Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations

Eugenio Rustico, Giuseppe Bilotta, Alexis Hérault, Ciro Del Negro, and Giovanni Gallo, *Member*, *IEEE*

Abstract—We present a multi-GPU version of GPUSPH, a CUDA implementation of fluid-dynamics models based on the smoothed particle hydrodynamics (SPH) numerical method. The SPH is a well-known Lagrangian model for the simulation of free-surface fluid flows; it exposes a high degree of parallelism and has already been successfully ported to GPU. We extend the GPU-based simulator to run simulations on multiple GPUs simultaneously, to obtain a gain in speed and overcome the memory limitations of using a single device. The computational domain is spatially split with minimal overlapping and shared volume slices are updated at every iteration of the simulation. Data transfers are asynchronous with computations, thus completely covering the overhead introduced by slice exchange. A simple yet effective load balancing policy preserves the performance in case of unbalanced simulations due to asymmetric fluid topologies. The obtained speedup factor (up to 4.5× for 6 GPUs) closely follows the expected one (5× for 6 GPUs) and it is possible to run simulations with a higher number of particles than would fit on a single device. We use the Karp-Flatt metric to formally estimate the overall efficiency of the parallelization.

Index Terms—GPU, multi-GPU, SPH, CUDA, fluid dynamics, numerical simulations, load balancing, parallel computing, HPC

1 Introduction

The numerical simulation of fluid flows is an important topic of research with applications in a number of fields, ranging from mechanical engineering to astrophysics, from special effects to civil protection. A variety of computational fluid-dynamics (CFD) models are available, some specialized for specific phenomena (shocks, thermal evolution, fluid/solid interaction, etc.) or for fluids with specific rheological characteristics (gas, water, mud, oil, petrol, lava, etc.).

The computational complexity of applied CFD models depend on the complexity of the model itself, as well as on the accuracy required for the specific application. When the numerical model exposes a high degree of parallelism, its implementation on parallel high-performance computing (HPC) platforms can help overcome the complexity and reduce execution times. Among the many possible parallel HPC solutions, a new approach that has emerged lately is the use of graphic processing units (GPUs), hardware initially developed for fast rendering of dynamic 3D scenes, as numerical processor for computationally intensive, highly parallel tasks.

- *E. Rustico and G. Gallo are with the Department of Mathematics and Informatics, University of Catania, Viale Andrea Doria 6, 95125 Catania Italy. E-mail: {rustico, gallo}@dmi.unict.it.*
- *G. Bilotta is with the Istituto Nazionale di Geofisica e Vulcanologia, Osservatorio Etneo, Piazza Roma 2, 95125 Catania, Italy and the Department of Mathematics and Informatics, University of Catania, Viale Andrea Doria 6, 95125 Catania Italy. E-mail: bilotta@ct.ingv.it.*
- *C. Del Negro is with the Istituto Nazionale di Geofisica e Vulcanologia, Osservatorio Etneo, Piazza Roma 2, 95125 Catania, Italy. E-mail: ciro.delnegro@ct.ingv.it.*
- *A. Hérault is with the Département Ingénierie Mathématique, Conservatoire National des Arts et Métiers, 292, Rue Saint Martin - 75141 Paris, France and the Istituto Nazionale di Geofisica e Vulcanologia, Osservatorio Etneo, Piazza Roma 2, 95125 Catania, Italy. E-mail: alexis.herault@cnam.fr.*

Manuscript received 22 Feb. 2012; revised 2 Nov. 2012; accepted 10 Dec. 2012; published online 21 Dec. 2012.
Recommended for acceptance by S. Ranka.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2012-02-0128.
Digital Object Identifier no. 10.1109/TPDS.2012.340.

1.1 The GPGPU Paradigm

Usage of GPUs for scientific computing goes back to the introduction of *programmable shaders* in 2001, but the true power of the shader hardware was only unlocked 6 years later, with the development of NVIDIA's CUDA technology [1] and the competing ATI Stream [2].

While typically running a lower clock rates than modern CPUs, a single GPU features a large number of computing cores (for more recent cards, in the order of thousands), making the hardware most appropriate for problems that exhibit a high level of parallelism on a fine data granularity, achieving as much as two orders of magnitude in speedup over single-core CPUs [3], [4].

Cost-effectiveness and easier programming, combined with the spread of GPU computing beyond the academic world has led to claims that the *GPU computing era* has begun [5], even though some have criticized the enthusiasm for GPGPU as "excessive," showing that a well-tuned CPU implementation, optimized for execution on recent multi-core processors, often reduces the flaunted 100× speedup reported by many works [6].

Additional information about GPGPU can be found in Appendix 1.1, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.340.

1.2 Multi-GPU Computing

With the rise of GPGPU, there has also been a growing interest in the possibility of concurrently exploiting more than one GPU; and while modern GPUs are far easier to program than in the early days of GPGPU, exploiting more than one GPU at a time is more challenging, due to technical and model constraints. Still, a variety of multi-GPU software is available in many fields, such as data compression and visualization [7], [8], [9], molecular dynamics [10], [11], [12], or CFD [13], [14], [15], [16], [17].

One of the most challenging issues in writing efficient multi-GPU code is proper load-balancing to ensure all devices are fully employed; this is a still a young research field, with only limited literature available (see, e.g., [18]).

1.3 From SPH to GPUSPH

Smoothed particle hydrodynamics (SPH) is a meshless Lagrangian numerical method for computational fluid-dynamics, originally developed by Gingold and Monaghan [19], [20] for astrophysics, and that has recently seen applications in a number of fields. We refer the reader to recent reviews on SPH [21], [22] for further information.

Direct per-particle computation of physical quantities and their derivatives ensures that SPH is easy to parallelize (one particle per thread), making it ideal for implementation on parallel architectures such as GPUs. One of the first GPU-based SPH solvers was developed by Amada et al. [23], which offloaded force computation to the GPU while the CPU took care of other tasks such as neighbor search.

The first implementations of the SPH method completely on the GPU is due to Kolb and Cuntz [24] and Harada et al. [25]. Their use of OpenGL and Cg (C for graphics) to program the GPU due to the lack of a better solution lead to some technical limitations that reduced the implementation performance.

The introduction of the CUDA architecture for NVIDIA cards in 2007 allowed to fully exploit the computational power of modern GPUs without the limitations imposed by having to go through the graphical engine. The first CUDA implementation of the SPH model [26] was developed at the Sezione di Catania of the Istituto Nazionale di Geofisica e Vulcanologia (INGV-CT) in cooperation with the Department of Civil Engineering of the Johns Hopkins University. It was inspired by the Fortran SPHysics code [27] and has been recently published as the open source project GPUSPH [28], used in applications ranging from coastal engineering [29], [30], [31] to lava flow simulation [32], [33], [34]. In the mean time, other CUDA implementations of SPH have emerged [35], [36]

1.4 Our Contribution

The work we present here is based on [26], which we recently extended to allow the distribution of the computation across multiple GPUs [37]. The possibility of using multiple devices brings at least two benefits to the method implementation, and specifically the possibility to run simulations whose data would not fit on a single device, as well as the possibility of completing simulations in a fraction of the time necessary on a single device.

In this paper, we present a newer version featuring a posteriori load balancing as well as several minor optimizations that also improve single-GPU performance. The optimizations lead to a speed-up of over $2\times$ in single-GPU execution that also reflects on multi-GPU runs. A formal analysis of the efficiency of the load balancing approach is also discussed.

The paper is structured in the following way: the essential elements of our single-GPU SPH implementation are highlighted, together with the recent optimizations that have more than doubled its performance. We then introduce our multi-GPU implementation, highlighting the enhancements done over our initial multi-GPU work. Some results are finally shown, to illustrate the practical gains of the new multi-GPU implementation, and an analysis of the results is presented to quantitatively assess the optimality of this second level of parallelism.

2 Single-GPU SPH

We start with an overview of the single-GPU structure of GPUSPH, as a necessary basis to understand the challenges posed by the multi-GPU implementation and how they have been overcome. For a more detailed description of the simulator and its performance results, we refer the reader to [26]. Some optimizations over the original implementation, and their effect on the performance of the code, are also highlighted here.

2.1 Kernels

The organization of the SPH method in CUDA kernels directly reflects the computing phases of the model, which are described in more detail in Appendix 2, which is available in the online supplemental material. The kernels are as follows:

1. **`BuildNeibs`**—For each particle, build the list of neighbors.
2. **`Forces`**—For each particle, compute the interaction with neighbors and its maximum allowed Δt.
3. **`MinimumScan`**—Select the minimum Δt among the maxima provided by each particle.
4. **`Euler`**—For each particle, update the scalar properties integrating over selected Δt.

All but the first kernel are executed twice for each iteration, due to our predictor/corrector integration scheme.

2.2 Fast Neighbor Search

During a single iteration the neighbors of each particle have to be accessed multiple times. It is, thus, faster to prepare a neighbor list once and iterate over it rather than searching for neighbors whenever they are needed. The naive $O(N^2)$ approach can be sped up by partitioning the domain in virtual cells with side equal to the influence radius, and sorting particles by a hash value computed as the linearized index of the cell they belong to. Neighbors of a particle can then be sought by only looking at the cells which are in neighboring *cells* [38]. Fig. 1 is a schematic representation of the virtual cells in 2D is shown in. We remark that this grid serves only to speed up the neighbor search, and has nothing to do with the SPH model.

Even with this approach, building the neighbor list for each particle still requires about 50 percent of the total simulation time. As a further optimization we only rebuild the neighbor list every kth iteration; with $k = 10$, this causes a negligible loss of precision while decreasing the time required for the neighbor list construction from 50 percent to about 10 percent of the total simulation time.

2.3 Optimizations

A few important changes were done to the GPUSPH code over the implementation described in [26]. None of these

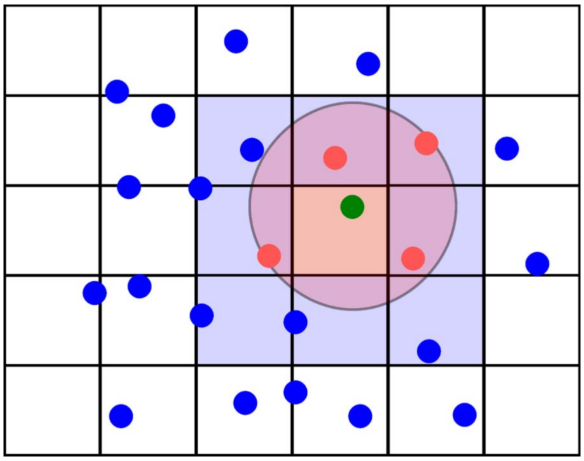


Fig. 1. If cells have side equal to the influence radius, neighbors of the green particle must reside inside in the immediate neighbor cells (light blue).

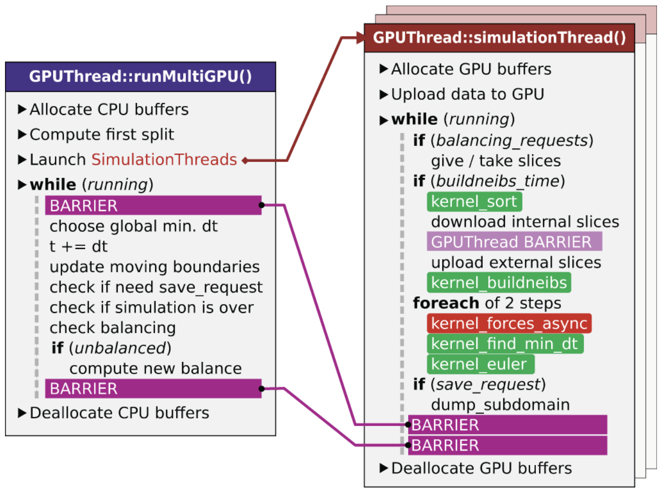


Fig. 2. Scheme of the simulator structure: GPU kernels are green, NPTL barriers are purple. The red `kernel_forces_async` method also manages asynchronous slice exchanges. Light purple barrier is only reached by `simulationThread`s (inter-GPU synchronization), dark barriers serve for communication between the main thread and the `simulationThread`s.

changes alter the underlying program structure, and all of them contribute to a significant speedup.

A detailed analysis of cache utilization was done on devices belonging to the Fermi architecture, and in a number of kernels textures for read-only access to the position array were replaced by direct access to the underlying array, resulting in a more effective use of both texture cache (now used for velocities and particle information only) and the new L2 cache available on Fermi devices.

Second, the layout of the neighbor list was changed. Previously, neighbors were laid out grouped per particle, with all the neighbors of the first particle followed by the neighbors of the second particle, and so on and so forth. Our new implementation interleaves neighbors, so that the first neighbor of all particles come first, followed by the second neighbor of all particles, and so on. On GPU, this ensures memory coalescence when particles handled by threads in the same warp traverse the neighbor list.

Finally, since CUDPP is not distributed with CUDA anymore, all dependencies on it were removed. The minimum scan for the Δt was replaced by a custom kernel, and the radix sort for particle reordering during neighbor list construction now relies on the `thrust` library shipped with CUDA since its fourth release.

2.4 Performance

As discussed in [26], the original implementation of GPUSPH could reach a speedup of about two orders of magnitude over an equivalent single-core CPU implementation. In terms of raw throughput, on a GTX 480 this has been measured at about 10,000 kips, a performance metric we introduced to gauge the performance of our code, discussed in more detail in Section 4.1.

With the optimizations reported in Section 2.3 the single-GPU performance has raised to about 24,000 kips, resulting in a net speedup of about $2.4\times$. Further details on the results of the single-GPU optimizations can be found in Appendix 3, which is available in the online supplemental material.

A more recent comparison of the optimized single-GPU version against a multithreaded CPU implementation of the code, running on recent multicore CPUs, is also presented in Appendix 4, which is available in the online supplemental material, showing a CPU performance of between 600 and 1,300 kips, indicating that a modern GPU outperforms a modern, multicore CPU by a factor between 20 and 40.

3 Multi-GPU GPUSPH

Exploiting the second level of parallelism offered by multiple GPUs required structural changes to the original GPUSPH code, to achieve a cleaner split between the global particle system management and the program sections dedicated to the management of individual devices. The code was also made multithreaded, with the main thread managing the entire particle system, and a separate additional host thread launched for each GPU. A general scheme of the multi-GPU design is presented in Fig. 2. The reader is referred to [37] for further details.

In the following, only the key structural elements of the multi-GPU design are recalled, as appropriate to understand the advancements over the previous multi-GPU version.

3.1 Splitting the Problem

The key idea for exploiting multiple GPUs concurrently in a single simulation is to split the total computational burden across the devices. The optimal way to achieve this depends on the model being used. In the case of our SPH model, since the steps in the algorithm for the single iteration are strictly sequential, while the SPH method itself is “embarrassingly parallel” with spatial locality, we prefer a spatial decomposition. The two major points that have to be looked for in designing the split are the need to access neighbors that might be on different GPUs, and the need to look for a minimum Δt across the entire domain.

We recall that, as described in Section 2.2, we use an auxiliary 3D grid of virtual cells to sort particles and speed up the neighbor list construction build. This same grid can

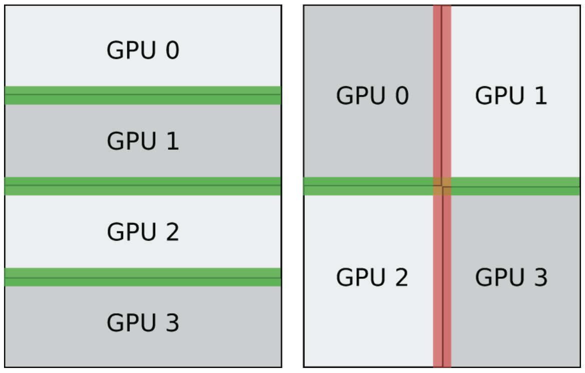


Fig. 3. Splitting a 2D domain along parallel or orthogonal axes. If data are linearized per row, green edges can be copied in one memory transfer, while red edges require many small transactions.

be exploited to assist in the spatial domain splitting in the multi-GPU case.

Specifically, we split the domain along planes that pass through the interface between cells. In the case of multiple splits, we forbid splitting along orthogonal planes, since transferring the edges of a subdomain would then become very expensive. Fig. 3 illustrates the situation in a simpler 2D case. Assuming that the cells are enumerated row-first, it is possible to transfer every green edge by requesting a single memory operation, since cells along a green edge occupy contiguous memory areas; on the other hand, transferring cells along a red edge requires many small transactions. It is, therefore, recommended that all the split lines (or planes in 3D) are aligned to the most significant axis used for the cell index linearization.

This is enforced in our code by defining a single splitting plane direction, and our domain is always split in slices that are parallel to each other. Different problems can prefer different split plane directions, and we leave this as an option that can be selected at compile time. The choice influences the enumeration of the cells, so that the linearization of their 3D grid coordinate follows the preferred split plane direction.

As a result of our choices, the computational domain can be considered decomposed in *slices*, parallel to each other and one cell thick, with each subdomain containing a number of consecutive slices. As a rule of thumb, the split direction should be chosen so that the number of particles per slice is minimized. The choice normally has little influence on the performance, since the time required for transferring overlapping slices is usually completely covered by the force computation, as will be shown later. Very asymmetric topologies, however, can benefit from an optimal splitting direction choice when load balancing, as the balancing granularity is at the slice level and the balancing operations are not covered (see Section 3.5).

3.2 Subdomain Overlap

To compute the forces acting on a particle p in its assigned subdomain, the GPU must be able to access the data for all neighbors of p. If p is in a slice at the edge of the subdomain, some of the neighbors might be in the edge slice of an adjacent subdomain (since the slice thickness is equal to the influence radius, neighbors

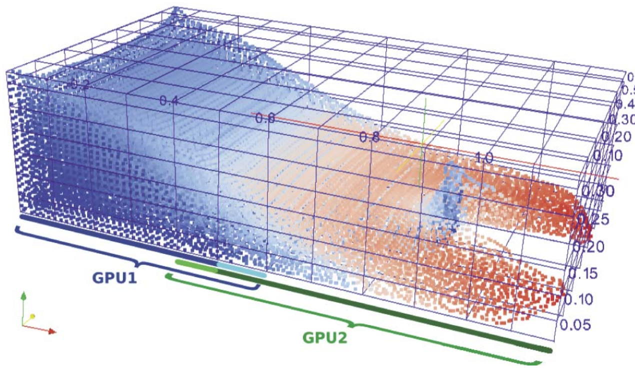


Fig. 4. Fluid volume with virtual cells, split on YZ plane. The blue line shows the internal cells of GPU1; light blue the external read-only slice updated by GPU2; internal cells of GPU2 are green, while the external slice is light green. Particles are colored by velocity and cells are not in scale for visualization purposes.

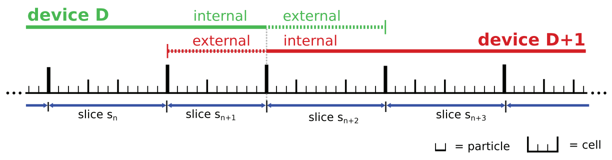


Fig. 5. List split technique. When particles are sorted by linearized cell index, the domain can be split by splitting the list of particles where slices begin.

cannot be further than that). Each GPU should, therefore, hold a copy of the edging slices from adjacent subdomains: the particles in its subdomain are read/write, and will be called *internal*, while particles copied from adjacent subdomains will be called *external*. A similar nomenclature will be used for slices.

A 3D example of a domain split illustrating subdomain overlap, taken from a real simulation, is illustrated in Fig. 4. Particles are colored by velocity, and the blue spot in the subdomain of the second GPU is due to the impact with an obstacle which was omitted for visualization clarity. The colored bars on the side of the domain mark the internal (resp. blue, green) and external (resp. light blue, light green) slices for each of the domains, highlighting the two-slice overlap between them. Fig. 5 represents the same subdivision from the viewpoint of the list of particles.

3.3 Kernels

Kernels were adapted to work on subdomains as described in the following.

The neighbor list (step 1) is only constructed for internal particles, but the search is done in both internal and external slices. Likewise, force computation (step 2) is only done on internal particles, since there is not enough data to compute forces on external particles.

By consequence, minimum scan (step 3) for the minimal Δt is only done on internal particles. The CPU then collects the local minimum computed by each GPU and finds the global minimum. The only requirement here is a barrier to wait for all the local minima to be ready.

Integration (step 4), on the other hand, is done on both internal and external particles, because data for particles in the overlap region are exchanged during force computation (see Section 3.4). A side effect of this choice is that

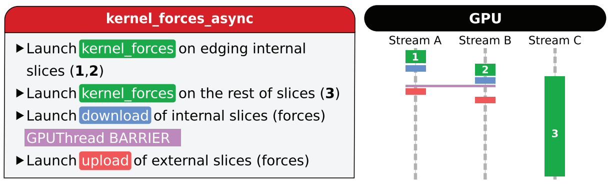

kernel_forces_async
Launch kernel_forces on edging internal slices (1,2)
Launch kernel_forces on the rest of slices (3)
Launch download of internal slices (forces)
GPUThread BARRIER
Launch upload of external slices (forces)
GPU
Stream A
Stream B
Stream C

Fig. 6. `kernel_forces_async` design. Edge forces are exchanged as they become ready, during computation of other forces.

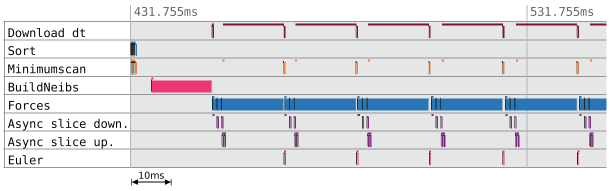


Fig. 7. Actual timeline of a multi-GPU GPUSPH simulation, with kernel lengths in scale (lengths smaller than 1 pixel have been rounded up). Only one GPU with two neighboring devices is shown. Little dots mark the moment operations were issued on the CPU. Downloading the Δt is blocking for the CPU (hence the long brown line) until the minimum scan is complete.

integration for particles in the overlap regions is done twice (once on each GPU they belong to); this does not introduce a measurable overhead, since the integration step barely saturates a GPU.

3.4 Hiding Slice Transfers

For neighbor list construction and force computation, each GPU needs up-to-date position and velocity for external particles. Exchanging the updated position and velocity after each integration step, between kernel launches, twice per iteration because of the predictor/corrector scheme, introduces a significant overhead: in some simulations, this results in simulations taking longer on two GPUs than on a single one.

To overcome this problem, we exploit the hardware capability to perform concurrent computations and data transfers, with an approach we already applied successfully to a different, cellular-automaton-based simulator [39]. We use the *asynchronous API* offered by the CUDA platform to begin the transfers as soon as the edging slices are ready, while the other ones are still being computed.

We initialize three *CUDA streams* for each device: two are used to enqueue operations on the edge slices and one for the remaining slices. As discussed in [37], we exchange *forces* only (see Fig. 6), letting each GPU integrate its own copy of the particles, as this is more convenient than exchanging positions and velocities after integration.

A single slice does not usually provide enough workload to saturate a GPU during execution of the `Forces` kernel: running the same kernel on more than one slice takes about the same time, and additionally it reduces the workload of the "central" launch. It is, thus, convenient to launch the kernels on the edging *stripes*, i.e., a small number of slices with enough particles to saturate the GPU. The actual transfer of edges, instead, will only include single edging slices.

Fig. 7 shows a detailed timeline of events on the middle GPU during an actual simulation of a `DamBreak3D` with 1 million particles on 3 GPUs. Slice exchange (purple) is indeed performed concurrently with forces computation,

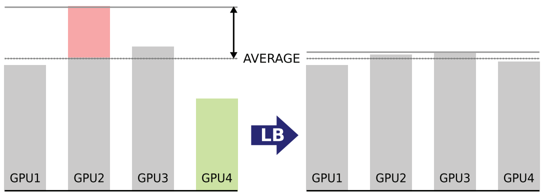


Fig. 8. Average execution times before and after load balancing.

starting as soon as the computation of forces on the respective edge is completed. In the plot, rectangles mark the start time and duration of the events on the device, while the little dots above them mark the time at which the operation is issued on the host. All operations are asynchronous to the host except for the download of the Δt, which is blocking on the host, as shown by the long brown lines. The plot was produced by a custom profiler-visualizer we developed to overcome the limits of the standard profilers provided with CUDA.

3.5 Load Balancing

In the ideal case of all the GPUs taking the same amount of time to complete each step, the simulation time is expected to speed up in a quasi-linear way (with the only exception of negligible constant factors such as the kernel launch latency).

In general, this is difficult to guarantee, and even small differences from the average time may lead to a considerable performance loss. A device taking $n\%$ more time than the average to perform all the operations between two synchronization barriers will worsen the whole simulation time by exactly $n\%$. The purpose of load-balancing is to distribute the load so that the gap between the longest and average runtime is kept as small as possible (see Fig. 8).

Dividing the fluid in parts of the same size (i.e., same number of particles) does not always lead to the optimal workload balance. Indeed, many unpredictable elements may influence the total computation time, such as the sparsity of neighbor particles since last sort, the fluid topology, branch divergences inside a kernel and even hardware factors such as PCI interrupts and bus congestion. No balancing model can take all these factors into account without relying on the execution time of the previous steps. An a posteriori load balancing technique is more appropriate here.

The key idea is very simple: we keep track of the time required by each GPUs for the `Forces` kernel and we ask the GPUs taking longer than the average to *give* a slice of their subdomain to GPUs taking less.

More in detail, we consider the average time A_g taken by a single GPU g to complete the forces kernel on the central set of particles over the last k iterations. A smart choice of k could be a multiple of the number of iterations between two reconstructions of the neighbor list, to minimize the number of sorts; in our case, $k = 10$. We then compute the cross-GPU average

$$A_G = \sum_{d=1}^{D} A_d / D,$$

and $A_{\delta g} = A_g - A_G$.

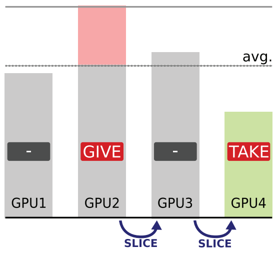


Fig. 9. Example of balancing: GPU2 is marked as GIVING, GPU4 as TAKING and GPU3 is requested to take a slice from GPU2 and give one to GPU4.

If $|A_{\delta g}| \geq T_{LB}$, with T_{LB} as balancing threshold, we mark the GPU g as *giving* (if $A_{\delta g} > 0$) or *taking* (if $A_{\delta g} < 0$). A giving GPU "sends" one slice to the taking one; if they are not neighboring, every intermediate GPU gives one slices and receives another at the appropriate edge (see Fig. 9). A_g is reset to wait for next k iterations. Listing 1 represents this algorithm in pseudocode.

The threshold T_{LB} must be big enough to avoid sending slices back and forth and small enough to trigger the balancing when needed. Let T_{slice} be the average time required to compute the `Forces` kernel on a single slice; because the granularity is at the slice level, it convenient to set T_{LB} proportionally to T_{slice}:

$$T_{LB} = H_{LB} \cdot T_{slice},$$

with H_{LB} as *balancing threshold coefficient*. As a general rule, the optimal value for H_{LB} depends on the problem topology as well as the number of GPUs used. However, in our tests the value $H_{LB} = 0.5$ has always led to acceptable, even when not optimal, performance (see, e.g., Fig. 10).

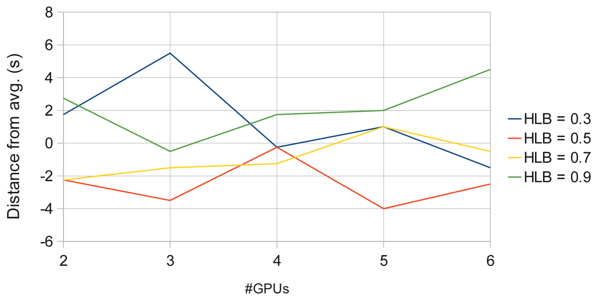
```
AG ← global avg. duration of Forces
Ts ← avg. duration of one slice
foreach GPU
  Ag ← self avg. duration of Forces
  delta ← Ag - AG
  if (abs(delta) > Ts * HLB_THRESHOLD)
    if (delta > 0)
      mark as GIVING_CANDIDATE
    else
      mark as TAKING_CANDIDATE
foreach GPU
  if a couple GIVING/TAKING is found
    mark GIVING_CANDIDATE as GIVING
    mark TAKING_CANDIDATE as TAKING
    foreach intermediate GPU
      mark as GIVING and TAKING
    break
```

Listing 1. Load balancing pseudocode

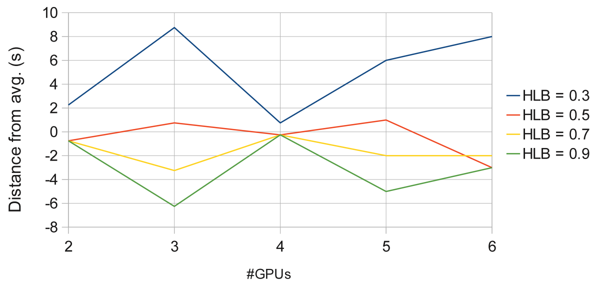
The oscillating behavior seen in Fig. 10b is due to the presence of a narrow obstacle that changes the general subdivision of the fluid: when simulating with an odd number of GPUs, one of them entirely contains the obstacle, while with an even number of devices the obstacle is "shared" between two of them.

It is worth mentioning, however, that these considerations are only interesting for the sake of theoretical research. The performance variations refer to simulations lasting several hundreds of seconds (from about 1,200 s with two devices to about 400 s with six devices) and are thus negligible in practice (<2% in the worst case).

Fig. 11 show the evolution of two variants for the `BoreInABox` simulation. The about 1.1 million particles are colored by device to highlight the dynamics of balancing. In the corridor variant, the workload keeps becoming more and more asymmetric, while in the walled `BoreInABox` the original symmetry is partially restored when the fluid redistributes evenly on the floor.



(a) `BoreInABox`



(b) `DamBreak3D`

Fig. 10. Performance of load balancing (distance in seconds from the average time) with respect to threshold H_{LB}.

4 Load-Balancing Performance

The effectiveness of the implemented load balancing policy has been tested by measuring the achieved performance during the simulation of a number of different problems, with different configurations. We will present here only the results for the `BoreInABox` problem with corridor. Further tests can be found in Appendix 5, which is available in the online supplemental material. The multi-GPU code also includes the optimizations discussed in Section 2.3.

4.1 Performance Metric

As we often simulate problems exhibiting different densities, topologies, and number of particles, the absolute execution time required by a simulation to complete is too simple a performance metric for our purposes. It is also not meaningful to measure the speedup in the Amdahl's [40] or Gustafson's [41] meanings, unless comparing simulations with identical settings and simulated time. We needed a more versatile metric abstracting from the specific simulation settings.

We measure the *total workload* of a simulation as the number of iterations multiplied by the number of particles; *throughput* can then be measured dividing the total workload by the runtime, in seconds. Due to the large number of particles and iterations in our simulations it is more convenient to measure performance in *thousands of iterations on particles per second*, shortened to *kip/s* or simply *kips*,

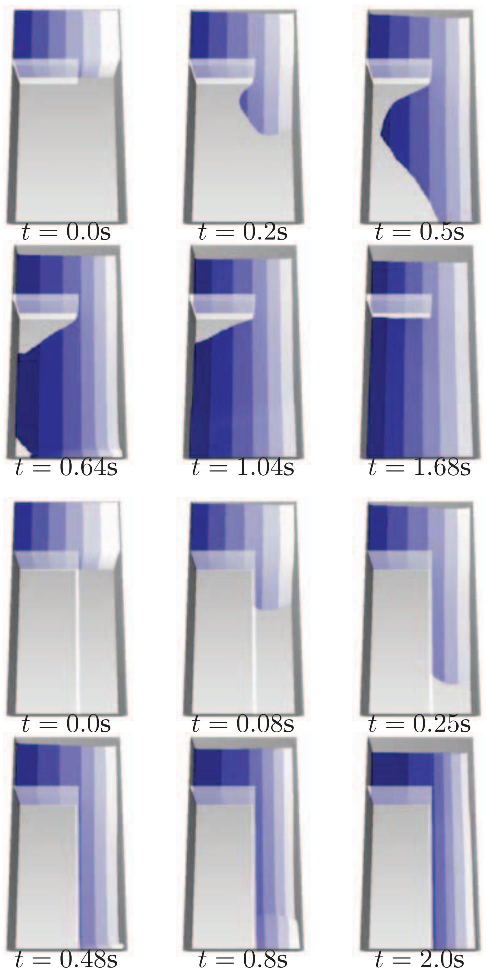


Fig. 11. Different phases of the `BoreInABox` problem simulation, wall (above) and corridor (below) variants, on 6 GPUs. Particles are colored by GPU number.

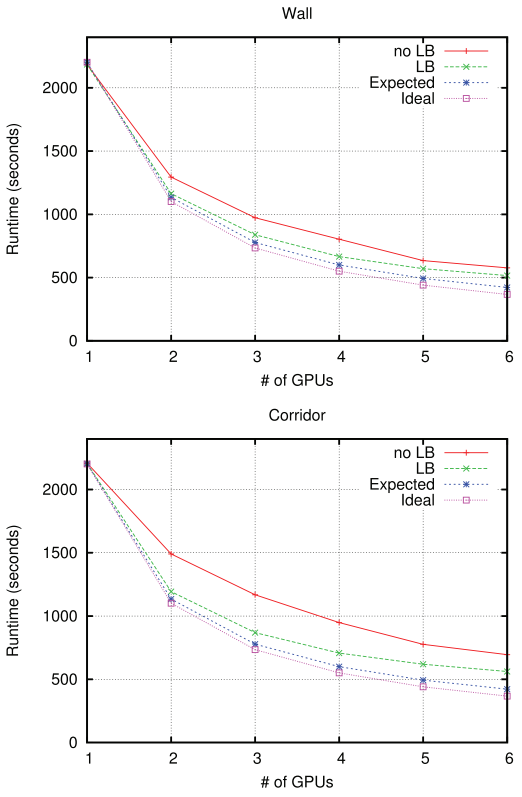


Fig. 12. Multi-GPU runtimes, `BoreInABox` wall and corridor variants, 1.6 million particles, 1 to 6 GPUs.

or *millions of iterations on particles per second*, shortened to *mip/s* or simply *mips* (not to be confused with the *MIPS* acronym typically used to indicate millions of instructions per second).

An advantage of this metric compared to the mere speedup is that it can be computed instantaneously at runtime, without waiting for a simulation to complete. This metric, however, is specific for particle methods and may not be suitable for other models.

The performance analysis will include data on actual runtimes (in seconds), throughput (in mips) as well as speedup as per Amdahl's meaning, to ease a possible comparison with any future or independent work. It is worth recalling that for any comparison to be accurate the same integration scheme and physical settings must be used. It is also advisable to simulate similar fluid topologies, as different topologies can still affect memory coalescence and thread/block scheduling.

4.2 Testbed

Our testing platform is a TYAN-FT72 rack mounting $6 \times$ GTX480 cards on as many second generation PCI-Express slots. The system is based on a dual-Xeon processor with 16 total cores (E5520 at 2.27 GHz, 8-MB cache) and 16-GB RAM in dual channel. Each GTX480 has 480 CUDA cores grouped in 15 multiprocessors (MPs), 48-KB shared memory per MP and 1.5-GB global memory with a measured datarate of about 3.5-GB/s host-to-device and 2.5-GB/s device-to-host (with 5.7-GB/s HtD and 3.1-GB/s DtH peak speeds on pinned buffers). The operating system is Ubuntu 10.04 x86_64, gcc 4.4.3, CUDA runtime 4.2 and NVIDIA video driver 295.59.

In GPUSPH terminology, a *problem* is the definition of a physical domain, fluid volumes and geometrical shapes (a scene) to simulate. The reference scenario was a box with 0.43 m^3 of water divided into 1.6 million particles for 1.5 s of simulated time; inner walls control the flow path in the two `BoreInABox` variants ("wall" and "corridor").

4.3 Results

When simulating an asymmetrical problem like `BoreInA-Box`, load balancing makes a significant difference in performance. During the 1.5 s simulated time of `BoreInA-Box`, the particles flow in the lateral corridor and one third of the simulation domain, considering Y as split axis, receives two thirds of the total fluid. As seen in Figs. 12 and 13 and Table 1, load balancing brings performance closer to the ideal one, with runtimes decreasing faster than without load balancing. In the plots, the measured runtimes, throughput and speedup is compared against both ideal results (assuming a perfectly parallelizable program) and the results that could be theoretically expected, assuming no overheads, from a rough estimate of the actually parallelizable part of the code. Further details about the expected speedup can be found in Appendix 3.1, which is available in the online supplemental material.

The effect of load balancing is more prominent in the corridor case than in the wall case, since in the latter case the fluid distribution tends to rebalance the workload.

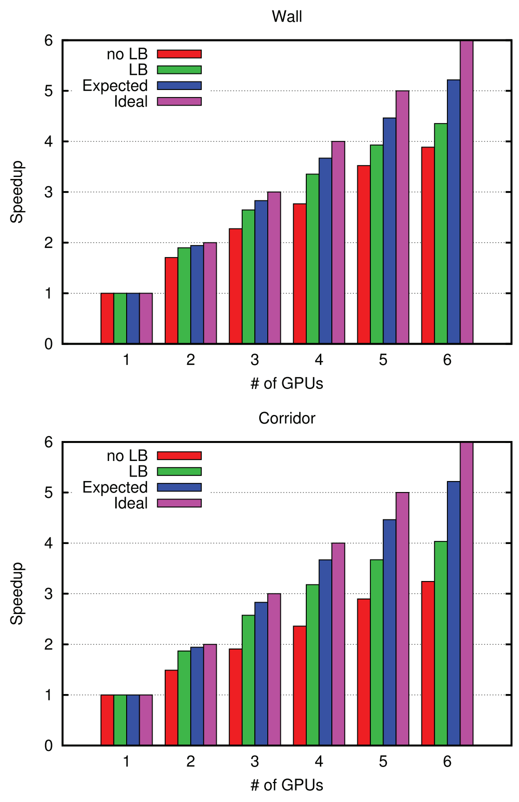


Fig. 13. Multi-GPU speedup, `BoreInABox` wall and corridor variants, 1.6 million particles, 1 to 6 GPUs.

4.4 Analysis

A common starting point to analyze a parallel implementation is to model the problem as made by a parallelizable part and a serial, non parallelizable one. The latter includes operations which are necessarily performed in a serial manner (e.g., file I/O) and operations that are introduced with the parallel implementation (*overhead*, e.g., load balancing). Let α be the parallelizable fraction of the problem and β the nonparallelizable one; we have $\alpha + \beta = 1$, with α, β nonnegative real numbers. For embarrassingly parallel problems α is equal or very close to 1; problems intrinsically serial, where each step depends on the outcome of the previous one, have $\beta \approx 1$.

4.4.1 Karp-Flatt Metric

The Karp-Flatt metric was proposed in 1990 as a measure of the efficiency of the parallel implementation of a problem [42]. The experimental measure of the serial fraction β of a problem is computed as

$$\beta_{KF} = \frac{1/S_e - 1/N}{1 - 1/N}, \tag{1}$$

where S_e is the *measured* speedup and N is the number of cores. The formula can be obtained rewriting Amdahl's for β, and the result can be read as the effective percentage of serial code. Since this a posteriori metric measures both serial time and introduced overhead, it is especially useful to apply to at least two executions with different values of N: a constant β_{KF} means that the parallelization is efficient, while a growing β_{KF} means that some overhead is being introduced by the growing number of cores.

TABLE 1
Multi-GPU Performance in mips, `BoreInABox` Wall and Corridor Variants, 1.6 Million Particles, 1 to 6 GPUs

	1	2	3	4	5	6
wall						
LB off	24.6	42	56.1	68.2	86.8	95.7
LB on	—	46.8	65.2	82.6	96.8	107.3
Ideal	—	49.3	73.9	98.6	123.1	147.8
corridor						
LB off	24.6	36.7	47	58.2	71.4	79.9
LB on	—	46.0	63.4	78.3	90.5	99.4
Ideal	—	49.2	74	98.5	123.1	147.7

TABLE 2
Computed Multi-GPU Serial Code Fraction (Karp-Flatt Metric), `BoreInABox` Wall and Corridor Variants, 1.6 Million Particles, 2 to 6 GPUs

	2	3	4	5	6
wall					
LB off	17.4%	15.9%	14.9%	10.5%	10.9%
LB on	5.4%	6.7%	6.5%	6.8%	7.6%
corridor					
LB off	34.3%	28.6%	23.1%	18.1%	17%
LB on	7%	8.2%	8.6%	9%	9.7%

4.4.2 Measured β

The empirically measured β_{KF} for the multi-GPU GPUSPH (see Table 2) confirms the overall efficiency of our multi-GPU implementation, but indicates that our current load-balancing strategy introduces some overhead.

More precisely, we see that, without load balancing, β_{KF} decreases steadily as the number of GPUs increases, indicating the lack of significant overhead. As expected, the strongly asymmetric corridor problem performs worse, 34 to 17 percent measured serial fraction, compared with the 17 to 10 percent in the wall problem.

With load balancing, the overall efficiency of the implementation improves, with the measured serial fraction consistently falling below 8 percent in the wall problem (30 percent improvement) and below 10 percent in the corridor problem (40 percent improvement). However, the Karp-Flatt metric in this case is increasing, indicating that the load-balanced executions have some uncovered overhead.

Some minor overheads occur with or without load balancing: for example, the uncovered data transfer caused by particles physically moving from one subdomain to the other. These small overheads have different relative weight depending on the topology of the simulation (e.g., their influence can be noticed in the nonmonotonic trend of β_{KF} in the wall problem), but their weight grows as the efficiency of the algorithm improves, and they are, therefore, more noticeable with load-balancing, where the overall efficiency is higher than without. However, an important source of overhead is introduced by the load balancing itself, since moving slices from 1 GPU to another during the balancing is not covered by kernel execution, in contrast to the slice exchange during standard operation.

5 Conclusions and Future Work

We presented a scalable, load-balanced, multi-GPU implementation of a CUDA-based SPH fluid simulator. Simulations scale almost linearly with the number of GPUs used. The effect of asymmetries in the topology of the simulated fluid is neutralized by newly introduced a posteriori load balancing system. Although the balancing policy is a first, simple attempt involving no specific signal-processing techniques, it shows a high robustness and excellent performance in almost all the tests we could perform. An empirical estimate of the serial fraction β of the simulator indicates an overall higher efficiency of the simulator with load-balancing than without, but also shows that the load-balancing introduces an overhead. Finally, we are able to run simulations with more particles than one device could fit, thus achieving the second aim of the multi-GPU implementation.

The main improvements we are currently working on are: a reduction of the load-balancing overhead, a more refined load-balancing algorithm with accurate temporal analysis and finer granularity, and finally the development of a multinode version of the simulator for GPU-based clusters, together with a more sophisticated domain decomposition strategy.

Acknowledgments

The authors would like to thank Prof. Robert A. Dalrymple from Johns Hopkins University, Baltimore, for his invaluable help during the development of the first multi-GPU GPUSPH prototype. The development of the multi-GPU version of GPUSPH was funded by Mind Technologies, Paris, France.

References

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, pp. 40-53, http://doi.acm.org/10.1145/1365490.1365500, Mar. 2008.

[2] ATI, *ATI Stream Computing—User Guide*, Dec. 2008.

[3] Z. Yang, Y. Zhu, and Y. Pu, "Parallel Image Processing Based on CUDA," *Proc. Int'l Conf. Computer Science and Software Eng.*, vol. 3, no. 208, pp. 198-201, 2008.

[4] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J.D. Owens, "Efficient Computation of Sum-Products on GPUs through Software-Managed Cache," *Proc. Second Ann. Int'l Conf. Supercomputing (ICS '08)*, pp. 309-318, http://doi.acm.org/10.1145/1375527.1375572, 2008.

[5] J. Nickolls and W.J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56-69, Mar. 2010.

[6] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *ACM SIGARCH Computer Architecture News* , vol. 38, pp. 451-460, http://doi.acm.org/10.1145/1816038.1816021, June 2010.

[7] E. Attardo, A. Borsic, and R. Halter, "A Multi-GPU Acceleration for 3D Imaging of the Prostate," *Proc. Int'l Conf. Electromagnetics in Advanced Applications (ICEAA)*, pp. 1096-1099, 2011.

[8] B. Jang, D. Kaeli, S. Do, and H. Pien, "Multi GPU Implementation of Iterative Tomographic Reconstruction Algorithms," *Proc. IEEE Sixth Int'l Conf. Symp. Biomedical Imaging: From Nano to Macro (ISBI '09)*, pp. 185-188, http://dl.acm.org/citation.cfm?id=1699872.1699919, 2009.

[9] M. Strengert, M. Magallón, D. Weiskopf, S. Guthe, and T. Ertl, "Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters," *Proc. Eurographics Symp. Parallel Graphics and Visualization (EGPGV '04)*, pp. 41-48, 2004.

[10] O. Villa, L. Chen, and S. Krishnamoorthy, "High Performance Molecular Dynamic Simulation on Single and Multi-GPU Systems," *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS)*, pp. 3805-3808, 2010.

[11] B.G. Levine, J.E. Stone, and A. Kohlmeyer, "Fast Analysis of Molecular Dynamics Trajectories with Graphics Processing Units-Radial Distribution Function Histogramming," *J. Computational Physics*, vol. 230, pp. 3556-3569, http://dx.doi.org/10.1016/j.jcp.2011.01.048, May 2011.

[12] J.C. Phillips, J.E. Stone, and K. Schulten, "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," *Proc. ACM/IEEE Conf. Supercomputing (SC '08)*, pp. 8:1-8:9, http://dl.acm.org/citation.cfm?id=1413370.1413379, 2008.

[13] R. Babich, M.A. Clark, and B. Joó, "Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics," *Proc. ACM/IEEE Int'l Conf. High Performance Computing, Networking, Storage, and Analysis (SC '10)*, pp. 1-11, http://dx.doi.org/10.1109/SC.2010.40, 2010.

[14] P. Richmond, D. Walker, S. Coakley, and D. Romano, "High Performance Cellular Level Agent-Based Simulation with FLAME for the GPU," *Briefings in Bioinformatics*, vol. 11, no. 3, pp. 334-347, http://bib.oxfordjournals.org/content/11/3/334.abstract, 2010.

[15] Y. Zhou, S. Song, T. Dong, and D.A. Yuen, "Seismic Wave Propagation Simulation Using Accelerated Support Operator Rupture Dynamics on Multi-GPU," *Proc. IEEE 14th Int'l Conf. Computational Science and Eng.*, pp. 567-572, 2011.

[16] P. Vidal and E. Alba, "A Multi-GPU Implementation of a Cellular Genetic Algorithm," *Proc. IEEE Congress Evolutionary Computation*, pp. 1-7, http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5586530, 2010.

[17] C. Obrecht, F. Kuznik, B. Tourancheau, and J.J. Roux, "Multi-GPU Implementation of the Lattice Boltzmann Method," *Computers and Math. with Applications*, vol. 65, pp. 252-261, http://dx.doi.org/10.1016/j.camwa.2011.02.020, 2011.

[18] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, "Dynamic Load Balancing on Single- and Multi-GPU Systems," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS)*, pp. 1-12, 2010.

[19] R.A. Gingold and J.J. Monaghan, "Smoothed Particle Hydrodynamics - Theory and Application to Non-Spherical Stars," *Monthly Notices of the Royal Astronomical Soc.*, vol. 181, pp. 375-389, Nov. 1977.

[20] J.J. Monaghan, "Smoothed Particle Hydrodynamics," *Ann. Rev. of Astronomy and Astrophysics*, vol. 30, pp. 543-574, 1977.

[21] J.J. Monaghan, "Smoothed Particle Hydrodynamics," *Reports on Progress in Physics*, vol. 68, no. 8, pp. 1703-1579, http://stacks.iop.org/0034-4885/68/i=8/a=R01, 2005.

[22] M. Gomez-Gesteira, B.D. Rogers, R.A. Dalrymple, and A.J. Crespo, "State-of-the-Art of Classical SPH for Free-Surface Flows," *J. Hydraulic Research*, vol. 48, no. sup1, pp. 6-27, http://www.tandfonline.com/doi/abs/10.1080/00221686.2010.9641242, 2010.

[23] T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara, "Particle-Based Fluid Simulation on GPU," *Proc. ACM Workshop General-Purpose Computing on Graphics Processors and SIGGRAPH Poster Session*, 2004.

[24] A. Kolb and N. Cuntz, "Dynamic Particle Coupling for GPU-Based Fluid Simulation," *Proc. 18th Symp. Simulation Technique*, pp. 722-727, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.2285, 2005.

[25] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed Particle Hydrodynamics on GPUs," *Proc. Computer Graphics Int'l Conf.*, pp. 63-70, 2007.

[26] A. Hérault, G. Bilotta, and R.A. Dalrymple, "SPH on GPU with CUDA," *J. Hydraulic Research*, vol. 48, pp. 74-79, 2010.

[27] M. Gómez-Gesteira, B. Rogers, R. Dalrymple, A. Crespo, and M. Narayanaswamy, User Guide for the SPHysics Code v1.2, 2007.

[28] A. Hérault, G. Bilotta, R. Dalrymple, E. Rustico, and C. Del Negro GPU-SPH, http://www.ce.jhu.edu/dalrymple/GPU/GPUSPH/Home.html, 2013.

[29] A. Hérault, A. Vicari, C. Del Negro, and R. Dalrymple, "Modeling Water Waves in the Surf Zone with GPU-SPHysics," *Proc. Fourth Workshop, SPHERIC, ERCOFTAC, Nantes*, 2009.

[30] R. Dalrymple and A. Hérault, "Levee Breaching with GPU-SPHysics Code," *Proc. Fourth Workshop, SPHERIC, ERCOFTAC, Nantes*, 2009.

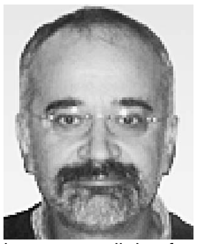
[31] R. Dalrymple, A. Hérault, G. Bilotta, and R.J. Farahani, “GPU-Accelerated SPH Model for Water Waves and Other Free Surface Flows,” *Proc. 31st Int'l Conf. Coastal Eng.*, 2010.
[32] G. Bilotta, A. Hérault, C. Del Negro, G. Russo, and A. Vicari, “Complex Fluid Flow Modeling with SPH on GPU,” *EGU General Assembly*, vol. 12, p. 12233, May 2010.
[33] A. Hérault, G. Bilotta, C. Del Negro, G. Russo, and A. Vicari, *SPH Modeling of Lava Flows with GPU Implementation,* World Scientific Series on Nonlinear Science, Series B, vol. 15, pp. 183-188, World Scientific Publishing Company, 2010.
[34] A. Hérault, G. Bilotta, A. Vicari, E. Rustico, and C. Del Negro, “Numerical Simulation of Lava Flow Using a GPU SPH Model,” *Annals of Geophysics,* vol. 54, no. 5, pp. 600-620, 2011.
[35] P. Goswami, P. Schlegel, B. Solenthaler, and R. Pajarola, “Interactive SPH Simulation and Rendering on the GPU,” *Proc. ACM SIGGRAPH/Eurographics Symp. Computer Animation (SCA '10),* pp. 55-64, http://dl.acm.org/citation. cfm?id=1921427.1921437, 2010.
[36] A. Crespo, J. Domínguez, M. Gómez-Gesteira, A. Barreiro, and B. Rogers User Guide for DualSPHysics Code v2.0, 2012.
[37] E. Rustico, G. Bilotta, A. Hérault, C. Del Negro, and G. Gallo, “Smoothed Particle Hydrodynamics Simulations on Multi-GPU Systems,” *Proc. Int'l Euromicro Conf. Parallel, Distributed and Network-Based Processing,* pp. 543-574, Feb. 2012.
[38] S. Green Particle Simulation Using CUDA, http://developer. download.nvidia.com/compute/DevZone/C/html/C/src/parti cles/doc/particles.pdf, 2010.
[39] E. Rustico, G. Bilotta, A. Hérault, C. Del Negro, and G. Gallo, “Scalable Multi-GPU Implementation of Cellular Automata Based Lava Simulations,” *Annals of Geophysics,* vol. 54, no. 5, pp. 592-599, 2011.
[40] G.M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” *Proc. Spring Joint Computer Conf. (AFIPS '67),* pp. 483-485, http://doi.acm.org/10.1145/1465482.1465560, 1967.
[41] J.L. Gustafson, “Reevaluating Amdahl's Law,” *Comm. ACM,* vol. 31, pp. 532-533, 1988.
[42] A.H. Karp and H.P. Flatt, “Measuring Parallel Processor Performance,” *Comm. ACM,* vol. 33, pp. 539-543, http://doi.acm.org/10.1145/78607.78614, May 1990.

Eugenio Rustico received the graduate degree in computer science in 2008 and was nominated for the Archimede Prize at Università di Catania, where he finalized the PhD thesis on multi-GPU fluid dynamics simulations. He had an intense collaboration with the Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Catania. He was also guest at the Université de Marn-la-Vallée, Paris, France, Conservatoire National des Arts et Métiers, Paris, France and Johns Hopkins University, Baltimore, US, where he developed, respectively, a multi-GPU version of the cellular-automaton-based MAGFLOW lava flow simulator and a multi-GPU version of the SPH-based fluid simulator GPUSPH.

Giuseppe Bilotta received the graduate degree in mathematics from the Università di Catania, Italy, in 2001 and the PhD degree in numerical analysis on approximation methods for Bézier curves from the same university. He is currently working as a postdoctoral researcher on a joint project involving the Università di Catania and the Istituto Nazionale di Geofisica e Vulcanologia (INGV). During the course of the project, he developed a CUDA implementation of the MAGFLOW cellular automaton model for lava flow simulations and contributed to GPUSPH, an open-source CUDA implementation of the SPH numerical method for the Navier-Stokes equations, jointly developed by the Università di Catania, the INGV and the Johns Hopkins University in Baltimore, Maryland.

Alexis Hérault received the graduate degrees in fluid mechanics and in mathematics from the Institut National Polytechnique de Grenoble (INPG), France, in 1991 and 1995, respectively, and the PhD degree in scientific information from the Université de Marne-la-Vallée, France, in 2008 on lava flow simulation. He is one of the contributors of GPUSPH, an open source CUDA implementation of Smoothed Particle Hydrodynamics method. Currently, he is an associate professor at Conservatoire National des Arts et Métiers, Paris, and an associate researcher at the Istituto Nazionale di Geofisica e Vulcanologia, Osservattorio Etneo, Catania, Italy.

Ciro Del Negro is a research director at the Istituto Nazionale di Geofisica e Vulcanologia, Italy and head of the Gravity and Magnetism Research Group of the Section of Catania. His research interests include the multidisciplinary modeling of magma dynamic processes occurring at different time scales via integrated analysis and joint inversion of gravimetric, magnetic and ground deformation data; spaceborne remote sensing techniques for the thermal monitoring of volcanic activity; numerical simulations of the spatial; and temporal evolution of eruptive phenomena for volcanic hazard assessment.



Giovanni Gallo received the graduate degree in mathematics from à di Catania, Dipartimento di Matematica e Informatica, Catania University, in 1986 and the PhD degree in computer science from New York University, in 1992, with a dissertation about the complexity of computer algebra algorithms. Since 1992, he has been a professor at Catania University and became a full professor and the chair of computer science studies in Catania in 2001. Since 2004, he has been an adjoint faculty at the Accademia di Belle Arti, Catania. His research interests include the field of digital imaging, medical imaging, geographical information systems, and computer graphics. He is a member of the IEEE.