GPU-SPHEROS: A GPU-Accelerated Versatile Solver Based on the Finite Volume Particle Method

Siamak Alimirzazadeh[1*], Ebrahim Jahanbakhsh[1,2], Audrey Maertens[1], Sebastian Leguizamon[1], François Avellan[1]

[1]Laboratory for Hydraulic Machines, École polytechnique fédérale de Lausanne, Lausanne, Switzerland

[2]Institute of Computational Science (ICS), Università della Svizzera italiana, Lugano, Switzerland

[*]siamak.alimirzazadeh@epfl.ch

***Abstract*– The Finite Volume Particle Method (FVPM) is a mesh-free Arbitrary Lagrangian-Eulerian (ALE) method for fluid flow simulations which includes many of the desirable features of mesh-based Finite Volume Method (FVM) and SPH. In this paper, we introduce a GPU-accelerated 3-D FVPM in-house solver, called GPU-SPHEROS. The solver features spherical particles support and, has been developed in CUDA featuring the Thrust library, and optimized CUDA kernels for both compute-bound and memory-bound algorithms. We achieved a substantial speedup by a factor of more than 9x on NVLink-based Tesla P100 Pascal GPU compared to a CPU node equipped with two Intel® Xeon® E5-2660 v2 CPUs. In the present paper, the deviation of a circular water jet impinging a flat plate has been also simulated entirely on GPU, as a case study.**

I. INTRODUCTION

FVPM is a mesh-free particle-based method, which is both locally conservative and consistent. FVPM includes many of the attractive features of both particle methods such as SPH [1] and conventional mesh-based FVM [2]. This makes the aforementioned method more robust for free-surface modelling and solid deformations. Hietel et al. [3] introduced FVPM in 2000 applied for compressible flow computations. In 2009, Nestor et al. [4] extended the method to incompressible flows. In 2014. Quinlan et al. [5] presented an exact method for computing FVPM interaction vectors in 2-D. This method then extended by Jahanbakhsh et al. [6] for 3-D cuboidal particles. While cuboidal particle supports have the advantage of very simple geometric computations, they have the disadvantages of directionality and discontinuous interactions. These drawbacks motivated Jahanbakhsh et al. [7] to use spherical particles support with top-hat kernels.

GPU-SHEROS is a GPU-accelerated in-house solver based on FVPM, which is used for numerical simulation of fluid flow. The capability of Graphic Processing Units (GPUs) to handle particle-based methods has been demonstrated by several studies such as [8], [9], [10]. Herault et al. [8] used GPU-SPHysics to simulate the water waves surf zone based on SPH method. They proposed various techniques to get a higher performance on GPU, e.g. efficient memory access and restriction of absolute minimum use of conditional instructions. Nocentino and Rhodes [9] described the optimization of memory access methods on GPU using Morton order indexing. They showed that Morton ordering reduces the number of memory transactions and provides more efficient memory access. Bédorf et al. [11] simulated the N-body problem entirely on GPUs. They used an octree algorithm based on space filling curves to perform particles' nearest neighbor search. They reported the speedup for each part. For example for sorting the data they achieved speedup of higher than 11x on GTX480.

II. NUMERICAL METHOD

A. Governing equations

The equations of motion for isothermal and weakly-compressible flows are derived from the mass and momentum conservation law [6].

$$\frac{d\rho}{dt} = -\rho\nabla.\boldsymbol{C} \qquad (1)$$

and

$$\rho\frac{d\boldsymbol{C}}{dt} = \nabla.(\boldsymbol{s} - p\boldsymbol{I}) + \rho\boldsymbol{g} \qquad (2)$$

where ρ is the density, $\boldsymbol{C}$ is the fluid velocity vector, p is the static pressure, $\boldsymbol{s}$ is the deviatoric stress tensor and $\boldsymbol{g}$ is the gravitational acceleration. *d/dt* denotes the substantial time derivative. For Newtonian fluids, the deviatoric stress $\boldsymbol{s}$ can be written as:

ALGORITHM I. GPU-SPHEROS OVERALL ALGORITHM

```
for each time step t
   for each particle i
      find the neighbor particles j
   end for
   for each particle i
      for each neighbor j
         Compute interaction vectors based on spherical-support
      end for
   end for
   for each particle i
      for each neighbor j
         Compute momentum flux from pressure and deviatoric stress
            $f_i = \sum_i ((\rho \boldsymbol{C}\dot{\boldsymbol{x}} - \rho \boldsymbol{C}\boldsymbol{C})_{ij} - \boldsymbol{P}_{ij} + \boldsymbol{G}_{ij}) . \Delta_{ij} - p_b \boldsymbol{B}_i$
         Compute mass flux including the smoothing mass term:
            $m_i = \sum_i ((\rho \dot{\boldsymbol{x}} - \rho \boldsymbol{C})_{ij} + \boldsymbol{G}_{ij}) . \Delta_{ij}$
         Compute volume flux:
            $\dot{V}_i = \sum_i \dot{\boldsymbol{x}}_{ij} . \Delta_{ij} + \dot{\boldsymbol{x}}_i . \boldsymbol{B}_i$
      end for
   end for
   for each particle i (using 2nd order Runge-Kutta)
      Update volume, mass and momentum
      Update density and compute pressure from eq. of state
      Compute velocity correction and update particle velocity
      Update particle position
   end for
   $t \leftarrow t + \Delta t$

end for
```

$$\boldsymbol{s} = 2\mu(\dot{\boldsymbol{\varepsilon}} - \frac{1}{3}\mathrm{tr}(\dot{\boldsymbol{\varepsilon}})\boldsymbol{I}) \tag{3}$$

where μ is the dynamic viscosity and $\dot{\boldsymbol{\varepsilon}}$ is the deformation rate tensor given by:

$$\dot{\boldsymbol{\varepsilon}} = \frac{\nabla \boldsymbol{C} + (\nabla \boldsymbol{C})^T}{2} \tag{4}$$

The pressure is calculated based on the following equation of state:

$$p = \frac{\rho_0 a^2}{\gamma}\left(\left(\frac{\rho}{\rho_0}\right)^\gamma - 1\right) \tag{5}$$

where a is speed of sound, ρ_0 is the reference density and γ is the constant coefficient which is taken equal to 7. In weakly compressible flow simulations, the speed of sound a is considered at least 10 times greater than the maximum fluid velocity to reduce the computational cost [6]. The mass and momentum conservation equations can be written in the following PDE form arising from the conservation law.

$$\frac{d\boldsymbol{U}}{dt} + \nabla . \boldsymbol{F}(\boldsymbol{U}) = 0 \tag{6}$$

where $\boldsymbol{U}$ and $\boldsymbol{F}$ represents the conserved variable the flux function, respectively which for fluid flow equations read:

$$\boldsymbol{U} = \begin{pmatrix} \rho \\ \rho \boldsymbol{C} \end{pmatrix} \tag{7}$$

and,

$$\boldsymbol{F} = \begin{pmatrix} \rho \boldsymbol{C} \\ \rho \boldsymbol{C} \otimes \boldsymbol{C} - \boldsymbol{s} + p\boldsymbol{I} \end{pmatrix} \tag{8}$$

Being a meshless method, FVPM is intended for problems where mesh-based methods may fail or have difficulties, such as moving or free boundaries or fluid-structure interaction problems.

B. Finite Volume Particle Method

The FVPM formulation for conservation laws, (9) reads:

$$\frac{d}{dt}(\boldsymbol{U}_i V_i) = \sum_j (\boldsymbol{U}_{ij} \otimes \dot{\boldsymbol{x}}_{ij} - \boldsymbol{F}_{ij}) \cdot \Delta_{ij} + (\boldsymbol{U}_b \otimes \dot{\boldsymbol{x}}_b - \boldsymbol{F}_b) \cdot \boldsymbol{B}_i \tag{9}$$

and

$$\frac{dV_i}{dt} = \sum_j \dot{\boldsymbol{x}}_{ij} \cdot \Delta_{ij} + \dot{\boldsymbol{x}}_b \cdot \boldsymbol{B}_i \tag{10}$$

with

$$\Delta_{ij} = \boldsymbol{\Gamma}_{ij} - \boldsymbol{\Gamma}_{ji} \tag{11}$$

$$\dot{\boldsymbol{x}}_{ij} = (\dot{\boldsymbol{x}}_j \boldsymbol{\Gamma}_{ij} - \dot{\boldsymbol{x}}_i \boldsymbol{\Gamma}_{ji}) \frac{\Delta_{ij}}{\Delta_{ij} \cdot \Delta_{ij}} \tag{12}$$

and,

$$\boldsymbol{B}_i = -\sum_j \Delta_{ij} \tag{13}$$

where $\boldsymbol{U}_i$ is the conserved variable of the i^{th} particle, V_i is its volume, $\boldsymbol{U}_{ij}$ and $\boldsymbol{F}_{ij}$ are the conserved variable and the flux function at the interface of particles i and j, respectively, whereas $\dot{\boldsymbol{x}}_{ij}$ is the velocity at which the interface moves. Similarly, $\boldsymbol{U}_b$, $\boldsymbol{F}_b$ and $\dot{\boldsymbol{x}}_b$ are the conserved variable, flux function and particle velocity at the boundary. $\boldsymbol{B}_i$ and $\boldsymbol{\Delta}_{ij}$ are the vectors which weight exchanged fluxes with the boundary and between particles, respectively. These vectors are computed from interaction vector $\boldsymbol{\Gamma}_{ij}$, which reads,

$$\boldsymbol{\Gamma}_{ij} = \int_\Omega \frac{\psi_i \nabla W_j}{\sigma(\boldsymbol{x})} \tag{14}$$

In (13), ψ_i denotes the Shepard shape function for the i^{th} particle defined as:

$$\psi_i = \frac{W_i(\boldsymbol{x})}{\sigma(\boldsymbol{x})} \tag{15}$$

where $W_i(x)$ is a kernel function,

$$W_i(\boldsymbol{x}) = W(\boldsymbol{x} - \boldsymbol{x}_i, h_i) \tag{16}$$

and σ is the kernel summation,

$$\sigma = \sum_j W_j(\boldsymbol{x}) \tag{17}$$

$W_i(x)$ is defined as zero outside Ω_i, the support of particle i.

$$W_i(\mathrm{x}) = \begin{cases} 1 & x \in \Omega_i \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

The particle volume V_i is defined as:

$$V_i = \int_{\Omega_i} \psi_i dV \tag{19}$$

In (16), h_i is known as the smoothing length of particle i, and defines the particle size and hence the spatial resolution of the scheme.

III. SPHEROS AND GPU-SPHEROS

SPHEROS is a FVPM versatile in-house solver, which inherits desirable features of both SPH and mesh-based FVM and is able to simulate the interaction between fluid, solid and silt. It has been already developed and validated for different CFD benchmarks [6]. GPU-SPHEROS is the GPU-accelerated version of the code and has been developed and implemented in CUDA in order to exploit the GPU many-core architecture. The overall algorithm of GPU-SPHEROS is shown in algorithm I, which includes three main parts:

a) Particles octree-based neighbor search.
b) Computing the particle interaction vectors based on a spherical-support.
c) Computing fluxes, forces, and 2nd-order Runge-Kutta time integration.

A. *Octree-based neighbor search*

In the present research, an octree algorithm which was introduced by Bédorf et al. [11], has been implemented for parallel particle neighbor search. Space filling curves (here, Morton-curve) have been used for tree construction, reordering particles data. In fact, the Morton keys of particles give a 1-D representation of the original n-D coordinate space and are computed using bitwise operations based on multi-level masking. After the Morton keys are generated, the particles ID are sorted based on their corresponding Morton keys to improve the memory access and cache-coherency during the next computations. A parallel radix sort algorithm provided by Thrust library has been used for sorting the data.

Once the particles ID are sorted based on their corresponding Morton keys, the particles have to be grouped into different tree cells. For this purpose, different levels of bitwise masking must be applied to particles Morton keys. The particles with identical masked Morton key are assigned to the same cell, which is also a branch of the tree. If the number of particles in one cell is less than the defined parameter N_{leaf}, then the next level of masking is not applied to the particles belonging to that cell which is then called leaf. The counting procedure of the particles is based on parallel stream compaction algorithms using the Thrust library. The masking and particles grouping process is repeated for every single level, sequentially, until all the particles are assigned to leaves or the maximal depth of the tree is reached, whichever occurs first. An example of a generated tree based on the Morton curve (or *z-curve*) method for N_{leaf} = 4 is shown in Figure 1.

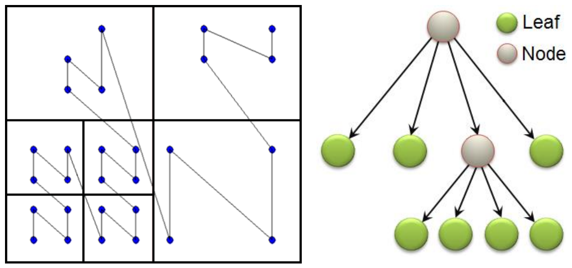

Figure 1. Schematic representation of a constructed quadtree (right) and example of paticle grouping based on space filling curve algorithm for 28 particles with N_{leaf} = 4 (left). For particle grouping into tree cells, we apply the binary level mask to particles Morton keys and group the particles with indentical key to the same cell. The cells with less than N_{leaf} particles are flagged as leaf (green cells) and are not masked further. The other cells are called nodes (which are colored in gray) and their particles are masked in the next level(s). We perform these flagging and checking process each level to construct the tree. The tree construction procedure will be stoped once all the particles are assigned to leaves, or when the maximal level is reached. In the present figure, a quadrtree with two level of masking has been represented schemtically as an example.

Once the tree is constructed, one can find the neighbor cells of each branch based on their Morton keys. In order to identify the neighbors of a particle, only the distance between particles belonging to the same leaf and its neighbor leaves needs to be checked, thus avoiding unnecessary computations. A highly-optimized kernel has been implemented for this task.

B. *Exact computation of interaction vectors*

Computing the interaction vectors requires that the supporting border of i^{th} particle $\partial\Omega_i$ be partitioned into sub-surfaces of constant σ values, which are termed elementary surfaces. Then the interaction vector $\boldsymbol{\Gamma}_{ij}$ is computed exactly as:

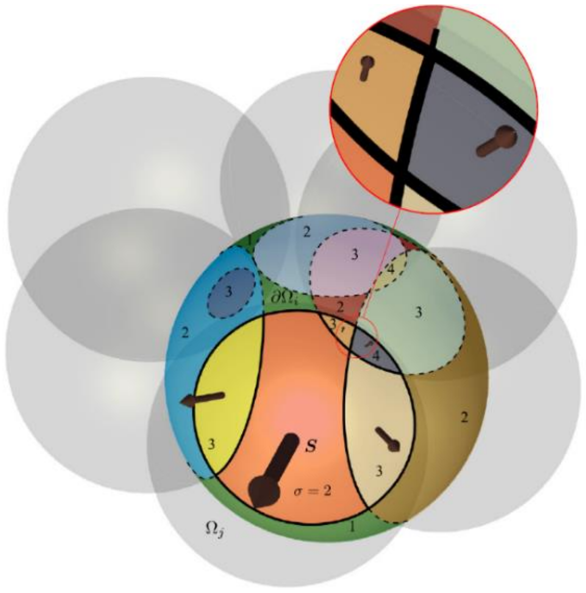

Figure 2. The intersection of $\partial\Omega_i$ spherical surface of i^{th} particle, with its neighboring particles. The spherical subsurfaces are shown in different colors with corresponding σ value [7].

$$\boldsymbol{\Gamma}_{ji} = \sum_{e \in (\Omega_j \cap \partial\Omega_i)} \left(\frac{\boldsymbol{S}_e}{\sigma_e^+ \sigma_e^-} \right) \tag{20}$$

where S_e denotes the area vector of the elementary surface e, Ω_j denotes the supporting volume of j^{th} particle, while σ_e^+ and σ_e^- denote the kernel summations outside and inside Ω_i, respectively.

Elementary surfaces can have complex or even disjointed shapes, making partitioning challenging. A schematic of partitioning of a spherical-support is shown Figure 2. The required computation of the area of elementary surfaces is also more difficult and computationally costly. The detailed algorithm has been explained in [7].

Computing the particles interaction vectors, $\boldsymbol{\Gamma}_{ij}$ and $\boldsymbol{\Gamma}_{ji}$ is the most costly part of GPU-SPHEROS. Once the particles neighbors identified, all the data are available to compute these vectors. Different non-concurrent CUDA kernels have been implemented for each part of the interaction vectors algorithm releasing one thread per particle to perform these computations. Since the size of some vectors have to be defined during the run-time, we pre-allocate a large memory size by estimating the upper-bound required memory. The kernels then perform the computations for a batch of particles in parallel, with subsequent batches released sequentially until all the interaction vectors have been computed. This way, we avoid dynamic memory operations (e.g. deallocation/reallocation and resizing) inside the kernels by providing sufficient pre-allocated memory for each batch. The batch size and the pre-allocated memory pool for each batch can affect both the amount of parallelization and the occupied memory size and there is a trade-off between size of batches and required memory. In fact, these values are set for each simulation in order to ensure a good performance and sufficient available memory for the simulation.

C. *Time integration and flux discretization*

Once the interaction vectors $\boldsymbol{\Gamma}_{ij}$ and $\boldsymbol{\Gamma}_{ji}$ are computed, the solver would be ready to compute forces and fluxes for all the particles. To compute these values, the discretized governing Eqs. (8) and (9) should be solved for each particle (see algorithm I). For time integration, we use the 2nd-order explicit Runge–Kutta predictor-corrector scheme and for numerical stability, the Courant–Friedrichs–Lewy (CFL) condition for 1st-order upwind discretization must be satisfied, i.e.

$$\Delta t \leq \frac{h_i}{|\boldsymbol{C}_i|} \tag{21}$$

Here, for a given CFL number, Δt is adapted for each time step as:

$$\Delta t = \text{CFL} \times \min\left(\frac{h_i}{|a_i + \boldsymbol{C}_i|} \right) \tag{22}$$

where a_i is the local sound speed.

Similar to interaction vectors, here we release one thread per particle to compute of fluxes and forces. For instance, the thread, which is released for the i^{th} particle, is responsible for all the mass and momentum exchanges between particle i and all its neighbors. Despite computing the interaction vectors, there is not any particle batching process for computing these values and we indeed can benefit from the maximum level of parallelization harnessing GPU many-core architecture without any sequential batch releasing procedure.

D. *Boundary conditions*

All the employed approaches to enforce boundary conditions have been presented in [6]. For no-slip wall boundary, we overlay a layer of boundary particles that their motion is fixed by the wall boundary dynamics. Their density is initially set to the fluid reference density.

For the inlet, the same approach is used to update the mass and the volume of the boundary particles. However, the inlet boundary particles move with the velocity known from the discharge rate and are fed to the system as soon as they cross the inlet border. In this case, new boundary particles replace the fed ones.

For the free-surface, no boundary particles are used but instead, a boundary flux is added to the mass, momentum and volume equations, as can be seen in Eqs. (9) and (10). More details can be found in [6] regarding the boundary conditions enforcement.

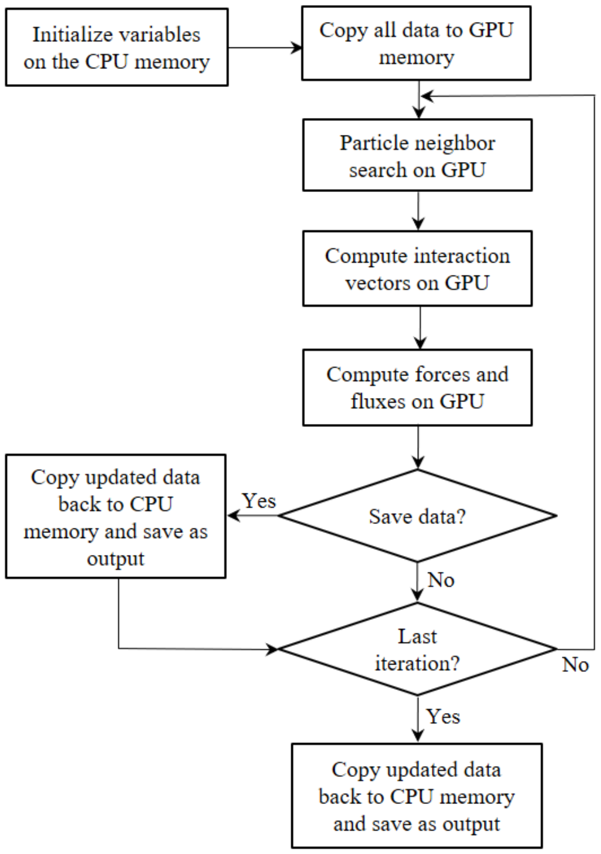
Initialize variables on the CPU memory
Copy all data to GPU memory
Particle neighbor search on GPU
Compute interaction vectors on GPU
Compute forces and fluxes on GPU
Save data?
Yes
Copy updated data back to CPU memory and save as output
No
Last iteration?
No
Yes
Copy updated data back to CPU memory and save as output

Figure 3. GPU-SPHEROS: flowchart of numerical simulation

E. *Data transfer*

Since PCIe or even NVLink are much slower than memory bandwidth of GPUs, we try to avoid any data transfer between CPU and GPU memory for a single GPU case except when we need to save data as output. All parts of the algorithm have been implemented on the GPU to be able to manage data exclusively on device memory, avoiding expensive host-device communication. The overall flowchart of an executing simulation is shown in Figure 3.

IV. CASE STUDY

The FVPM capability and accuracy has been already validated for different test cases such as viscous flow in a 2-D and 3-D lid-driven cavity, free-surface flow during impingement of a liquid jet on a flat plate and, moving boundary problems, addressing key aspects of the method [6,7].

However, for the present paper, deviation of a circular water jet impinged on a flat plate has been simulated using GPU-SPHEROS and the results have been compared to the experimental data measured by Kvicinsky et al. [12]. A schematic outline of this case study is represented in Figure 4.

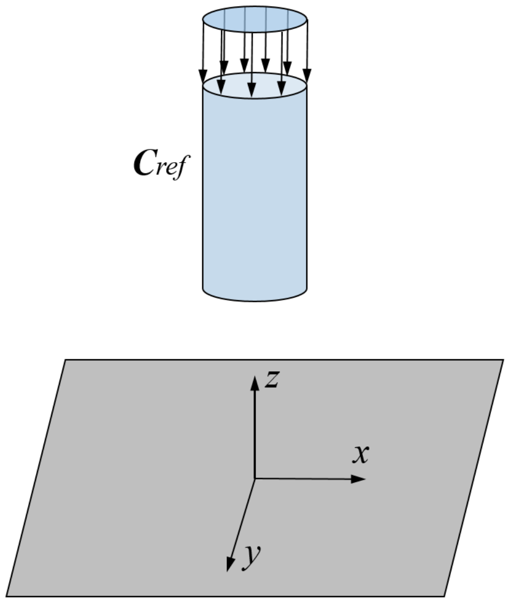

Figure 4. Schematic outline of the setup for impinging jet case study.

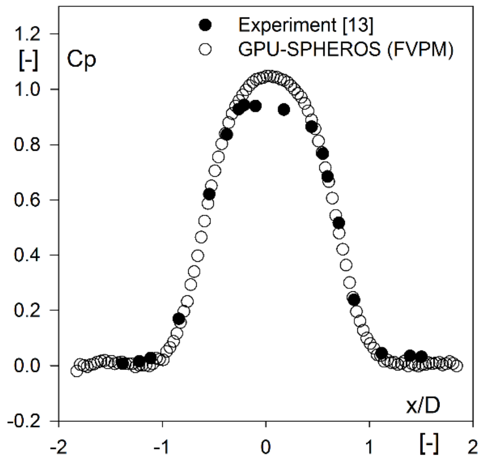

Figure 5. The pressure coefficient C_p plotted along x axis. FVPM solution is compared to experimental data [12]

The reference velocity at the inlet is $|C_{ref}| = 19.81$ m s^{-1} and the gravity acceleration $\boldsymbol{g}$ is 9.81 m s^{-2}. We inject jet particles as a circular inlet boundary and the plate is assumed as no-slip wall boundary. The center of the plate corresponds to the center of the Cartesian coordinate system and the plate is perpendicular to the jet. The inlet of the jet has the diameter of $D = 0.03$ m and is located at $Z = 2.5{\times}D$ above the flat plate. Figure 5 depicts the pressure coefficient C_p, along the x axis. C_p is averaged in the period ranging between t = 0.025 s and t = 0.05 s to filter out the pressure oscillations caused by compressibility. The particles impinging the flat plate is shown in Figure 6.

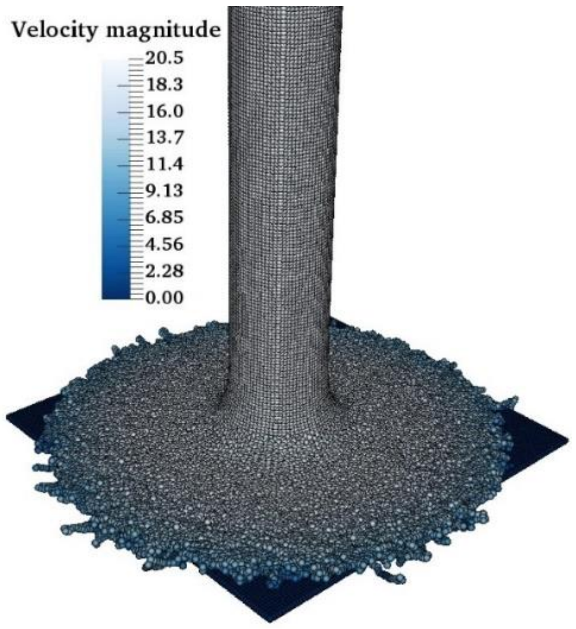

Figure 6. Water jet paticles impinging on a flat plat

TABLE I. OPTIMIZATION PROCEDURE OF VOLUME INTEGRAL GRADIENT KERNEL FOR 3.3×10^5 PARTICLES ON TESLA K40

level	*Technique/optimization*	*Time [ms]*
0	Thrust sequential reduction inside kernel	~100
1	for loop inside the kernel instead of thrust reduction inside kernel	23.69
2	Unrolling loops	5.27
3	Using vector type load/store	3.23
4	Pointer aliasing	2.89

V. OPTIMIZATION

The optimization procedure for GPU-SPHEROS is mainly focused on memory access efficiency, since it affects the performance of the GPU, considerably. Here we provide an optimization example for a kernel, which computes the pressure volume integral gradients. The volume integral gradient is computed based on (22).

$$\nabla p_i = \frac{1}{V_i}\sum_j \frac{p_i + p_j}{2}\Delta_{ij} \qquad (22)$$

The optimization process for this kernel is summarized in TABLE I. It is important to note that before optimization, the data have been sorted based on the particles Morton code to improve the memory particles Morton code to improve the memory access efficiency.

All the optimizations have been applied and tested on a Tesla K40 GPU. The applied techniques are explained below:

- Using “for loop” inside the kernel was more efficient than the reduction using the Thrust library inside kernel.
- Unrolling the loop inside the kernel reduces dynamic instruction count, due to fewer compare and branch operations. The compiler can also improve the Instruction Level Parallelism (ILP) due to availability of independent instruction.
- Using vector type arrays such as “double4” instead of 64-bit “double” non-vectors can improve the memory access efficiency by grouping same datatypes together. At the instruction level, a multi-word vector load or store only requires a single instruction to be issued, and total instruction latency for particular memory transaction will be decreased.
- By restricting pointers, we promise the compiler that two or more pointers will never overlap the memory. This helps the compiler to apply further optimizations.

The number of threads per block has been also optimized to improve the kernel performance even if the occupancy is degraded. In fact, a higher performance is not always achieved with higher occupancy. Readers are referred to [13] for more details on the relation between occupancy and performance.

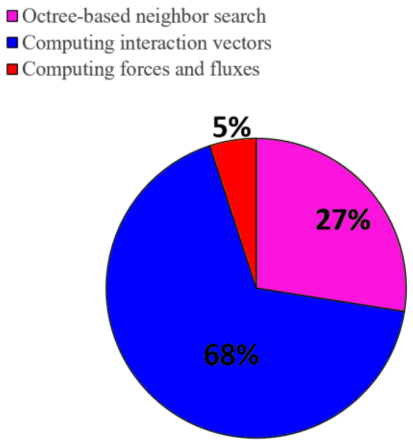

Figure 7. Time percentage of three different parts of the overall algorithm of GPU-SPHEROS

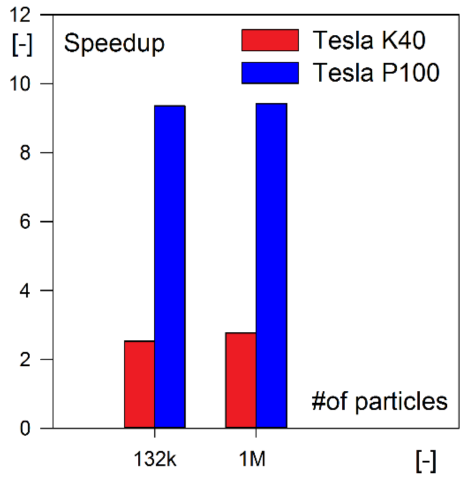

Figure 8. Achieved speedup: PCIe-based Tesla K40 and NVLink-enabled Tesla P100 vs Intel® Xeon® E5-2660 v2

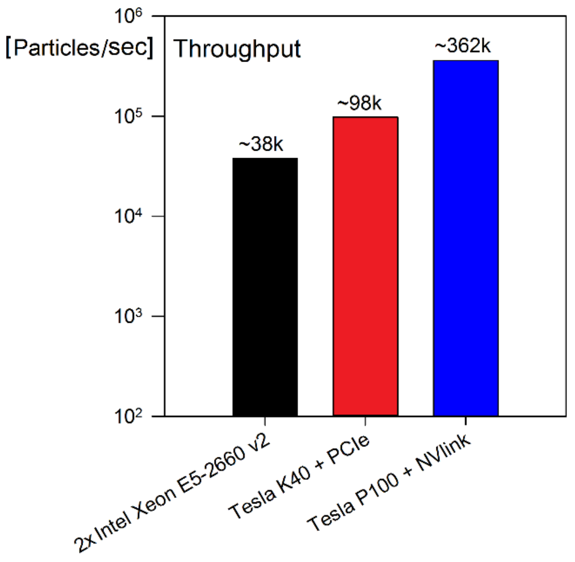

Figure 9. Solver throughput for Intel® Xeon® E5-2660 v2, PCIe-based Tesla K40 and NVLink-enabled Tesla P100

VI. SPEEDUP

After optimization, the performance of GPU-SPHEROS on both PCIe-based Tesla K40 and NVLink-based Tesla P100 has been compared to CPU version. The CPU version utilizes MPI for parallelization on multi-core CPU nodes. The weight of all three parts of the GPU-SPHEROS algorithm is shown in Figure 7. As one can see, computing the interaction vectors is around 68% of the total computations. Reasonably, this part is the priority for further optimizations.

However, we already achieved a substantial speedup by parallelizing the computations on GPU many-core architecture. For instance, on NVLink-based Tesla P100 with Pascal new architecture, the software is 9.5x faster compared to one CPU node equipped with two Intel® Xeon® E5-2660 v2 CPUs. Each CPU has 20 cores with activated hyper-threads. The speedup and software throughput are shown in Figures 8 and 9, respectively. We achieved almost same speedup for different problem sizes. To measure the speedup, we generated a uniform distribution of the particles in a cube and then applied a $\pm 0.1 h_i$ random disturbance to the particles' position. This way, the generated distribution is very similar to the realistic simulations, which we perform.

VII. CONCLUSION

In the present paper, General Purpose GPU (GPGPU) computing has been utilized to accelerate SPHEROS as a particle-based FVPM solver. The data used to compute interaction vectors and exchanged fluxes, have been already sorted during the neighbor search process using a radix sort parallel algorithm to avoid inefficient memory access. On NVLink-based Tesla P100, we could achieve almost 10x faster running speed compared to one CPU node equipped with two Intel® Xeon® E5-2660 v2 processors. We observed that the code is executed 3.8x faster on NVLink-based Tesla P100 compared to PCIe-based Tesla K40, since its theoretical bandwidth and double-precision peak performance are 2.5x and 3.7x higher, respectively. The code has been optimized for a Pascal-based architecture GPU and is actually expected to run on a Pascal-based multi-GPU cluster. The preferred candidate to further speed up the software is the interaction vectors computations, which takes around 70% of overall time. The next candidate can be the octree-based neighbor search, which constitutes around 27% of the simulation overall time.

ACKNOWLEDGMENT

This research was supported by the Swiss Commission for Technology and Innovation (CTI) with grant N° 17568.1. PFEN.IW.

The authors would also like to thank NVIDIA for the GPU grant program as well as their technical support.

REFERENCES

[1] R.A. Gingold, J.J. Monaghan, *Smoothed particle hydrodynamics-theory and application to non-spherical stars*, Mon. Not. R. Astron. Soc. 181 (1977) 375–389.

[2] R.J. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, Vol. 31, Cambridge university press, 2002

[3] D. Hietel, K. Steiner, J. Struckmeier, *A finite-volume particle method for compressible flows*, Math. Models Methods Appl. Sci. 10 (9) (2000) 1363–1382.

[4] R.M. Nestor, M. Basa, M. Lastiwka, N.J. Quinlan, *Extension of the finite volume particle method to viscous flow*, J. Comput. Phys. 228 (5) (2009) 1733–1749.

[5] Nathan J. Quinlan, Libor Lobovsky, Ruairi M. Nestor, *Development of the meshless finite volume particle method with exact and efficient calculation of interparticle area*,

Computer Physics Communications, Volume 185, Issue 6, June 2014, Pages 1554–1563

[6] E. Jahanbakhsh, C. Vessaz, A. Maertens and F. Avellan, *Development of a Finite Volume Particle Method for 3-D fluid flow*, Computer Methods in Applied Mechanics and Engineering, vol. 298, p. 80-107, 2016.

[7] E. Jahanbakhsh, A. Maertens, N.J. Quinlan, C. Vessaz, F. Avellan, *Exact finite volume particle method with spherical-support kernels*, Comput. Methods Appl. Mech. Engrg. 317 (2017) 102‒127.

[8] A. Hérault, A. Vicari, C. del Negro, and R.A. Dalrymple, *Modeling Water Waves in the Surf Zone with GPU-SPHysics*, in *Proceeding of the Fourth SPHERIC Workshop*, Nantes, 2009.

[9] A. E. Nocentino and P. J. Rhodes, *Optimizing memory access on GPUs using Morton order indexing*, in *Proceedings of the 48th Annual Southeast Regional Conference*, *ACM*, New York, USA, 2010.

[10] D. Valdez-Balderas, J. M. Domínguez and B. D. Rogers, *Towards accelerating* smoothed *particle hydrodynamics simulations for free-surface flows on multi-GPU clusters*, Journal of Parallel and Distributed Computing, vol. 73, no. 11, pp. 1483-1493, 2013.

[11] J. Bédorf, E. Gaburov and S. P. Zwart, *A sparse octree gravitational N-body code that runs entirely on the GPU processor*, Journal of Computational Physics, vol. 231, no. 7, pp. 2825-2839, 2012.

[12] S. Kvicinsky, F. Longatte, F. Avellan, J. Kueny, *Free surface flows: Experimental* validation *of the Volume of Fluid (VOF) method in the plane wall case*, in: Proceedings of 3rd ASME/JSME, San Francisco, ASME, New York, 1999, pp. 1‒8.

[13] http://www.nvidia.com/content/gtc-2010/pdfs/2238_gtc2010.pdf (last access on May 3rd, 2017)