# A journey from single-GPU to optimized multi-GPU SPH with CUDA

E. Rustico, G. Bilotta, G. Gallo
Dipartimento di Matematica e Informatica
Università di Catania
Catania, Italy
{rustico,bilotta,gallo}@dmi.unict.it

A. Hérault
Conservatoire des Arts et Métiers
Département Ingénierie Mathématique
Paris, France
alexis.herault@cnam.fr

C. Del Negro
Istituto Nazionale di Geofisica e Vulcanologia
Sezione di Catania
Catania, Italy
ciro.delnegro@ct.ingv.it

R. A. Dalrymple
Department of Civil Engineering
Johns Hopkins University
Baltimore, MD, USA
rad@jhu.edu

***Abstract*—We present an optimized multi-GPU version of GPUSPH, a CUDA implementation of fluid-dynamics models based on the Smoothed Particle Hydrodynamics (SPH) numerical method. 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 exploit 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 overlap 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 closely follows the ideal one and it is possible to run simulations with a higher number of particles than would fit on a single device. efficiency of the parallelization.**

## I. 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 (gasses, water, mud, oil, petrol, lava, etc). The Smoothed Particle Hydrodynamics (SPH) model, initial developed by Ginghold and Monaghan [1] and Lucy [2], has seen a growing interest in recent years, thanks to its flexibility and the possibility of application to a wide variety of problems.

The flexibility of SPH comes at the cost of higher computational costs compared to other methods (e.g. mesh methods like finite differences or finite volumes). However, since it exposes a high degree of parallelism, its implementation on parallel high-performance computing (HPC) platforms is conceptually straightforward, significantly reducing execution times for simulations.

Among the many possible parallel HPC solutions, an approach that has emerged lately is the use of GPUs (Graphic Processing Units), hardware initially developed for fast rendering of dynamic three-dimensional scenes, as numerical processor for computationally-intensive, highly parallel tasks.

Although initial attempts to exploit the computational power of GPUs go back to the introduction of the first programmable shaders in 2001, the break-through for GPGPU (General-purpose Programming on GPU) was the introduction in 2007 of CUDA, a hardware and software architecture released by NVIDIA with explicit support for computing on GPUs [3].

While typically running a lower clock rates, a single GPU features a large number of compute units (for more recent cards, in the order of thousands of cores per GPU) and much higher memory bandwidth than what is found on standard desktop or server motherboards.

Although serial execution does not gain much from GPU execution, its large multi-core structure makes it the ideal computing platform for algorithms that exhibit a high level of parallelism on a fine data granularity, such as SPH. For such problems, a well-tuned GPU implementation can easily achieve two orders of magnitude in speed-up of standard single-core CPU implementations.

The cost-effectiveness and the ease of utilization of modern GPUs have led to a widespread usage of GPU computing even outside the commercial and academic world, leading to what some to claim to be the *GPU Computing Era* [4]. It should be mentioned, however, that some have criticized the enthusiasm for GPGPU as being 'excessive', showing that a well-tuned CPU implementation, optimized for execution on recent multi-core processors, often reduces the flaunted $100\times$ speedup reported by many works [5].

We present an optimized, multi-GPU implementation of the SPH method. Our work is based on the open-source GPUSPH code, the first CUDA implementation of SPH, developed in recent years by the Sezione di Catania of the Istituto Nazionale di Geofisica e Vulcanologia (INGV-CT) in cooperation with the Department of Mathematics and Computer Science of the University of Catania and the Department of Civil Engineering of the Johns Hopkins University [6].

We extend GPUSPH to allow the distribution of the computational workload across multiple GPUs connected to the same host machine. The use of more than one GPU allows large simulations to complete in shorter times, roughly proportional to the number of devices used, as well as to simulate problems that would be too large to fit on a single device.

The single-GPU SPH model is briefly introduced in section III, followed by the new multi-GPUs version, with a description of the strategies employed to cover the latency of cross-device data transfer and to balance the computational load across devices. Optimization of the implementation for a single device are then discussed, followed by some results illustrating the benefits of multiple device usage, including very high-resolution SPH simulations.

## II. Related work

The first GPU-assisted implementation of the SPH method was developed by Amada [7], which off-loaded the force computation to the GPU while the CPU was delegated with tasks such as neighbor search. A pure GPU implementation (with no computing assistance from the host CPU) was later developed by Kolb and Cuntz [8] and Harada et al. [9].

The introduction of the CUDA architecture for NVIDIA cards in 2007 allowed the computational power of modern GPUs to be fully exploited without the limitations imposed by having to go through the graphical engines. The first CUDA implementation of the SPH method was developed by the authors [6]. Inspired by the open-source Fortran SPHysics code [10], it has been recently published as the open-source project GPUSPH [11]. Another CUDA implementation of SPH became open source in march 2012 [12]. The same authors presented last year a multi-GPU version [13], at the moment not open source and without load balancing.

GPUSPH itself has found a number of applications ranging from coastal engineering [14]–[16] to lava flow simulation, for which a specialized version with support for non-Newtonian viscous fluids and temperature-dependent parameters has been developed [17]–[19].

## III. Single-GPU SPH

We now present an overview of GPUSPH. The overall structure of the single-GPU implementation is essential to understand the challenges posed by the multi-GPU implementation presented later. A number of improvements over the original implementation described in [6] are also discussed here.

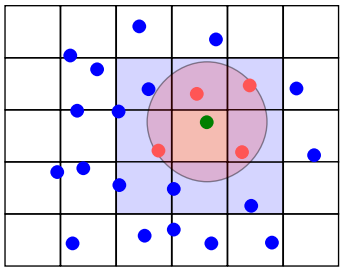

Fig. 1. If cells have side equal to the influence radius, neighbors of the green particle must reside inside in the immediate neighbor cells (light blue).

### A. Kernels

The organization of the SPH method in CUDA kernels directly reflects the computing phases of the SPH model:

1) **`BuildNeibs`** - For each particle, build the list of neighbors;
2) **`Forces`** - For each particle, compute the interaction with neighbors;
3) **`MinimumScan`** - Select the minimum `dt` among the maxima provided by each particle;
4) **`Euler`** - For each particle, update the scalar properties integrating over selected `dt`.

The kernel for force computation(`Forces`) and the kernel for integration (`Euler`) are actually run twice because of the two-step integrator scheme. All kernels have been written from scratch, except for the particle sorting during the neighbor search phase and the minimum scan, for which standard libraries provided by NVIDIA are used.

### B. Fast neighbor search

As with most implementations of the SPH method, a neighbor list is constructed and maintained through several iterations of the simulation, to speed up the neighbor search in phases such as force computation or other optional corrections.

To speed up the construction of the neighbor list itself, the computational domain is partitioned with a regular grid with cell size equal to the SPH influence radius (i.e. kernel radius time smoothing length); an example of such a construction in two dimensions is shown in fig. 1.

The particles are then indexed by the cell they fall in and sorted according to their cell index. This allows the construction of the neighbor list for each particle to be built by only looking at the particles in the 27 cells (in three dimensions) closest to the particle location.

The auxiliary grid for the neighbor list construction is not used during computation but, as we will discuss in section IV, it is also of assistance in the domain partitioning for the multi-GPU implementation.

Sorting the particles and building the neighbor list still accounts for about $50\%$ of the computational time for a single integration step. Hence, the neighbor list is only updated every $k$ iteration, with $k$ being a parameter that can be set by the user, with a default of $k = 10$. This reduces the time spent

in the neighbor list construction to about $10\%$ of the total computational time for a simulation.

## IV. From single- to multi- GPU

Exploiting a second level of parallelism required some structural changes to the single-GPU GPUSPH code. The CPU code needed a complete re-engineering, while the GPU kernels underwent only minor changes. Some minor features were temporarily disabled for testing purposes, such as periodic boundaries, and will be enabled again soon. Here follows an overview of the challenges we had to overcome on the model side and on the technical side.

### A. Splitting the problem

The key idea for exploiting multiple GPUs for the same simulation is that the total computational burden must be fairly split among the devices. The way the problem is split and the path the data has to follow depends on the nature of the problem. There could be no unique optimal solution for a problem, as different splits may perform differently according to the characteristics of a specific problem *instance*.

SPH is a purely parallel method except for the fact that every particle needs to interact with its neighbors. This locality constraint, together with the need to search for a globally minimum timestep, required a special consideration when designing the multi-GPU version of the simulator.

There are different ways to split a SPH simulation across multiple GPUs. A first possibility could be to split the problem in the domain of the computations: we could assign each phase of the computation to a different device, thus arranging a *pipeline*. This method, however, still needs the entire set of particles to be transferred at every iteration across all the devices, and does not scale easily as the number of devices increases. Another possibility comes naturally from the fact that particles are arranged in a *list*. The list enumerates the particles regardless of their spatial position: list locality does not correspond to spatial proximity. We could think of splitting the list in subsets and assign each subset to a different device; unfortunately, this is not feasible because we have no guarantee that the neighbors of a particle reside on the same device and accessing single particles in separate devices is very costly. Although SPH is a meshless method, we have seen in section III-B that particles are sorted and indexed by means of a grid of virtual cells of the same size of the influence radius to speedup the neighbor search. We can exploit this ordering to split the fluid on a spatial basis and handle a minimum overlapping of subdomains needed by the locality requirement. This is actually an extension of the previously described list-split that takes into account a pre-existent order constraint. We choose the spatial split because of its simplicity, scalability and robustness.

### B. Split planes

While a spatial split in theory could operate on any three-dimensional plane, we focus on the cartesian ones (the planes orthogonal with the cartesian axes), as the split is based on the cubic cells used for fast neighbor search. Although it could be technically possible to split along different planes simultaneously, the transfer of the edge of a subdomain could be very expensive. Fig. 2 illustrates the problem in a simple two-dimensional case. Assuming that the domain is linearized in a row-first fashion, it is possible to transfer every green edge by requesting a single memory operation, while the red edges require many small transactions. As a general rule, it is recommended that all the split planes are aligned to the most significant axis used for the linearization. Thus, we chose not to allow splits along different planes within the same simulation: the 3D domain is split in slices all parallel with each other.

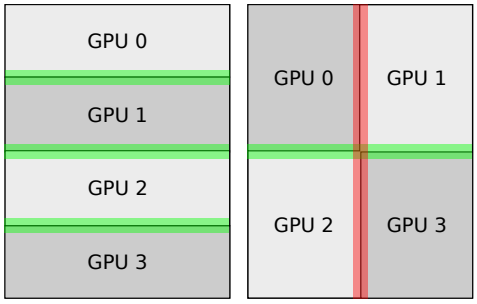


Fig. 2. Splitting a 2D domain along the same axis or along orthogonal axes. Assuming the data are linearized per row, it is possible to copy each of the green edges in one memory transfer, while the red edges require many small transactions.

It is still possible to choose a different split plane for each simulation at compile time. This is implemented by defining three different compiler macros and letting the programmer choose the most appropriate one for the given problem; the way cells are enumerated and 3D grid is linearized changes accordingly. As a rule of thumb, one should choose the split that minimizes the number of particles per slices (i.e. the sections of the fluid being cut). For most problems the choice has no big consequences as the time required for transferring an overlapping slice, whichever the plane, is completely covered by the force computation, as will be shown later. Very asymmetric topologies, however, may benefit from a proper cut when load balancing, as the balancing granularity is at the slice level and the balancing operations are not covered (see section IV-G).

### C. Subdomain overlap

Many particles residing near the edge of a subdomain have neighbors in the edging devices. To access all their neighbors without generating too many small memory transfers, each device needs a copy of the first slice of neighboring devices as read-only information for the interaction with neighbors, and sends its edging slices as read-only copy to neighboring devices. Recalling that the split is done by means of the grid of virtual cells, and that cells have size equal to the influence radius, the total overlap between neighboring subdomains is exactly wide as twice as the influence radius.

We refer to the assigned read/write particles as *internal* ones and to the read-only particles as *external*. When referring to slices, *internal* and *external* usually implies *edging* (i.e. first or last slice of the device subdomain).

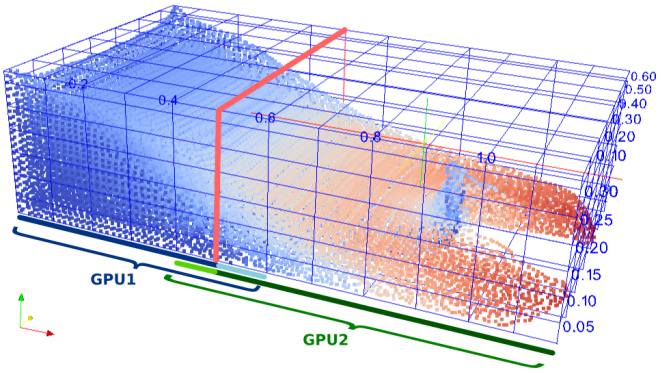


Fig. 3. Fluid volume with virtual cells, split on YZ plane. The blue line shows the internal cells of GPU n.1; light blue the external read-only slice updated by GPU n.2; internal cells of GPU n.2 are green, while the external slice is light green. Particles are colored by velocity and cells are not in scale for visualization purposes.

Figure 3 represents one possible split of a simulation domain. The subdomains assigned to two devices and their overlaps are highlighted. The position of the particles is taken from an actual simulation and particles are colored by velocity. The blue spot in device n.2 is due to the impact with an obstacle not drawn for visualization purposes. Fig. 4 represents the same subdivision from the viewpoint of the list of particles.

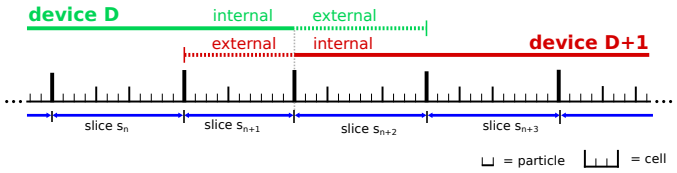


Fig. 4. Representation of the the list split technique. Once particles are sorted by cells and 3D cells are linearized, it is possible to split the 3D domain by splitting the list of particles in specific addresses where 3D slices begin.

### D. Kernels

We now discuss about the kernels and what changes were necessary to pass from one GPU to multiple GPUs.

Working on a subset of the global domain is trivial for step 1 (neighbor search) and 4 (integration) of the model, but we have to be careful only in running them on the appropriate particles subset. In particular, the neighbor search reads all internal and external particles but produces the neighbor list of internal particles only. The integration, instead, is run on all particles, as the external ones are exchanged as forces, as later explained, and new positions have to be computed. Because we run the integration also on the external particles, in the whole multi-GPU simulation the overlapping slices are integrated twice. Fortunately, this overhead is really narrow, as the integration step barely saturates the GPUs.

Step 3 (minimum scan) is straightforwardly extended to $n$ GPUs: each device finds its local minimum and sends it to the CPU, which quickly compares the few local minima and finds the global one. A CPU and GPU barrier waits for all the local minima to be ready. Step 2 requires a further remark. The neighbors of the external particles may not be available, as the next 3D slice reside on a different device. Therefore, any force computed on an external particle would be *partial* and should not contribute to the computation of the final `dt`. We therefore run the force computation kernel only on the internal particles; external ones are accessed only as neighbors of internal ones.

### E. Hiding slice transfers

Each device needs an updated copy of the neighboring slices at each iteration. More specifically, we need updated positions and velocities of particles in neighboring slices each time forces are computed and integrated; due to the predictor-corrector integration scheme, this update has to be done twice for every iteration. This introduces a conspicuous overhead. The contribute to the whole simulation time greatly varies according to the density and topology of the simulated fluid. In some cases it can even make a multi-GPU simulation perform worse than a single-GPU one.

To overcome this problem we exploit the hardware capability of performing concurrent computations and data transfers, treasuring the experience we maturated with a different, Cellular-Automaton based simulator [20]. 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; we will use two of them to enqueue operations about the edging borders and one for the remaining slices. Except for the first and the last device, which only have one neighboring device and thus one edging slice, all devices have two edging slices. We will describe the behavior of a device with two neighbors, as the devices with only one neighbor are just a simpler case. A first, simple design would be to issue first a `Forces` kernel on edging slices; download the edges as soon as they are ready, while computing the forces on the remaining slices; run the `MinimumScan`; run `Euler` kernel on the non-edging slices while uploading the updated external edges; finally, integrate also the edging slices as soon as the uploads are complete.

Another possibility could be to exchange the *forces* of the overlapping slices instead of the integrated positions and velocities. This would have two major advantages:

1) We can start uploading the external slices while the forces on the non-edging slices are still being computed; because the `Forces` kernel takes longer than `Euler`, transfers are more likely to be completely hidden.
2) We need to transfer less data (forces instead of positions and velocities). This is true unless we need additional structures, such as $\tau$ coefficients for SPS correction [21].

As already mentioned, this comes at the small price of running the integration kernel also on external particles. We chose the latter approach, that is represented in fig. 5. The minimum scan and the integration are not encapsulated in the method anymore, as all asynchronous transfers finish before the `Forces` kernel does.

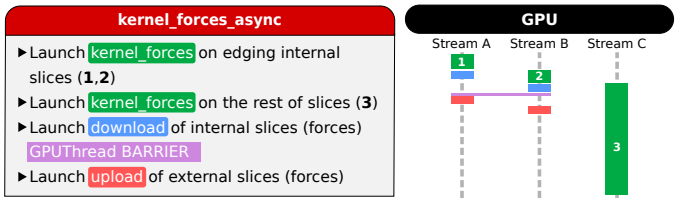


Fig. 5. Final design of `kernel_forces_async` method. Note that the forces are exchanged instead of the positions; as a consequence, `Euler` kernel must be run also on external particles.

Fig. 6 shows the actual sequence of events as profiled during the simulation of a `DamBreak3D` with 1 million particles on 3 GPUs. The exchange of slices (purple) is actually performed concurrently with computation of forces, and effectively starts as soon as the computation of forces on edges is completed. While the rectangles plot the start time and duration of the events on the device, the little dots above them mark the timestamps of the same operations as issued on the host. All operations are asynchronous to the host except for the download of the `dt`, which is blocking for the time represented by the long brown line. We produced fig. 6 with a custom profiler-visualizer we developed ad hoc to overcome the limits of the standard profilers provided with CUDA 3.2 and CUDA 4.0.

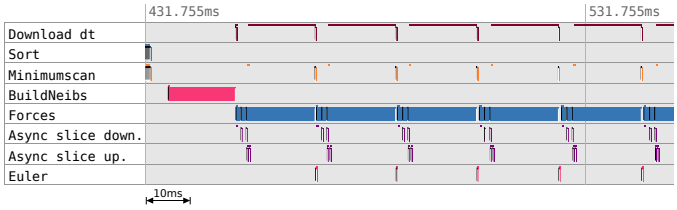


Fig. 6. Actual timeline of a multi-GPU GPUSPH simulation, with kernel lengths in scale (lengths smaller than 1 pixel have been rounded up). Only one GPU with 2 neighboring devices is shown. The little dots mark the moment operations were issued on the CPU. Downloading the `dt` is blocking for the CPU (from which, the long brown line) until the minimum scan is complete.

### F. Simulator design

We encapsulate all pointers and CUDA calls needed to handle a GPU into the class `GPUThread` (not to be confused with the homonymous GPU instances of a kernel). The main thread allocates one `GPUThread` per device and each `GPUThread` starts a dedicated `pthread` in constant communication with the associated device. The main thread tracks the simulation time and periodically requests a subdomain dump from the GPUs according to a user-defined save frequency. We synchronize the threads through a GPU flush (`cudaThreadSynchronize`) and a signal/wait CPU barrier based on the NPTL implementation of POSIX threads.

It is possible to specify at command line several simulation options and it is possible to run a simulation on a heterogeneous set of devices, even belonging to different hardware generations. The data is not multi-GPU aware: it is possible to save the state of a single-GPU simulation and to restore it in a multi-GPU environment, and vice-versa.

### G. 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 speedup 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 the overall performance loss will be proportional to the performance of the worst performing device. 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 $n\%$.

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. It is advisable to implement an *a posteriori* load balancing technique.

The key idea is 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 $A_{\delta g} = A_g - A_G$.

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. $A_g$ is then reset to wait for next $k$ iterations.

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 $T_{slice}$ 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*. In our tests $H_{LB} = 0.5$ performed quite well in the general case. A manual fine-tuning in specific simulations may lead to slightly better results, although in our tests fine-tuning $H_{LB}$ only resulted in negligible performance improvements (the order of tens of seconds with respect to one hour of simulation).

Figure 7 shows different snapshots of a `BoreInABox` simulation with about 1.1 million particles. Each particle is colored according to the device it belongs to, so that the dynamics of balancing are highlighted.

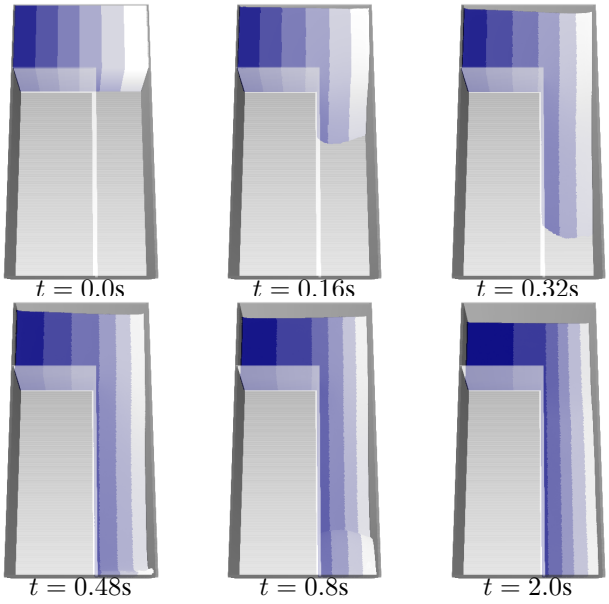


Fig. 7. Snapshots of different phases of the `BoreInABox` problem simulation, corridor variant, on 6 GPUs. Each particle is colored according to the device number it belongs to.

## V. Results

### A. Performance metrics

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.

We propose to measure the amount of work done within a simulation as the number of fulfilled iterations times the number of particles. This can be considered as the number of single iterations completed (imagining, for example, a hypothetical single-core GPU). To abstract from the simulation length we can simply divide by the real time. We chose seconds as time units and, as the number of particles often exceeds the million, we found useful to consider thousands of them. We are thus measuring *thousands of iterations on particles per second*; in short form, we call this unit *kip/s* or simply *kips*. Another advantage of this metric compared to the mere speedup is that it is possible to compute the instantaneous speed at runtime, with no need to wait for a simulation to complete. This metric is specific for particle methods and may not be suitable for other models.

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| DamBreak3D | | | | | | |
| LB off | 9,977 | 19,149 | 27,802 | 35,213 | 39,784 | 45,599 |
| LB on | - | 19,380 | 27,191 | 35,336 | 42,578 | 49,491 |
| Ideal | - | 19,955 | 29,932 | 39,910 | 49,887 | 59,865 |
| BoreInABox | | | | | | |
| LB off | 8,770 | 12,713 | 16,275 | 20,649 | 25,657 | 29,418 |
| LB on | - | 16,940 | 24,115 | 30,548 | 36,800 | 41,745 |
| Ideal | - | 17,541 | 26,311 | 35,082 | 43,852 | 52,623 |

TABLE I
KIP/S, WITHOUT AND WITH LOAD BALANCING (LB), `DamBreak3D` AND `BoreInABox`, 1.6 MILLION PARTICLES.

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.

### B. Test platform

Our testing platform is a TYAN-FT72 rack mounting 6×GTX480 cards on as many 2nd generation PCI-Express slots. The system is based on a dual-Xeon processor with 16 total cores (E5520 at 2.27GHz, 8MB cache) and 16GB RAM in dual channel. Each GTX480 has 480 CUDA cores grouped in 15 multiprocessors, 64kB shared memory/L1 cache per MP and 1.5GB global memory with a measured datarate of about 3.5GB/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).

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 $0.43m^3$ of water divided into 1.6 million particles for 1.5s of simulated time. An obstacle breaks the fluid in the first variant (DamBreak3D); two walls change the flow path in other two (BoreInABox "wall" and "corridor" versions).

The operating system is Ubuntu 10.04 x86_64, gcc 4.4.3, CUDA runtime 3.2 and NVIDIA video driver 285.05.09.

Videos of the reference simulation run on different number of devices are available at http://www.dmi.unict.it/~rustico/sphvideos.

### C. Analysis

The load balancing policy and algorithm we implemented is not perfect and, while it works reasonably well in practice, may not converge to the optimal balance in some classes of situations, remaining stuck in a local minima or in a "ping-pong" slice exchange. This has to be considered as a first attempt to overcome the technical difficulties arising from on-the-fly subdomain resizing and needs further improvements.

The effectiveness of the implemented load balancing policy has been tested by measuring the achieved performance during the simulation of two problems, `DamBreak3D` and the

BoreInABox, with about 1.6 million particles each, from 1 to 6 GPUs. Table I show the measured kips/s; charts 8 and 9 plot the execution times for visual comparison.

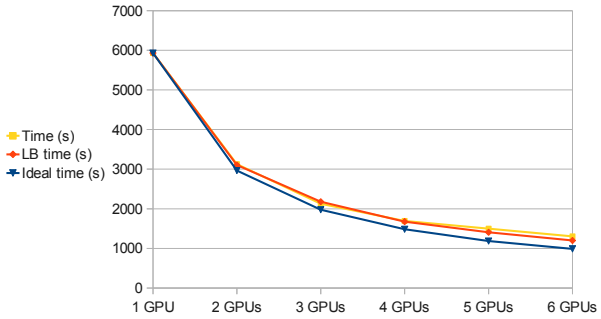


Fig. 8. Multi-GPU GPUSPH execution times for a simulation with 1.6 million particles, DamBreak3D problem, 1-6 GPUs.

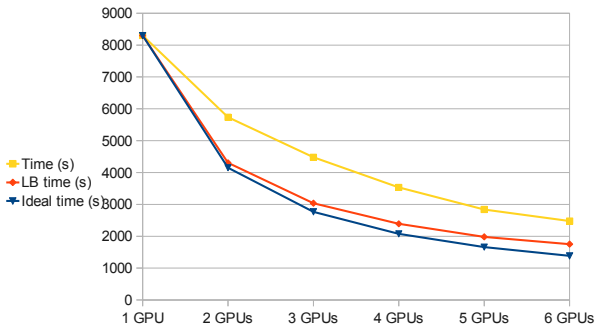


Fig. 9. Multi-GPU GPUSPH execution times for a simulation with 1.6 million particles, BoreInABox problem, 1-6 GPUs.

The DamBreak3D problems presents a high level of symmetry, especially when splitting the problem along the Y axis. The number of particles assigned to each GPU is roughly constant during the whole simulated time (1.5s) and the load balancing is not expected to make a big change. Simulating over 2 and 4 GPUs lead to an advantage of roughly 1% total simulation time; on 5 and 6 GPUs simulations run between 6% and 13% faster than without any balancing. Nevertheless, simulating on 3 GPUs with load balancing activated takes surprisingly about 1% *longer*. This is a consequence of the domain shape and distribution of particles. With 3 GPUs, two of them are assigned to lateral stripes (thus, have a higher percent of static particle-border interactions) while the central one deals with the obstacle. The naturally balanced particle distribution highlights the little overhead of balancing attempts (112 slices moved over about 35k iterations), without any performance gain.

When simulating an asymmetrical problem like BoreInABox, however, load balancing makes a big performance difference. During the 2.5s simulated time of BoreInABox, 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. In fig 9 it is possible to see how load balancing keeps the performance close to the ideal one, while without balancing the performance drops (or, execution time jumps up). In such cases, load balancing also allows for bigger simulations, as it is possible to reduce the allocation margin factor $M_f$ and save space for further particles.

## VI. Further optimizations

Aside from the development of a multi-GPU implementation, the single-GPU code was also improved by introducing a number of optimizations tuned for the new hardware produced by NVIDIA since the first version of GPUSPH.

### A. Interleaved neighbor list

In the first version of GPUSPH, the neighbor list was stored as a sequence of consecutive "buckets", such that all the neighbors of the first particles were stored consecutively, followed by all the neighbors of the second particle, and so on. This storage structure follows the standard approach used on CPU, as well as the approach used in the first GPU implementations, as it allows exploiting the texture caching feature presents in most GPUs.

As an optimization, the structure was replaced with an interleaved structure, grouping particles according to the block size used for kernel launches. For each group of particles with consecutive indices, the neighbor are stored by putting all the first neighbors first, followed by all the second neighbors, and so on. The neighbors of a single particle are thus stored with a stride $s$, which is also chosen to be a multiple of 32 (the hardware *warp size*) to improve memory alignment.

The strided neighbor list and the improved alignment ensure that neighbor list access are properly *coalesced*, a feature of GPU computing that significantly reduce memory latency by allowing the data of multiple particles to be loaded in a single memory transaction.

### B. Fermi L1 cache

While older cards only offered caching through the use of a special hardware function called *textures*, newer GPUs from NVIDIA, code-named Fermi, also feature an L1 cache that can further improve memory access times to the main GPU memory (global memory).

While all arrays were accessed as textures in the original version of GPUSPH, we now optimize memory access on Fermi cards by keeping all linear-access arrays in global memory, and distributing random-access arrays between texture and global memory. The calibration of the arrays to put in texture versus global memory was done by trial and error to find the optimal combination.

### C. Block size

The block size in CUDA represents the number of particles that are concurrently processed by a single multiprocessor of the device. In the original version of GPUSPH, the block size for force computation was limited to 64, with larger block

sizes offering no benefits due to the kernel runtimes being limited by memory accesses.

The optimization described in sections VI-A and VI-B greatly improved memory access, allowing us to raise the block size to 128 for force computations.

### D. Performance gain

The optimizations described in this section led to an implementation measured to be 2 to 3 times faster than the original GPUSPH code.

For example, in a `DamBreak` simulation with one million particles the neighbor list construction drops from about 52ms to about 34ms, while force computation exhibits a much higher improvement, from 51ms to about 19ms. By comparing the runtimes for 10 iterations (which include one `buildNeibs` and 20 force computations) we get 1106ms before optimization versus 446ms with optimizations, for an actual observed speed-up of $2.5\times$.

This performance gain has been measured mainly on the single-GPU implementation, but since the benefit independently affects the performance of each device, it carries over almost unchanged to the multi-GPU version.

## VII. Conclusions and future work

We presented a scalable multi-GPU implementation of the CUDA-based GPUSPH fluid simulator. Simulations scale almost linearly with the number of GPUs used. A dynamic *a posteriori* load balancing policy neutralizes the effect of asymmetries in the topology of the simulated fluid. Despite the simplicity of the balancing policy, involving no signal-processing or other advanced techniques, the system showed an excellent performance in almost all the tests we performed. A second aim of the multi-GPU implementation was also achieved, that is to run simulations with more particles than could fit in one device. Finally, we could almost halve the execution time of the neighbor search step by interleaving the lists of neighbors of different particles, leading to an improved coalescence in memory accesses.

The present simulator runs on a single node featuring multiple GPUs. The main improvement we are currently working on regards the development of a multi-node version of the simulator for GPU-based clusters. This also requires a more complex domain decomposition strategy and a more sophisticated load balancing policy, with an accurate temporal analysis and a possibly finer granularity.

## References


[1] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics - Theory and application to non-spherical stars," *Mon. Not. Roy. Astron. Soc.*, vol. 181, pp. 375–389, Nov. 1977.

[2] L. Lucy, "A numerical approach to the testing of fission hypothesis," *Astrophysical Journal*, no. 82, pp. 1013–1020, 1977.

[3] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, March 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[4] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, pp. 56–69, 2010.

[5] 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," *SIGARCH Comput. Archit. News*, vol. 38, pp. 451–460, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1816038.1816021

[6] A. Herault, G. Bilotta, and R. A. Dalrymple, "SPH on GPU with CUDA," *Journal of Hydraulic Research*, vol. 48, no. Extra Issue, pp. 74–79, 2010.

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

[8] A. Kolb and N. Cuntz, "Dynamic particle coupling for GPU-based fluid simulation," in *In Proc. of the 18th Symposium on Simulation Technique*, 2005, pp. 722–727. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.2285

[9] T. Harada, S. Koshizuka, and Y. Kawaguchi, "Smoothed particle hydrodynamics on GPUs," in *Computer Graphics International*, 2007, pp. 63–70.

[10] M. Gmez-Gesteira, B. Rogers, R. Dalrymple, A. Crespo, and M. Narayanaswamy, "User guide for the SPHysics code v1.2," 2007.

[11] A. Hérault, G. Bilotta, R. Dalrymple, E. Rustico, and C. D. Negro, "GPU-SPH," http://www.ce.jhu.edu/dalrymple/GPU/GPUSPH/Home.html. [Online]. Available: http://www.ce.jhu.edu/dalrymple/GPU/GPUSPH/Home.html

[12] A. C. Crespo, J. M. Dominguez, A. Barreiro, M. Gómez-Gesteira, and B. D. Rogers, "Gpus, a new tool of acceleration in cfd: Efficiency and reliability on smoothed particle hydrodynamics methods," *PLoS ONE*, vol. 6, no. 6, p. e20685, 06 2011. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0020685

[13] A. C. D. Valdez-Balderas, JM Dominguez and B. Rogers, "Develoing massively parallel SPH simulations on multi-GPU cluster," in *Proc. 6th International SPHERIC Workshop, Hamburg*, June 2011, pp. 340–347.

[14] A. Hérault, A. Vicari, C. Del Negro, and R. Dalrymple, "Modeling water waves in the surf zone with GPU-SPHysics," in *Proc. Fourth Workshop, SPHERIC, ERCOFTAC, Nantes*, 2009.

[15] R. Dalrymple and A. Hérault, "Levee breaching with GPU-SPHysics code," in *Proc. Fourth Workshop, SPHERIC, ERCOFTAC, Nantes*, 2009.

[16] R. Dalrymple, A. Hérault, G. Bilotta, and R. J. Farahani, "GPU-accelerated SPH model for water waves and other free surface flows," in *Proc. 31st International Conf. Coastal Engineering, Shanghai*, 2010.

[17] G. Bilotta, A. Hérault, C. Del Negro, G. Russo, and A. Vicari, "Complex fluid flow modeling with SPH on GPU," *EGU General Assembly 2010, held 2-7 May, 2010 in Vienna, Austria, p.12233*, vol. 12, pp. 12 233–+, May 2010.

[18] A. Hérault, G. Bilotta, C. Del Negro, G. Russo, and A. Vicari, *SPH modeling of lava flows with GPU implementation*, ser. World Scientific Series on Nonlinear Science, Series B. World Scientific Publishing Company, 2010, vol. 15, pp. 183–188.

[19] 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, 2011, accepted.

[20] 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, 2011, accepted.

[21] B. Rogers and R. Dalrymple, "Three-dimensional SPH-SPS modeling of wave breaking," in *Symposium on Ocean Wave Measurements and Analysis (ASCE)*, Madrid, 2005.