Computer Physics Communications 184 (2013) 617–627

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

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

Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method

Jose M. Domínguez, Alejandro J.C. Crespo *, Moncho Gómez-Gesteira

EPHYSLAB Environmental Physics Laboratory, Universidade de Vigo, Campus As Lagoas s/n, 32004, Ourense, Spain

ARTICLE INFO

Article history:
Received 21 November 2011
Received in revised form
16 October 2012
Accepted 18 October 2012
Available online 26 October 2012

Keywords:
Optimization
Performance
HPC
GPU
Meshfree methods
SPH
OpenMP
CUDA

ABSTRACT

Much of the current focus in high performance computing (HPC) for computational fluid dynamics (CFD) deals with grid based methods. However, parallel implementations for new meshfree particle methods such as Smoothed Particle Hydrodynamics (SPH) are less studied. In this work, we present optimizations for both central processing units (CPU) and graphics processing units (GPU) focused on a Lagrangian Smoothed Particle Hydrodynamics (SPH) method. In particular, the obtained performance and a comparison between the most efficient implementations for CPU and GPU are shown using the DualSPHysics code.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

Numerical methods are demanded as useful tools in engineering and science to solve complex problems. The main advantage of this approach is the capability to simulate complex scenarios without building costly scale models, and provide physical data that can be difficult, or even impossible, to measure in a real model. The physical governing equations of a numerical method can be solved with the help of a grid using a Eulerian description or can be solved without it following a Lagrangian description. We will here focus on meshfree methods, which make easier the simulation of problems with large deformations, complex geometries, nonlinear material behavior, discontinuities and singularities. In particular, the so-called Smoothed Particle Hydrodynamics (SPH) method will be employed here as a benchmark. The SPH technique was developed during the 1970s in the field of astrophysics [1] and has been applied to different engineering fields such as fluid dynamics. There, problems involving free-surface flows, gravity currents, breaking waves and wave impact on structures are some of the most relevant applications.

Two important aspects must be considered in a numerical method. The first one is the correct implementation of the physical governing equations and the accuracy of the mathematical algorithms. The second one is directly related to the nature of the hardware needed to execute the model. Each kind of platform (desktops, workstations, clusters…) used to perform numerical simulations presents its own advantages and limitations. Parallelization methods and optimization techniques are essential to perform simulations at a reasonable execution time. Therefore, in order to obtain the best performance, the code must be optimized and parallelized as much as possible according to the available hardware resources.

Two different hardware technologies can be employed; Central Processing Units (CPUs) and Graphics Processing Units (GPUs). Current CPUs have multiple processing cores, making possible the distribution of the workload of a program among the different cores and dividing the execution time. In addition, CPUs also present SIMD (Single Instruction, Multiple Data) which allows performing an operation on multiple data simultaneously. The parallelization task can be performed on CPUs by using MPI (Message Passing Interface) or OpenMP (Open Multi-Processing) as shown by [2] for large SPH simulations. In addition, parallel computing with multiple CPUs has also been used for SPH simulations in recent works (e.g. [3–6]). More recently, [7] presented a parallel SPH implementation on multi-core CPUs. On the other hand, research can also be conducted with the promising GPU technology for problems that previously required high performance computing (HPC). Initially, GPUs were developed exclusively for graphical purposes. However, the demands of the games and multimedia market forced the

* Corresponding author. Tel.: +34 988387255.
E-mail addresses: jmdominguez@uvigo.es (J.M. Domínguez), alexbexe@uvigo.es (A.J.C. Crespo), mggesteira@uvigo.es (M. Gómez-Gesteira).

0010-4655/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2012.10.015

rapid increase in performance that has led to parallel processors with computing power in floating-point much higher than the CPU ones. Moreover, the emergence of languages such as the Compute Unified Device Architecture (CUDA) facilitates the programming of GPUs for general purpose applications. Therefore, the GPGPU programming (General Purpose on Graphics Processing Units) has recently experienced a strong growth in all fields of scientific computing. Thus, GPU technology can accelerate numerical models reducing the computational cost. For example, [8] developed a general purpose molecular dynamics (MD) numerical code that runs entirely on a GPU. Another example is the work of [9] where the Boltzmann equation was solved on GPUs exhibiting speedups of up to two orders of magnitude over a single-core CPU. In the particular case of SPH, the first implementations of the method on GPUs were carried out by [10,11]. Following the release of CUDA, different SPH models have been implemented on GPUs during the last few years [12,13]. However programming GPUs (using CUDA or OpenCL) is neither straightforward nor applicable to any sort of algorithms, which is often one of the reasons for not choosing GPUs as the default processing unit.

SPHysics is a Smoothed Particle Hydrodynamics code mainly developed to deal with free-surface flow phenomena. The model has been jointly developed by the Johns Hopkins University (US), the University of Vigo (Spain) and the University of Manchester (UK). The SPHysics code can simulate complex fluid dynamics, including wave breaking, dam breaks, solid objects sliding into the water, wave impact on structures, etc. The first serial code was developed in FORTRAN (see [14] for a complete description of the code) showing its reliability and robustness for 2D [15–17] and 3D [18–20] problems. However, the main drawback of SPH models in general and of SPHysics code in particular is their high computational cost. Implementations that exploit the parallelism capabilities of the current hardware should be developed to carry out the simulation of realistic domains at a reasonable runtime. A new code named DualSPHysics [13] has been developed inspired by the formulation of the former FORTRAN SPHysics. The code, which can be freely downloaded from www.dual.sphysics.org, was implemented using both C++ and CUDA programming languages and can be executed both on CPUs and on GPUs.

It has been demonstrated that DualSPHysics achieves different speedups with different CUDA-enabled GPUs as shown in [13]. However, getting the most out of these parallel architectures is not trivial. This paper describes different strategies for CPU and GPU optimizations applied to DualSPHysics. Some of the GPU optimizations applied here to the SPH model present some of the well-known suggested basic optimizations described in the CUDA manuals [21], such as expose as much parallelism as possible, minimize CPU–GPU data transfers, optimize memory usage for maximum bandwidth, minimize code divergence, optimize memory access patterns, avoid non-coalesced accesses and maximize occupancy to hide memory latency. However other GPU optimizations intrinsic to the SPH method will be presented here analyzing their impact on the efficiency achieved with two different GPU architectures (Tesla and Fermi cards). GPU performance will also be compared to CPU-multi-cores. All comparisons are as fair as possible, since the most efficient implementations are adopted for each architecture. Furthermore, [22] remarks on the importance of optimizations which are independent of the parallelization. Therefore, different improvements will be shown before and after-parallelization. The reader is invited to use some of these optimization techniques and apply them to their own codes. Some optimizations are intrinsic to the SPH method since some of them are based on the Lagrangian nature of the meshfree methods.

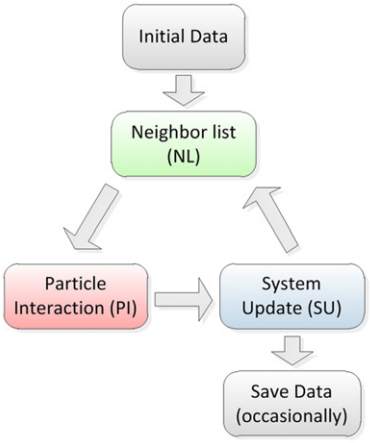


Fig. 1. Conceptual diagram summarizing the implementation of a SPH code.

2. SPH implementation

Smoothed Particle Hydrodynamics is a meshfree, Lagrangian, particle method for modeling fluid flows. A brief summary of the SPH formalism is presented in the Appendix. Since the fluid is treated as a set of particles, the hydrodynamic equations of motion are integrated on each particle. The conservation laws of continuum fluid dynamics written in the form of partial differential equations need to be transformed into the Lagrangian formalism (particles). Using an interpolation function that provides the weighted estimate of the field variables at a discrete point (particle), the interactions among fluid parcels are evaluated as sums over neighboring particles. Only particles located at a distance shorter than r_{cut} will interact. Thus, physical magnitudes (position, velocity, mass, density, pressure) are computed for each particle as an interpolation of the values of the nearest neighboring particles. The governing equations expressing conservation laws of continuum fluid dynamics are computed for pair-wise interactions of fluid particles; however a different formulation is needed to describe the interaction with boundary particles. Therefore, two different types of particles are considered; fluid particles and boundaries. A more complete description of the SPH method can be found in [23–25].

DualSPHysics is organized in three main stages that are repeated each time step: (i) creating a neighbor list; (ii) computing particle interactions for momentum and continuity conservation equations (Eqs. (A.4) and (A.5) respectively); and (iii) integrating in time to update all the physical properties of the particles in the system (Eq. (A.7)). A diagram of the SPH code can be seen in Fig. 1. The first stage is creating the neighbor list (NL). As we mentioned above, particles only interact with neighboring particles located at a distance less than r_{cut}. Thus, the domain is divided into cells of size ($r_{cut} \times r_{cut} \times r_{cut}$) to reduce the neighbor search to only the adjacent cells and the cell itself. The cell-linked list described in [26] was implemented in DualSPHysics. Another traditional method to perform a neighbor search is creating an array with all the real neighbors of each particle of the system (named a Verlet list), however the main problem of this approach is its higher memory requirements compared to the cell-linked list. In the present work, two different cell lists were created; the first one with fluid particles and the second one with boundary particles.

The second stage is the particle–particle interaction (PI), where each particle checks which particles contained in the adjacent cells and in its own cell are real neighbors (placed at a distance shorter than r_{cut}). Then, the equations of conservation of mass and momentum are computed for the pair-wise interaction of particles. The interaction boundary–boundary is not necessary. Thus, only

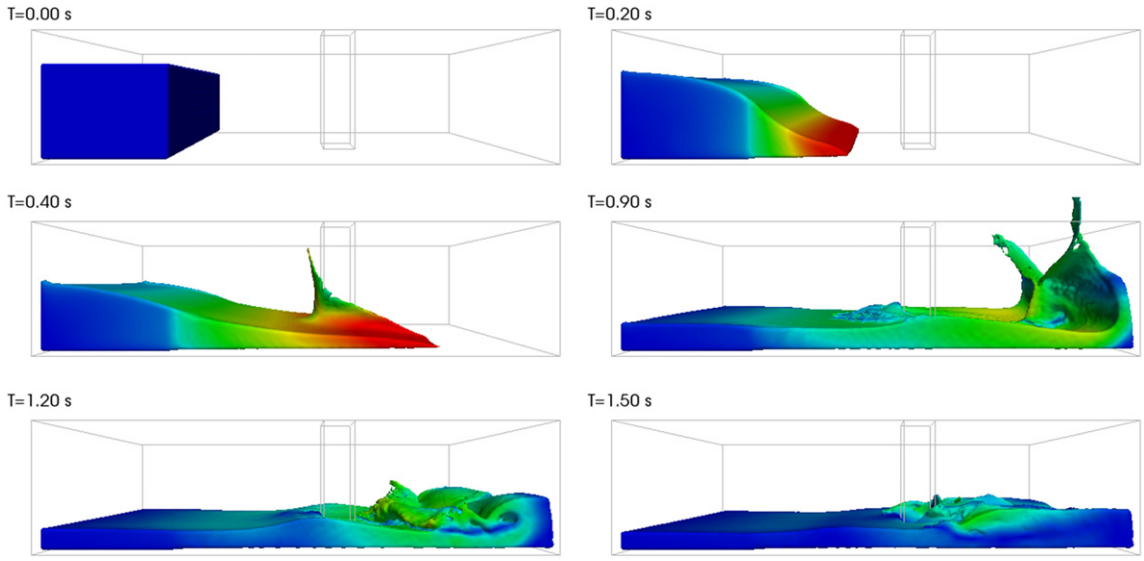


Fig. 2. Different instants of the dam break evolution.

Table 1
SPH formulation and references.

Time integration scheme	Verlet [28]
Time step	Variable [27]
Kernel function	Cubic spline kernel [29]
Viscosity treatment	Artificial ($\alpha = 0.25$) [30]
Equation of state	Tait equation [31]
Boundary condition	Dynamic boundaries [32]

fluid–fluid (F–F), fluid-boundary (F–B) and boundary–fluid (B–F) interactions should be carried out. Finally, the system is updated (SU) so, once the interaction forces between particles are calculated, the physical quantities are integrated in time at the next time step.

The testbed used in this work is the gravity induced collapse of a water column (see Fig. 2). Similar cases were simulated to validate the SPHysics [18] and the DualSPHysics [13] codes. This testcase will be used to analyze the performance of the optimizations and implementations presented in the following sections.

As mentioned above, the SPH method is expensive in terms of computational time. For example, a simulation of the dam break evolution during 1.5 s of physical time using 300,000 particles takes more than 15 h on a single-core machine (before the optimizations described in this paper). The first limitation is the small time step (10^{-6}–10^{-5} s) imposed by forces and velocities [27]. Thus, in this case, more than 16,000 steps are needed to complete the 1.5 s of physical time. On the other hand, each particle interacts with more than 250 neighbors, which implies a large number of interactions (operations) in comparison with the methods based on a mesh (Eulerian methods) where only a few grid nodes are taken into account. In this case, the particle interaction takes 99% of the total computational time when executed on a single-core CPU and more than 95% on a GPU. Thus, all the efforts to increase the performance of the code must be focused on reducing the execution time of the particle interaction stage.

Finally, we should mention that SPHysics [14,25] allows numerous parameterizations to compute boundary conditions, viscosity, time stepping, etc. In this particular study, the chosen options used in DualSPHysics are summarized in Table 1.

3. Strategies for CPU optimization

Some features intrinsically linked to the Lagrangian nature of SPH models should be described before going into details about optimization strategies. The physical variables corresponding to each particle (position, velocity, density…) are stored in arrays. During the NL stage, the cell to which each particle belongs is determined. This makes it possible to reorder the particles (and the arrays with particle data) following the order of the cells. Thus, if particle data are closer in the memory space, the access pattern is more regular and efficient. Another advantage is the ease of identifying the particles that belong to a cell by using a range since the first particle of each cell is known. In this way, the interaction between particles is carried out in terms of the interaction between cells. All the particles inside a cell interact with all the particles located in the same cell and in adjacent cells. Force computations between two particles will be carried out when they are closer than the interaction range (r_{cut}).

First of all, some standard and well-known CPU optimization techniques have been applied to DualSPHyiscs:

- Applying symmetry to particle–particle interaction (Newton's third law) since $\nabla_a W_{ab} = -\nabla_b W_{ba}$ in Eqs. (A.4) and (A.5).
- Splitting the domain into smaller cells to diminish the number of false neighbors which are the particles located at a distance larger than the kernel support r_{cut}, being $r_{cut} = 2h$ in Eq. (A.3).
- Using special instruction sets of SIMD type that allow us to perform operations on data sets.

The main CPU optimization described in this work is the implementation of a multi-core programming with OpenMP. Current CPUs have several cores or processing units, so it is essential to distribute the computation load among them to maximize the CPU performance and to accelerate the SPH code. As mentioned above, there are two main options to implement a parallel code in CPU, namely MPI and OpenMP. MPI is particularly suitable to distribute memory systems where each processing unit only has access to a portion of the system memory and the different processes need to exchange data by passing messages. However the architecture used in this work uses a shared memory system, where each process can directly access all memory without the extra cost of the message passing in MPI. OpenMP is a portable and flexible programming interface whose implementation does not involve major changes in the code. Using OpenMP, multiple threads for a process can be easily created. These threads are distributed among all the cores of the CPU sharing the memory. Thus, there is no need to duplicate data or to transfer information among threads. Due to these reasons, OpenMP was used in DualSPHysics.

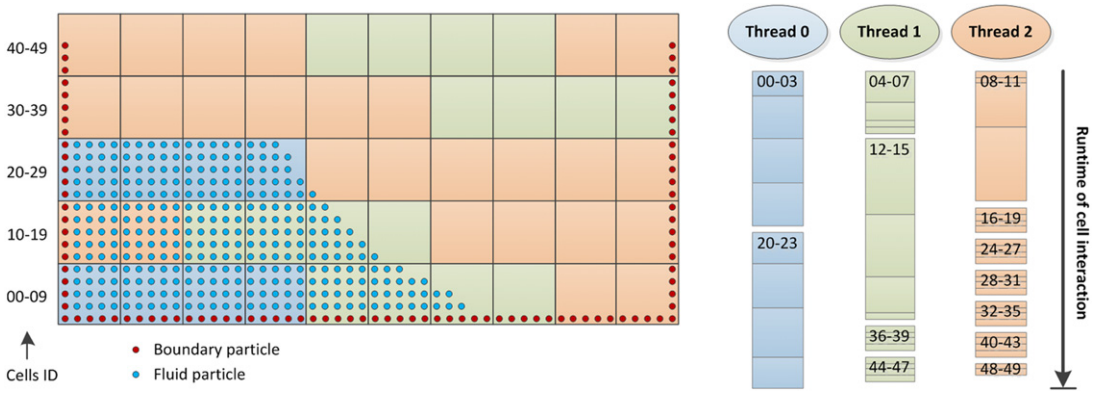


Fig. 3. Example of dynamic distribution of cells (in blocks of 4) among 3 execution threads according to the execution time of each cell.

Several parts of the SPH code can be parallelized, which is especially important for force calculation that is the most expensive part of the code. The minimum execution unit of each thread is the cell, so that all particles of the same cell are processed sequentially. Neighboring particles are searched in the surrounding cells and the particle interaction is computed. However, it is not straightforward to apply symmetry to particle–particle interactions when several execution threads of a CPU are used in parallel since the concurrent access to the same memory positions for read–write particle forces can give rise to unexpected results. In addition, special attention should be paid to the load balancing to distribute equally the work among threads. Therefore, three different approaches were proposed to avoid concurrent accesses and obtain load balancing:

Asymmetric: Concurrent access occurs in force computation when applying symmetry, since the thread that computes the summation of the forces on a given particle also computes the forces on the particles placed in the neighborhood of the first one. Nevertheless, these neighboring particles may be simultaneously processed by another thread. To avoid this conflict, symmetry is not applied in a first approach. The load balancing is achieved by using the dynamic scheduler of OpenMP. Cells can be assigned (usually in blocks of 10) to the threads as they run out of workload. Fig. 3 shows an example of the dynamic distribution of cells (in blocks of 4) among 3 execution threads according to the execution time of each cell, which depends on the number of neighboring particles. The main advantage is the ease of implementation, the main drawback being the loss of symmetry.

Symmetric: In this approach, the dynamic scheduler of OpenMP is also employed distributing cells in blocks of 10 among different threads. The difference with the previous case lies in the use of the symmetry in the computation of the particle–particle interaction. Now the concurrent memory access is avoided since each thread has its own memory space to allocate variables where the forces on each particle are accumulated. Thus, the final value of the interaction force for each particle is obtained by combining the results once all threads have finished. This final value is also computed by using multiple threads. The advantage of this approach is the use of the symmetry in all the interactions and the easy implementation of the load dynamic balancing. The main drawback is the increase in memory requirements, which depends on the number of threads. Note that memory duplication for each thread is efficient when using a system with a small number of threads (as in the hardware available in this work), but it does not scale on a system with much wider CPUs which can execute many more threads. For example, the memory requirement increases by a factor of 2 when passing from 1 to 8 threads in the testcase.

Slices: The domain is split into slices, so that the number of slices is the number of available execution threads. Symmetry is applied to the interactions among cells that belong to the same slice, but not to the interactions with cells from other slices. Thus, symmetry is used in most of the interactions (depending on the width of the slices). The thickness of the slices is adjusted to distribute the runtime of the particle interactions within each slice (dynamic load balancing). The division is periodically updated to keep the slices as balanced as possible. This thickness is adjusted according to the computation time required for each slice during the last time steps, which allows for a more correct dynamic load balancing. The main drawbacks are the higher complexity of the code and the higher runtime associated to the dynamic load balancing.

4. Strategies for GPU optimization

As mentioned in Section 2, the execution of particle interactions for the present testcase takes 99% of the total runtime in a single-core CPU. For the GPU implementation, we started from a hybrid implementation where only the PI stage was implemented on the GPU (partial GPU implementation) as shown in the left panel of Fig. 4. The implementation of the particle interactions on GPUs is different from the CPU one. Instead of computing the interaction of all particles of one cell with the neighboring cells as implemented on CPUs, now each particle looks for all its neighbors sweeping all the adjacent cells. Thus, particle interactions can be implemented on the GPU for only one particle using one execution CUDA thread to compute the force resulting from the interaction with all its neighbors. Using this approach, the symmetry of the force computation cannot be applied since several threads could be trying to modify simultaneously the same memory position giving rise to an error. This problem could be avoided by using synchronization barriers, but this approach would be inefficient due to its high computational cost.

To minimize the CPU–GPU transfers, the SPH code is implemented on GPU keeping data in the GPU memory (see right panel of Fig. 4). Therefore, the CPU–GPU communications are drastically reduced and only some particular magnitudes like position, velocity or density will be recovered from GPUs at some time steps. Moreover, when the other main stages of the SPH method (neighbor list and system update) are implemented on the GPU, the computational time devoted to those processes decreases.

As previously described for CPUs, particle data (position, velocity…) are stored in arrays but, in this case, the arrays are stored on the GPU memory. During the neighbor list stage on GPU, the cells where the particles are located are calculated first. Then, the implementation of the *radixsort* algorithm from NVIDIA [33] is used to obtain the position of the particles when they are reordered according to cells, so all the arrays with particle data are reordered. Finally, a new array is created to determine which particles belong to a given cell. Thus, the array *CellBeginEnd* contains the first and the last particle of every cell. Fig. 5 shows an example of how this

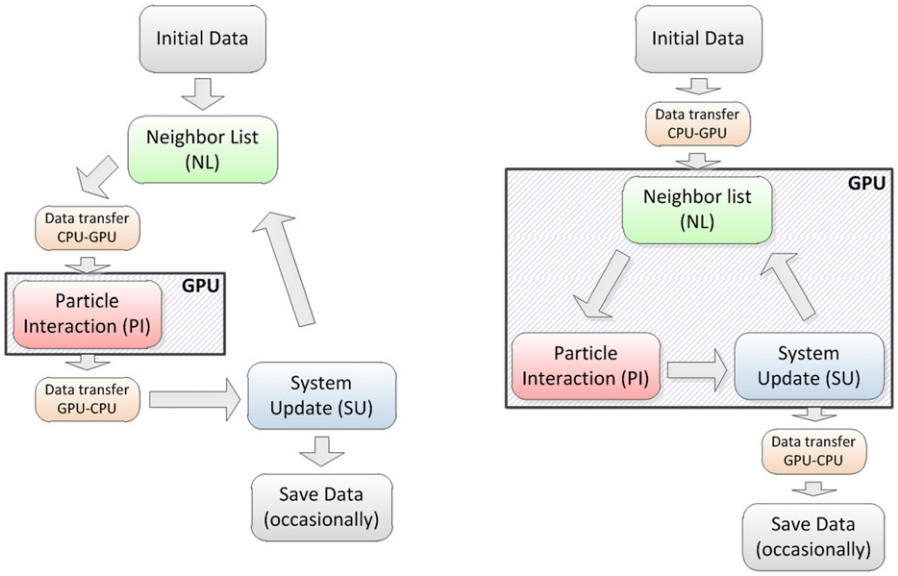


Fig. 4. Conceptual diagram of the partial (left) and full (right) GPU implementation of the SPH code.

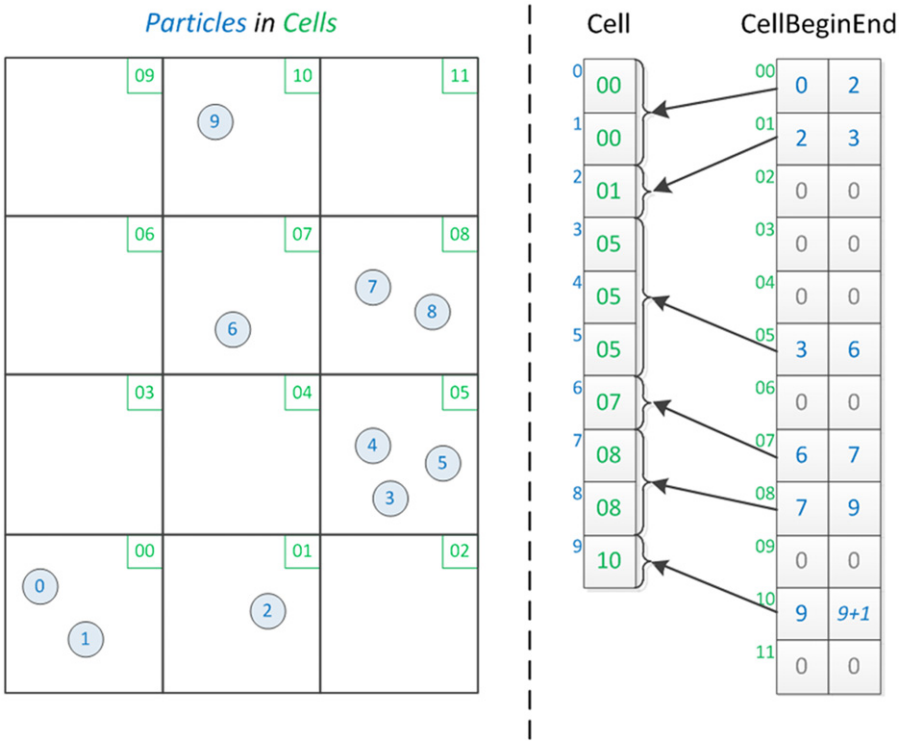


Fig. 5. Example of how *CellBeginEnd* is created starting from the particle distribution in cells depicted in the left panel.

array is created starting from the particle distribution depicted in the left panel of the figure. The method to create *CellBeginEnd*, which is described in [34], has proven to be efficient even for empty cells.

The system update can be easily parallelized on a GPU since only the physical properties of the particles must be calculated at the next time step (Eq. (A.7)). The major arduousness is computing the maximum and minimum values of some variables (force, velocity and sound speed) needed to estimate the value of the variable time step (Eq. (A.8)). This calculation is optimized using a reduction algorithm for GPUs based on [35]. This algorithm allows us to obtain the maximum or minimum values of a huge data set taking advantage of the parallel programming on GPUs.

The main difference between the full GPU implementation presented here and the work of [11] is that they implemented a classical SPH approach on GPU before the appearance of CUDA in 2007 using shader programs written in C for Graphics. In this work, the full GPU implementation is performed using the parallel programming CUDA as described below. CUDA is more independent of the particular hardware. This allows the code to be run on new forthcoming GPU cards more efficiently. On the other hand, CUDA makes easy the maintenance and the updating of the code when including more complex algorithms and new SPH formulations. The work by [8] also developed a MD code entirely executed on the GPU but implementing a different approach for neighbor list, giving rise to different efficiencies in terms of

performance and memory requirements. They implemented the Verlet list, so the number of particles that can be simulated in the memory space of one GPU card is much smaller than the number of particles presented here.

This implementation presents different problems to be solved:

- *Code divergence*: GPU threads are grouped into sets of 32 named *warps* in the CUDA language. When a task is being executed over a warp, the 32 threads carry out this task simultaneously. However, due to conditional flow instructions in the code, not all the threads will perform the same operation, so the different tasks are executed sequentially, giving rise to a significant loss of efficiency. This divergence problem appears during particle interaction since each thread has to evaluate which potential neighbors are real neighbors before computing the force.
- *No coalescent memory accesses*: The global memory of the GPU is accessed in blocks of 32, 64 or 128 bytes, so the number of accesses to satisfy a warp depends on how grouped the data are. A regular memory access is not possible in particle interactions since each particle has different neighbors and, therefore, each thread will access different memory positions which may eventually be far from the rest of the positions in the warp.
- *No balanced workload*: Warps are executed in *blocks* in the CUDA terminology. When a block is going to be executed, some resources are assigned and they will not be available for other blocks till the end of the execution. So, since each thread may have a different number of neighbors, one thread may need to perform more interactions than the rest. Thus, the warp can be under execution while the rest of the threads of the same warp, or even of the block, can have finished. Thus, the performance is reduced due to the inefficient use of the GPU resource.

Several optimizations have been developed to avoid or minimize the problems previously described. First of all, *maximizing the occupancy of GPU* and *reducing global memory accesses* are some of the well-known basic optimizations described in the CUDA manuals [21] which must be always considered when porting a code to GPU. Then, more GPU optimizations intrinsic to the SPH method such as *simplifying the neighbor search, adding a more specific CUDA kernel of interaction* and *the division of the domain into smaller cells* will be described

Maximizing the occupancy of GPU. Occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU or Streaming Multiprocessor (SM). Since the access to the GPU global memory is irregular during the particle interaction, it is essential to have the largest number of active warps in order to hide the latencies of memory access and maintain the hardware as busy as possible. The number of active warps depends on the registers required for the CUDA kernel, the GPU specifications (see Table 2) and the number of threads per block. The first option could be reducing the number of registers per thread, however this implies the increase of memory accesses and the number of computations in the interaction kernel. Another option is adjusting the block size in an automatic way according to the registers of the kernel and the hardware specifications. Fig. 6 shows the obtained occupancy for different numbers of registers and for different computational capabilities of the GPU card when using 256 threads and using other block sizes. For example, the occupancy of a GPU sm13 (compilation with compute capability 1.3) for 35 registers is 25% (dashed blue line) using 256 threads, but it can be 44% (solid blue line) using 448 threads.

Reducing global memory accesses. When computing the SPH forces during the PI stage, the six arrays described in Table 3 are used. The arrays *csound, prrhop* and *tensil* were previously calculated for each particle using *rhop* to avoid calculating them for each interaction of the particle with all its neighbors. The number of memory accesses in the interaction kernel can be reduced by grouping part of these arrays (*pos*+*press* and *vel*+*rhop* are combined to create two arrays of 16 bytes each) and avoiding reading values that can be calculated from other variables (*csound* and *tensil* are calculated from *press*). Thus, the number of accesses to the global memory of the GPU is reduced from 6 to 2 and the volume of data to be read from 40 to 32 bytes.

Table 2
Technical specifications of GPUs according to the compute capability.

Technical specifications	1.0	1.1	1.2	1.3	2.×
Max. of threads per block	512				1024
Max. of resident blocks per SM	8				
Max. of resident warps per SM	24		32		48
Max. of resident threads per SM	768		1024		1536
Max. of 32-bit registers per SM (K)	8		16		32

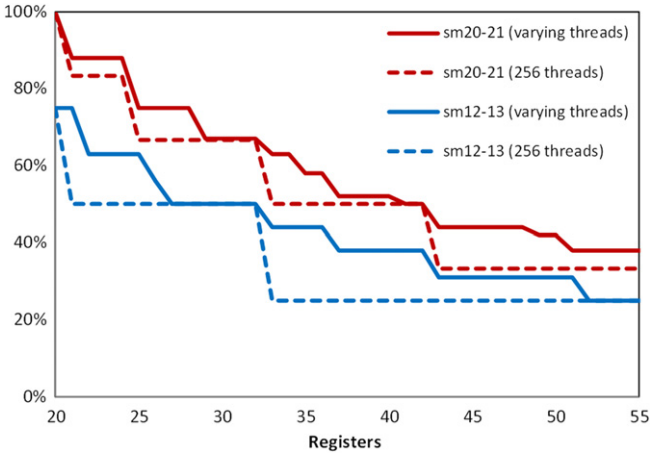


Fig. 6. Occupancy of the GPU for different numbers of registers with a variable and a fixed block size of 256 threads.

Table 3
List of variables needed to calculate forces.

Variable	Size (bytes)	Description
pos	3×4	Position in X, Y and Z
vel	3×4	Velocity in X, Y and Z
rhop	4	Density
csound	4	Speed of sound
prrhop	4	Ratio between pressure and density
tensil	4	Tensile correction following [36]

Simplifying the neighbor search. During the GPU execution of the interaction kernel, each thread has to look for the neighbors of its particle sweeping through the particles that belong to its own cell and to the surrounding cells, a total of 27 cells since symmetry cannot be applied. However, this procedure can be optimized when simplifying the neighbor search. This process can be removed from the interaction kernel when the range of particles that could interact with the target particle is previously known. Since particles are reordered according to the cells and cells follow the order of X, Y and Z axes, the range of particles of three consecutive cells in the X-axis ($\text{cell}_{x,y,z}$, $\text{cell}_{x+1,y,z}$, $\text{cell}_{x+2,y,z}$) is equal to the range from the first particle of $\text{cell}_{x,y,z}$ to the last of $\text{cell}_{x+2,y,z}$. Thus, the 27 cells can be defined as 9 ranges of particles. The 9 ranges are colored in Fig. 7. The interaction kernel is significantly simplified, when these ranges are known in advance. Thus, the memory accesses decrease and the number of divergent warps is reduced. In addition, GPU occupancy increases since less registers are employed in the kernel. The main drawback is the higher memory requirements due to the extra 144 bytes needed per cell.

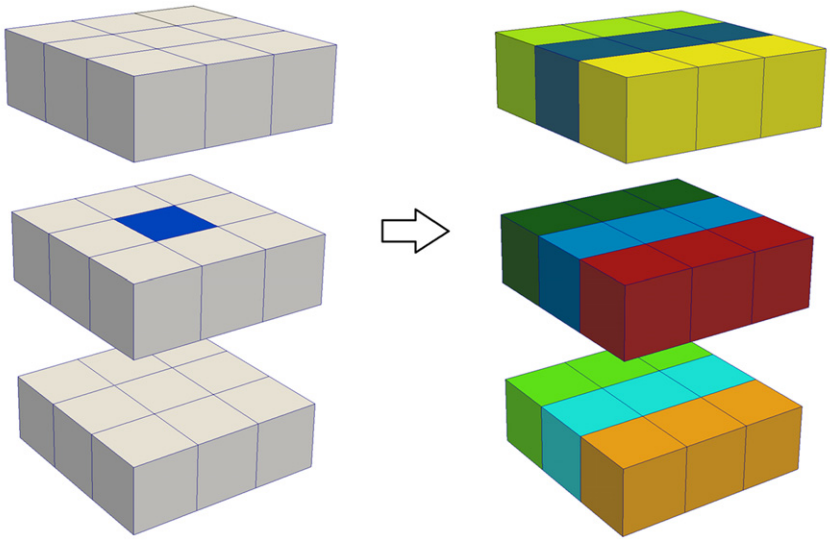


Fig. 7. Interaction cells in 3D without symmetry but using 9 ranges of three consecutive cells (right) instead of 27 cells (left).

Adding a more specific CUDA kernel of interaction. Initially, the same CUDA kernel was used to calculate all interaction forces B–F, F–B and F–F. However, symmetry in the force computation cannot be efficiently applied and the best option is implementing a specific kernel for the B–F interaction (boundary–fluid) because only a subset of the fluid particles is required to be computed for the boundaries. The effect of this optimization on the overall performance is negligible when the number of boundary particles is small in comparison with the number of fluid ones. On the other hand, the access to the global memory of the GPU is two orders of magnitude slower than the access to other registers. In order to minimize these accesses, each thread starts storing all its particle data in registers, so the thread only needs to read data corresponding to the neighbor particles. The same approach is applied to store the forces, which are accumulated in registers and written in global memory at the end. As described before, there are two types of particles (boundaries and fluids), so there are three interactions to calculate all the forces (F–F, F–B and B–F). Therefore, data of the fluid particles associated to the threads are read twice (when fluid particles interact with other fluid particles and when they interact with boundaries) and the same occurs when writing results in the global memory. A way to avoid this problem is carrying out the interaction F–F and F–B in the same CUDA kernel with a single data load and a single final writing instead of two.

Division of the domain into smaller cells. As mentioned in the optimization applied in the CPU implementation, the procedure consists in dividing the domain into cells of size $r_{cut}/2$ instead of size r_{cut} in order to increase the percentage of real neighbors. Using cells of size r_{cut} on the GPU implementation, the number of pair-wise interactions decreases. The disadvantage is the increase in memory requirements since the number of cells is 8 times higher and the number of ranges of particles to be evaluated in the neighbor search increases from 9 to 25 (using 400 bytes per cell).

5. Results

The DualSPHysics code will be used to run the testcase described above (see Fig. 2). In this section, the results of applying all the optimizations developed in this work are presented. Thus, the relative performance of the CPU and GPU implementations at different levels of optimization is studied. In addition the impact of the different GPU optimizations is studied for two different GPU architectures (Tesla and Fermi).

The system used for the CPU performance testing:

- *Hardware*: Intel® Core ™ i7 940 at 2.93 GHz (4 physical cores, 8 logical cores with Hyper-threading, with 6 GB of 1333 MHz DDR3 RAM)
- *Operating system*: Ubuntu 10.10 64-bit
- *Compiler*: GCC 4.4.5 (compiling with the option—O3).

The system used for the GPU performance testing:

- *Hardware* 1: NVIDA GTX 480 (15 multiprocessors, 480 cores at 1.37 GHz with 1.5 GB of 1848 MHz GDDR5 RAM and compute capability 2.0)
- *Hardware* 2: NVIDA Tesla 1060 (30 multiprocessors, 240 cores at 1.3 GHz with 4 GB of 1600 MHz GDDR3 RAM and compute capability 1.3)
- *Operating system*: Debian GNU/Linux 5.0 (Lenny) 64-bit.
- *Compiler*: CUDA 3.2 (compiling with the option—use_fast_math).

Both CPU and GPU results show the speedup obtained when comparing the performance of the different implementations which was measured as the number of time steps computed per second. Note that all the results presented here were obtained using single precision. A study using double precision can also be carried out since the latest CUDA-enabled GPU cards present improved support for double precision.

The speedup obtained with the multi-core implementation on CPU of the SPH code for different numbers of particles is observed in Fig. 8. In the figure, the performance of the different OpenMP implementations (using 8 threads) is compared with the most efficient single-core version (that includes symmetry, SIMD instructions and cell size equal to $r_{cut}/2$). The most (less) efficient implementation is *Symmetric* (*Asymmetric*). A speedup of 4.5× is obtained with *Symmetric* when using 8 threads. The approaches that divide the domain into slices (*Slices*) offer a higher performance when increasing the number of particles since the number of cells also increases, allowing a better distribution of the workload among the 8 execution threads. Using *Slices*, the efficiency does not depend on the direction of fluid movement. Similar performance is achieved when creating the slices in the *X* or *Y*-direction, since the workload is distributed equally among the slices.

Table 4 shows the computational times and speedups on CPUs using the most efficient version of OpenMP (*Symmetric*) with 4 and 8 threads compared to the single-core CPU version. Note that the evaluation of the speedup is not expected to be linear with the number of threads since the available CPU hardware is the

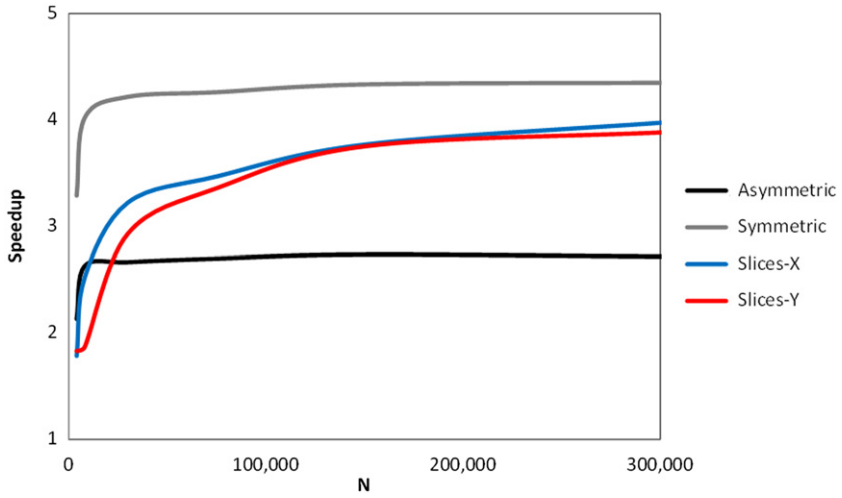


Fig. 8. Speedup achieved on CPU for different numbers of particles (N) with different OpenMP implementations (using 8 logical threads) in comparison with the most efficient single-core version that includes all the previous optimizations.

Table 4
Speedup achieved on CPU simulating 300,000 particles when using 4 and 8 threads compared to the single CPU version.

Version	Total simulation time (s)	Number of steps	Computed steps per second	Speedup vs. CPU single-core
CPU single-core	24,520	16,282	0.66	*1.0*×
CPU 4 threads	6,375	16,275	2.55	3.9×
CPU 8 threads	5,414	16,284	3.01	4.6×

Table 5
Improvement achieved on GPU simulating 1 million particles when applying the different GPU optimizations using GTX 480 and Tesla 1060.

	GTX 480		Tesla 1060	
	Optimization (%)	Cumulative (%)	Optimization (%)	Cumulative (%)
Maximizing the occupancy of GPU	7.3	7.3	17.4	17.4
Reducing global memory accesses	18.9	27.6	28.9	51.3
Simplifying the neighbor search	3.1	31.5	12.9	70.8
Specific CUDA kernel of interaction	2.6	34.9	11.3	90.1
Division of the domain into smaller cells	22.7	65.4	12.8	114.5

Intel® Core ™ i7 with 4 physical cores and 8 logical cores with Hyper-threading and Table 4 shows results using logical cores instead of physical ones. Therefore, the parallel CPU version with 8 threads is 4.6 times faster than the single-core version and the speedup is 3.9 using 4 threads.

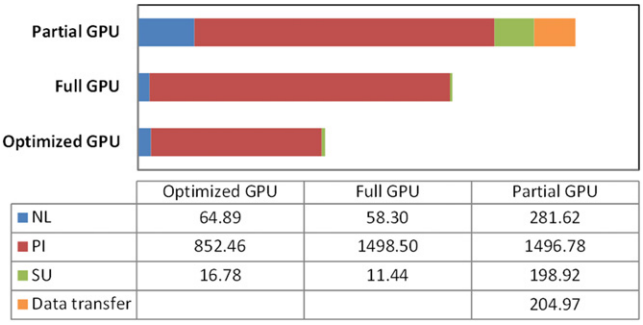
Table 5 summarizes the improvement achieved on the GPU cards GTX 480 and Tesla 1060 when using the different optimization strategies described in Section 4. All results were obtained simulating the same testcase with 1 million particles. Two variables are shown: the percentage of improvement obtained when applying each individual optimization and the cumulative improvement achieved when including the present and the previous optimizations. Also there can be observed the effect of optimizations on both GPU architectures; Tesla 1060 corresponds to the generation of GPUs with 240 cores and with compute capability 1.3 (see Table 2) and GTX 480 corresponds to the Fermi architecture with 480 cores and with compute capability 2.0 (see Table 2). In fact, this different behavior of both GPU cards is related not only to the compute capability and the number of cores but also to the number of registers and some kind of cache memory available in the Fermi GPUs that reduces conflicts when accessing to the global memory. For example, *maximizing the occupancy of GPU* presents a better improvement with the Tesla card than with the GTX. Due to the lower occupancy provided by the compute capability sm13 in comparison to sm20, the margin of improvement is higher for the Tesla card (see Fig. 6). In contrast, the impact of *dividing the domain into smaller cells* is more important with the GTX. The divergence diminishes when using smaller cells but the irregular accesses to memory increase and the GTX card presents that kind of cache memory that helps to mitigate the negative effect of the irregular accesses while the Tesla cannot. Considering the cumulative response of applying all the optimizations, the fully optimized GPU code for the GTX 480 is 1.65 times faster than the basic GPU version without optimizations and, in the case of Tesla 1060, the achieved speedup was 2.15.

The full implementation of the SPH code on GPU is basic since when neighbor list (NL), particle interaction (PI) and system update (SU) are implemented on a GPU, the CPU–GPU data transfer is avoided in each time step. Fig. 9 shows the computational runtimes using the GTX 480 for different GPU implementations (partial, full and optimized) simulating 500,000 particles of the testcase. *Partial GPU* implementation corresponds to a preliminary version where only the PI stage was implemented on GPU, in the *full GPU* version the three stages of the SPH code are executed on GPU and *optimized GPU* is the final version including all the proposed optimizations described in Section 4. It can be observed that the time dedicated to the CPU–GPU data transfer in the partial implementation is 9.4% of the total runtime. The CPU–GPU communications are not necessary at each time step when the SPH code is totally implemented on GPU. The runtimes of the NL and SU stages decrease when both parts of the code are also implemented on GPU. Finally, the computational time of the PI stage is reduced to about 40% when applying all the developed optimization strategies.

The comparison between CPU and GPU can be observed in Table 6. The table summarizes the execution runtimes, the number of computed steps and the achieved speedups. Note that the

Table 6
Results of the CPU and GPU simulations.

Version	Number of particles	Total simulation time (h)	Number of steps	Computed steps per second	Speedup vs. CPU single-core	Speedup vs. CPU 8 threads
CPU single-core	503,492 1,011,354	14.6 40.7	19,855 26,493	0.4 0.2	*1.0*× *1.0*×	– –
CPU 8 threads	503,492 1,011,354	3.2 9.1	19,806 26,511	1.7 0.8	4.6× **4.5×**	*1.0*× *1.0*×
GPU Tesla 1060	503,492 1,011,354	0.5 1.5	19,832 26,509	10.2 4.9	26.8× **27.3×**	5.8× **6.1×**
GPU GTX 480	503,492 1,011,354	0.3 0.7	19,830 26,480	21.2 10.1	55.7× **56.2×**	12.2× **12.5×**



	Optimized GPU	Full GPU	Partial GPU
NL	64.89	58.30	281.62
PI	852.46	1498.50	1496.78
SU	16.78	11.44	198.92
Data transfer			204.97

Fig. 9. Computational runtimes (in seconds) using GTX 480 for different GPU implementations (partial, full and optimized) when simulating 500,000 particles.

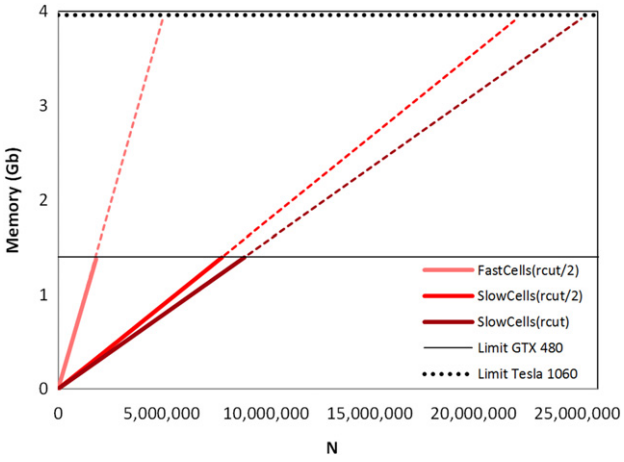


Fig. 10. Memory usage for different GPU versions implemented in DualSPHysics.

speedup has been measured here as the ratio between the number of time steps computed per second by the different versions. The data correspond to the most efficient implementation on GPU versus the multi-core implementation on CPU (*symmetric* with 8 threads) and the single-core implementation. Thus, for example, for one million particles, the performance of the CPU is 0.2 time steps per second using the single-core version and 0.8 using the multi-core version, while 10.1 time steps per second can be computed with a GPU GTX 480. The whole simulation takes one day, 16 h and 45 min on the Intel® Core ™ i7 and only 42 min on the GTX 480, resulting in a speedup of 56.2× (vs. single-core CPU) and 12.5× (vs. CPU with 8 logical threads). It can be also observed that the speedups with GTX 480 (Fermi technology) are twice those obtained with Tesla 1060, which belongs to a previous generation of GPU cards as mentioned above. Note that, usually, the works about parallel hardware to accelerate SPH published before the appearance of GPUs showed speed-ups considering CPU clusters versus a single core. When proving the capability of GPU computations for engineering applications, relative runtimes can be useful, so the speedup in comparison with a single CPU core is also shown to give an idea of the order of speedup that is possible when using GPU cards instead of large cluster machines.

The fastest GPU implementation uses all the GPU optimizations including *dividing the domain into smaller cells*, whose main disadvantage is the increase in memory requirements as mentioned above. Therefore, the maximum number of particles that can be simulated in a GTX 480 using the optimized GPU version of DualSPHysics is only 1.8 million. Accordingly, three different versions of the code are implemented to avoid this limitation. These different GPU versions are available in DualSPHysics and the fastest one is automatically selected by the code depending on the memory requirements of the simulation. The first version contains all the GPU optimizations and it is named *FastCells*($r_{cut}/2$), the second one, named *SlowCells*($r_{cut}/2$), is implemented without the optimization of *simplifying the neighbor search* and the third version, named *SlowCells*(r_{cut}), is implemented without *simplifying the neighbor search* and without *dividing the domain into smaller cells*. The memory usage for these three different GPU versions can be seen in Fig. 10. Note the black solid line represents the limit of memory that can be allocated on a GTX 480 (less than 1.4 GB) and the dotted line is the limit for the Tesla 1060 (less than 4 GB). Using all these different versions, which will be automatically selected for each run depending on memory requirements, DualSPHysics allows simulating up to 9 million particles with a GTX 480 and more than 25 million with a Tesla 1060. The execution times corresponding to the three GPU versions (*FastCells*($r_{cut}/2$), *SlowCells*($r_{cut}/2$) and *SlowCells*(r_{cut})) and the times of the single-core and multi-core CPU versions are also summarized in Fig. 11.

6. Conclusions and future work

A code based on the SPH technique has been developed to deal with free-surface flow problems. SPH is a particle meshless method with the benefits and problems inherent to its Lagrangian nature. A CPU–GPU solver named DualSPHysics is used to simulate a dam break flow impacting on a structure. Different strategies for CPU and GPU optimizations have been developed to speed up the results. The main advantages and disadvantages of each optimization are addressed.

Four optimizations are implemented for the CPU code. The first one applies symmetry in particle interactions, the second one divides the domain into smaller cells, the third one uses SSE instructions and the fourth one uses OpenMP to implement multi-core executions. Three different approaches of the multi-core implementations are presented. The most efficient version uses the dynamic scheduler of OpenMP to achieve the load dynamic balancing and applies symmetry to particle interactions. Thus, the most efficient OpenMP implementation outperforms the single-core by 4.5× using the available 8 logical cores provided by the CPU hardware used in this study.

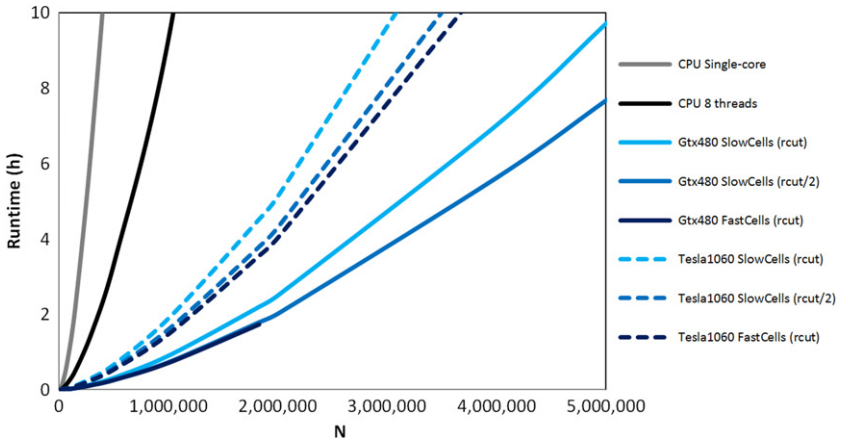


Fig. 11. Runtimes for different CPU and GPU implementations.

Several optimizations are presented for the GPU implementations; maximization of occupancy to hide memory latency, reduction of global memory accesses to avoid non-coalesced memory accesses, simplification of the neighbor search, optimization of the interaction kernel and division of the domain into smaller cells to reduce code divergence. The optimized GPU version of the code outperforms the GPU implementation without optimizations by a factor on the order of 1.65 using a GTX 480 (Fermi architecture) and 2.15 using a Tesla 1060. In general, the design improvements included in the new Fermi GPUs make these cards less sensitive to the programming task.

The GPU parallel computing developed here can accelerate serial SPH codes with a speedup of 56.2× when using the Fermi card. Finally, the speedup of the GPU over a multi-core CPU is 12.5 when using a multi-threaded approach.

Further work is being conducted to implement these codes on multi-core GPU clusters combining hybrid CUDA, OpenMP and MPI. This parallel programming approach has been successfully performed with typical problems such as matrix multiplications or sorting algorithms in [37].

Acknowledgments

This work was supported by Xunta de Galicia under project Programa de Consolidación e Estructuración de Unidades de Investigación Competitivas (Grupos de Referencia Competitiva), financed by the European Regional Development Fund (FEDER) and also funded by the Ministerio de Economía y Competitividad under the Project BIA2012-38676-C03-03. We want to acknowledge Orlando Garcia Feal and Diego Perez Montes for their technical help.

Appendix. SPH formulation

In SPH, the fundamental principle is to approximate any function $A(\mathbf{r})$ by

$$A(\mathbf{r}) = \int A(\mathbf{r}')W(\mathbf{r}-\mathbf{r}', h)d\mathbf{r}' \tag{A.1}$$

where h is the smoothing length and W is the weighting function or kernel.

This expression, in discrete notation, leads to an approximation of the function at a particle (interpolation point) a, where the summation is over all the particles b within the region of compact support of the kernel function:

$$A(\mathbf{r}_a) \approx \sum_b \frac{m_b}{\rho_b} A(\mathbf{r}_b)W(\mathbf{r}_a-\mathbf{r}_b, h) \tag{A.2}$$

where m and ρ are the mass and density of the particles.

Thus, the smoothing length defines the area of influence of the kernel at which the contribution with the rest of the particles can be neglected. These smoothing kernel functions must fulfill several properties [30], such as positivity inside a defined zone of interaction, compact support, normalization and monotonically decreasing with distance. Kernels depend on the smoothing length and the non-dimensional distance between particles given by $q = r/h$, r being the distance between particles a and b. The cubic spline kernel has been one of the most widely used smoothing function in the SPH literature since it resembles a Gaussian function while having a narrower compact support and presents a compact support (it is equal to zero for $q > 2$ or $r > 2h$):

$$W(q) = \alpha_D \begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & 0 \le q \le 1 \\ \frac{1}{4}(2-q)^3 & 1 \le q \le 2 \\ 0 & q \ge 2 \end{cases} \tag{A.3}$$

$\alpha_D = 1/(\pi\, h^3)$ being the normalization constant in 3D. Here, the tensile correction [36] is activated when using kernels whose first derivative goes to zero with q. Note that the r_{cut} defined in the text will be $2h$ here.

The conservation of momentum is solved using the equation proposed by [30] where the force exerted onto the particle (a) as the result of the particle interactions with all its neighbors (particles b) is computed:

$$\frac{d\mathbf{v}_a}{dt} = -\sum_b m_b \left(\frac{P_b}{\rho_b^2} + \frac{P_a}{\rho_a^2} + \Pi_{ab} \right) \nabla_a W_{ab} + \mathbf{g} \tag{A.4}$$

$\mathbf{v}$ being velocity, P pressure, ρ density, m mass, $\mathbf{g}$ the gravitational acceleration and W_{ab} the kernel function that depends on the distance between particles a and b. Π_{ab} is the viscous term according to the artificial viscosity in [30] that depends on the free parameter α (chosen here to be 0.25).

The equation of the conservation of mass is used to determine the changes in fluid density assuming that the mass of each particle is constant. Thus, the continuity equation in SPH form [30] is:

$$\frac{d\rho_a}{dt} = \sum_b m_b \mathbf{v}_{ab} \cdot \nabla_a W_{ab}. \tag{A.5}$$

Following Tait's equation of state [31], the relationship between pressure and density is assumed to follow the expression:

$$P_a = B\left[\left(\frac{\rho_a}{\rho_0} \right)^7 - 1 \right] \tag{A.6}$$

where the reference density of the fluid is $\rho_0 = 1000\ \mathrm{kg\ m^{-3}}$ the reference density, the parameter $B = c_0^2\rho_0/\gamma$ and the speed of sound at the reference density $c_o = c(\rho_o) = \sqrt{(\partial P/\partial\rho)}|_{\rho_o}$.

There are several ways to develop the solution of the SPH equations in time. It is advisable to use at least a second-order accurate scheme in time since the particles represent computation points moving according to the governing dynamics. In the present work, a Verlet [28] algorithm is used. In general, variables are calculated following:

$$\begin{aligned}
\mathbf{v}_a^{n+1} &= \mathbf{v}_a^{n-1} + 2\Delta t\frac{d\mathbf{v}_a^n}{dt};\\
\mathbf{r}_a^{n+1} &= \mathbf{r}_a^n + \Delta t\frac{d\mathbf{r}_a^n}{dt} + 0.5\Delta t^2\frac{d\mathbf{v}_a^n}{dt};\\
\rho_a^{n+1} &= \rho_a^{n-1} + 2\Delta t\frac{d\rho_a^n}{dt}
\end{aligned} \tag{A.7a}$$

and once every 40 time steps, variables are calculated according to

$$\begin{aligned}
\mathbf{v}_a^{n+1} &= \mathbf{v}_a^n + \Delta t\frac{d\mathbf{v}_a^n}{dt};\\
\mathbf{r}_a^{n+1} &= \mathbf{r}_a^n + \Delta t\frac{d\mathbf{r}_a^n}{dt} + 0.5\Delta t^2\frac{d\mathbf{v}_a^n}{dt};\\
\rho_a^{n+1} &= \rho_a^n + \Delta t\frac{d\rho_a^n}{dt}.
\end{aligned} \tag{A.7b}$$

A variable time-step Δt is considered according to [27]:

$$\begin{aligned}
\Delta t &= 0.2\cdot\min(\Delta t_f, \Delta t_{cv});\\
\Delta t_f &= \min(\sqrt{h/|\mathbf{f}_a|})_a;\\
\Delta t_{cv} &= \min_a \frac{h}{c_s + \max_b\left|\frac{h\mathbf{v}_{ab}\mathbf{r}_{ab}}{\mathbf{r}_{ab}^2}\right|}
\end{aligned} \tag{A.8}$$

where Δt_f is based on the forcing terms and Δt_{cv} combines the CFL (Courant–Friedrich–Levy) condition and the viscous diffusion term.

The so-called dynamic boundary conditions [32] are employed in this work. It consists of creating particles that satisfy the same equations of continuity (A.5) and state (A.6) as the fluid particles, but their positions are not updated using the momentum equation. Thus, the evolution in time of density/pressure of the boundary particles creates a repulsion mechanism to prevent fluid particles from penetrating the limits of the domain or solid objects.

References

[1] R.A. Gingold, J.J. Monaghan, Smoothed particle hydrodynamics: theory and application to non-spherical stars, Monthly Notices of the Royal Astronomical Society 181 (1977) 375–389.
[2] R.J. Goozee, P.A. Jacobs, Distributed and shared memory parallelism with a smoothed particle hydrodynamics code, Australian and New Zealand Industrial and Applied Mathematics Journal 44 (2003) C202–C228.
[3] B.D. Rogers, R.A. Dalrymple, P.K. Stansby, D.R.P. Laurence, Development of a parallel SPH code for free-surface wave hydrodynamics, in: Proceedings of the 2nd SPHERIC International Workshop, Madrid, 2007, pp. 111–114.
[4] C. Moulinec, R. Issa, J.-C. Marongiu, D. Violeau, Parallel 3-D SPH simulations, Computer Modeling in Engineering & Sciences 25 (3) (2008) 133–148.
[5] A. Ferrari, M. Dumbser, E.F. Toro, A. Armanini, A new 3D parallel SPH scheme for free surface flows, Computers & Fluids 38 (2009) 1203–1217.
[6] P. Maruzewski, D. Le Touzé, G. Oger, F. Avellan, SPH high-performance computing simulations of rigid solids impacting the free-surface of water, Journal of Hydraulic Research 48 (2010) 126–134.
[7] M. Ihmsen, N. Akinci, M. Becker, M. Teschner, A parallel SPH implementation on multi-core CPUs, Computer Graphics Forum 30 (2011) 99–112.
[8] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 227 (2008) 5342–5359.
[9] A. Frezzotti, G.P. Ghiroldi, L. Gibelli, Solving the Boltzmann equation on GPUs, Computer Physics Communications 182 (2011) 2445–2453.
[10] A. Kolb, N. Cuntz, Dynamic particle coupling for GPU-based fluid simulation, in: Proceedings of the 18th Symposium on Simulation Technique, 2005, pp. 722–727.
[11] T. Harada, S. Koshizuka, Y. Kawaguchi, Smoothed particle hydrodynamics on GPUs, in: Proceedings of Computer Graphics International, 2007, pp. 63–70.
[12] A. Herault, G. Bilotta, R.A. Dalrymple, SPH on GPU with CUDA, Journal of Hydraulic Research 48 (2010) 74–79.
[13] A.J.C. Crespo, J.M. Dominguez, A. Barreiro, M. Gómez-Gesteira, B.D. Rogers, GPUs, a new tool of acceleration in CFD: efficiency and reliability on smoothed particle hydrodynamics methods, PLoS One 6 (6) (2011) e20685. http://dx.doi.org/10.1371/journal.pone.0020685.
[14] M. Gómez-Gesteira, B.D. Rogers, A.J.C. Crespo, R.A. Dalrymple, M. Narayanaswamy, J.M. Domínguez, SPHysics—development of a free-surface fluid solver—Part 1: theory and formulations, Computers & Geosciences 48 (2012) 289–299.
[15] M. Gómez-Gesteira, D. Cerqueiro, C. Crespo, R.A. Dalrymple, Green water overtopping analysed with an SPH model, Ocean Engineering 32 (2) (2005) 223–238.
[16] R.A. Dalrymple, B.D. Rogers, Numerical modeling of water waves with the SPH method, Coastal Engineering 53 (2006) 141–147.
[17] A.J.C. Crespo, M. Gómez-Gesteira, R.A. Dalrymple, Modeling dam break behavior over a wet bed by a SPH technique, Journal of Waterway, Port, Coastal and Ocean Engineering 134 (6) (2008) 313–320.
[18] M. Gómez-Gesteira, R. Dalrymple, Using a 3D SPH method for wave impact on a tall structure, Journal of Waterway, Port, Coastal and Ocean Engineering 130 (2) (2004) 63–69.
[19] A.J.C. Crespo, M. Gómez-Gesteira, R.A. Dalrymple, 3D SPH Simulation of large waves mitigation with a dike, Journal of Hydraulic Research 45 (5) (2007) 631–642.
[20] B.D. Rogers, R.A. Dalrymple, P.K. Stansby, Simulation of caisson breakwater movement using SPH, Journal of Hydraulic Research 48 (2010) 135–141.
[21] Nvidia, CUDA Programming Guide, 4.0. 2011. http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf.
[22] N.G. Dickson, K. Karimi, F. Hamze, Importance of explicit vectorization for CPU and GPU software performance, Journal of Computational Physics (2011) http://dx.doi.org/10.1016/j.jcp.2011.03.041.
[23] J.J. Monaghan, Smoothed particle hydrodynamics, Reports on Progress in Physics 68 (2005) 1703–1759.
[24] M.B. Liu, G.R. Liu, Smoothed particle hydrodynamics (SPH): an overview and recent developments, Archives of Computational Methods in Engineering 17 (2010) 25–76.
[25] M. Gómez-Gesteira, B.D. Rogers, R.A. Dalrymple, A.J.C. Crespo, State-of-the-art of classical SPH for free-surface flows, Journal of Hydraulic Research 48 (2010) 6–27.
[26] J.M. Dominguez, A.J.C. Crespo, M. Gómez-Gesteira, J.C. Marongiu, Neighbour lists in smoothed particle hydrodynamics, International Journal for Numerical Methods in Fluids 67 (2010) 2026–2042.
[27] J.J. Monaghan, A. Kos, Solitary waves on a Cretan beach, Journal of Waterway, Port, Coastal and Ocean Engineering 125 (3) (1999) 145–154.
[28] L. Verlet, Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, Physical Review 159 (1967) 98–103.
[29] G.R. Liu, Mesh Free Methods: Moving Beyond the Finite Element Method, CRC Press, 2003.
[30] J.J. Monaghan, Smoothed particle hydrodynamics, Annual Review of Astronomy and Astrophysics 30 (1992) 543–574.
[31] G.K. Batchelor, Introduction to Fluid Dynamics, Cambridge University Press, UK, 1974.
[32] A.J.C. Crespo, M. Gómez-Gesteira, R.A. Dalrymple, Boundary conditions generated by dynamic particles in SPH methods, CMC. Computers, Materials, & Continua 5 (3) (2007) 173–184.
[33] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proceedings of IEEE International Parallel & Distributed Processing Symposium, 2009.
[34] S. Green, CUDA Particles, NVIDIA Whitepaper Packaged with CUDA Toolkit, NVIDIA Corporation, 2010.
[35] M. Harris, Optimizing parallel reduction in CUDA, NVIDIA Presentation Packaged with CUDA Toolkit, NVIDIA Corporation, 2007.
[36] J.J. Monaghan, SPH without tensile instability, Journal of Computational Physics 159 (2000) 290–311.
[37] C. Yang, C. Huang, C. Lin, Hybrid CUDA, OpenMP and MPI parallel programming on multicore GPU clusters, Computer Physics Communications 182 (2011) 266–269.