We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



Chapter

Efficient Simulation of Fluids

Pierre Thuillier Le Gac, Emmanuelle Darles, Pierre-Yves Louis and Lilian Aveneau

Abstract

Fluid simulation is based on Navier-Stokes equations. Efficient simulation codes may rely on the smooth particle hydrodynamic toolbox (SPH), a method that uses kernel density estimation. Many variants of SPH have been proposed to optimize the simulation, like implicit incompressible SPH (IISPH) or predictive-corrective incompressible SPH (PC-ISPH). This chapter recalls the formulation of SPH and focuses on its effective parallel implementation using the Nvidia common unified device architecture (CUDA), while message passing interface (MPI) is another option. The key to effective implementation is a dedicated accelerating structure, and therefore some well-chosen parallel design patterns are detailed. Using a rough model of the ocean, this type of simulation can be used directly to simulate a tsunami resulting from an underwater earthquake.

Keywords: fluid simulation, SPH, CUDA, MPI, Navier-Stokes, tsunami

1. Introduction

Submarine earthquakes may generate tremendous disasters for human, like what occurred during the Tohoku earthquake in 2011. Even if their seismic waves may damage buildings and structures when they occur close to the coast, the tsunami they generally cause are a massive risk for humans. Indeed, the energy produced by a massive undersea quake is transmitted into the water at high speed and results in a high wave when it arrives on the coast.

To avoid human losses, tsunami's simulations can help to inform the governments and society about the risks before and after a submarine earthquake. This chapter presents solutions for implementing such simulations. The main objective is to be able to calculate the propagation of the tsunami wave into the ocean and then to simulate efficiently its effects when the wave reaches the coast. These kinds of simulation can be done in two dimensions considering only the profile of the coast or in three dimensions when all the topography is considered. In both cases, the simulation must handle how water is affected by the earthquake wave.

Viscosity is the measure depicting how a fluid resists deformations. Even water is considered having a non-nil viscosity: so, this parameter must be considered carefully for tsunami simulations. Water simulation relies on Navier-Stokes equations that describe the motion of a viscous fluid. Unfortunately, Navier-Stokes equations cannot be solved directly like it is the case for many differential equations. The only way to obtain a solution at a given time consists of approximating it through simulation. In practice, two family of methods may be used. The first one consists in discretizing the simulation space into small parts and to do the

simulation considering fixed cells in this discrete space (mesh approach). Well-known methods are the finite element, the finite difference, and the finite volume. A second alternative approach is the smoothed particle hydrodynamics (SPH), introduced in astrophysics in 1977 [1, 2], which is applied in computer graphics [3], oceanography, and many other fields. The latter is particularly interesting for tsunami simulation, since the most important part of the simulation is not in the ocean but rather on the ground. This implies that a part of the fluid will cover the coast. This heterogeneity makes the mesh-free SPH approach more adapted.

This chapter is organized as follows. Section 2 presents the basics of SPH, detailing the different involved mathematical expressions and steps and previous implementations proposed in the literature. Section 3 presents a parallel implementation of SPH: it recalls the main parallel patterns and how they are used to obtain a reliable and fast simulation. Before the conclusions, Section 4 presents some results for a simple case of tsunami.

2. SPH formulation

SPH is a Lagrangian approach, meaning that particles representing tiny parts of the fluids may move during the simulation. It is based on density estimation applied to moving particles, leading to an approximation of the Navier-Stokes equations. This section recalls these equations and presents the basics of SPH.

2.1 Navier-Stokes equations

Navier-Stokes equations model the dynamics of a fluid. They rely on the Newton second law, stating that the sum of the forces applied on a body equals the product of its mass by its acceleration ($\sum F = \mathrm{m} \cdot a$). In practice, it is a system of two equations: the mass continuity equation and the momentum equation. The first one is given by

$$\frac{\partial \rho}{\partial \mathrm{t}} + \nabla(\rho u) = 0 \tag{1}$$

where ρ designs the fluid density, ∇ is the gradient operator, and u is the flow velocity. The momentum equation is

$$\rho\left(\frac{\partial u}{\partial t} + \mathrm{u} \cdot \nabla u\right) = -\nabla \mathrm{p} + \mu \nabla^2 \mathrm{u} + \rho g \tag{2}$$

where $\cdot\nabla$ is the divergence operator, ∇^2 the Laplacian operator, p the pressure, μ the dynamic viscosity coefficient, and g the gravity term. The right part of the momentum equation represents the sum of the forces that the fluid undergoes, where $-\nabla \mathrm{p}$ is the pressure force, $\mu\nabla^2\mathrm{u}$ the viscosity force, and ρg the gravitational force. With the fluid velocity function being unknown, it is not possible to compute analytically its divergence. Then, the momentum equation is nonlinear.

Nevertheless, some methods allow to calculate an approximation of these two equations. Most of them regularly discretize the Euclidean space and calculate an approximation by using the finite difference theorem. The advection term (the left part of the momentum equation) is approximated placing particles into the grid and then computing their displacement. In other words, each grid cell contains a given amount of fluid, and the algorithm calculates the exchanges between adjacent cells.

Such a solution is quite difficult to use into environment where some large part (like ocean) and highly detailed parts must be considered together.

Another method to approximate the Navier-Stokes equations is SPH. The Euclidean space is no more discretized. Instead, it considers some moving particles representing the fluid and their interactions. Each particle comes with its specific velocity, pressure, density, and viscosity. Then, the total derivative allows to approach the advection term (those between parentheses on the left part of the momentum equation) by a single derivative term $\frac{du}{dt}$. For a given particle i, the momentum equation becomes

$$\rho_i\left(\frac{du_i}{dt}\right) = -\nabla p_i + \mu_i \nabla^2 u_i + \rho_i g \tag{3}$$

Therefore, the acceleration a_i of the particle i is given by

$$a_i = \frac{du_i}{dt} = -\frac{\nabla p_i}{\rho_i} + \frac{\mu_i \nabla^2 u_i}{\rho_i} + g \tag{4}$$

2.2 Introduction to SPH

SPH relies on the kernel density estimation [4]. When we only have some samples of a given function, we can estimate its value at a new location using a kernel function W and the following estimation:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} W\left(\left\|x - x_j\right\|\right) f(x_i) \tag{5}$$

The kernel function W must be a unit positive function. In other words, it must satisfy the two following properties:

$$\forall x \in \mathbb{R},\ W(x) \geq 0 \tag{6}$$

$$\int_{\mathbb{R}} W(x)dx = 1 \tag{7}$$

We denote W_h a kernel function with bounded support $[0, h]$. This simply means that $W_h(x) = 0$ for all $x < 0$ or $x \geq h$. This mathematical tool is used to approximate any scalar field A_i for any particle i:

$$A_i = \sum_{j=1}^{n} A_j \frac{m_j}{\rho_j} W_h\left(\left\|x - x_j\right\|\right) \tag{8}$$

where m_j and ρ_j are, respectively, the mass and the density of the jth particle. Useful kernels for liquids are given in [3]. From this simple expression, we can deduce the estimation of the gradient and the Laplacian of a scalar A_i:

$$\nabla A_i = \sum_{j=1}^{n} A_j \frac{m_j}{\rho_j} \nabla W_h\left(\left\|x - x_j\right\|\right) \tag{9}$$

$$\nabla^2 A_i = \sum_{j=1}^{n} A_j \frac{m_j}{\rho_j} \nabla^2 W_h\left(\left\|x - x_j\right\|\right) \tag{10}$$

These formulas allow to calculate the density of any particle, the gradient of the pressure, and the Laplacian of the velocity to approximate a solution of the Navier-Stokes equations. For each particle i, the SPH algorithm follows:

- Compute the density ρ_i:

$$\rho_i = \sum_{j=1}^{n} m_j W_h\left(\left\|x - x_j\right\|\right) \tag{11}$$

- Compute the pressure p_i:

$$p_i = k(\rho_i - \rho_0) \tag{12}$$

where k is the gas constant and ρ_0 is the rest density.

- Compute f_i, the sum of the forces at particle i of pressure, viscosity, surface tension, and gravity:

$$f_i^{pressure} = -\sum_{j=1}^{n} m_j \frac{p_i + p_j}{2\rho_j} \nabla W_h\left(\left\|x_i - x_j\right\|\right) \tag{13}$$

$$f_i^{viscosity} = \mu \sum_{j=1}^{n} m_j \frac{u_j - u_i}{\rho_j} \nabla^2 W_h\left(\left\|x - x_j\right\|\right) \tag{14}$$

$$f_i^{surface} = -\sigma \nabla^2 cs_i \frac{n_i}{|n_i|} \tag{15}$$

$$f_i^{gravity} = \rho_i g \tag{16}$$

where σ is the tension coefficient relating to the interface between the fluid and the exterior (the air), n_i is the normal vector to a particle i, and cs_i is the color field of the particle i.

- Compute the velocity u_i and the new particle position x_i using a small integration time step Δt:

$$u_i = u_i + \Delta t \frac{f_i}{\rho_i} \tag{17}$$

$$x_i = x_i + \Delta t u_i \tag{18}$$

The SPH simulation uses these formulas to compute the positions of the particles for a given time length through an iterative procedure. The particles' interactions are very important: we use a rather small support (small h value) for the kernel function in order to limit the number n of neighboring particles. Then, in any good implementation, one of the key elements is the neighboring handling. Using a parallel processor, this can be achieved with a low complexity, allowing to reach short computation times.

2.3 SPH algorithms

SPH method presented in Section 2.2 is quite immediate to implement [3]. Using a small kernel support, the calculation of the forces that apply to a given particle is quite fast, since only a few numbers of neighbors have to be considered. Nevertheless, the neighborhood needs to be efficiently computed and stored to accelerate the calculations. This needs to be done for each time step. To do that, a regular grid is the faster solution. The size of a grid cell is set as the radius of the kernel support. Then, to find the neighbors of a given particle, it is enough to consider the cells

surrounding the one containing this particle. In dimension 2 this leads to 9 cells (including the cell containing the particle) and 27 in dimension 3.

The SPH method described in Section 2.2 has been extended to solve some accuracy problem with incompressible fluids, for instance, predictive-corrective incompressible SPH (PC-ISPH), incompressible SPH (ISPH), and implicit incompressible SPH (IISPH) [5–7]. In Ref. [7], comparisons between these three techniques show that IISPH is faster than PC-ISPH and ISPH, mainly since it allows to use bigger time steps. Hence, this chapter focusses on an implementation of IISPH. This evolved method is also more complex than classical SPH, and then each time step uses more calculations (but they are longer, so it is faster still). More precisely, for each particle it calculates the density ρ_i and the forces of viscosity, surface tension, and gravity like in SPH method. It adds the calculation of the advection velocity, which is the portion of the velocity independent to the pressure exerted by the other particles:

$$u_i^{adv} = u_i + \frac{\Delta t}{m_i}\left(f_i^{viscosity} + f_i^{surface} + f_i^{gravity}\right) \tag{19}$$

The IISPH algorithm calculates the advection factor d_{ii} and the advection coefficient a_{ii}:

$$\mathrm{d_{ii}} = -\Delta \mathrm{t}^2 \sum_j^n \frac{m_j}{\rho_i^2} W_h(x_i - x_j) \tag{20}$$

$$a_{ii} = \sum_j^n m_j (d_{ii} - d_{ji}) \Delta W_h(x_i - x_j) \tag{21}$$

The IISPH algorithm continues with the calculation of pressure's forces. It is done through at least two corrective loops to enforce the minimization of the difference between the rest density and the sum of the density of all particles. First, this loop calculates the advection density:

$$\rho_i^{adv} = \rho_i + \Delta t \sum_j^n m_j \left(u_i^{adv} - u_j^{adv}\right) \cdot \Delta W_h(x_i - x_j) \tag{22}$$

Second, it calculates the following term per particle that will be used many times in the next steps:

$$\sum_{\mathrm{j}}^{\mathrm{n}} \mathrm{d_{ij}} \mathrm{p_j^l} = \Delta \mathrm{t}^2 \sum_{\mathrm{j}}^{\mathrm{n}} -\frac{m_j}{\rho_j^2} \mathrm{p_j^l} \Delta \mathrm{W_h}(\mathrm{x_i} - \mathrm{x_j}) \tag{23}$$

where l is the iteration number of the corrective loop. Notice that for $l = 0$, IISPH uses $p_i^0 = \omega p_i$, with $\omega = 0.5$.

Then, the IISPH corrective loop continues by computing for each particle the pressure force thanks to the following expression:

$$f_i^{pressure} = \sum_j^n -m_i m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right) \Delta W_h(x_i - x_j) \tag{24}$$

where p_i is the pressure at a particle i:

$$p_i = (1 - \omega) p_i^l + \frac{\omega}{a_{ii}} (\rho_0 - \rho_i^{adv} - \sum p_i^l) \tag{25}$$

This last term is computed using the displacement factors:

$$\sum p_i^l = \sum_j^n m_j \left(\sum_j^n d_{ij} p_j^l - d_{jj} p_j^l - \sum_{k \neq i}^n d_{jk} p_k^l \right) \cdot W_h(x_i - x_j) \tag{26}$$

All these calculations should be made in parallel to reduce the computation times, using a tuned implementation, for instance, using message passing interface (MPI) for high-performance computing (HPC) or using the Nvidia common unified device architecture (CUDA) on graphics processing unit (GPU) for simpler computers.

3. Parallel SPH implementation

An efficient SPH implementation relies on parallelism at some level. A fully parallel solution may become a very efficient solution, as previous works have shown it. While most of the calculations may be done considering a single particle into a single core, finding the neighboring particles that play a role in the density, the pressure, and the external forces needs collaboration between different cores.

Using the texture mechanism available with GPU, working with the neighbors is quite simple and efficient. Nevertheless, this implies to store all the particles into a regular grid at each time step during the simulation. This part is somewhere the most complicated, and the key step for an efficient implementation.

This section first presents the main parallel patterns (MAP, SORT, SCAN, etc.) and then shows how they can be combined to write a new fast parallel SPH solver.

3.1 Parallel patterns

Writing a parallel algorithm is not as simple as writing a sequential algorithm. This truism is based on the necessary consideration of the collaborations between the different processors of a parallel machine: all the processors must work in concert, and not isolated as in a sequential approach. These collaborative aspects are the main difficulty. How to make sure all these processors expect when it's needed and work to the fullest when no synchronization is required?

Rather than writing a parallel algorithm based on classical sequential patterns, parallel patterns make it possible to write a parallel algorithm directly, abstracting the underlying machine. These patterns rely on very simple parallel architecture, called the parallel random-access memory (PRAM). It assumes a synchronization between an infinite set of processors and an infinite amount of memory [8].

3.1.1 Simple parallel patterns

Simple parallel patterns do not need synchronization. This means that, using a GPU or an HPC, they may be run without any difficulties, even with less processors than needed. The simpler one is the MAP, or transform, that consists in applying a given function f to an input data to obtain the output. The key of this pattern is about the localisation of the data: input and output are generally considered as vectors (or arrays). Then, MAP applied to data at the same index:

$$Y_i = \mathrm{f}(X_i) \quad (\mathrm{MAP}) \tag{27}$$

Figure 1 describes this pattern on small arrays.

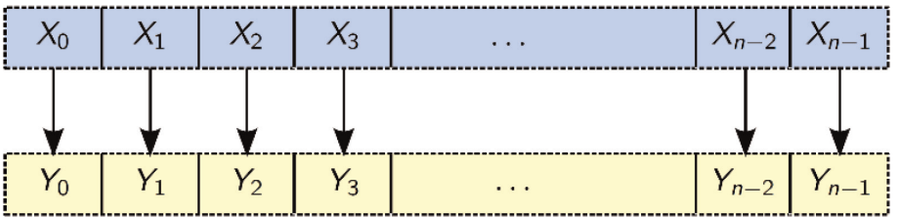


Figure 1.
Illustration of the MAP parallel pattern.

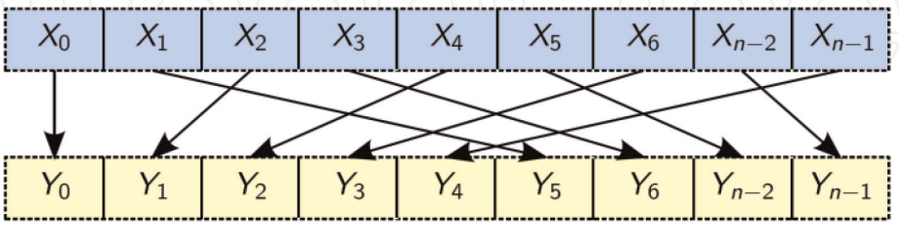


Figure 2.
The SCATTER and GATHER patterns move data using a permutation.

In many occasions, it is necessary to write the result at a new location, another index. When each possible destination index is used once and only once, we obtain a quite simple parallel pattern called SCATTER. It consists of writing the input data from location i to the destination location $map(i)$, map being a permutation function. **Figure 2** illustrates this parallel pattern.

In the same spirit, the GATHER parallel pattern writes at index i data coming from index $map(i)$, using again a permutation function. To differentiate between a SCATTER and a GATHER, you should remember that at first we read contiguous data, while in the second, we write at contiguous location. This is resumed with the following two expressions:

$$Y_{map(i)} = X_i \quad \text{(SCATTER)} \tag{28}$$

$$Y_i = X_{map(i)} \quad \text{(GATHER)} \tag{29}$$

PRAM model is very useful to write efficient algorithms on theory. Nevertheless, at the end these algorithms run on real computers, with a limited amount of memory and a fixed number of processors. Brent's theorem links the theoretical computation time on PRAM model with the one obtained using only p processors: an algorithm made in $O(1)$ using m processors will run in $O\left(\frac{m}{p}\right)$ using only p processors. This allows to predict the behavior of (very) simple algorithm on a GPU.

3.1.2 Advanced parallel patterns

In many cases, some degree of collaboration is needed between processors. This leads to some more complicated parallel patterns. A very common parallel pattern using such a collaboration is the SORT that sorts data according to a given order. It is used in previous SPH implementation for building the neighbors' grid. The SORT pattern is based on the PARTITION pattern that moves values with respect to a given predicate. More precisely, for n values X_i and using the predicate $P_i \in [0, 1]$, the PARTITION pattern moves the values X_i for which $P_i = 1$ at the beginning of the resulting array, the others at the end (see **Figure 3** for a simple example).

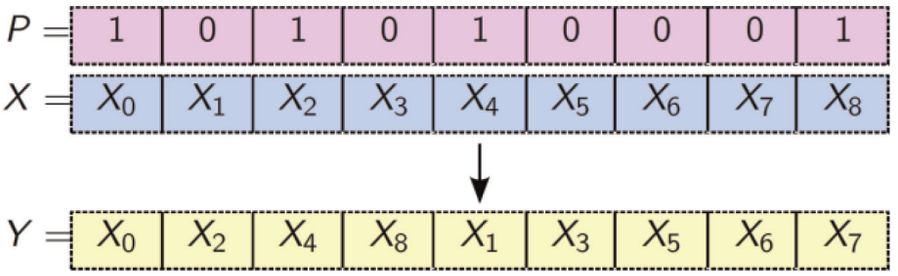


Figure 3.
Illustration of the PARTITION pattern for nine input values; the values with predicate 1 are put at the beginning of the output, the others at the end.

These complex patterns are built using a fundamental pattern called SCAN. It corresponds to a prefix sum of values, according to the following expression:

$$Y_i = \bigoplus_{j=0}^{i} X_j \quad (\text{INCLUSIVE} - \text{SCAN}) \tag{30}$$

The fundamental pattern exists in two versions: inclusive and exclusive ones. The first corresponds to the expression given above, doing a sum-up to the current output position. The exclusive version omits the current position, doing a sum-up to $i-1$ and using a nil value for Y_0 (generally, using 0):

$$Y_i = \bigoplus_{j=0}^{i-1} X_i \quad (\text{EXCLUSIVE} - \text{SCAN}) \tag{31}$$

Figure 4 shows that these two versions of SCAN are almost the same, except the shift between the resulting arrays: the values obtained with inclusive version correspond to the ones obtained with the exclusive version at the same position plus one.

Another pattern of interest into this chapter is the REDUCE that allows to calculate a single value from an array of values and using any given associative binary function:

$$Y = \bigoplus_{j=0}^{n-1} X_i \quad (\text{REDUCE}) \tag{32}$$

For instance, using $X = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ and the classical integer sum as binary operator, this pattern returns $Y = 55$, the sum of the 10 first non-nil integers.

These complex patterns have roughly speaking all the same complexity, in $O(\log n)$ on a PRAM machine and $\mathrm{O}\left(\frac{n}{p}\log\frac{n}{p}\right)$ using p processors only. Nevertheless, since they are built using the SCAN, PARTITION and SORT are in practice more complex and take more time. A fast implementation of the SORT pattern relies on the radix sort algorithm that loops over the number of digits of the maximum key to sort, thus having a practical complexity in $O(32\log n)$ with 32-bit integers.

X_i	1	1	1	1	1	1	1	1
SCAN	1	2	3	4	5	6	7	8
PRESCAN	0	1	2	3	4	5	6	7

Figure 4.
Differences between the inclusive and exclusive SCAN patterns.

The last programming tool this section covers is the atomic operation notion. A load-modify-write operation cannot be handled in parallel program without caution. Let us consider two processors doing a "plus one" in parallel at the same time. The addition is done by the CPU using registers (local memory to the CPU). Hence the variables to add need to be loaded from the main memory, then added, and then stored into the main memory. If the two processors do the load-modify-write operation at the same time exactly on the same variable, then the result is false. If the processors are not exactly synchronized, the result is certainly false also: to be correct, the two operations must be done sequentially. Atomic operations provide this behavior, performing the read-modify-write operation for one and only one processor at a time.

Obviously, other parallel patterns exist. They are not discussed in this chapter since they are not used in our SPH implementation.

3.2 Grid building

Previous SPH implementations use the SORT pattern to build the neighbors' grid [7, 9, 10]. The first step consists in calculating the grid index of each particle, using a MAP. Next, the particles are sorted with respect to this index. Then, it is necessary to compute the number of particles per cell and the starting position of each cell. In [9], atomic operations are used for these two operations: the minimum for the first particle into each cell and the addition for the number of particles per cell.

In Ref. [7], authors follow a similar approach with the particle sort with respect to their cell index but using a MAP to mark the start and the end of each cell with respect to the sorted cell indices, considering their unicity.

The main problem is that the sorting algorithm takes a large part of the computation time, near 30% according to [10]. In this chapter, we avoid the full sorting by combining simple parallel patterns and atomic operations. Our grid building algorithm is summarized in **Figure 5**.

This algorithm uses the Nvidia Thrust API with some freedom to shorten it. First, at line 8 the number of particles per grid cell is set to zero. Next, like with previous methods at line 9, the index of each particle is calculated with a MAP. Using a second MAP at line 10, the particle cell offset is calculated using an atomic addition. More precisely, we use the CUDA `int. atomicAdd(int*cc, int. a)` function that adds `a` to the variable `*cc` and returns the old content of `*cc`. Since atomic operations are done in sequence, the number of particles per cell is correctly computed. Moreover, each particle receives the old counter value, which is 0 for the first atomic operation execution, 1 for the second, and so on up to $m - 1$ for the last particle added to the

```
1   void GridBuilding(
2       device_vector<Particle>& Particles,
3       device_vector<unsigned>& cell_start
4   ) {
5       device_vector<Particle> p(Particles);
6       device_vector<unsigned> index(Particles.size());
7       device_vector<unsigned> local_offset(Particles.size());
8       memset(local_offset, 0, local_offset.size());
9       transform(p, index, Particle2Index());
10      transform(index, local_offset, Index2LocalOffset());
11      exclusive_scan(local_offset, cell_start);
12      transform(index, local_offset, local_offset, Local2Global(cell_start));
13      scatter(p, local_offset, Particles);
14  }
```

Figure 5.
Our algorithm to build the neighbors' grid.

cell, *m* being the number of particles added into the cell. These values are used to scatter all the particles to their local position into the grid, at line 13. But, before to do that we need to calculate the global grid offset, corresponding to the position of the first particle of each cell. This is done using an exclusive SCAN at line 11 to compute the global offset, followed by a MAP at line 12 to calculate each particle global offset.

In practice, this algorithm can be optimized in many ways. First, the device vectors can be allocated only once, and not each time the grid is built. Second, the first two transforms (lines 9 and 10) can be mixed into one. This will limit the memory loading into device registers, known as a major performance limitation with GPU. At last, the last transform (line 12) and the scatter (line 13) can be mixed into a single call again to minimize the memory bandwidth usage. Moreover, the particles' data must be split into multiple arrays for efficiency (one array for position, one for density, one for pressure, etc.) as in [10].

3.3 Main algorithm

The most difficult part of the implementation of the IISPH method is the construction of the neighbors' grid, as for any non-mesh density kernel method. The rest of the calculation is rather simple and relies on two parallel models: the MAP for all the loops on particles and the REDUCE to control the termination of the corrective loop in the calculation of the pressure force.

It is noticeable that the IISPH loop to correct the pressure force runs on the CPU, because there is no available global synchronization on the GPU. Then, the REDUCE is used to return a value from the GPU to the CPU, to decide if more corrections are needed or not. Nevertheless, since this just consists of sending one real value, it is not a big bottleneck.

Moreover, many calculations use data from the neighbors (pressure, density, position, etc.). L1 GPU's memory is used to accelerate these calculations, reducing the computation time around a third in our experiment. Notice also that the IISPH corrective loop amortizes the neighbors' grid building. In our experiments the grid building now represents less than 10 percent of the full computation time.

4. Experiments

The IISPH is a valid solution to simulate a tsunami [11]. Its main advantage regarding a discrete method is that it does not need to refine the mesh near the obstacles, like the coast and the buildings. Moreover, the wave can go everywhere, including interfering with the beach, buildings, infrastructure, etc.

In this chapter, we illustrate the tsunami simulation using IISPH algorithm through a rather simple scenario. It contains a short coast ending with a mountain. We put a building just after the beach. The main difficulty, if either, consists of generating the solitary wave. A tsunami, for instance, is generated by an earthquake at long distance. The produced wave runs at 200 meter per second (720 km/h). We do not need to simulate the propagation of the wave since its epicenter, which is quite difficult with long distance: it needs very long simulation time to see the wave reaching the beach, and obviously it needs a huge amount of memory to handle the sea between the two distant locations. Instead, we simulate the wave into a rather small space. We can predict the time of arrival to the beach, assuming we know the exact distance between the beach and the earthquake location.

In [11], authors solve the solitary wave solution of Boussinesq. They calculate the wave paddle displacement using the equation:

$$X(t) = Ct - \frac{\theta}{\kappa} \tag{33}$$

where C is the wave velocity, t is the time in the simulation frame, κ is the decay coefficient, and θ is given using Newton's method by

$$\theta^{l+1} = \theta^{l} - \frac{\theta^{l} - \kappa Ct + \frac{H}{h}\tan\left(\theta^{l}\right)}{l + \frac{H}{h}\operatorname{sech}^{2}\left(\theta^{l}\right)} \tag{34}$$

More precisely, θ is the solution of the following problem:

$$X(t) = \frac{H}{\kappa h}\tanh(\kappa(Ct - X(t))) \tag{35}$$

where H and h are the wave height and the water depth, respectively.

While this method works on CPU, it is not well-suited for a CUDA implementation of the IISSPH, mainly because the number of iterations of the Newton-Raphson method depends on the input values, and so is not constant per particle.

Hence, in this chapter we use a different but simple technique. The wave is produced using a piston wave generator. Here, the piston is a huge virtual object that moves the water to reach the speed of the wave. The length and the speed of the piston movement are calibrated to obtain the good height and speed of the tsunami solitary wave.

Figure 6 illustrates such a simple wave simulation, before the tsunami wave arrives. **Figure 7** shows the wave arriving at the building at $t = 4$ s in the simulation frame. In **Figure 8**, at $t = 8$ s, the building is completely below the ocean that returns

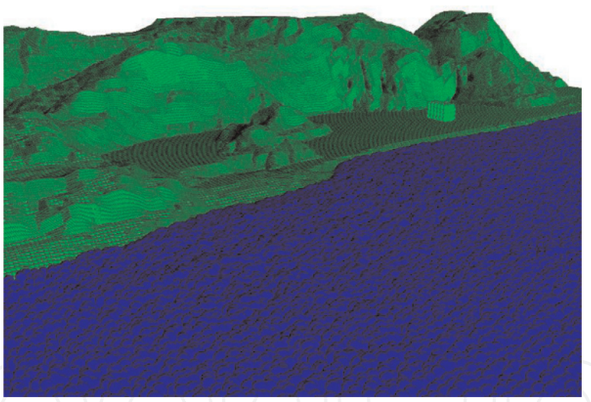


Figure 6.
Tsunami simulation before the solitary wave arrives.

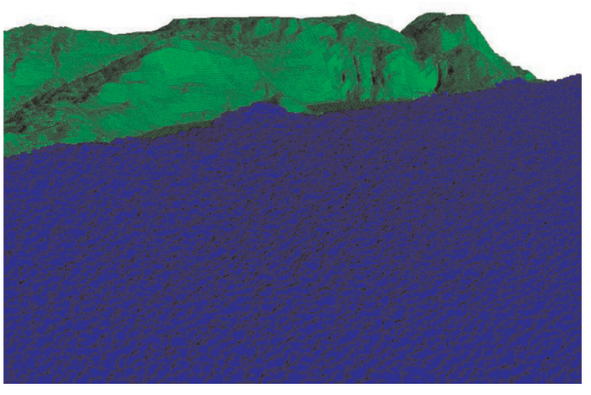


Figure 7.
Tsunami wave reaching the building near the beach.

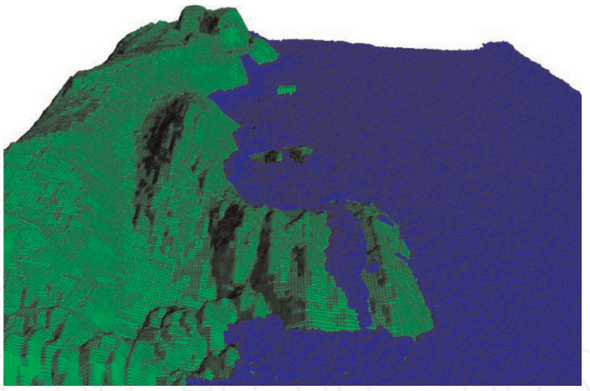


Figure 8.
The tsunami wave engulfed the building and the coast.

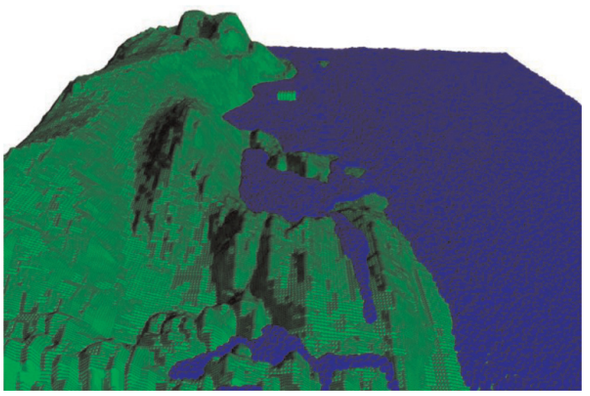


Figure 9.
The tsunami wave begins to pull off the coast. With flat coasts, this may take some time.

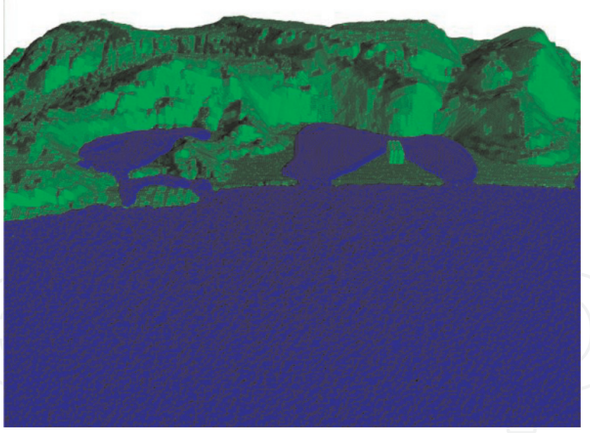


Figure 10.
After a longer time, the tsunami wave has almost completely disappeared.

into its bed after some more time (see **Figures 9** and **10**). The building is made with fixed particles that nevertheless are considered with the moving particles of the fluid. This allows an interaction between two kinds of particles, and it permits to obtain the pressure and force applied to the building, for instance, to check if it will resist or not.

5. Conclusions

This chapter focusses on the simulation of a tsunami solitary wave. Such a wave is mainly produced by submarine earthquake and may provoke vast disasters for

human living near the coasts. Such phenomena also may produce strong degradation on buildings and structures, in turn inducing human loss as what happened after the Tohoku earthquake in 2011. To avoid these disasters, it is important to be able to validate the robustness of structures and buildings near the dangerous coasts and to inform population after an always unpredictable submarine earthquake.

To achieve these goals, it is necessary to produce robust and fast fluid simulator software. To simulate a tsunami wave, a good candidate is the SPH method. Since it does not need the usage of a fix mesh like in discrete techniques, it allows to handle correctly the wave running on the beach and after. Moreover, it correctly handles the contact with buildings and structures, allowing to simulate the forces that they undergo.

This chapter recalls the implicit incompressible SPH method, which is one of the fastest among the SPH ones. The parallel implementation for GPU is detailed in depth, with a fast algorithm to build the neighbors' grid, avoiding the classical sorting method which is more time-consuming.

At last, this chapter proposes a simple tsunami wave simulation using a piston wave generator, a simple solution for implementing and providing valuable results. It can be used to simulate tsunami generated by submarine earthquake occurring in a pattern of seismic source mechanism when both the location and intensity are estimated.

Author details

Pierre Thuillier Le Gac[1], Emmanuelle Darles[1], Pierre-Yves Louis[2] and Lilian Aveneau[1*]

1 XLIM, UMR 7252, CNRS, University of Poitiers, France

2 LMA, UMR 7348, CNRS, University of Poitiers, France

*Address all correspondence to: lilian.aveneau@univ-poitiers.fr

IntechOpen

© 2019 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. CC BY

References

[1] Gingold RA, Monaghan JJ. Smoothed particle hydrodynamics: Theory and application to non-spherical stars. Monthly Notices of the Royal Astronomical Society. 1997;**181**(3): 375-389. DOI: 10.1093/mnras/181.3.375

[2] Lucy LB. A numerical approach to the testing of the fission hypothesis. The Astronomical Journal. 1977;**82**: 1013-1024

[3] Müller M, Charypar D, Gross M. Particle-based fluid simulation for interactive applications. In: Proceedings of the 2003 ACM SIGGRAPH/ Eurographics Symposium on Computer Animation. 2003. pp. 154-159

[4] Silverman B. Density Estimation for Statistics and Data Analysis. New York: Routledge; 1998. DOI: 10.1201/ 9781315140919

[5] Solenthaler B, Pajarola R. Predictive-corrective incompressible SPH. In: Hoppe H, editor. ACM SIGGRAPH 2009 papers (SIGGRAPH '09). New York, NY, USA: ACM; 2009. 6 p. DOI: 10.1145/1576246.1531346

[6] Shao S, Lo EYM. Incompressible SPH method for simulating Newtonian and non-Newtonian flows with a free surface. Advances in Water Resources. 2003;**26**(6):787-800. DOI: 10.1016/ S0309-1708(03)00030-7

[7] Ihmsen M, Cornelis J, Solenthaler B, Horvath C, Teschner M. Implicit incompressible SPH. IEEE Transactions on Visualization and Computer Graphics. 2014;**20**(3):426-435. DOI: 10.1109/TVCG.2013.105

[8] Blelloch GE. Vector Models for Data-Parallel Computing. Cambridge, Massachusetts, London, England: MIT Press; 1990

[9] Goswami P, Schlegel P, Solenthaler B, Pajarola R. Interactive SPH simulation and rendering on the GPU. In: Eurographics/ACM SIGGRAPH Symposium on Computer Animation; 2010. pp. 1-10

[10] Bilotta G, Zago V, Hérault A. Design and implementation of particle systems for meshfree methods with high performance. In: High Performance Parallel Computing. IntechOpen; 2018. DOI: 10.5772/intechopen.81755

[11] Sampath R, Montanari N, Akinci N, Prescott S, Smith C. Large-scale solitary wave simulation with implicit incompressible SPH. Journal of Ocean Engineering and Marine Energy. 2016; **2**(3):313-329. DOI: 10.1007/s40722-016-0060-8