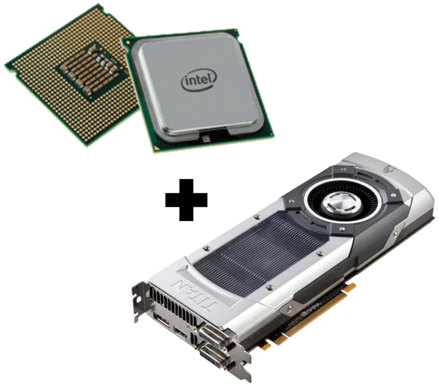
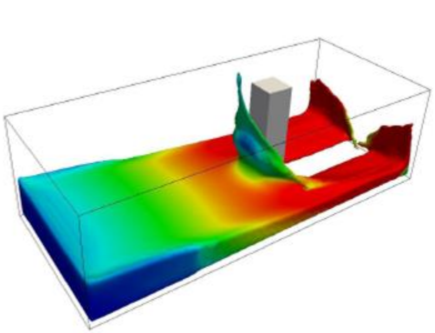
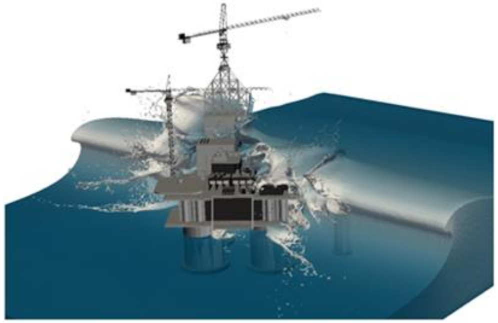
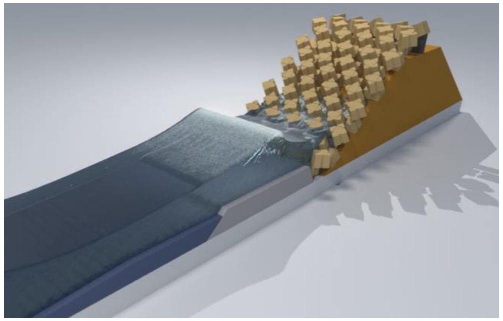
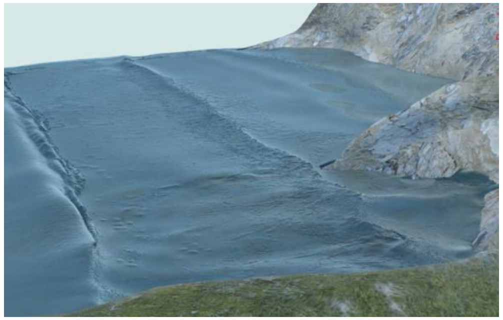
Optimisation and SPH Tricks











José Manuel Domínguez Alonso

jmdominguez@uvigo.es

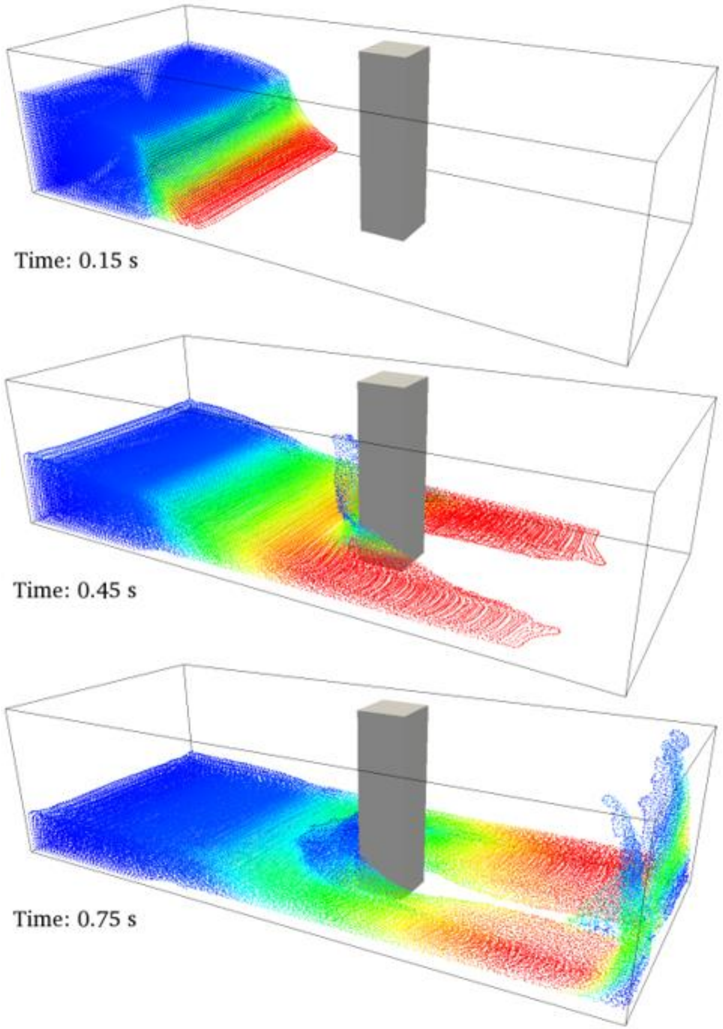
EPHYSLAB, Universidade de Vigo, Spain

Outline

1. **Introduction**
    - **1.1. Why is SPH too slow?**
    - **1.2. High Performance Computing (HPC)**
    - **1.3. DualSPHysics project**
2. DualSPHysics implementation
    - 2.1. Implementation in three steps
    - 2.2. Neighbour list approaches
3. CPU acceleration
4. GPU acceleration
    - 4.1. Parallelization problems in SPH
    - 4.2. GPU optimisations
5. Multi-GPU acceleration
    - 5.1. Dynamic load balancing
    - 5.2. Latest optimisations in Multi-GPU
    - 5.3. Large simulations
6. Future improvements

1.1. Why is SPH too slow?

The SPH method is very expensive in terms of computing time.



For example, a simulation of this dam break

300,000 particles

+

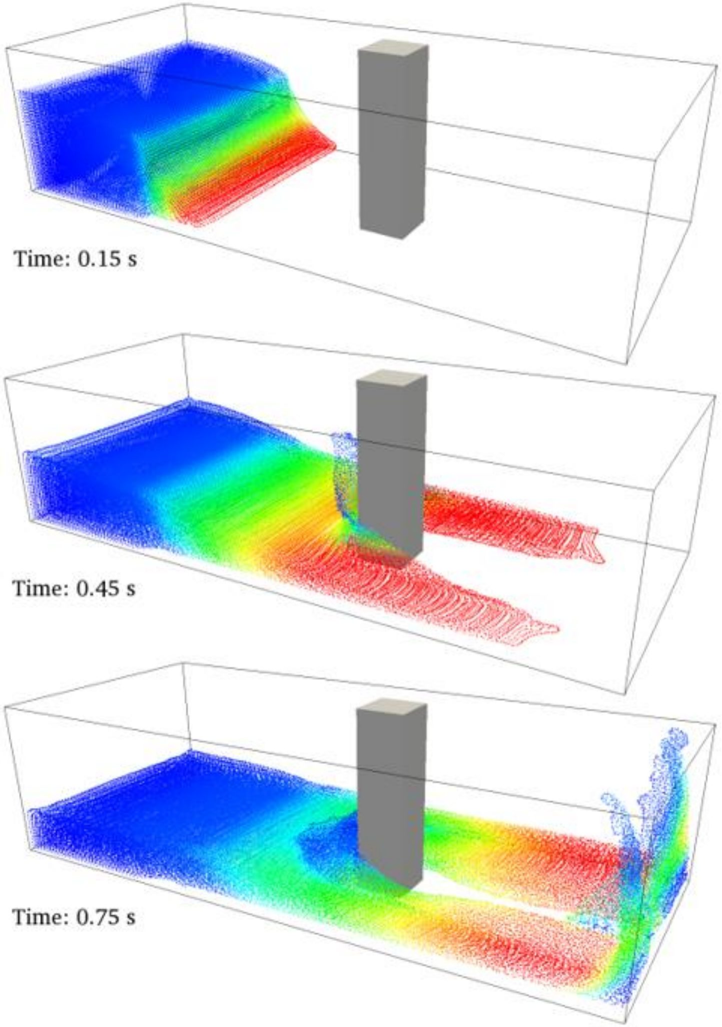
1.5 s (*physical time*)

→

Takes more than
15 hours
(*execution time*)

1.1. Why is SPH too slow?

The SPH method is very expensive in terms of computing time.

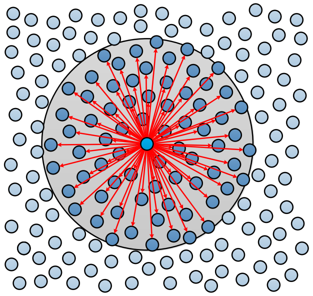


For example, a simulation of this dam break

300,000 particles ✚ **1.5 s** (*physical time*) ➡ Takes more than **15 hours** (*execution time*)

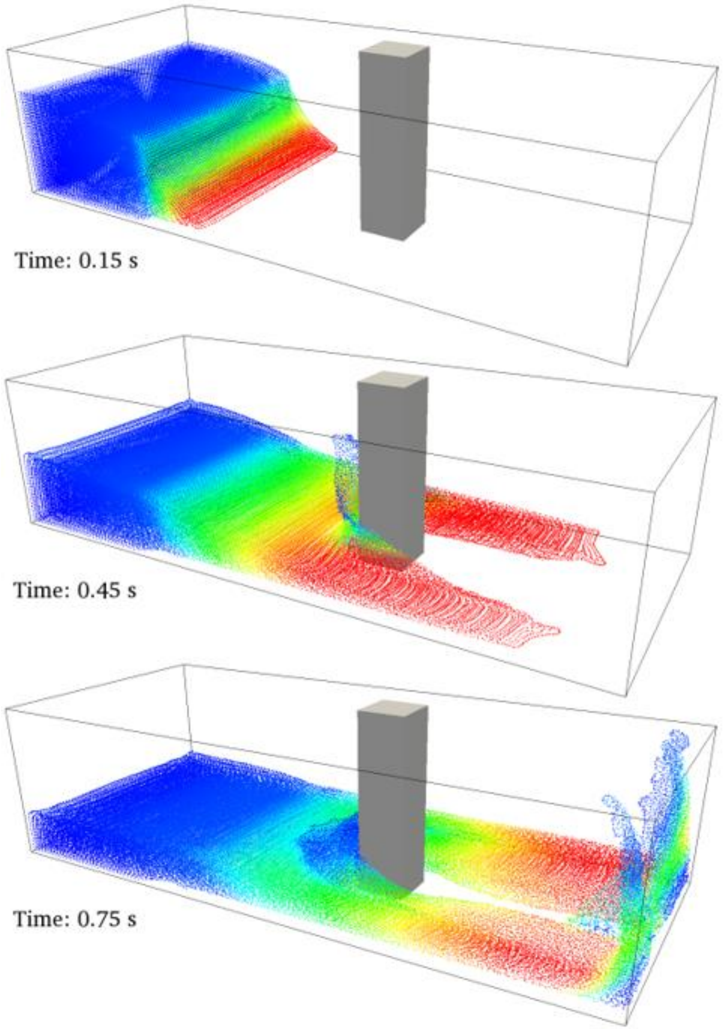
because:

- Each particle interacts with **more than 250 neighbours**.



1.1. Why is SPH too slow?

The SPH method is very expensive in terms of computing time.



For example, a simulation of this dam break

300,000 particles

+

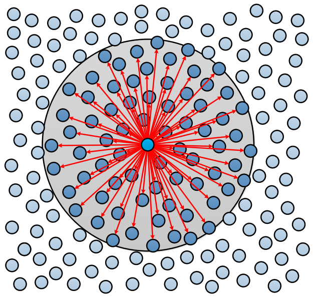
1.5 s (*physical time*)

➡

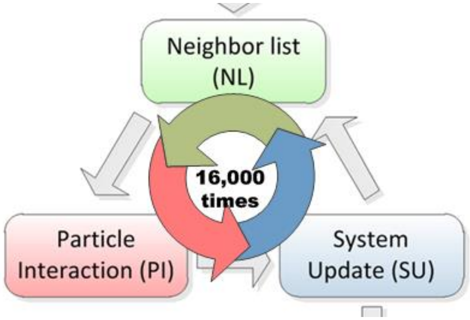
Takes more than
15 hours
(*execution time*)

because:

- Each particle interacts with **more than 250 neighbours**.



- **Δt=10^{-5}-10^{-4} so more than 16,000 steps** are needed to simulate 1.5 s of physical time.



1.1. Why is SPH too slow?

Drawbacks of SPH:

- SPH presents a **high computational cost** that increases when increasing the number of particles.

✚

- The simulation of **real problems** requires a high resolution which implies simulating **millions of particles**.

⬇

The **time required** to simulate a few seconds is **too large**. One second of physical time can take several days of calculation.

IT IS NECESSARY TO USE HPC TECHNIQUES TO REDUCE THESE COMPUTATION TIMES.

1.2. High Performance Computing (HPC)

HPC includes multiple techniques of parallel computing and distributed computing that allow you to execute several operations simultaneously.

The main techniques used to accelerate SPH are:

- **OpenMP** (Open Multi-Processing)
  - Model of parallel programming for systems of shared memory.
  - Portable and flexible programming interface using directives.
  - Its implementation does not involve major changes in the code.
  - The improvement is limited by the number of cores.

Multi-core processor

OpenMP is the best option to optimize the performance of the multiple cores of the current CPUs.

1.2. High Performance Computing (HPC)

HPC includes multiple techniques of parallel computing and distributed computing that allow you to execute several operations simultaneously.

The main techniques used to accelerate SPH are:

- **MPI** (Message Passing Interface)
  - Message-passing library specification for systems of distributed memory: parallel computers and clusters.
  - Several processes are communicated by calling routines to send and receive messages.
  - The use of MPI is typically combined with OpenMP in clusters by using a hybrid communication model.
  - Very expensive for a small research group.

MPI cluster

MPI IS THE BEST OPTION TO COMBINE THE RESOURCES OF MULTIPLE MACHINES CONNECTED VIA NETWORK.

1.2. High Performance Computing (HPC)

HPC includes multiple techniques of parallel computing and distributed computing that allow you to execute several operations simultaneously.

The main techniques used to accelerate SPH are:

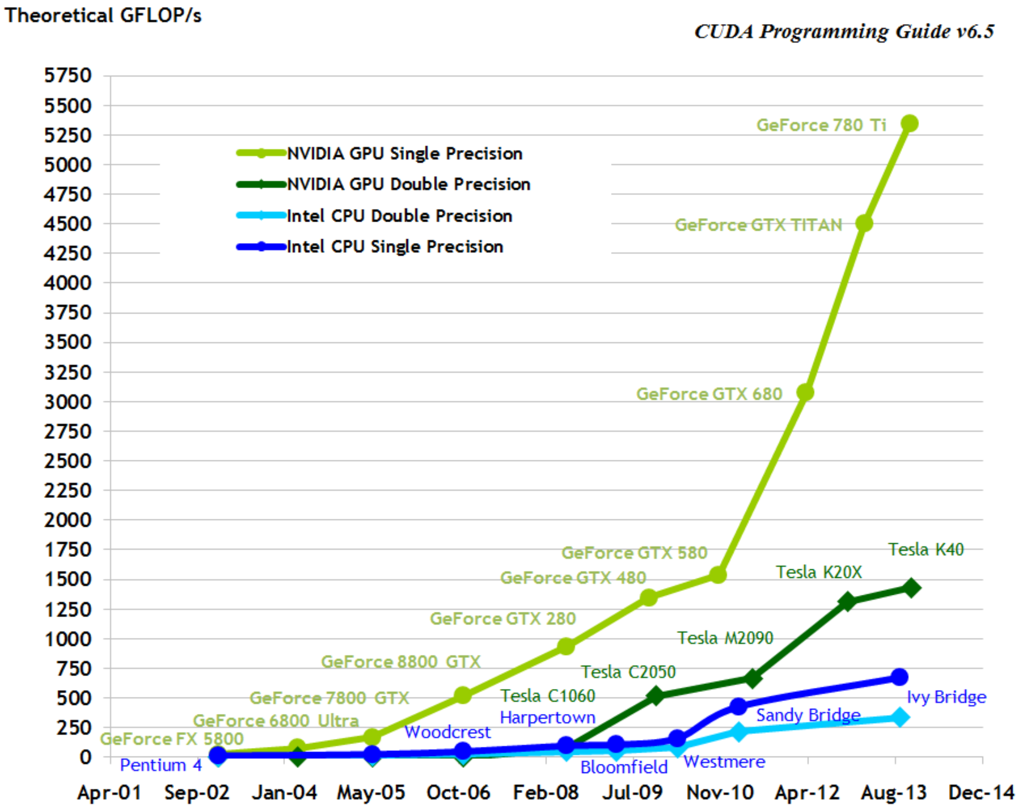
- **GPGPU** (General-Purpose Computing on Graphics Processing Units)
  - It involves the study and use of parallel computing ability of a GPU to perform general purpose programs.
  - New general purpose programming languages and APIs (such as Brook and **CUDA**) provide an easier access to the computing power of GPUs.
  - New implementation of the algorithms used in CPU is necessary for an efficient use in GPU.

GPU

1.2. High Performance Computing (HPC)

Graphics Processing Units (GPUs)

- powerful parallel processors
- designed for graphics rendering
- their computing power has increased much faster than CPUs.



CUDA Programming Guide v6.5

Advantages: GPUs provide a high calculation power with very low cost and without expensive infrastructures.

Drawbacks: An efficient and full use of the capabilities of the GPUs is not straightforward.

Outline

1. Introduction
   - 1.1. Why is SPH too slow?
   - 1.2. High Performance Computing (HPC)
   - 1.3. DualSPHysics project

2. **DualSPHysics implementation**
   - **2.1. Implementation in three steps**
   - **2.2. Neighbour list approaches**

3. CPU acceleration

4. GPU acceleration
   - 4.1. Parallelization problems in SPH
   - 4.2. GPU optimisations

5. Multi-GPU acceleration
   - 5.1. Dynamic load balancing
   - 5.2. Latest optimisations in Multi-GPU
   - 5.3. Large simulations

6. Future improvements

2. DualSPHysics implementation

First version in late 2009.

It includes **two implementations**:

- **CPU**: C++ and OpenMP.
- **GPU**: CUDA.

Both options optimized for the best performance of each architecture.

Why two implementations?

This code can be used on machines with GPU and without GPU.

It allows us to make a fair and realistic comparison between CPU and GPU.

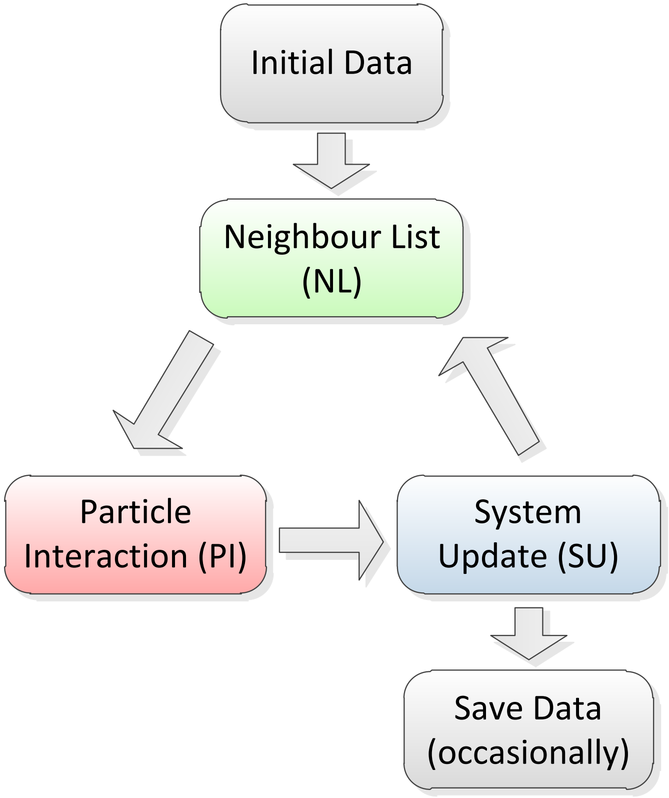
Some algorithms are complex and it is easy to make errors difficult to detect. So they are implemented twice and we can compare results.

It is easier to understand the code in CUDA when you can see the same code in C++.

Drawback: It is necessary to implement and to maintain two different codes.

2.1. Implementation in three steps

For the implementation of SPH, the code is organised in **3 main steps** that are repeated each time step till the end of the simulation.

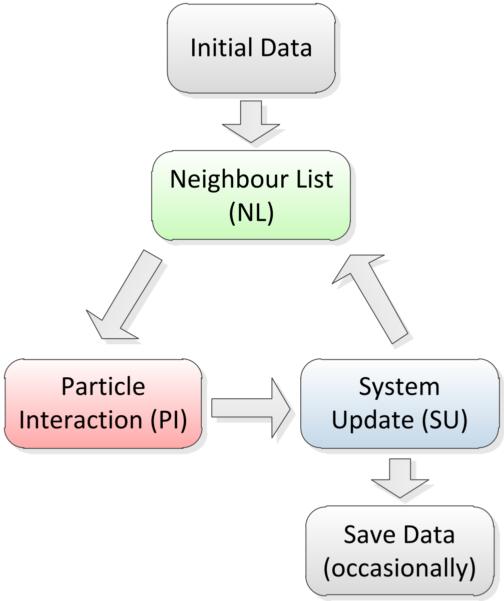


Neighbour list (NL):
Particles are grouped in cells and reordered to optimise the next step.

Particle interactions (PI):
Forces between particles are computed, solving momentum and continuity equations.
This step takes **more than 95%** of execution time.

System update (SU):
Starting from the values of computed forces, the magnitudes of the particles are updated for the next instant of the simulation.

2.2. Neighbour list approaches



Particle Interaction (PI) consumes more than 95% of the execution time. However, its implementation and performance depends greatly on the Neighbour List (NL).

NL step creates the neighbour list to optimise the search for neighbours during particle interaction.

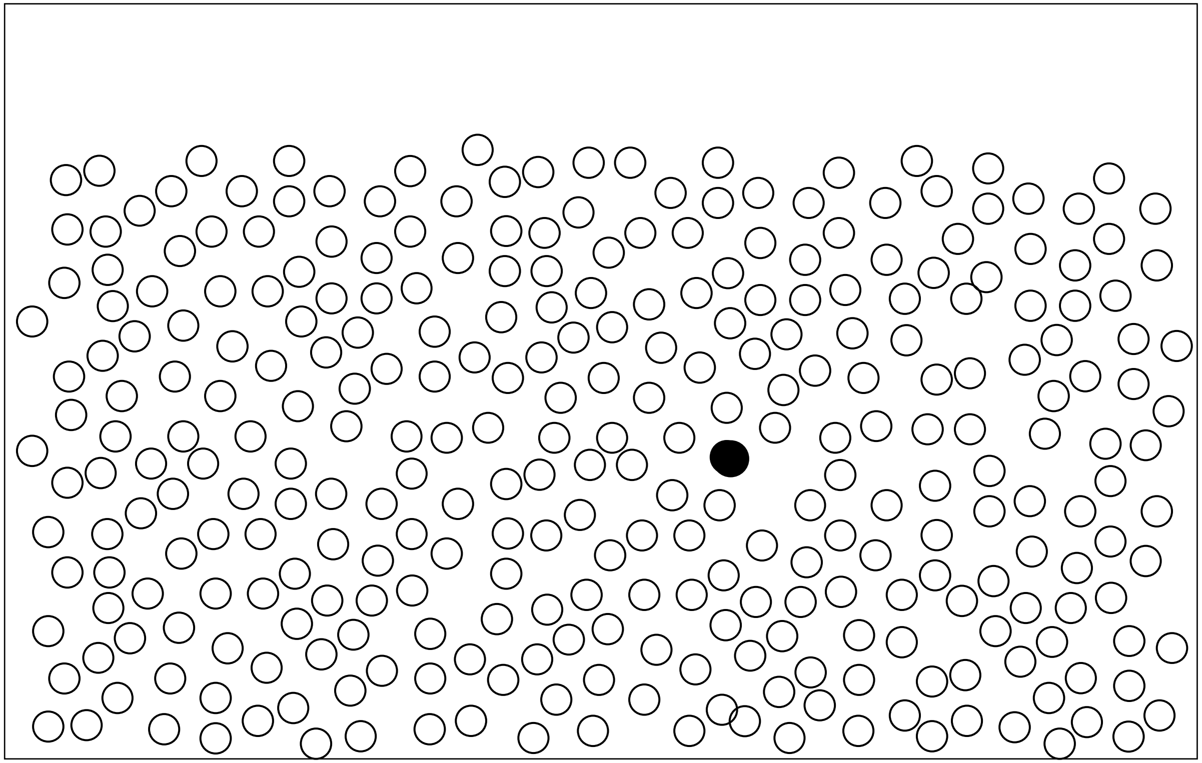
Two approaches were studied:

- **Cell-linked list (CLL)**
- **Verlet list (VL)**
  - **Classical Verlet List (VL_C)**
  - **Improved Verlet List (VL_X)**

2.2. Neighbour list approaches

Cell-linked List (CLL)

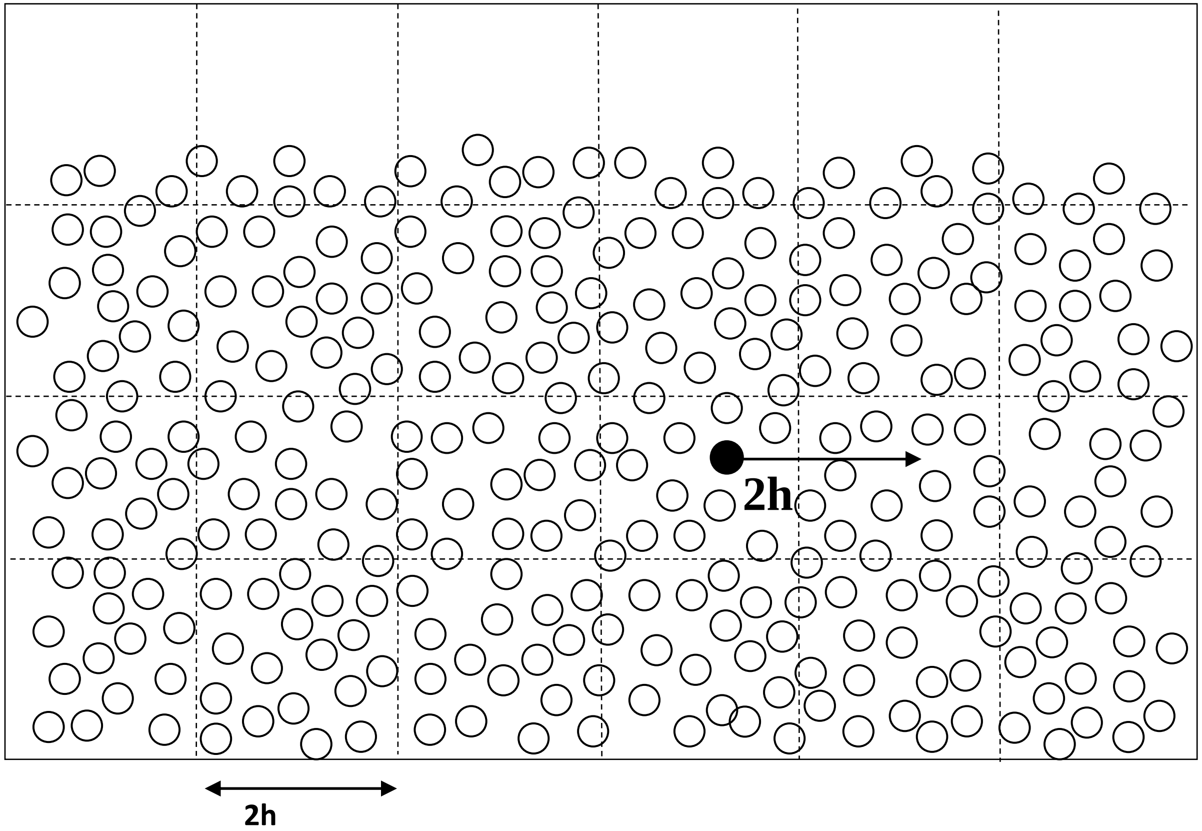
- The computational domain is divided in cells of side *2h* (cut-off limit).
- Particles are stored according to the cell they belong to.
- So each particle only looks for its potential neighbours in the adjacent cells.



2.2. Neighbour list approaches

Cell-linked List (CLL)

- The computational domain is divided in cells of side *2h* (cut-off limit).
- Particles are stored according to the cell they belong to.
- So each particle only looks for its potential neighbours in the adjacent cells.

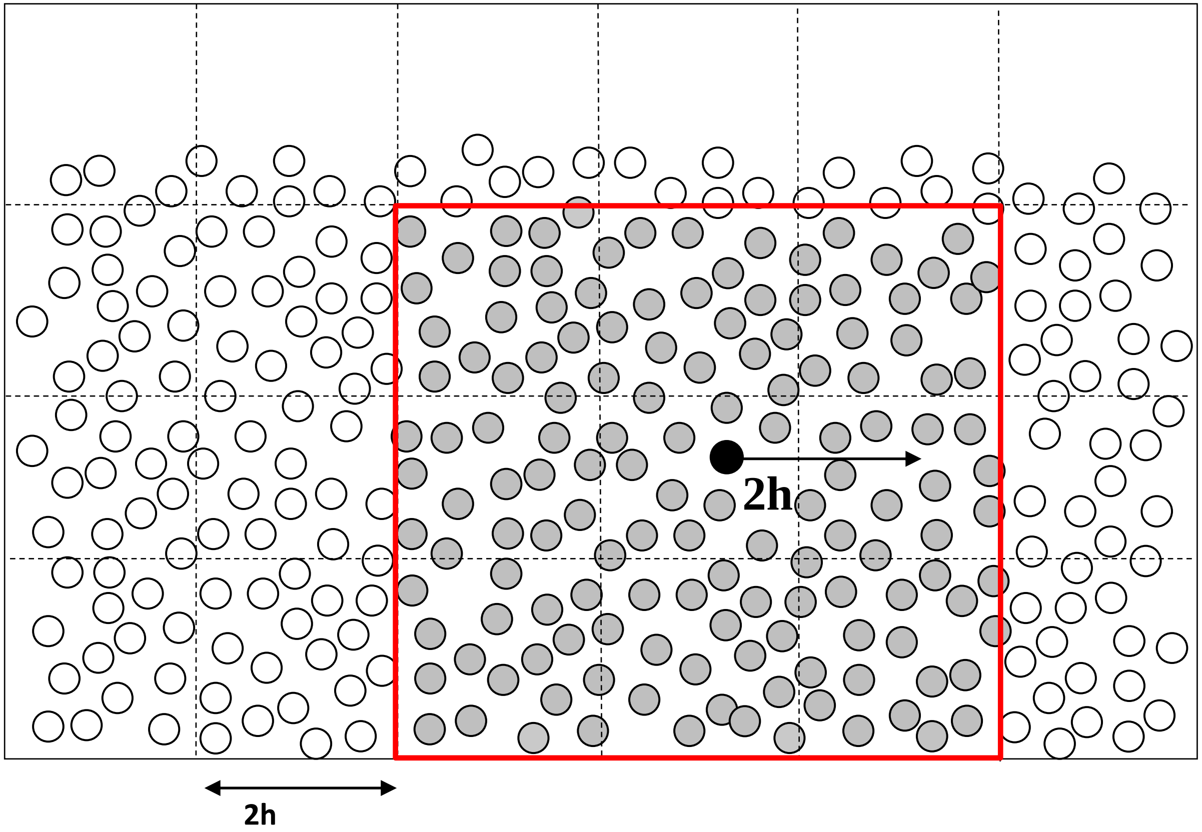


In this example:

2.2. Neighbour list approaches

Cell-linked List (CLL)

- The computational domain is divided in cells of side *2h* (cut-off limit).
- Particles are stored according to the cell they belong to.
- So each particle only looks for its potential neighbours in the adjacent cells.



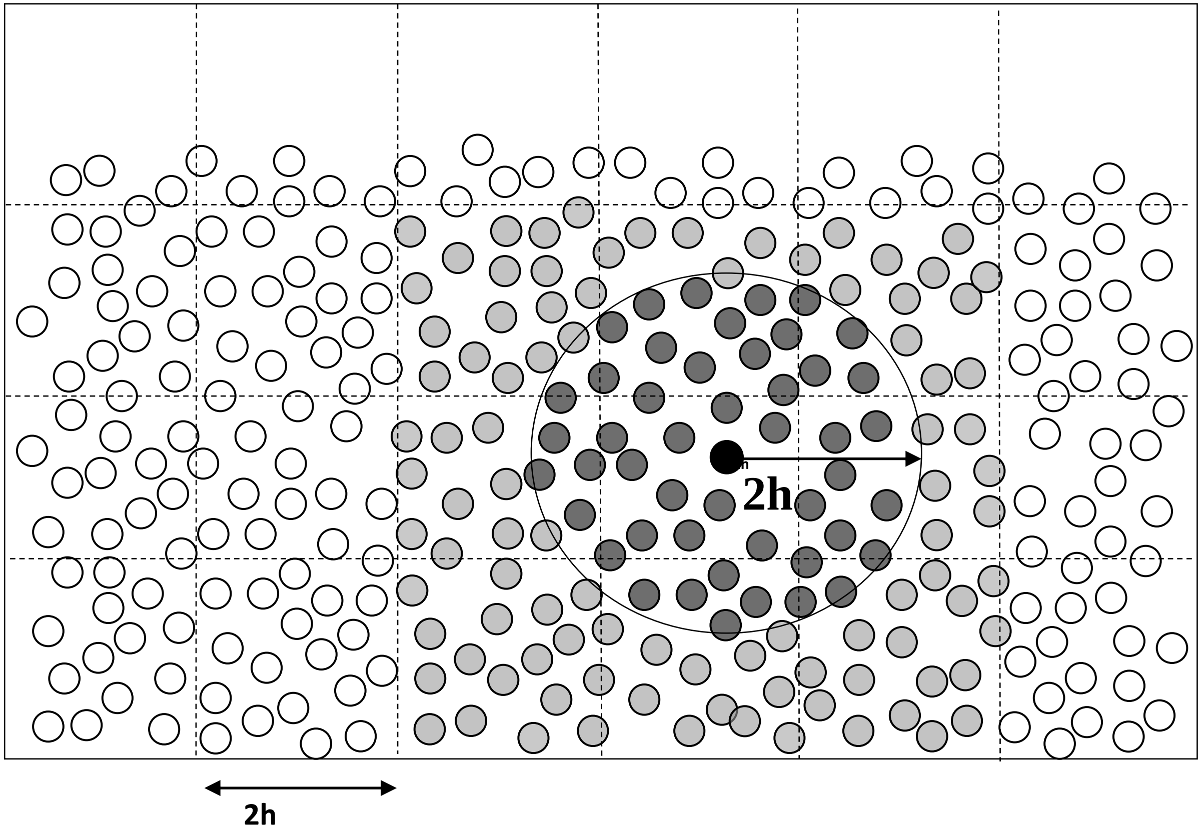
In this example:

141 Potential neighbours
(gray particles)

2.2. Neighbour list approaches

Cell-linked List (CLL)

- The computational domain is divided in cells of side *2h* (cut-off limit).
- Particles are stored according to the cell they belong to.
- So each particle only looks for its potential neighbours in the adjacent cells.



In this example:

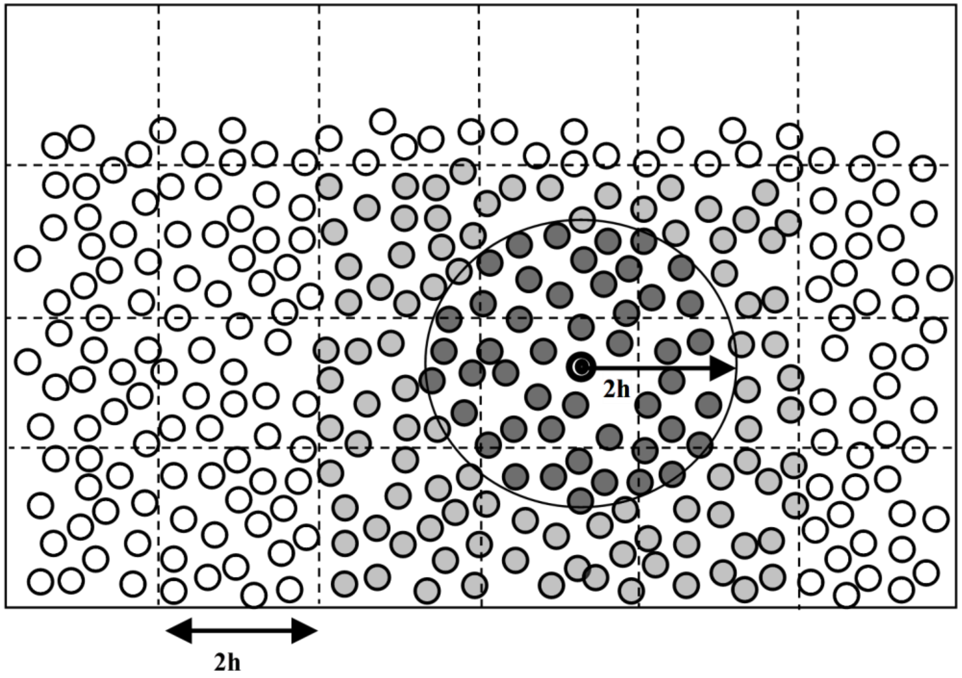
141 Potential neighbours
(gray particles)

47 real neighbours
(dark gray particles)

2.2. Neighbour list approaches

Verlet List

- *The computational domain is divided in cells of side 2h (cut-off limit).*
- *Particles are stored according to the cell they belong to.*
- *So each particle only looks for its potential neighbours in the adjacent cells.*
- **Array of real neighbours is created for each particle.**



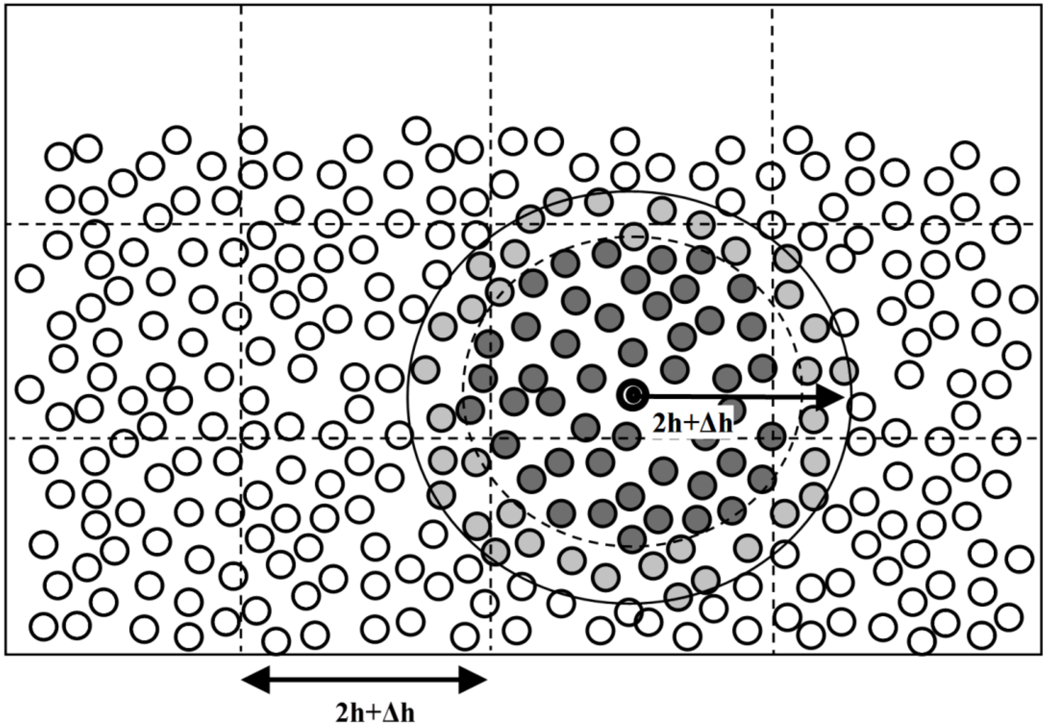
Array with real neighbours of...

a_1	particle a
a_2	
a_3	
a_4	
b_1	particle b
b_2	
b_3	
c_1	particle c
c_2	
c_3	
c_4	
d_1	particle d
d_2	
...	

2.2. Neighbour list approaches

Improved Verlet List (VL_X)

- Δh is calculated in the same way as in VL_C but the number of steps the list is kept (X instead of C) is only tentative.
- The constant v=1 (instead of 1.2) is used because no extra distance is necessary.
- **The same list can be used for several time steps.**



$$\Delta h = v(2 \cdot V_{max} \cdot C \cdot dt)$$

V_{max}: maximum velocity

C: time steps that list is fixed

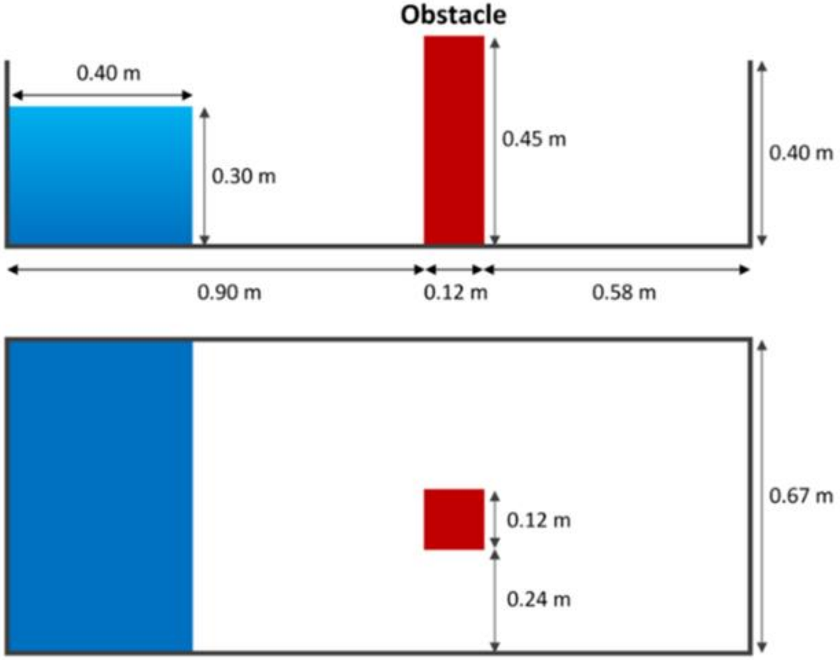
dt: physical time for one time step

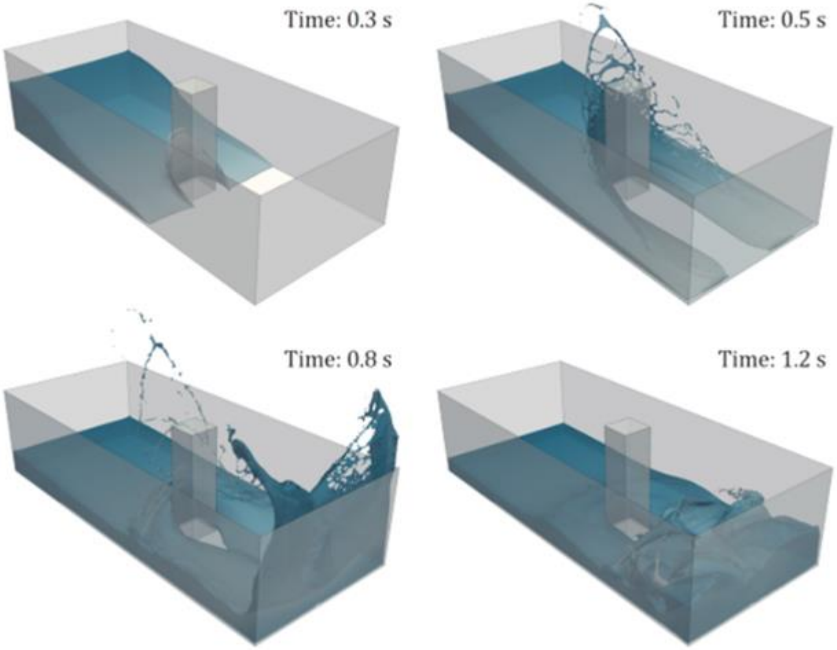
v: constant to remove inaccuracies in calculations

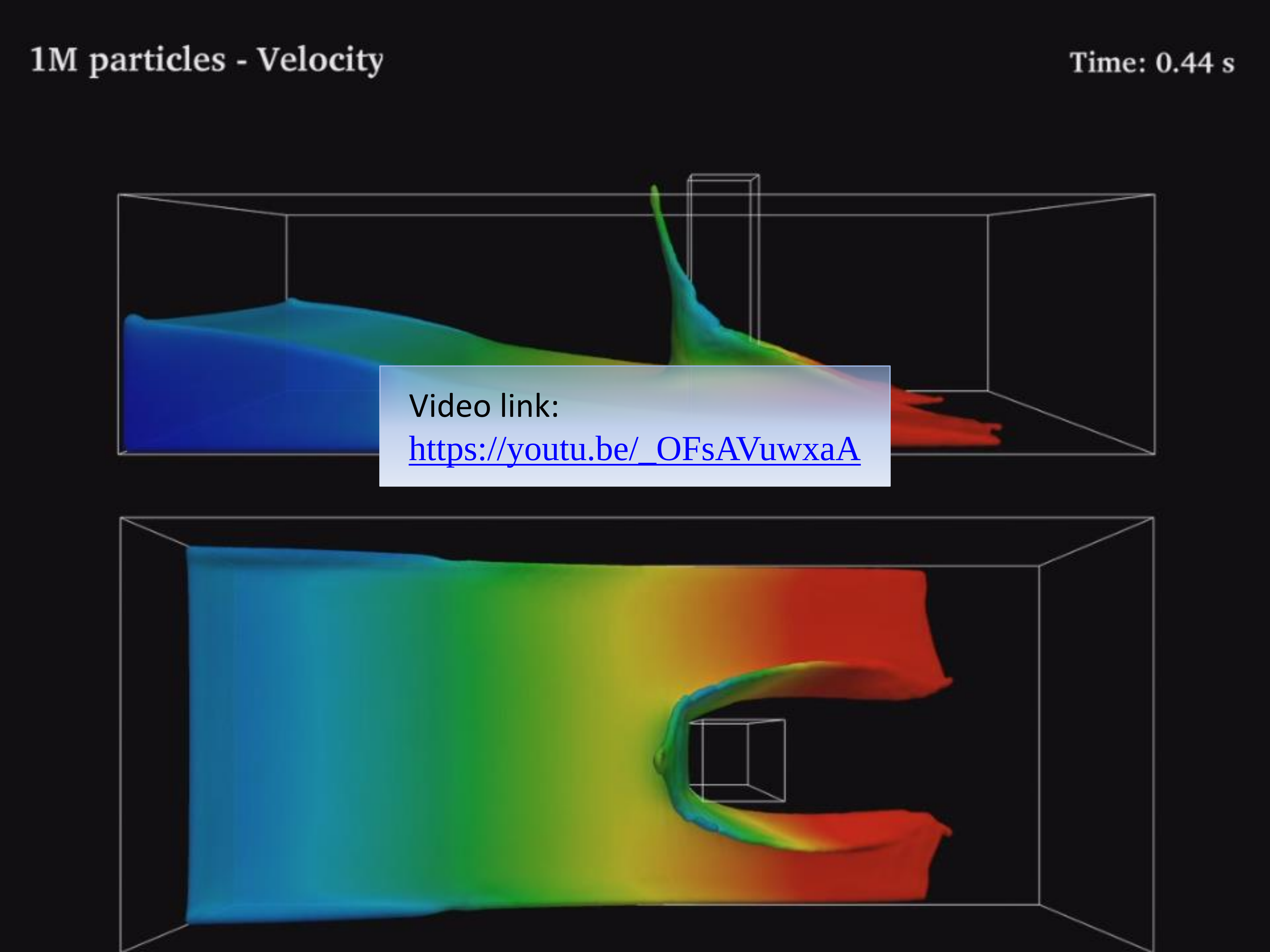
2.2. Neighbour list approaches

Testcase for results

- **Dam break flow impacting on a structure** (experiment of Yeh and Petroff at the University of Washington).
- Physical time of simulation is **1.5 seconds**.
- The number of used particles varies from 4,000 to 150,000.







Video link:
https://youtu.be/_OFsAVuwxaA

2.2. Neighbour list approaches

The best Neighbour List approach is…

	Computational runtime	Memory requirements
CLL (Cell-linked list)	**fast**	**minimum**
VL_X (improved Verlet list)	the fastest (only 6% faster than CLL)	very heavy (30 times more than CLL)
VL_C (classical Verlet list)	the slowest	the most inefficient

DualSPHysics is designed to simulate large number of particles. So that, **Cell-linked list is the best option** to be implemented since it provides the **best balance between the performance and the memory usage**.

Outline

1. Introduction
   - 1.1. Why is SPH too slow?
   - 1.2. High Performance Computing (HPC)
   - 1.3. DualSPHysics project
2. DualSPHysics implementation
   - 2.1. Implementation in three steps
   - 2.2. Neighbour list approaches
3. **CPU acceleration**
4. GPU acceleration
   - 4.1. Parallelization problems in SPH
   - 4.2. GPU optimisations
5. Multi-GPU acceleration
   - 5.1. Dynamic load balancing
   - 5.2. Latest optimisations in Multi-GPU
   - 5.3. Large simulations
6. Future improvements

3. CPU acceleration

Previous ideas:

SPH is a Lagrangian model so particles are moving during simulation.

Each time step NL sorts particles (data arrays) to improve the memory access in PI stage since the access pattern is more regular and efficient.

Another advantage is the ease to identify the particles that belongs to a cell by using a range since the first particle of each cell is known.

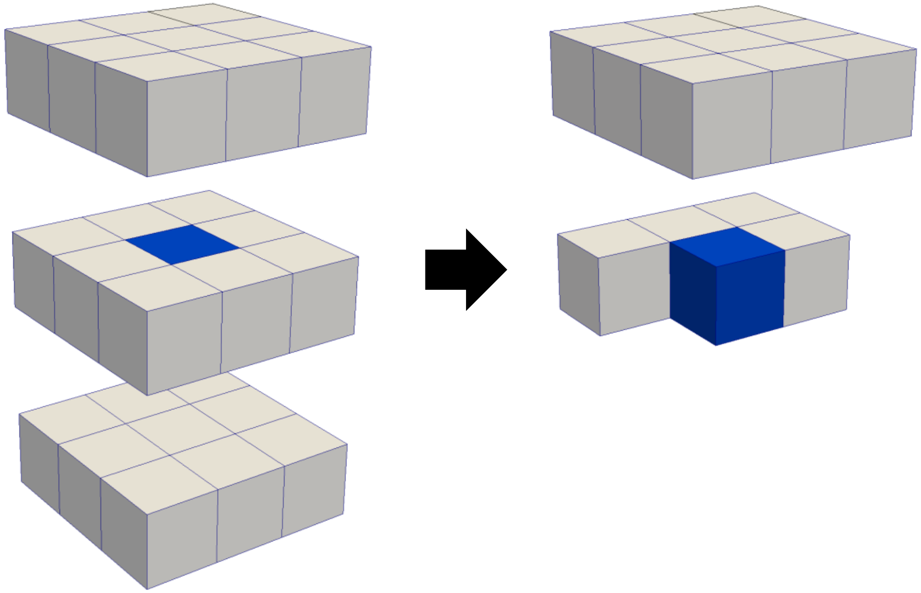
Four optimizations have been applied to DualSPHysics:

- Applying symmetry to particle-particle interaction.
- Splitting the domain into smaller cells.
- Using SSE instructions.
- Multi-core implementation using OpenMP.

3. CPU acceleration

Applying symmetry to particle-particle interaction.

- The force exerted by a particle, *i*, on a neighbour particle, *j*, has the same magnitude but opposite direction when the force is exerted by particle *j* on neighbour *i*.
- The number of interactions to be evaluated can be reduced by two, which decreases the computational time.



In 3-D, each cell only interacts with **13 cells** and partially with itself, **instead of 27 cells**.

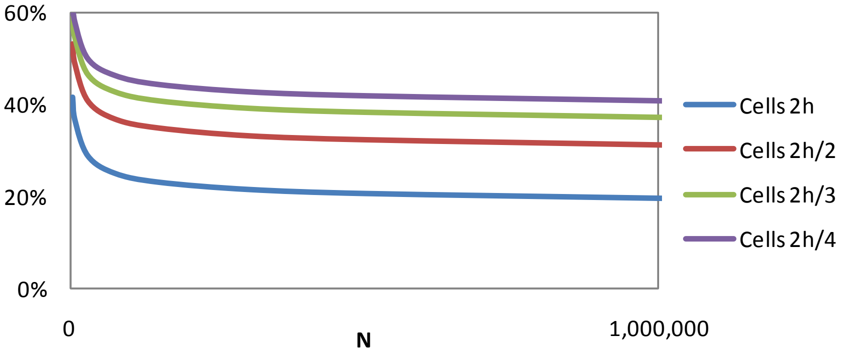
3. CPU acceleration

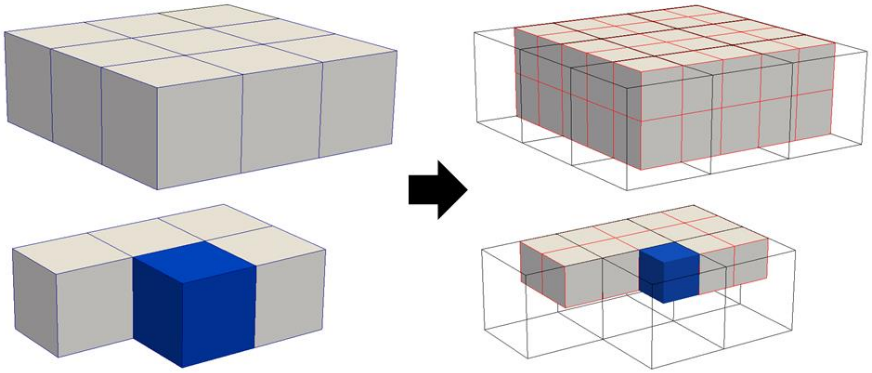
Splitting the domain into smaller cells.

- The domain is split into cells of size (2*h*×2*h*×2*h*) to reduce the neighbour search to only the adjacent cells.
- But using cell size 2*h* in 3-D only 19% of potential neighbours are real neighbours.
- Reducing cell size in half (*h*), the percentage of real neighbours is increased to 31%.

The drawbacks to use *h* instead of 2*h* is:

- The number of cells is multiplied by 8 (in 3-D), increasing memory requirements.
- Each cell has to interact with 63 cells instead of 14 cells although the volume and neighbours is lower.

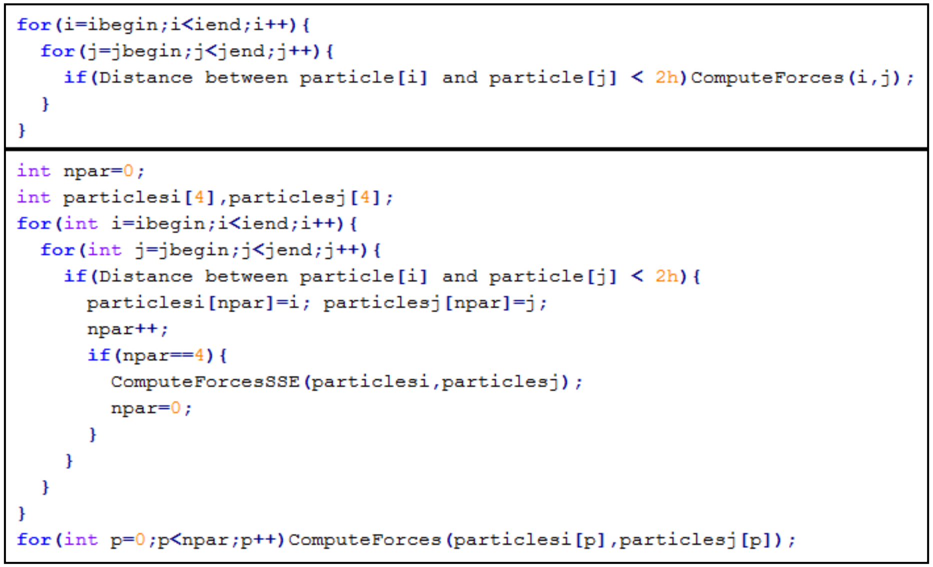




3. CPU acceleration

Using SSE instructions.

- The current CPUs have special instruction sets (SSE, SSE2, SEE3…) of SIMD type (*Single Instruction*, *Multiple Data*) that allow performing operations on data sets.
- An explicit vectorization is applied, grouping particle interactions into packs of 4 interactions, to obtain the best performance on the CPU.
- **Drawbacks:** coding is quite cumbersome and automatic use of these SIMD instructions is not always efficient.



```
for(i=ibegin;i<iend;i++){
  for(j=jbegin;j<jend;j++){
    if(Distance between particle[i] and particle[j] < 2h)ComputeForces(i,j);
  }
}
```

```
int npar=0;
int particlesi[4],particlesj[4];
for(int i=ibegin;i<iend;i++){
  for(int j=jbegin;j<jend;j++){
    if(Distance between particle[i] and particle[j] < 2h){
      particlesi[npar]=i; particlesj[npar]=j;
      npar++;
      if(npar==4){
        ComputeForcesSSE(particlesi,particlesj);
        npar=0;
      }
    }
  }
}
for(int p=0;p<npar;p++)ComputeForces(particlesi[p],particlesj[p]);
```

3. CPU acceleration

Multi-core implementation using OpenMP.

- The current CPUs have several cores or processing units.

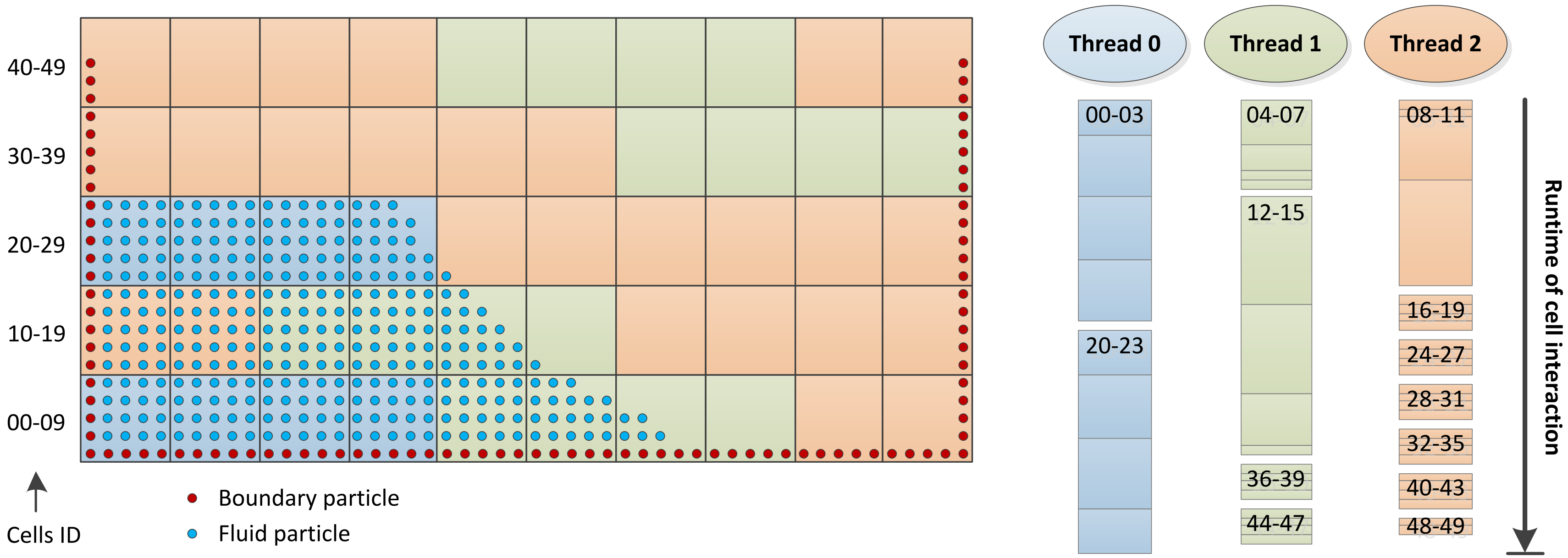
- **OpenMP is the best option** to optimize the performance for systems of shared memory like multi-core CPUs.
  - It can be used to **distribute the computation load** among CPU cores to maximize the performance and to accelerate the SPH code.
  - It **is portable and flexible** whose implementation does not involve major changes in the code.

- The use of OpenMP in particle interaction presents **2 problems**:
  - Several execution threads try to modify the same memory locations simultaneously when symmetry is applied (**race conditions**).
  - **Dynamic load balancing is necessary** since the particles are not distributed evenly.

3. CPU acceleration

Multi-core implementation: Asymmetric approach

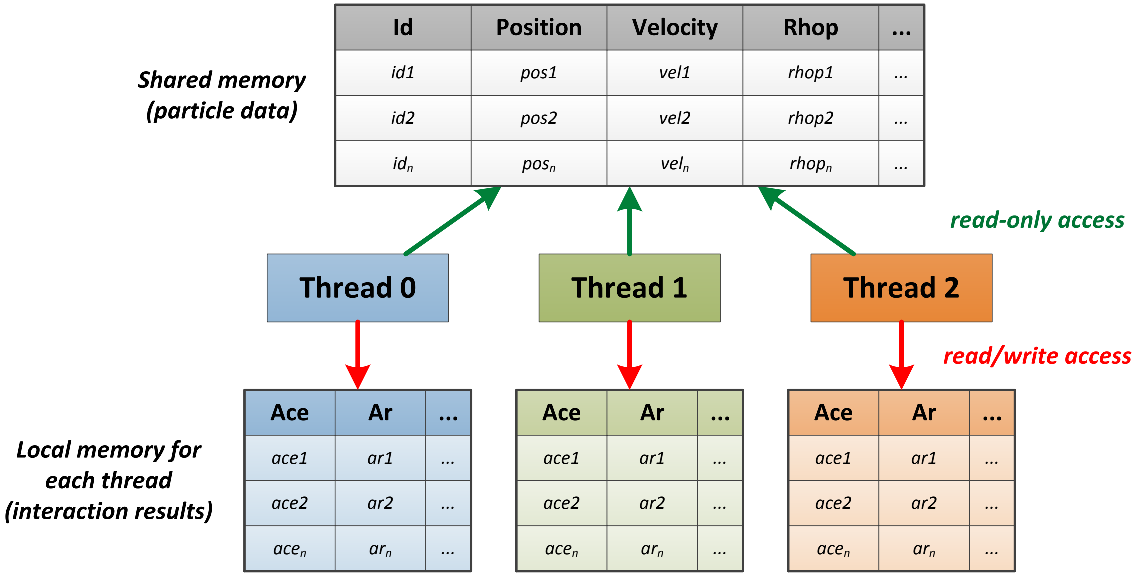
- The **symmetry is not applied** in particle interaction to avoid concurrent access to memory.
- The load balancing is achieved by **using the dynamic scheduler of OpenMP**.
- Particle cells are assigned (in blocks of 10) to the execution threads when they run out of workload.



3. CPU acceleration

Multi-core implementation: Symmetric approach

- The **symmetry is applied** in particle interaction.
- The concurrent memory access is avoided since **each thread has its own memory space** to allocate variables where the forces on each particle are accumulated.
- **Drawback: memory requirement increases** by a factor of 2 when passing from 1 to 8 threads.
- The dynamic scheduler of OpenMP is also employed distributing cells in blocks of 10 among different execution threads (like asymmetric approach).



Shared memory (particle data)

Id	Position	Velocity	Rhop	...
id1	*pos1*	*vel1*	*rhop1*	...
id2	*pos2*	*vel2*	*rhop2*	...
id$_n$	*pos*$_n$	*vel*$_n$	*rhop*$_n$	...

Local memory for each thread (interaction results)

Ace	Ar	...
ace1	*ar1*	...
ace2	*ar2*	...
ace$_n$	*ar*$_n$	...

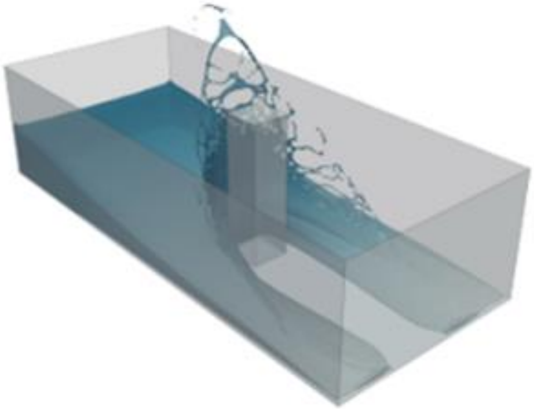
Ace	Ar	...
ace1	*ar1*	...
ace2	*ar2*	...
ace$_n$	*ar*$_n$	...

Ace	Ar	...
ace1	*ar1*	...
ace2	*ar2*	...
ace$_n$	*ar*$_n$	...

3. CPU acceleration

Testcase for results

- **Dam break** flow impacting on a structure.
- Simulating **1.5 seconds** of physical time.

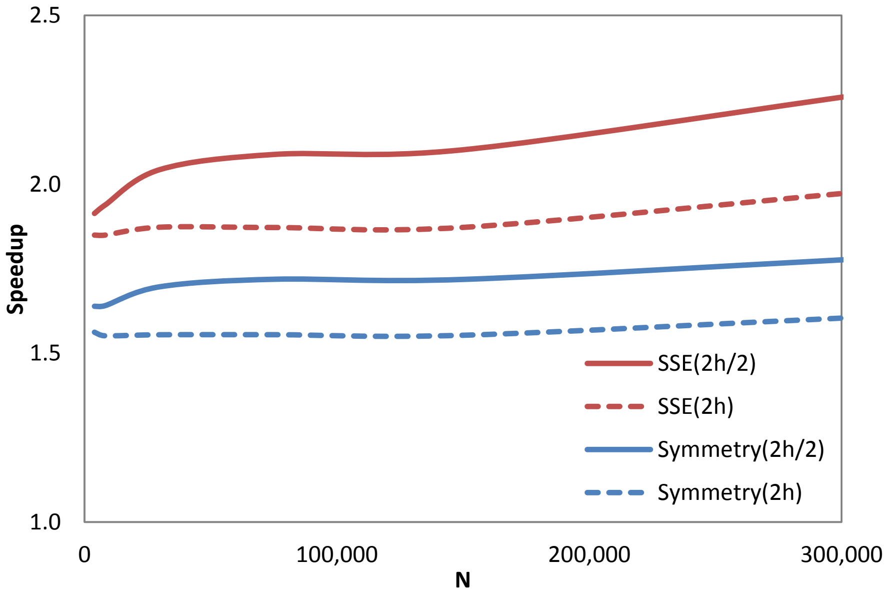


Hardware and configuration for results

- **Hardware:** Intel® Core ™ i7 940 at 2.93 GHz (4 physical cores, 8 logical cores with Hyper-threading), with 6 GB of DDR3 RAM memory at 1333 MHz.
- **Operating system:** Ubuntu 10.10 64-bit.
- **Compiler:** GCC 4.4.5 (compiling with the option –O3).

3. CPU acceleration

Speedup for different number of particles (N) when **applying symmetry**, the use of **SSE instructions**. Two different **cell sizes (*2h* and *2h/2*)** were considered.

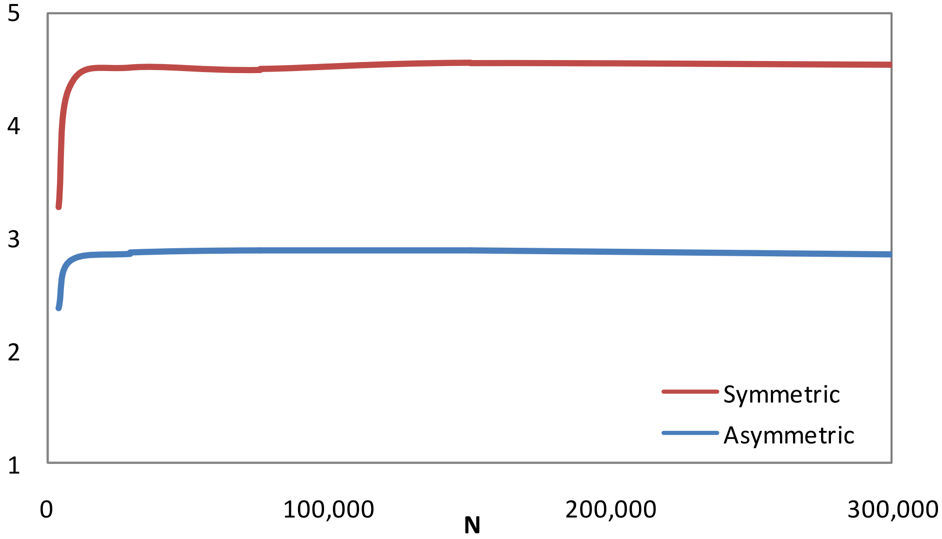


Using **300,000 particles**, the **maximum speedup** was **2.3x** using Symmetry, SSE and cell size *2h/2.*

Speedup was obtained when compared to the version without optimizations.

3. CPU acceleration

Speedup for different number of particles (N) **with different OpenMP implementations** (using 8 logical threads) in comparison with the most efficient single-core version (symmetry, SSE and cell size *2h/2*).



Symmetric approach is the **most efficient** (speedup 4.5x using 8 threads).

Speedup was obtained when compared to the most efficient single-core version.

Outline

1. Introduction
    1.1. Why is SPH too slow?
    1.2. High Performance Computing (HPC)
    1.3. DualSPHysics project

2. DualSPHysics implementation
    2.1. Implementation in three steps
    2.2. Neighbour list approaches

3. CPU acceleration

4. **GPU acceleration**
    4.1. Parallelization problems in SPH
    4.2. GPU optimisations

5. Multi-GPU acceleration
    5.1. Dynamic load balancing
    5.2. Latest optimisations in Multi-GPU
    5.3. Large simulations

6. Future improvements

4. GPU acceleration

GPU implementation

DualSPHysics was implemented using the CUDA programming language to run SPH method on Nvidia GPUs.

Important: An efficient and full use of the capabilities of the GPUs is not straightforward. It is necessary to know and to take into account the details of the GPU architecture and the CUDA programming model.

Differences regarding the CPU implementation:

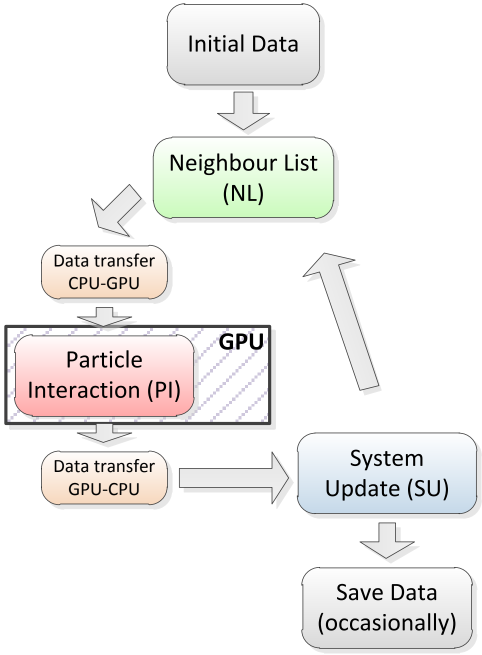
- Each GPU thread calculates the interaction between a target particle and its neighbours.
- The symmetry is not used in particle interaction because it cannot be applied efficiently on GPU.

Two initial GPU implementation were tested: Partial and full GPU implementation.

4. GPU acceleration

Partial GPU implementation

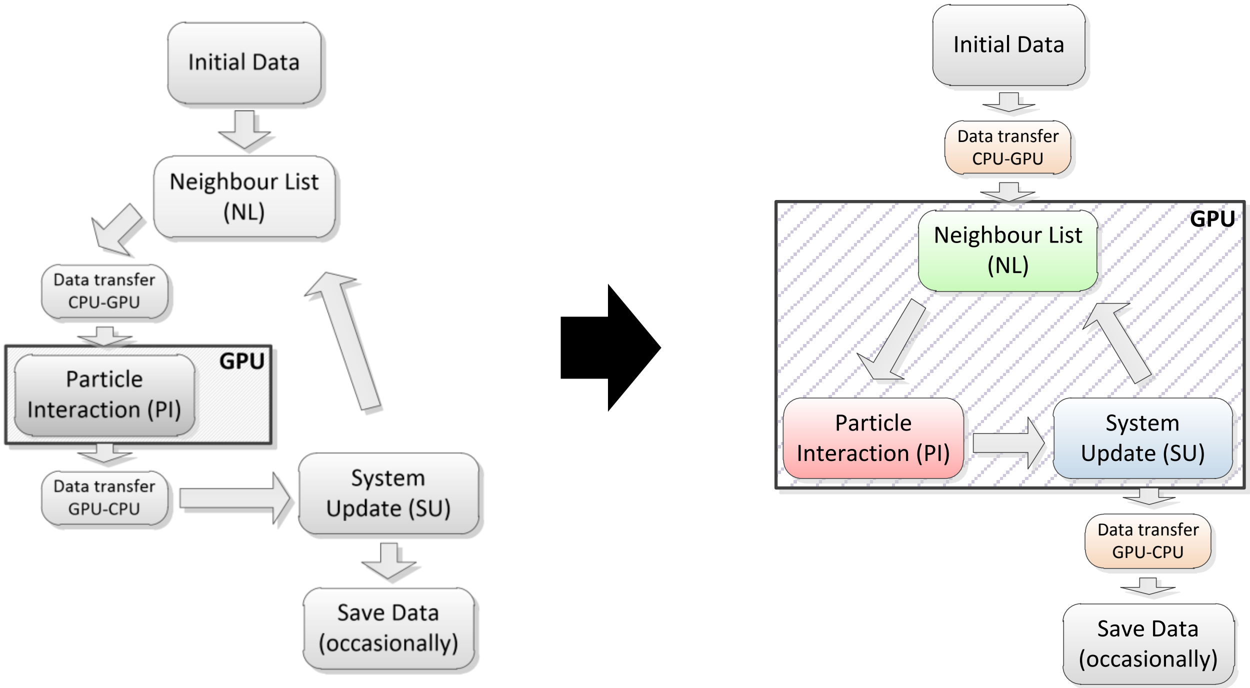
- **GPU is used only in particle interaction** since this part consumes over 90% of the execution time.
- **Drawback:** Particle data and neighbour list information must be transferred from CPU to GPU and the interaction results from GPU to CPU each time step.



4. GPU acceleration

Full GPU implementation

- **GPU is used in all steps** (Neighbour List, Particle Interaction and System Update).
- This approach is the most efficient since:
  - All particle data is kept in GPU memory and the **transfers CPU-GPU are removed**.
  - **Neighbour List and System Update are parallelized**, obtaining a speedup also in this part of the code.



4.1. Parallelization problems in SPH

Problems in Particle Interaction step

These problems appears since each thread has to interact with different neighbours.

- **Code divergence:**

  GPU threads are grouped into sets of 32 (*warps*) which execute the same operation simultaneously. When there are different operations in one warp these operations are executed sequentially, giving rise to a significant loss of efficiency.

- **No coalescent memory accesses:**

  The global memory of the GPU is accessed in blocks of 32, 64 or 128 bytes, so the number of accesses to satisfy a warp depends on how grouped data are. In SPH a regular memory access is not possible because the particles are moved each time step.

- **No balanced workload:**

  Warps are executed in *blocks* of threads. To execute a block some resources are assigned and they will not be available for other blocks till the end of the execution. In SPH the number of interactions is different for each particle so one thread can be under execution, keeping the assigned resources, while the rest of threads have finished.

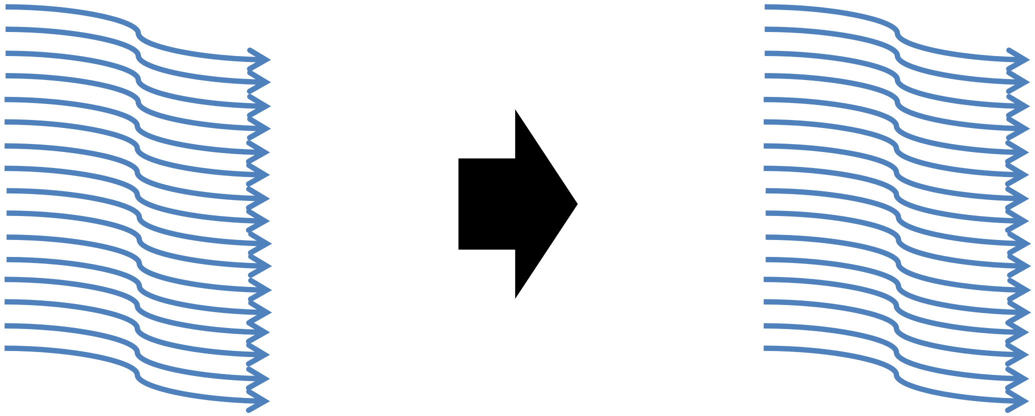
4.1. Parallelization problems in SPH

Problems in Particle Interaction step

- **Code divergence:**

  GPU threads are grouped into sets of 32 (*warps*) which execute the same operation simultaneously. When there are different operations in one warp these operations are executed sequentially, giving rise to a significant loss of efficiency.

NO DIVERGENT WARPS



32 threads executing the same task over 32 values

32 threads executed simultaneously

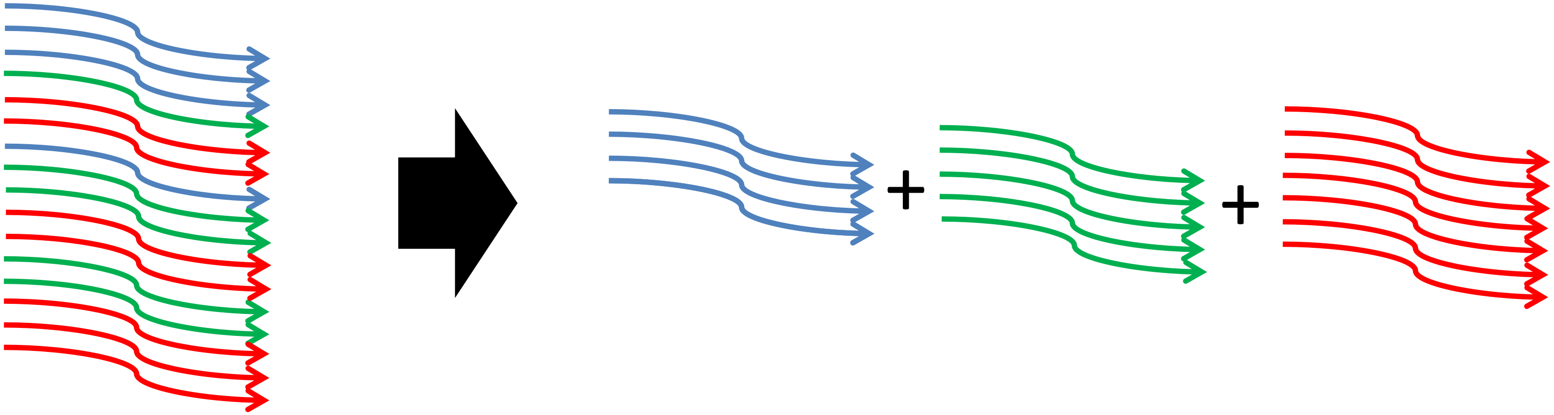
4.1. Parallelization problems in SPH

Problems in Particle Interaction step

- **Code divergence:**

  GPU threads are grouped into sets of 32 (*warps*) which execute the same operation simultaneously. When there are different operations in one warp these operations are executed sequentially, giving rise to a significant loss of efficiency.

DIVERGENT WARPS !!!



32 threads executing
three different tasks (IF)
over 16 values

execution of the 32 threads
will take the runtime needed to carry out the
three tasks **sequentially**

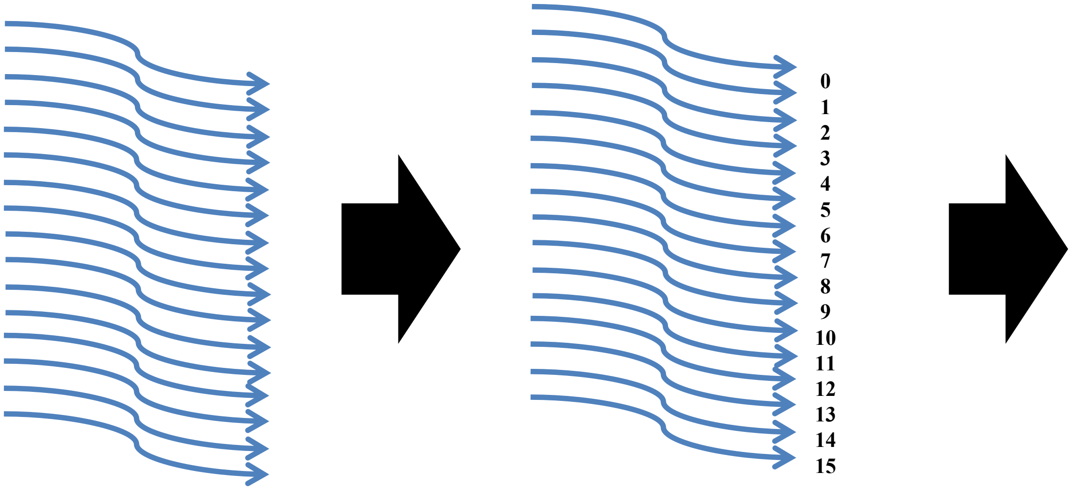
4.1. Parallelization problems in SPH

Problems in Particle Interaction step

- **No coalescent memory accesses:**

  The global memory of the GPU is accessed in blocks of 32, 64 or 128 bytes, so the number of accesses to satisfy a warp depends on how grouped data are. In SPH a regular memory access is not possible because the particles are moved each time step.

COALESCED ACCESS



Only 1 access to memory is required

16 threads executing over 16 values

16 values stored in 16 **consecutive** memory positions

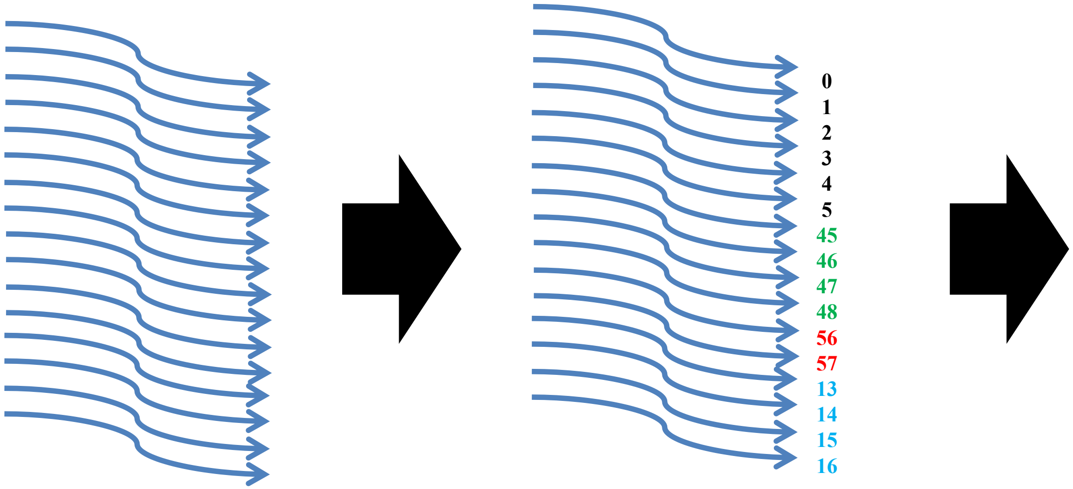
4.1. Parallelization problems in SPH

Problems in Particle Interaction step

- **No coalescent memory accesses:**

  The global memory of the GPU is accessed in blocks of 32, 64 or 128 bytes, so the number of accesses to satisfy a warp depends on how grouped data are. In SPH a regular memory access is not possible because the particles are moved each time step.

NON COALESCED ACCESS



4 memory accesses are required

16 threads executing over 16 values

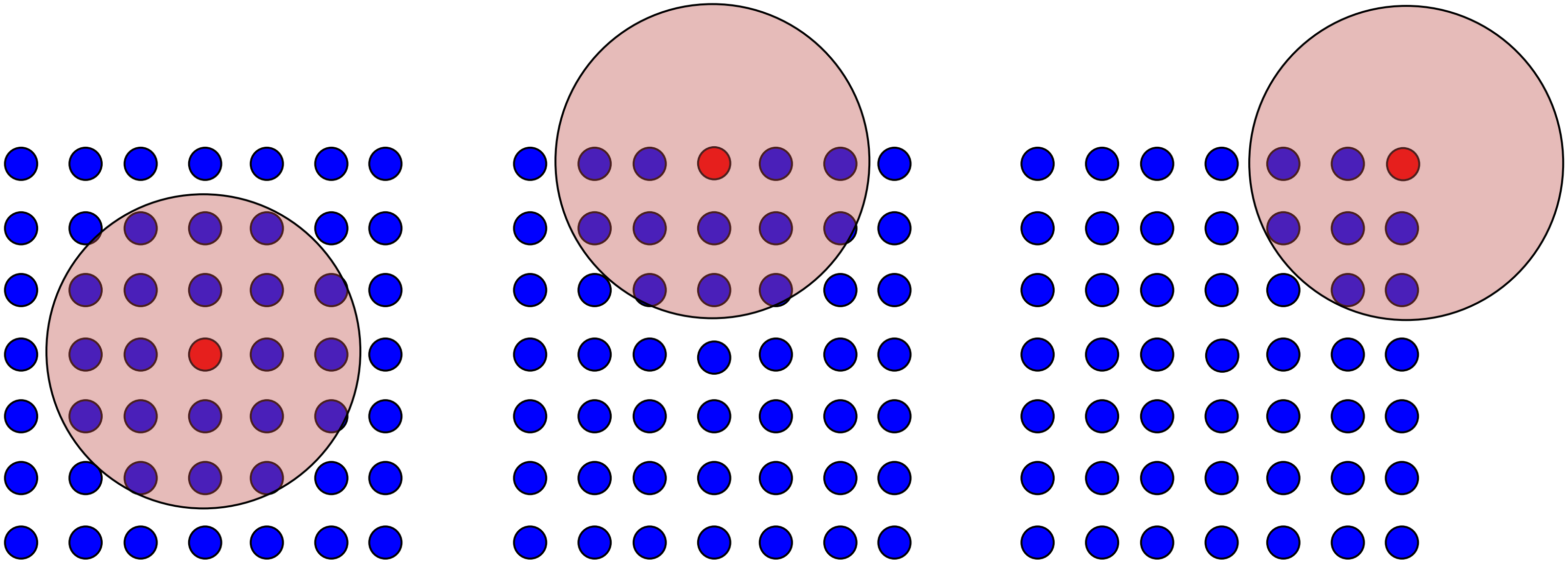
16 values stored in **no consecutive** memory positions

4.1. Parallelization problems in SPH

Problems in Particle Interaction step

- **No balanced workload:**

  Warps are executed in *blocks* of threads. To execute a block some resources are assigned and they will not be available for other blocks till the end of the execution. In SPH the number of interactions is different for each particle so one thread can be under execution, keeping the assigned resources, while the rest of threads have finished.



20 neighbours **12 neighbours** **7 neighbours**

4.2. GPU optimisations

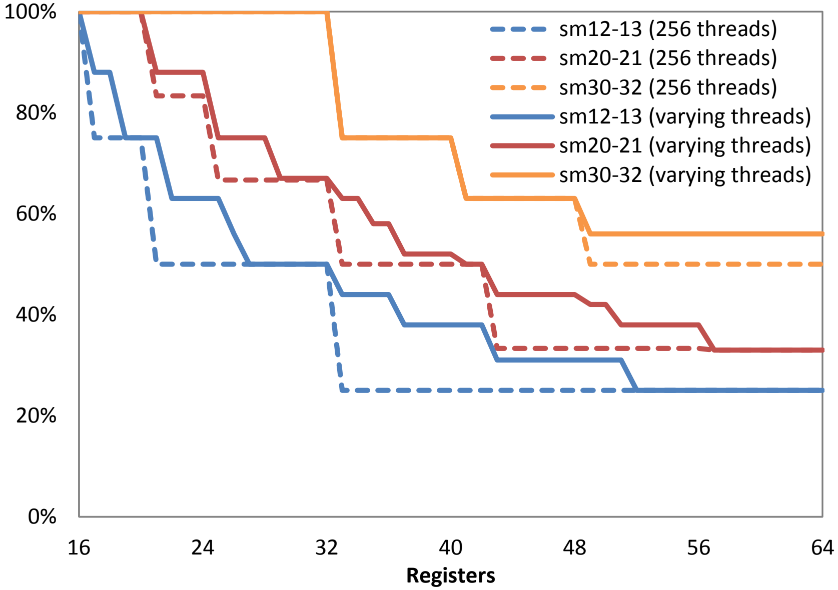
Five optimizations have been applied to DualSPHysics to avoid or minimize the problems previously described.

- Maximizing the occupancy of GPU.
- Reducing global memory accesses.
- Simplifying the neighbour search.
- Adding a more specific CUDA function of interaction.
- Division of the domain into smaller cells.

4.2. GPU optimisations

Maximizing the occupancy of GPU

- Occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU or Streaming Multiprocessor (SM).
- It is essential to have the largest number of active warps in order to hide the latencies of memory access since the access to the GPU global memory is irregular.



For example, using a GPU with compute capability 1.3 (sm13) for 35 registers.

The occupancy is

25% using 256 threads per block

but can be

44% using 448 threads.

CUDA 6.5 includes several runtime functions to help in occupancy calculations and launch configuration

4.2. GPU optimisations

Reducing global memory accesses

- The number of memory accesses in the interaction kernel can be reduced by
  - grouping some arrays used in particle interaction (*pos+press* and *vel+rhop* are combined to create two arrays of 16 bytes each one).
  - avoid reading values that can be calculated from other variables (*csound* and *tensil* are calculated from *press*).
- The number of accesses to the global memory of the GPU is reduced from 6 to 2 and the volume of data to be read from 40 to 32 bytes.

Variable	Size (bytes)	Description
pos	3 x 4	Position in X,Y and Z
vel	3 x 4	Velocity in X,Y and Z
rhop	4	Density
csound	4	Speed of sound
prrhop	4	Ratio between pressure and density
tensil	4	Tensile correction following

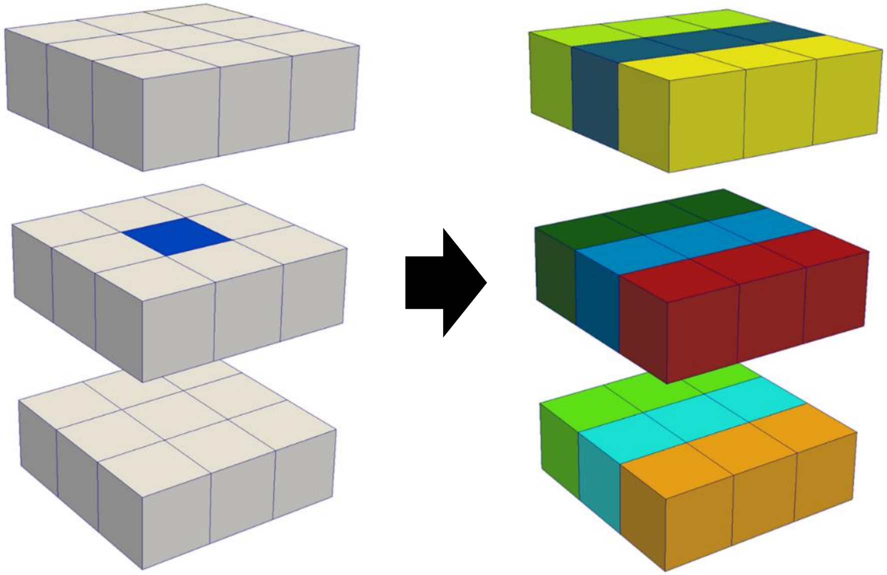
Variable	Size (bytes)	Description
pos+press	4 x 4	Position + Pressure
vel+rhop	4 x 4	Velocity + Density
csound	0	Calculated from press
prrhop	0	Calculated from press
tensil	0	Calculated from press

4.2. GPU optimisations

Simplifying the neighbour search

- Each particle has to interact with particles in surrounding cells (27 cells).
- These 27 cells can be defined as 9 ranges of particles since particles in adjacent cells are in consecutive memory positions.
- The neighbour search can be optimised using these ranges instead of cells.
- **Drawback:** Extra 144 bytes needed per cell.

Each particle interacts with **27 cells**



Each particle interacts with **9 ranges**

4.2. GPU optimisations

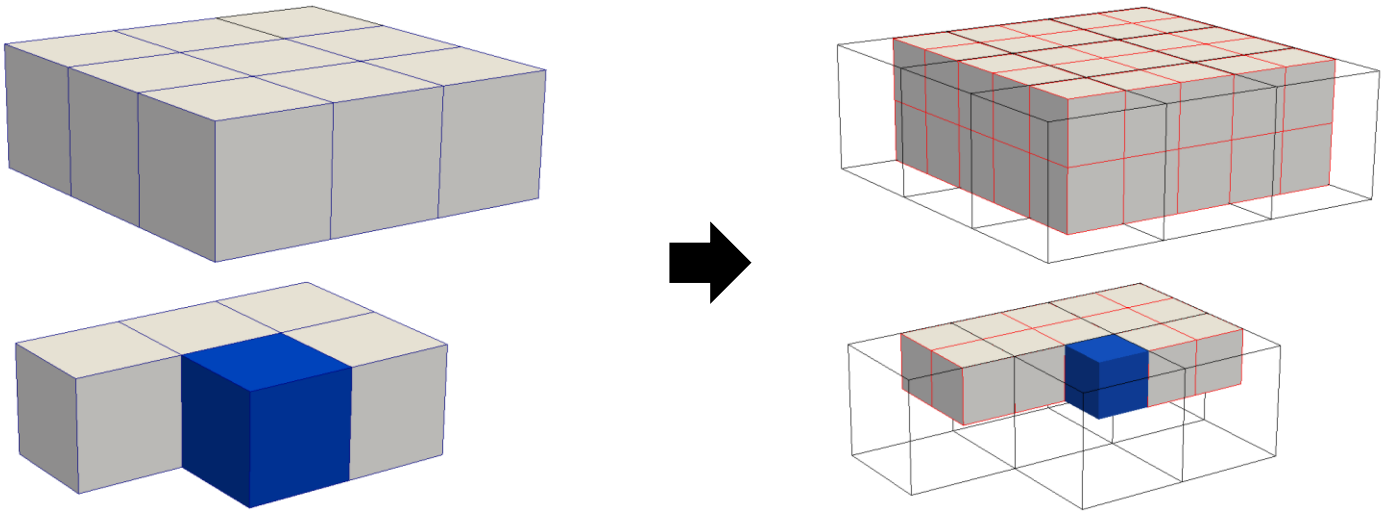
Adding a more specific CUDA function of interaction

- Initially, the same CUDA kernel was used to calculate all interaction forces boundary-fluid (B-F), fluid-boundary (F-B) and fluid-fluid (F-F).
- Is more efficient use a specific kernel for the B-F interaction because this interaction is simpler and it can be optimised.
- To minimise the access to the global memory of the GPU the interaction F-F and F-B can be merged in one kernel execution. Thus the particle data and result data of each thread is loaded and saved once instead twice.

4.2. GPU optimisations

Division of the domain into smaller cells

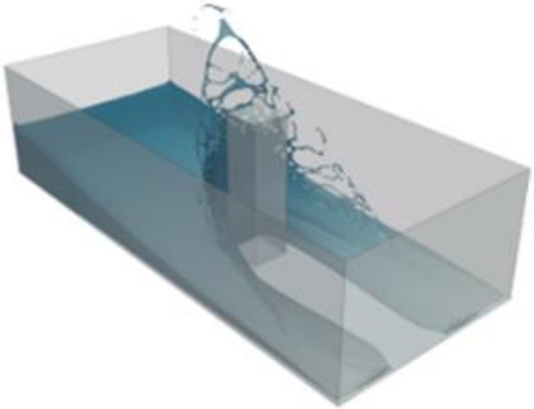
- It is the same optimization used in CPU implementation.
- Reducing cell size in half (*h*), the percentage of real neighbours is increased to 31%.
- **Drawback:** The memory requirements increases because the number of cells is 8 times higher and the number of ranges of particles to be evaluated in the neighbour search increases from 9 to 25 (using 400 bytes per cell).



4.2. GPU optimisations

Testcase for results

- **Dam break** flow impacting on a structure.
- Simulating **1.5 seconds** of physical time.

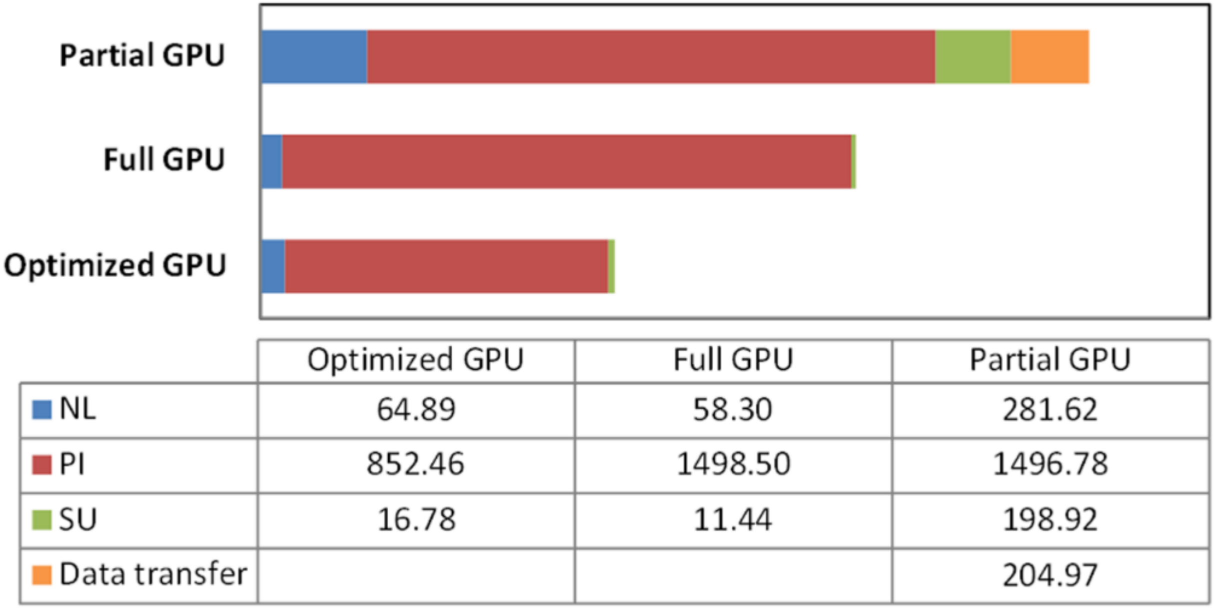


Hardware for results

	Number of cores	Processor clock	Memory space	Compute capability
Intel Xeon X5500	1-8	2.67 GHz		
Tesla 1060	240	1.30 GHz	4 GB	1.3
GTX 480	480	1.40 GHz	1.5 GB	2.0
GTX 680	1536	1.14 GHz	2 GB	3.0
Tesla K20	2496	0.71 GHz	5 GB	3.5
GTX Titan	2688	0.88 GHz	6 GB	3.5

4.2. GPU optimisations

Computational runtimes (in seconds) using GTX 480 for different GPU implementations (partial, full and optimized) when simulating 500,000 particles.



	Optimized GPU	Full GPU	Partial GPU
🟦 NL	64.89	58.30	281.62
🟥 PI	852.46	1498.50	1496.78
🟩 SU	16.78	11.44	198.92
🟧 Data transfer			204.97

Full GPU is **1.26x** faster than Partial GPU.

Optimized GPU is **2.12x** faster than Partial GPU.

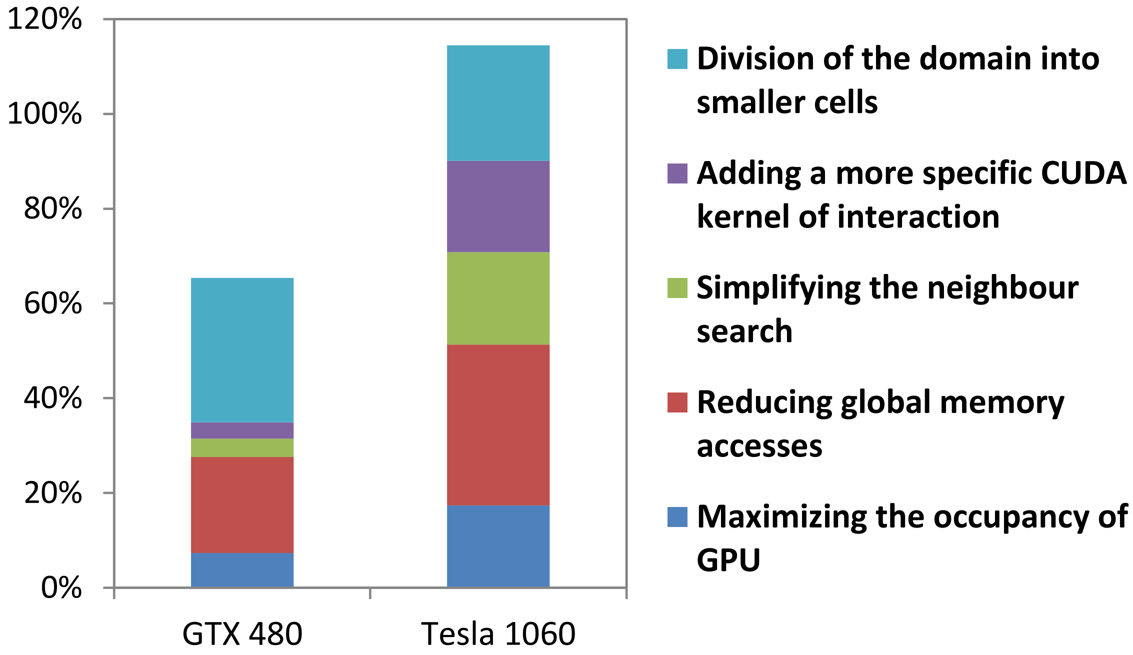
4.2. GPU optimisations

Improvement achieved on GPU simulating 1 million particles when **applying** the different **GPU optimisations** using GTX 480 and Tesla 1060.

Speedup of fully optimized GPU code over GPU code without optimizations is:

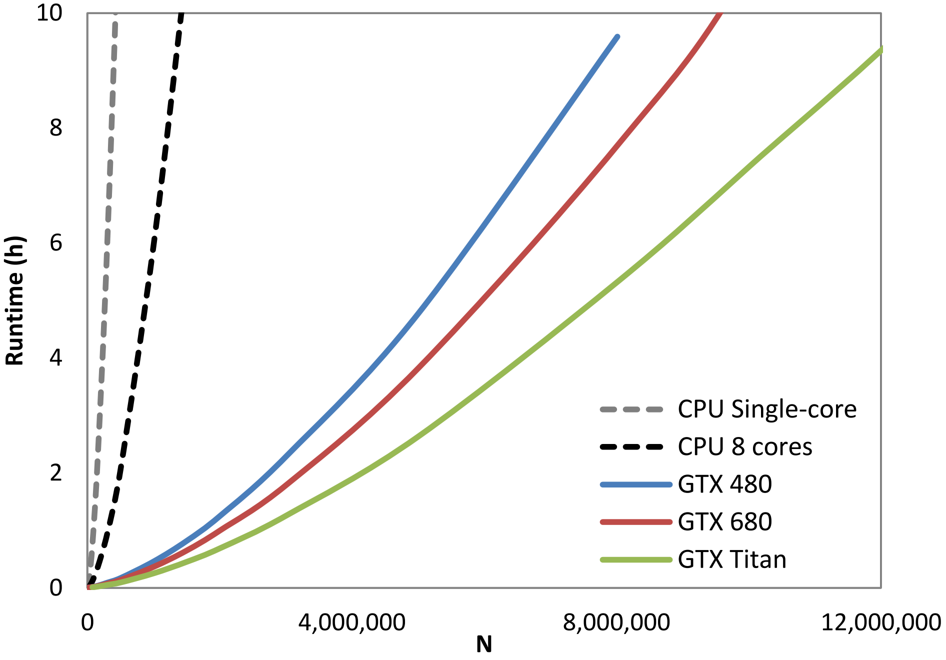
1.65x for **GTX 480**

2.15x for **Tesla 1060**

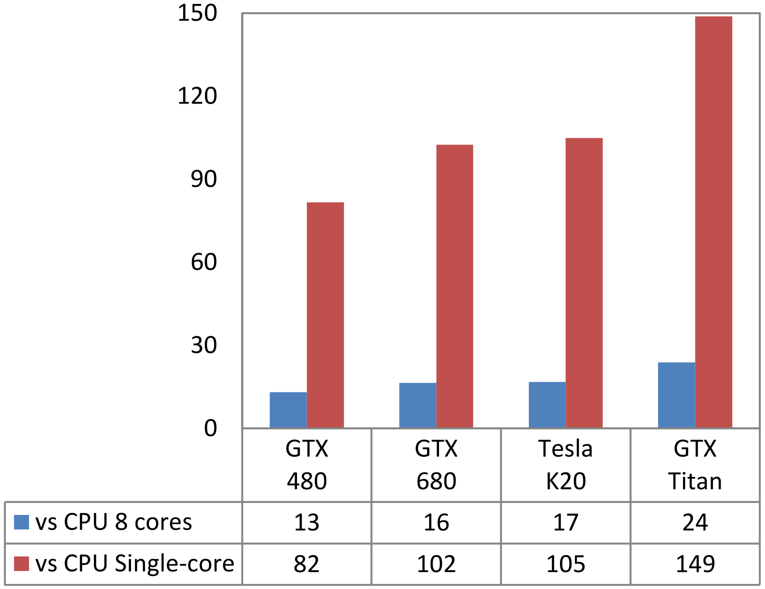


4.2. GPU optimisations

Runtime for CPU and different GPU cards.



Speedups of GPU against CPU simulating 1 million particles.



	GTX 480	GTX 680	Tesla K20	GTX Titan
🟦 vs CPU 8 cores	13	16	17	24
🟥 vs CPU Single-core	82	102	105	149

After optimising the performance of DualSPHysics on CPU and GPU...

The most powerful GPU (**GTX Titan**) is **149 times faster** than CPU (single core execution) and **24 times faster** than the CPU using all 8 cores.

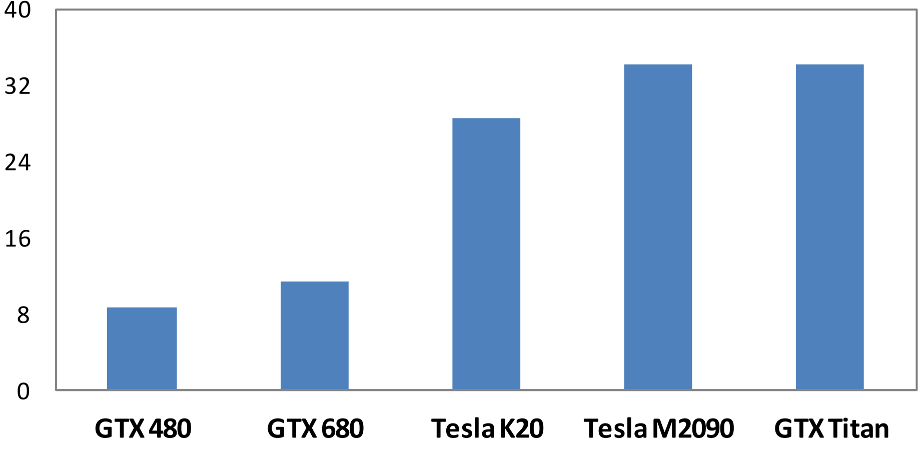
4.2. GPU optimisations

The simulation of **real cases implies huge domains with a high resolution**, which implies simulating tens or hundreds of million particles.

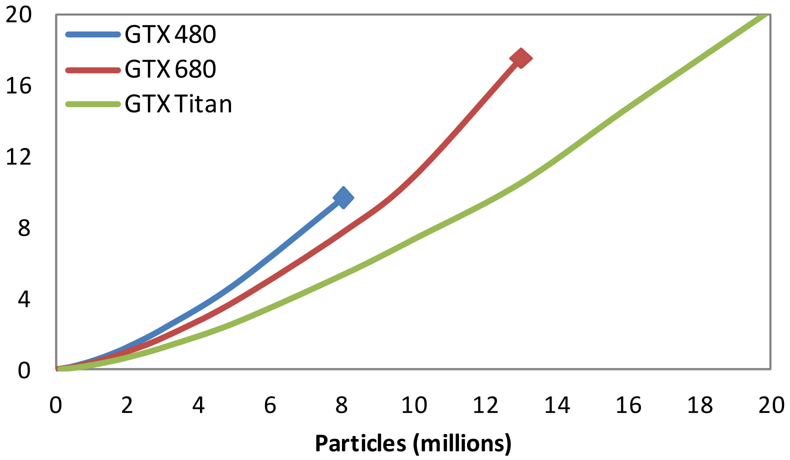
The use of one GPU presents important **limitations**:

- Maximum number of particles depends on the memory size of GPU.
- Time of execution increases rapidly with the number of particles.

Maximum number of particles (millions)



Runtime (hours)



Outline

1. Introduction
   1.1. Why is SPH too slow?
   1.2. High Performance Computing (HPC)
   1.3. DualSPHysics project

2. DualSPHysics implementation
   2.1. Implementation in three steps
   2.2. Neighbour list approaches

3. CPU acceleration

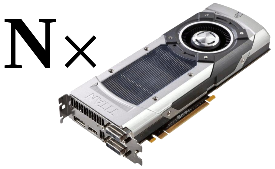
4. GPU acceleration
   4.1. Parallelization problems in SPH
   4.2. GPU optimisations

5. **Multi-GPU acceleration**
   5.1. Dynamic load balancing
   5.2. Latest optimisations in Multi-GPU
   5.3. Large simulations

6. Future improvements

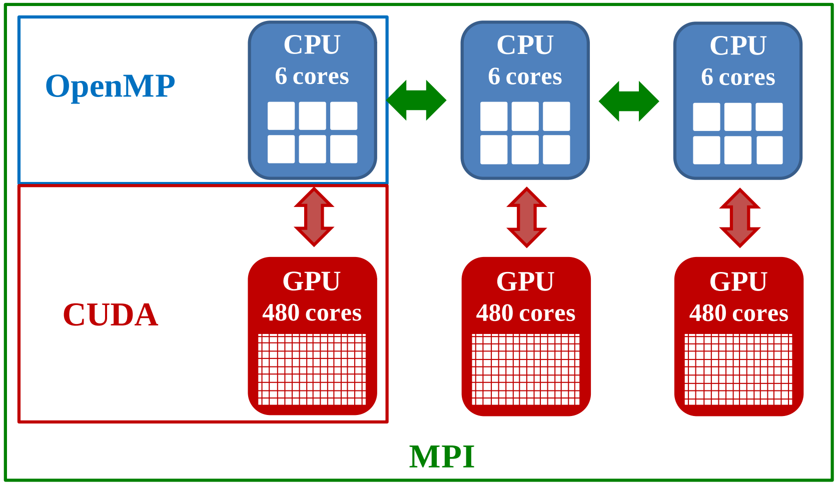
5. Multi-GPU implementation

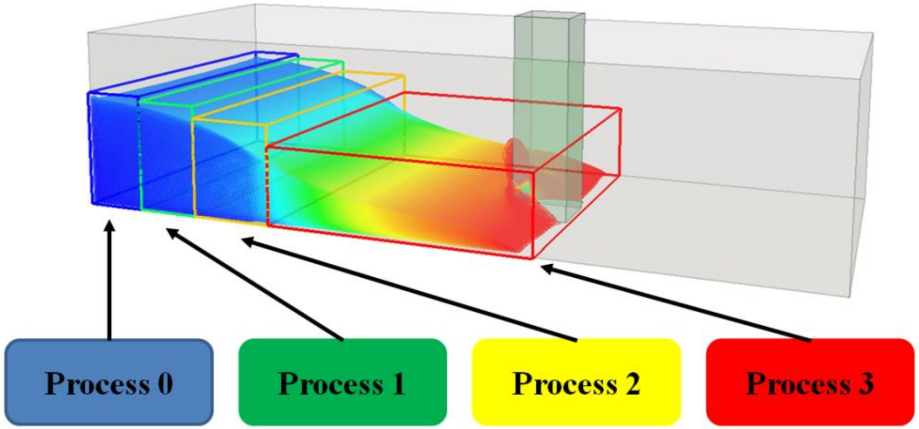
N×



MPI is used to combine resources of multiple machines connected via network.

The **physical domain** of the simulation **is divided among the different MPI processes**. Each process only needs to assign resources to manage a subset of the total amount of particles for each subdomain.





5. Multi-GPU implementation

N×

The use of MPI implies an **overcost**:

- **Communication**: Time dedicated to the interchange of data between processes.
- **Synchronization**: All processes must wait for the slowest one.

Solutions:

- **Overlapping** between force computation and communications: while data is transferred between processes, each process can compute the force interactions among its own particles. In the case of GPU, the CPU-GPU transfers can also be overlapped with computation using *streams* and *pinned memory*.
- **Load balancing**. A dynamic load balancing is applied to minimise the difference between the execution times of each process.

5.1. Dynamic load balancing

N×

Due to the nature Lagrangian of the SPH method, is necessary to balance the load throughout the simulation.

FIRST approach according to the **number of fluid particles**

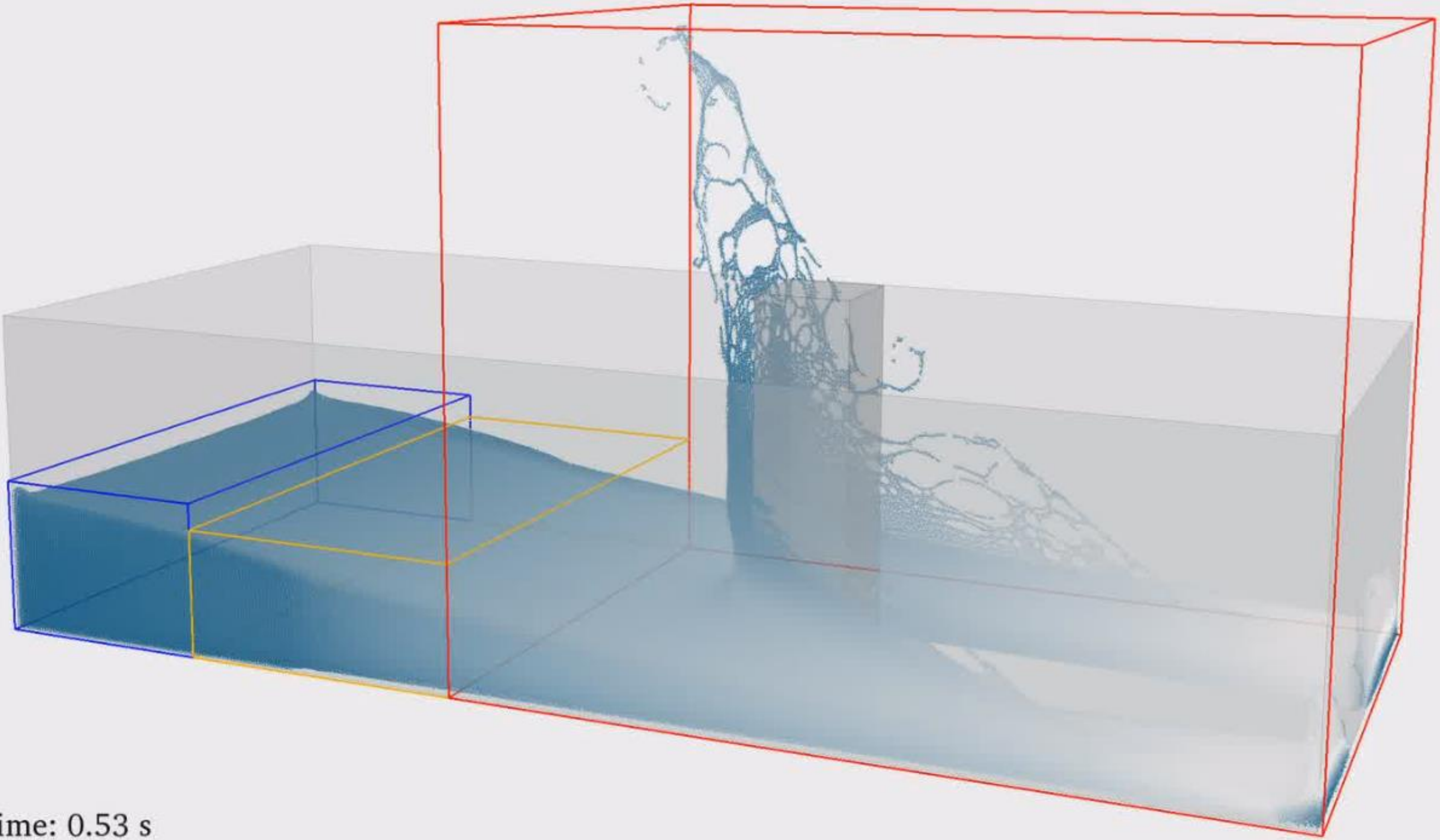
> The number of particles must be redistributed after some time steps to get the workload balanced among the processes and minimise the synchronisation time.

SECOND approach according to the **required computation time** of each device

> Enables the adaptation of the code to the features of a heterogeneous cluster achieving a better performance.

GPUs: 3 x GTX480
MPI: Dynamic Balancing-Np
Particles: 6 Millions
Steps: 42,624
Runtime: 2.6 hours

DualSPHysics

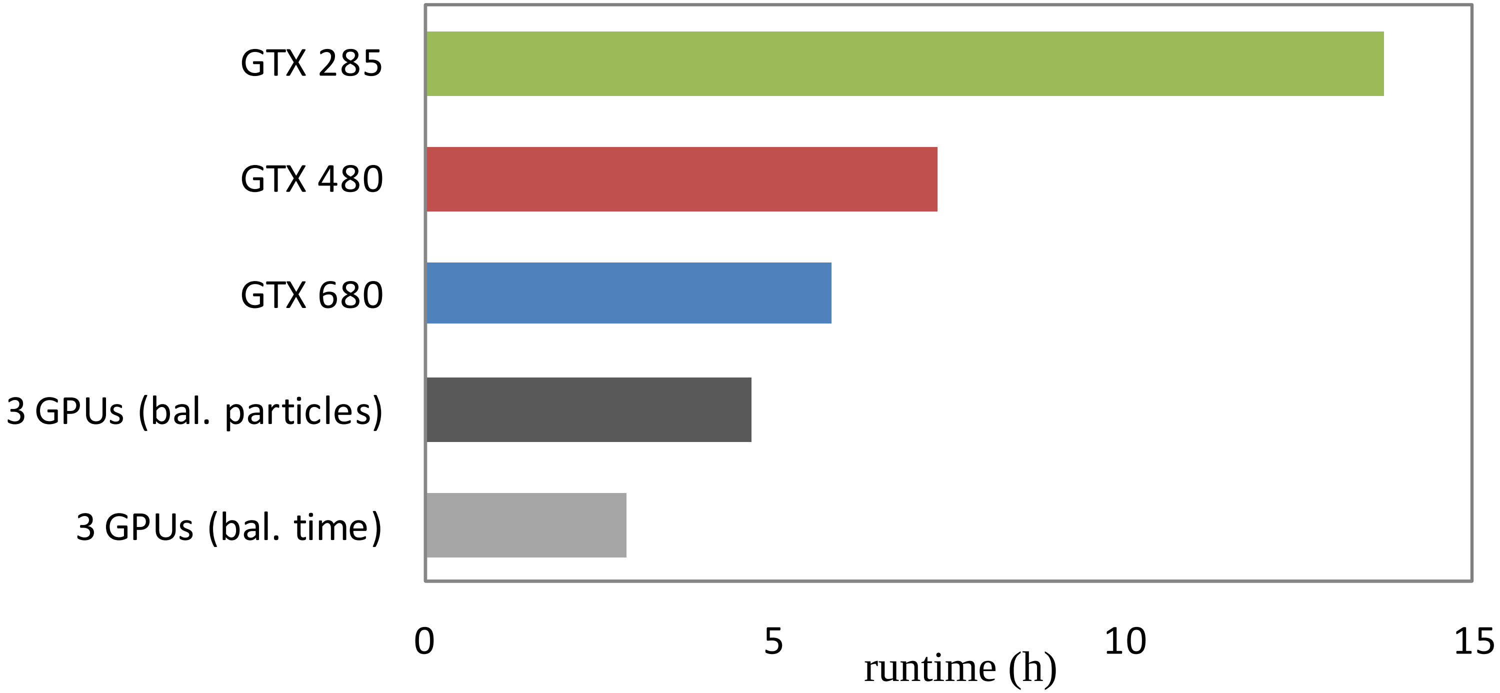


Time: 0.53 s

5.1. Dynamic load balancing

N×

Results using one GPU and several GPUs with dynamic load balancing



Hardware: GTX 680 GTX 480 GTX 285

5.1. Dynamic load balancing

N×

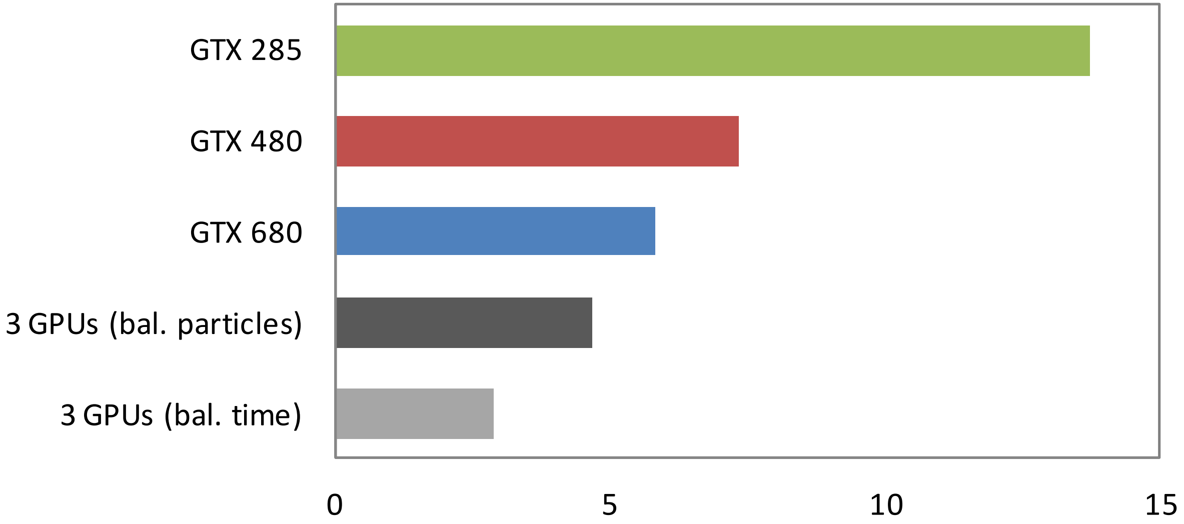
Results using one GPU and several GPUs with dynamic load balancing

- Using the fastest GPU (GTX 680) **5.8 hours**
- Using three different GPUs

  According to the number of fluid particles **4.6 hours**

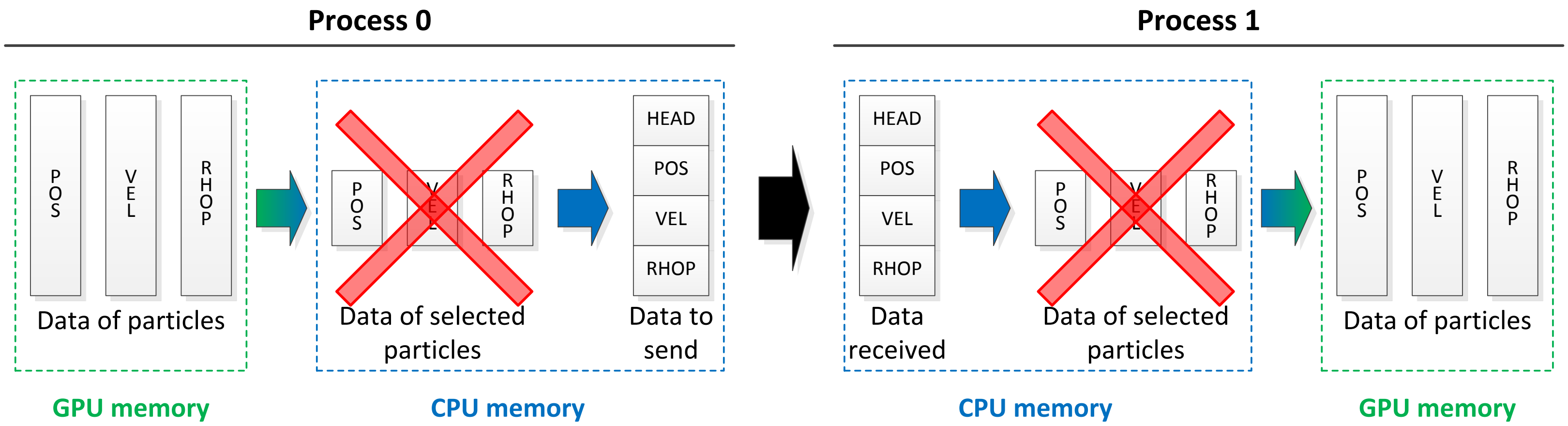
  According to the required computation time **2.8 hours**

The second approach is 1.7x faster than first approach and 2.1x faster than one GPU.



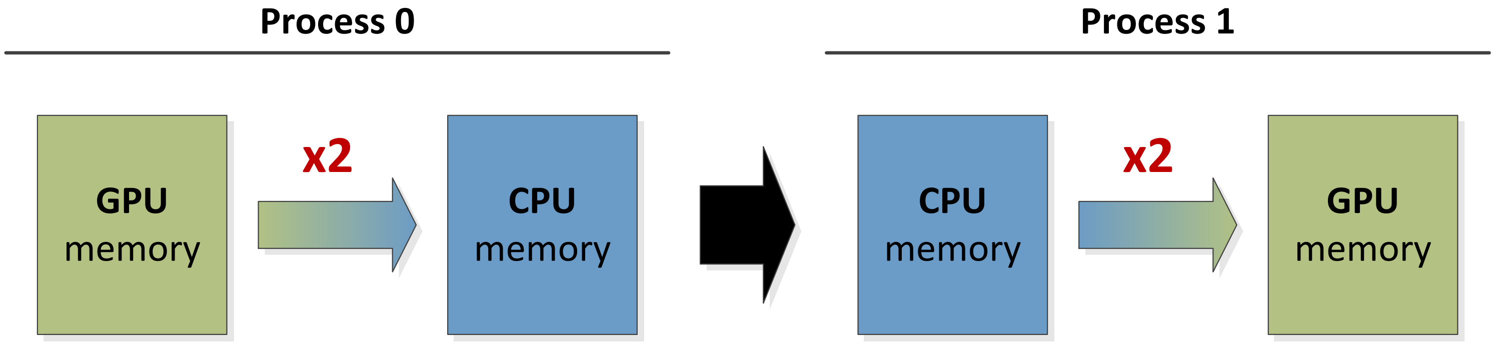
5.2. Latest optimisations in Multi-GPU

Removing buffers during MPI communication:
In the previous version, to send data from CPU to GPU, data were initially transferred in variables *pos*, *vel*, *rhop* and then were copied in a buffer and this buffer was sent with MPI. To receive from GPU to CPU, data are received grouped in a buffer, then copied to variables on CPU (*pos*, *vel*, *rhop*) and these variables are transferred to GPU.
Now, instead of copying GPU data into CPU variables, data is directly copied in the buffer that will be sent with MPI. When receiving data, all are grouped in a buffer and they are copied from the buffer to the GPU variables. Thus, data are not copied in variables *pos*, *vel*, *rhop* of CPU.



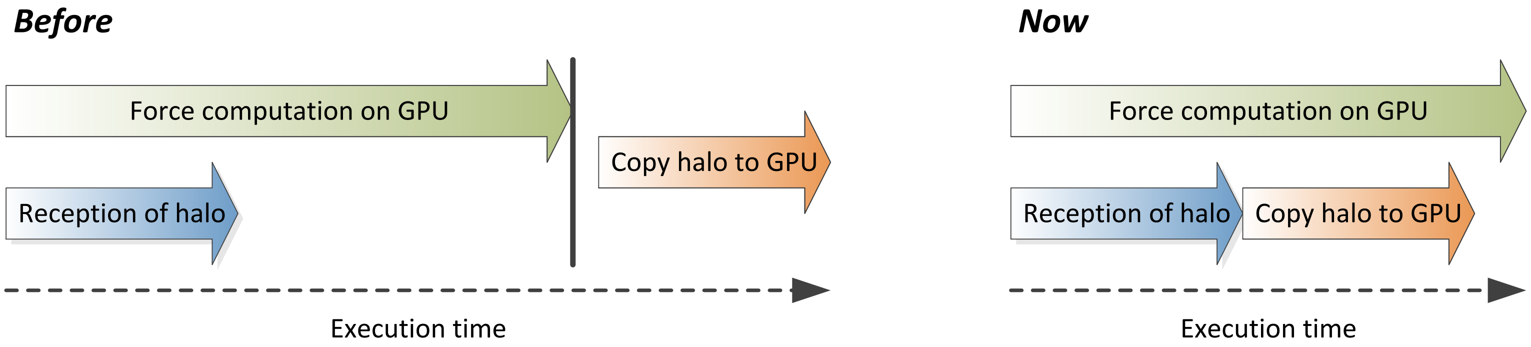
5.2. Latest optimisations in Multi-GPU

Use of pinned memory for faster CPU-GPU transfers :
The memory of CPU that will be used for transfers with GPUs is pinned memory. In this way, the operative system will keep available that memory in RAM memory. Transfers between GPU and pinned memory are twice faster.



5.2. Latest optimisations in Multi-GPU

Overlap between CPU-GPU transfers and GPU computing:
In the previous version, transfers between CPU and GPU were always synchronous, so the process waits since the transfer is requested until it is completed.
This does not mean that there was no overlap with the background processes that are responsible for receiving and sending data in MPI.
Now asynchronous CPU-GPU transfers and CUDA streams are used to overlap GPU calculation with data transfers are also employed.

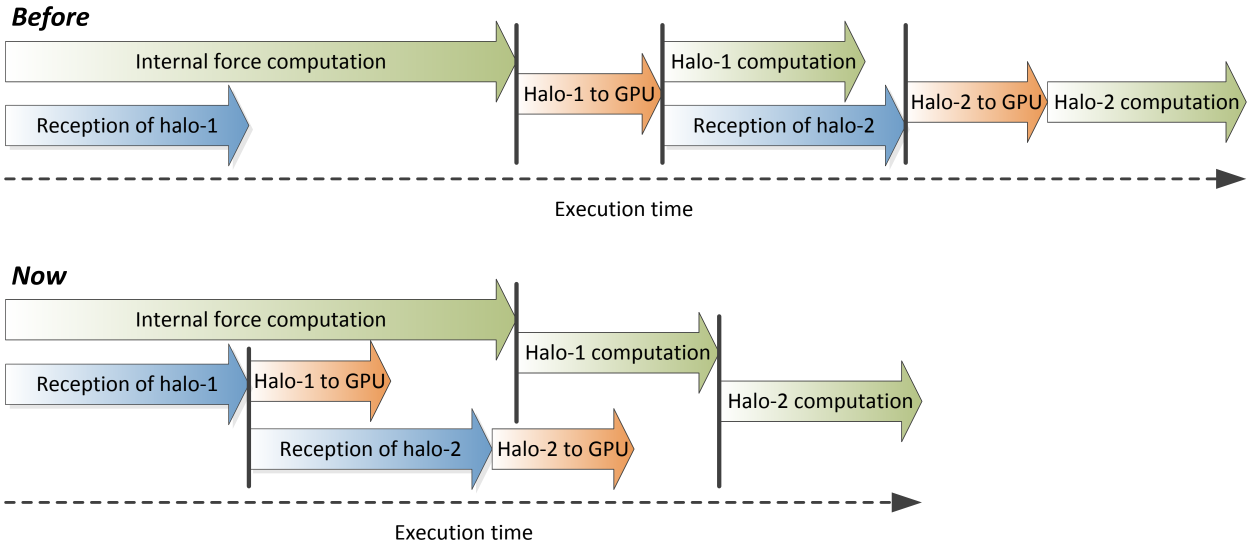


5.2. Latest optimisations in Multi-GPU

Overlap between internal force computation and reception of two halos:

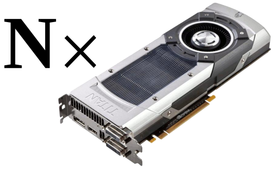
In the previous version, the computation of forces of the particles of a process (lasting long) was overlapped only with the reception of the first halo, while the reception of the second halo was overlapped only with computation of first halo (much shorter). Thus, the first halo overlapped well but not the second one.

Now, thanks to the use of asynchronous CPU-GPU transfers, the reception of both halos overlaps with the internal force computation since it is possible to complete the reception (copy data to GPU) while forces on GPU are being computed.



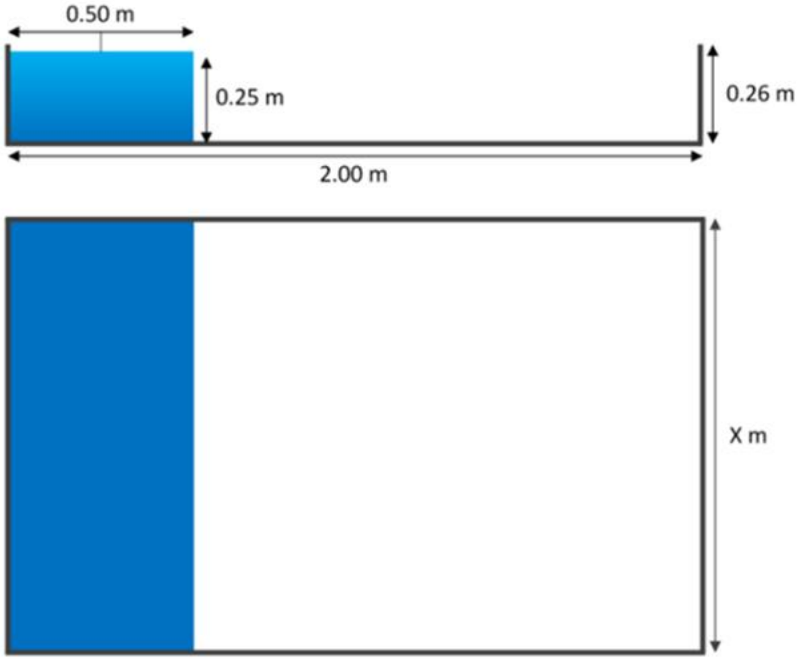
5.2. Latest optimisations in Multi-GPU

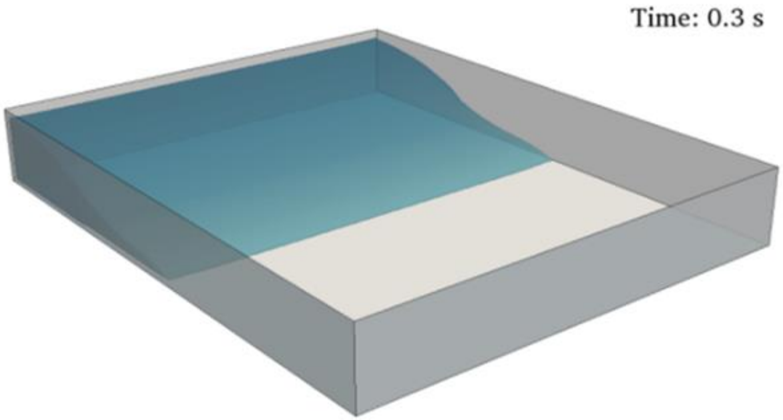
N×



Testcase for results

- **Dam break flow**.
- Physical time of simulation is **0.6 seconds**.
- The number of used particles varies from 1M to 1,024M particles.





5.2. Latest optimisations in Multi-GPU

N×

Results of efficiency

The simulations were carried out in the **Barcelona Supercomputing Center** BSC-CNS (Spain). This system is built with **256 GPUs Tesla M2090**.

All the results presented here were obtained single precision and Error-correcting code memory (ECC) disabled.

BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Activity at BARCELONA SUPERCOMPUTING CENTER:
“Massively parallel Smoothed Particle Hydrodynamics scheme using GPU clusters”

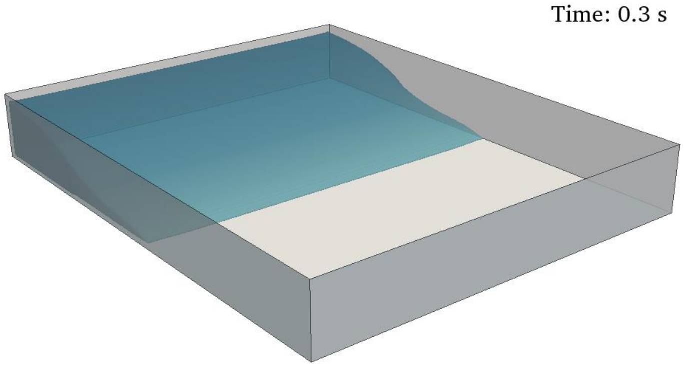
5.2. Latest optimisations in Multi-GPU

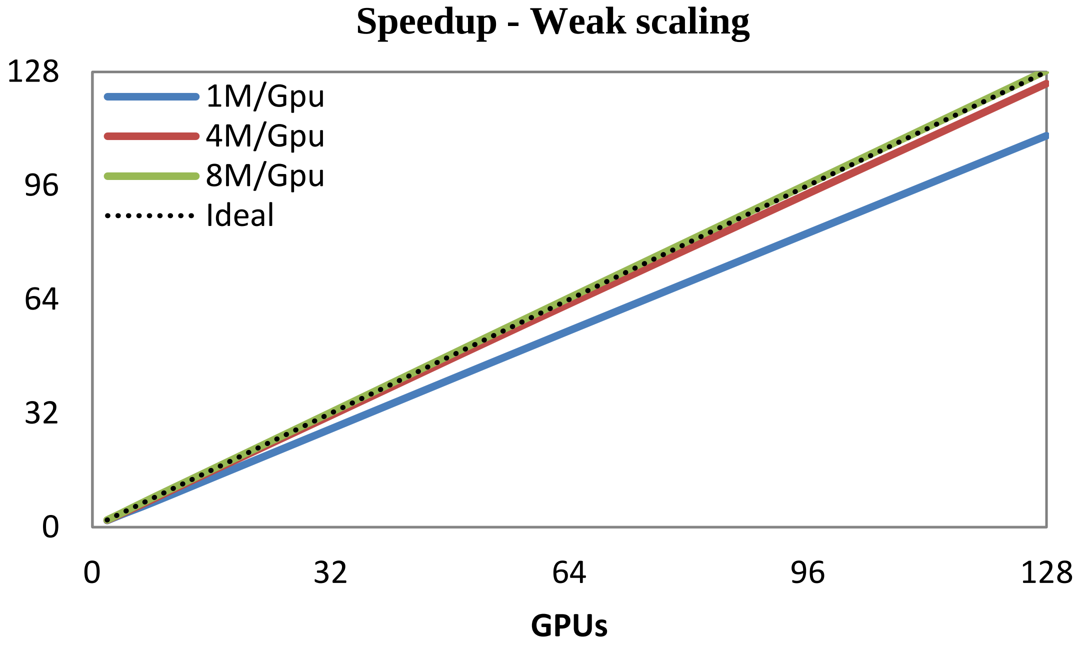
N×

Efficiency close to 100% simulating 4M/GPU with 128 GPUs Tesla M2090 of BSC.

This is possible because the time dedicated to tasks exclusive of the multi-GPU executions (communication between processes, CPU-GPU transfers and load balancing) is minimum.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

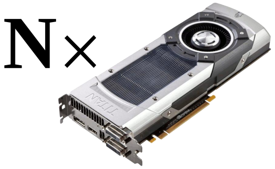




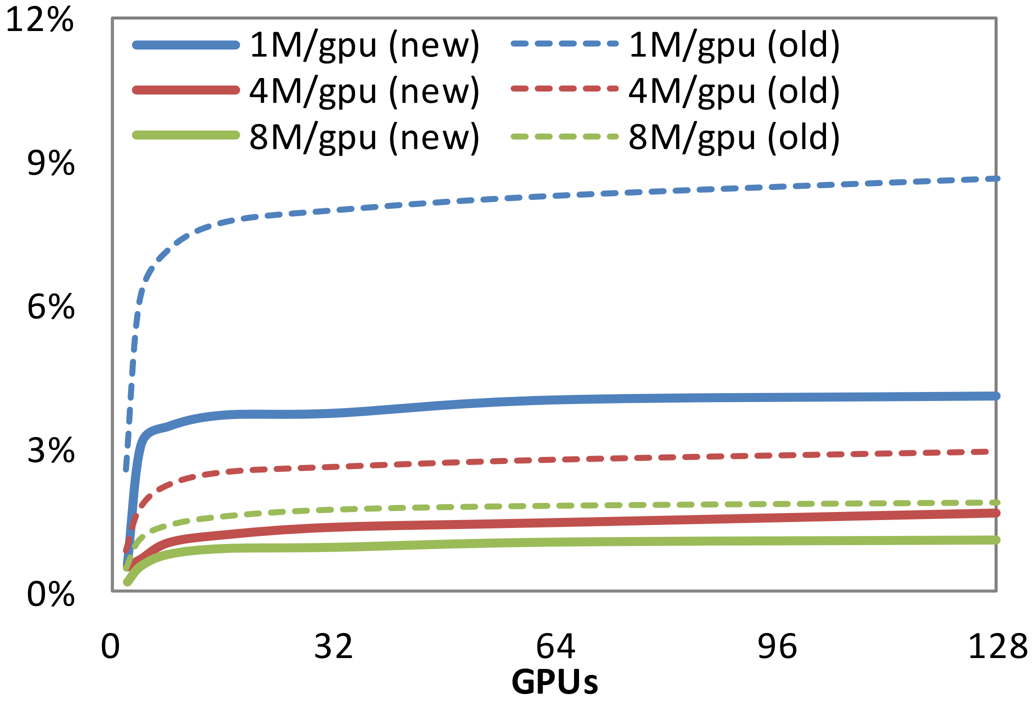
$$S(N) = \frac{T(N_{ref}) \cdot N}{T(N) \cdot N_{ref}}$$

$$E(N) = \frac{S(N)}{N}$$

5.2. Latest optimisations in Multi-GPU



Percentage of time dedicated to tasks exclusive of the multi-GPU executions including the latest improvements (solid line).



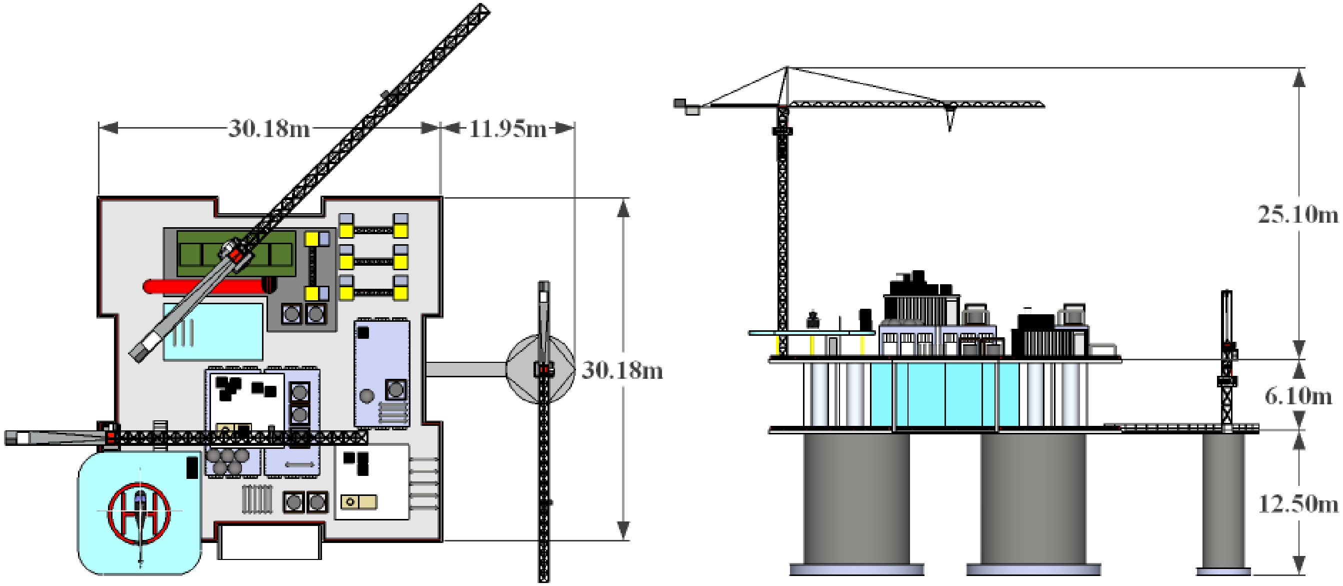
The latest improvements **have reduced this percentage by half** for different number of GPUs and different number of particles

5.3. Large simulations

64×

Simulation of 1 billion SPH particles

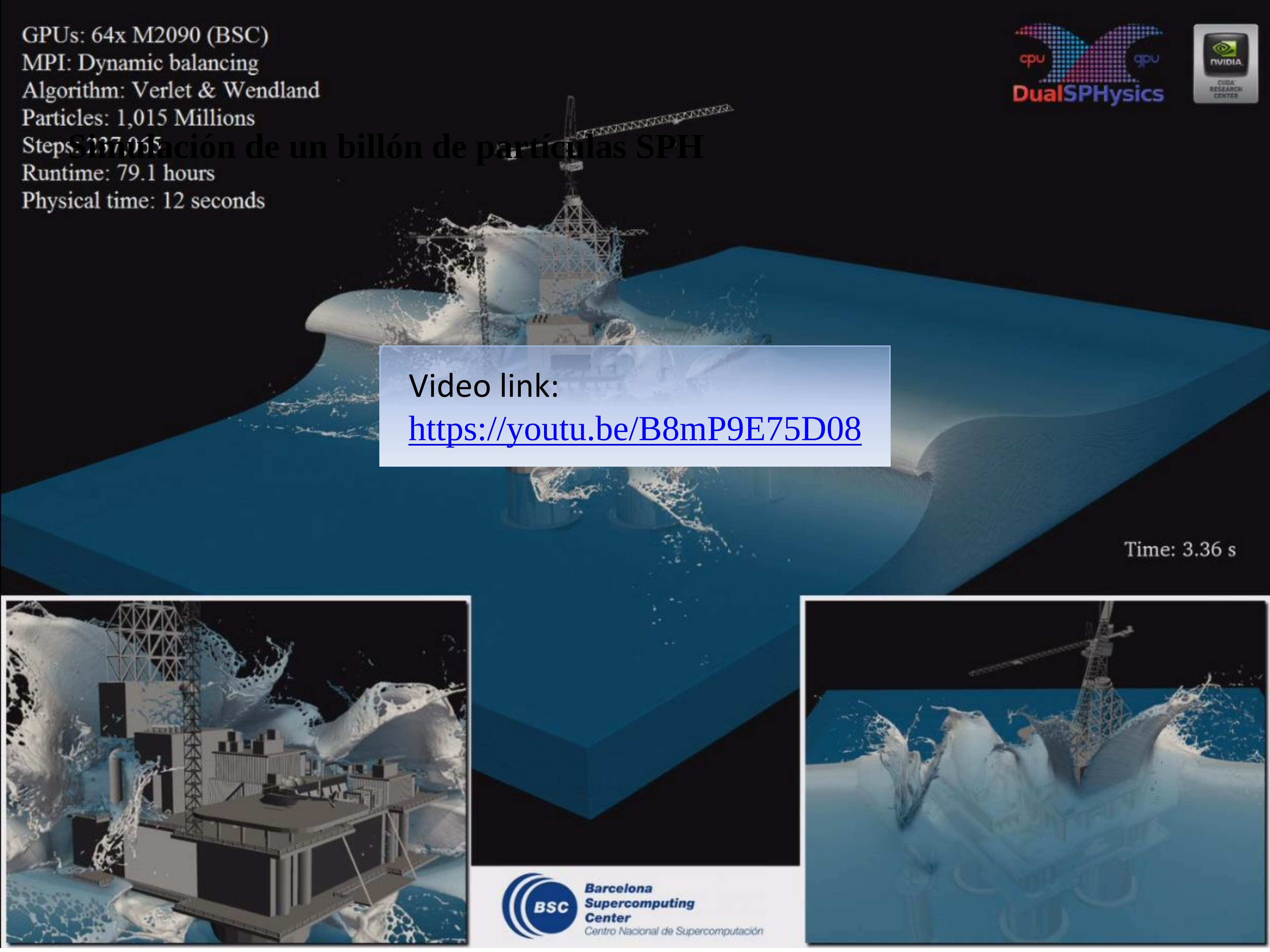
Large wave interaction with oil rig using 10^9 particles



dp= 6 cm, h= 9 cm
np = 1,015,896,172 particles
nf = 1,004,375,142 fluid particles
physical time= 12 sec
# of steps = 237,065 steps
runtime = 79.1 hours

using 64 GPUs Tesla M2090 of the BSC-CNS

Simulación de un billón de partículas SPH



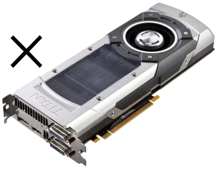
GPUs: 64x M2090 (BSC)
MPI: Dynamic balancing
Algorithm: Verlet & Wendland
Particles: 1,015 Millions
Steps: 237,065
Runtime: 79.1 hours
Physical time: 12 seconds

cpu
gpu
DualSPHysics
NVIDIA
CUDA RESEARCH CENTER
Time: 3.36 s
BSC
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Video link:
https://youtu.be/B8mP9E75D08

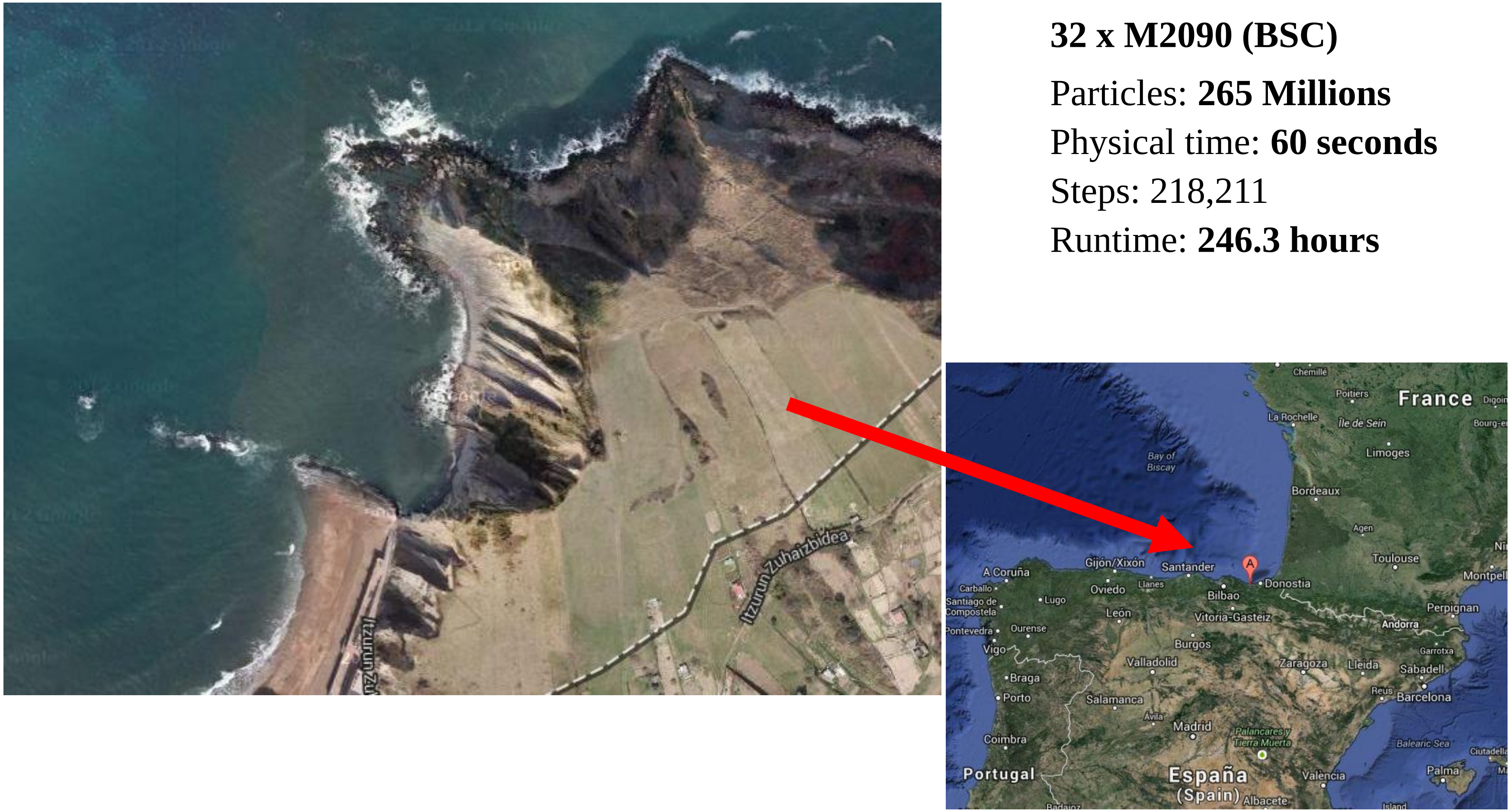
5.3. Large simulations

32×



Simulation of a real case

Using 3D geometry of the beach Itzurun in Zumaia-Guipúzcoa (Spain) in Google Earth



32 x M2090 (BSC)

Particles: **265 Millions**

Physical time: **60 seconds**

Steps: 218,211

Runtime: **246.3 hours**

Video links:
https://youtu.be/nDKlrRA_hEA
https://youtu.be/kWS6-0Z_jIo

Outline

1. Introduction

    1.1. Why is SPH too slow?

    1.2. High Performance Computing (HPC)

    1.3. DualSPHysics project

2. DualSPHysics implementation

    2.1. Implementation in three steps

    2.2. Neighbour list approaches

3. CPU acceleration

4. GPU acceleration

    4.1. Parallelization problems in SPH

    4.2. GPU optimisations

5. Multi-GPU acceleration

    5.1. Dynamic load balancing

    5.2. Latest optimisations in Multi-GPU

    5.3. Large simulations

6. Future improvements

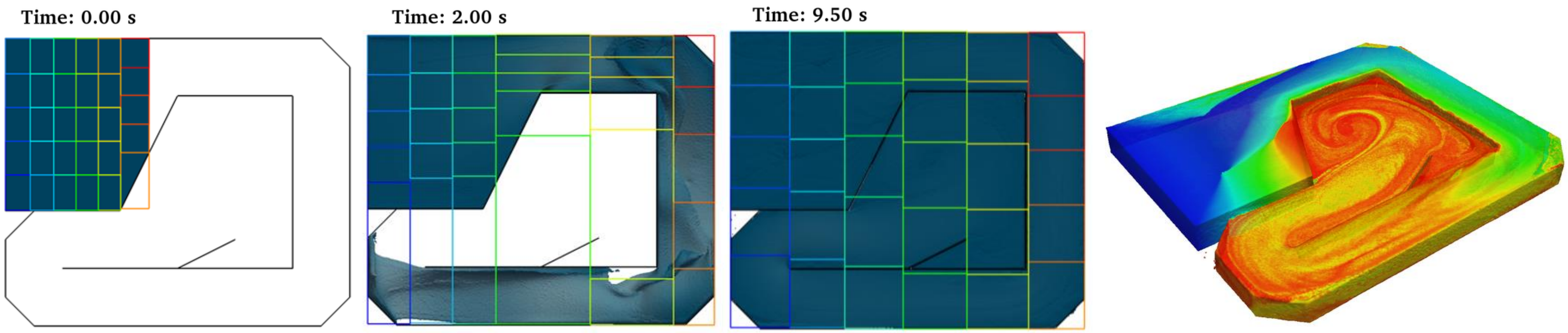
6. Future improvements

N×

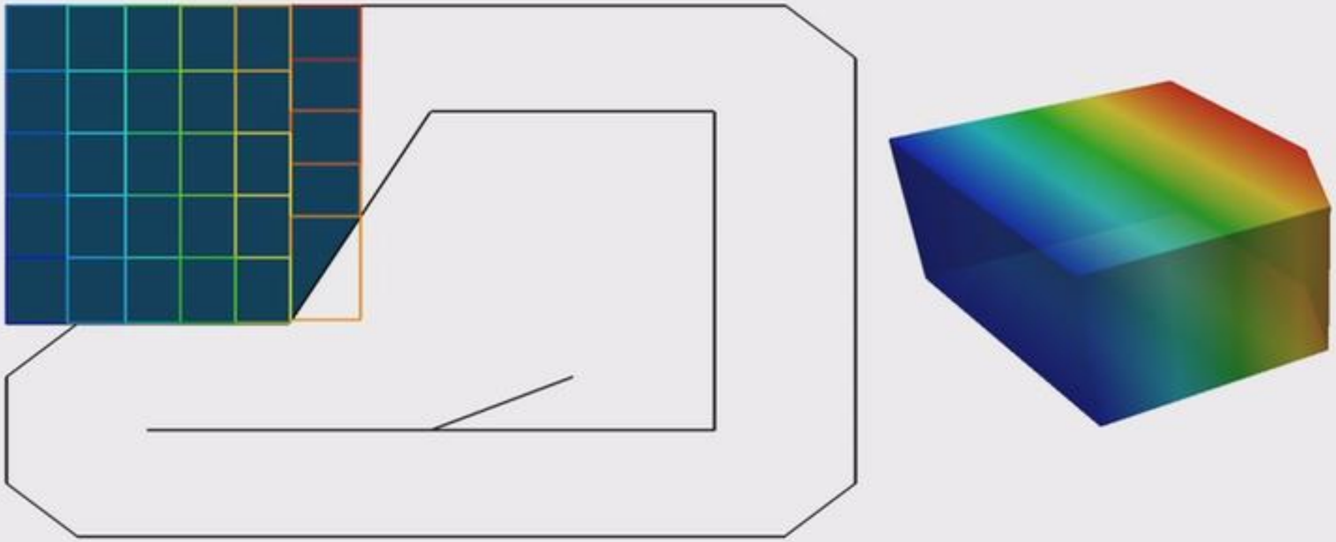
Decomposition in 2D and 3D for Multi-GPU

- Now only 1D but 2D and 3D will be implemented in the future.
- 1D approach is correct when the domain of simulation is very narrow but this approach is not well adapted to other domains.
- A 2D and 3D decomposition is necessary for a better distribution of the work load when using hundreds of GPUs.

Example of the 2D decomposition we are working on



2D decomposition

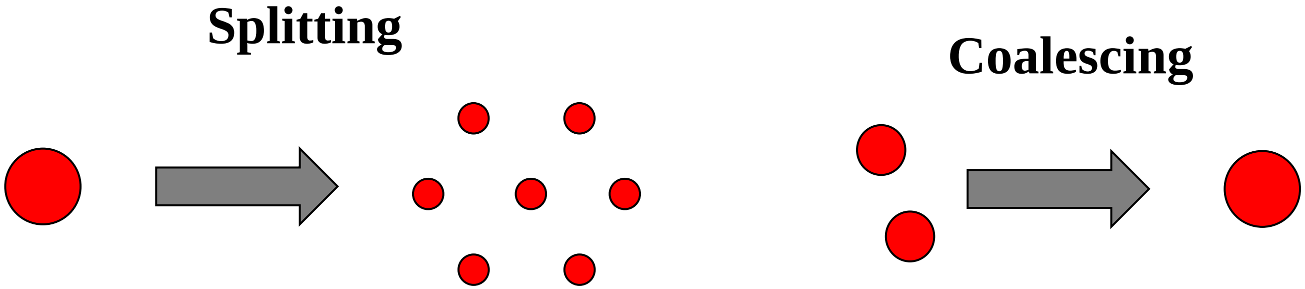


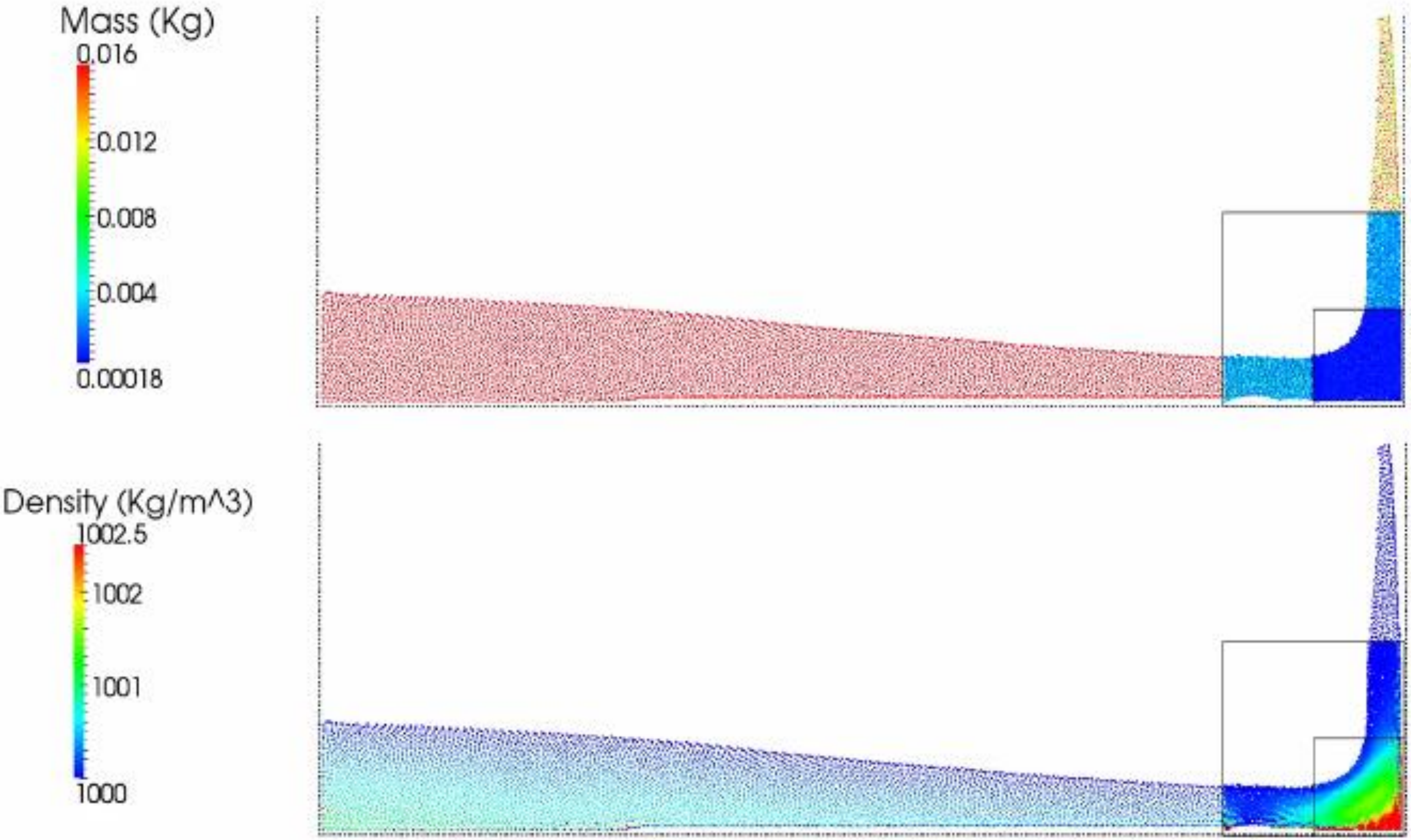
Time: 0.00 s

6. Future improvements

Variable resolution (splitting & coalescing)

- Variable resolution is imperative to simulate large problems with SPH.
- Higher resolution is only used where it is necessary, to reduce the number of particles to simulate.

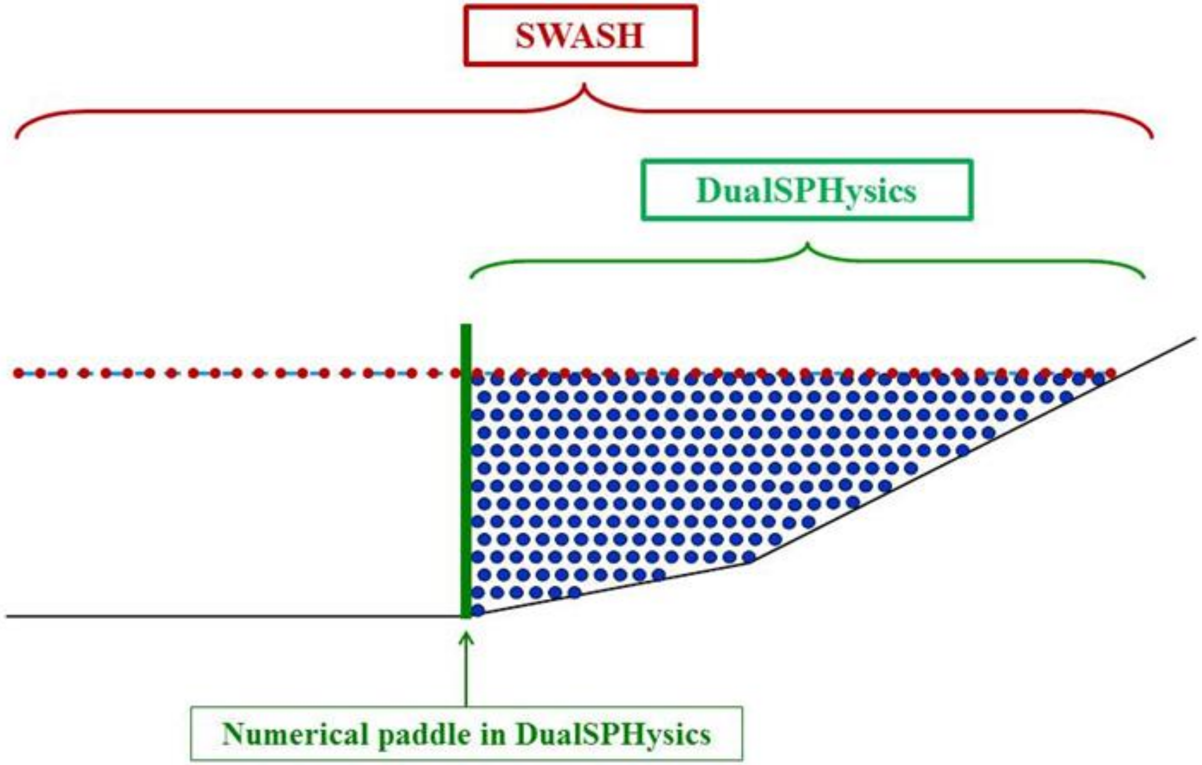


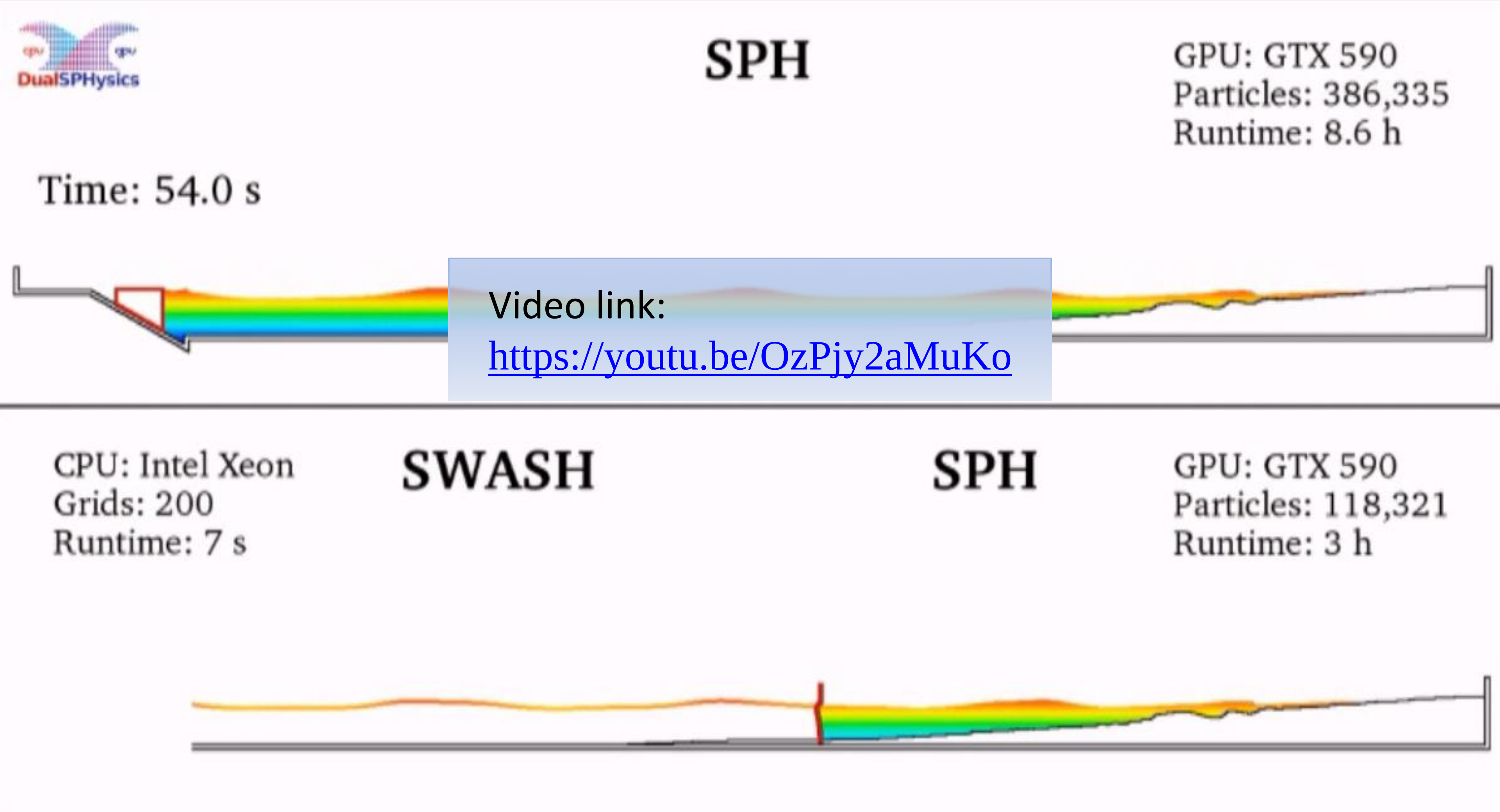

Mass (Kg)
0.016
0.012
0.008
0.004
0.00018
Density (Kg/m^3)
1002.5
1002
1001
1000

6. Future improvements

Coupling between SWASH and SPH

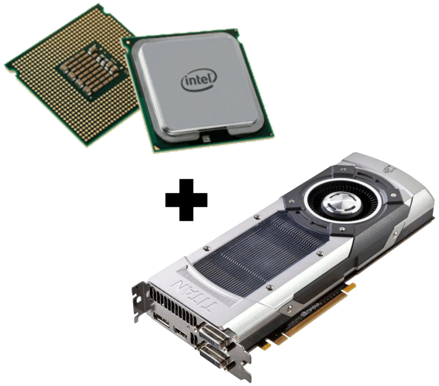
- The study of wave propagation from deep ocean to near shore is difficult using a single model because multiple scales are present both in time and in space.
- A hybrid model is necessary to combine capabilities of a wave propagation model (SWASH) and DualSPHysics.

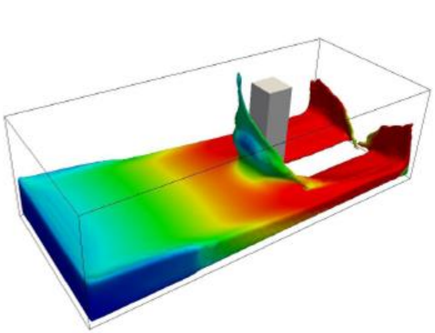


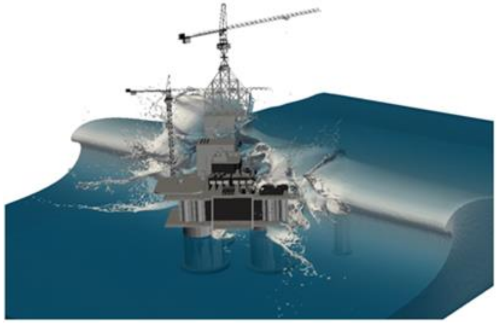


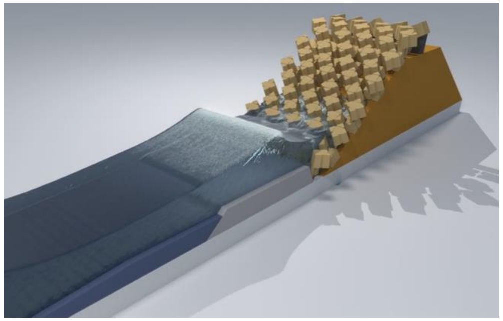
Video link: https://youtu.be/OzPjy2aMuKo

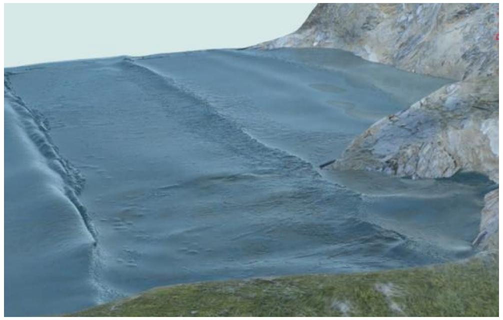
Optimisation and SPH Tricks











José Manuel Domínguez Alonso

jmdominguez@uvigo.es

EPHYSLAB, Universidade de Vigo, Spain