problem is reflected also on the efficiency rates, which are relatively poor (17% for OpenCL and 22% for CUDA). We note that the CPU SSE implementation is outperforming the OpenCL kernel, and the CUDA kernel as well if the system size is less than 4096 × 2048.

In double precision, the CUDA implementation is almost 30 per cent faster than OpenCL. We attribute this difference to the disparity in the number of registers of the respective PTX programs (50 versus 74) and therefore their dissimilar occupancy (0.25 versus 0.18). We note that, for the largest system size considered, we observe a speed-up of 40× for the SSE implementation, 35× for OpenCL and 45× for CUDA.



single precision	B (GB s^{-1})	P (GFLOP s^{-1})	EFF (%)	ACCEL (–)
○ CPU, 1T C++	0.5	0.6	25	—
○ CPU, 16T C++	8	11	28	16×
○ CPU, 16T SSE	20	38	78	59×
○ CPU, OpenCL	64	94	68	145×
○ CPU, CUDA	65	99	72	151×

double precision	B (GB s^{-1})	P (GFLOP s^{-1})	EFF (%)	ACCEL (–)
○ CPU, 1T C++	0.3	0.4	35	—
○ CPU, 16T C++	11	6.8	43	16×
○ CPU, 16T SSE	16	16	69	37×
○ CPU, OpenCL	12	15	17	35×
○ CPU, CUDA	14	19	22	45×

Figure 5. (i) Performance against the system size, (ii) bandwidth, computing performances and efficiency for (a) single- and (b) double-precision computations. (Online version in colour.)

5. Conclusions

We have presented two techniques to efficiently compute grid–particle interpolations on GPUs and multicore CPUs. On the computing hardware considered in this work, the CPU technique shows a maximum performance of 44 GFLOP s^{-1} in single precision, corresponding to 95 per cent of the maximum achievable performance. The GPU implementation of grid–particle interpolation employs textures (OpenCL images) and adopts a superposition of Boolean weights to replace potentially diverging control flow instructions and memory indirections. In single precision, the GPU technique achieves a maximum computing performance of 94–99 GFLOP s^{-1}, reaching 68–72% of the attainable performance. With respect to a single-threaded plain C++ implementation, the present CPU technique is up to 70 times faster, whereas the GPU technique is 150 times faster. Compared with the plain C++ 16-threaded CPU execution, for the CPU technique we observe an acceleration factor of 3.7× and 9× on the GPU.

In double precision, the CPU technique achieves a peak performance of 22 GFLOP s^{-1}, which corresponds to 69 per cent of the estimated achievable performance, whereas the GPU implementation reaches 15–19 GFLOP s^{-1}, attaining an efficiency of around 20 per cent. The CPU implementation shows a speed-up of 40× with respect to the single-threaded plain C++ counterpart, whereas the GPU codes show a maximal acceleration of 45×. Over the multi-threaded C++ implementation, the CPU and GPU techniques show a maximum acceleration of 2.4× and 2.8×, respectively.

We would like to thank Adrian Ulrich, Dr Oliver Byrde and Teodoro Brasacchio, from the Brutus cluster support team at ETH Zurich, for the invaluable help and support in using the GPUs and CPUs employed in this work. We would also like to thank Dr Michael Bergdorf (D. E. Shaw research) and Babak Hejazialhosseini (CSE Laboratory, ETH Zurich) for their very helpful suggestions and discussions.

References

1 Hockney, R. W. & Eastwood, J. W. 1988 *Computer simulation using particles*, 2nd edn. Bristol, UK: Institute of Physics Publishing.

2 Birdsall, C. K. 1991 Particle-in-cell charged-particle simulations, plus Monte-Carlo collisions with neutral atoms, pic-mcc. *IEEE Trans. Plasma Sci.* **19**, 65–85. (doi:10.1109/27.106800)

3 Stantchev, G., Dorland, W. & Gumerov, N. 2008 Fast parallel particle-to-grid interpolation for plasma pic simulations on the GPU. *J. Parallel Distrib. Comput.* **68**, 1339–1349. (doi:10.1016/j.jpdc.2008.05.009)

4 Rossinelli, D., Bergdorf, M., Cottet, G. H. & Koumoutsakos, P. 2010 GPU accelerated simulations of bluff body flows using vortex particle methods. *J. Comput. Phys.* **229**, 3316–3333. (doi:10.1016/j.jcp.2010.01.004)

5 Koumoutsakos, P. 2005 Multiscale flow simulations using particles. *Annu. Rev. Fluid Mech.* **37**, 457–487. (doi:10.1146/annurev.fluid.37.061903.175753)

6 Williams, S., Waterman, A. & Patterson, D. 2009 Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **52**, 65–76. (doi:10.1145/1498765.1498785)

7 McCalpin, J. D. 1995 Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture* (*TCCA*) *Newsletter*, December, pp. 19–25.