HENRY
Hydraulic Engineering Repository

Ein Service der Bundesanstalt für Wasserbau

Conference Paper, Published Version

Rustico, Eugenio; Jankowski, Jacek; Hérault, Alexis; Bilotta, Giuseppe; Del Negro, Ciro

Multi-GPU, multi-node SPH implementation with arbitrary domain decomposition

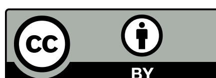
Verfügbar unter/Available at: https://hdl.handle.net/20.500.11970/100905

Vorgeschlagene Zitierweise/Suggested citation:
Rustico, Eugenio; Jankowski, Jacek; Hérault, Alexis; Bilotta, Giuseppe; Del Negro, Ciro (2014): Multi-GPU, multi-node SPH implementation with arbitrary domain decomposition. In: 9th International SPHERIC Workshop, Paris, France, June, 03-05 2014. Chatou: EDF.

Standardnutzungsbedingungen/Terms of Use:

Die Dokumente in HENRY stehen unter der Creative Commons Lizenz CC BY 4.0, sofern keine abweichenden Nutzungsbedingungen getroffen wurden. Damit ist sowohl die kommerzielle Nutzung als auch das Teilen, die Weiterbearbeitung und Speicherung erlaubt. Das Verwenden und das Bearbeiten stehen unter der Bedingung der Namensnennung. Im Einzelfall kann eine restriktivere Lizenz gelten; dann gelten abweichend von den obigen Nutzungsbedingungen die in der dort genannten Lizenz gewährten Nutzungsrechte.

Documents in HENRY are made available under the Creative Commons License CC BY 4.0, if no other license is applicable. Under CC BY 4.0 commercial use and sharing, remixing, transforming, and building upon the material of the work is permitted. In some cases a different, more restrictive license may apply; if applicable the terms of the restrictive license will be binding.



Multi-GPU, multi-node SPH implementation with arbitrary domain decomposition

Eugenio RUSTICO*,
Jacek JANKOWSKI
Bundesanstalt Für Wasserbau
Karlsruhe, Germany
eugenio.rustico@baw.de

Alexis HÉRAULT
Conservatoire National
des Arts et Métiers
Paris, France
& Sezione di Catania
Istituto Nazionale
di Geofisica e Vulcanologia
Catania, Italy

Giuseppe BILOTTA,
Ciro DEL NEGRO
Sezione di Catania
Istituto Nazionale
di Geofisica e Vulcanologia
Catania, Italy

***Abstract*—We present a restructured version of GPUSPH [4], [8], [11], a CUDA-based implementation of SPH. The new version is extended to allow execution on multiple GPUs on one or more host nodes, making it possible to concurrently exploit hundreds of devices across a network, allowing the simulation on larger domains and at higher resolutions. Partitioning of the computational domain is not limited anymore to parallel planes and can follow arbitrary, user-defined shapes at the resolution of individual cells, where the cell is defined by the auxiliary grid used for fast neighbor search. This allows optimal partitioning even in the case of complex domains, such as rivers with U-turns. The version we present also includes many additional features that have been developed on GPUSPH. Particularly important are: the uniform precision work by Hérault *et al.* [13], which is essential for numerical robustness in the case of very large ratios between the domain size and particle resolution; a compact neighbor list, which allows larger subdomains to be loaded on each device; the semi-analytical boundary conditions by Ferrand *et al.* [9], [12], and support for floating objects [14]. All of these features are seamlessly supported in single-GPU, multi-GPU and multi-node modes.**

I. INTRODUCTION

From a computational perspective, one of the most important benefits of the weakly-compressible Smoothed Particle Hydrodynamics (SPH) numerical method is its embarrassingly parallel nature. This has led to a number of implementations of SPH for high-performance parallel computing platforms, most recently focusing largely on Graphic Processing Units (GPUs) [4], [5] and similar architectures, which have shown to be very efficient alternatives to traditional CPU clusters both in terms of performance/price and in terms of performance/ power consumption ratios.

GPUs hold tens of compute units with hundreds of processing elements which operate in parallel to concurrently execute a large number of instances of *computing kernels*, the equivalent of standard CPU functions, in a shared-memory model (all processing elements can access the same global memory), allowing a well-tuned GPU program to easily perform $100\times$ faster than equivalent serial CPU implementations.

Limitations in GPU usage are largely determined by memory occupation, since even the most expensive, compute-dedicated GPUs are currently limited to 6GB of RAM for a single device, almost an order of magnitude less than the amount of memory that can be found on a high-end workstation. To circumvent this limitation, and to further reduce runtime, a second level of parallelism needs to be introduced, by using multiple GPUs connected to the same host CPU.

Multi-GPU usage in SPH has been shown in [8], [10], [11], and is based on the principle of domain decomposition, where each device is assigned either a fraction of the total amount of particles in the simulation, or a section of the computational domain. Compared to single-GPU programming, using multiple GPUs introduces a layer of complexity due to the distributed memory (each device only has access to a fraction of the entire particle system) and the need to exchange data between different devices.

In this paper we present an enhanced version of GPUSPH, that introduces an additional layer of parallelism by allowing the distribution of the computation across multiple GPUs connected to multiple host machines. To achieve this, the multi-GPU version of GPUSPH [8], [11] has been restructured, removing restrictions in the domain decomposition, refactoring data transfers to allow transparent network usage, and including a number of enhancements such as homogeneous precision (see Hérault *et al.* elsewhere in these proceedings) which are essential to avoid numerical issues in the large-scale high-resolution simulations which are made possible by the new opportunity to distribute computations across hundreds of devices.

We will first present an overview of the general and technical features of GPUSPH for single devices (section II), which will provide a basis to introduce the changes necessary to support multiple devices on one or multiple host machines, discussed in section III with the implementation details and technical challenges. Results are presented and discussed in section IV-B, leading to the conclusions in secion V.

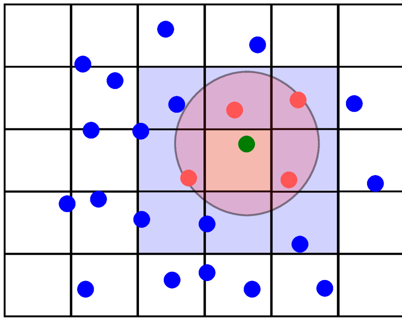


Fig. 1. If cells have side equal to the influence radius, neighbors of the green particle must reside inside in the immediate neighbor cells (light blue).

II. Single-GPU SPH

We recall here only the fundamental aspects of the structure of GPUSPH for single devices, as necessary to understand the challenges faced to introduce the improvements presented further on. For additional details, the reader is referred to [4] and [11].

A. Computational kernels

The structure of the general SPH simulation translates directly into the computational phases of the implementation. On GPU, these are implemented through a number of computational kernels, each dedicated to a separate task:

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 smallest Δt among the maxima computed by `Forces`;
4) **`Euler`** — For each particle, integrate the particle motion and properties over the selected Δt.

Additional kernels are also present to (optionally) compute corrections such as the Shepard or MLS density smoothing, or as auxiliary methods before forces computation when specific simulation parameters require it. For example, additional computational steps are required for γ renormalization coefficient when using semi-analytical boundaries [9], [12] and for the sub-particle scale (SPS) viscosity model [2], and an additional parallel reduction is executed on object particles to obtain the total force and torque acting on floating objects [14].

GPUSPH uses an explicit predictor/corrector integration scheme, so the `Forces`, `MinimumScan` and `Euler` kernels are executed twice for each timestep.

B. Cell-based domain subdivision

In SPH each particle needs to access the data of the neighboring particles multiple times for each time step. It is thus faster to prepare a neighbor list once and iterate over it rather than searching for neighbors every time they are needed. The naive $O(N^2)$ approach for neighbor search can be significantly sped up by partitioning the domain in virtual cells with side no less than 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* [3]. Figure 1 is a schematic representation of the virtual cells in 2D.

In the version of GPUSPH we present here, as an advancement over the implementations discussed in [4], [8], [11], the cells used for the fast neighbor search are also used to improve the numerical precision of the code by writing the position of each particle in terms of (integer) cell index i and (fractional) position within the cell f. Throughout the simulation, the coordinates (f, i) are used in place of the global position of the particles in the absolute reference system, ensuring homogeneous precision in the computation of the distances between the particles, for arbitrary domain sizes and origin location: this improvement is particularly of interest for the multi-device version of GPUSPH, since it allows the simulation of domains of arbitrary dimensions without loss of precision. Further details of the benefits of this approach are discussed in Hérault *et al.* [13].

Even with the cell-based fast neighbor search, building the neighbor list for each particle still requires about 50% the total simulation time. As a further optimization we only rebuild the neighbor list every k-th iteration, with k a user-controlled parameter. Due to the small timestep imposed by the explicit integration scheme, using $k = 10$ causes a negligible loss of accuracy, while decreasing the time required for the neighbor list construction from 50% to about 10% of the total simulation time.

C. Reduced memory consumption

The use of the cell-based coordinates for particles introduced with the homogeneous precision is also a basis for an additional optimization, in terms of reduced memory consumption for the neighbor list. Compared to the previous versions of GPUSPH [4], [8], [11] which used to store the actual index of each neighbor, we now store the relative offset of neighbor index with respect to the index of the first particle in the cell, which is itself stored in a separate `cellStart` array. A special flag is used to separate neighbors belonging to different cells. Since the number of particles in a cell is less than 2048, each neighbor can be encoded in a `short` occupying half the memory used in previous implementations. As a result, GPUSPH is now able to hold about 2.5 million particles per GB of RAM with default settings.

D. Performance

As discussed in [8], [11], we measure throughput in Iterations times Particles Per Second (IPPS), as a resolution-independent performance measurement. On the current generation of GPUs, GPUSPH achieves performances in the order of 10 MIPPS (millions of IPPS).

III. Multi-GPU, multi-node GPUSPH

A multi-GPU version of GPUSPH has already been presented in [8], [11]. The mentioned multi-GPU version of

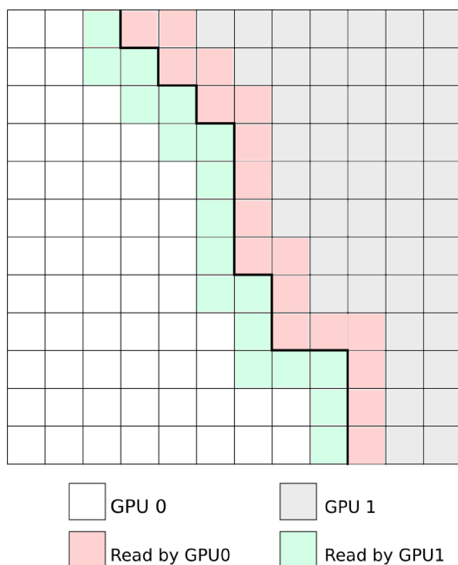


Fig. 2. Two-dimensional example of domain subdivision highlighting internal, external and edge cells.

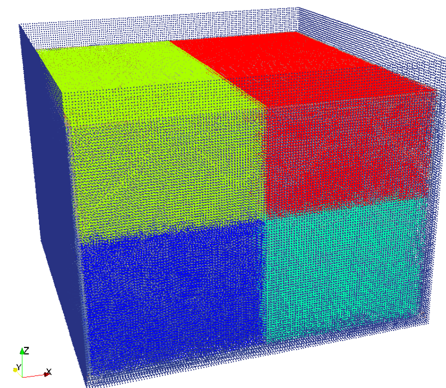


Fig. 3. Still-water problem with 2M particles split across 4 devices by two orthogonal planes, achieving a throughput of 39 MIPPS. Particles are colored by device.

GPUSPH allowed distribution of the simulation across multiple GPUs connected to the same host machine, and was based on the concept of domain decomposition: the computational domain was split in sections assigned to each devices, with the restriction that the split between sections had to be a plane orthogonal to the main axis used in the linearization of the indices of the auxiliary cells. In [11], the split was dynamically adjusted at runtime to achieve automatic load balancing.

Motivations for the restrictions in the domain splitting were simplicity of implementation, efficiency in data transfer, and the effective limit of 8 GPUs imposed by the requirement that all devices be connected to the same host (8 devices being the maximum number of devices connected to a PCI bus).

We extend the work presented in [8], [11] by allowing the distribution of the computation across devices connected to one *or more* host machines, allowing a networked cluster of GPU-enabled machines to be exploited concurrently for a single simulation. The multi-node version of GPUSPH we present here shared much of the general structure with the multi-GPU, single-node version previously presented, but lifts the more significant restrictions imposed in the older version.

A. Splitting the problem

As discussed in [8], spatial domain decomposition is the most efficient method to distribute the computational burden of an SPH simulation across multiple devices. Each GPU is therefore assigned a specific partition of the whole domain, but in order to compute the forces acting on the particles which are found in its domain, each GPU must also hold a copy of the data of the neighboring particles, which may be physically located in the adjacent domain partition assigned to a different GPU. This introduces the concept of *internal* particles (those in the domain partition assigned to the specific GPU), *external* particles (neighboring particles in domain partitions assigned to other GPUs), and *edge* particles (internal particles which are external for the GPUs managing the adjacent domain partitions), as illustrated in Fig. 2.

Management of data transfers across devices and accounting of internal, external and edge particles is greatly simplified by relying on the grid structure described in section II-B, introduced for fast neighbor search and now also used for particle positions to achieve homogeneous precision.

In [8], [11], the domain split was further restricted to the granularity of *slices* of cells, orthogonal to the main direction in the cell index linearization, in order to ensure that edge cells were stored consecutively and could therefore be transferred more efficiently. For multi-node simulations, which can be potentially distributed across tens or even hundred of devices, such a restriction would severely limit the applicability and efficiency of GPUSPH. In the new version we present here, the restriction is lifted, and domain decomposition can be done at the granularity of individual cells. The algorithm used for the initial domain split can be user-defined, and some common algorithms based on parallel and orthogonal splits are also offered as pre-defined options (see Fig. 3 and 4).

Arbitrary splitting comes at the cost of some additional memory consumption, since the particle system is now augmented by a `DeviceMap`, which encodes the index of the host node and GPU to which each cell is assigned. Data transfer is efficiently managed in bursts, as described in section III-C.

B. Kernels

The compute kernels have not undergone significant changes from [8], [11], since the only difference visible at the GPU level in the transition to multi-node is the cell granularity in the decomposition. Since this is handled by encoding the zone of the particle in the hash, the particle sorting automatically takes care of grouping the particles in the most appropriate memory locations.

As a recap, we recall that with our multi-device approach most kernels (the `Forces` kernel, as well as the kernels for

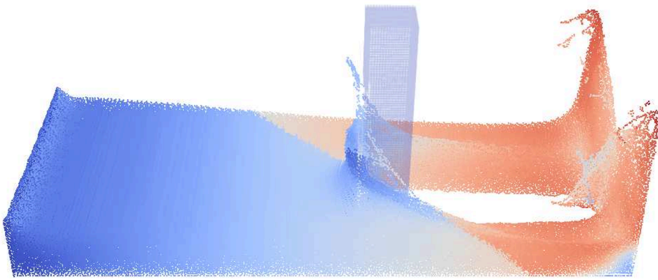


Fig. 4. Dam-break problem distributed across two devices by a plane which is not parallel to any of the axis. Particles are colored by device and shaded by velocity.

Shepard, MLS, γ or SPS computation) are always executed on internal particles only.

We also recall that `Euler` kernel barely manages to saturate single GPUs and it is therefore not suitable to hide the transfers times of positions and velocities of the edge particles. Exchanging the forces acting on the particles, instead of their positions and velocities, decreases the amount of data that needs to be exchanged and allows transfers to be started concurrently with a part of the `Forces` kernel. The computation of the forces is therefore split into two chunks, the first acting on a fraction of the domain that includes edge particles, and the second consisting of internal, non-edge particles only. The latter chunk is executed concurrently to the exchange of data for the edge particles.

This scheme implies that each GPU must integrate both the internal and external particles. Due to the very small runtimes of the `Euler` kernel, this has no perceivable impact on the overall simulation performance.

C. Bursts

The mentioned split constraint in the previous versions of multi-GPU GPUSPH automatically caused the edge cells to be sorted at the end of the particle array. This does not hold anymore with the possibility of arbitrary domain splits: edge particles might end up in a sparse distribution if sorted according solely to the cell index, which would make them particularly inefficient to read/write during transfers between devices.

To maintain an efficient separation between internal and edge cells, the two highest bits of the particle hash (used for sorting) encode information about the internal, edge and external status of each particle, ensuring that particles in the same spatial condition are kept adjacent in memory: particles in internal non-edge cells are stored first, followed by particles in edge cells, followed by external particles.

Although this makes the split policy independent from the cell linearization, it is still not guaranteed that the internal cells that are going to be read by a neighbor device are always consecutive in the particle list. In fact, cells can now have multiple neighbor devices, and the series of edge cells that two or more devices need to read might overlap and/or interlace; on the other hand, making one transfer per cell would bring excessive overhead.

To address this problem, an optimal list of *bursts of cells* is computed according to the current *DeviceMap*. If two cells are adjacent in memory and they have the same recipient device, they also appear in the same burst. Each burst is characterized by a sending or recipient device id, transfer direction, transfer *scope* and cached particle indices, which are updated after each sort. The transfer scope is either *intra-node* or *inter-node*: the former is enacted directly by a single peer device copy, while the latter requires explicit *send* and *receive* operations with the underlying network communication library.

This solution allows to perform only the minimal set of transfers across neighboring devices. With simple split policies, the optimal set consists of one single transfer. The bursts sizes and the particle indices are cached and are not exchanged until their value might have changed.

The same transfer hiding technique used in [11] is still implemented. Since the second chunk of the `Forces` kernel is issued asynchronously with the host, it covers both intra-node and inter-node transfers scopes. We recall that the technique of partitioning the workload of a kernel, with the aim of transferring part of the output data as soon as it is ready, is often referred to as *striping*.

D. Network communication

Inter-node communication relies on the widely used MPI standard. Since GPU-based clusters recently grew in popularity, some MPI implementations started to offer GPU-specific features. One important feature for applications requiring frequent data transfers across devices of the same network, as GPUSPH does, is RDMA (Remote Direct Memory Access) capability, which allows for *zero-copy networking*.

Low-level communication libraries need to stage the data in temporary buffers before sending them to the network adapter. These data might be in turn need to be staged in an application buffer as an output from the GPU device, resulting in two host copies before the actual network communication. RDMA allows the MPI library to fetch the data directly from the device memory, reducing the host overhead by about one half [6]. GPUDirect is the commercial name of one implementation of this mechanism between NVIDIA GPU devices and Mellanox network adapters [7]. MPI libraries supporting this technology are sometimes referred to as *CUDA-aware*, and allow GPU memory pointers to be passed directly to the MPI function calls.

GPUSPH can optionally exploit the GPUDirect technology if the underlying MPI library supports it. Our tests relied on the MVAPICH2 library [1], which was the first CUDA-aware MPI implementation. However, MVAPICH2 does not currently support multiple devices per process whe using GPUDirect. We overcame this limitation by running on each node multiple single-GPU processes instead of one multi-GPU process. This comes at the price of doubling the memory usage on host side; luckily this is not a problem as the major memory limitations

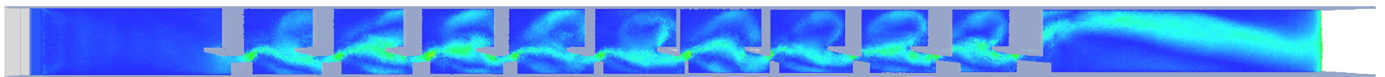


Fig. 5. Overview of the full fish-pass geometry.

GPUs	MIPPS ideal	MIPPS w/o striping	MIPPS w/ striping	Speedup w/ striping
1	12	12	12	1.00×
2	24	23	23	1.92×
3	36	33	34	2.83×
4	48	43	44	3.67×
5	60	53	54	4.50×
6	72	59	61	5.08×

TABLE I
PERFORMANCE AND EFFECTIVE SPEED-UP FOR THE DAMBREAK3D PROBLEM, WITH AND WITHOUT STRIPING.

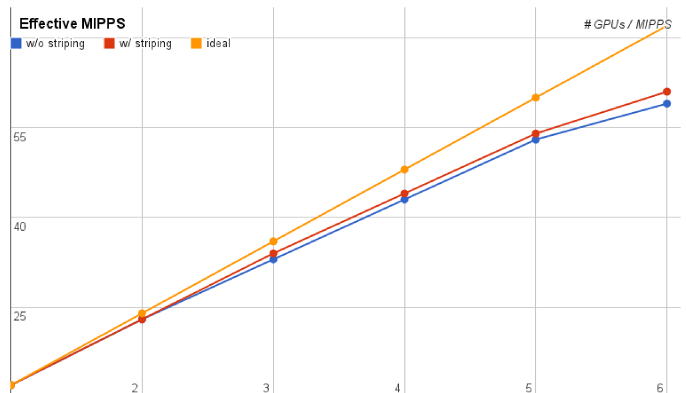


Fig. 6. Throughput (in MIPPS) for a multi-GPU simulation across 6 GPUs.

is always the device side, since the GPU holds double buffers and device-specific arrays (such as the neighbor list) which are not needed on the host side.

IV. EFFICIENCY EVALUATION

A. Testbeds

We performed our tests on two different testsbeds: a TYAN-FT72 rack mounting a 16-cores Intel Dual Xeon processor and 6× NVIDIA GTX480 cards (1.5GB of RAM per GPU) and a cluster of 8× Bullx 515 blades, each carrying 2× 8-cores Intel Xeon, 2× Mellanox ConnectX-3 IB adapters and 2× NVIDIA Tesla K20m (4GB of RAM per GPU). CUDA runtime 5.5 is installed in both setups.

In GPUSPH terminology, a *problem* is the definition of a physical domain, fluid volumes and geometrical shapes (a scene) to simulate. The reference scenario for performance evaluations was a box with 0.43m^3 of water (called DamBreak3D). Other problems we are currently developing include a fish way (FishPass), which has been used in the multi-node cluster to evaluate the performance of the simulator with a real application including non-trivial features such as the creation and destruction of particles (inlets and outlets).

B. Results

1) Performance scaling: As mentioned in section II-D, we measure throughput in MIPPS. In the case of multi-device simulations, the MIPPS value is accounted for the whole simulations, multiplying the total number of particles of the system by the total number of iterations, then divided by the total runtime.

We can evaluate both strong and weak scaling of our implementation by running different sets of tests.

For strong scaling, we fix the number of particles and evaluate how the performance scales across an increasing number of GPUs. These results are summarized in Table I and plotted in Fig. 6 for a DamBreak3D problem on the 6-GPU machine, both with and without the use of striping to hide cross-device transfers. Results are compared against the ideal performance (obtained multiplying the number of GPUs by the performance of a single device), and the effective speed-up is computed. We can observe that the effect of striping becomes more evident as the number of used devices increases, although relatively small (about 3.3%). The performance loss with respect to the ideal scaling also increases with the number of GPUs, although in the worst case (6 devices) we still achieve a speed-up which is over 80% of the ideal one.

For weak scaling, we fix the number of particles per device, and show how using multiple devices allows us to simulate larger systems. In the DamBreak3D case, we work with 1.6M particles per device. To increase the total number of particles in the system as the number of GPU increases, we reduce the inter-particle spacing (i.e. increase the resolution), keeping the number of particles per GPU constant. Results are summarized in Table II. To assess the weak scaling of our implementation, we compare the effective *MIPPS per GPU* by dividing the MIPPS of each test by the number of GPUs: the weak scaling is then defined as the ratio of *MIPPS per GPU* of the case with n GPUs against the case with 1 GPU. Again, the worst-case scenario (6 GPUs) shows an efficiency of 82%.

2) Full-scale multi-node simulations: A major benefit of exploiting all the devices of a network is the possibility to sum up the capacity of all the nodes for denser and/or bigger simulations. The main limitation of graphic devices is currently the amount of memory, even considering recent models in high-end market segments. This does not regard only the mere number of particles of a problem, but can even hinder its feasibility, especially when the ratio between the domain size and the smallest geometry detail is very high.

Consider for example a simulation of a full-length fish-pass

GPUs	n. of parts	Δp	seconds	iterations	MIPPS	Weak scaling of MIPPS/GPU
1	1.6M	0.004	3,300	23,445	12	1.00
2	3.2M	0.003225	4,100	29,485	23	0.96
3	4.8M	0.0028	5,100	34,160	33	0.92
4	6.4M	0.00256	5,700	37,564	43	0.90
5	8.0M	0.00238	6,500	40,691	50	0.83
6	9.6M	0.002234	7,100	43,483	59	0.82

TABLE II
ASSESSMENT OF WEAK SCALING IN THE DAMBREAK3D CASE, WITH STRIPING ENABLED.

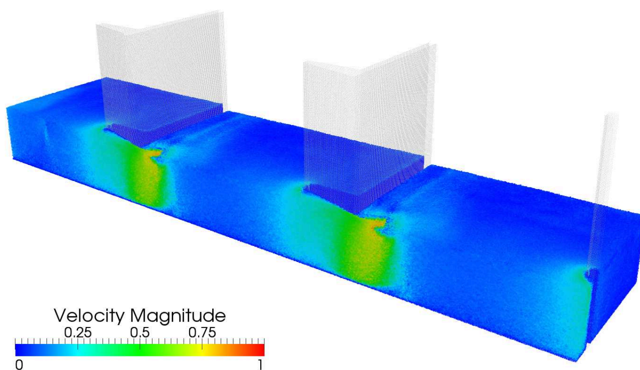


Fig. 7. Three-device section (10M particles) of a fish-pass simulation with 50M particles across 16 devices on 8 nodes. Particles are colored by velocity.

with 9 pools plus the inlet and outlet area (Fig. 5). The full channel is 16m long, while the smallest geometric detail (the wall thickness) is 0.04m wide. To obtain meaningful results, we must be able to resolve the smallest details with a sufficient number of particles (at least 10): we must therefore have $\Delta p \leq 0.004$m, resulting in no less than 50M particles, which is larger than the amount that can fit even on a high-end device.

The possibility to simulate the Fishpass problem in a multi-node environment was fundamental for the quality of the results, which we are still validating against laboratory measurements. We have run simulations with 50M particles distributed across 16 devices, with a split roughly correspondent to a pool for each device (Fig. 7). The performance of such simulations is $\simeq 145$ MIPPS with naive transfers and $\simeq 151$ MIPPS with all the optimizations enabled.

Since 12 MIPPS is the average performance of a single device of one node, the speedup is slightly higher than $12.6\times$, with $16\times$ being the theoretical maximum. In this case the scaling efficiency is $\simeq 79\%$, which is lower than the single-node multi-GPU efficiency, but still a very satisfactory initial result. The results suggest that further analysis of the MPI performance is needed.

We have also run a DamBreak3D simulation with 107 million particles across all the 16 devices of the network and we could potentially increase this number by tweaking specific parameters (e.g. tuning the size of the neighbor list to fit the needs of a specific problem).

V. CONCLUSIONS AND FUTURE WORK

We presented a scalable multi-node, multi-GPU implementation of the CUDA-based SPH fluid simulator GPUSPH. It is possible to partition the simulation with an arbitrary split function, with minimal overlap of neighboring cells.

A smart cell addressing technique ensures that neighboring cells are compacted in an optimal number of bursts for data exchanges; this allows for minimal data transfers over the network as well as across GPUs on the same machine.

Communication overheads are covered overlapping computations and data transfers. Performance scales almost linearly with the number of devices, with an efficiency of over 80% both in terms of strong and weak scaling in single-node simulations, and over 75% in multi-node simulations.

Multi-node and multi-GPU simulations support the same feature set of single-GPU ones, including floating objects, semi-analytical boundaries, SPS viscosity and homogeneous precision. The latter is particularly important for multi-device simulations, as it allows problems with almost arbitrarily large domains and high resolutions, without loss of accuracy.

Since the load-balancing in the previous multi-GPU, single-node version of GPUSPH is too simple to deal with the new partitioning opportunities introduced in the multi-node version of GPUSPH, we are currently working on an advanced load balancing engine with cell granularity which will further improve the simulator performance, particularly in the case of unbalanced topologies or hardware with heterogeneous computational powers.

REFERENCES

[1] Liu J., Wu J., Panda D.K. (2004) *High Performance RDMA-Based MPI Implementation over InfiniBand* International Journal of Parallel Programming **32**(3):167–198.
[2] Rogers B.D., Dalrymple R.A. (2005) *Three-Dimensional SPH-SPS Modeling of Wave Breaking*, Symposium on Ocean Wave Measurements and Analysis, ASCE, Madrid.
[3] Green S. (2010) *Particle simulation using CUDA*. [Online.] Available: http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/particles/doc/particles.pdf
[4] Hérault A., Bilotta G., Dalrymple R.A. (2010) *SPH on GPU with CUDA*, J. Hydr. Res. **48**:74–79.

[5] Crespo A.J.C., Dominguez J.M., Barreiro A., Gómez-Gesteira M., Rogers B.D. (2011) *GPUs, a new tool of acceleration in CFD: Efficiency and reliability on Smoothed Particle Hydrodynamics methods*, PLoS ONE, **6**(6):e20685. doi:10.1371/journal.pone.0020685.

[6] Wang H., Potluri S., Luo M., Singh A.K., Sur S., Panda D.K. (2011) *MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters* Computer Science — Research and Development **26**(3–4):257–266.

[7] Shainer G., Ayoub A., Lui P., Liu T., Kagan M., Trott C.R., Scantlen G., Crozier P.S. (2011) *The development of Mellanox/NVIDIA GPUDirect over InfiniBand—a new model for GPU to GPU communications* Computer Science — Research and Development **26**(3–4):267–273.

[8] Rustico E., Bilotta G., Hérault A., Del Negro C., Gallo G. (2012) *Smoothed Particle Hydrodynamics simulations on Multi-GPU Systems* in *20th nternational Euromicro Conference on Parallel, Distributed and Network-Based Processing* PDP, pp:384–391.

[9] Ferrand M., Laurence D., Rogers B.D., Violeau D., Kassiotis C. (2012) *Unified semi-analytical wall boundary conditions for inviscid, laminar or turbulent flows in the meshless SPH method*, Int. J. Num. Meth. Fluids **71**(4):446–472.

[10] Domínguez J.M., Crespo A.J.C., Valdez-Balderas D., Rogers B.D., Gómez-Gesteira M. (2013) *New multi-GPU implementation for Smoothed Particle Hydrodynamics on heterogeneous clusters.* Computer Physics Communications **184**:1848–1860. doi:10.1016/j.cpc.2013.03.008

[11] Rustico E., Bilotta G., Hérault A., Del Negro C., Gallo G. (2014) *Advances in Multi-GPU Smoothed Particle Hydrodynamics Simulations*, IEEE Transactions on Parallel and Distributed Systems **25**(1):43–52 doi:10.1109/TPDS.2012.340

[12] Mayrhofer A., Ferrand M., Kassiotis C., Violeau D., Morel F.-X. (2014) *Unified semi-analytical wall boundary conditions in SPH: analytical extension to 3-D*, Num. Alg. doi:10.1007/s11075-014-9835-y

[13] Hérault A., Bilotta G., Dalrymple R.A. (2014) *Achieving the best accuracy in an SPH implementation*, in *9th International SPHERIC Workshop*

[14] Bilotta G., Vorobyev A., Hérault A., Mayrhofer A., Violeau D., *Modeling real-life flows in hydraulic waterworks with GPUSPH*, in *9th International SPHERIC Workshop*