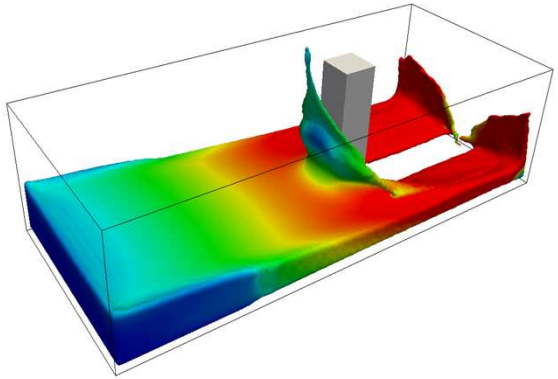
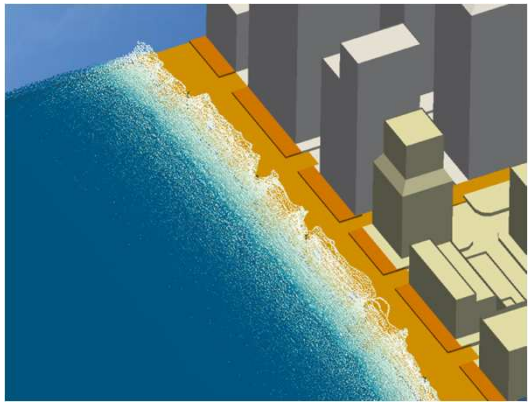
Aplicación de nuevas tecnologías HPC a un código de simulación de fluidos



cpu gpu

DUAL-SPHysics



A.J.C. Crespo, J.M. Dominguez, A. Barreiro and M. Gómez-Gesteira

Environmental | Physics | Laboratory

EPhysLab

Universidade Vigo

Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

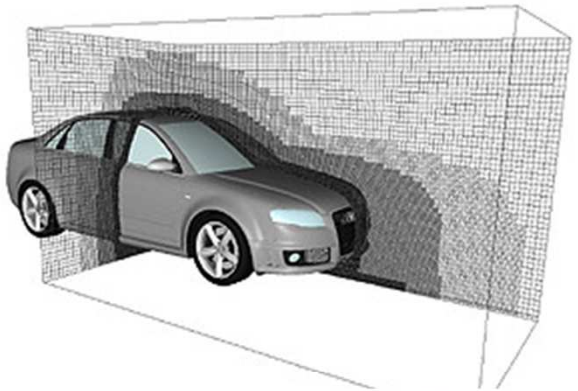
Numerical methods

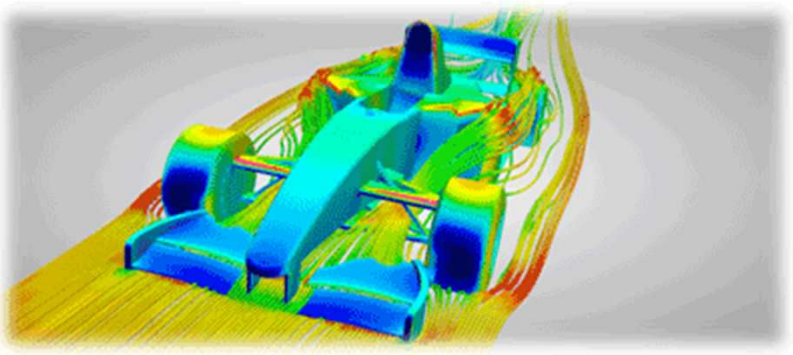
Definition and motivation

Numerical methods are **useful tools in engineering and science** to solve complex problems.

The idea is **simulating a physical problem with a numerical model**.

Its main advantage is to simulate complex scenarios **without building costly scale models**, and provide **data difficult or impossible to measure in a real model**.





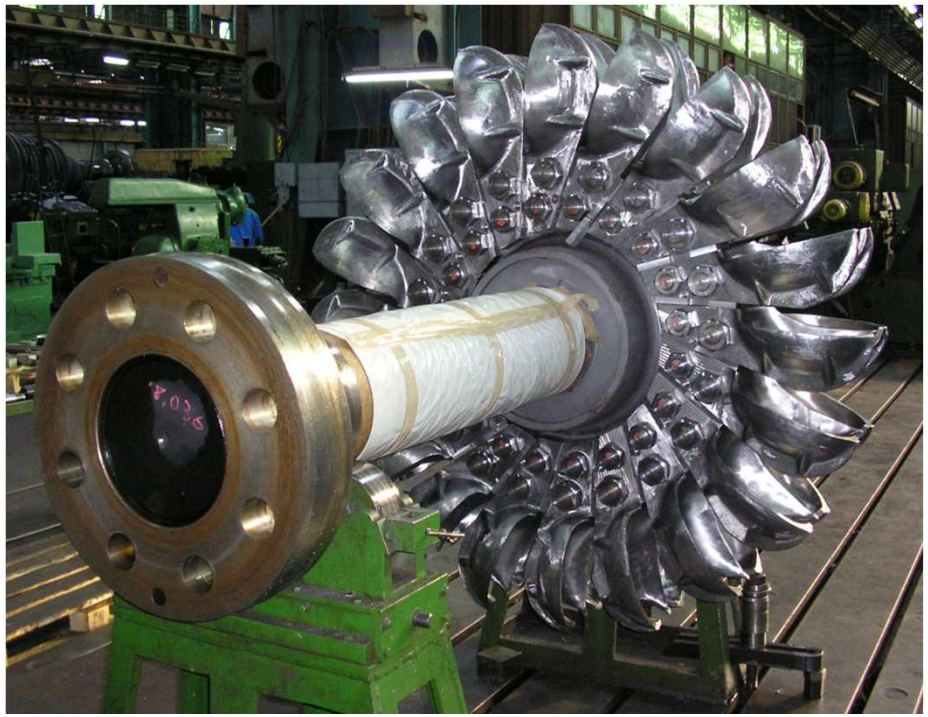
Numerical methods

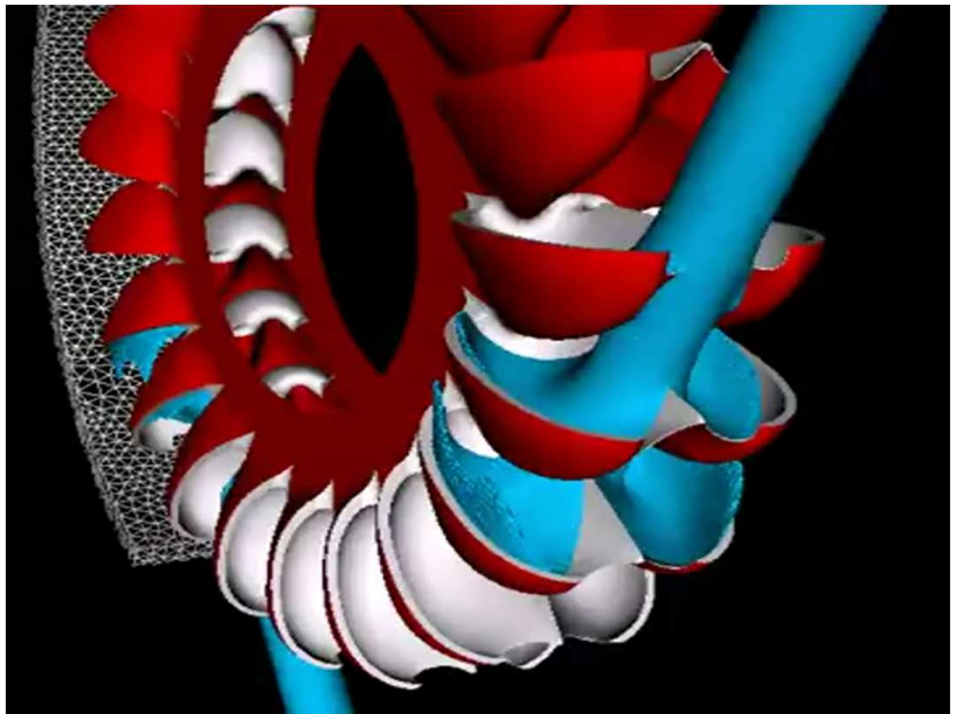
Definition and motivation

Numerical methods are **useful tools in engineering and science** to solve complex problems.

The idea is **simulating a physical problem with a numerical model**.

Its main advantage is to simulate complex scenarios **without building costly scale models**, and provide **data difficult or impossible to measure in a real model**.

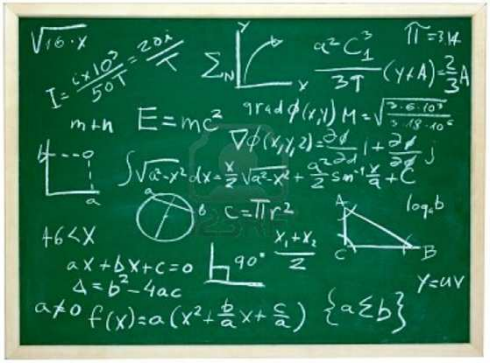




Numerical methods

Aspects to build an accurate and fast numerical method

1) Correct implementation of the physical governing equations and the accuracy of the mathematical algorithms. **IT IS UP TO YOU**



2) Powerful hardware to execute the simulation. **IT IS NOT UP TO US**

Therefore, in order to obtain the best performance, the code must be optimized and parallelized as much as possible according to the available resources of hardware.

Numerical methods

Current hardware to execute our numerical methods

Central Processing Units (CPUs):

The current CPUs have **multiple processing cores,** making possible the distribution of the workload of a program among the different cores dividing the execution time.

CPUs also present **SIMD instructions** (Single Instruction, Multiple Data) that allow an operation on multiple data simultaneously.

The **parallelization** task on CPUs can be mainly performed by **using MPI (Message Passing Interface) or OpenMP (Open Multi-Processing).**

Graphics Processing Units (GPUs):

Research can be also conducted with the new GPU technology for problems that previously required high performance computing (HPC).

Recently the GPGPU programming (General Purpose on Graphics Processing Units) has experienced a strong growth in all fields of the scientific computing.

A DETAILED DESCRIPTION ABOUT GPU WILL BE SHOWN LATER

TOP SUPERCOMPUTERS IN THE WORLD

TOP500 List - November 2010 (1-100)

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

Power data in KW for entire system

http://www.top500.org/list/2010/11/100

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
2	DOE/SC/Oak Ridge National Laboratory United States	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	224162	1759.00	2331.00	6950.60
3	National Supercomputing Centre	Nebulae - Dawning TC3600 Blade, Intel				

1º Tianhe-1A (China) 2.57 petaflops/s (consumption: 4040.00 KW) (with GPUs)

2º Jaguar (USA) 1.75 petaflops/s (consumption: 6950.60 KW) (CPUs)

5	DOE/SC/LBNL/NERSC United States	Hopper - Cray XE6 12-core 2.1 GHz / 2010 Cray Inc.	153408	1054.00	1288.63	2910.00
6	Commissariat a l'Energie Atomique (CEA) France	Tera-100 - Bull bullx super-node S6010/S6030 / 2010 Bull SA	138368	1050.00	1254.55	4590.00
7	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78	2345.50
8	National Institute for Computational Sciences/University of Tennessee United States	Kraken XT5 - Cray XT5-HE Opteron 6-core 2.6 GHz / 2009 Cray Inc.	98928	831.70	1028.85	3090.00
9	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution / 2009 IBM	294912	825.50	1002.70	2268.00
10	DOE/NNSA/LANL/SNL United States	Cielo - Cray XE6 8-core 2.4 GHz / 2010 Cray Inc.	107152	816.60	1028.66	2950.00

TOP SUPERCOMPUTERS IN THE WORLD

TOP500 List - June 2011 (1-100)

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

Power data in KW for entire system

http://www.top500.org/list/2011/06/100

Rank	Site	Computer/Year Vendor	Cores	R_{max}	R_{peak}	Power
1	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect / 2011 Fujitsu	548352	8162.00	8773.63	9898.56
2	National Supercomputing Center in Tianjin China	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C / 2010 NUDT	186368	2566.00	4701.00	4040.00
5	Technology Japan	Xeon 6C X5670, Nvidia GPU, Linux/Windows / 2010 NEC/HP	73278	1192.00	2287.63	1398.61
9	(CEA) France	S6010/S6030 / 2010 Bull SA	138368	1050.00	1254.55	4590.00
10	DOE/NNSA/LANL United States	Roadrunner - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband / 2009 IBM	122400	1042.00	1375.78	2345.50

1º K computer (Japan) 8.16 petaflops/s (consumption: 9898.56 KW) (SPARC64)

2º Tianhe-1A (China) 2.57 petaflops/s (consumption: 4040.00 KW) (with GPUs)

3º Jaguar (USA) 1.75 petaflops/s (consumption: 6950.60 KW) (CPUs)

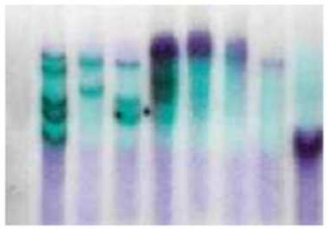
The use of GPU co-processors is consolidated as a key component in HPC

GPUs are an accessible tool to accelerate SPH,
all numerical methods in CFD and any computational method

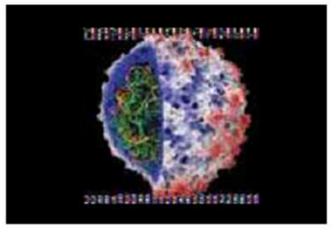
5X
Digital Content Creation
Adobe

18X
Video Transcoding
Elemental Technologies

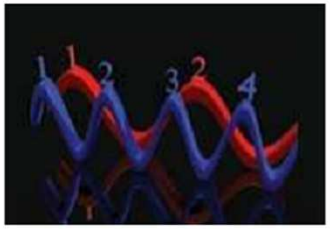
20X
3D Ultrasound
TechniScan



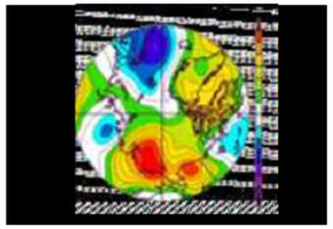
30X
Gene Sequencing
U of Maryland



36X
Molecular Dynamics
U of Illinois, Urbana-Champaign



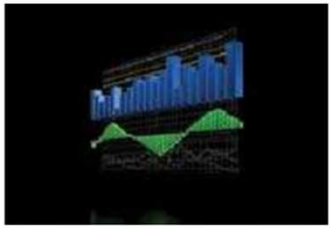
50X
MATLAB Computing
AccelerEyes



80X
Weather Modeling
Tokyo Institute of Technology

100X
Astrophysics
RIKEN

146X
Medical Imaging
U of Utah



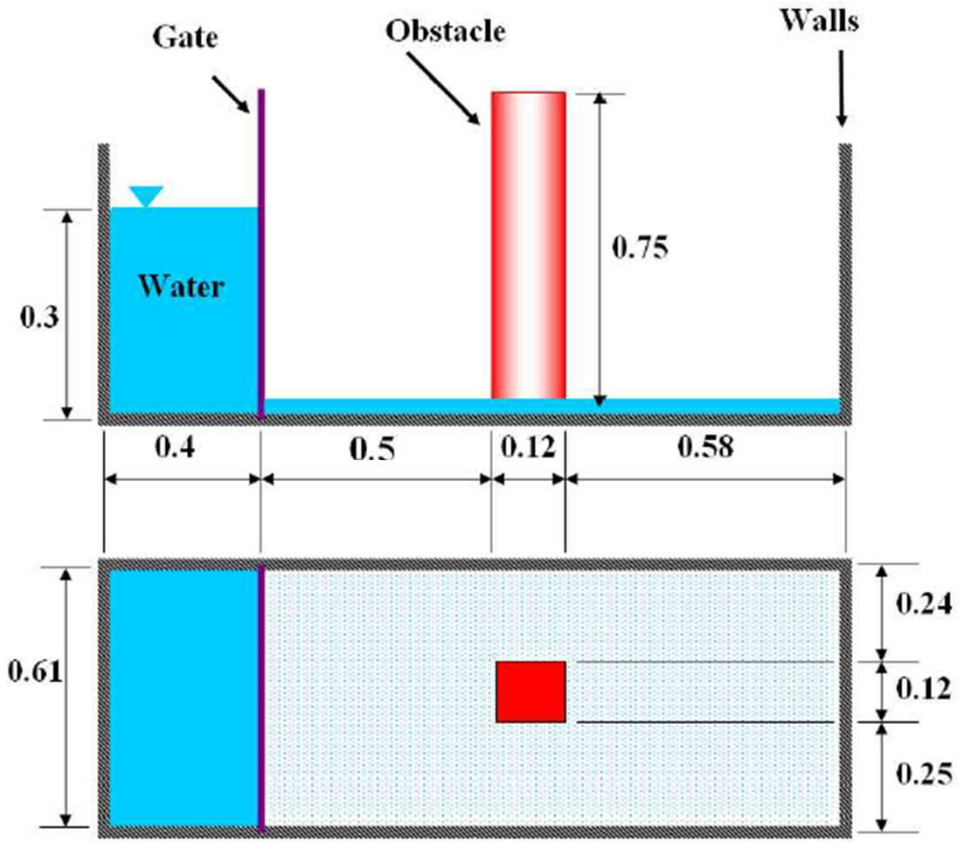
149X
Financial Simulation
Oxford University

http://www.nvidia.com

Numerical methods

How long can be a simulation???

30,000 particles
1.5s of physical time
prototype dimensions
simple geometries



Yeh and Petroff experiment

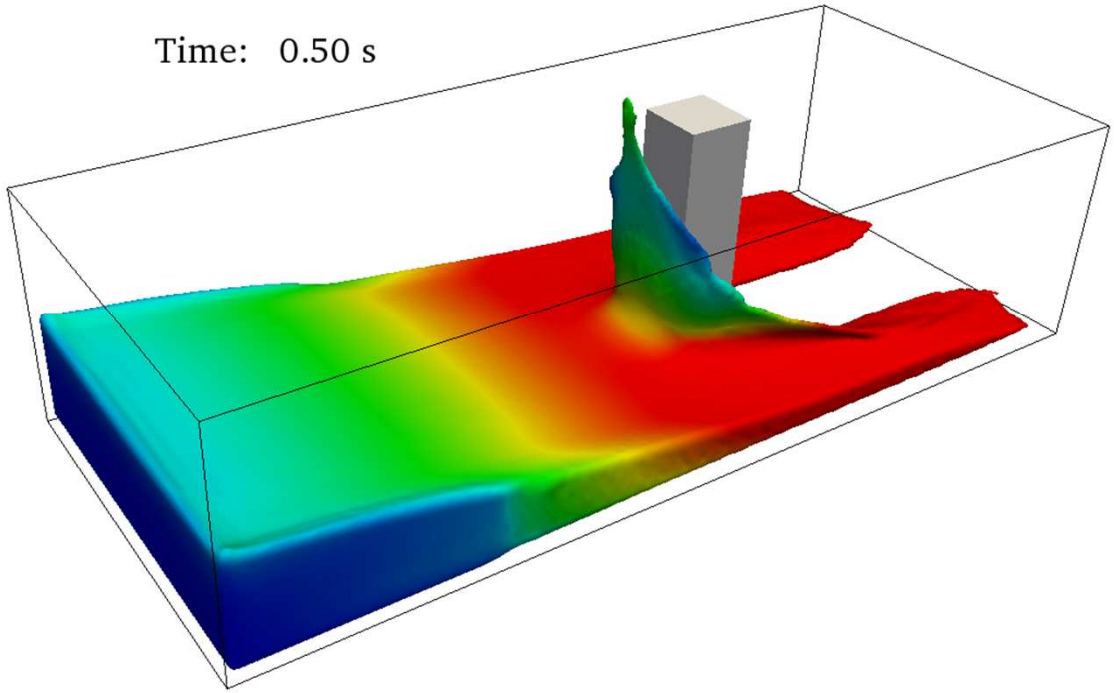
Numerical methods

How long can be a simulation???

2004: over 2-3 hours

2008: 1 hour

2010: 12 mins en single-core CPU
3 mins en multi-core CPU
17 segundos en GPU



Numerical methods

How long can be a simulation???

2004: over 2-3 hours

2008: 1 hour

2010: 10 mins en single-core CPU
3 mins en multi-core CPU
17 segundos en GPU

2011???

Numerical methods

How long can be a simulation???

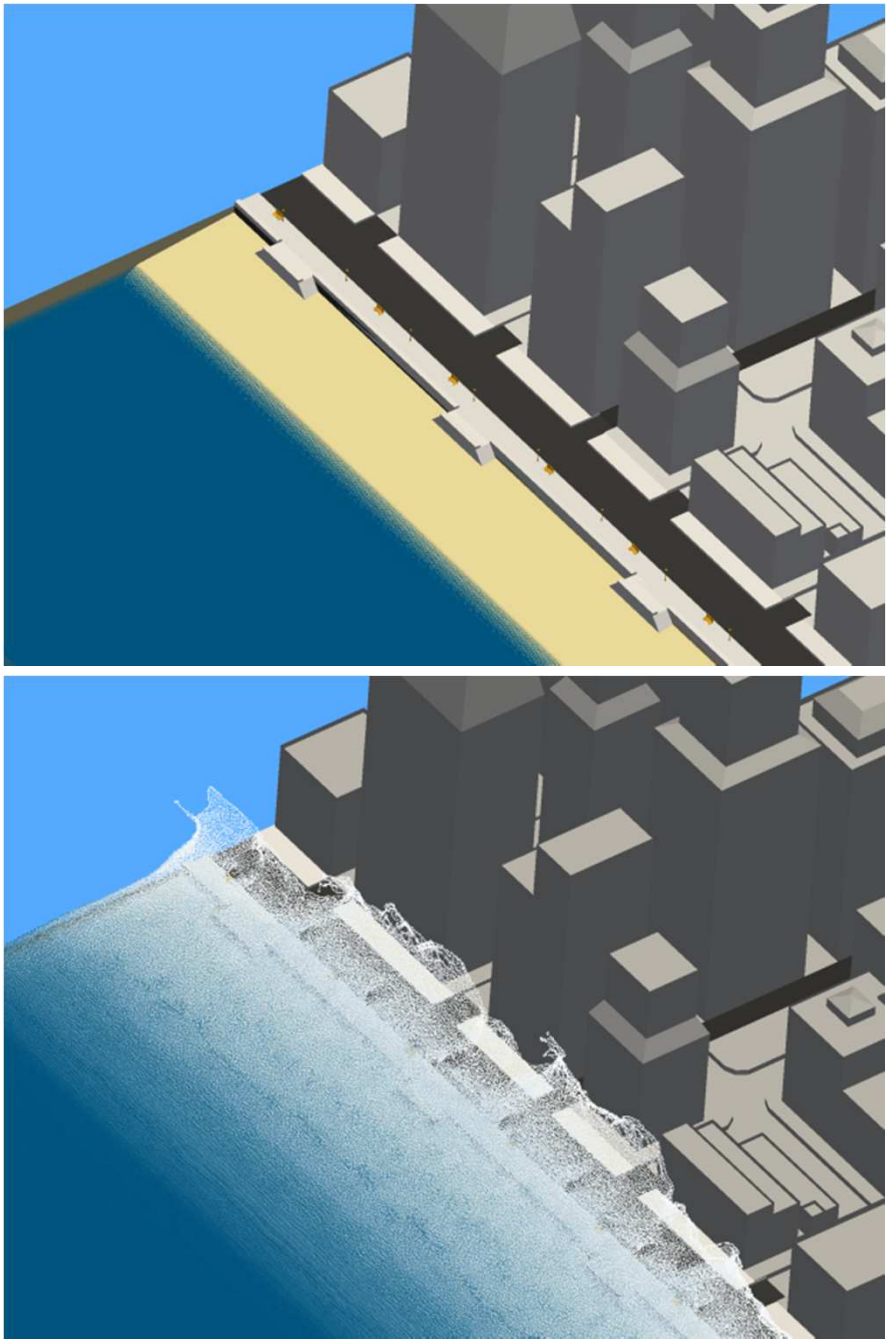
2004: over 2-3 hours

2008: 1 hour

2010: 10 mins en single-core CPU
3 mins en multi-core CPU
17 segundos en GPU

2011:
5,000,000 particles
24s of physical time
real dimensions
complex geometries

3 hours on GPU



Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

SPH method

Different descriptions for numerical methods

The physical governing equations of a numerical method can be solved:

- with the help of a grid using an Eulerian description or
- without it with a Lagrangian description.

The meshfree methods make easier the simulation of problems:

- with large deformations
- advanced material
- complex geometries
- nonlinear material behavior
- discontinuities and singularities.

Meshfree methods are used for solid mechanics as well as for fluid dynamics.

We will focus here on the meshfree particle method named SPH:

Smoothed Particle Hydrodynamics

SPH method

PHYSICAL GOVERNING EQUATIONS

EULERIAN DESCRIPTION
(spatial description)

LAGRANGIAN DESCRIPTION
(material description)

COMPUTATIONAL METHODS

GRID-BASED METHODS

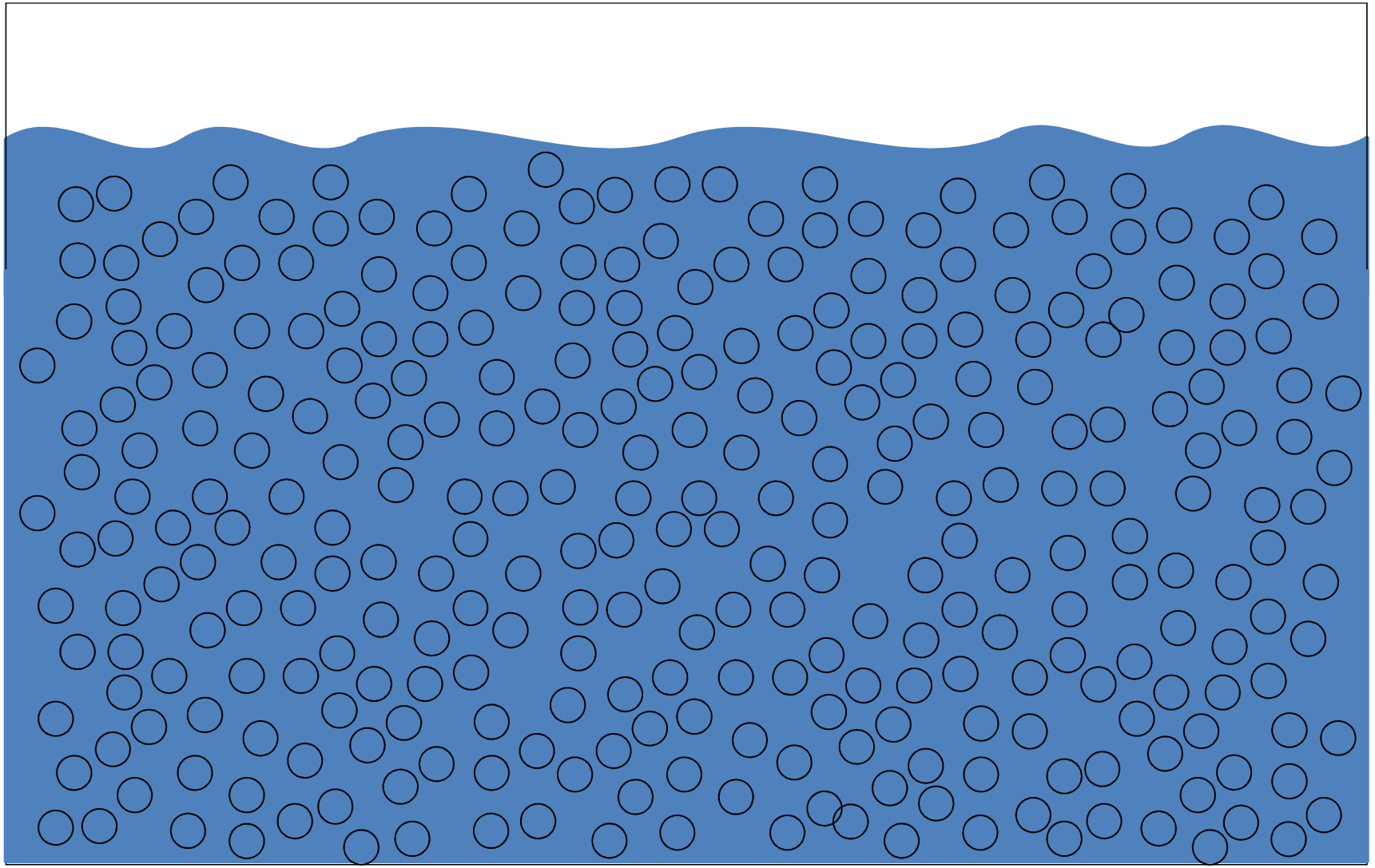
MESHFREE METHODS

MESHFREE PARTICLE METHODS
(particle represents a part of the continuum domain)

SMOOTHED PARTICLE HYDRODYNAMICS

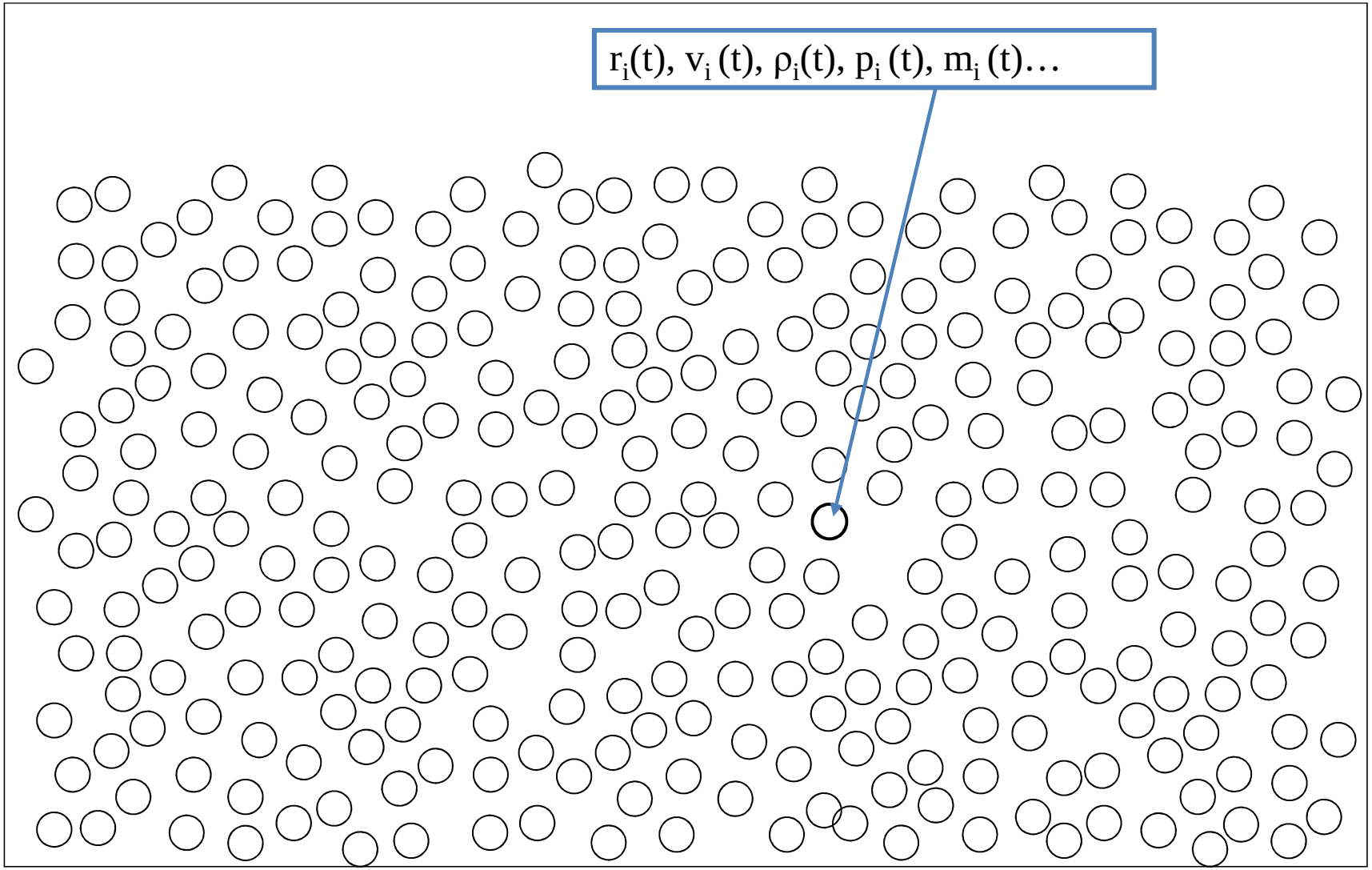
SPH method

The fluid is treated as a set of particles.



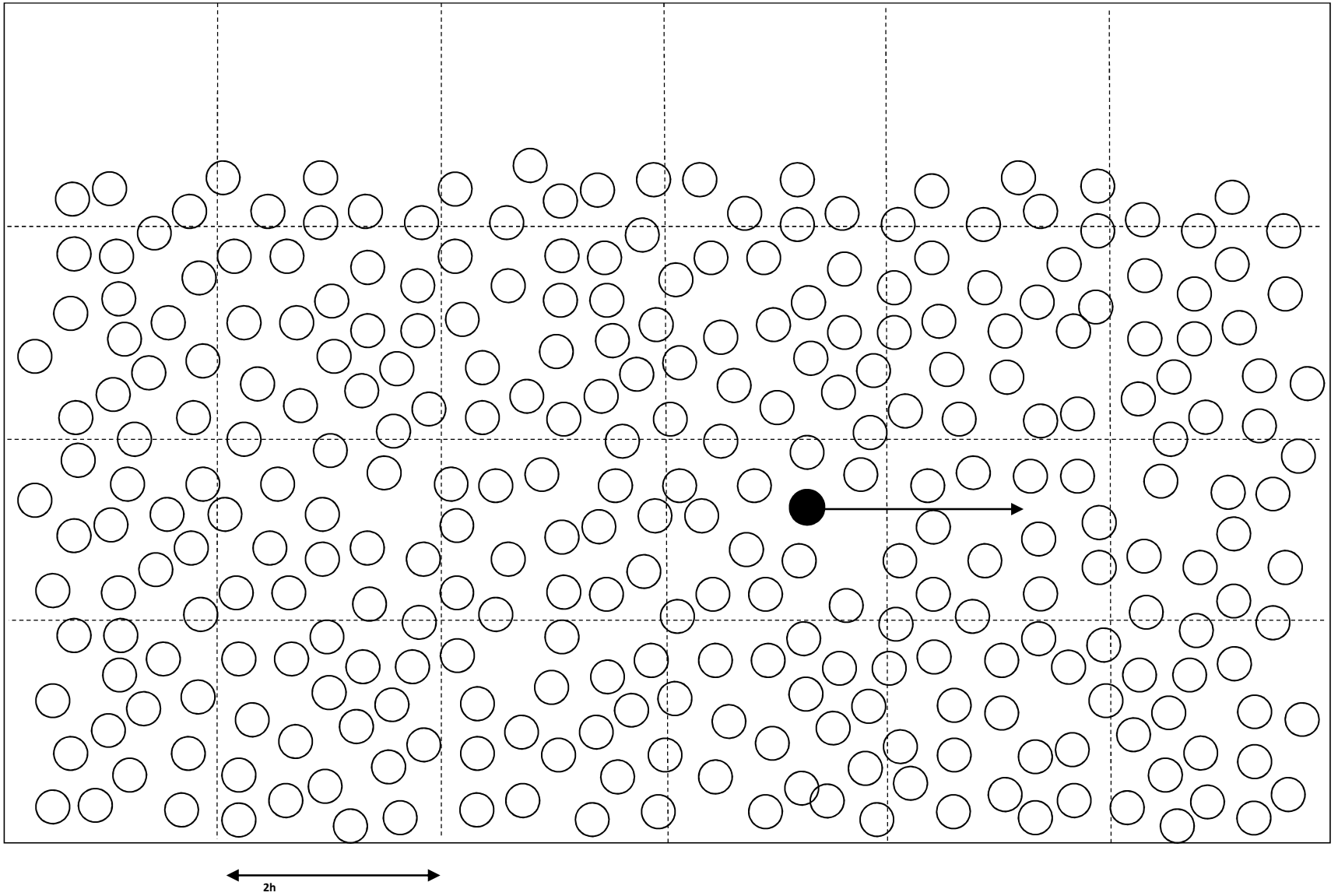
SPH method

Position, velocity, mass, density, pressure of each particle is known.



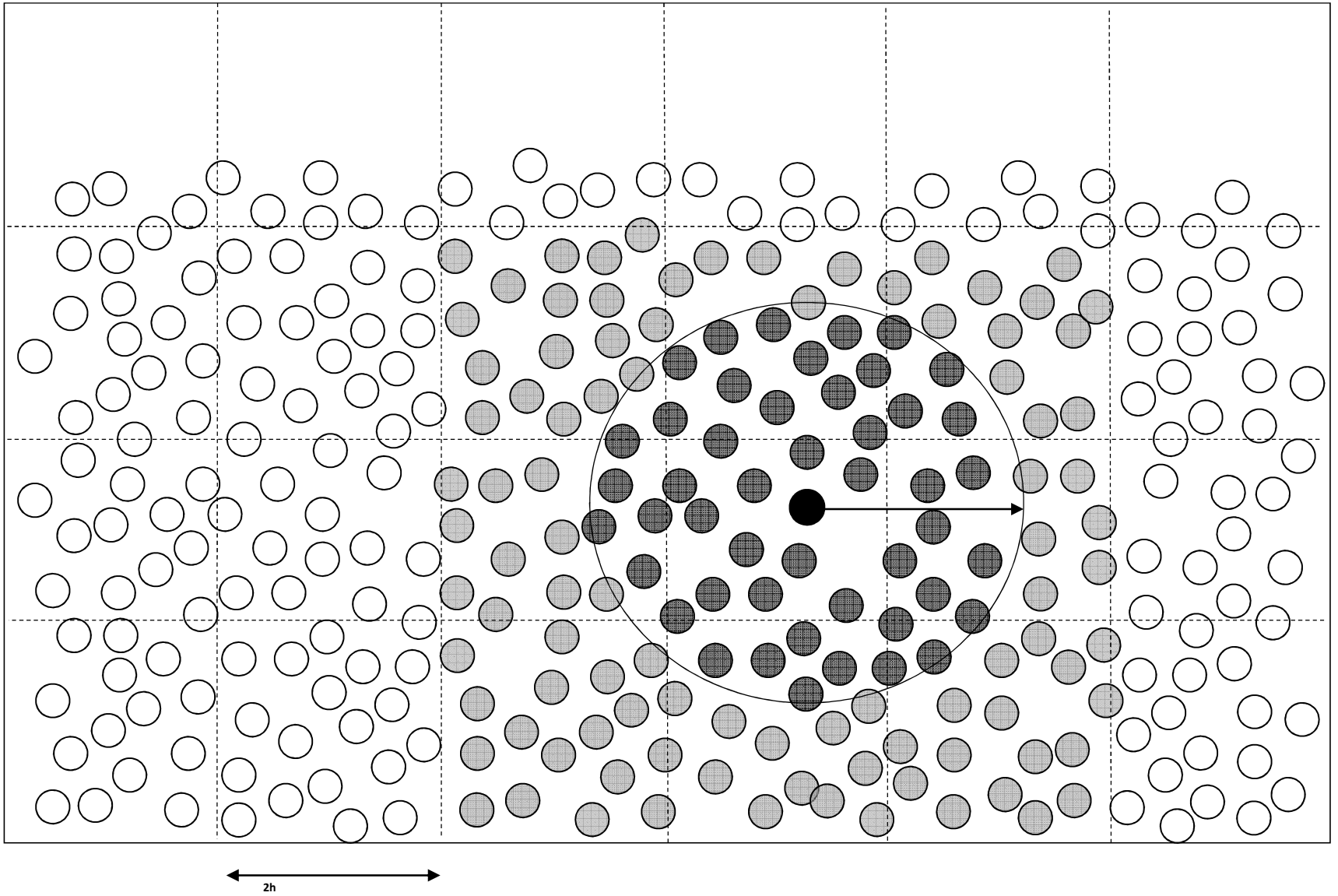
SPH method

NEIGHBOR LIST



SPH method

NEIGHBOR LIST



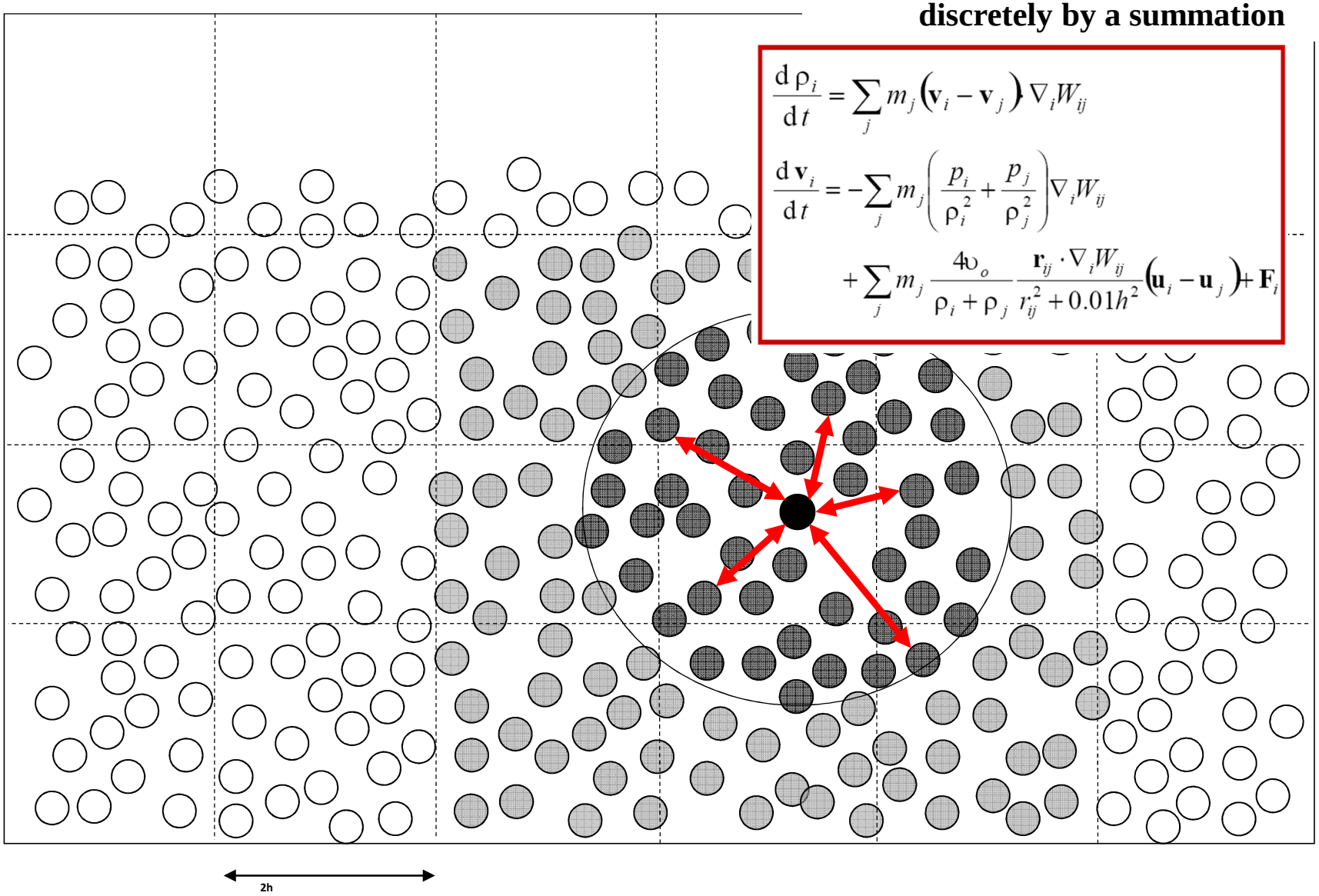
SPH method

NEIGHBOR LIST | PARTICLE INTERACTION

Navier-Stokes equations approximated discretely by a summation

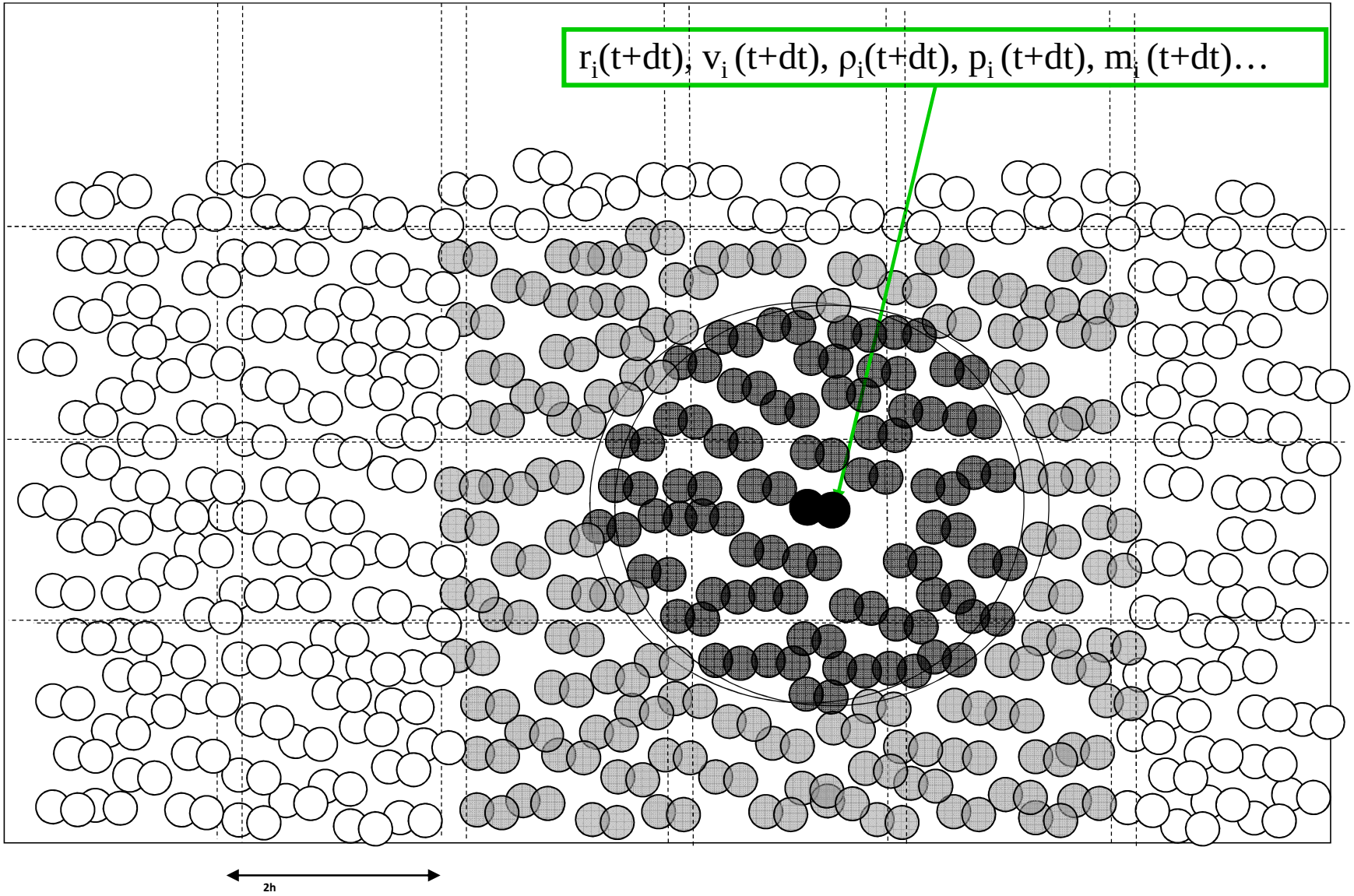
$$\frac{\mathrm{d}\rho_i}{\mathrm{d}t} = \sum_j m_j (\mathbf{v}_i - \mathbf{v}_j) \nabla_i W_{ij}$$

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = -\sum_j m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla_i W_{ij} + \sum_j m_j \frac{4\upsilon_o}{\rho_i + \rho_j} \frac{\mathbf{r}_{ij} \cdot \nabla_i W_{ij}}{r_{ij}^2 + 0.01h^2} (\mathbf{u}_i - \mathbf{u}_j) + \mathbf{F}_i$$



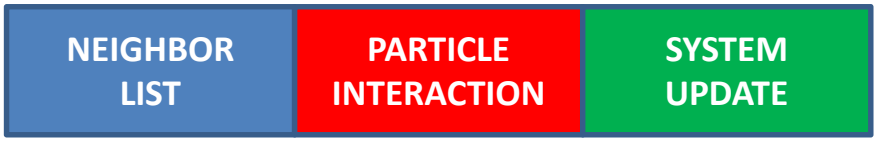
SPH method

NEIGHBOR LIST | PARTICLE INTERACTION | SYSTEM UPDATE



SPH method

- Conceptually, an SPH code is an iterative process consisting of three main steps:

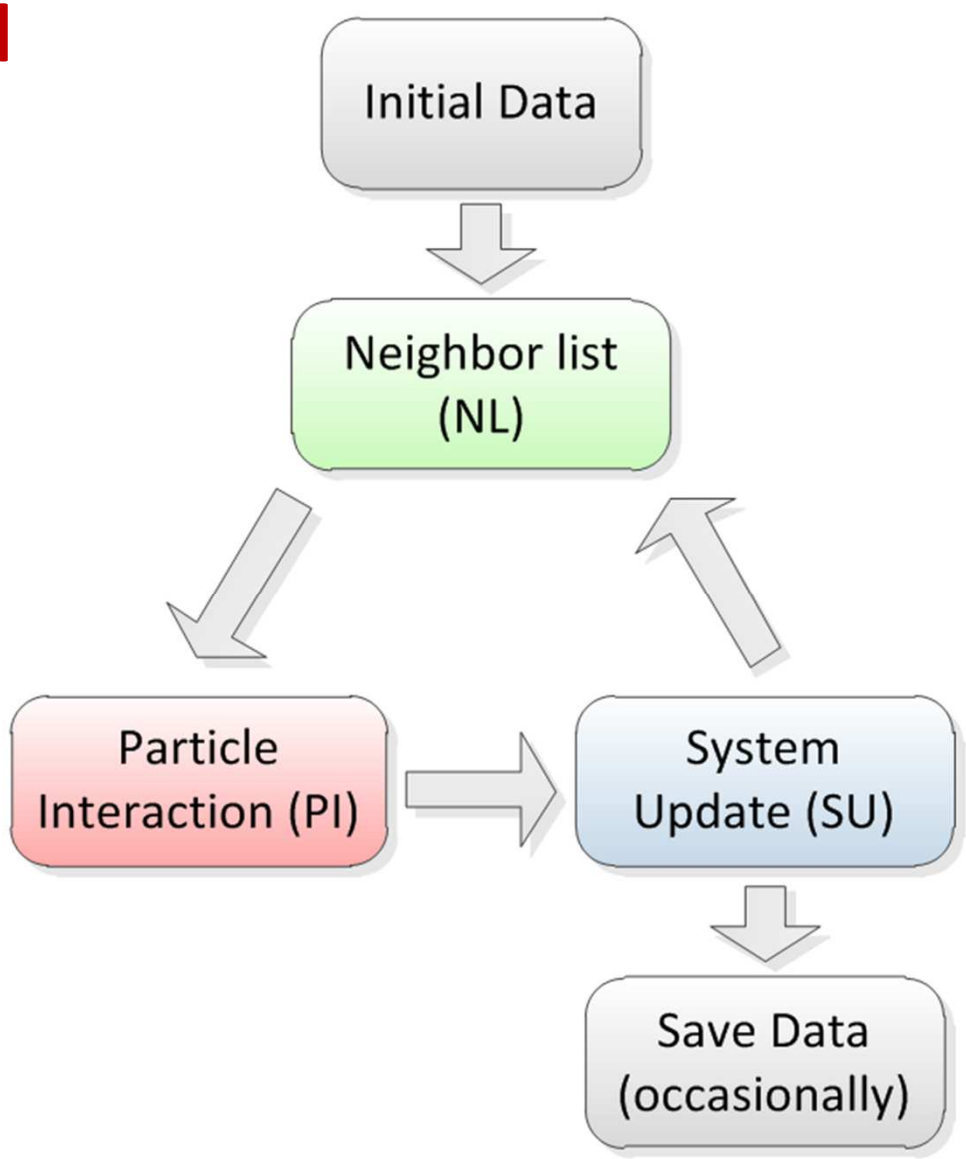


-***neighbour list***:
particles only interact with surrounding particles located at a given distance so the domain is divided in cells of the kernel size to reduce the neighbour search to the adjacent cells;

-***particle interaction*:**
each particle only looks for neighbours at the adjacent cells, after verifying that the distance between particles lies within the support of the kernel, the conservation laws of continuum fluid dynamics are computed for the pair-wise interaction of particles;

-***system update***:
once the forces between neighbouring particles have been evaluated, all physical magnitudes of the particles are updated at the next time step.

SPH method



Conceptual diagram of the implementation of a SPH code

SPH method: drawbacks

The applicability of particle-based simulations is typically limited by **two constraints: (i) simulation time, and (ii) system size**

Thus, **to obtain physically meaningful information** from a simulation, one must be able to **simulate a large-enough system for long-enough times**.

In the SPH method, applications such as the study of coastal processes and flooding hydrodynamics, have been **limited until now by the maximum number of particles** in order to perform simulations within reasonable times.

Big simulations for free-surface flows:

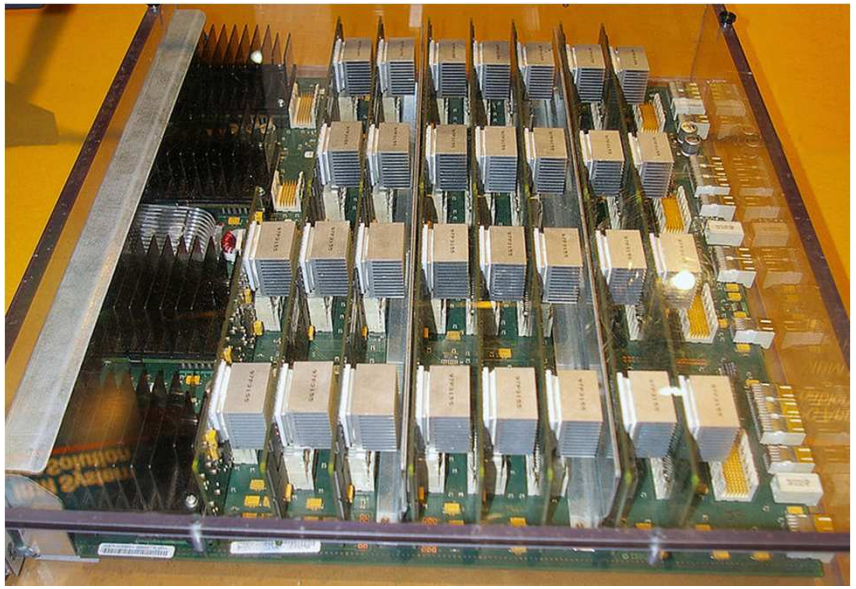
40 million – EDF on a Blue Gene
120 million – EPFL on a Blue Gene
200 million – ECL

SPH method: drawbacks

Blue Gene is a computer architecture project to produce several supercomputers, designed to reach operating speeds in the PFLOPS (petaFLOPS) range, and currently reaching sustained speeds of nearly 500 TFLOPS (teraFLOPS): Blue Gene/L, Blue Gene/C, Blue Gene/P, and Blue Gene/Q

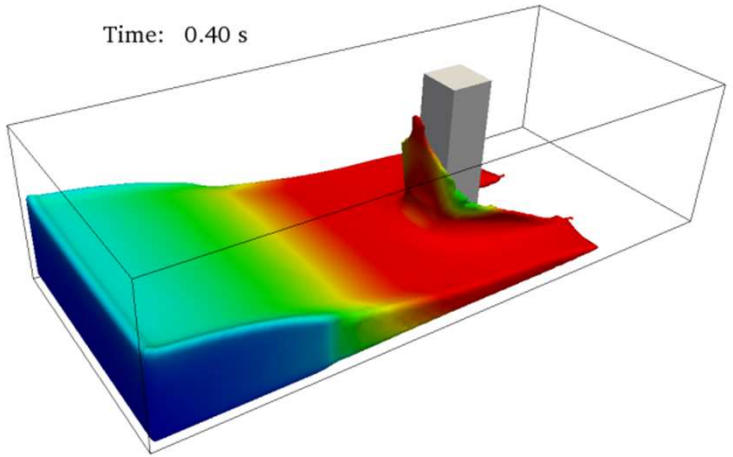
In November 2007, the LLNL Blue Gene/L remained at the number one spot as the world's fastest supercomputer: 478 TFLOPS

A Blue Gene/P supercomputer



One Blue Gene/L node board

SPH runtimes



Example: Dam break evolution during 1.5s

- using 30,000 particles takes 12 mins,

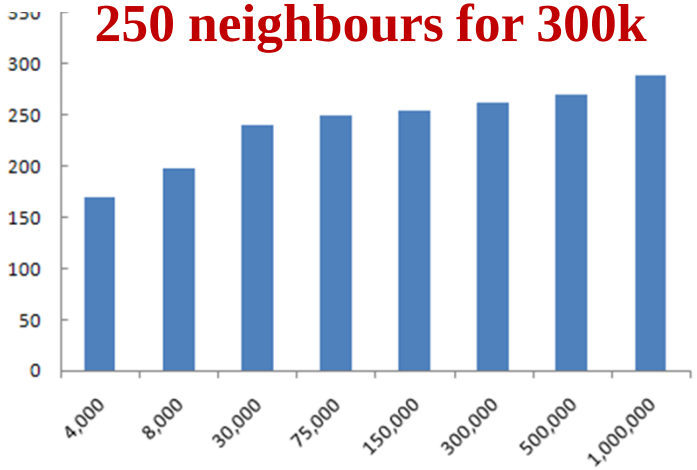
if no meaninful information

- using 300,000 particles takes 9.2 hours on a single-core machine

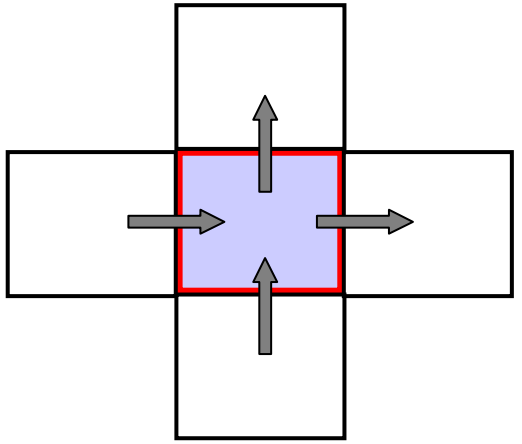
Why SPH is so expensive in time??

- small Δt with weakly compressible SPH scheme
  $O(10^{-6}\text{-}10^{-5})$ with 300,000 particles and more than 16,000 steps

- high number of neighbours/interactions per particle
  250 neighbours for 300k



In 2-D FVM only 4 neighbouring cells



Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

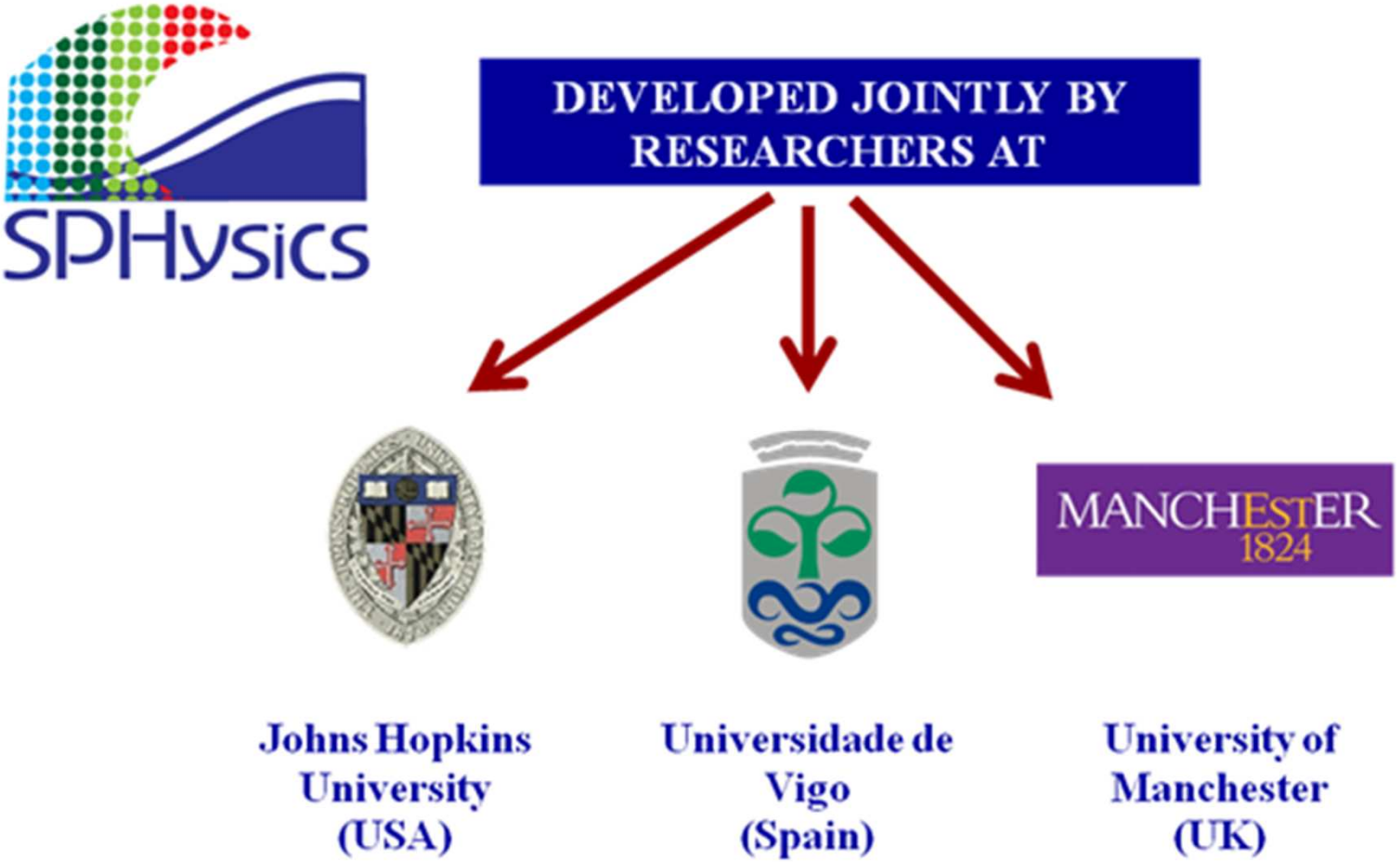
- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

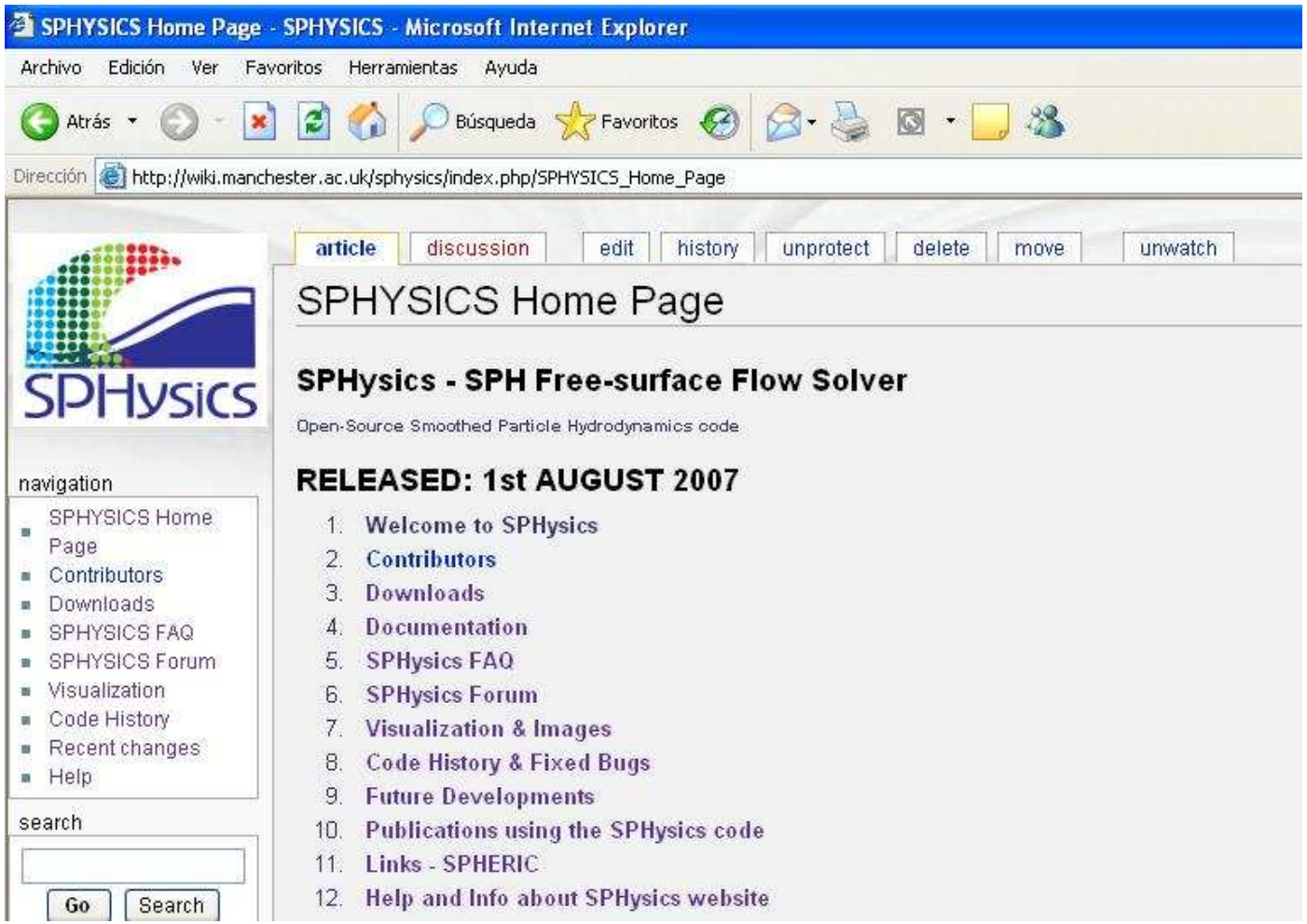
SPHysics project



SPHysics is a Smoothed Particle Hydrodynamics code primarily to study free-surface flow phenomena. It has been jointly developed by Johns Hopkins University (U.S.A.), the University of Vigo (Spain) and the University of Manchester (United Kingdom).

SPHysics project

As result of this research, a first serial code was developed in FORTRAN

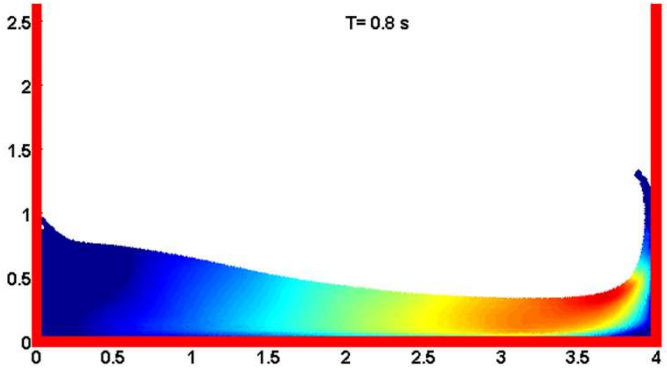


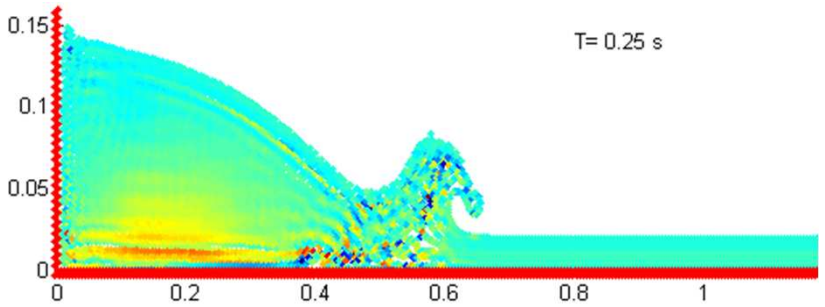
http://sphysics.org

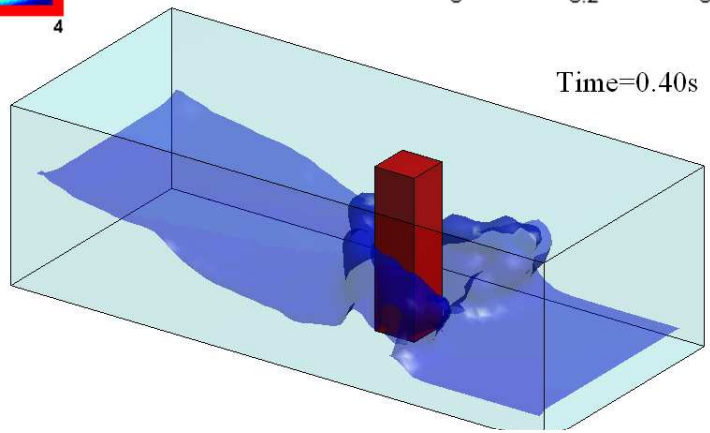
> 20,000 downloads !!!

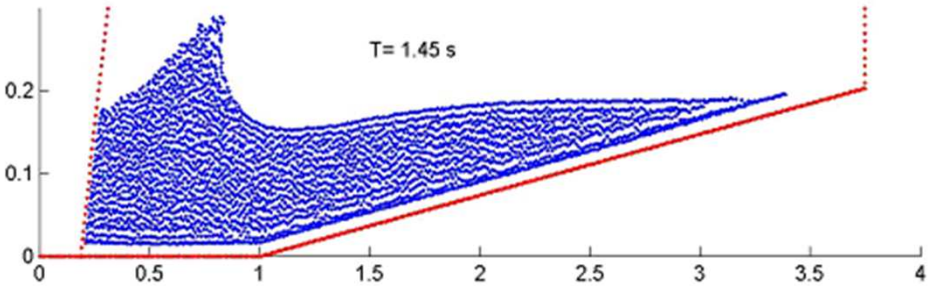
SPHysics project

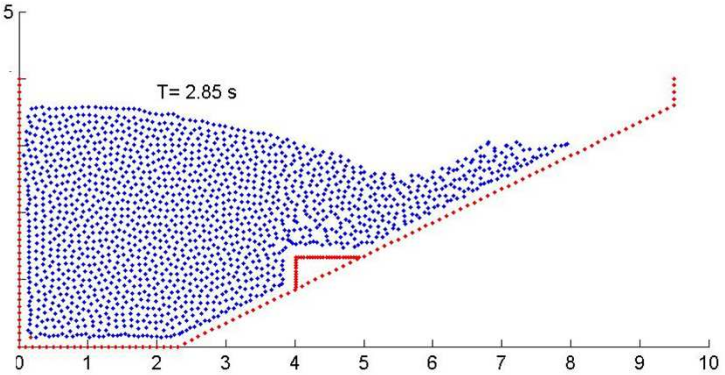
The SPHysics group has focused its research mainly on wave propagation and interaction with coastal structures, in 2D and 3D.



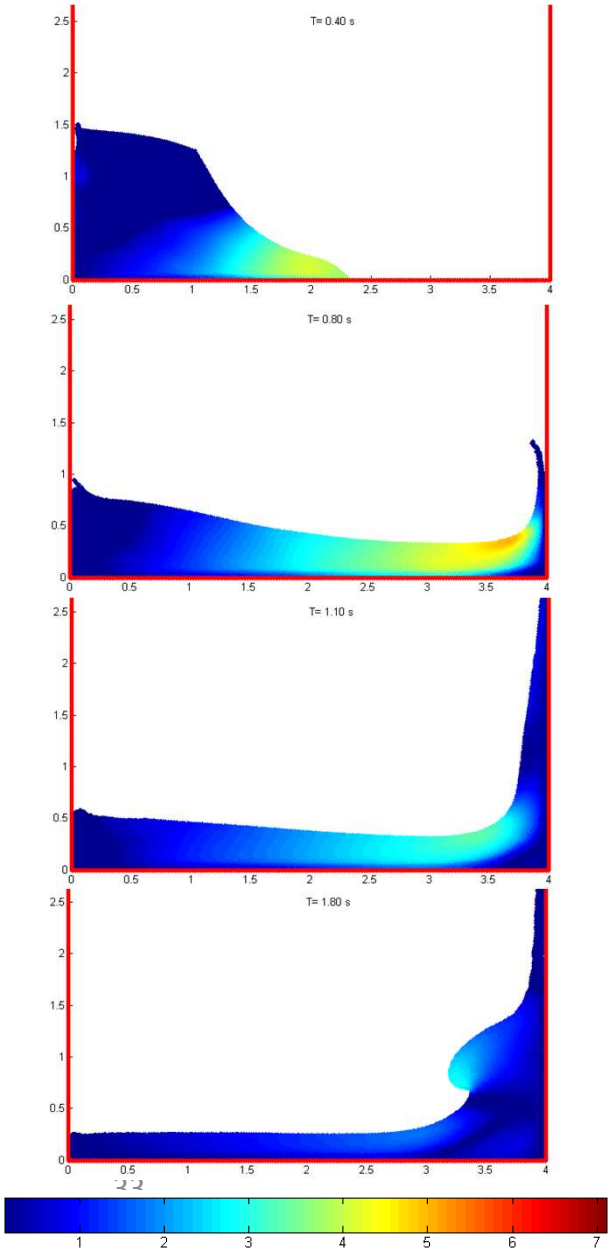


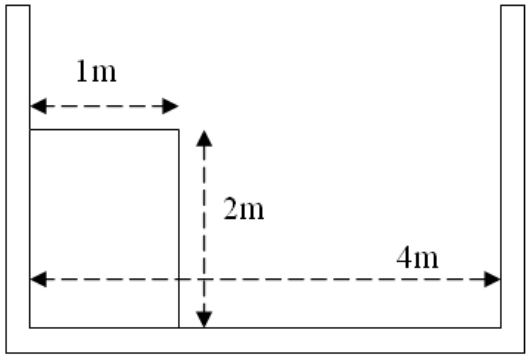




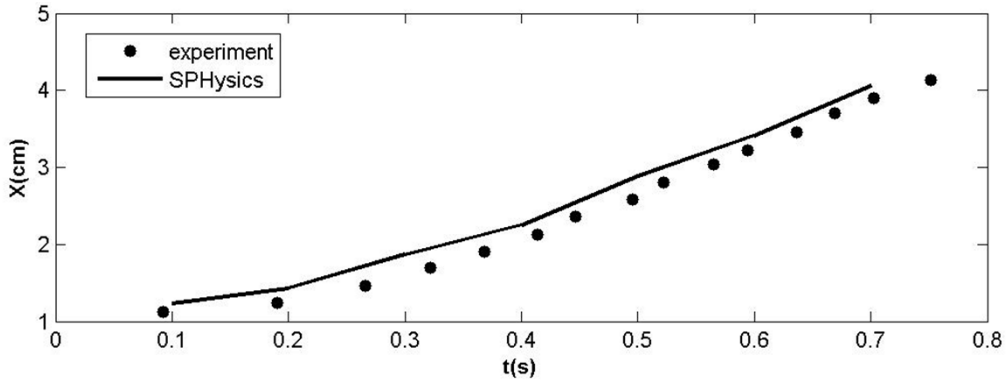


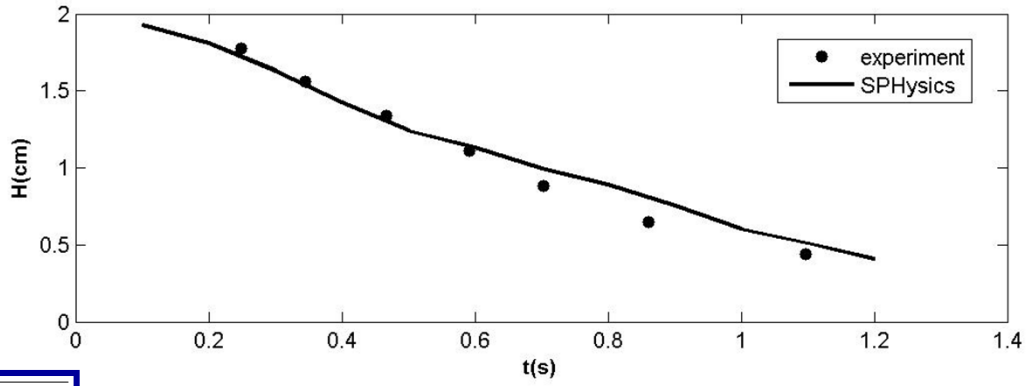
VALIDATION: Dynamic boundary conditions





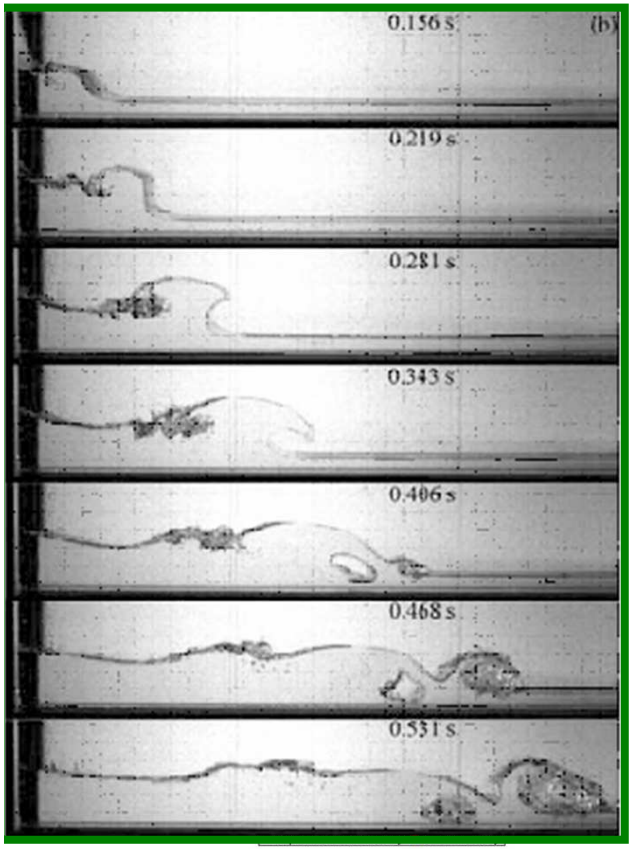
Koshizuka and Oka, 1996

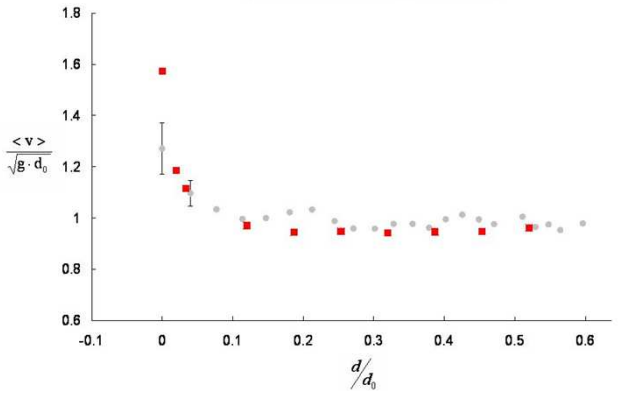


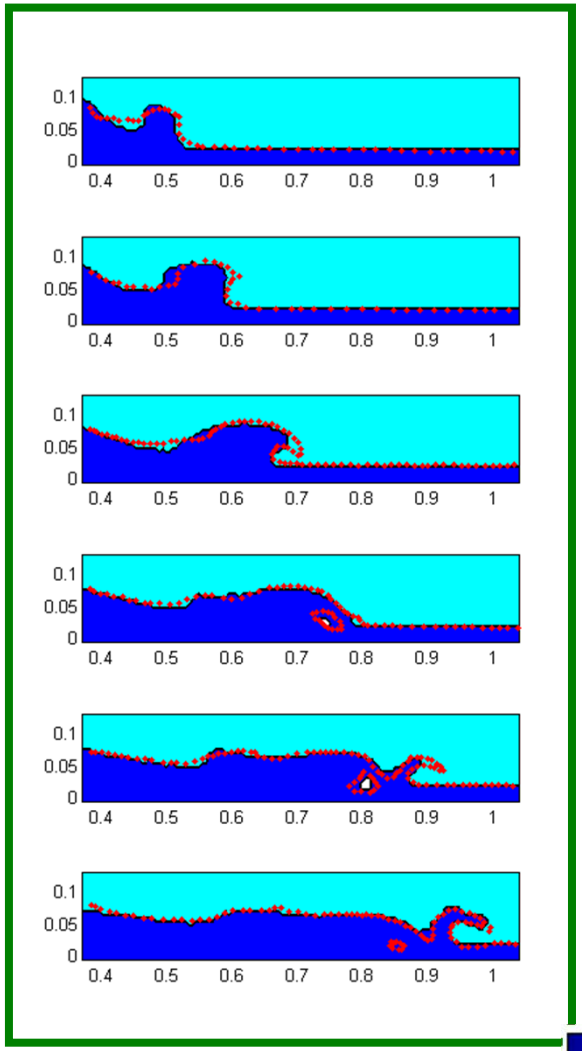


$$v = \sqrt{v_x^2 + v_z^2}$$

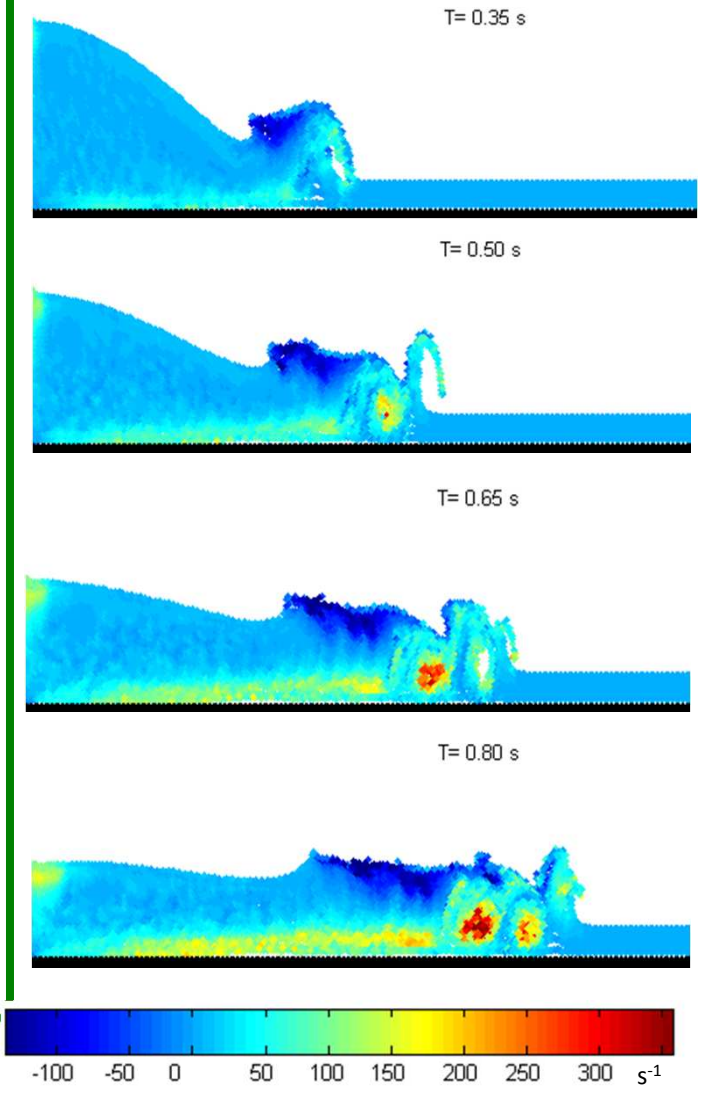
VALIDATION: 2D Dam break behavior



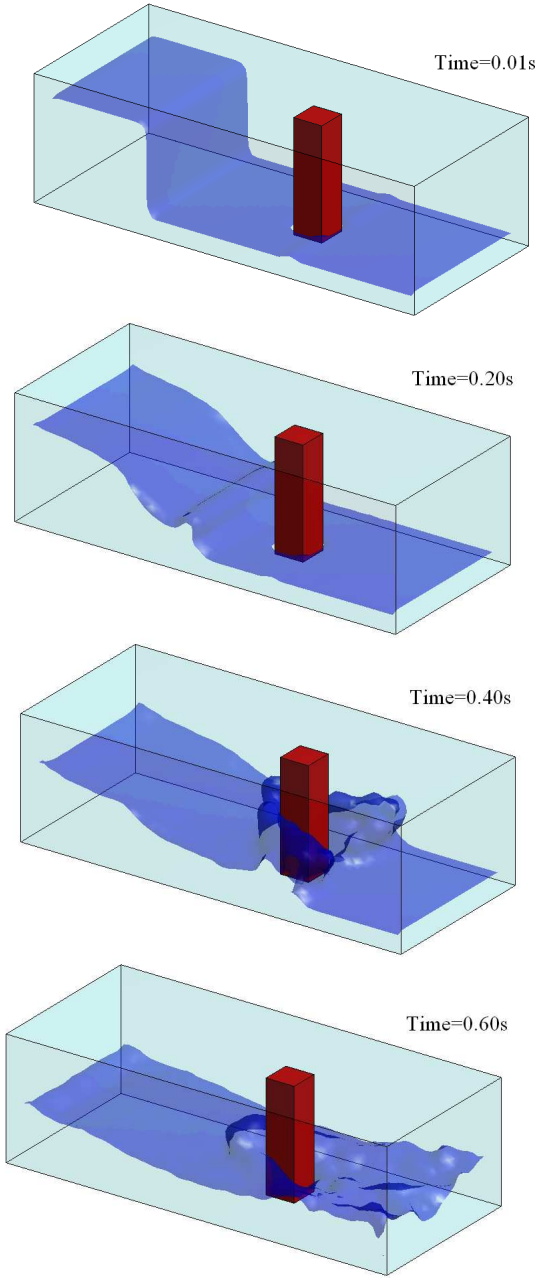


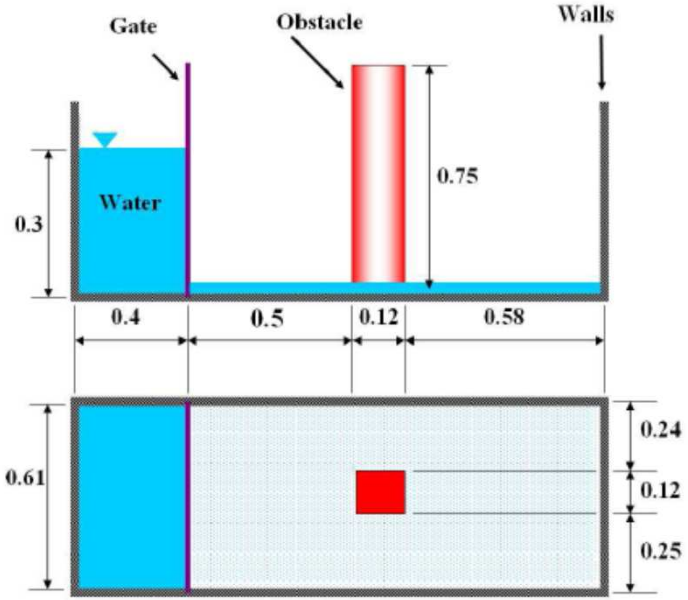


Janosi *et al.*, 2004

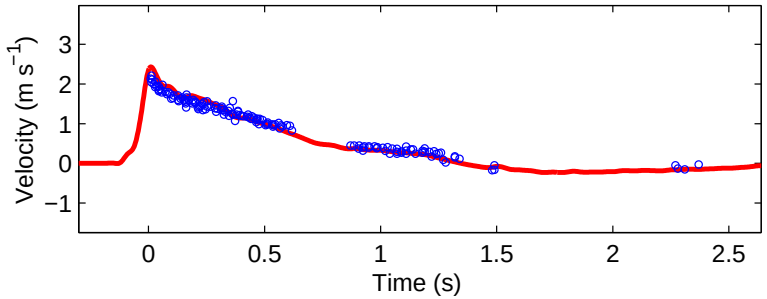


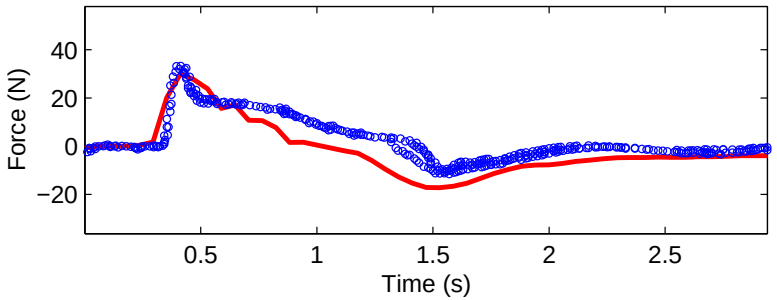
VALIDATION: 3D Wave-structure interaction





Yeh and Petroff experiment





SPHysics project

However, the SPHysics presents a high computational cost.
To perform simulations in a reasonable runtime with domains as large as the real systems, we need to **develop implementations that can exploit all the parallelism of the current hardware systems**.

FORTRAN

C++

OpenMP

CUDA

Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- **How to accelerate SPH**

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

Proposals to accelerate SPH

C++

OpenMP

CUDA

Proposals to accelerate SPH

How to accelerate your SPH code with less than ….

250 EUROS

Intel® Core ™ i7 940 at 2.93GHz with 4 cores

Multi-core
OpenMP

450 EUROS

GTX 480 at 1.40GHz with 480 cores

GPU
CUDA

5,000 EUROS

2 Intel Xeon E5620 + 4 GTX480

Multi-GPU
CUDA + MPI

Proposals to accelerate SPH

How to accelerate your SPH code with less than ….

250 EUROS

Intel® Core ™ i7 940 at 2.93GHz with 4 cores

Multi-core
OpenMP
4.5x

450 EUROS

GTX 480 at 1.40GHz with 480 cores

GPU
CUDA
55x

5,000 EUROS

2 Intel Xeon E5620 + 4 GTX480

Multi-GPU
CUDA + MPI
113x

Proposals to accelerate SPH

How to accelerate your SPH code with less than ….

55x not only means

that a simulation of 55 mins can be performed in 1 min

but also large simulations can be performed in a reasonable computational runtime

and different tests can be performed in a short time

Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

Multi-core implementation

Programming languages: OpenMP

Implementation techniques and optimizations

Available hardware: Multi-core CPUs

Results and speedups

Multi-core implementation

OpenMP is used to implement the multi-core SPH code.

ADVANTAGES:

OpenMP is a **portable and flexible** programming model.

Its implementation is straightforward and **no significant changes** in comparison to the single-core code are required.

The time dedicated to communication between different execution threads is reduced since the **same shared memory is used**.

DISADVANTAGES:

Using OpenMP on its own means that this parallelization and potential speedup are **limited to a small number of cores** (i.e. the number of cores existing on the compute node).

Multi-core implementation

Programming languages: OpenMP

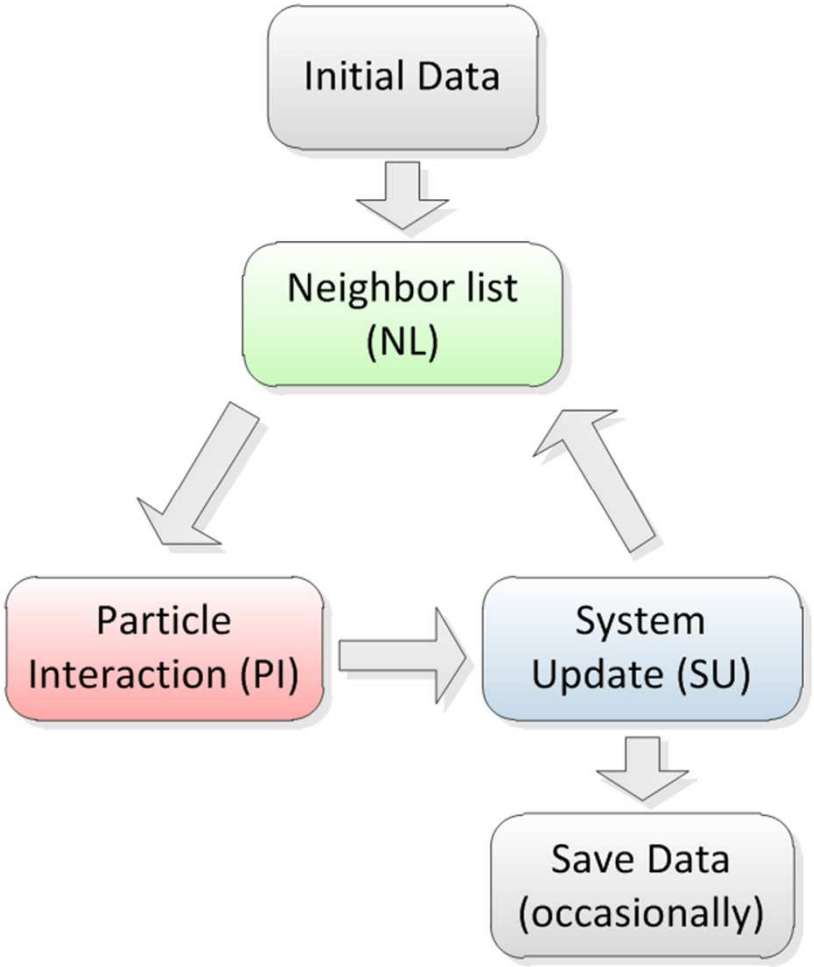
Implementation techniques and optimizations

Available hardware: Multi-core CPUs

Results and speedups

Multi-core implementation

Implementation techniques and optimizations



Conceptual diagram of the CPU implementation of a SPH code

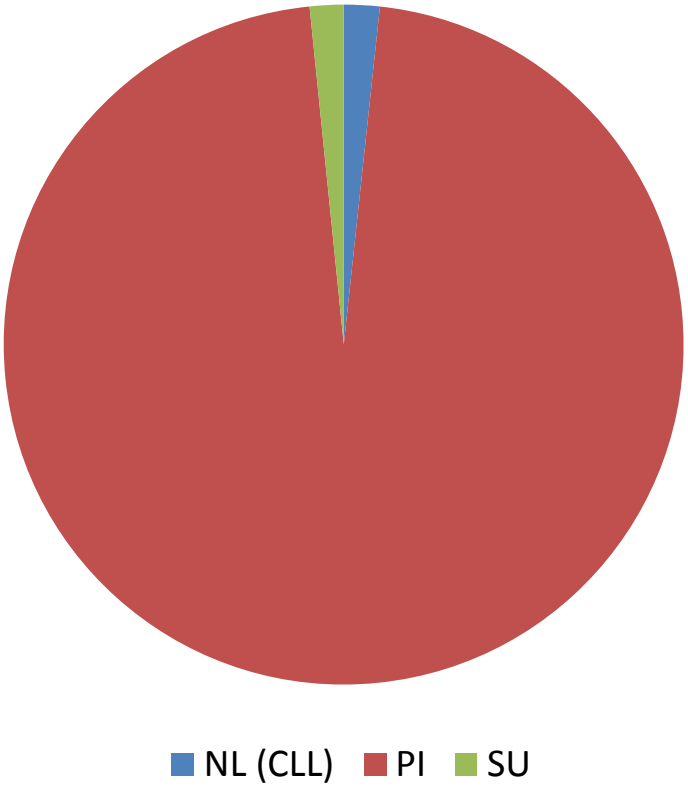
Multi-core implementation

Parallelization of… ???????

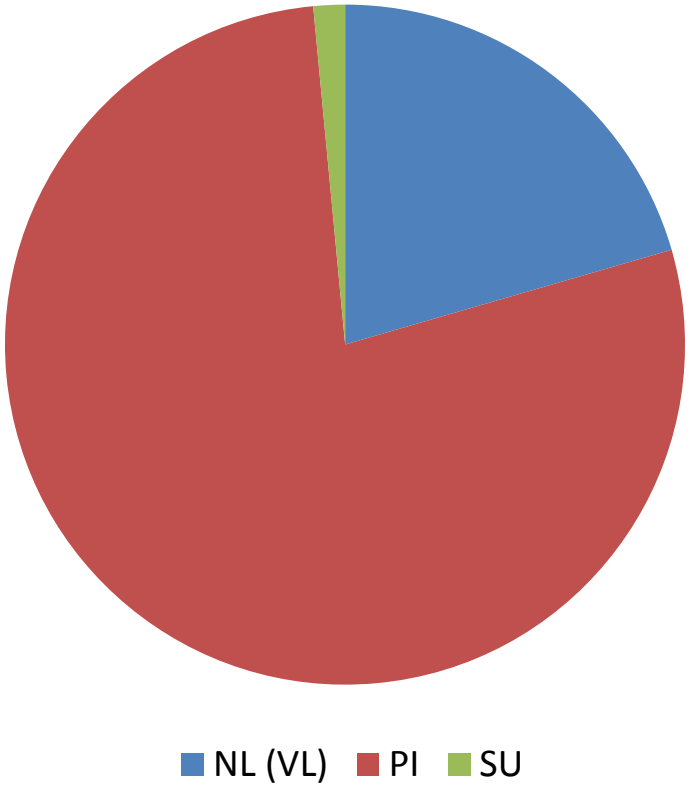
In the case of a dam-break simulation:

Dominguez et al. “Neighbour lists in Smoothed Particle Hydrodynamics”. IJNMF, 2010.

Cell linked list



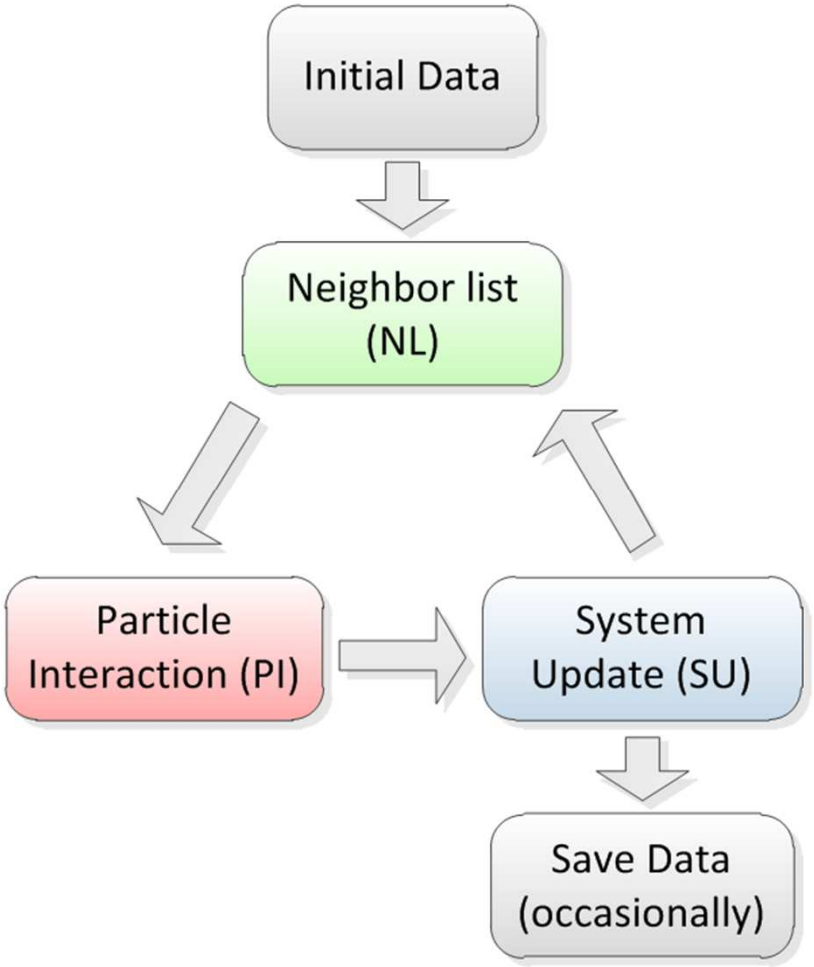
Verlet list



Force computation is the most expensive step of SPH in terms of computational runtime. This is a key process that must be implemented in parallel in order to accelerate SPH.

Multi-core implementation

Implementation techniques and optimizations



Conceptual diagram of the CPU implementation of a SPH code

Most of the sequential tasks and operations that involve a loop over all particles are performed using the different cores of the same CPU.

Several parts of the SPH code can be parallelized, but mainly, the force calculation since it is the most expensive part of the method.

Problems with this parallel programming;

- the concurrent access to the same memory positions for read-write giving rise to unexpected results.
- the load balancing to distribute equally the work among threads.

Multi-core implementation

Implementation techniques and optimizations

Symmetry in particle interaction

SSE instructions

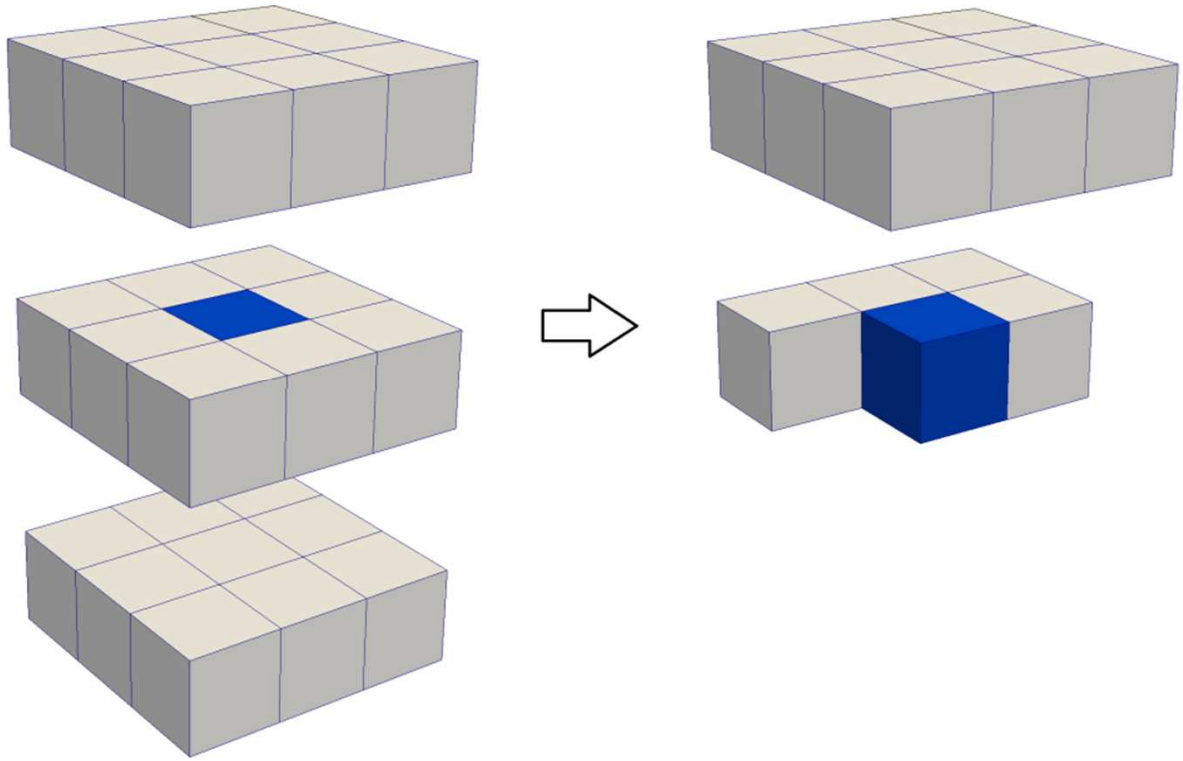
Dynamic load balancing

Multi-core implementation

Implementation techniques and optimizations

Symmetry in particle interaction:

The concurrent access to memory to write is avoided since each thread has its own memory space where the forces of each particle are accumulated.



In 3D, each cell interacts with 14 cells (right) instead of 27 (left).

Multi-core implementation

Implementation techniques and optimizations

SSE instructions

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

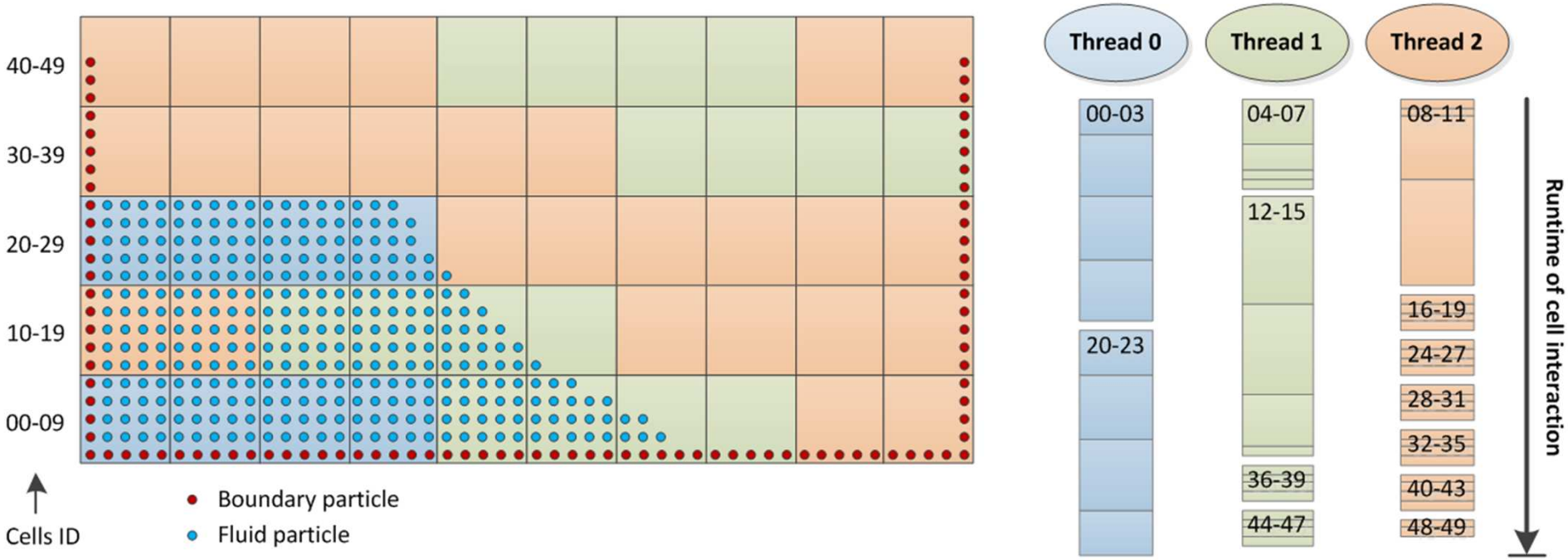
```
int npar=0;
int particlesi[4],particlesj[4];
for(int i=ibegin;i<iend;i++){
  for(int j=jbegin;j<jend;j++){
    if(Distance between particle[i] and particle[j] < 2h){
      particlesi[npar]=i; particlesj[npar]=j;
      npar++;
      if(npar==4){
        ComputeForcesSSE(particlesi,particlesj);
  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

Pseudocode in C++ of the force computation between particles of two cells without vectorial instructions (up) and grouping in blocks of 4 pair-wise of interaction using SSE instructions (down).

Multi-core implementation

Implementation techniques and optimizations

Dynamic load balancing



The dynamic scheduler of OpenMP is also employed distributing cells in blocks of 10 among different threads.

Multi-core implementation

Programming languages: OpenMP

Implementation techniques and optimizations

Available hardware: Multi-core CPUs

Results and speedups

Multi-core implementation

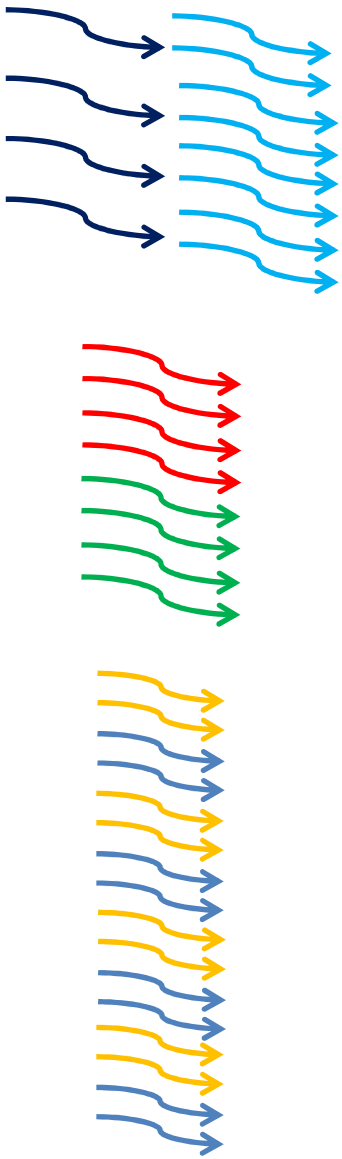
Available hardware

Desktop	**Intel® Core ™ i7 940 at 2.93GHz with 4 cores with Hyper-threading**		**250 EUROS**
Workstation	**2 x Intel Xeon X5500 at 2.67 GHz with 2x4 cores**		**900 EUROS**
Supercomputing Center	**8 x Intel dual-core Itanium Montvale at 1.6 GHz with 16 cores**	CESGA	

Multi-core implementation

Available hardware

		# threads
Desktop	**Intel® Core ™ i7 940 at 2.93GHz with 4 cores with Hyper-threading**	**8 logical threads with Hyper-threading**
Workstation	**2 x Intel Xeon X5500 at 2.67 GHz with 2x4 cores**	**8 threads**
Supercomputing Center	**8 x Intel dual-core Itanium Montvale at 1.6 GHz with 16 cores**	**16 threads**



Multi-core implementation

Programming languages: OpenMP

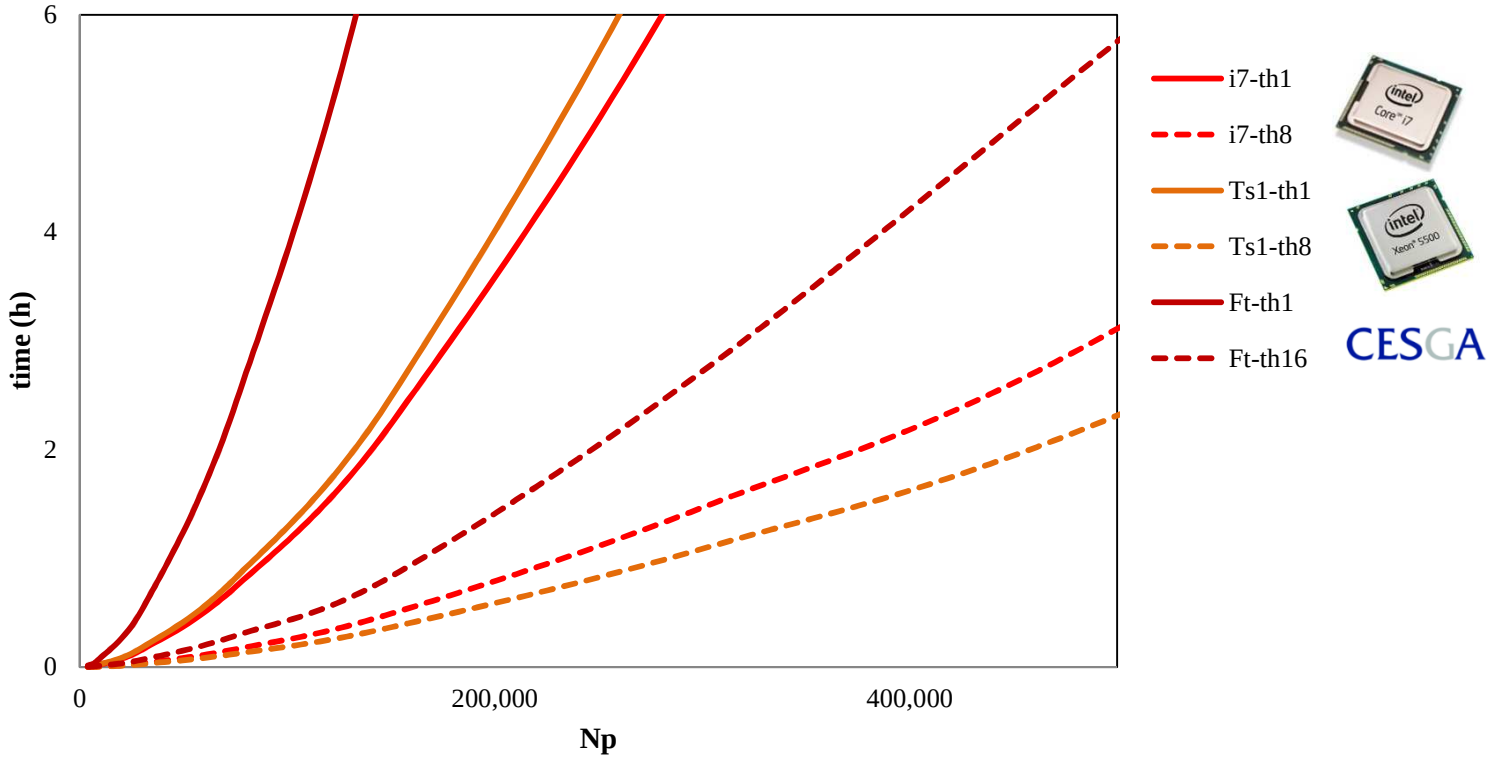
Implementation techniques and optimizations

Available hardware: Multi-core CPUs

Results and speedups

Multi-core implementation

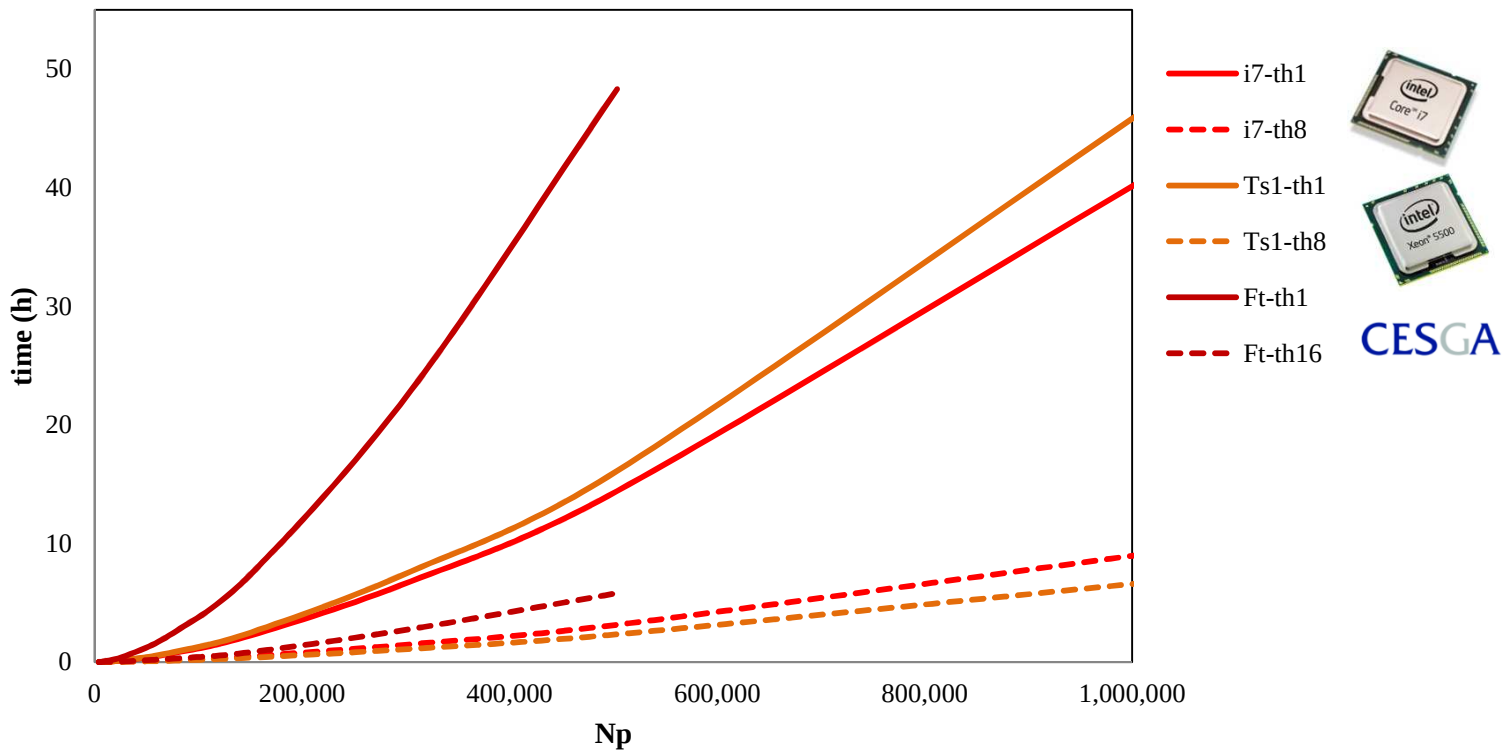
CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
2 x Intel Xeon X5500 at 2.67 GHz with 2x4 cores
8 x Intel dual-core Itanium Montvale at 1.6 GHz with 16 cores



Computational runtimes with the Multi-core CPU model

Multi-core implementation

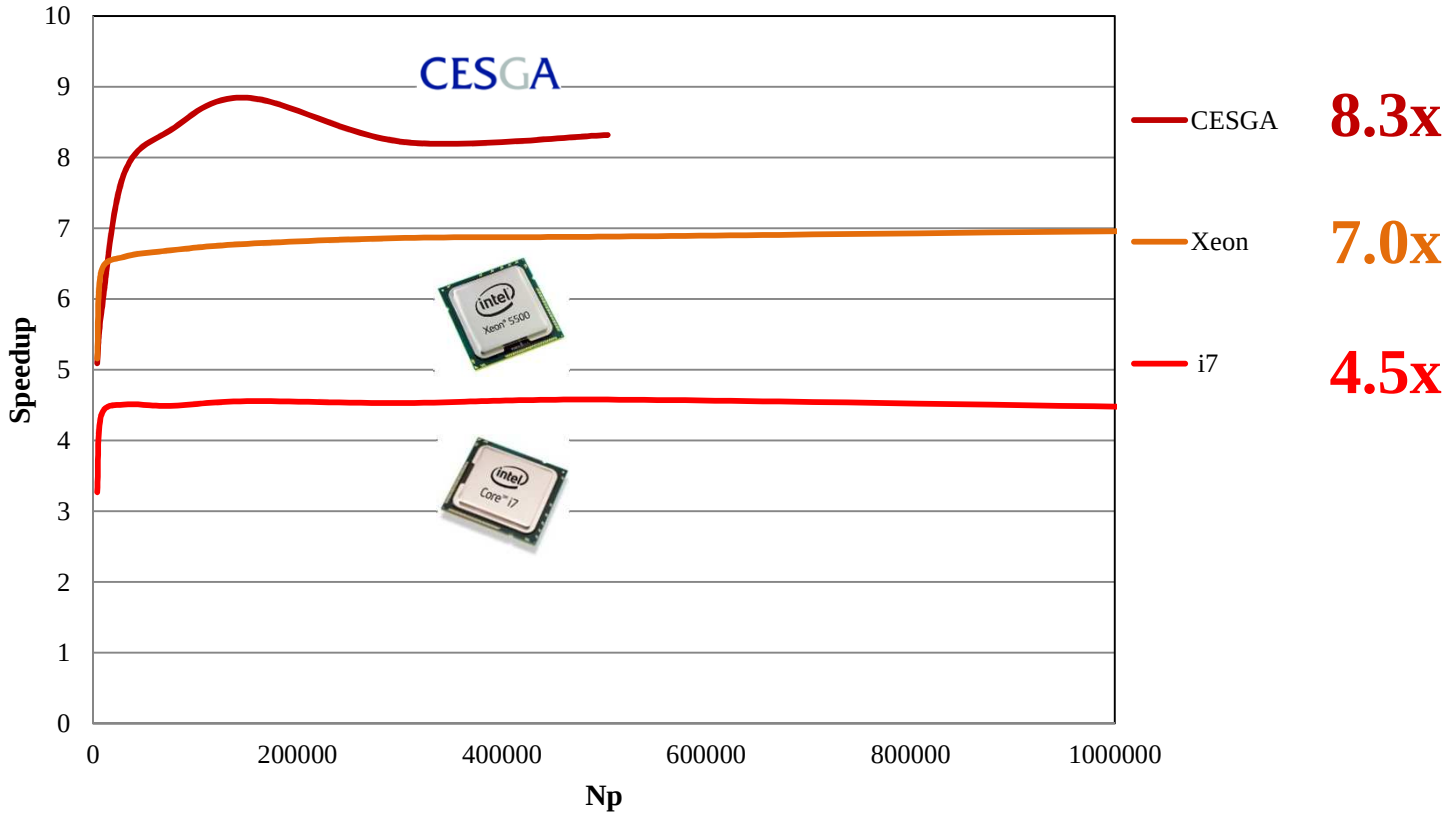
CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
2 x Intel Xeon X5500 at 2.67 GHz with 2x4 cores
8 x Intel dual-core Itanium Montvale at 1.6 GHz with 16 cores



Computational runtimes with the Multi-core CPU model

Multi-core implementation

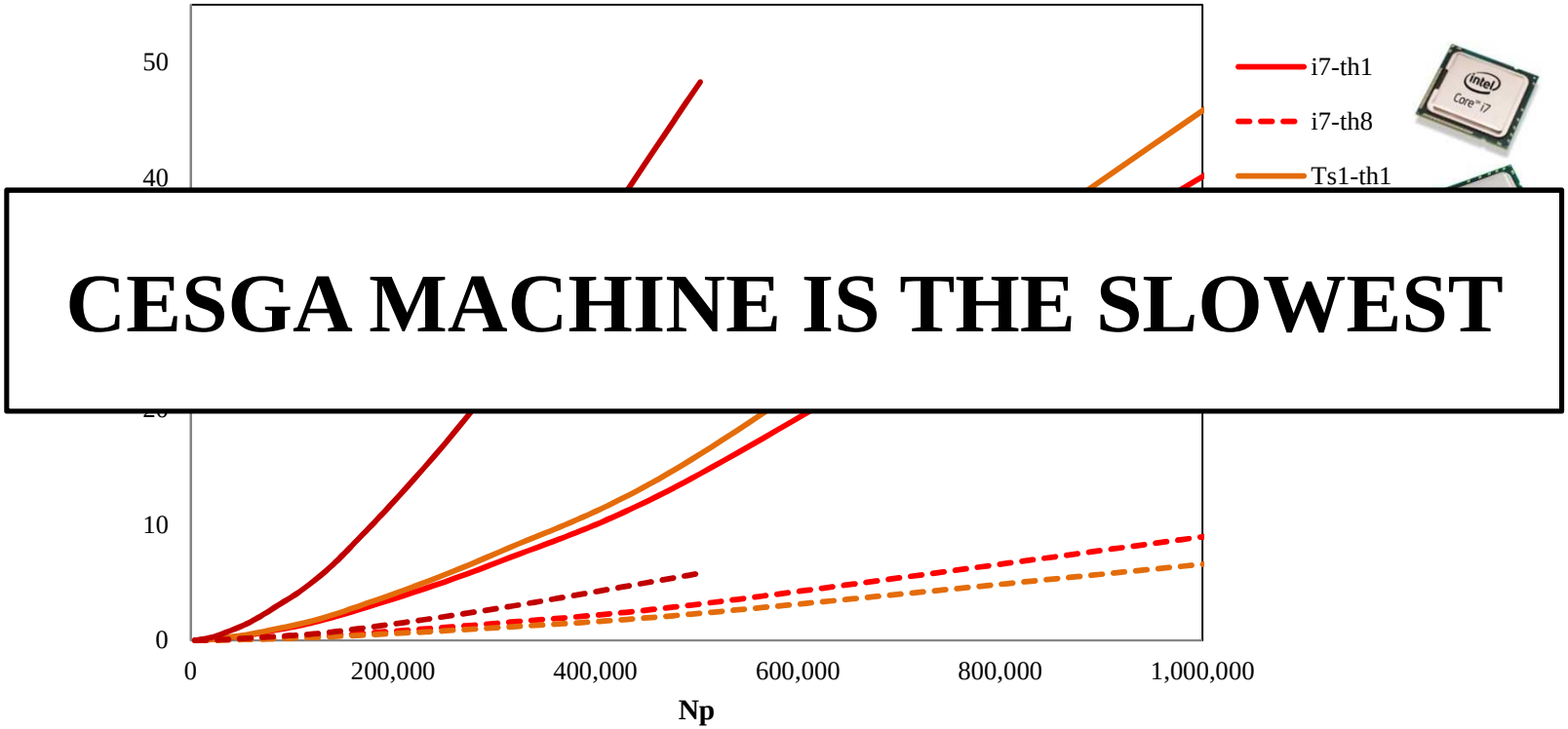
CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
2 x Intel Xeon X5500 at 2.67 GHz with 2x4 cores
8 x Intel dual-core Itanium Montvale at 1.6 GHz with 16 cores



Computational runtimes with the Multi-core CPU model

Multi-core implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
2 x Intel Xeon X5500 at 2.67 GHz with 2x4 cores
8 x Intel dual-core Itanium Montvale at 1.6 GHz with 16 cores

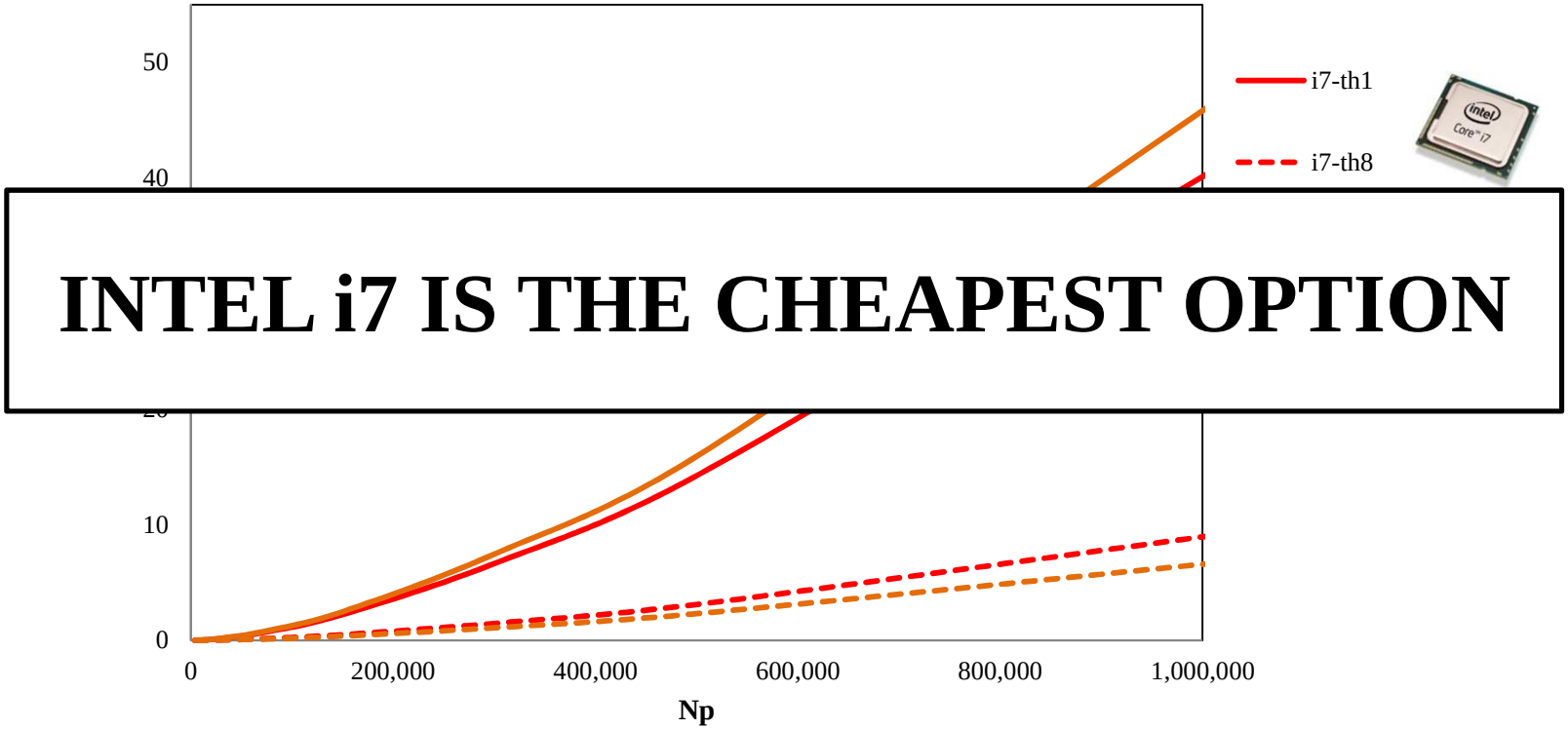


CESGA MACHINE IS THE SLOWEST

Computational runtimes with the Multi-core CPU model

Multi-core implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
2 x Intel Xeon X5500 at 2.67 GHz with 2x4 cores

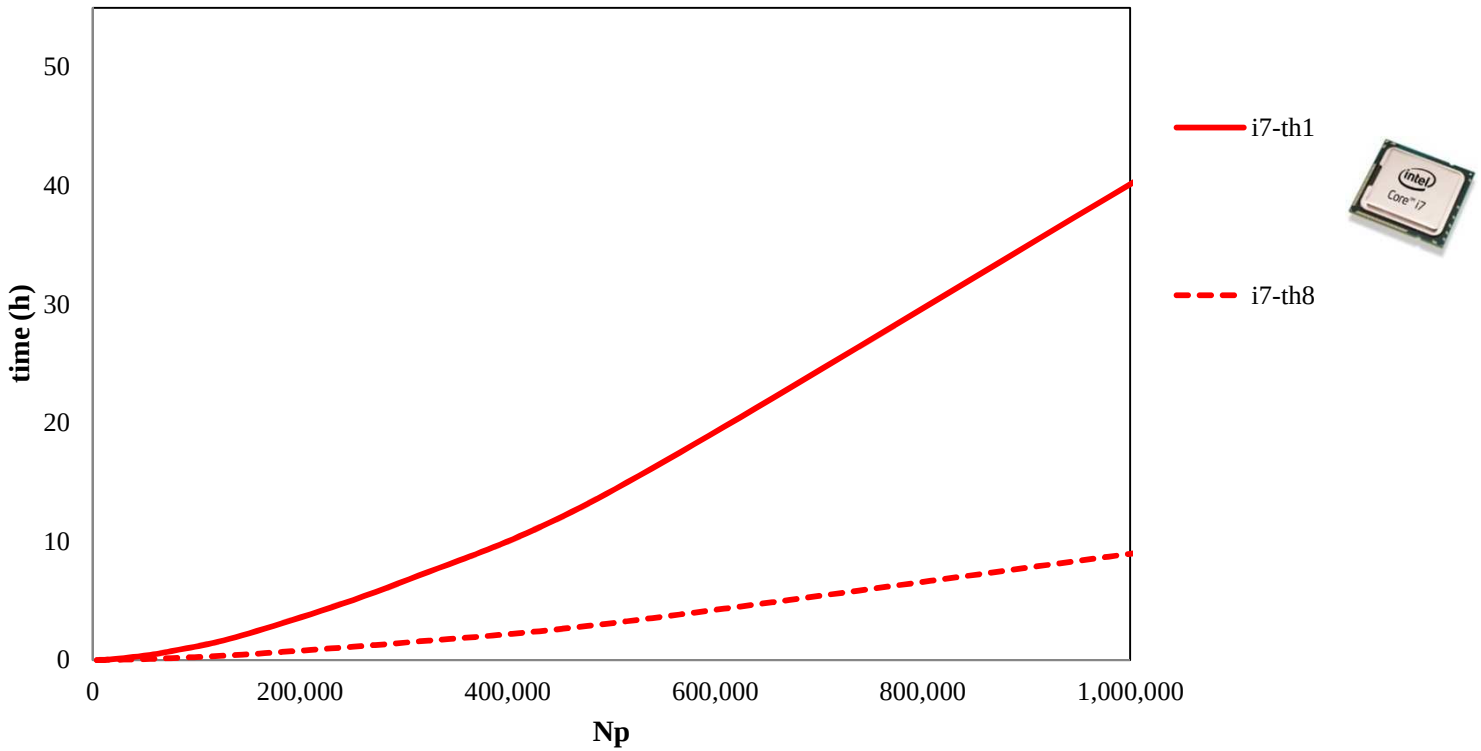


INTEL i7 IS THE CHEAPEST OPTION

Computational runtimes with the Multi-core CPU model

Multi-core implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores



Computational runtimes with the Multi-core CPU model

Multi-core implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores

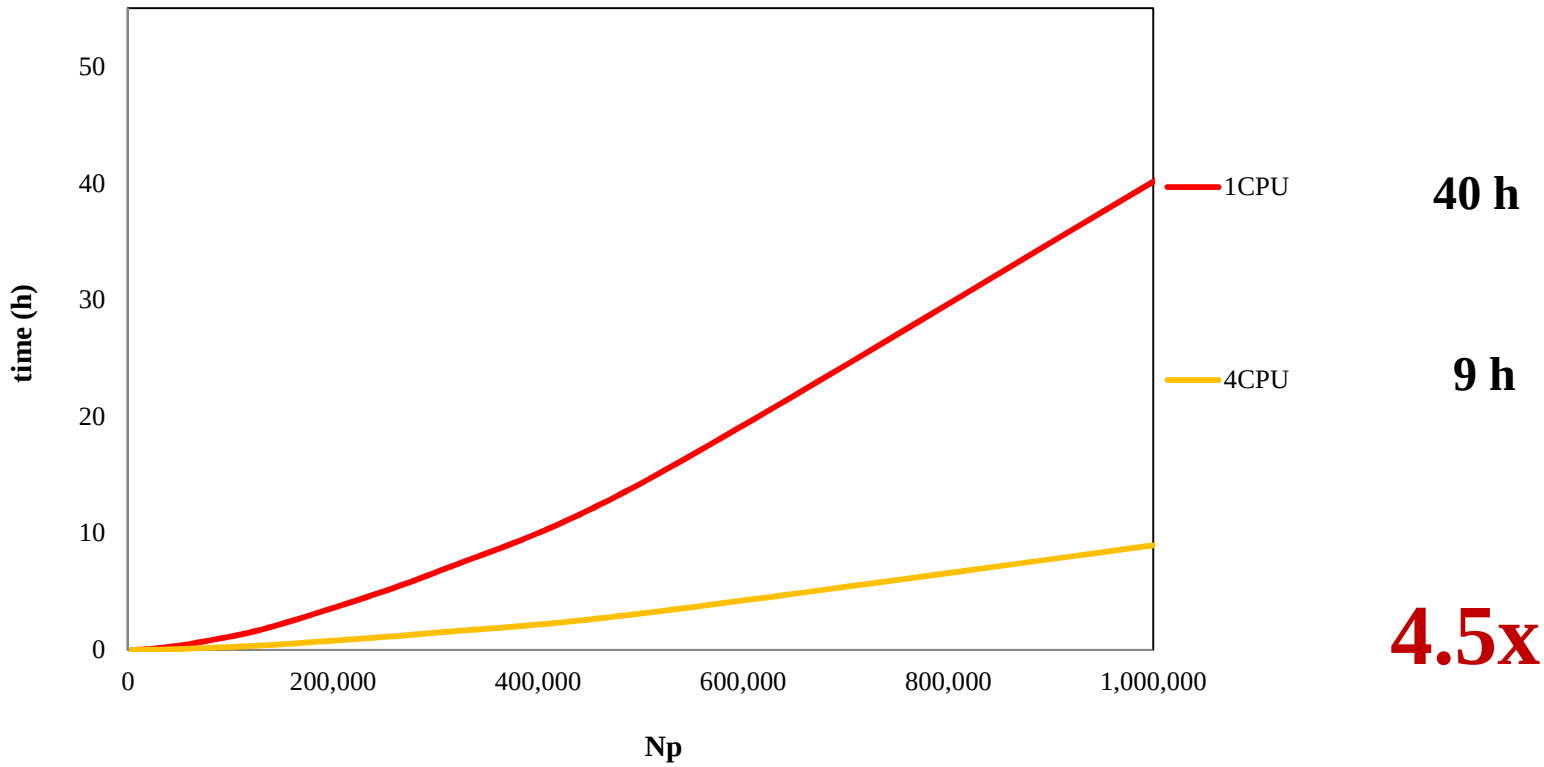
1M	hours	Speedup vs. 1 CPU	Speedup vs. 4 CPU
1CPU	40.71	**1.00**	
4CPU	9.09	4.48	**1.00**

250 EUROS

Speedup of using the Multi-core CPU model

Multi-core implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores



40 h

9 h

4.5x

Computational runtimes with the Multi-core CPU model

Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

GPU implementation

Introduction to GPUs and CUDA

Implementation techniques and optimizations

Available hardware: GPUs

Results and speedups

Graphics Processing Unit

•GPUs are a new technology imported from the computer **games industry.**

Videogame FIFA 12

Graphics Processing Unit

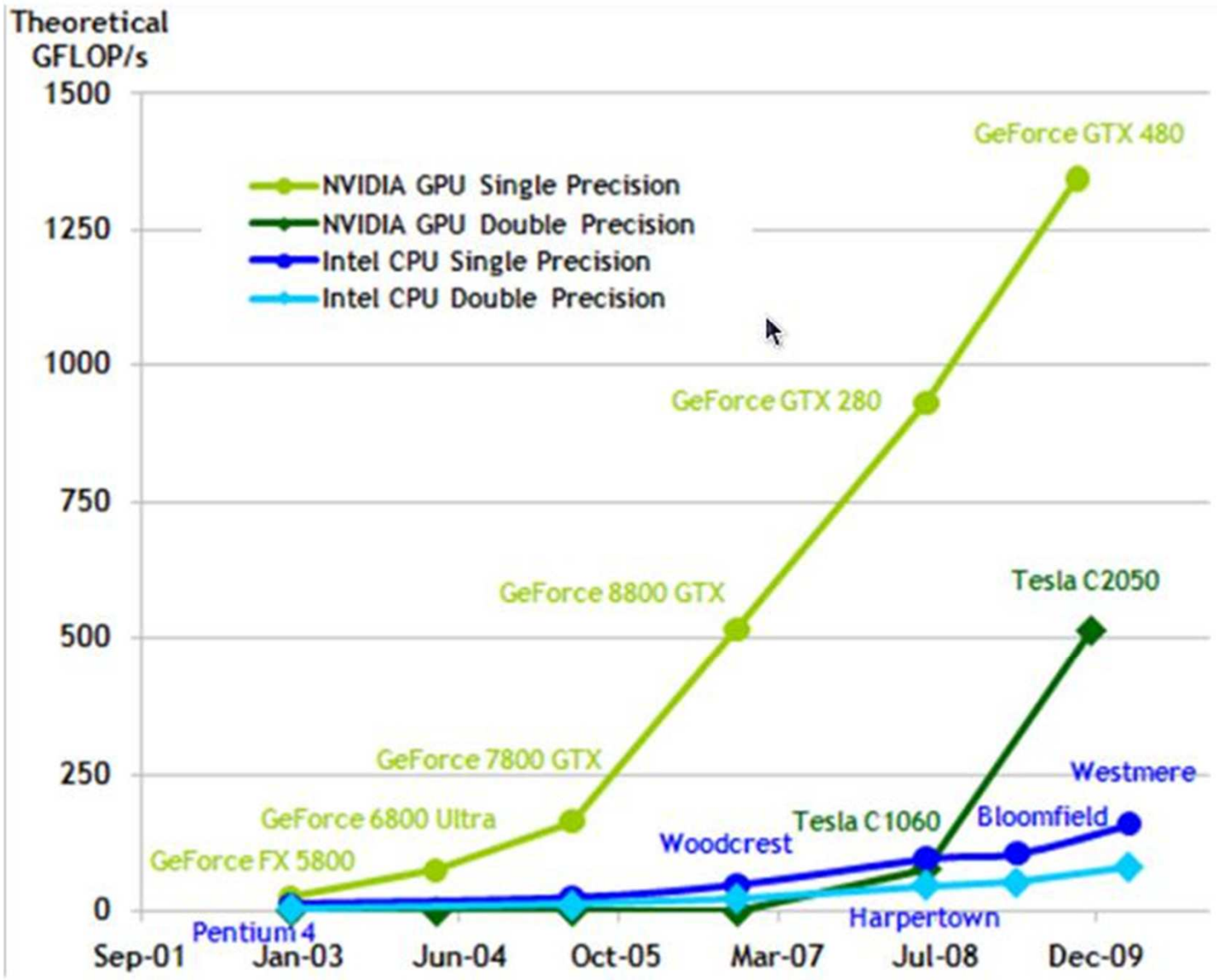
- GPUs are a new technology imported from the computer **games industry.**

- GPUs are designed to **treat large data flows.**

- Due to the development of the video games market and multimedia, their computing power **has increased much faster than CPUs**.

CPUs double their capacity each 18 months (x1.5 anual, x60 decade)

GPUs double their capacity each 12 months (x2.0 anual, x1000 decade)



Graphics Processing Unit

•GPUs are a new technology imported from the computer **games industry.**

•GPUs are designed to **treat large data flows.**

•Due to the development of the video games market and multimedia, their computing power **has increased much faster than CPUs**.

•GPUs are massively **multithreaded manycore chips**. For example, with a GTX480 card a maximum of 23,040 threads can be in execution simultaneously.

•GPUs appear to be an **accessible alternative to accelerate SPH** since they are **cheap** and **ease-of-maintenance** in comparison with large cluster machines.

•The GPU parallelisation technique uses the **CUDA developed by NVIDIA**.

•HOWEVER, AN EFFICIENT AND FULL USE OF THE CAPABILITIES OF THE GPU IS NOT STRAIGHTFORWARD.

Graphics Processing Unit

General ideas about CUDA:

Kernel: code to be executed on GPU

Thread: execution task

Block: group of threads that executes the kernel

Grid: array of blocks that executes the same kernel

Warp: active blocks assigned to a multiprocessor

are executed in groups of 32 tasks-threads

Graphics Processing Unit

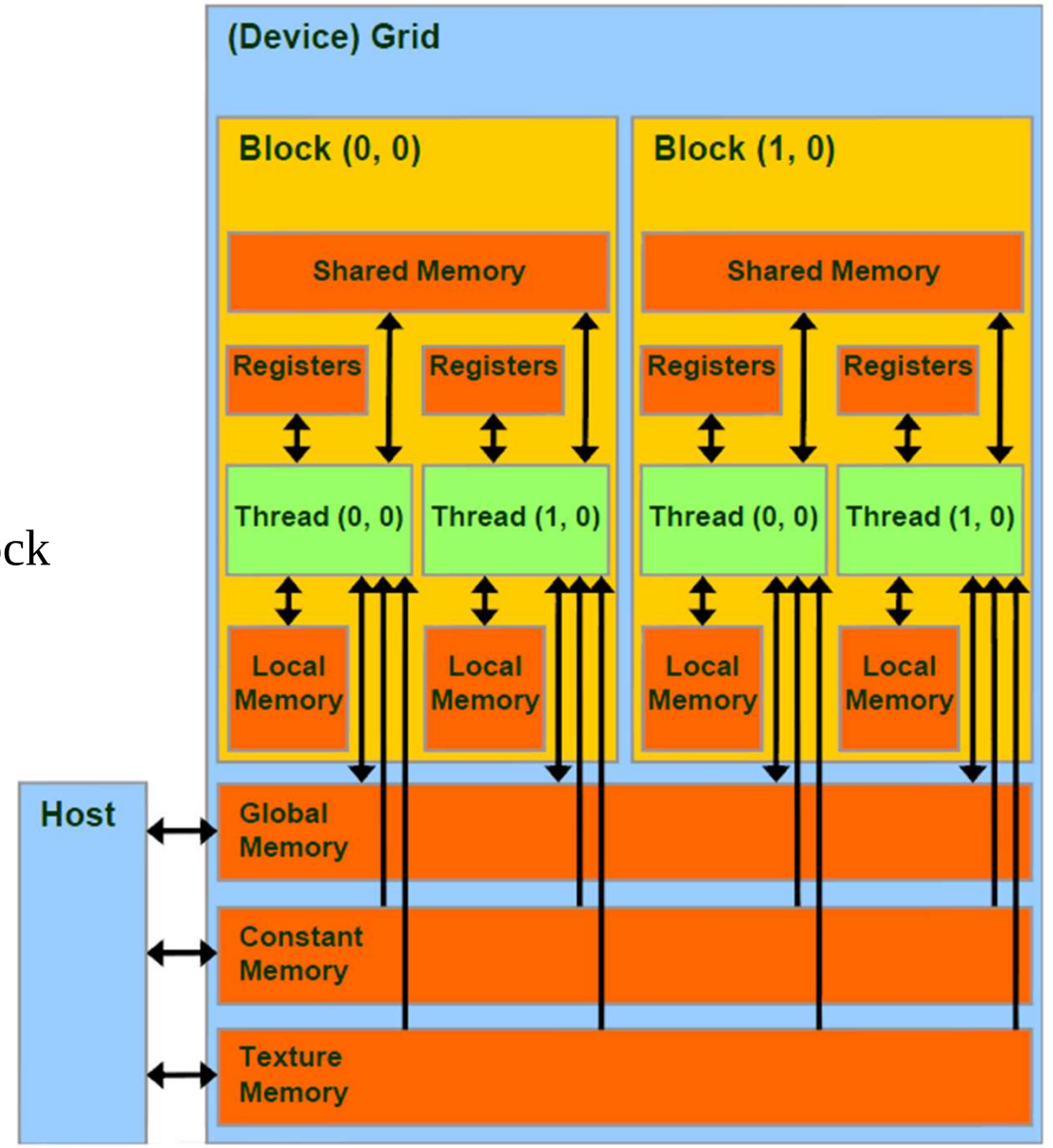
Memory Architecture

Thread: private *local memory*

Block: *shared memory*
visible to all threads of the block

All threads have access to the same *global memory*

2 read-only memory spaces:
the *constant memory*
and *texture memory* espaces



GPU implementation

Introduction to GPUs and CUDA

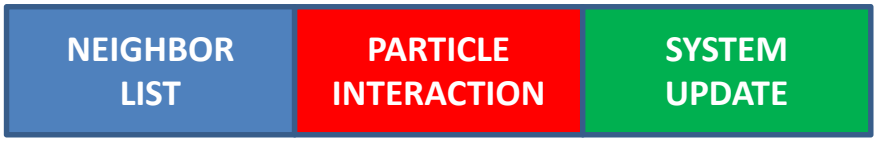
Implementation techniques and optimizations

Available hardware: GPUs

Results and speedups

GPU implementation

- Conceptually, an SPH code is an iterative process consisting of three main steps:



-***neighbour list***:
particles only interact with surrounding particles located at a given distance so the domain is divided in cells of the kernel size to reduce the neighbour search to the adjacent cells;

-***particle interaction*:**
each particle only looks for neighbours at the adjacent cells, after verifying that the distance between particles lies within the support of the kernel, the conservation laws of continuum fluid dynamics are computed for the pair-wise interaction of particles;

-***system update***:
once the forces between neighbouring particles have been evaluated, all physical magnitudes of the particles are updated at the next time step.

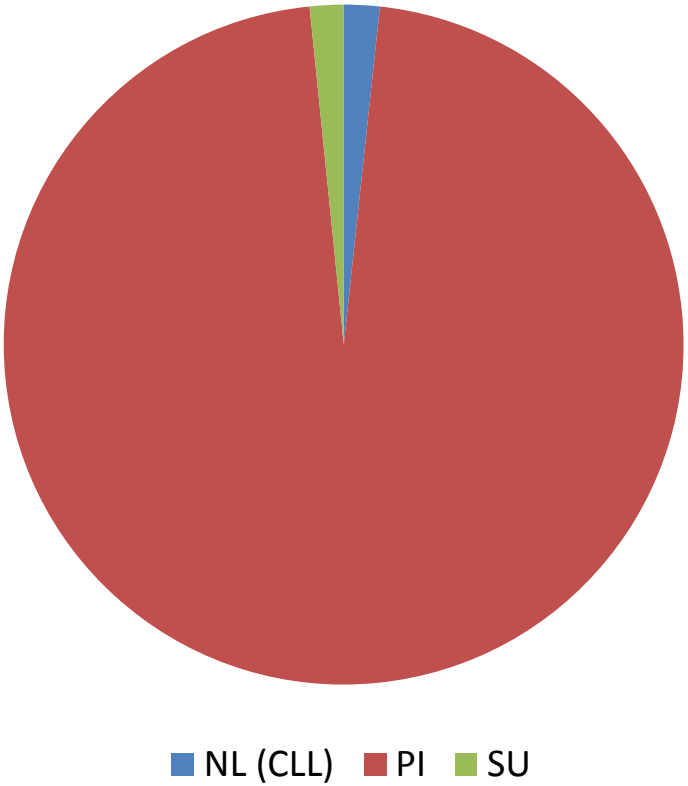
GPU implementation

GPU implementation of… ???????

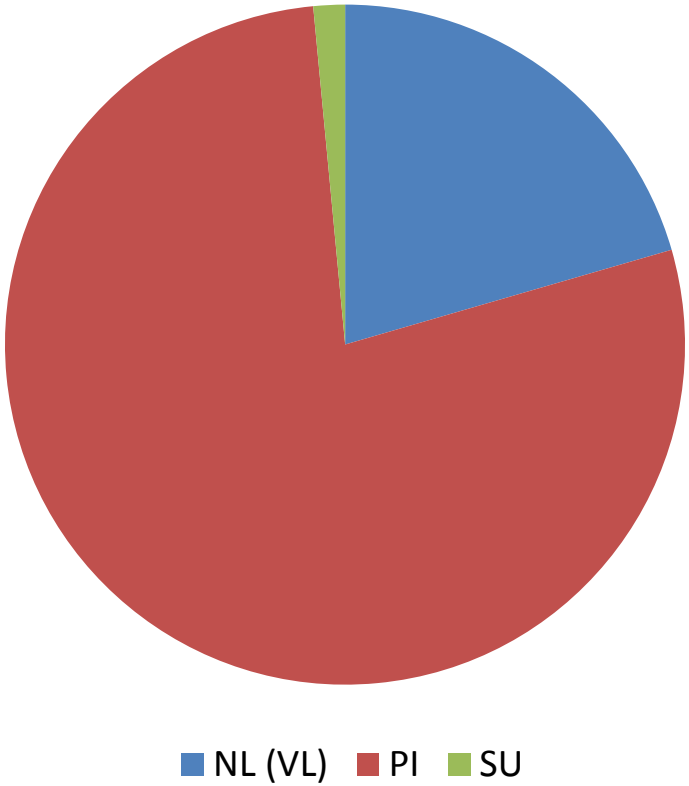
In the case of a dam-break simulation:

Dominguez et al. “Neighbour lists in Smoothed Particle Hydrodynamics”. IJNMF, 2010.

Cell linked list

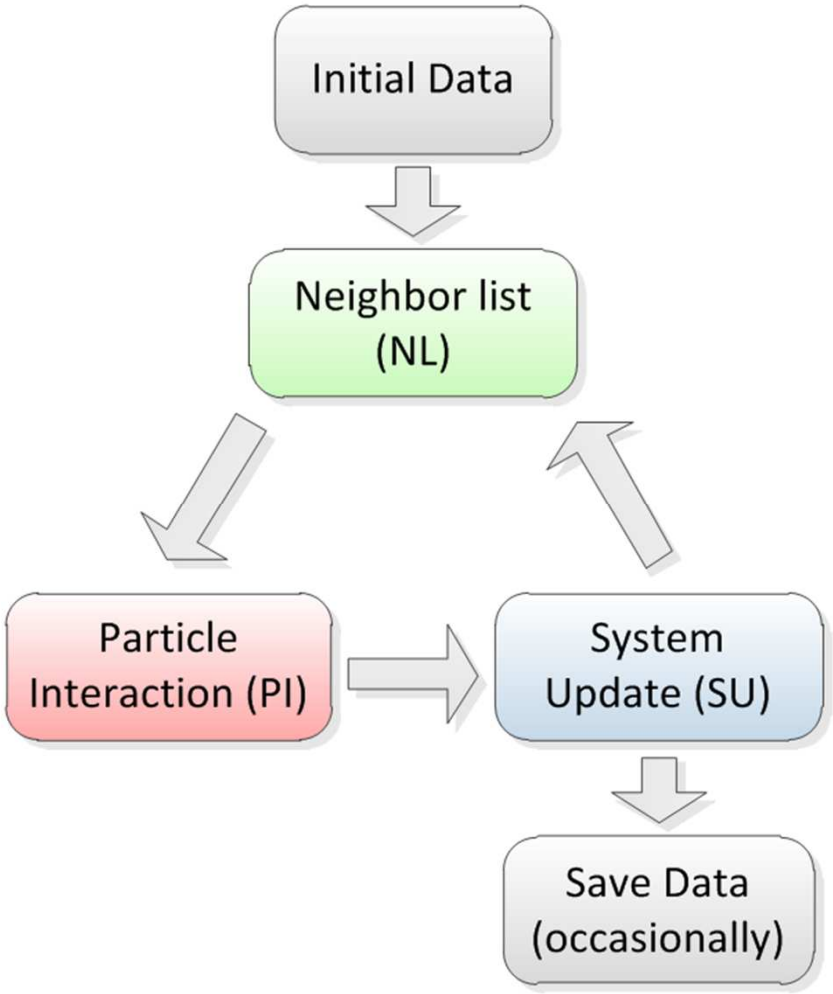


Verlet list

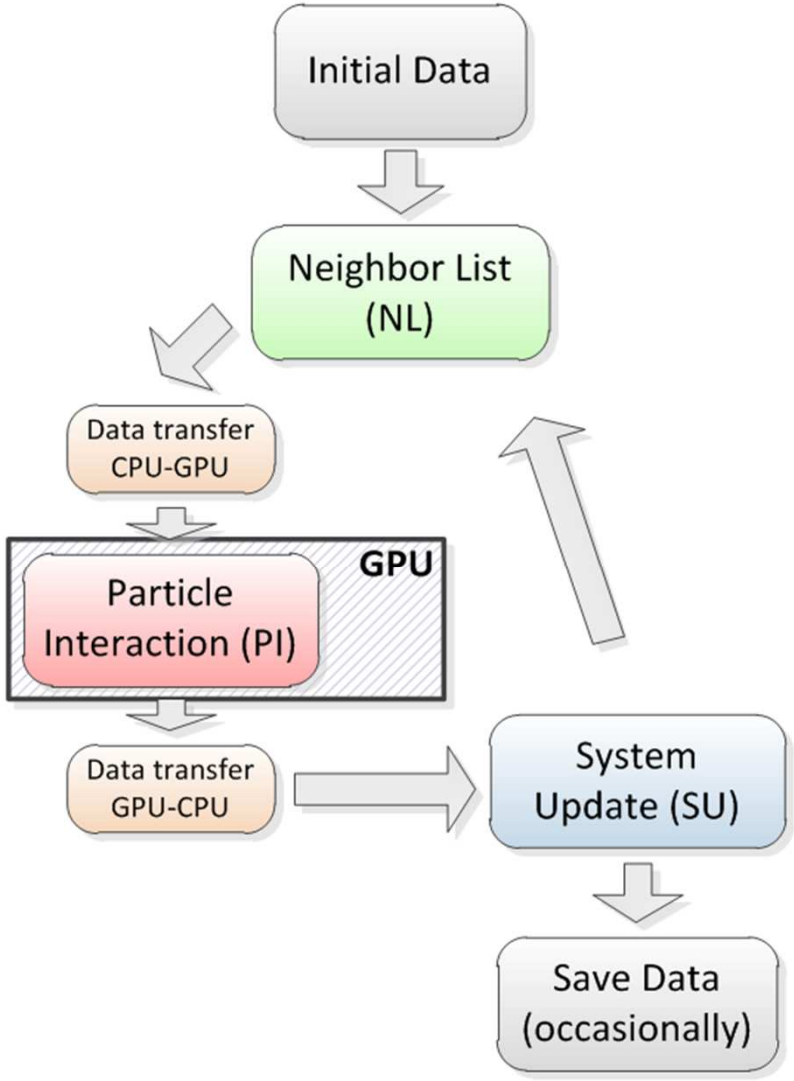


Force computation is the most expensive step of SPH in terms of computational runtime. This is a key process that must be implemented in parallel in order to accelerate SPH.

GPU implementation



Conceptual diagram of the CPU implementation of a SPH code

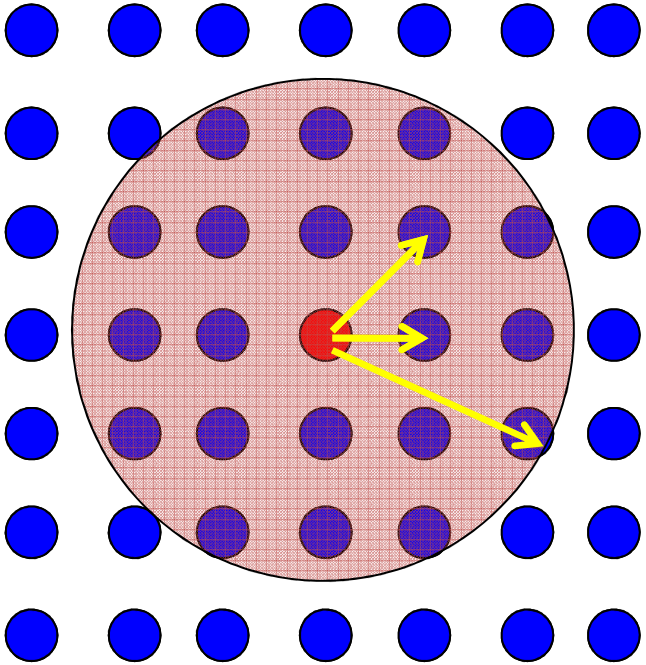


Conceptual diagram of the GPU implementation of a SPH code

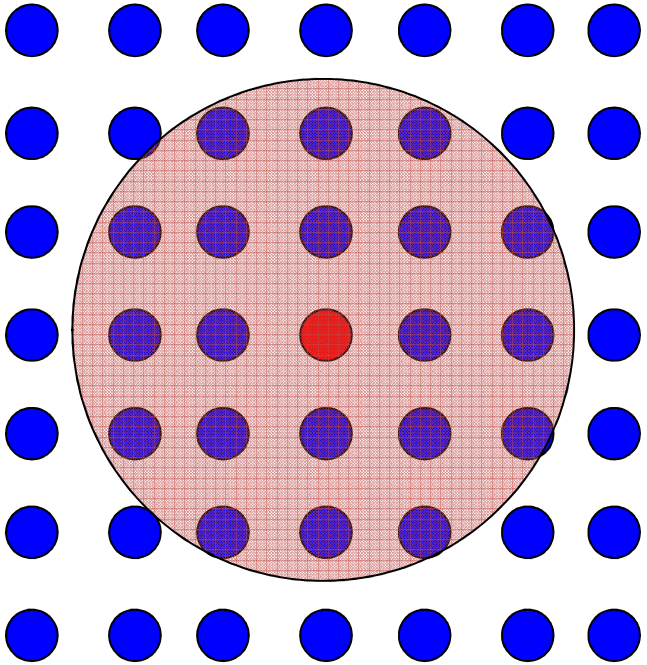
GPU implementation

GPU implementation of PARTICLE INTERACTION:

CPU / C++



GPU / CUDA



$i \rightarrow j = 1$

$i \rightarrow j = 2$

$i \rightarrow j = 3$

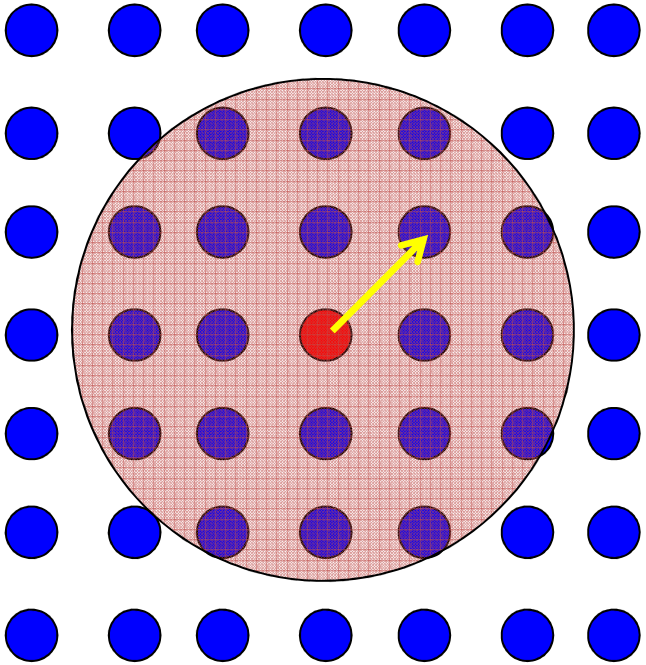
$\mathbf{a}_{ij=3}$

$$\mathbf{a}_i = \mathbf{a}_{i1} + \mathbf{a}_{i2} + \mathbf{a}_{i3} + \ldots$$

GPU implementation

GPU implementation of PARTICLE INTERACTION:

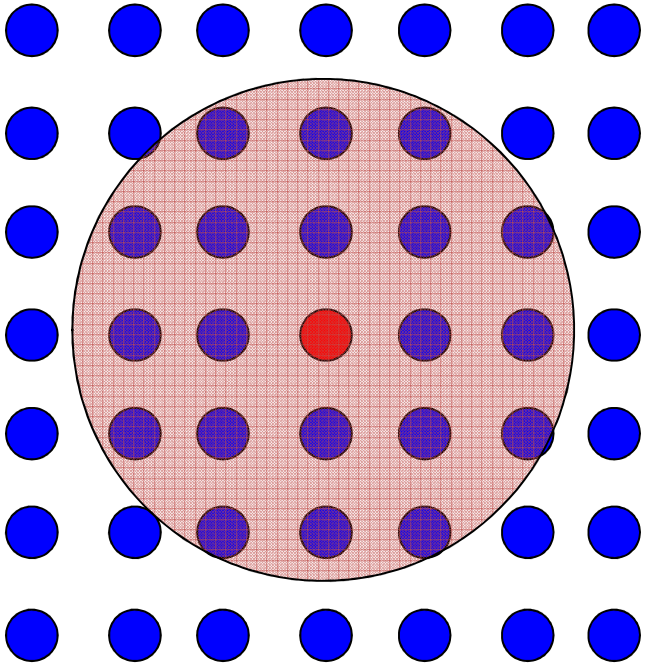
CPU / C++



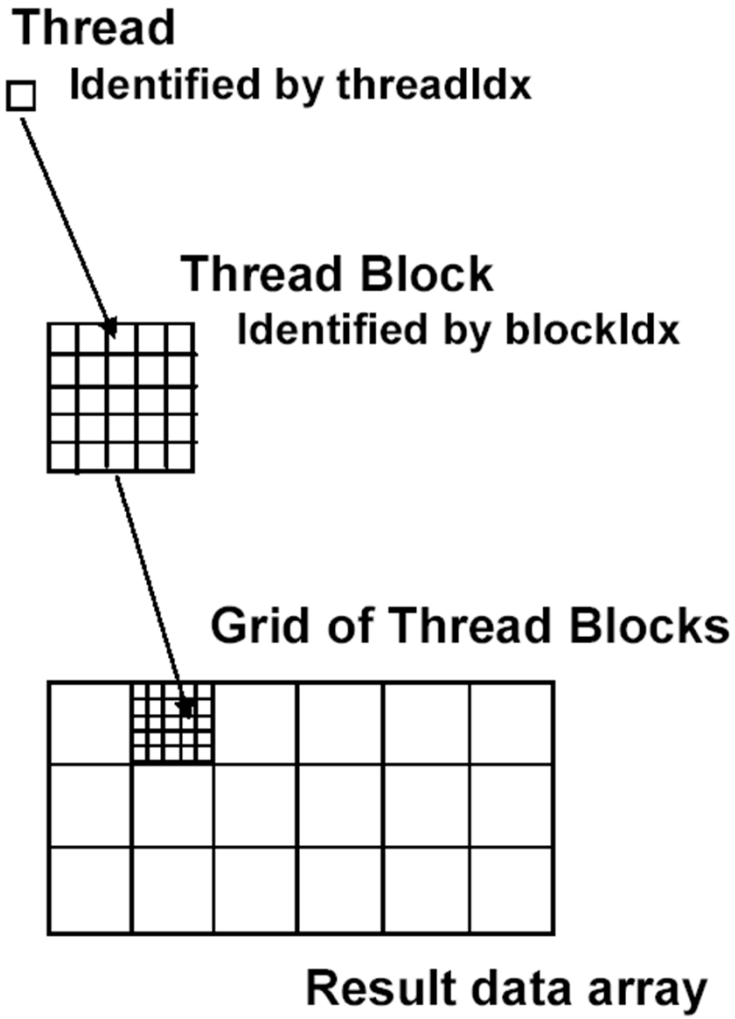
$i \rightarrow j = 1$ $\quad a_{ij=1}$

$a_i = a_{i1} + \ldots$

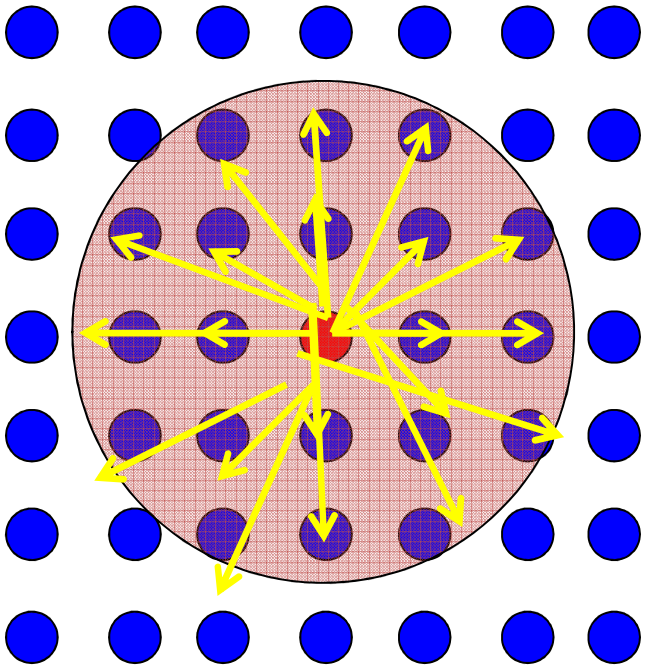
GPU / CUDA



GPU implementation



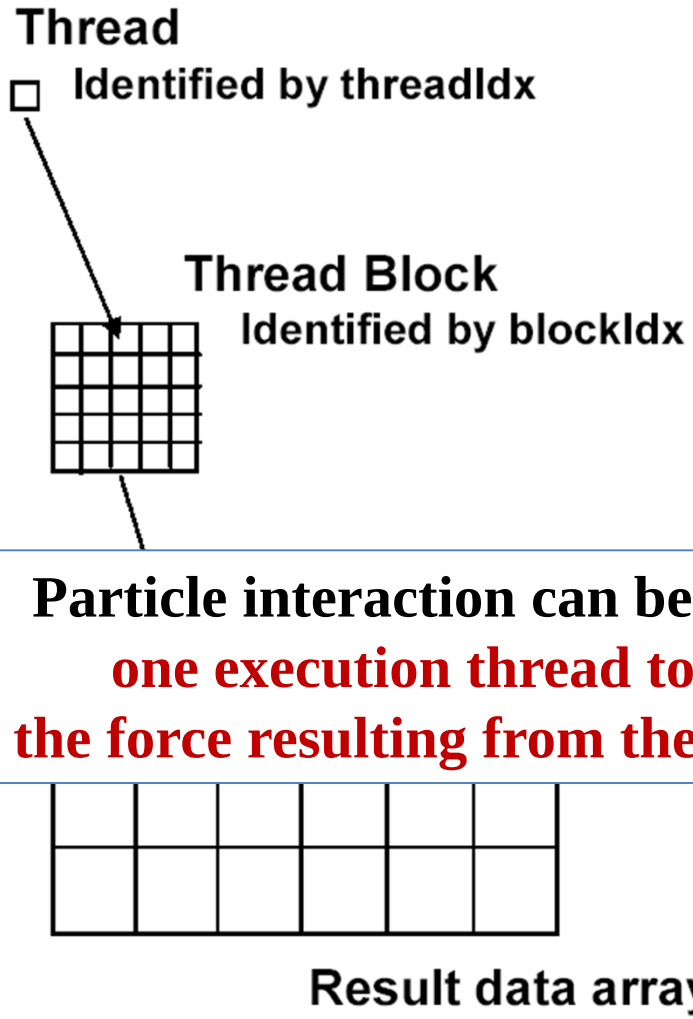
GPU / CUDA



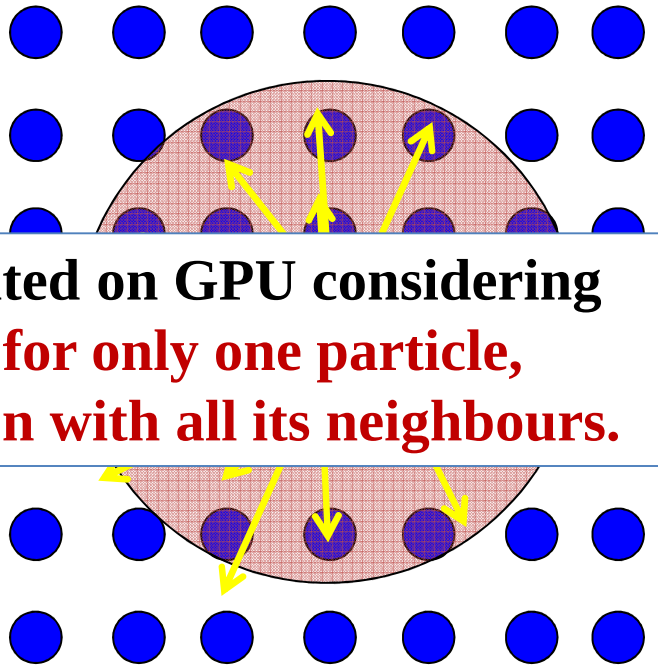
$$\left(\frac{d\mathbf{u}}{dt}\right)_i = -\sum_j m_j \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2}\right) \nabla_i W_{ij} + \mathbf{g}$$

$i \rightarrow j$

GPU implementation



GPU / CUDA

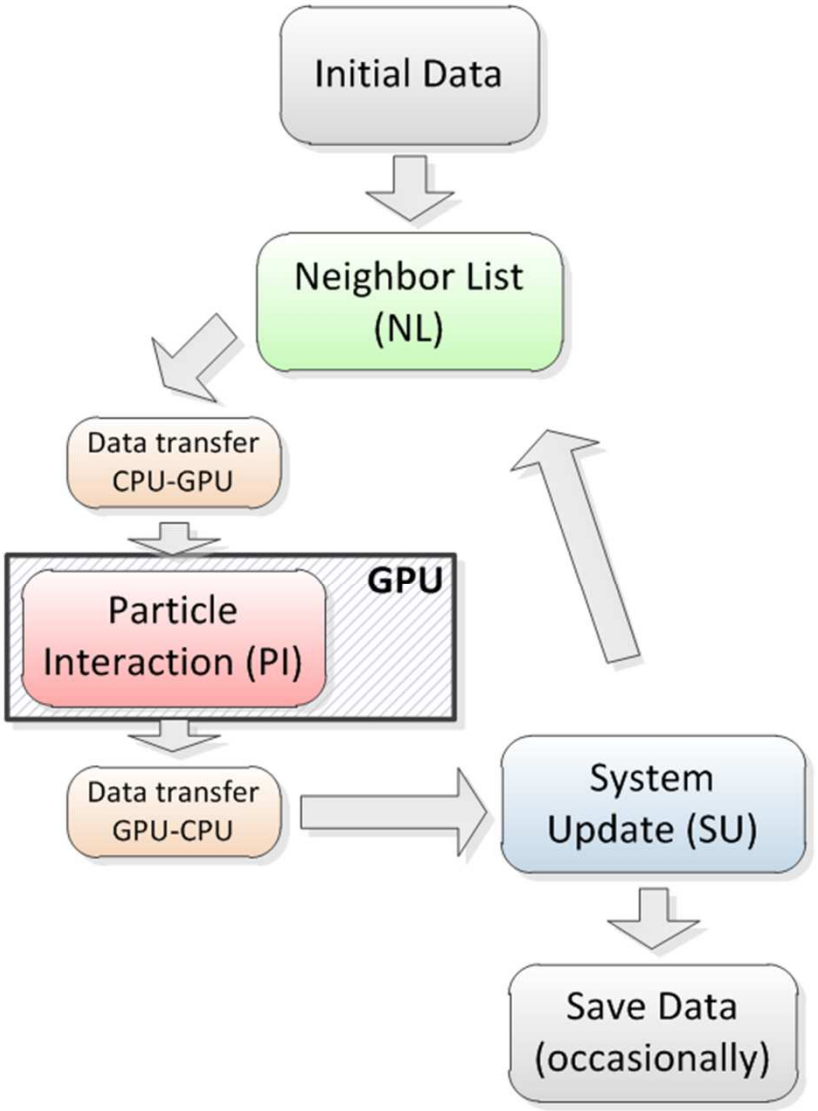


Particle interaction can be implemented on GPU considering
one execution thread to compute, for only one particle,
the force resulting from the interaction with all its neighbours.

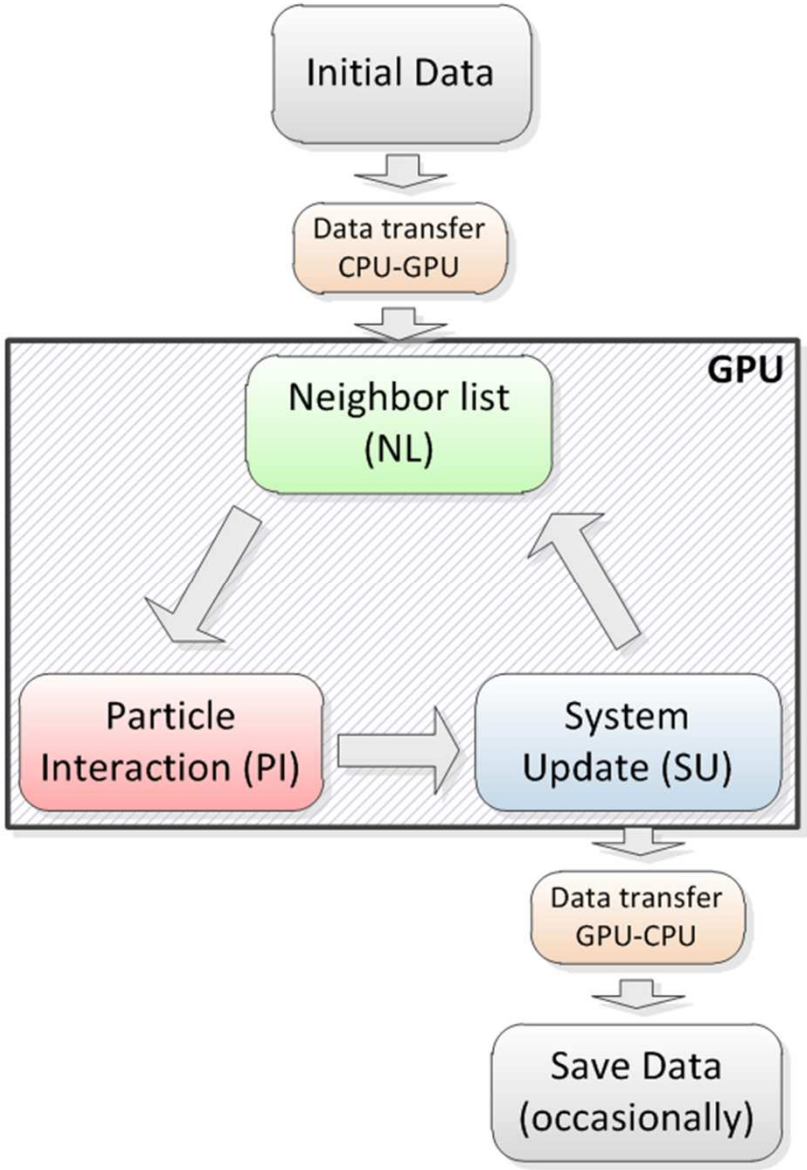
$$\left(\frac{d\mathbf{u}}{dt}\right)_i = -\sum_j m_j \left(\frac{p_j}{\rho_j^2} + \frac{p_i}{\rho_i^2}\right) \nabla_i W_{ij} + \mathbf{g}$$

$i \rightarrow j$

GPU implementation



Conceptual diagram of the partial GPU implementation of a SPH code



Conceptual diagram of the full GPU implementation of a SPH code

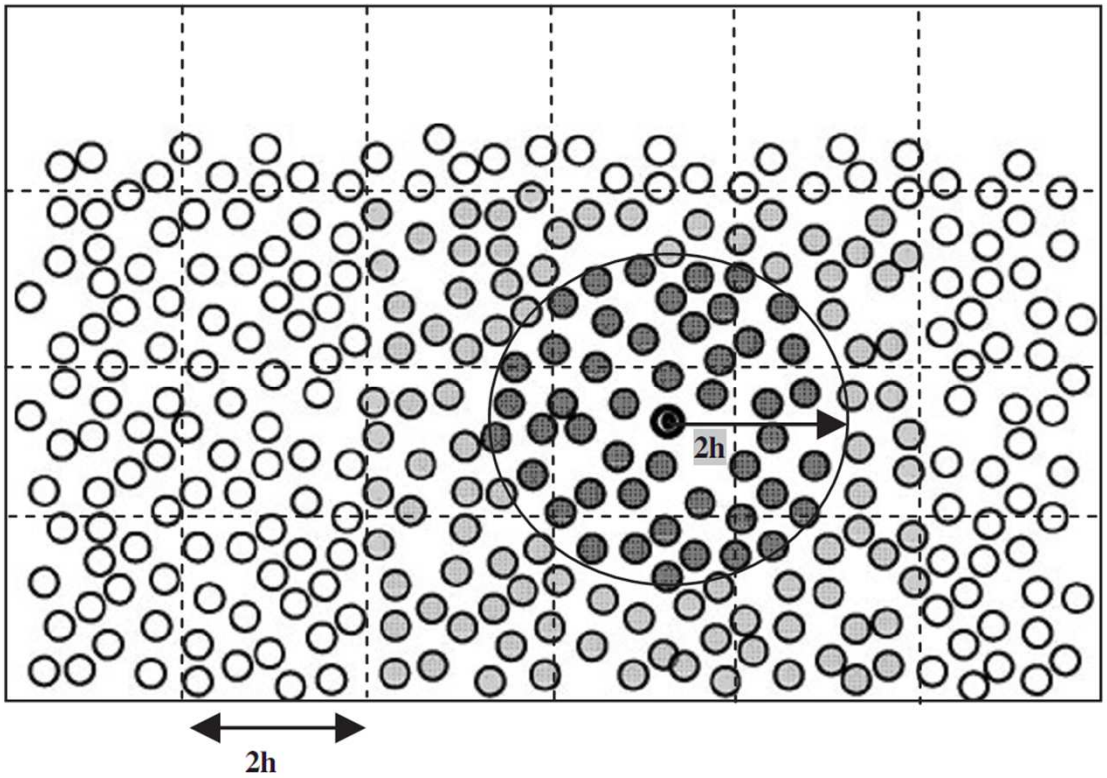
GPU implementation

Dominguez et al. “Neighbour lists in Smoothed Particle Hydrodynamics”. IJNMF, 2010.

GPU implementation of NEIGHBOUR LIST:

Cell-linked list, divided in different operations:

(i) domain division into square cells of side 2h, (or the size of the kernel domain)
(ii) determining the cell to which each particle belongs,
(iii) reordering the particles according to the cells (***radixsort algorithm by CUDA***)
(iv) ordering all arrays with data associated to each particle and, finally
(v) generating an array with the position index of the first particle of each cell.



GPU implementation

GPU implementation of SYSTEM UPDATE:

This process consists on tasks that can be easily parallelized as updating the values of all particle data for the next time step.

We need:

- the current particle data
- acceleration and density derivative
- computing the new value of the time step according to Monaghan and Kos (1999)
- the maximum and minimum values of different variables (force, velocity and sound speed) are calculated using the ***reduction algorithm by CUDA***.

GPU implementation

RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

Expose as much parallelism as possible

Minimize CPU ↔GPU data transfers

Optimize memory usage for maximum bandwidth

Minimize divergent warps

Optimize memory access patterns

Avoiding non-coalesced accesses

Maximize occupancy to hide latency

Graphics Processing Unit

PROBLEMS TO BE SOLVED:

-Memory usage

-Divergence

-Coalescence

-Occupancy

Graphics Processing Unit

Memory usage

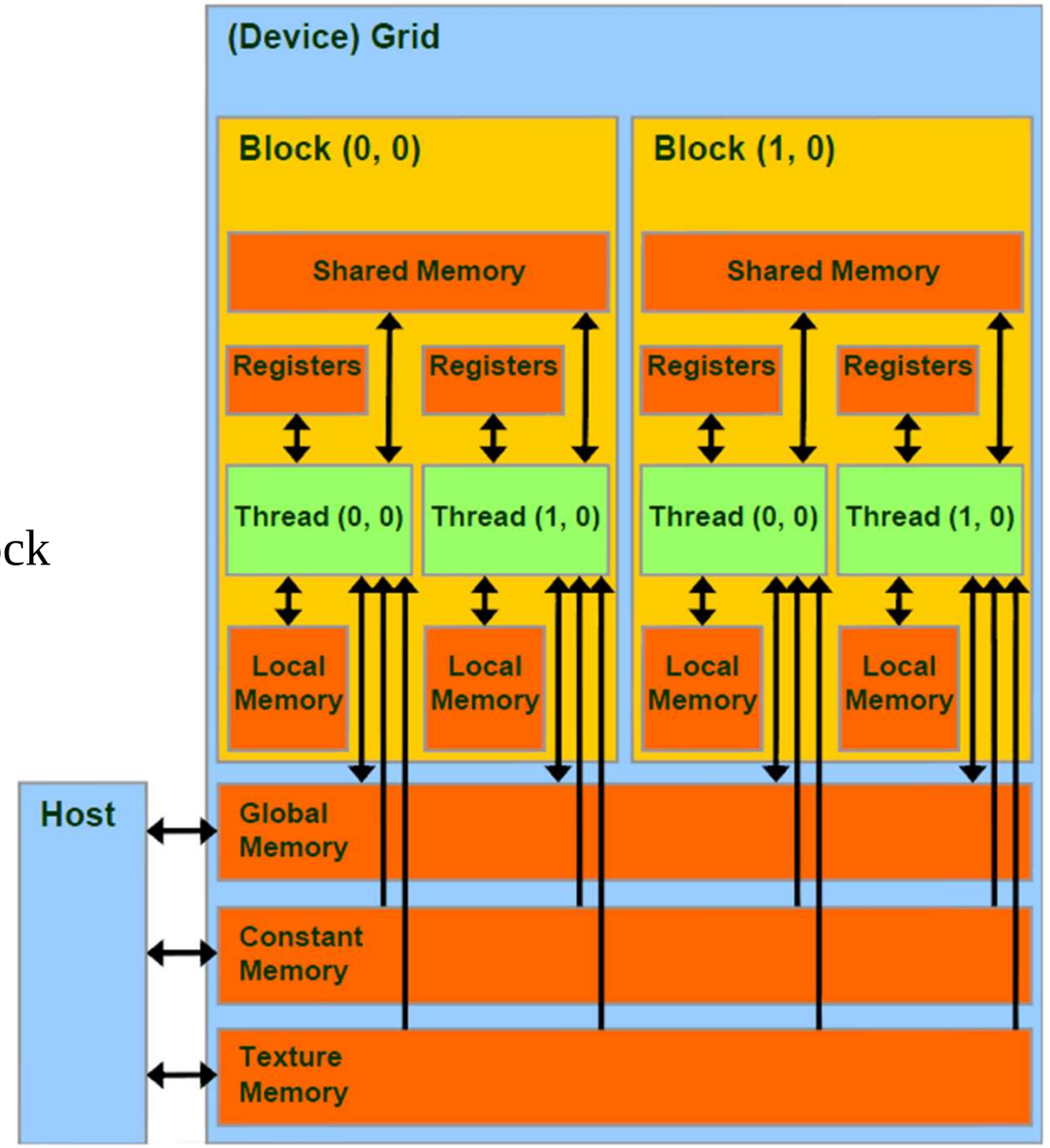
Memory Architecture

Thread: private *local memory*

Block: *shared memory*
visible to all threads of the block

All threads have access to the same *global memory*

2 read-only memory spaces:
the *constant memory*
and *texture memory* espaces



Graphics Processing Unit

Divergence

GPU threads are grouped in sets of 32 named *warps* (CUDA language).

When a task is being executed over a warp, the **32 threads carry out this task simultaneously**.

However, **due to conditional flow instructions in the code**, not all the threads will perform the same operation, so **the different tasks are executed in a sequential way** giving rise to a high loss of efficiency.

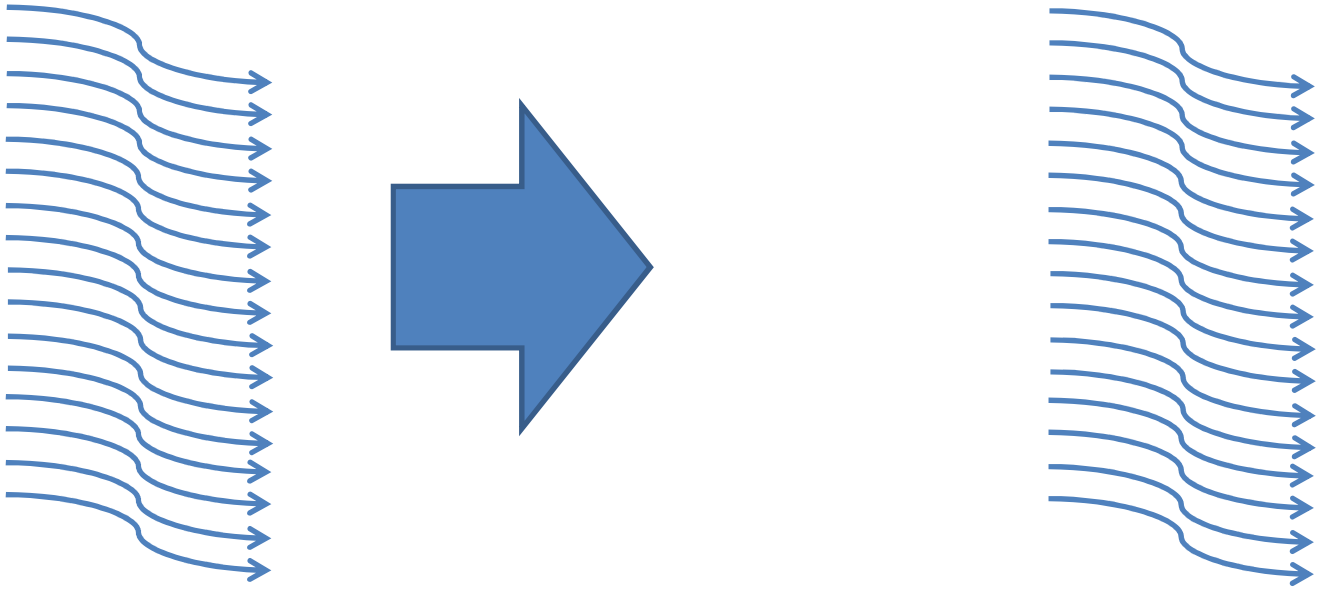
This divergence problem appears during **particle interaction** since each thread has to evaluate what potential neighbors are real neighbors of the particle before computing the force

Graphics Processing Unit

Divergence

each colour represents a task

NO DIVERGENT WARPS



16 threads executing the same task over 16 values

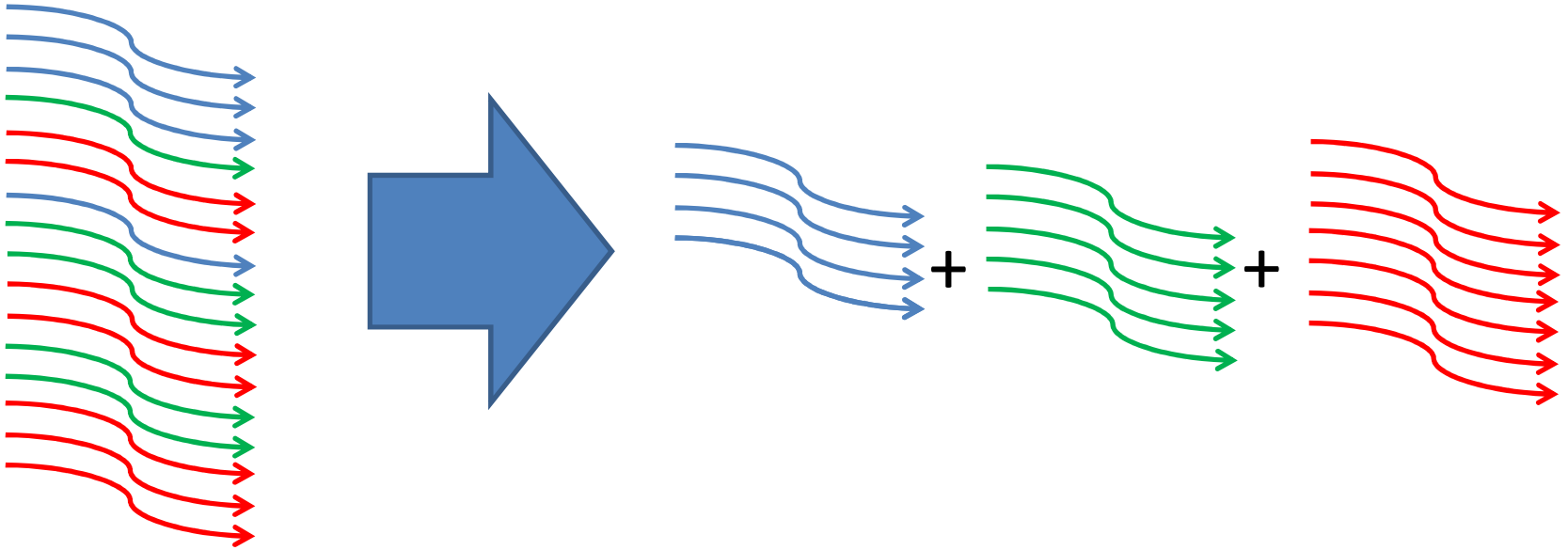
16 threads executed simultaneously

Graphics Processing Unit

Divergence

each colour represents a task

DIVERGENT WARPS !!!



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

execution of the 16 threads
will take the runtime needed to carry out
the three tasks **sequentially**

Graphics Processing Unit

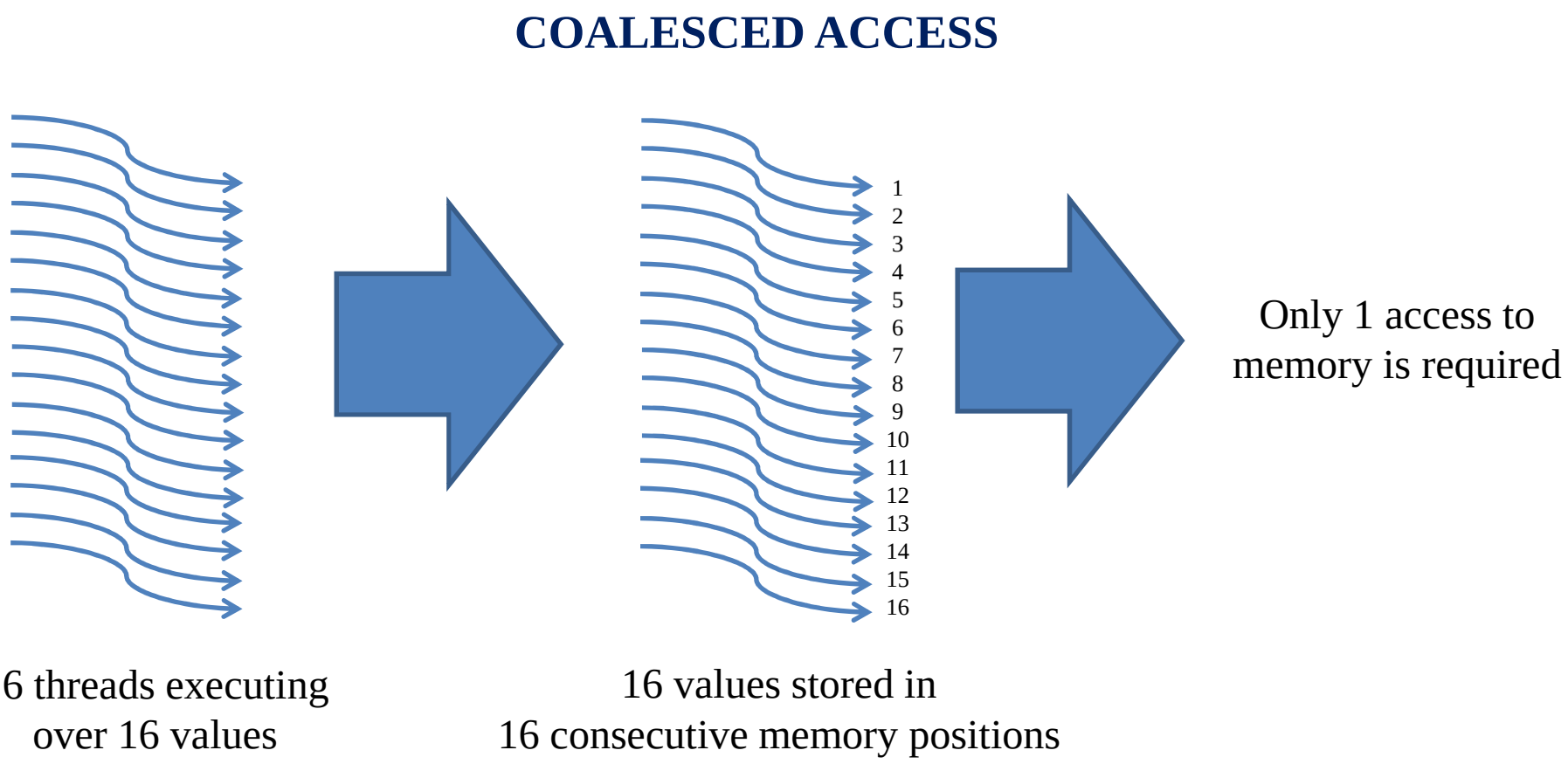
Coalescence

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 particle interaction,
although particle data are reordered according to the cells they belong to, a regular memory access is not possible since each particle has different neighbors and **therefore each thread will access to different memory positions**, which may, eventually, be far from the rest of the positions in the warp.

Graphics Processing Unit

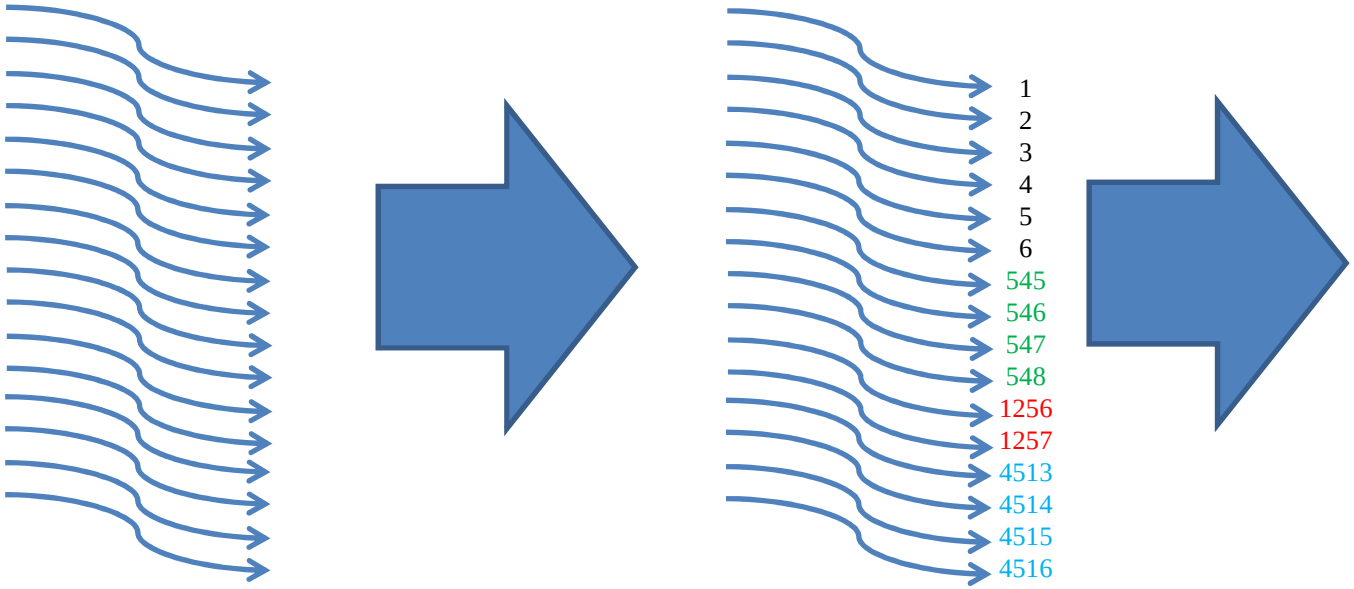
Coalescence



Graphics Processing Unit

Coalescence

NON COALESCED ACCESS



16 threads executing over 16 values

16 values stored in different memory positions

4 memory accesses are required

Graphics Processing Unit

Occupancy

Occupancy is **the ratio of active warps to the maximum number of warps supported on a multiprocessor** of the GPU or Streaming Multiprocessor (SM).

Since the access to the GPU global memory is very irregular during the particle interaction, it is **essential to have the largest number of active warps** in order to hide the latencies of memory access and maintain the hardware as busy as possible.

The **number of active warps depends on** the registers required for the CUDA kernel, the GPU specifications and the number of threads per block.

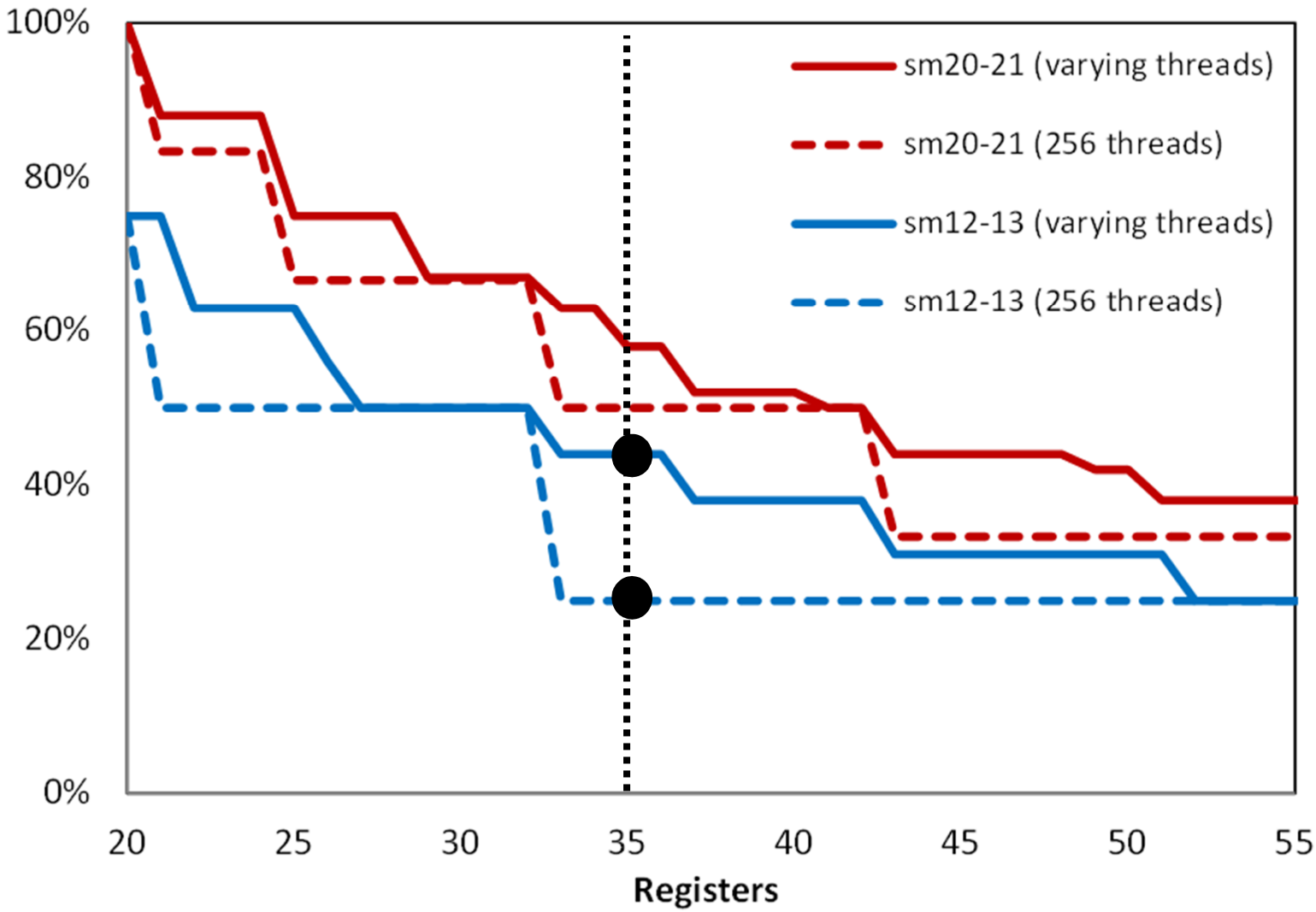
Graphics Processing Unit

Occupancy

Technical specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**
Max. of threads per block	512				1024
Max. of resident blocks per SM	8				
Max. of resident warps per SM	24		32		48
Max. of resident threads per SM	768		1024		1536
Max. of 32-bit registers per SM	8 K		16 K		32 K

Graphics Processing Unit

Occupancy



GPU implementation

RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

Expose as much parallelism as possible

Minimize CPU ↔GPU data transfers

Optimize memory usage for maximum bandwidth

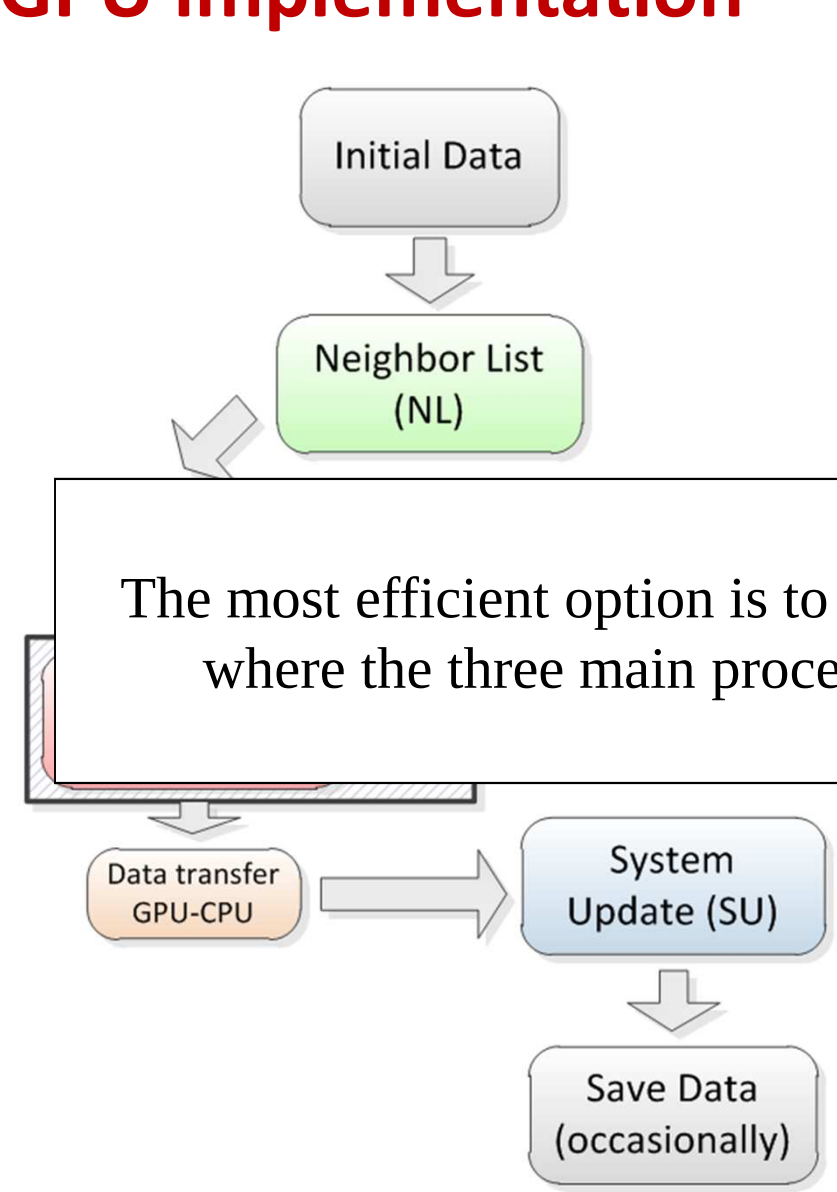
Minimize divergent warps

Optimize memory access patterns

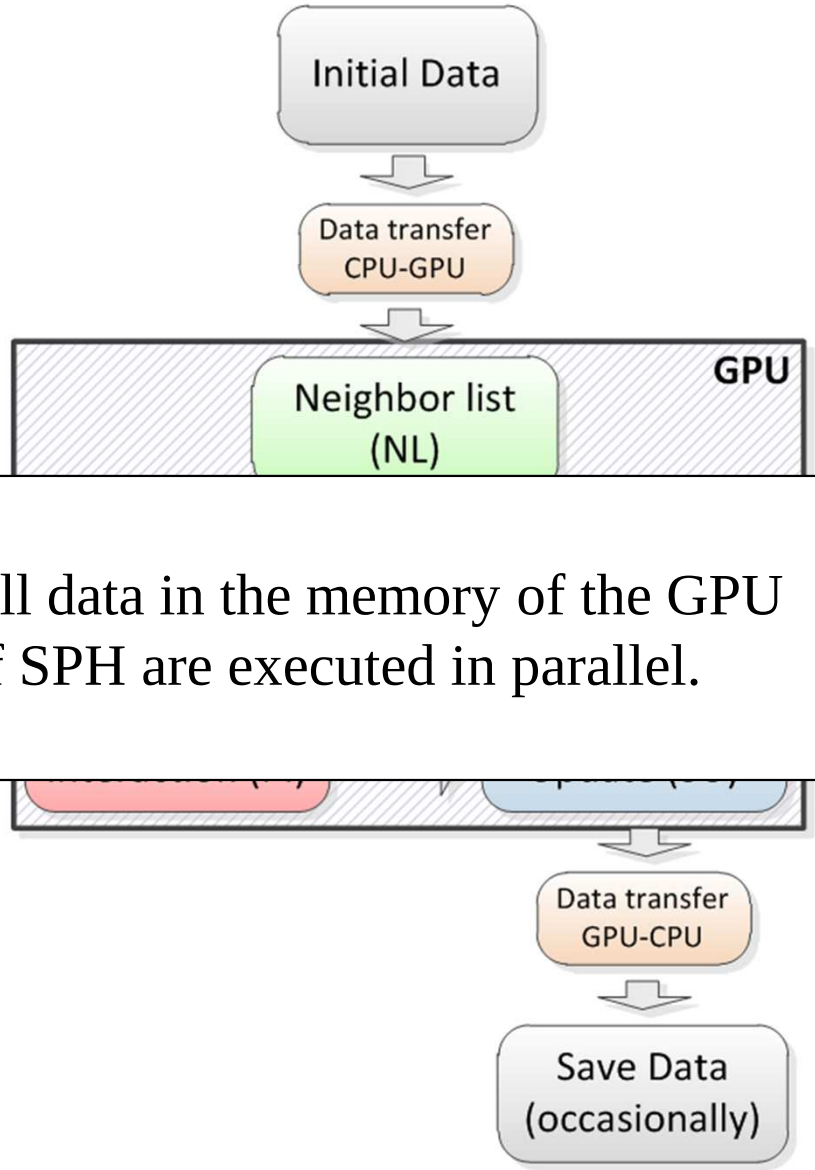
Avoiding non-coalesced accesses

Maximize occupancy to hide latency

GPU implementation



Conceptual diagram of the partial GPU implementation of a SPH code



Conceptual diagram of the full GPU implementation of a SPH code

The most efficient option is to keep all data in the memory of the GPU where the three main processes of SPH are executed in parallel.

GPU implementation

RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

Expose as much parallelism as possible **DONE**

Minimize CPU ↔GPU data transfers

Optimize memory usage for maximum bandwidth

Minimize divergent warps

Optimize memory access patterns

Avoiding non-coalesced accesses

Maximize occupancy to hide latency

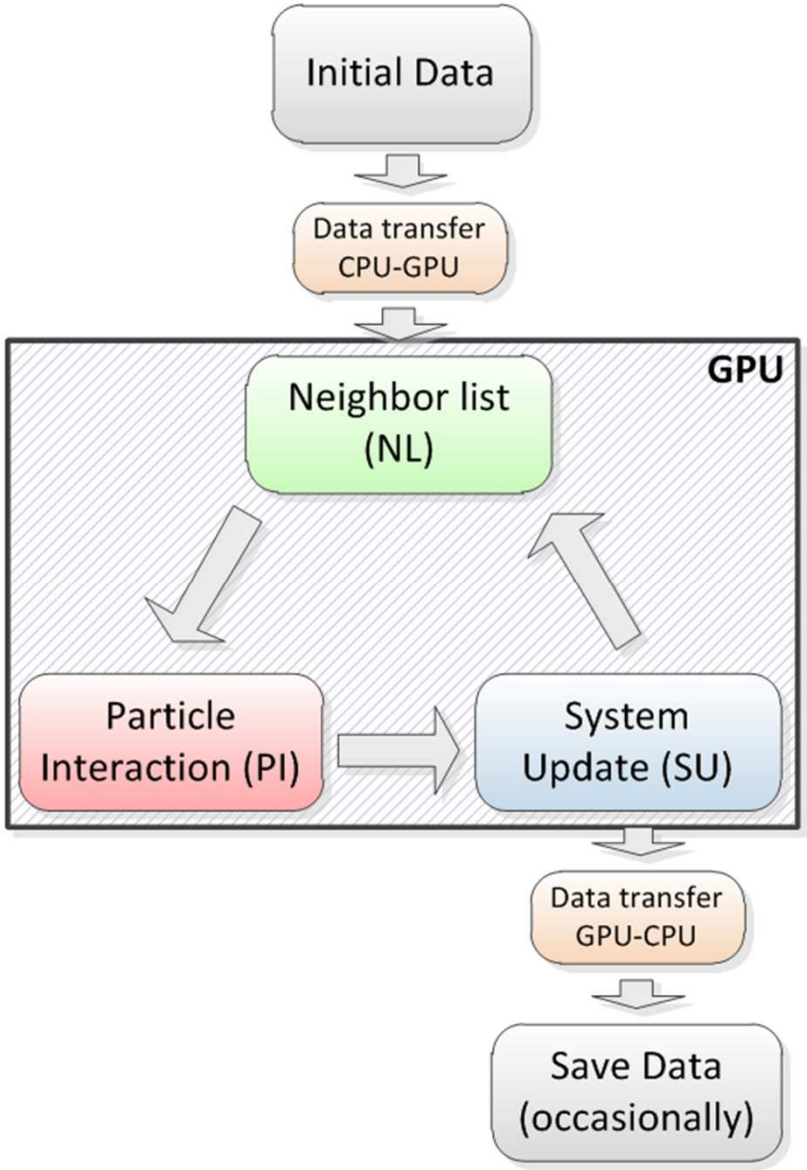
GPU implementation

CPU-GPU COMMUNICATION

- Initially, data were allocated on CPU, so there is a first memory transfer (from CPU to GPU)

- All particle information remains on the GPU memory.

- When saving data is required, only the desired data is transfer from GPU to CPU.



Conceptual diagram of the full GPU implementation of a SPH code

GPU implementation

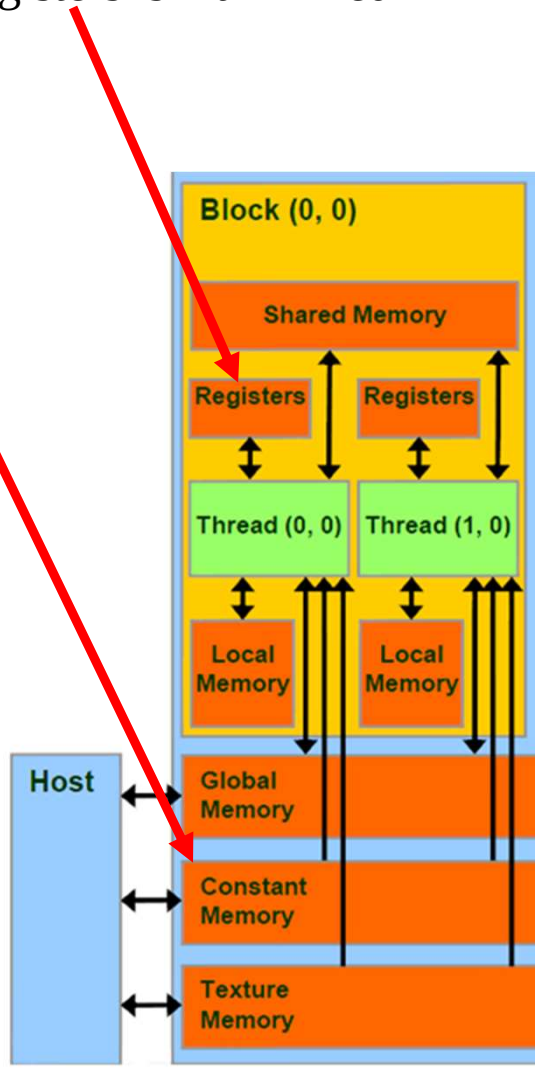
RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

- Expose as much parallelism as possible **DONE**
- Minimize CPU ↔GPU data transfers **DONE**
- Optimize memory usage for maximum bandwidth
- Minimize divergent warps
- Optimize memory access patterns
- Avoiding non-coalesced accesses
- Maximize occupancy to hide latency

GPU implementation

The use of the constant memory and registers is maximized



GPU implementation

RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

Expose as much parallelism as possible **DONE**

Minimize CPU ↔GPU data transfers **DONE**

Optimize memory usage for maximum bandwidth **IMPROVED**

Minimize divergent warps

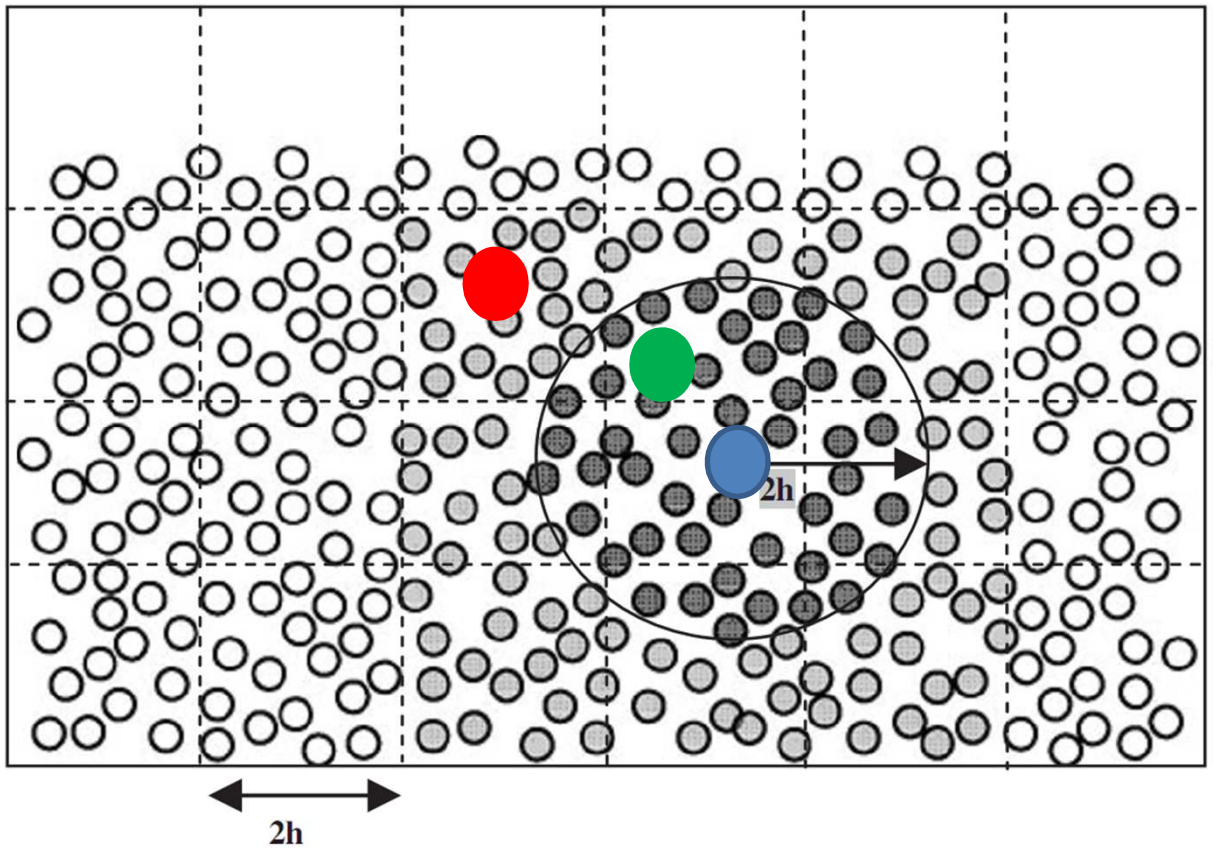
Optimize memory access patterns

Avoiding non-coalesced accesses

Maximize occupancy to hide latency

GPU implementation

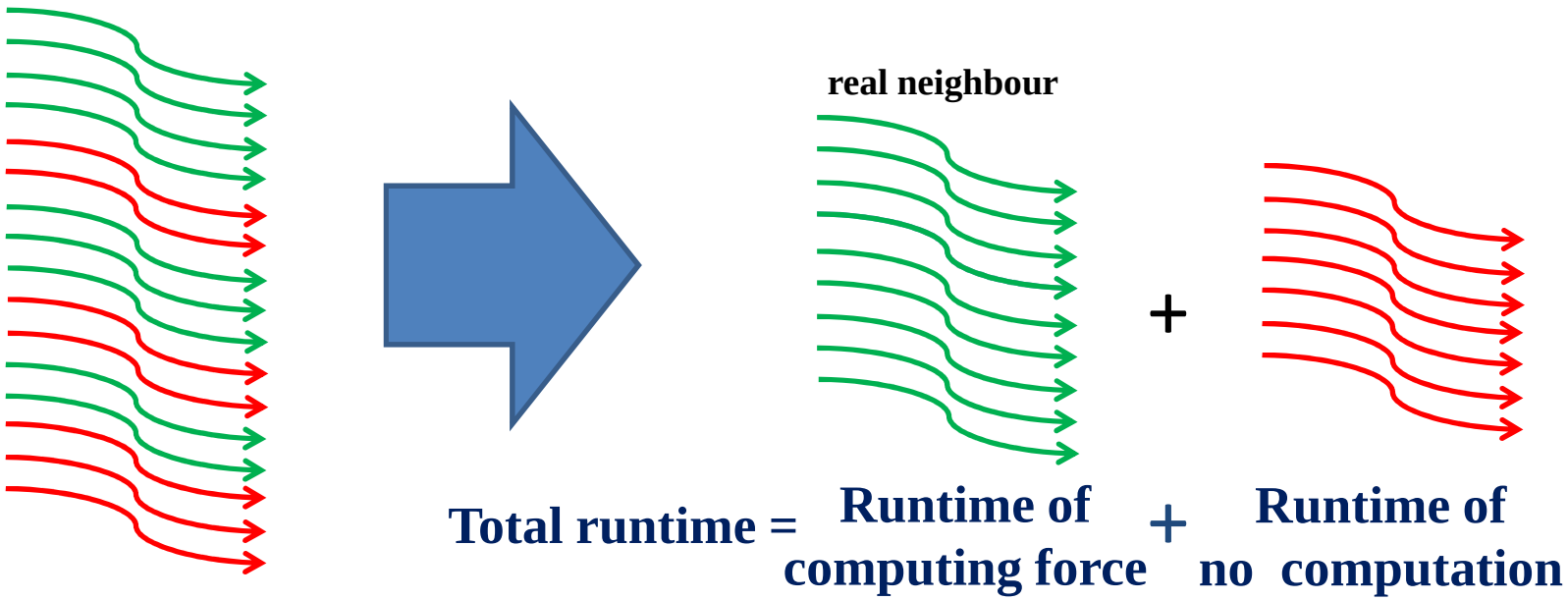
- **code divergence** can appear since when the possible neighbours of a particle are evaluated, some of them are real neighbours and the force computation is carried out while other particles are not real neighbours and no computation is performed.



GPU implementation

Divergence

DIVERGENT WARPS !!!



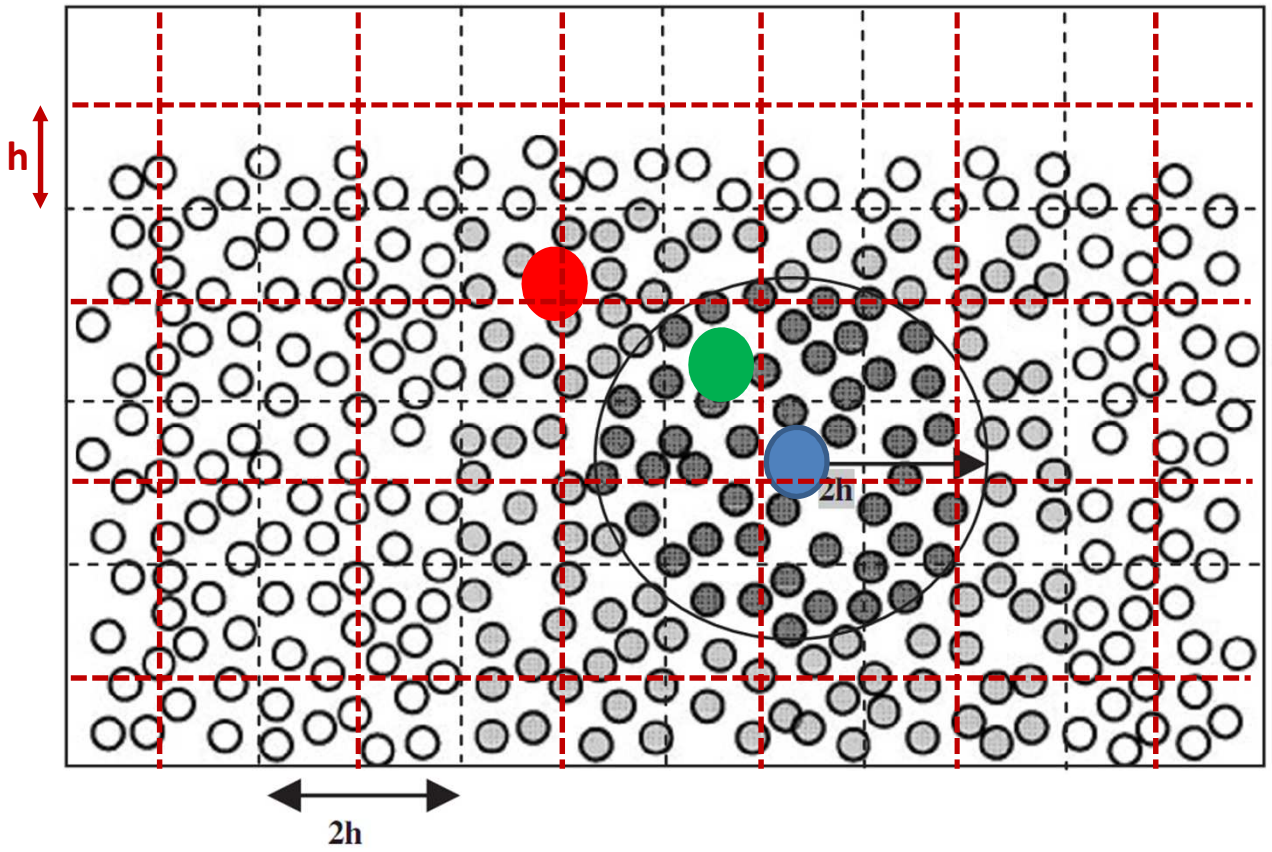
16 threads executing
two different tasks
over 16 values

execution of the 16 threads
will take the runtime needed to carry
out the two tasks sequentially

GPU implementation

- code divergence can appear since when the possible neighbours of a particle are evaluated, some of them are real neighbours and the force computation is carried out while other particles are not real neighbours and no computation is performed.

- using cells of size *h* instead of *2h*, we increase the number of half-warps with only one execution task (all real neighbours or none), thus we reduce the divergence



GPU implementation

RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

Expose as much parallelism as possible **DONE**

Minimize CPU ↔GPU data transfers **DONE**

Optimize memory usage for maximum bandwidth **IMPROVED**

Minimize divergent warps **IMPROVED**

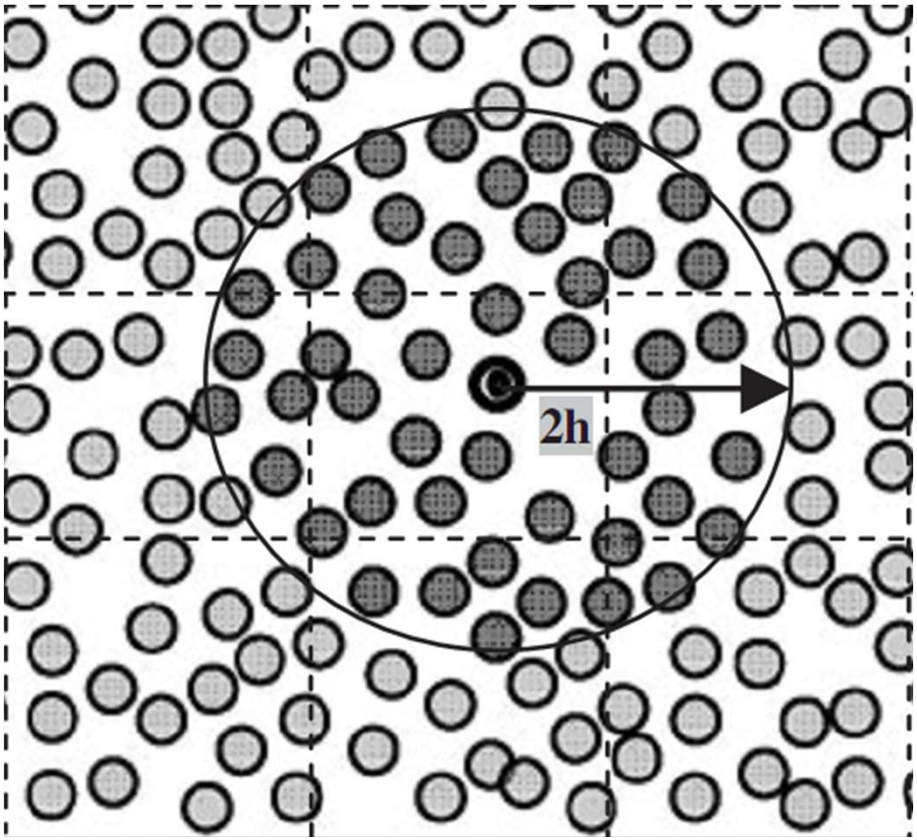
Optimize memory access patterns

Avoiding non-coalesced accesses

Maximize occupancy to hide latency

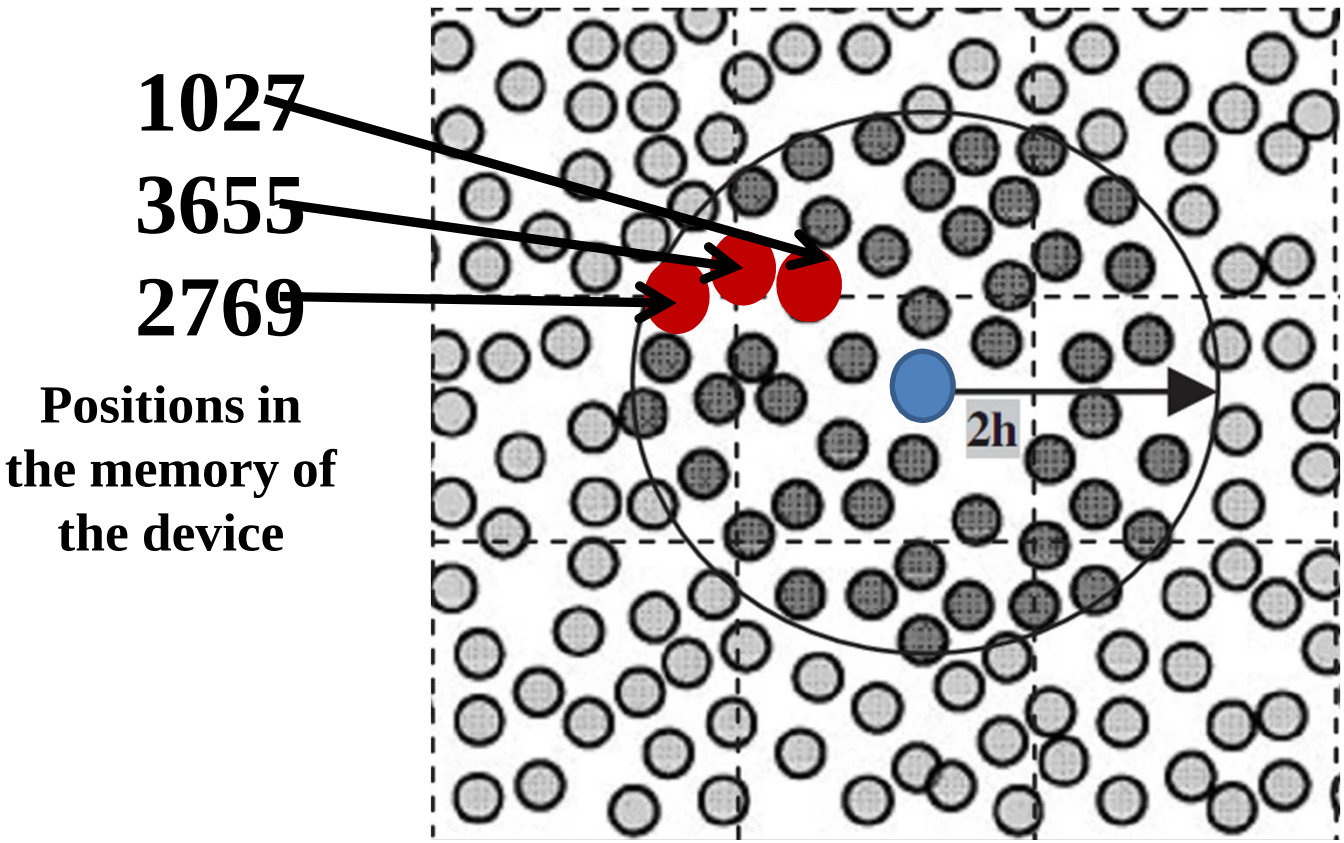
GPU implementation

- the access to the global memory of the device is irregular because there is no way to organise the data to get a coalescent access for all the particles.



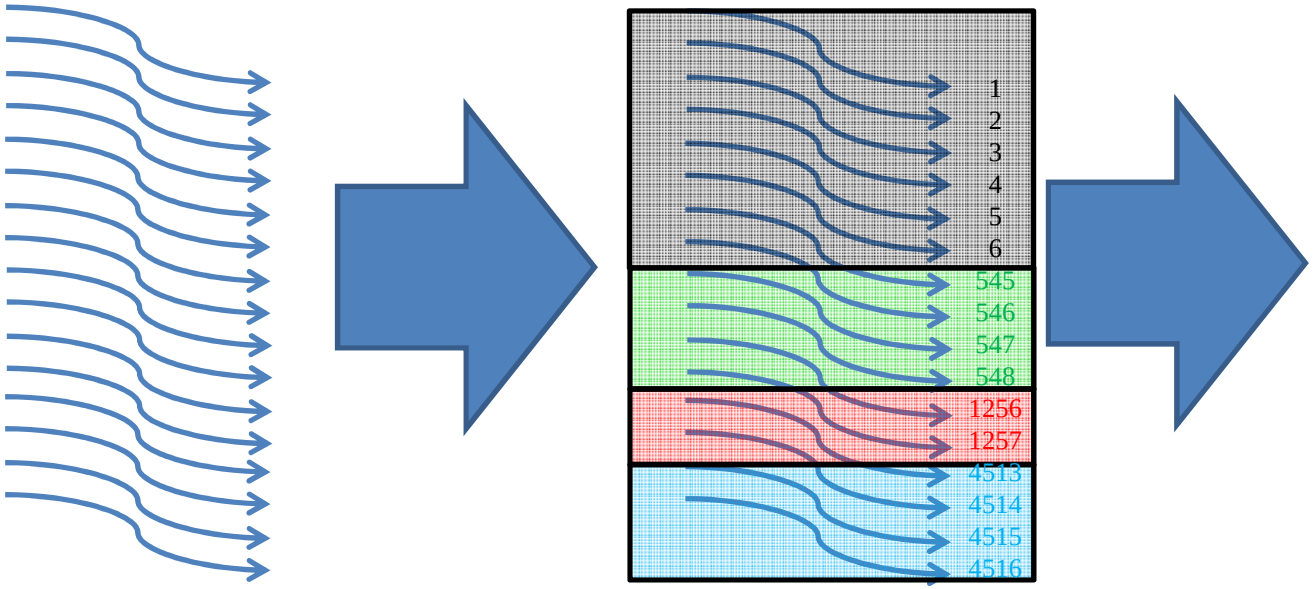
GPU implementation

- the access to the global memory of the device is irregular because there is no way to organise the data to get a coalescent access for all the particles.



GPU implementation

Coalescence

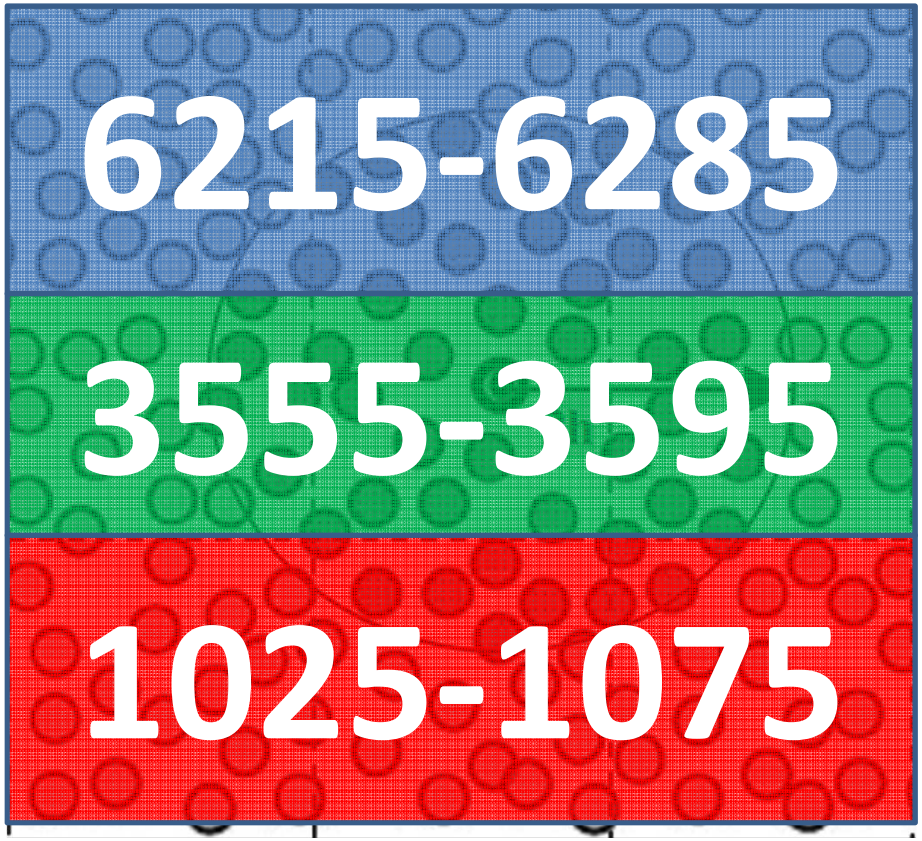


GPU implementation

- the access to the global memory of the device is irregular because there is no way to organise the data to get a coalescent access for all the particles.

- reordering particles, more neighbours are stored in consecutive memory positions

Positions in the memory of the device



GPU implementation

RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

Expose as much parallelism as possible **DONE**

Minimize CPU ↔GPU data transfers **DONE**

Optimize memory usage for maximum bandwidth **IMPROVED**

Minimize divergent warps **IMPROVED**

Optimize memory access patterns **IMPROVED**

Avoiding non-coalesced accesses **IMPROVED**

Maximize occupancy to hide latency

GPU implementation

We use a file with the **number of registers** we can use for all the execution kernels **depending on the compute capability** of the card.

Starting from this value we can **adjust the block size depending on the GPU** card to obtain the **maximum occupancy**.

```
CudaSphApi.cu
tmpxft_00001a5c_00000000-8_CudaSphApi.compute_20.cudafe1.gpu
tmpxft_00001a5c_00000000-12_CudaSphApi.compute_20.cudafe2.gpu
CudaSphApi.cu
tmpxft_00001a5c_00000000-6_CudaSphApi.compute_10.cudafe1.gpu
tmpxft_00001a5c_00000000-16_CudaSphApi.compute_10.cudafe2.gpu
CudaSphApi.cu
tmpxft_00001a5c_00000000-3_CudaSphApi.compute_12.cudafe1.gpu
tmpxft_00001a5c_00000000-20_CudaSphApi.compute_12.cudafe2.gpu
CudaSphApi.cu
ptxas info    : Compiling entry function '_Z27KerCsComputeStepSymplectic2ILb0ELb0EEvjjPjP6float3S2_PfS2_S2_S3_S3_S2_ff' for 'sm_20'
ptxas info    : Used 21 registers, 120 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z26KerCsComputeStepSymplecticILb0ELb0EEvjjPjP6float3S2_PfS2_S2_S3_S3_S2_S2_f' for 'sm_20'
ptxas info    : Used 25 registers, 124 bytes cmem[0], 152 bytes cmem[2]
ptxas info    : Compiling entry function '_Z22KerCsComputeStepVerletILb0ELb0EEvjjPfS0_S0_PjP6float3S3_S3_S3_S3_S3_fff' for 'sm_20'
ptxas info    : Used 23 registers, 132 bytes cmem[0], 152 bytes cmem[2]
ptxas info    : Compiling entry function '_Z27KerCsComputeStepSymplectic2ILb0ELb1EEvjjPjP6float3S2_PfS2_S2_S3_S3_S2_ff' for 'sm_20'
ptxas info    : Used 19 registers, 120 bytes cmem[0], 152 bytes cmem[2]
ptxas info    : Compiling entry function '_Z26KerCsComputeStepSymplecticILb0ELb1EEvjjPjP6float3S2_PfS2_S2_S3_S3_S2_S2_f' for 'sm_20'
ptxas info    : Used 20 registers, 124 bytes cmem[0], 152 bytes cmem[2]
ptxas info    : Compiling entry function '_Z22KerCsComputeStepVerletILb0ELb1EEvjjPfS0_S0_PjP6float3S3_S3_S3_S3_S3_fff' for 'sm_20'
ptxas info    : Used 21 registers, 132 bytes cmem[0], 152 bytes cmem[2]
ptxas info    : Compiling entry function '_Z31KerCsInteractionFt_KHdivBgFluidIL8TpKernel2ELb0ELb0ELj2EEvjjjjjPjP4int2S3_P6float4S5_S1_P6float3PfS7_S8_' for 'sm_20'
ptxas info    : Used 44 registers, 136 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z29KerCsInteractionFt_KHdivFluidIL8TpKernel2ELb0ELb0ELj25EEvjjjPjP5uint2S3_P6float4S5_S1_P6float3PfS7_S8_' for 'sm_20'
ptxas info    : Used 33 registers, 128 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z31KerCsInteractionFt_KHdivBgBoundIL8TpKernel2ELb0ELj2EEvjjjjPjP4int2P6float4S5_S1_PfS6_' for 'sm_20'
ptxas info    : Used 36 registers, 104 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z31KerCsInteractionFt_KHdivBgFluidIL8TpKernel2ELb0ELb1ELj2EEvjjjjjPjP4int2S3_P6float4S5_S1_P6float3PfS7_S8_' for 'sm_20'
ptxas info    : Used 48 registers, 136 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z29KerCsInteractionFt_KHdivBoundIL8TpKernel2ELb0ELj25EEvjjPjP5uint2P6float4S5_S1_PfS6_' for 'sm_20'
ptxas info    : Used 28 registers, 96 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z29KerCsInteractionFt_KHdivFluidIL8TpKernel2ELb0ELb1ELj25EEvjjjPjP5uint2S3_P6float4S5_S1_P6float3PfS7_S8_' for 'sm_20'
ptxas info    : Used 37 registers, 128 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z31KerCsInteractionFt_KHdivBgFluidIL8TpKernel2ELb0ELb0ELj1EEvjjjjjPjP4int2S3_P6float4S5_S1_P6float3PfS7_S8_' for 'sm_20'
ptxas info    : Used 44 registers, 136 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z29KerCsInteractionFt_KHdivFluidIL8TpKernel2ELb0ELb0ELj9EEvjjjPjP5uint2S3_P6float4S5_S1_P6float3PfS7_S8_' for 'sm_20'
ptxas info    : Used 33 registers, 128 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
ptxas info    : Compiling entry function '_Z31KerCsInteractionFt_KHdivBgBoundIL8TpKernel2ELb0ELj1EEvjjjjPjP4int2P6float4S5_S1_PfS6_' for 'sm_20'
ptxas info    : Used 36 registers, 104 bytes cmem[0], 152 bytes cmem[2], 4 bytes cmem[16]
```

GPU implementation

RECIPES TO COOK SPH-GPU

Basic strategies for Performance Optimization

Expose as much parallelism as possible **DONE**

Minimize CPU ↔GPU data transfers **DONE**

Optimize memory usage for maximum bandwidth **IMPROVED**

Minimize divergent warps **IMPROVED**

Optimize memory access patterns **IMPROVED**

Avoiding non-coalesced accesses **IMPROVED**

Maximize occupancy to hide latency **DONE**

GPU implementation

Introduction to GPUs and CUDA

Implementation techniques and optimizations

Available hardware: GPUs

Results and speedups

GPU implementation Available hardware

GPU	# of cores	Processor clock (GHz)	Memory (GB)	Power usage (watts)	Heat (ºC)	Price (€)
M1060	240	1.36	4	188	*good cooling system for server*	1200*
GTX 285	240	1.48	1	204	105	330*
GTX 480	**480**	**1.40**	**1.5**	**384**	**94**	**450***
GTX 590	1024	2x1.54	3	496	74	650

* year 2009

GPU implementation

Introduction to GPUs and CUDA

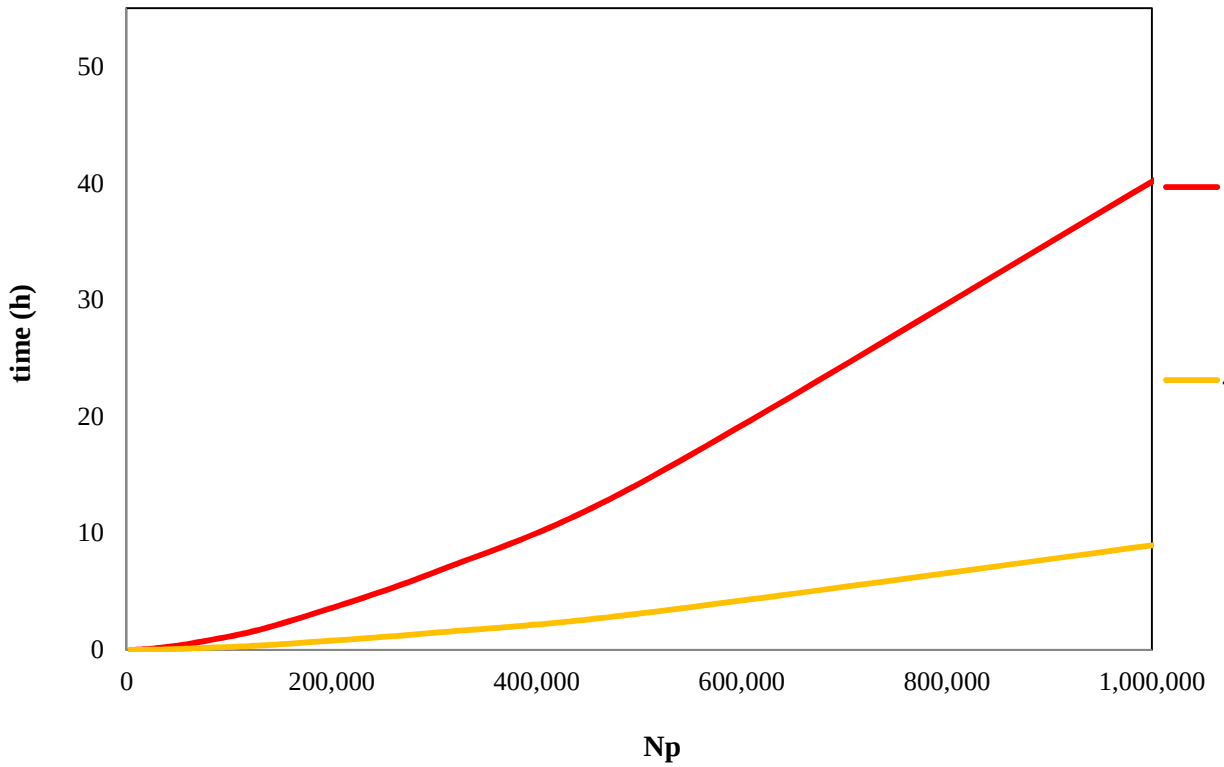
Implementation techniques and optimizations

Available hardware: GPUs

Results and speedups

GPU implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores



40 h

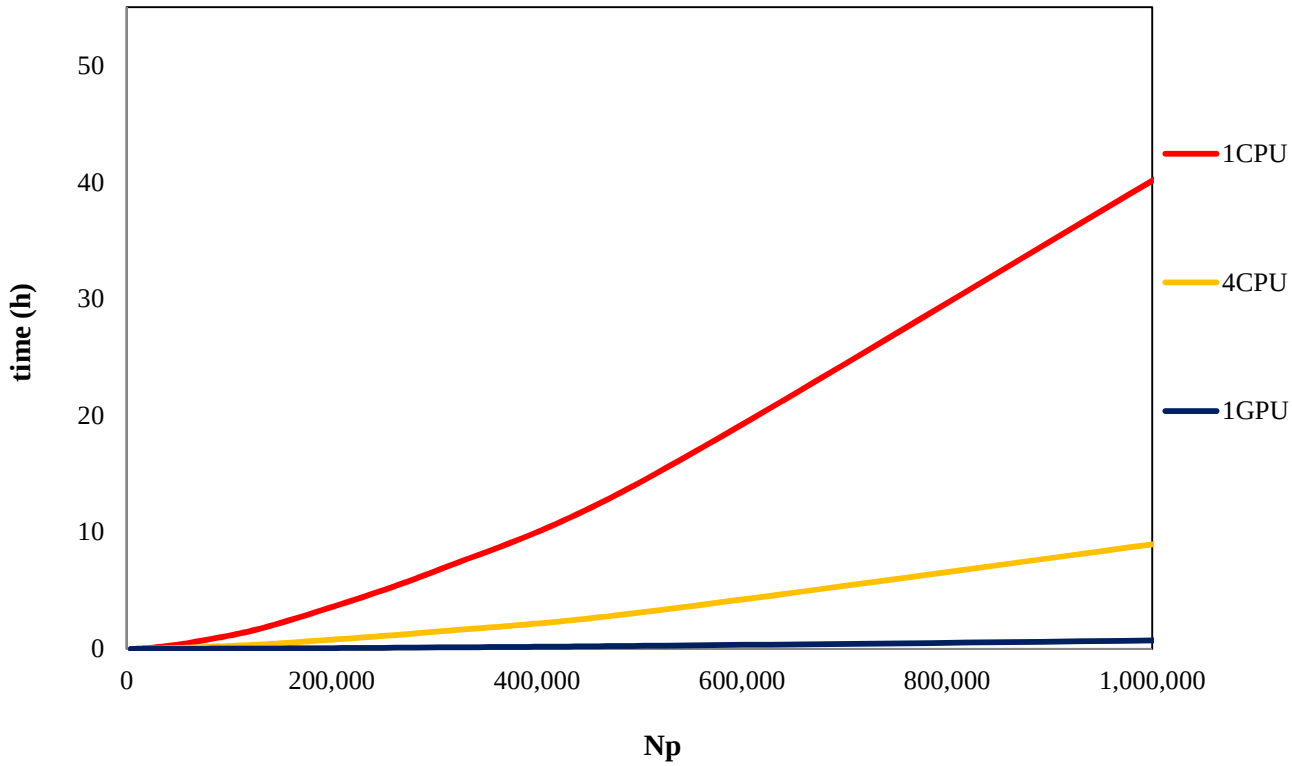
9 h

4.5x

Computational runtimes with the Multi-CPU model

GPU implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
GPU GTX 480 at 1.40GHz with 480 cores



40 h

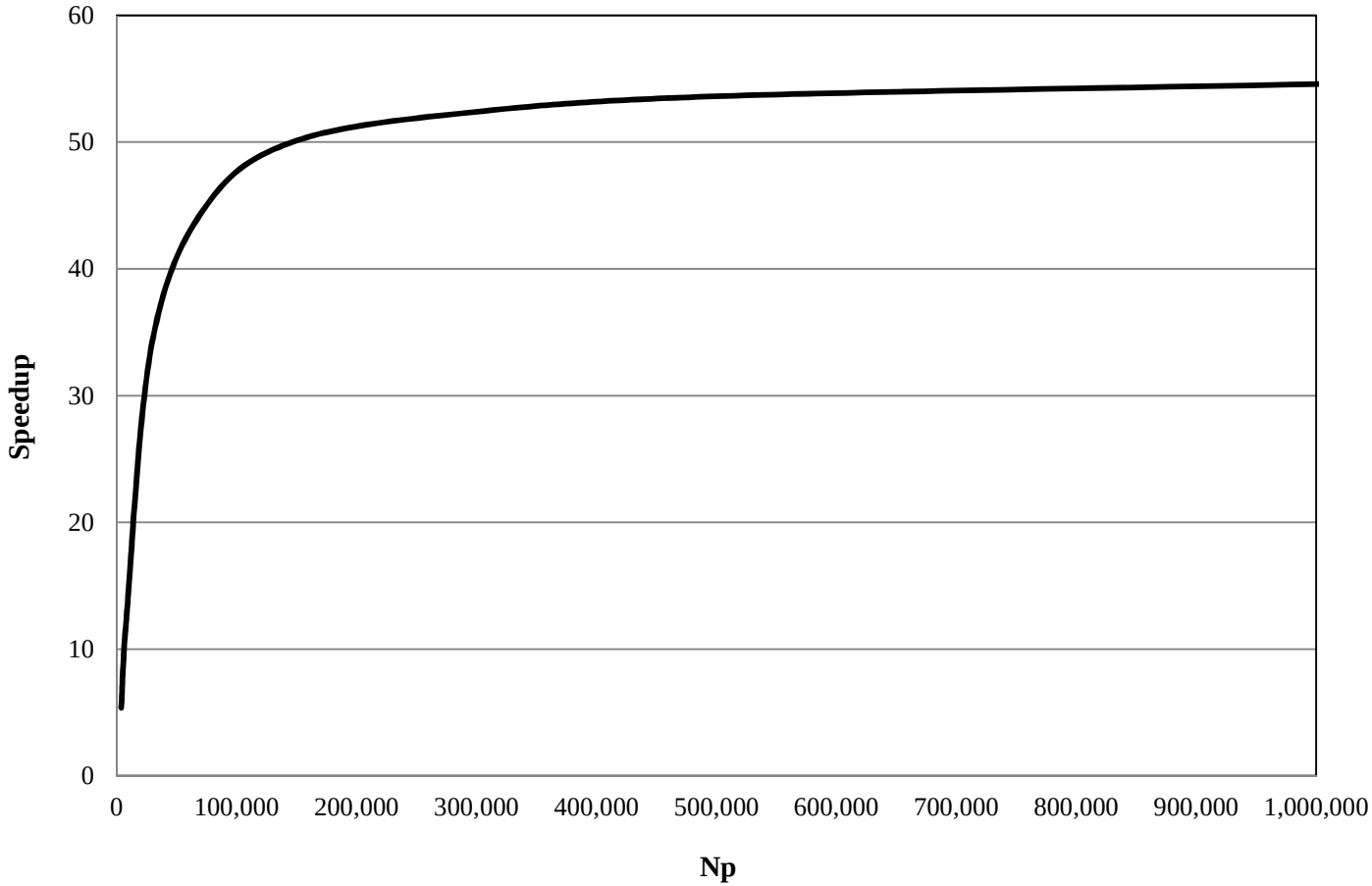
9 h

45 min

Computational runtimes with the Multi-CPU and GPU model

GPU implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
GPU GTX 480 at 1.40GHz with 480 cores



55x

Speedup of using the GPU model

GPU implementation

The FERMI card is **55** times more efficient than the best CPU single-core code.

The achieved **performance can be compared to the big cluster machines**.

Following Maruzewski et al. 2010:
100 cores with efficiency of 60% of the supercomputer IBM Blue Gene/L
to equal the speedup achieved by only a Fermi Card

1 GTX480= 100 cores of BlueGene

GPU implementation

The FERMI card is **55** times more efficient than the best CPU single-core code.

The achieved **performance can be compared to the big cluster machines**.

Following Maruzewski et al. 2010:
100 cores with efficiency of 60% of the supercomputer IBM Blue Gene/L
to equal the speedup achieved by only a Fermi Card

1 GTX480= 450 EUROS

GPU implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
GPU GTX 480 at 1.40GHz with 480 cores

1M	hours	Speedup vs. 1 CPU	Speedup vs. 4 CPU
1CPU	40.71	**1.00**	
4CPU	9.09	4.48	**1.00**
1GPU	0.75	54.61	12.19

450 EUROS

Speedup of using the GPU model

Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- **Multi-GPU implementation**

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

Multi-GPU implementation

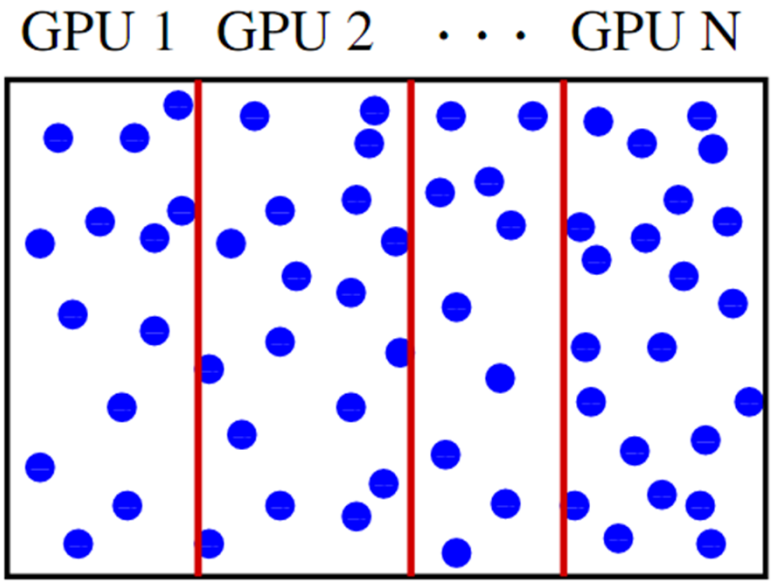
The memory requirements are still a **limitation for a single GPU**, using more than one GPU appears to be the best development to continue accelerating SPH simulations.

In order to allow different devices communicating with each other, the Message Passing Interface **(MPI) is used jointly with CUDA** to implement a multi-GPU version of SPH.

MPI presents the advantage of using different compute nodes hosting multiple devices instead of only one as it happens with OpenMP.

Multi-GPU implementation

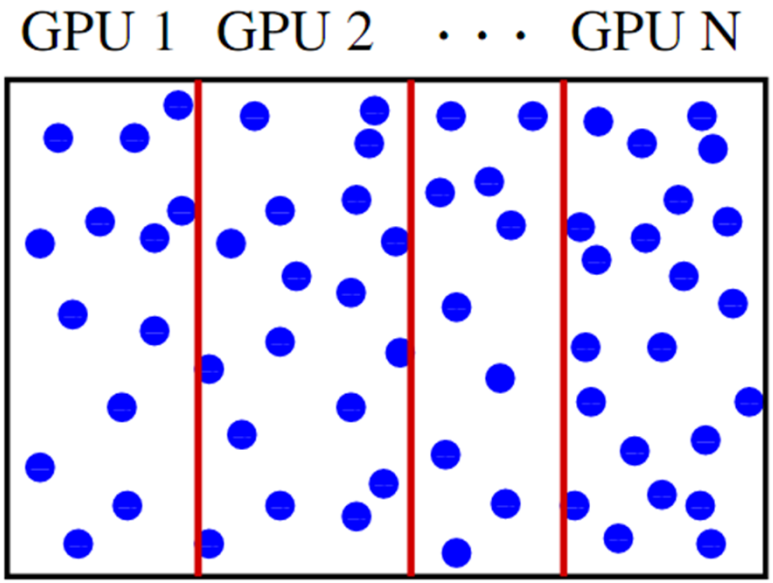
The multi-GPU implementation of Valdez-Balderas et al., 2011 consists of assigning **different portions of the physical system to different GPUs**

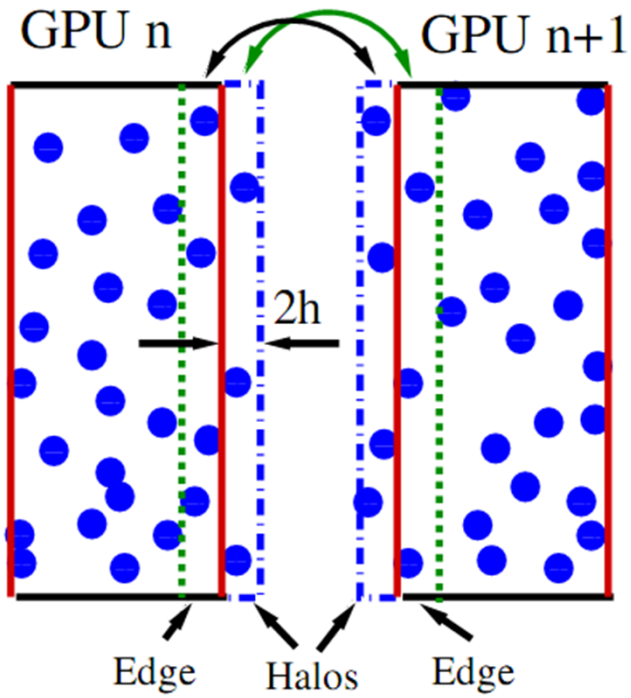


Multi-GPU implementation

The multi-GPU implementation of Valdez-Balderas et al., 2011 consists of assigning **different portions of the physical system to different GPUs**

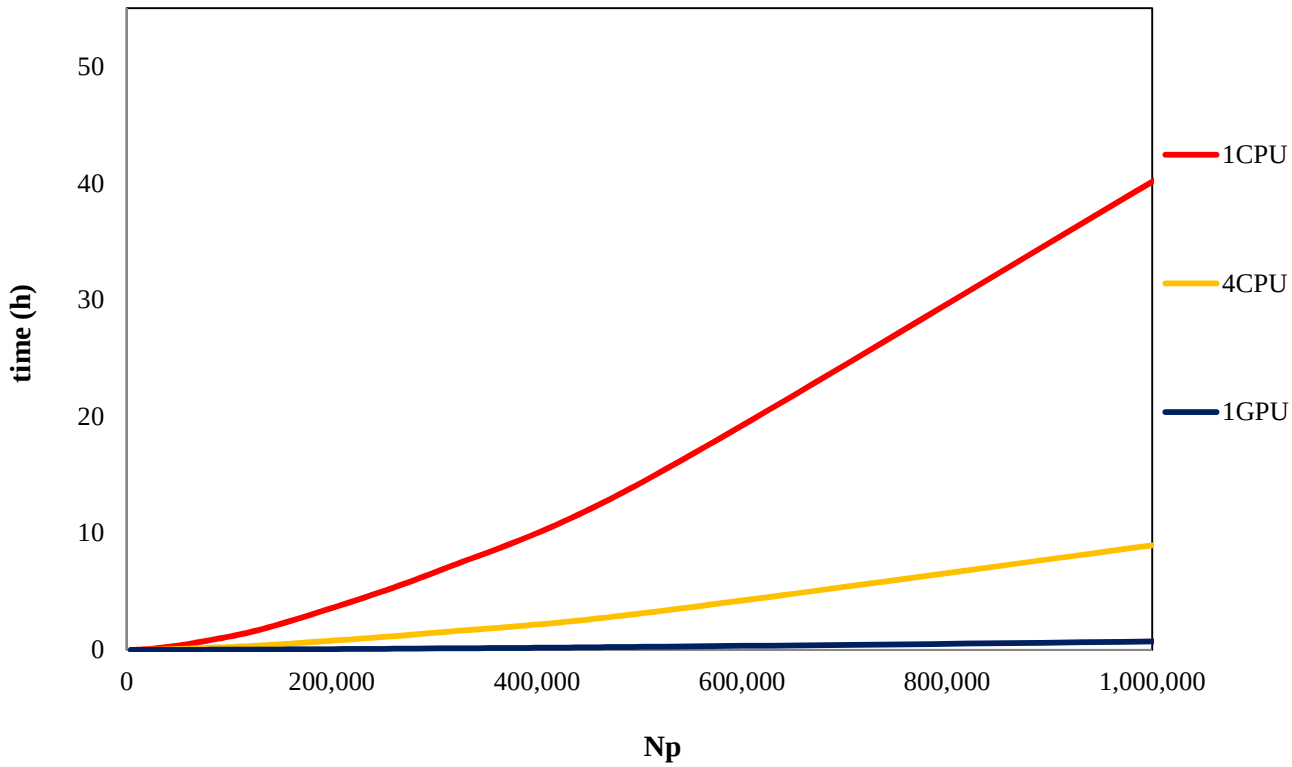
After each computation step, **data needs to be transferred between devices**;
-the information of particles that migrate between GPUs (physical sub-domains)
-or particles that belong to shared spaces where data is used by several GPUs.





Multi-GPU implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
GPU GTX 480 at 1.40GHz with 480 cores



40 h

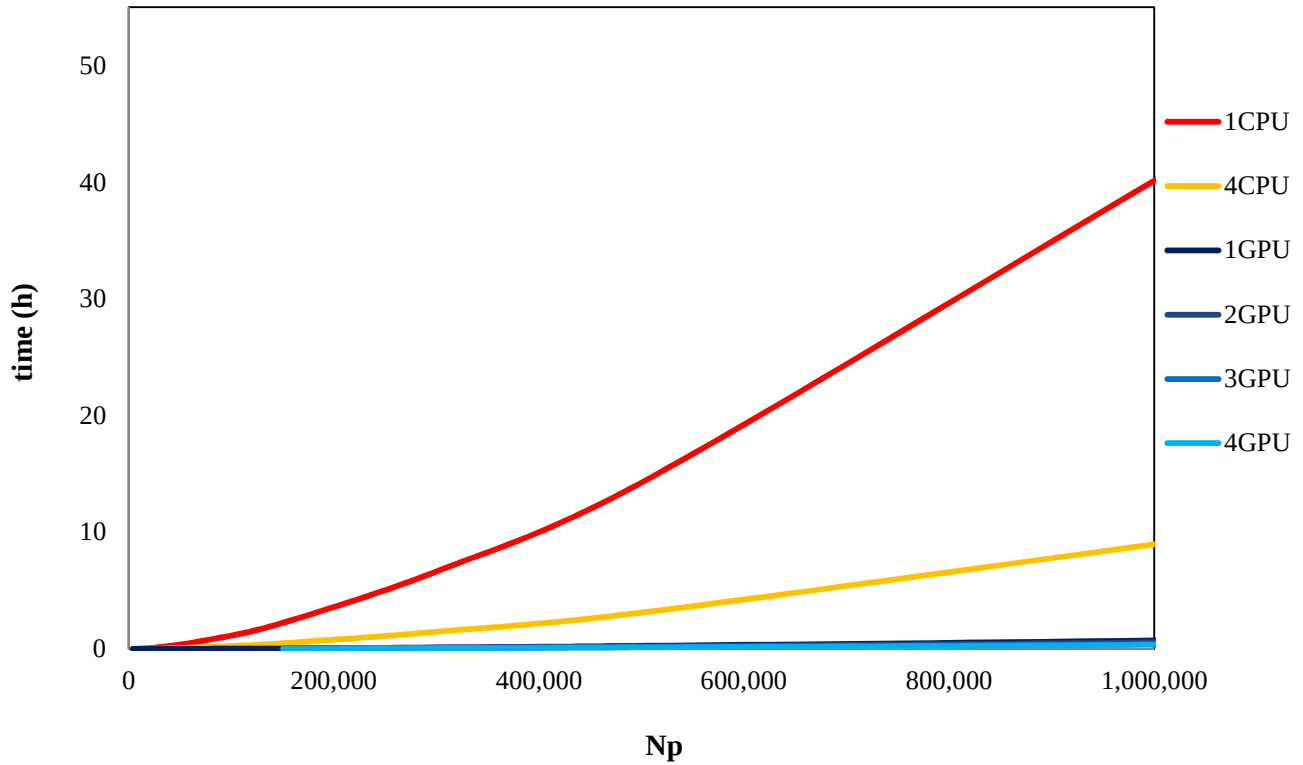
9 h

45 min

Computational runtimes with the Multi-CPU and GPU model

Multi-GPU implementation

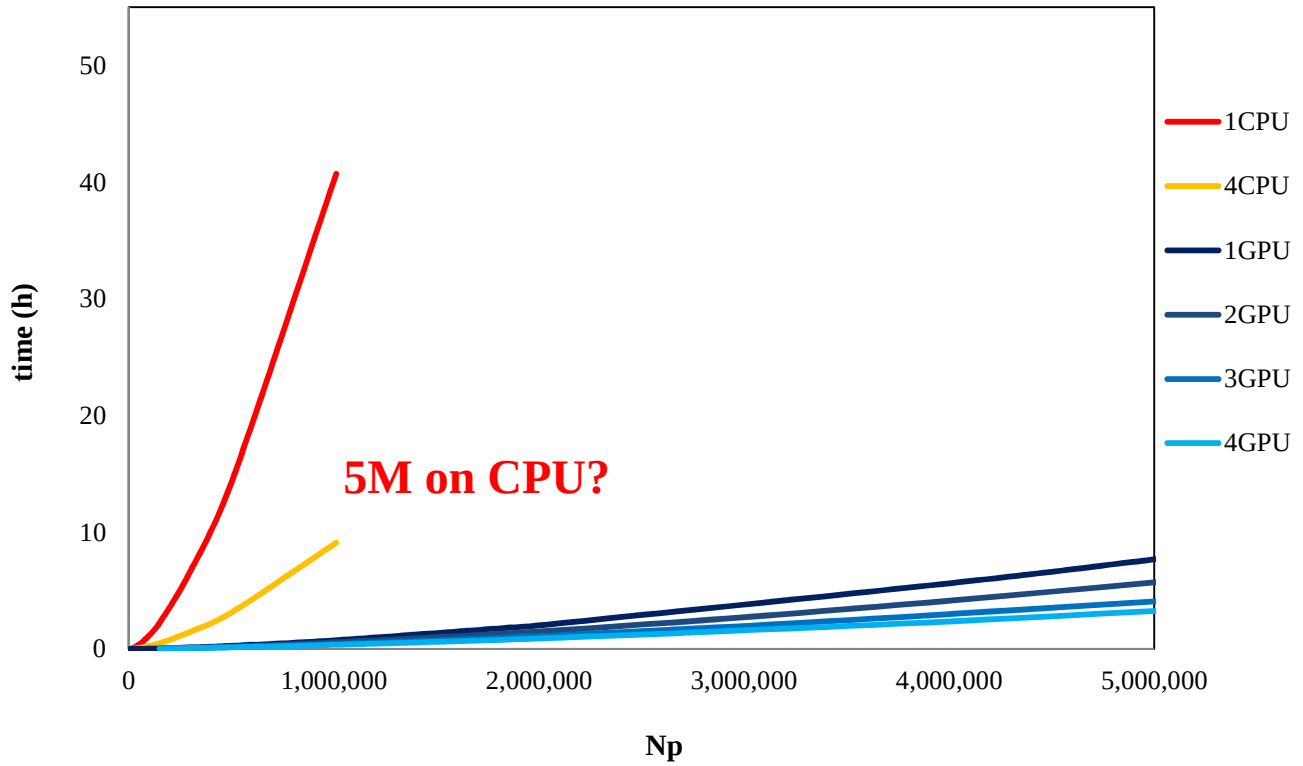
CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
GPU GTX 480 at 1.40GHz with 480 cores



Computational runtimes with the Multi-CPU and GPU model

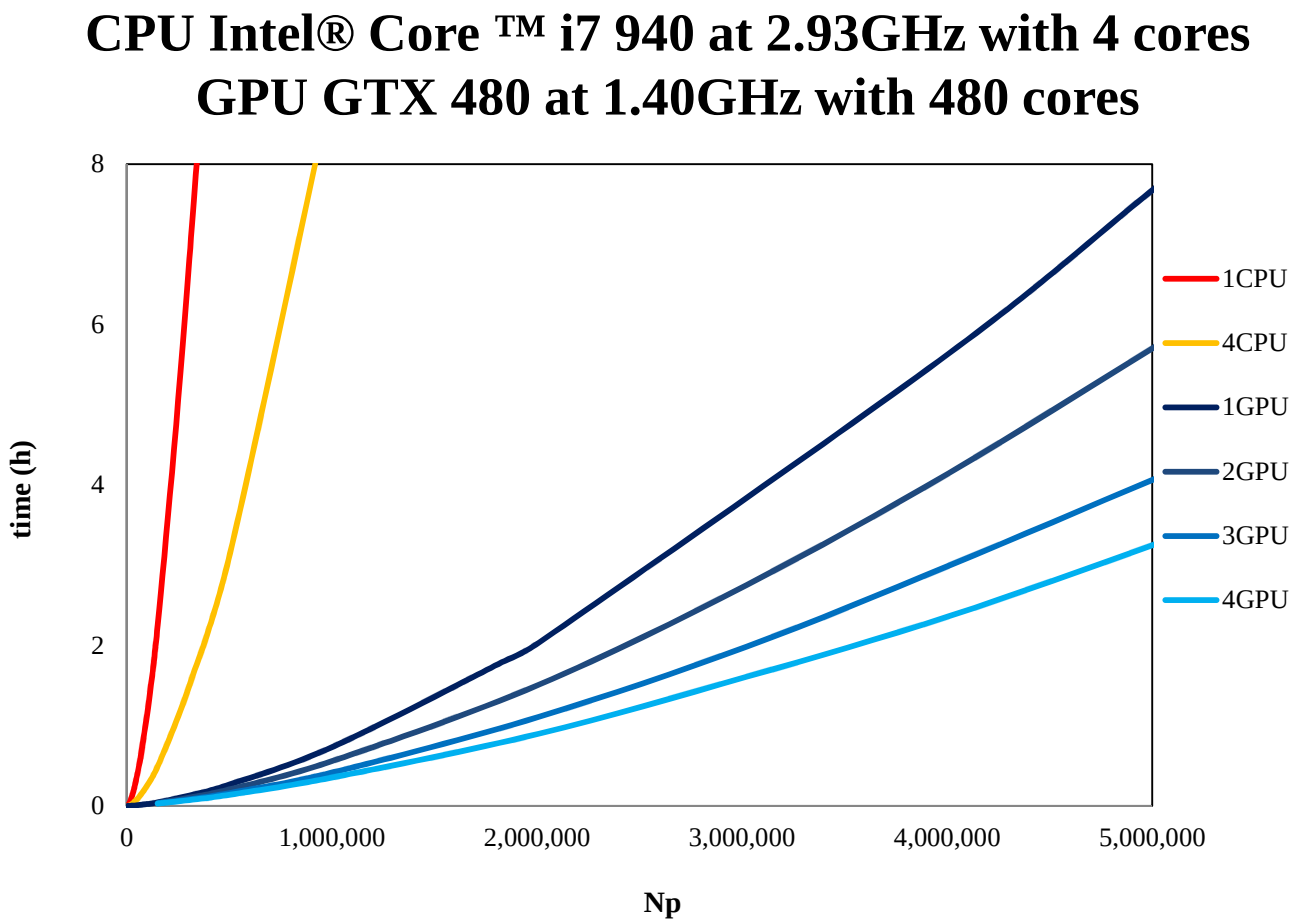
Multi-GPU implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
GPU GTX 480 at 1.40GHz with 480 cores



Computational runtimes with the Multi-CPU and Multi-GPU model

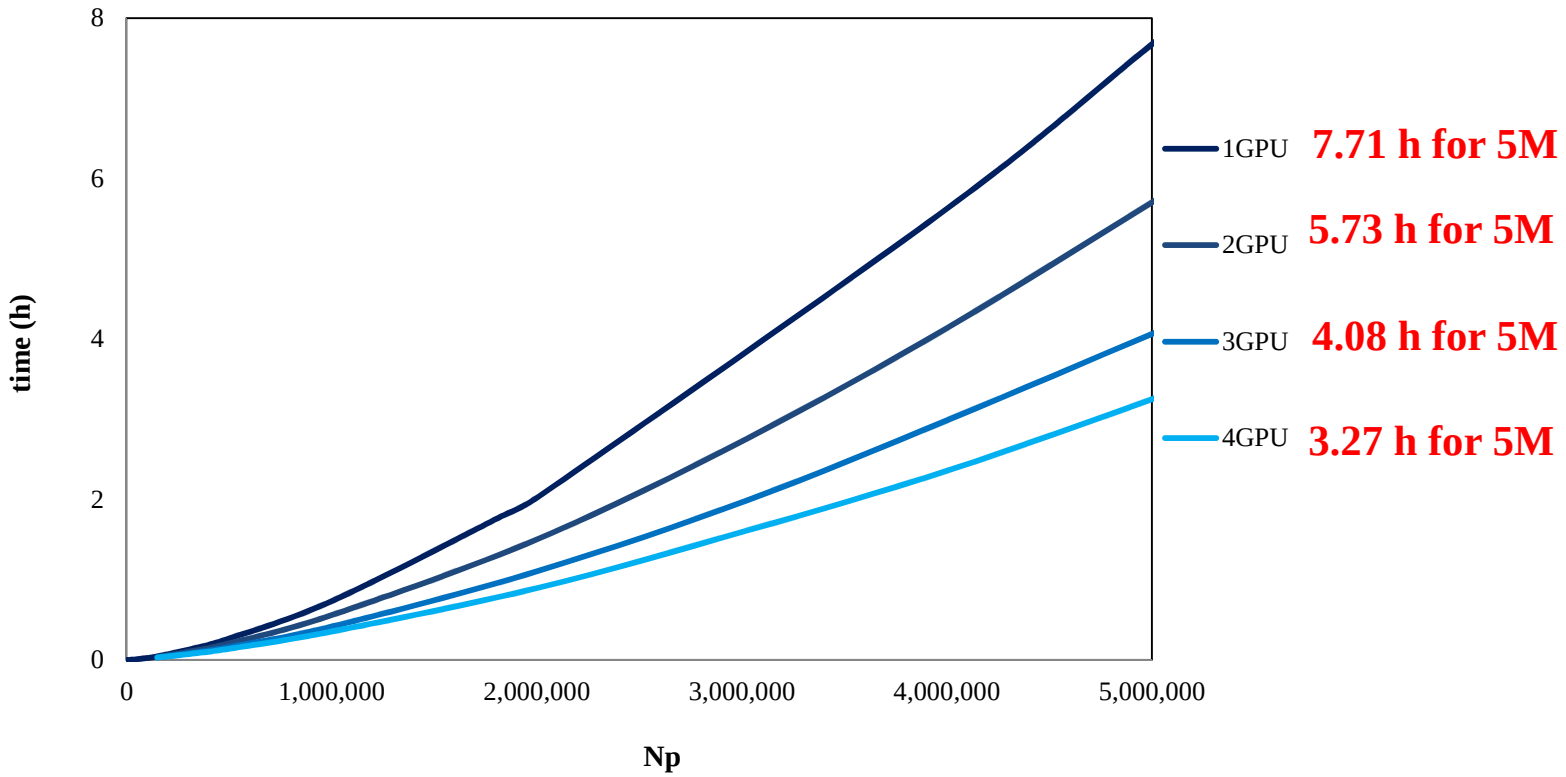
Multi-GPU implementation



Computational runtimes with the Multi-CPU and Multi-GPU model

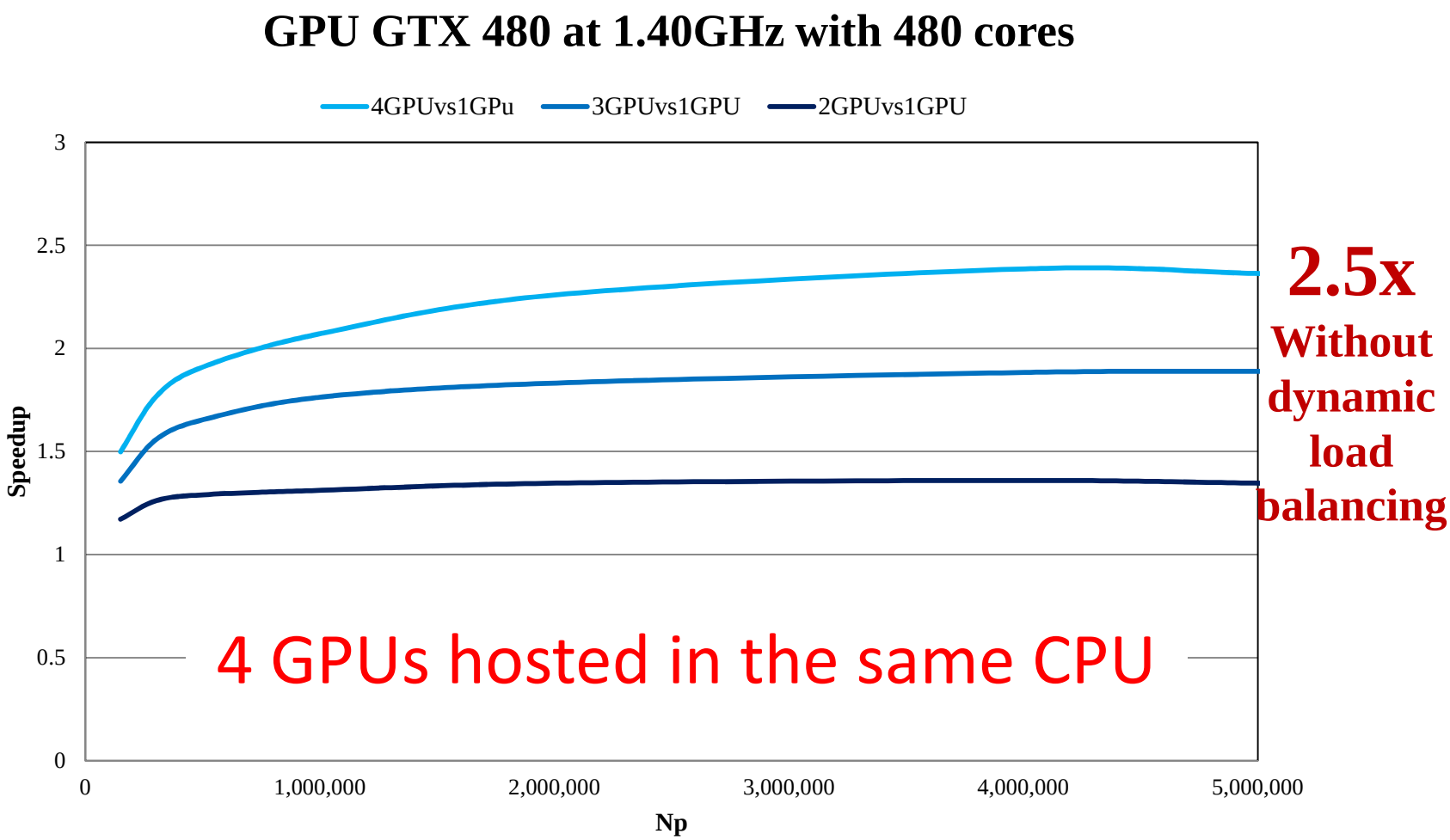
Multi-GPU implementation

GPU GTX 480 at 1.40GHz with 480 cores



Computational runtimes with the Multi-GPU model

Multi-GPU implementation



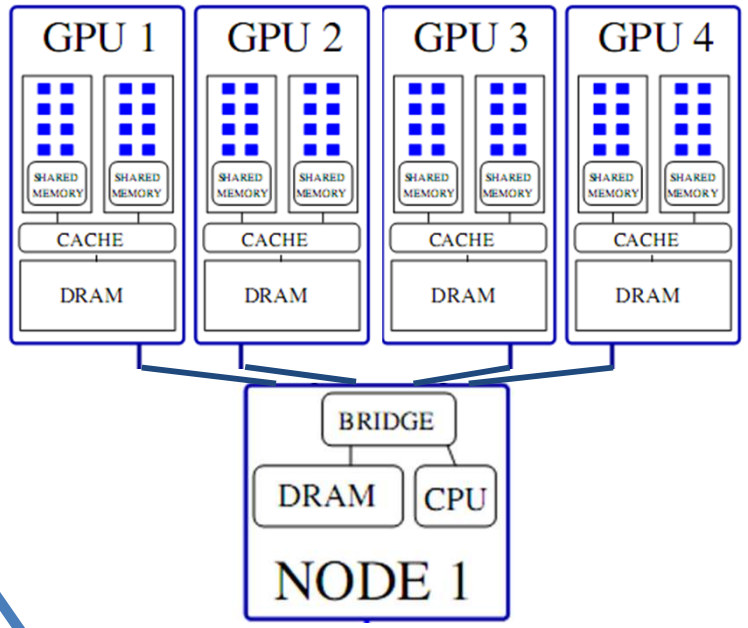
Speedup of using the Multi-GPU model

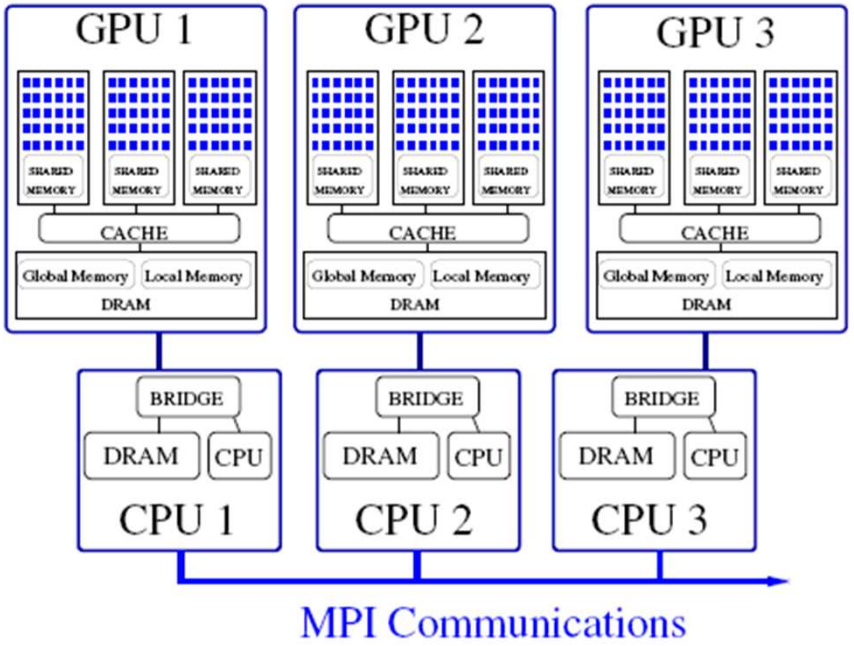
Multi-GPU implementation

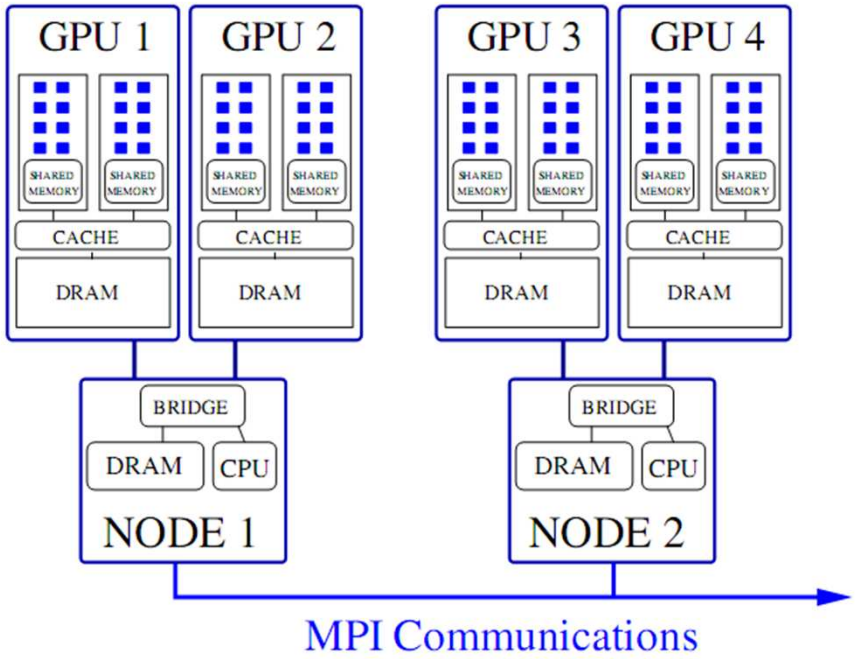
4 GPUs hosted in the same CPU

4 GPUs hosted in 2 CPUs (2 per node)

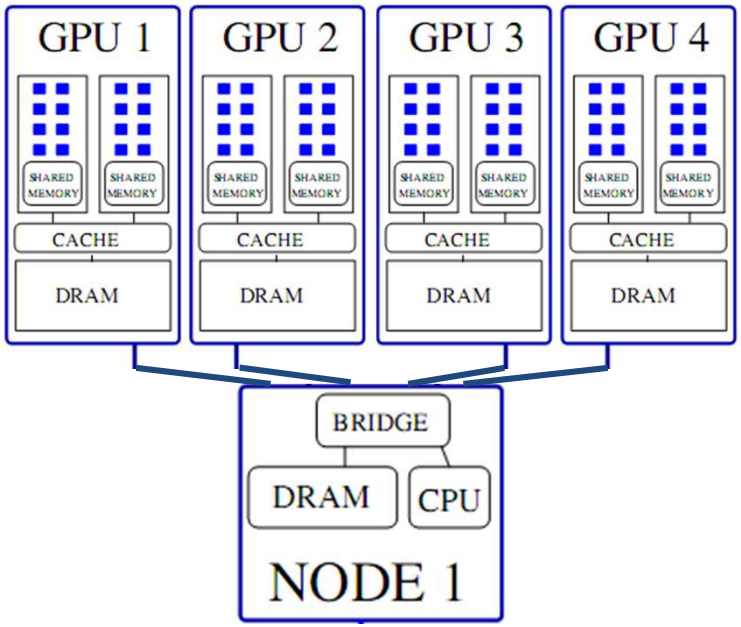
3 GPUs hosted in 3 different CPUs







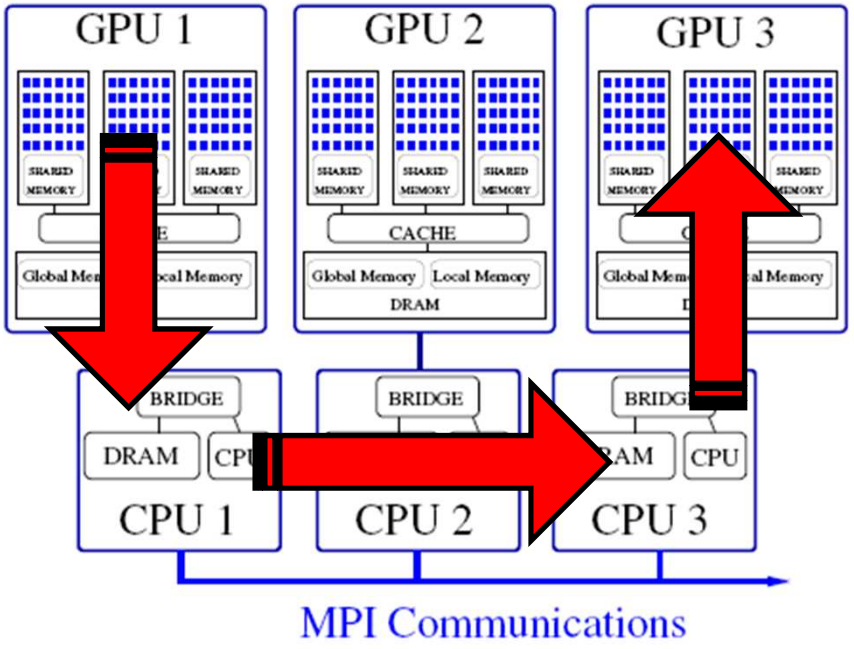
Multi-GPU implementation

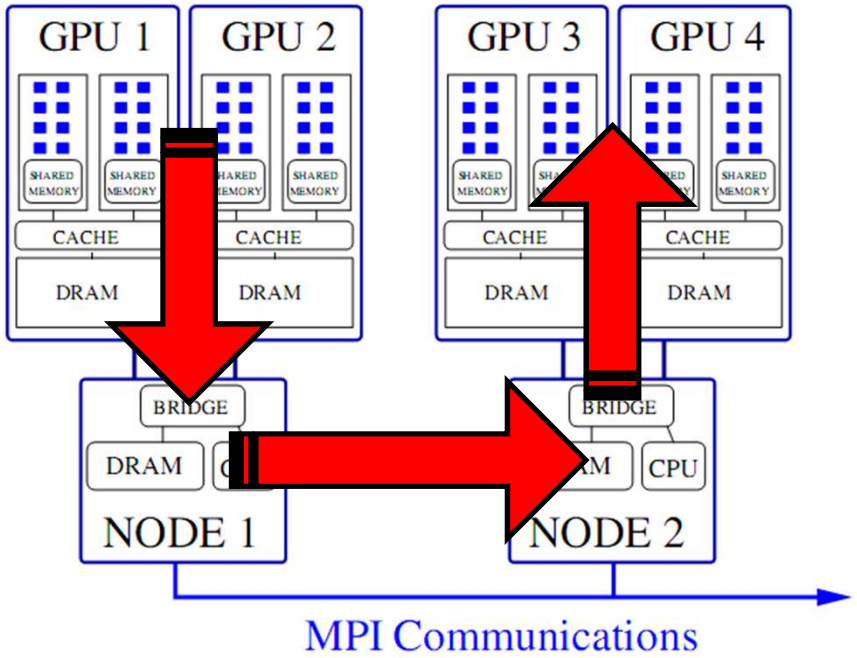


4 GPUs hosted in the same CPU

no cost in the inter-CPU communications

Multi-GPU implementation





IF GPUs hosted in different CPUs

With the CUDA v4.0, direct GPU-GPU communication will be supported.

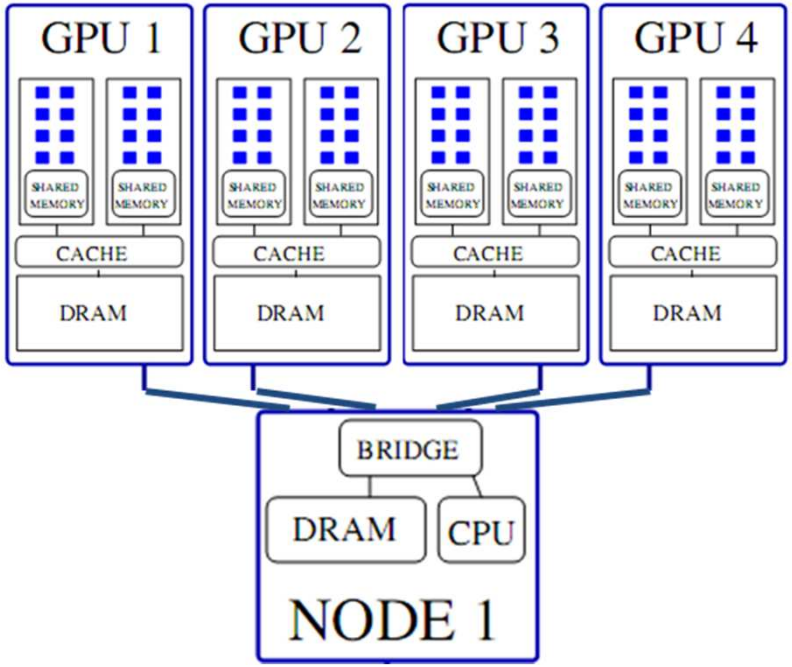
But now, the memory transfer between different GPUs is carried out by

GPU-CPU

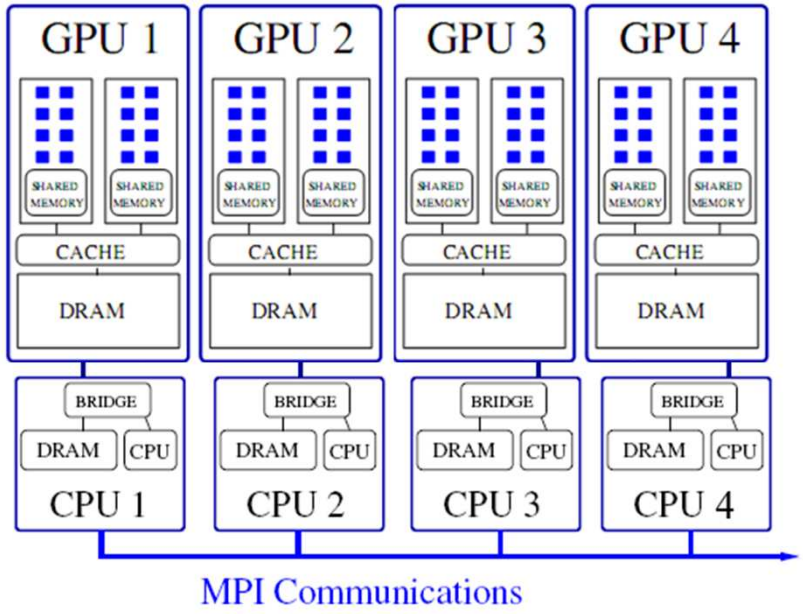
CPU-CPU using MPI

CPU-GPU communications

Multi-GPU implementation

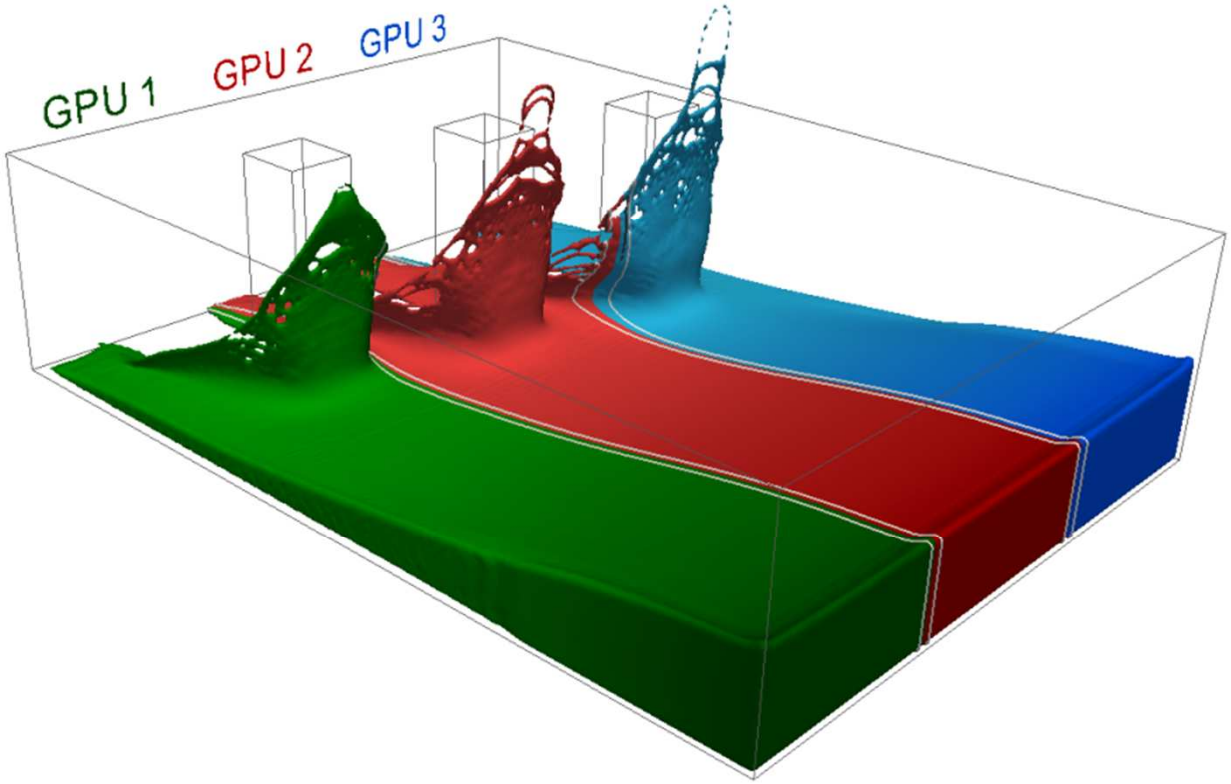


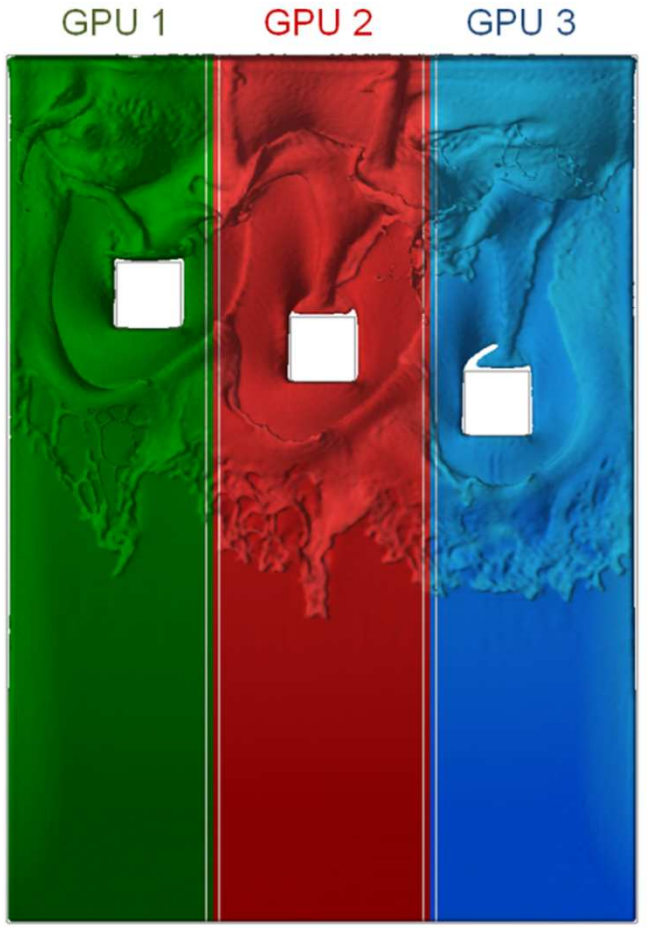
4 GPUs hosted in the same CPU



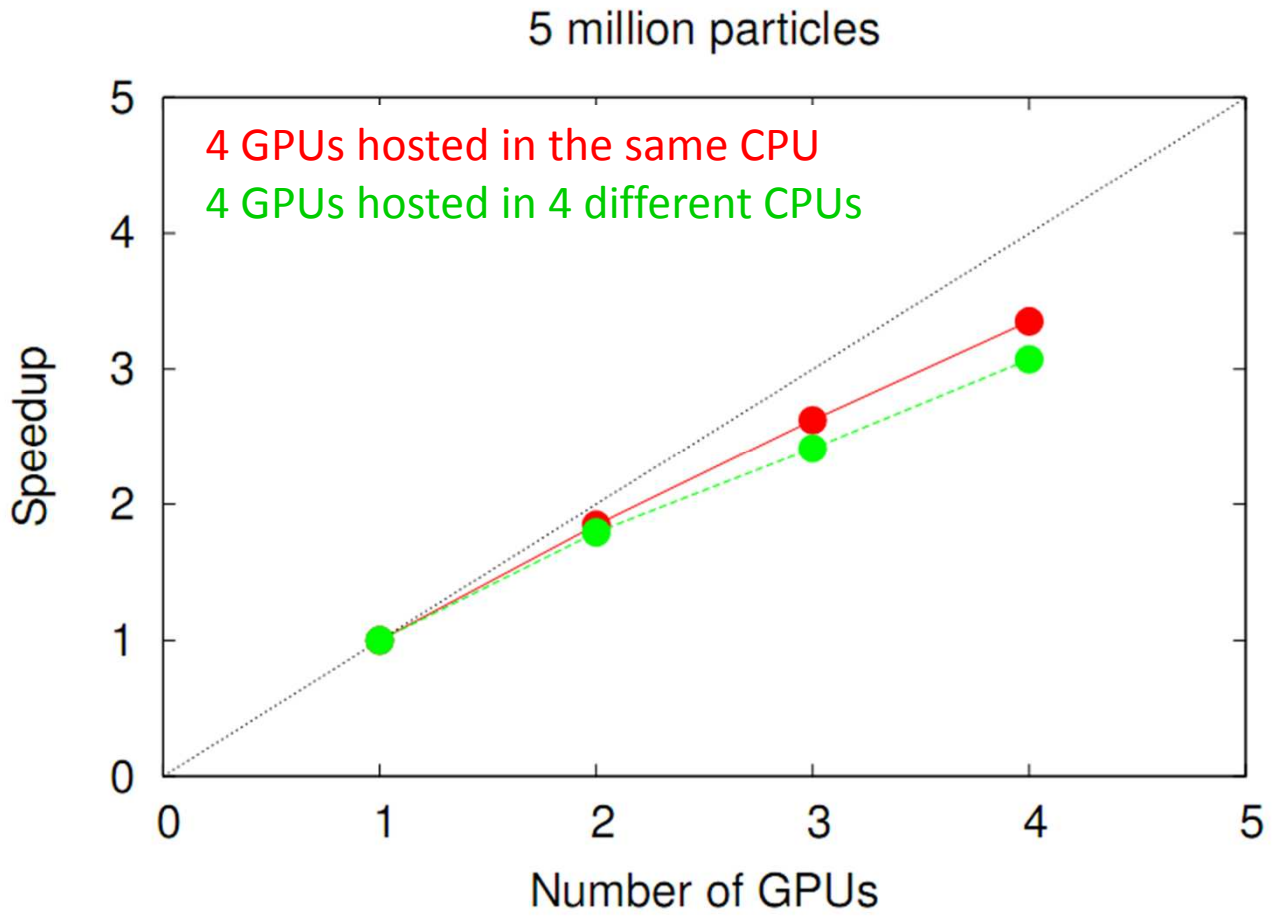
4 GPUs hosted in 4 different CPUs

Multi-GPU implementation



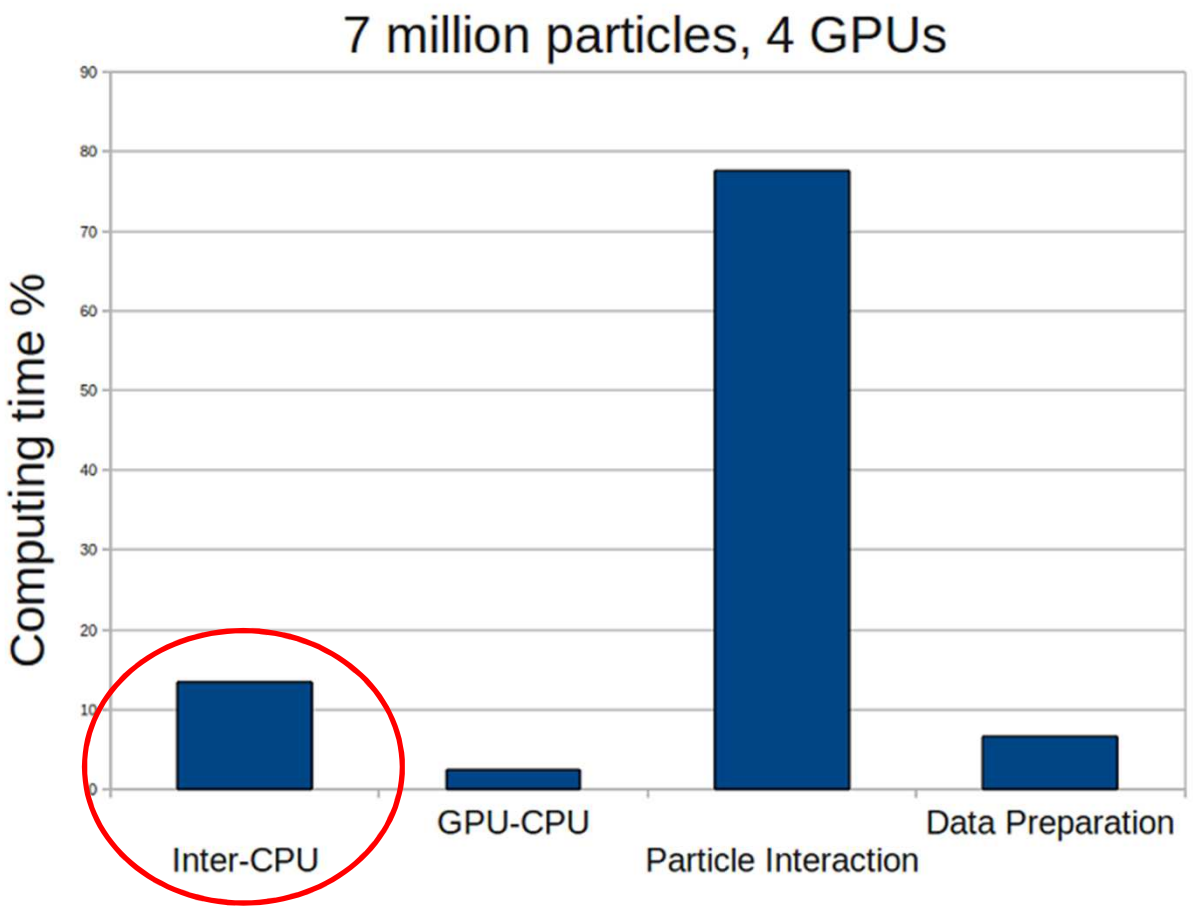


Multi-GPU implementation



cost in the inter-CPU communications

Multi-GPU implementation



inter-CPU cost increases with the number of particles

4 GPUs in 4 different CPUs

Multi-GPU implementation

CPU Intel® Core ™ i7 940 at 2.93GHz with 4 cores
GPU GTX 480 at 1.40GHz with 480 cores

1M	hours	Speedup vs. 1 CPU	Speedup vs. 4 CPU
1CPU	40.71	**1.00**	
4CPU	9.09	4.48	**1.00**
1GPU	0.75	54.61	12.19
2GPU	0.57	71.63	15.99
3GPU	0.42	96.33	21.51
4GPU	0.36	113.22	25.28

5,000 EUROS

Speedup of using the Multi-GPU model

Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

Applications

RENEWABLE WAVE ENERGY RESEARCH

Numerical tool to design the devices and to describe their behaviour

WAVE DRAGON

PELAMIS

POWERBUOY

Applications

COASTAL PROTECTION

The passage of storms near coastal areas gives rise to dangerous waves on the shore line.

Dangerous waves in San Sebastián coast, 2005

Storms effect in A Coruña coast, 2008

Applications

COASTAL PROTECTION

Natural disasters have occurred in the last years.

Hurricane Katrina in New Orleans, 2005

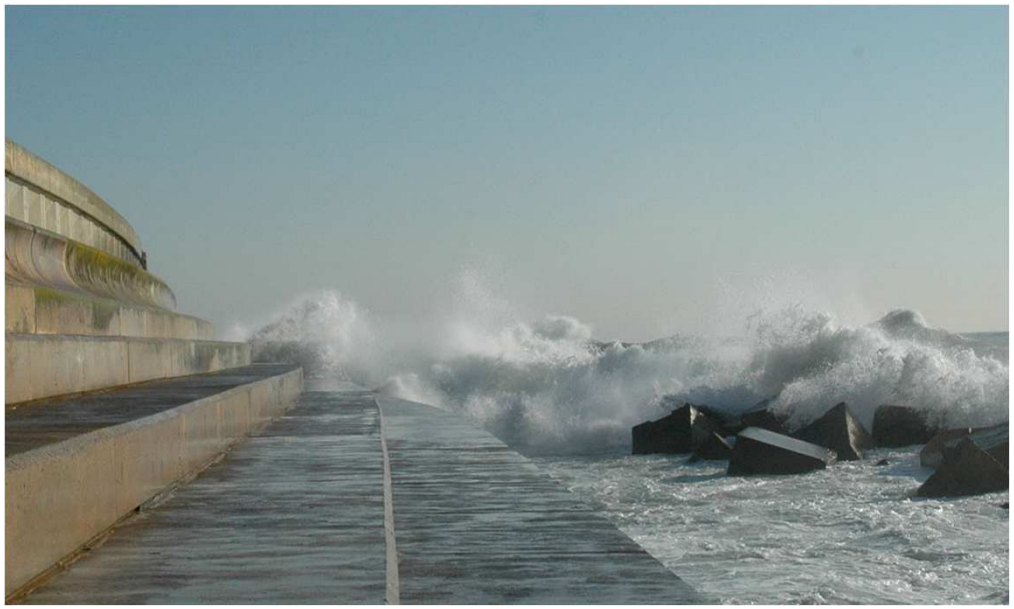
Tsunami in Japan, 2011

Applications

HARBOUR DESIGN

Real scenarios must be studied in detailed

Dikes overtopping



Port Olimpic in Barcelona

Applications

Coastal protection

Harbour design

Industrial applications

Coastal protection

Simulating million particles in a few hours allows us to investigate:

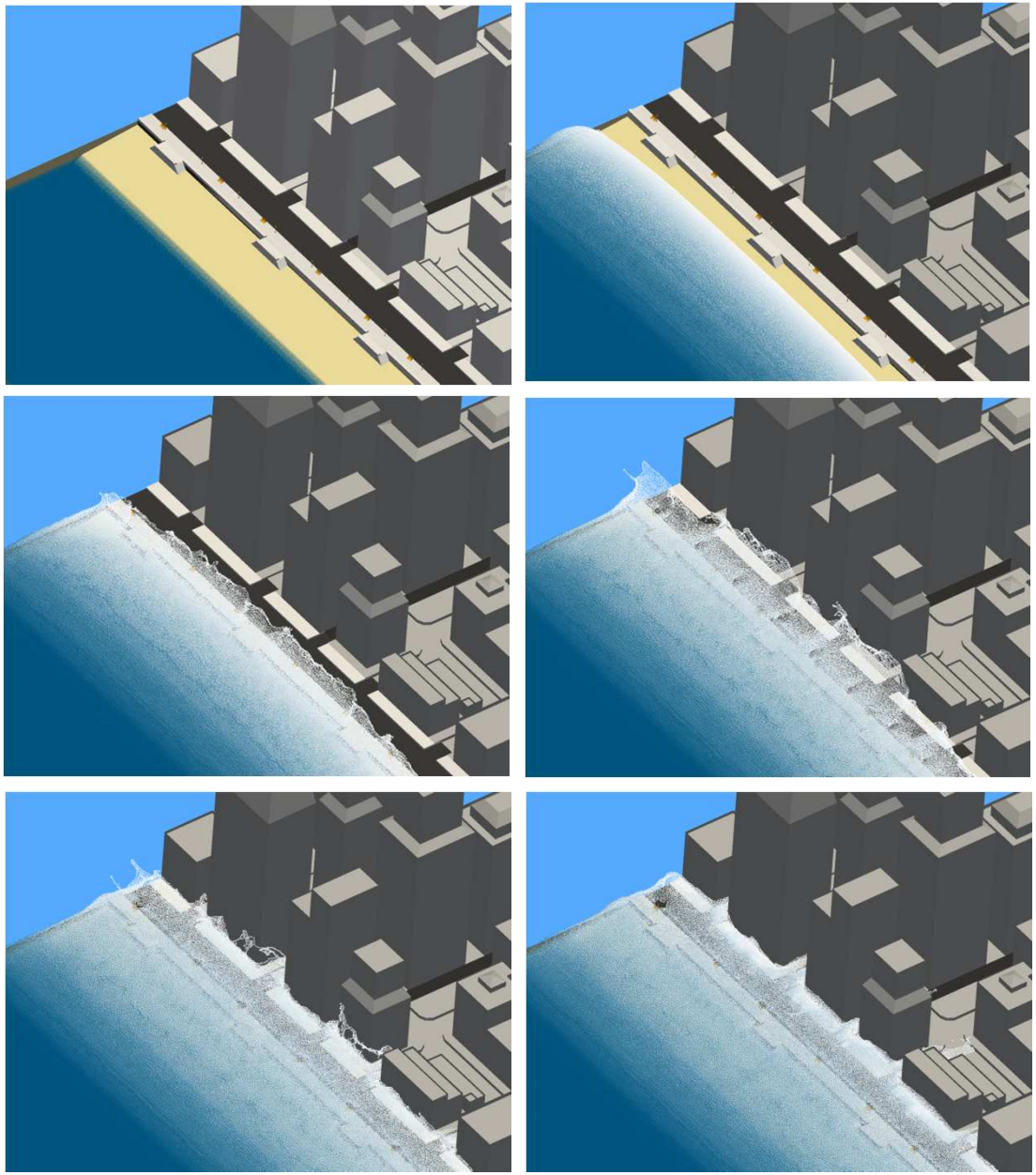
- **the damage due to extreme waves**
- **the flooded areas**
- **valuable information about overtopping**
- **risk maps in coastal areas**

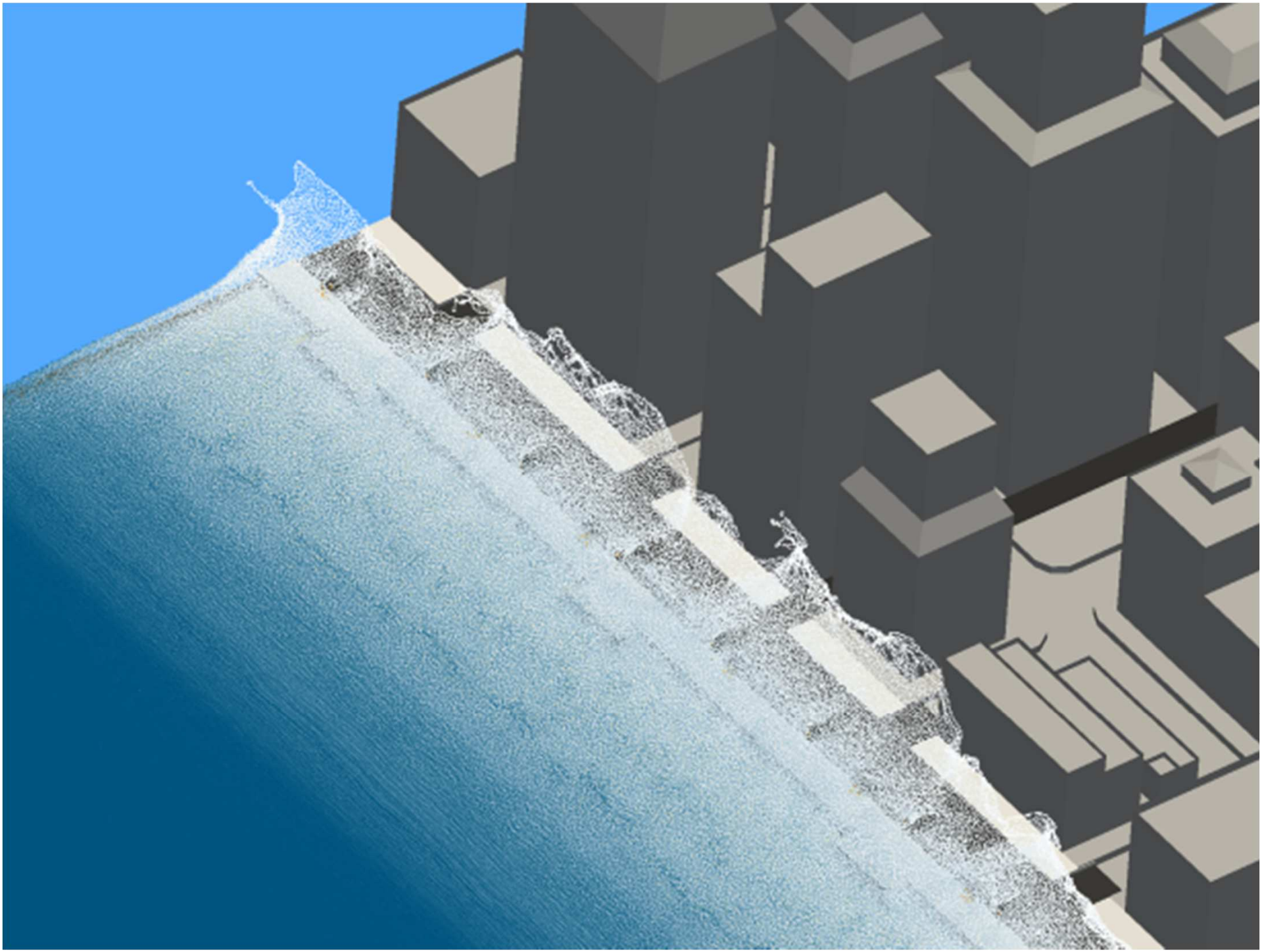
Coastal protection

Promenade-wave interaction with 5,342,325 particles

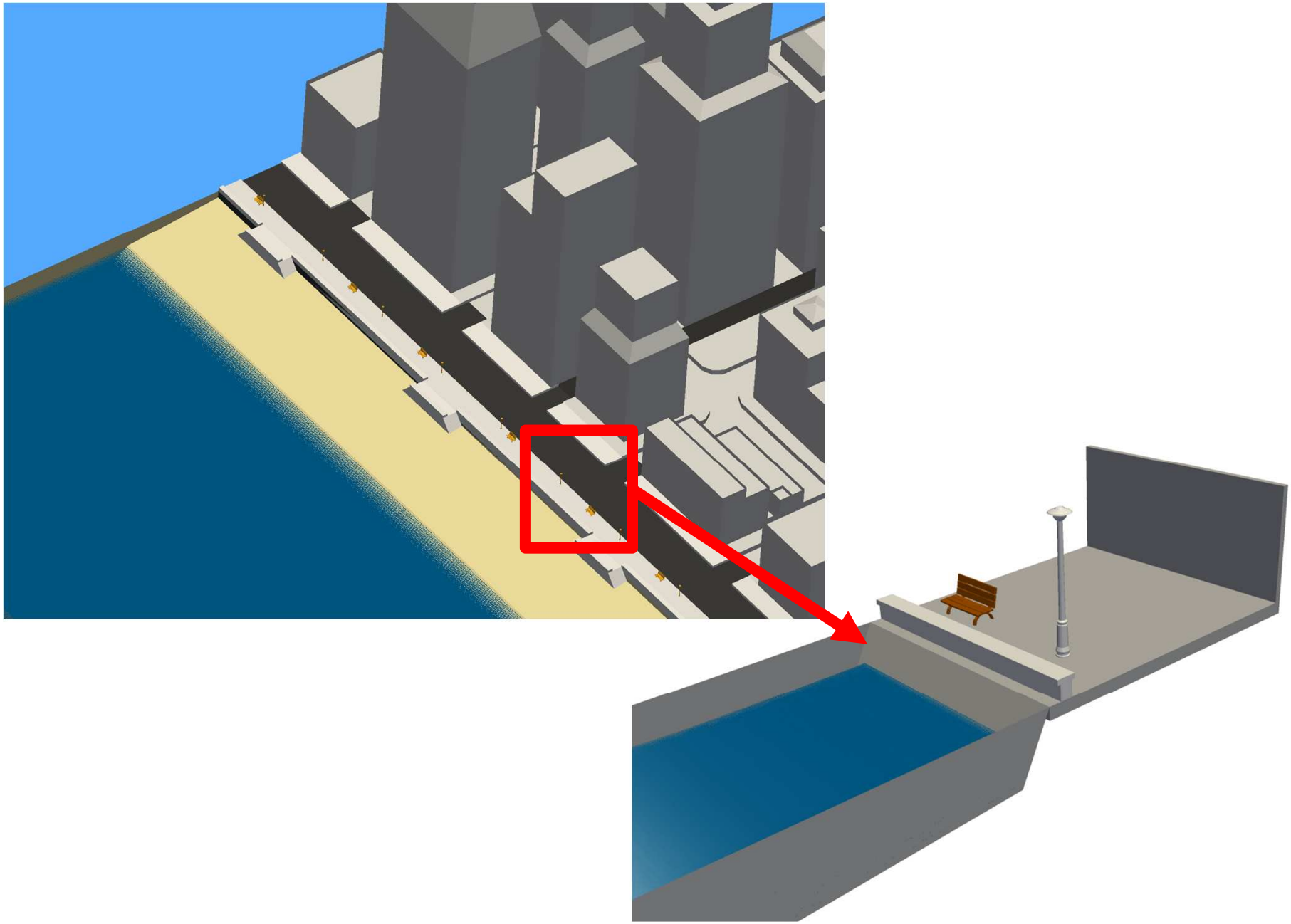
domain 600m x 600m x 450m
dp= 1.2m
h=1.8m

24 seconds of physical time
36,128 steps
take 3.4 hours on GTX480





Coastal protection

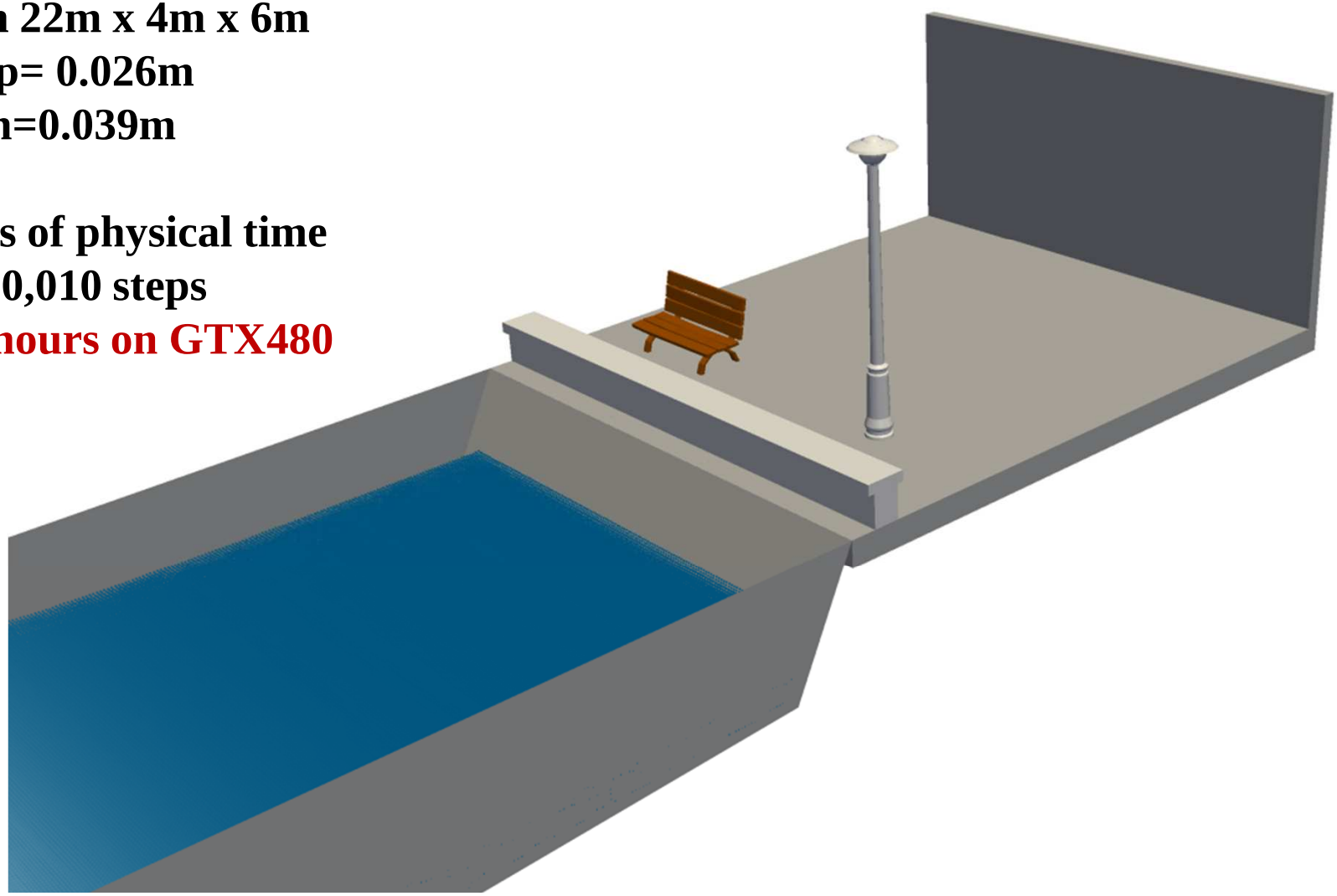


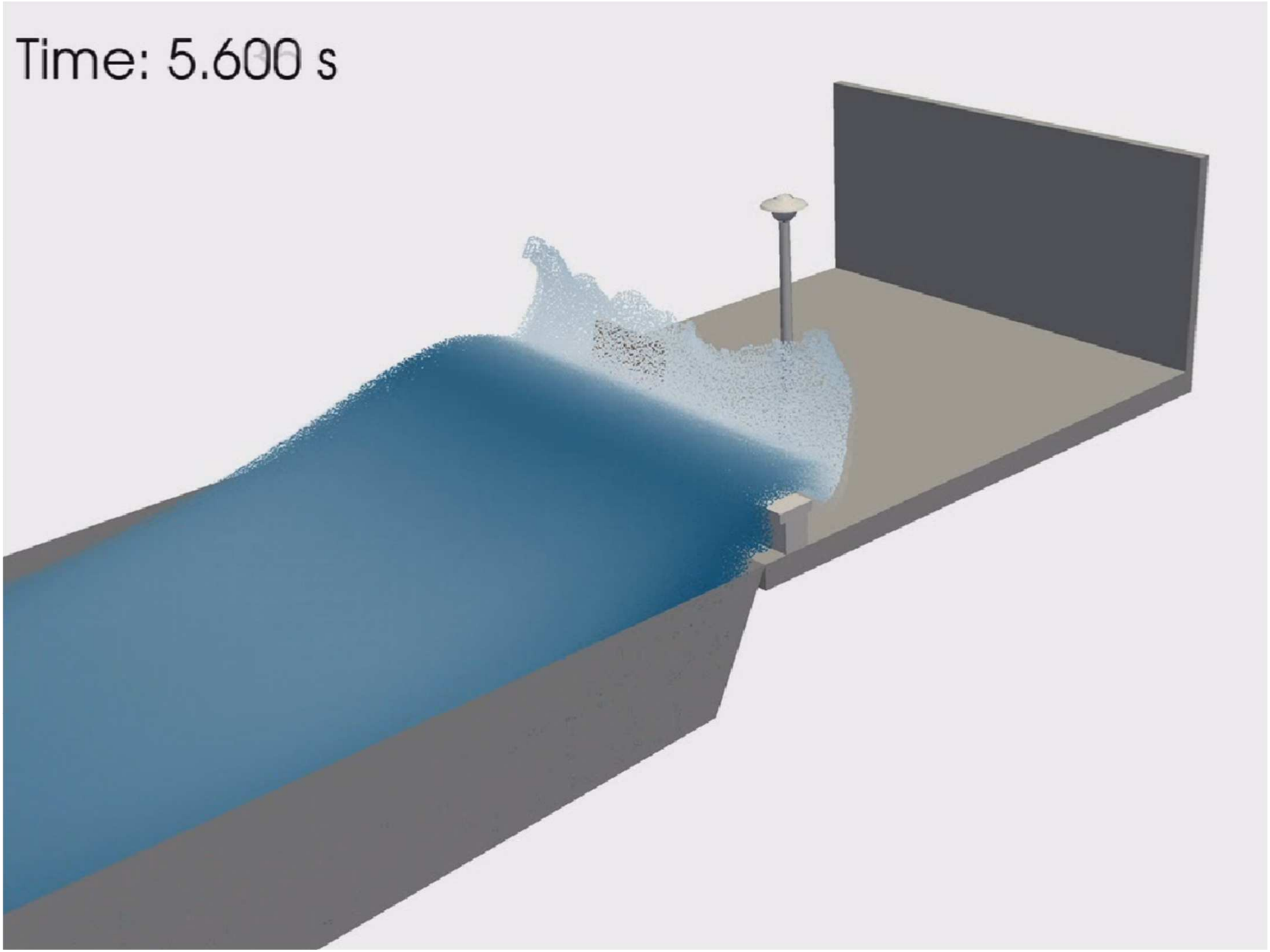
Coastal protection

Seawalk-wave interaction with 3,425,379 particles

domain 22m x 4m x 6m
dp= 0.026m
h=0.039m

8 seconds of physical time
120,010 steps
take 7.5 hours on GTX480

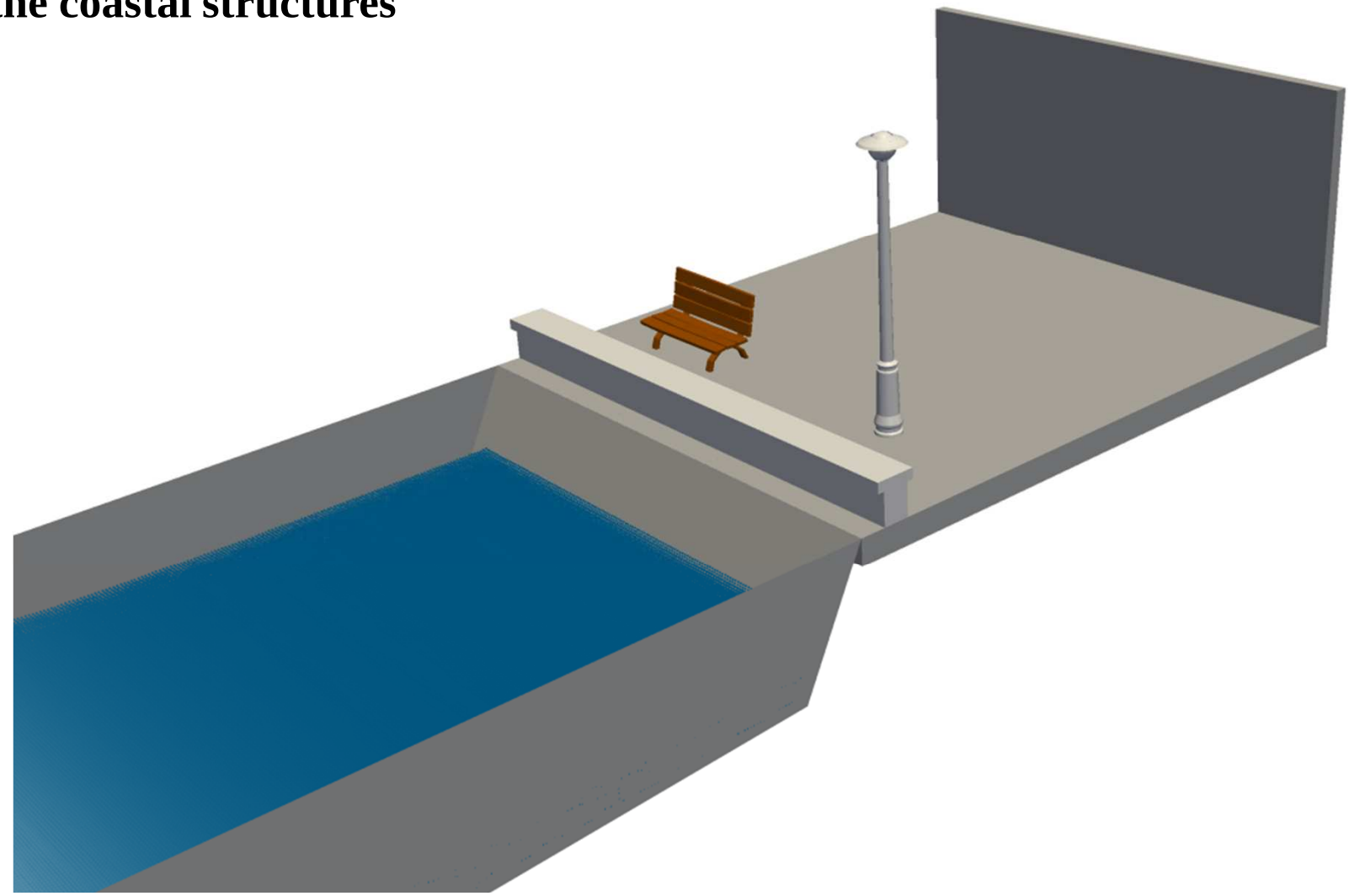



Time: 5.600 s

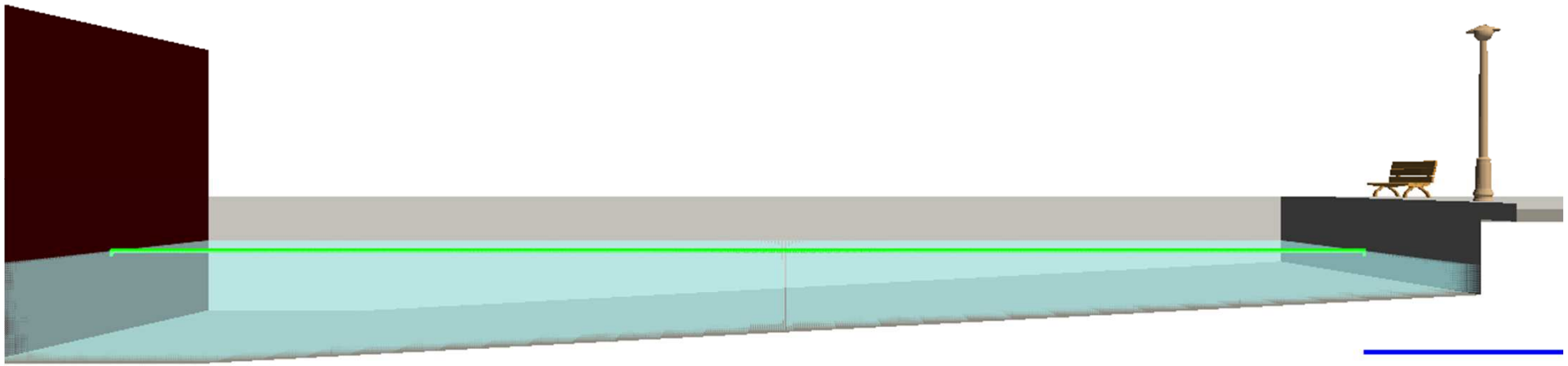
Coastal protection

We can measure the wave height

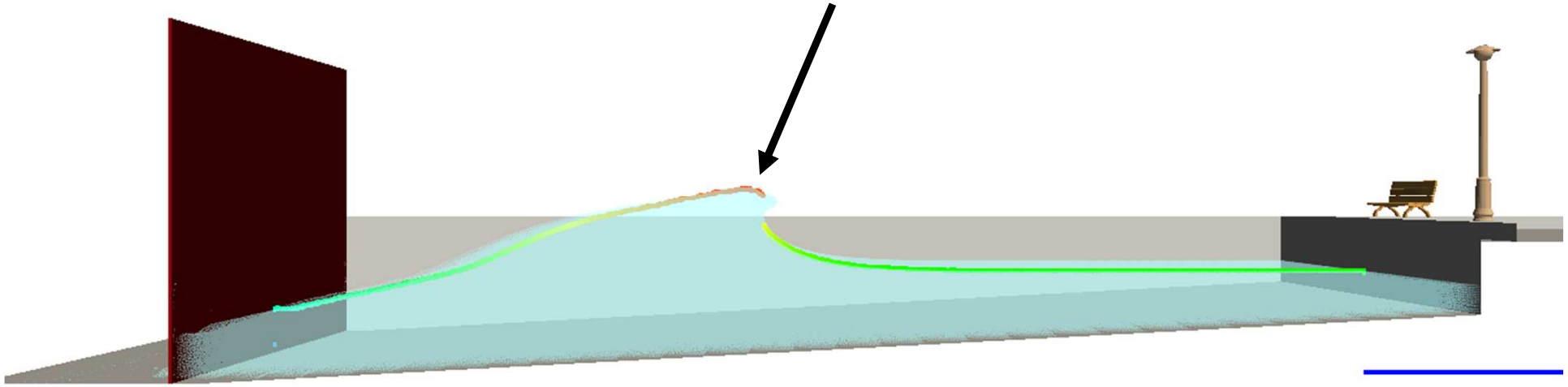
We can compute the forces exerted onto the coastal structures



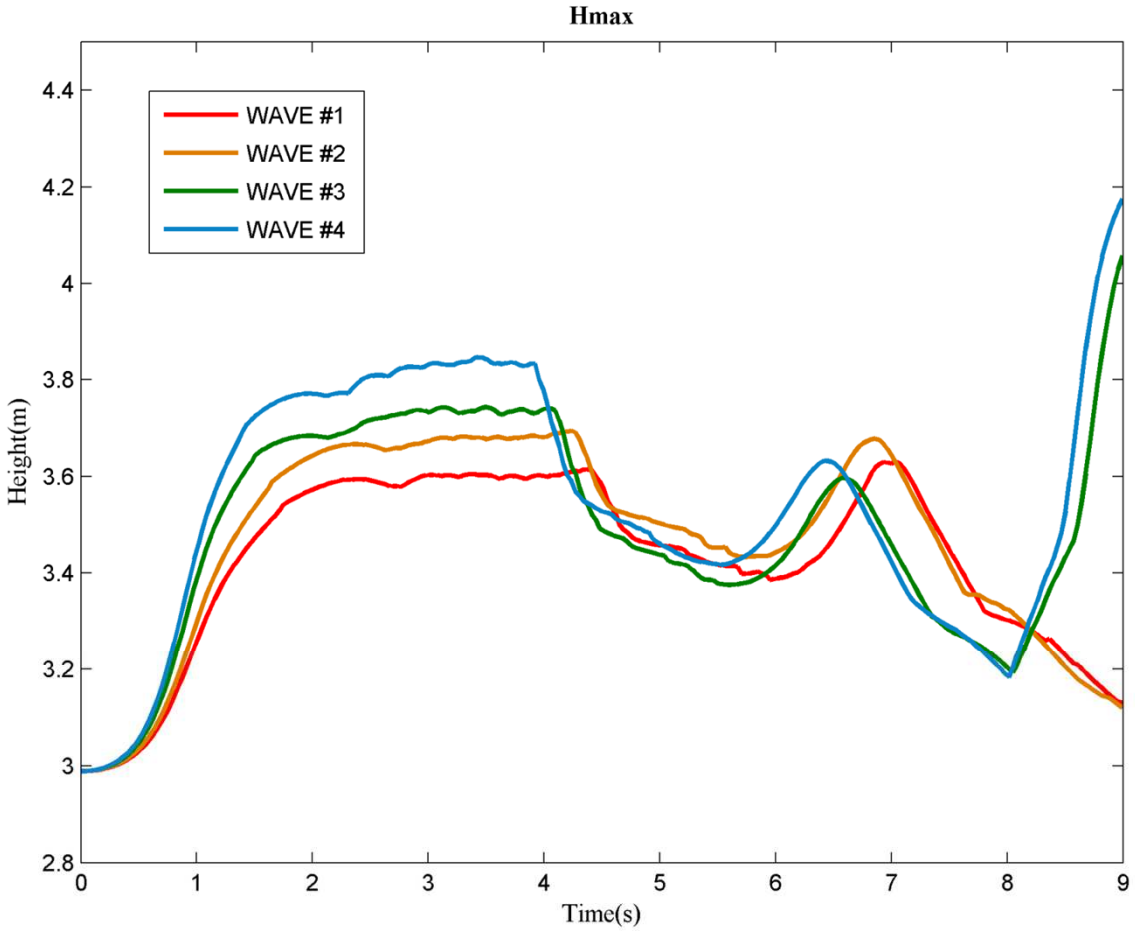
Coastal protection

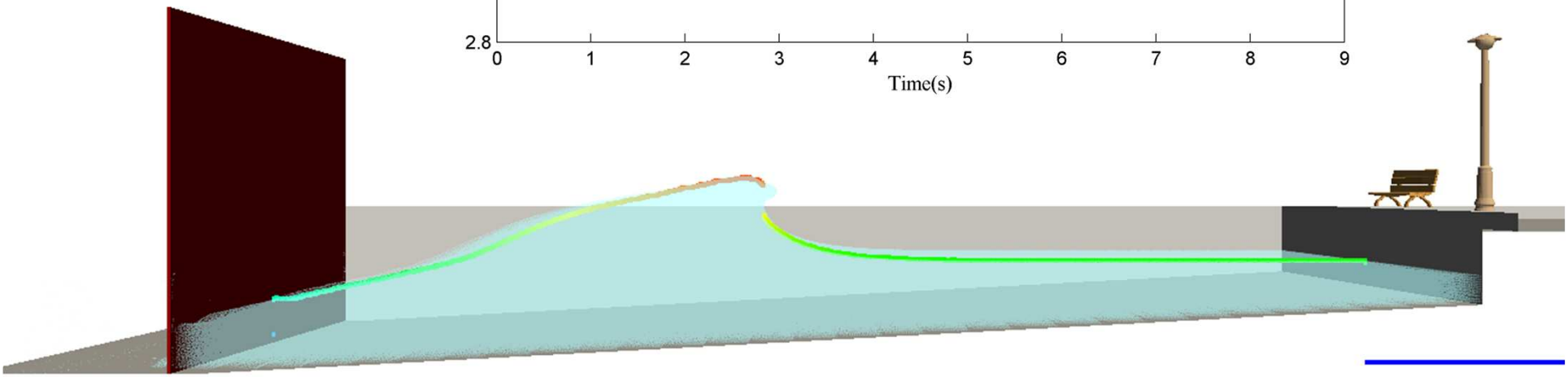


$H_s = 4\ m\ ;\ T_p = 2\ s$



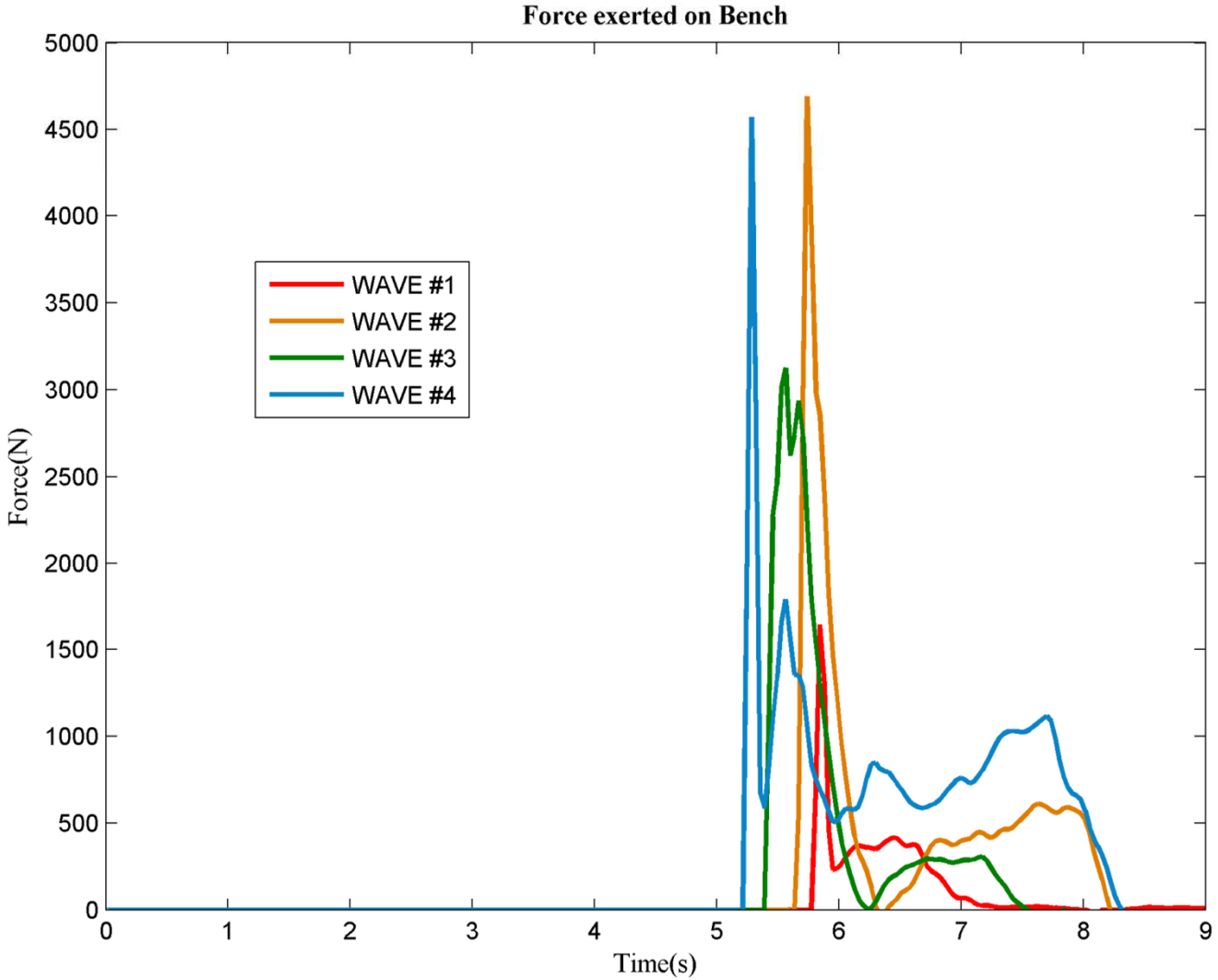
Coastal protection





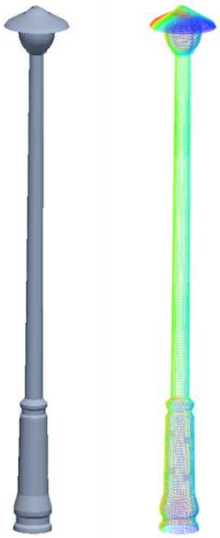
Coastal protection

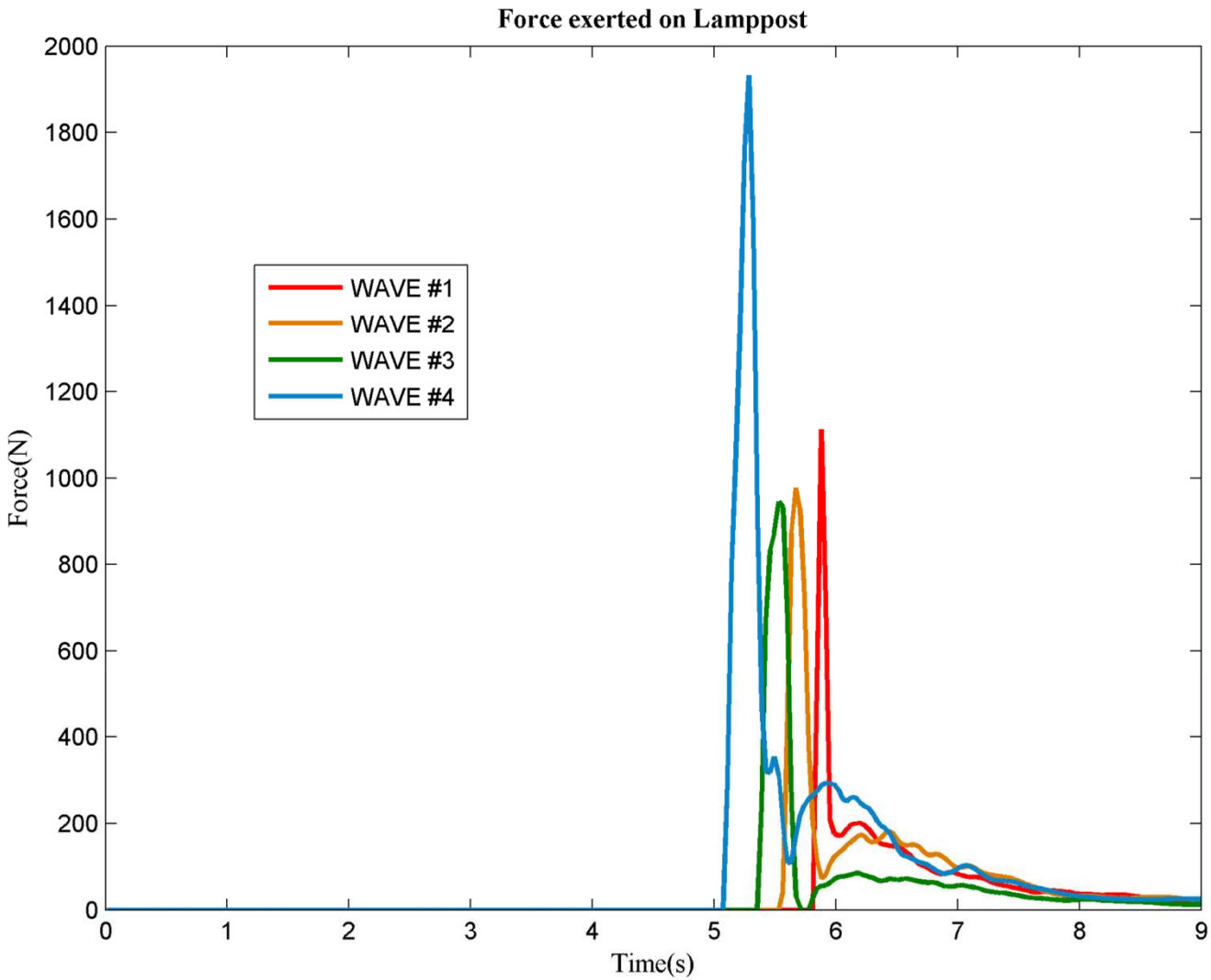
BENCH



Coastal protection

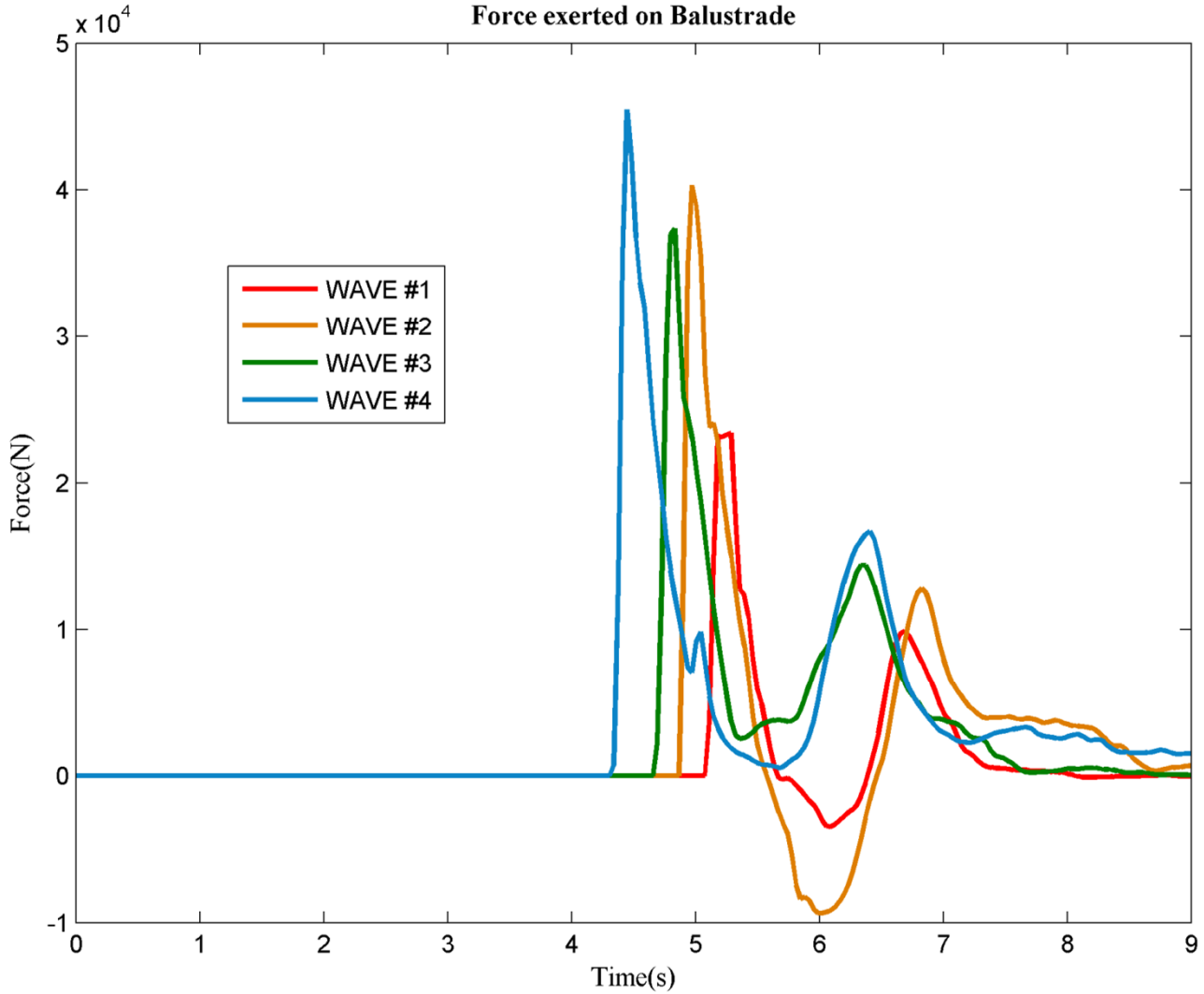
LAMPPOST





Coastal protection

BALUSTRADE

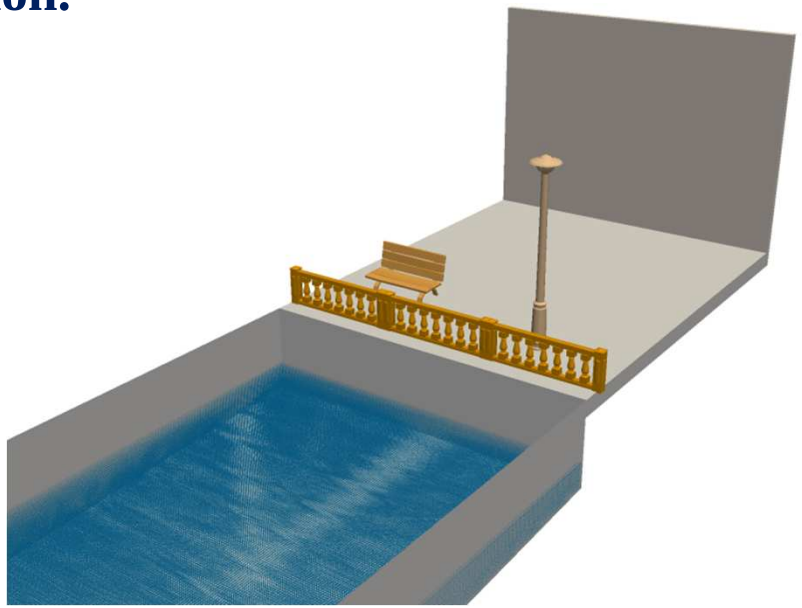


Coastal protection

Simulating million particles in a few hours allows us to investigate:

- **the damage due to extreme waves**
- **the flooded areas**
- **valuable information about overtopping**
- **risk maps in coastal areas**

Now we can simulate different H_s and T_p of the incoming waves and design the best scenerio for mitigation.

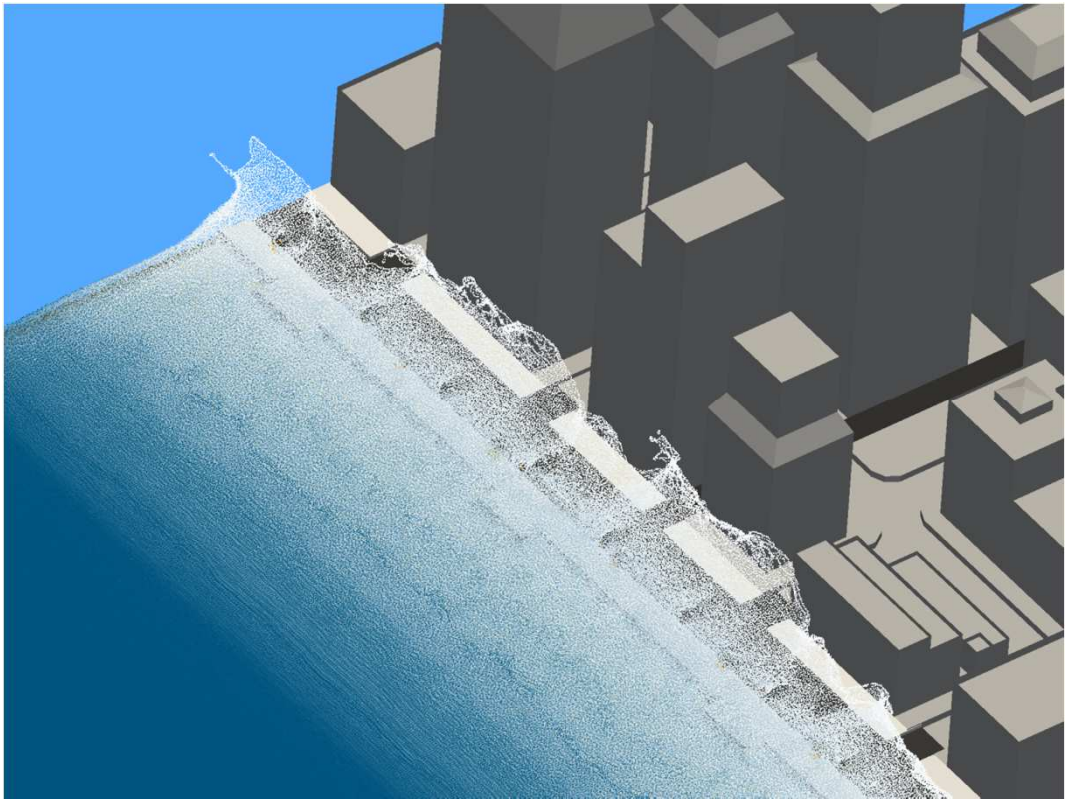


Coastal protection

Promenade-wave interaction with 5,342,325 particles

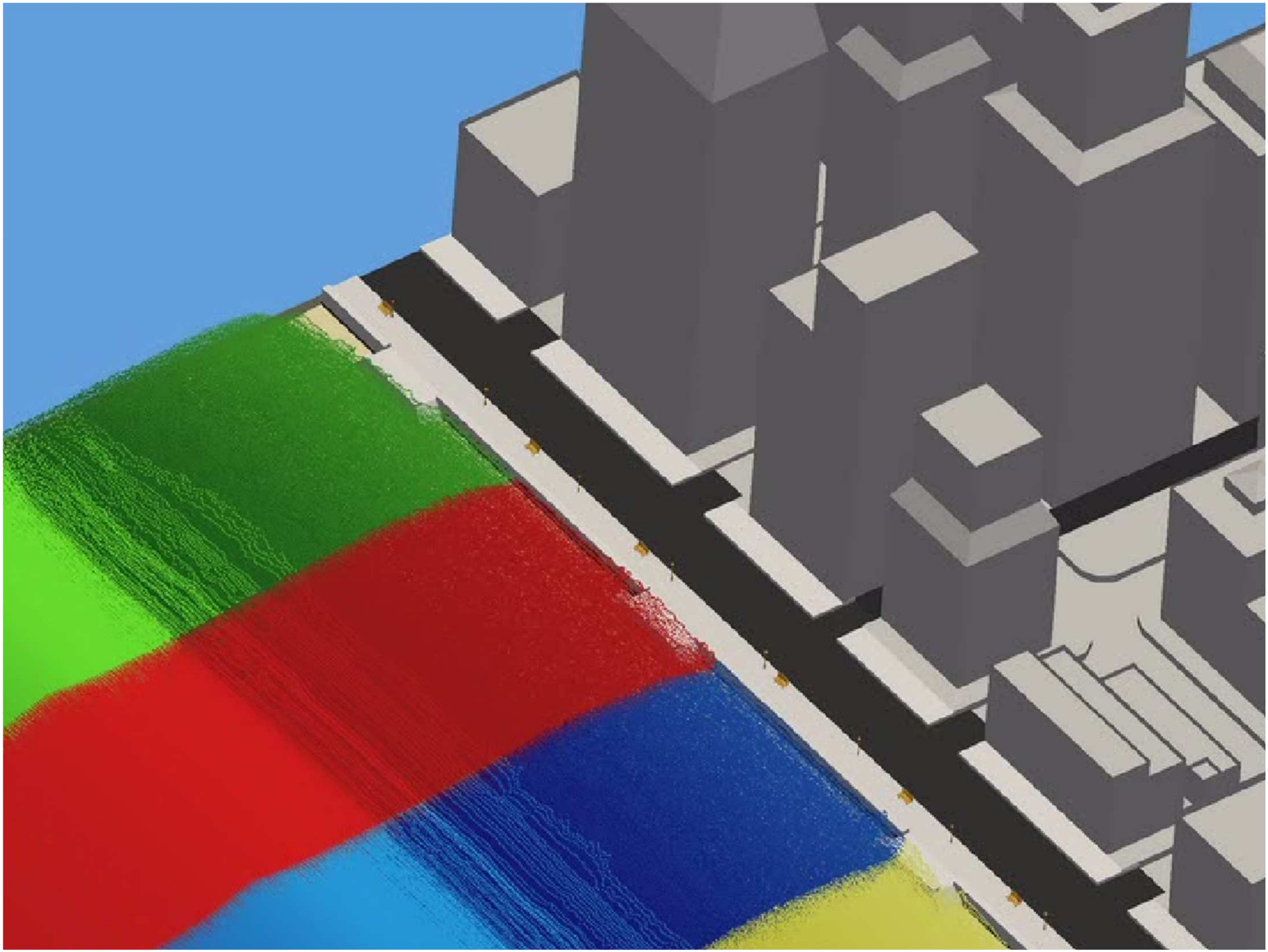
domain 600m x 600m x 450m
dp= 1.2m
h=1.8m

24 seconds of physical time
36,128 steps
take 3.4 hours on GTX480

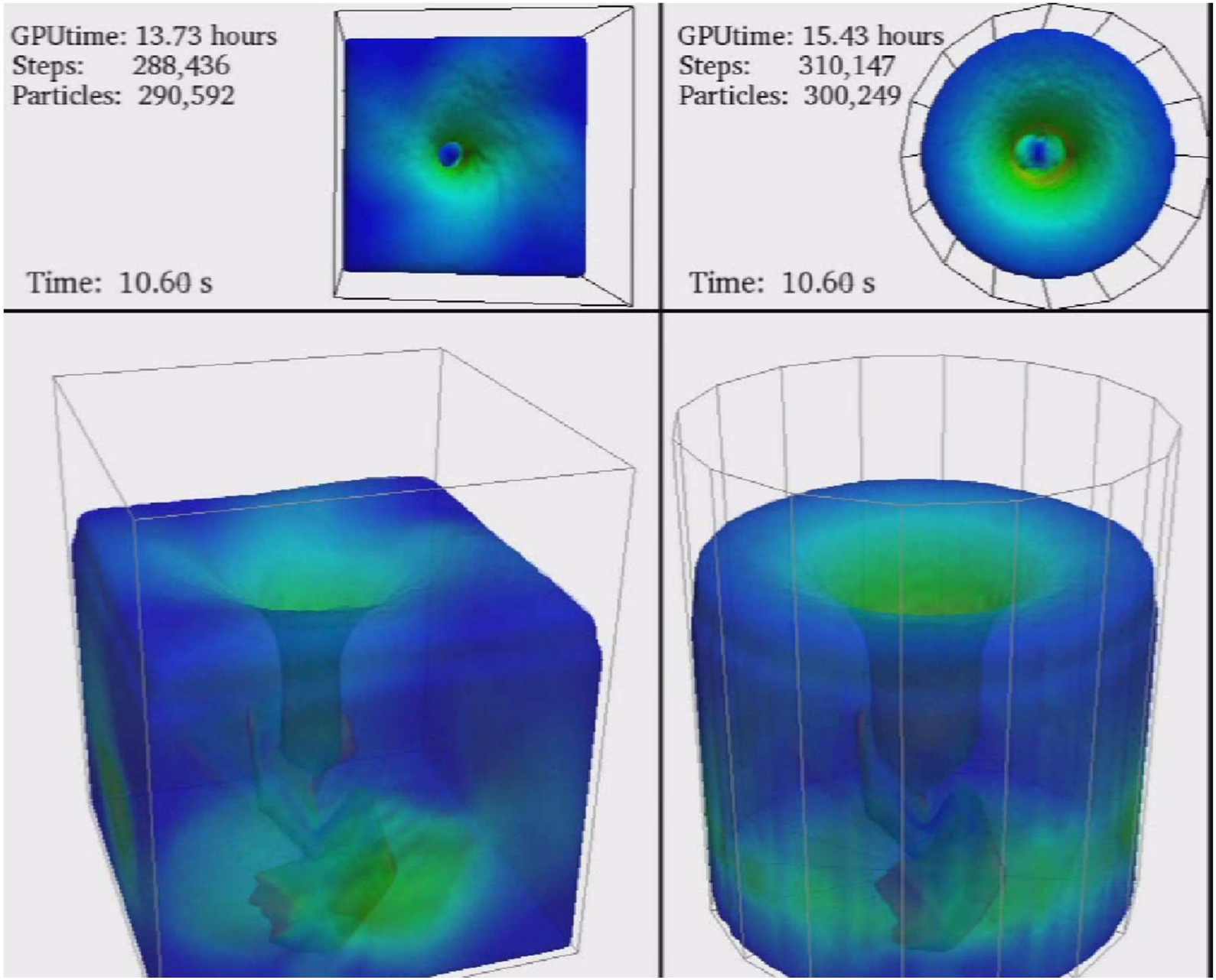


5,342,325 particles close to the maximum memory space of ONE GTX 480

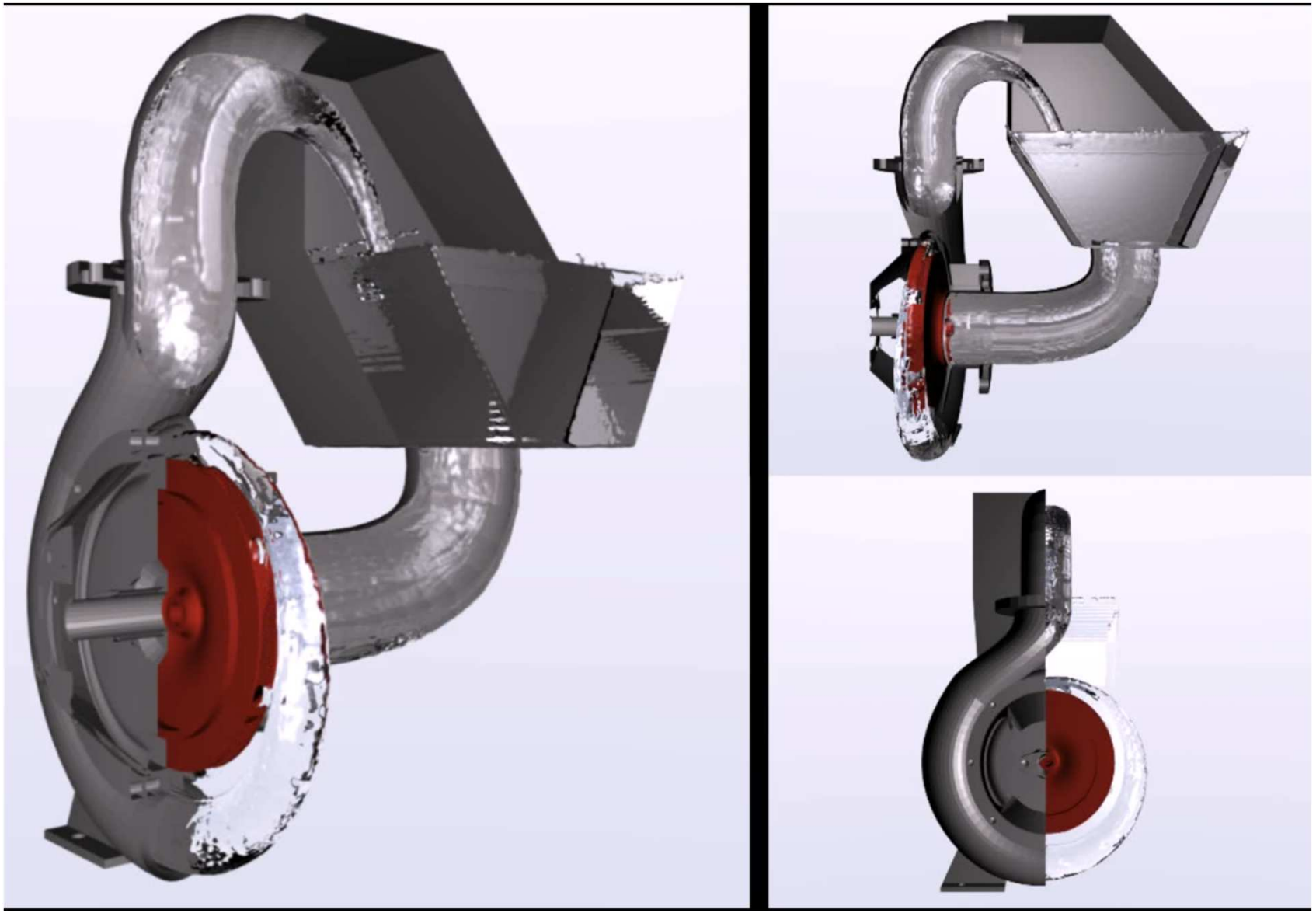
If we need h<1.8m, we will need multi-GPUs



Industrial applications



Industrial applications



Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

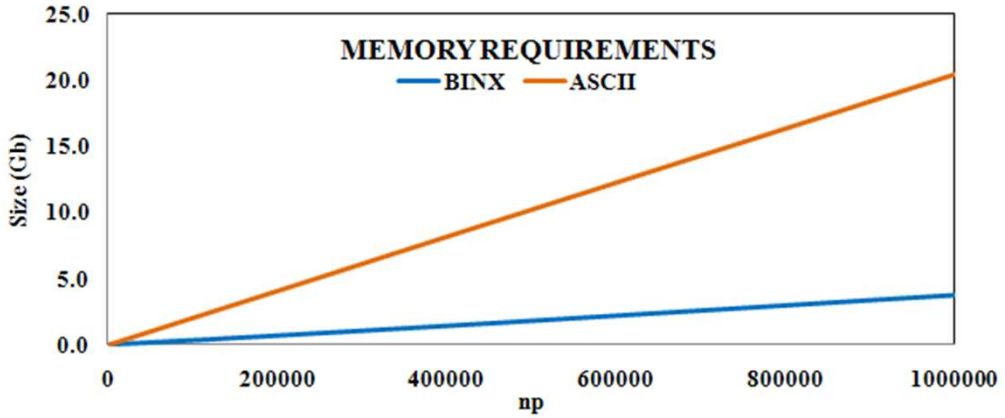
- Needs when accelerating the code: format files, pre/post-processing

- DualSPHysics code

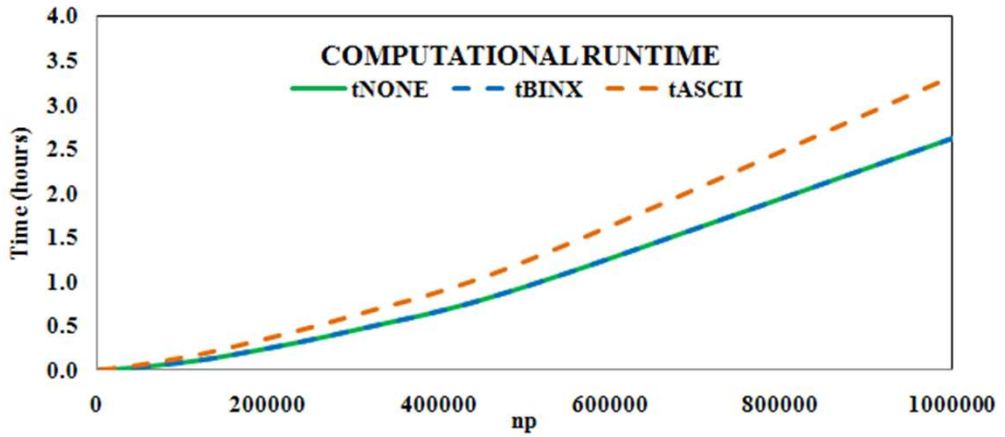
FILE FORMAT

Memory requirements and computational runtime for different output data formats.

150 files are saved for a physical time of 1.5 seconds.



Binary format BINX consumes less memory, reduction of 80% compared to ASCII.



Time dedicated to save the output data in binary format takes the 0.1% of the total simulation.

PRE-PROCESSING

In order to create a real **complex geometry** to reproduce an industrial problem the first main issue is the resolution with which the objects are represented.

To obtain realistic results with SPH it is appropriate that the **initial geometry** is as **close** as possible **to a real industrial problem**.

This drawback can be solved when several million particles are used in the simulation. Thus, a new pre-processing tool has been developed to deal with more complex geometries: **GenCase** code.

PRE-PROCESSING

3DS DXF DWG GIS H5PART CSV MAX SHP CAD
PLY STL VTK

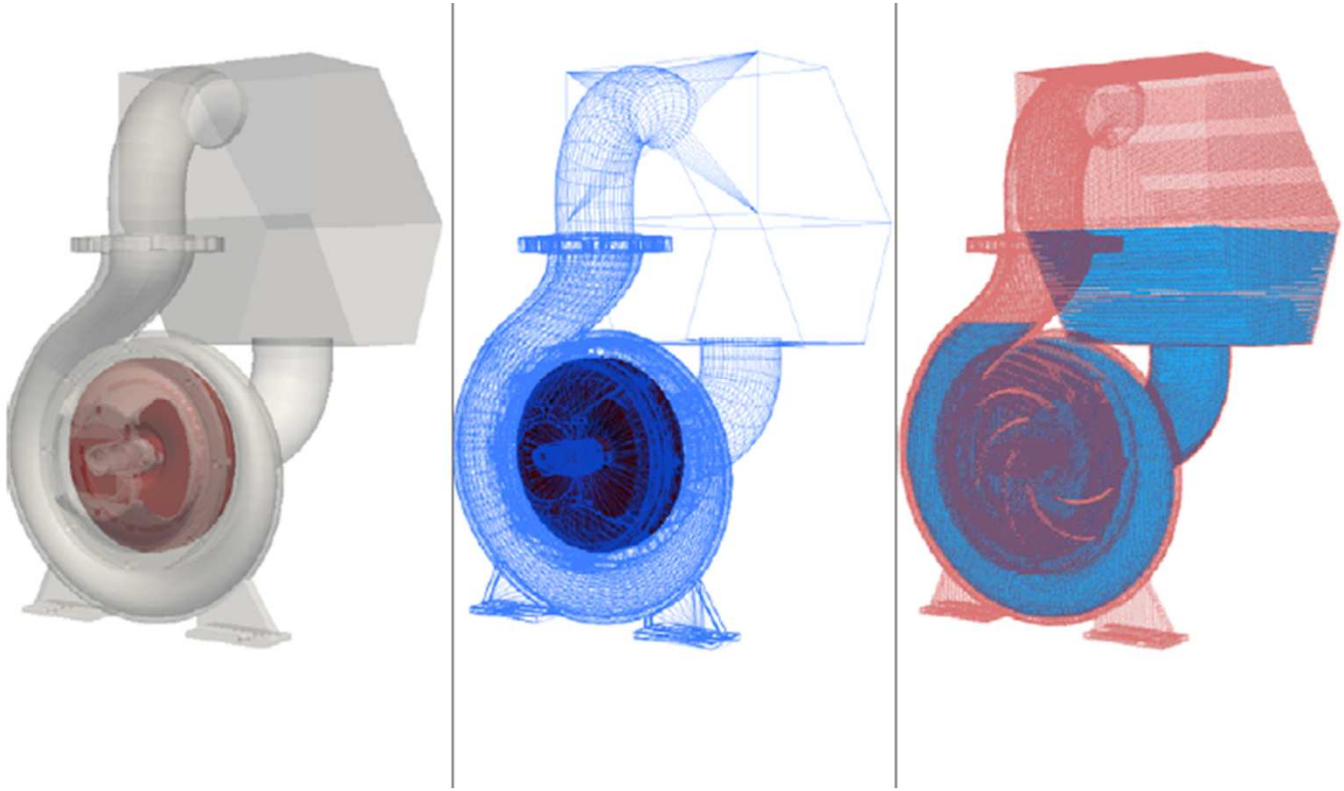
PLY -> exportable using BLENDER
STL -> exportable using 3DSTUDIO
VTK -> PARAVIEW

(files that contain vertices and polygons)

PLY, STL and VTK can be loaded by GenCase
The file is actually a set of triangles, each of these triangles are converted to particles

PRE-PROCESSING

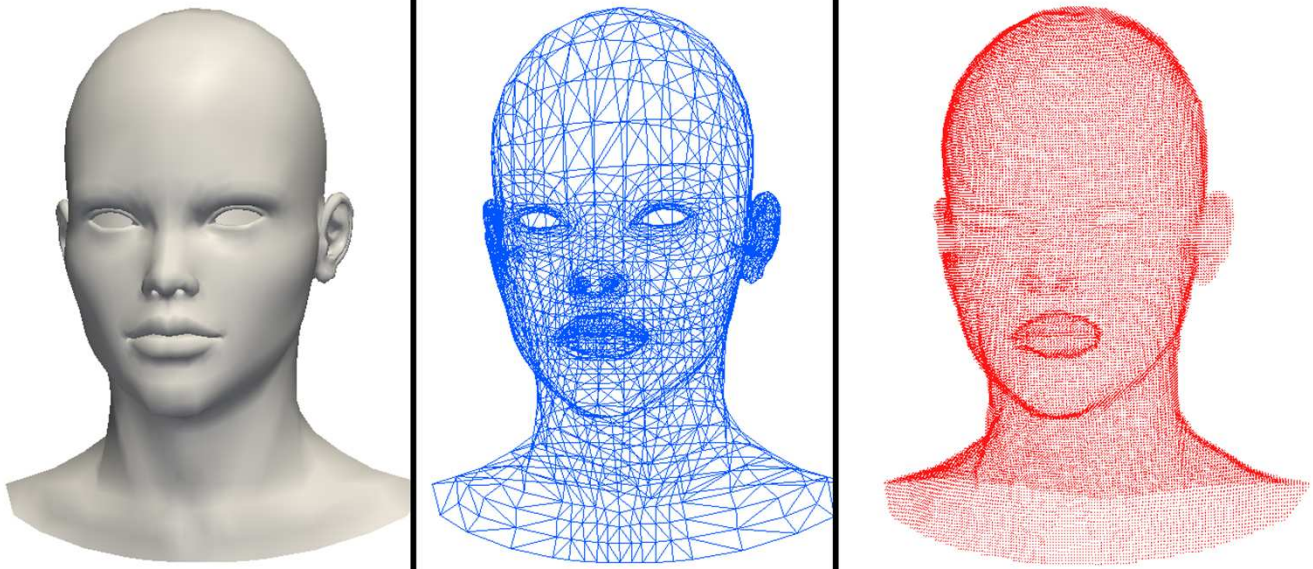
Importing CAD files



PLY, STL and VTK can be loaded by GenCase
The file is actually a set of triangles, each of these triangles are converted to particles

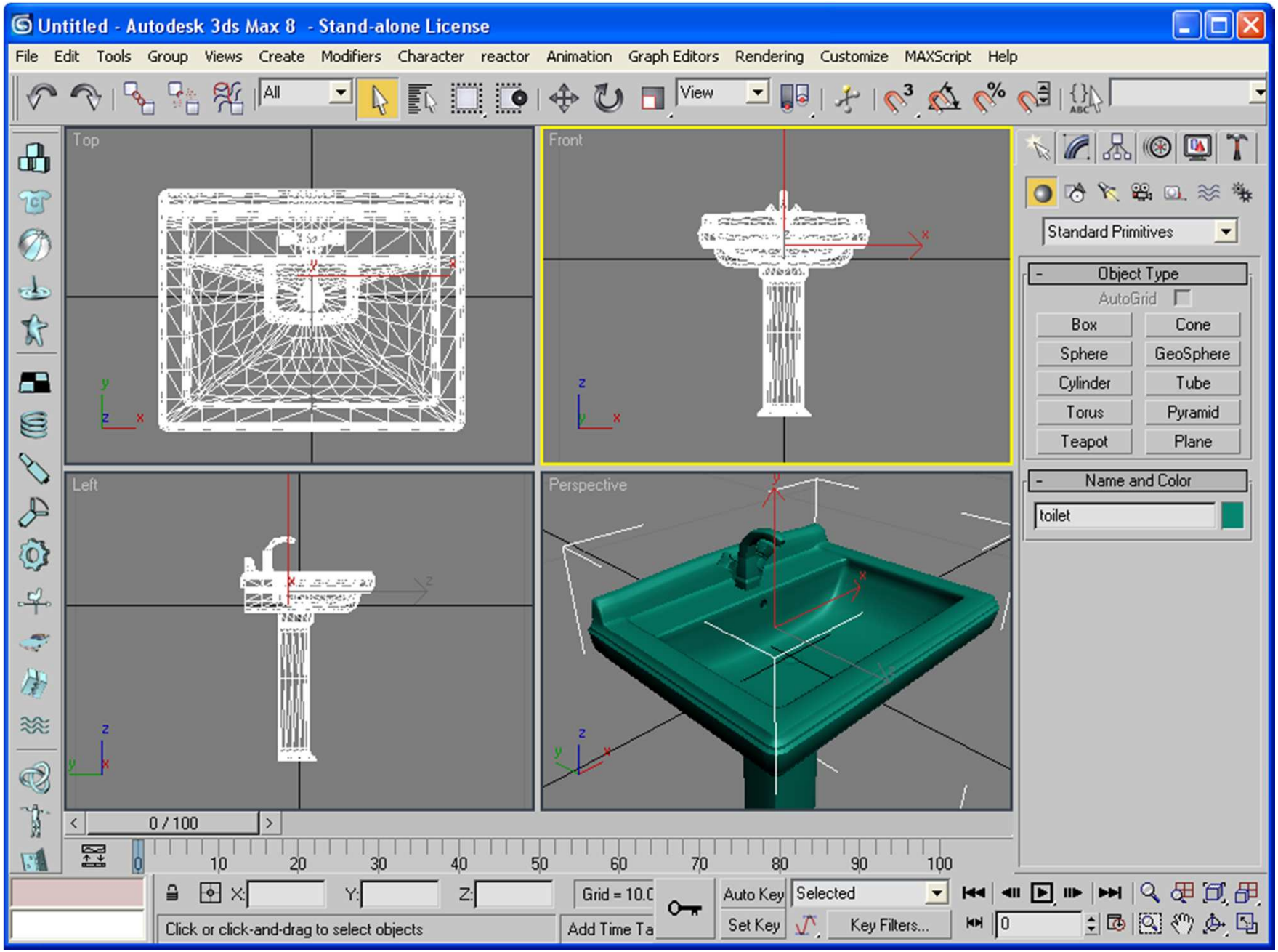
PRE-PROCESSING

Importing 3DStudio objects



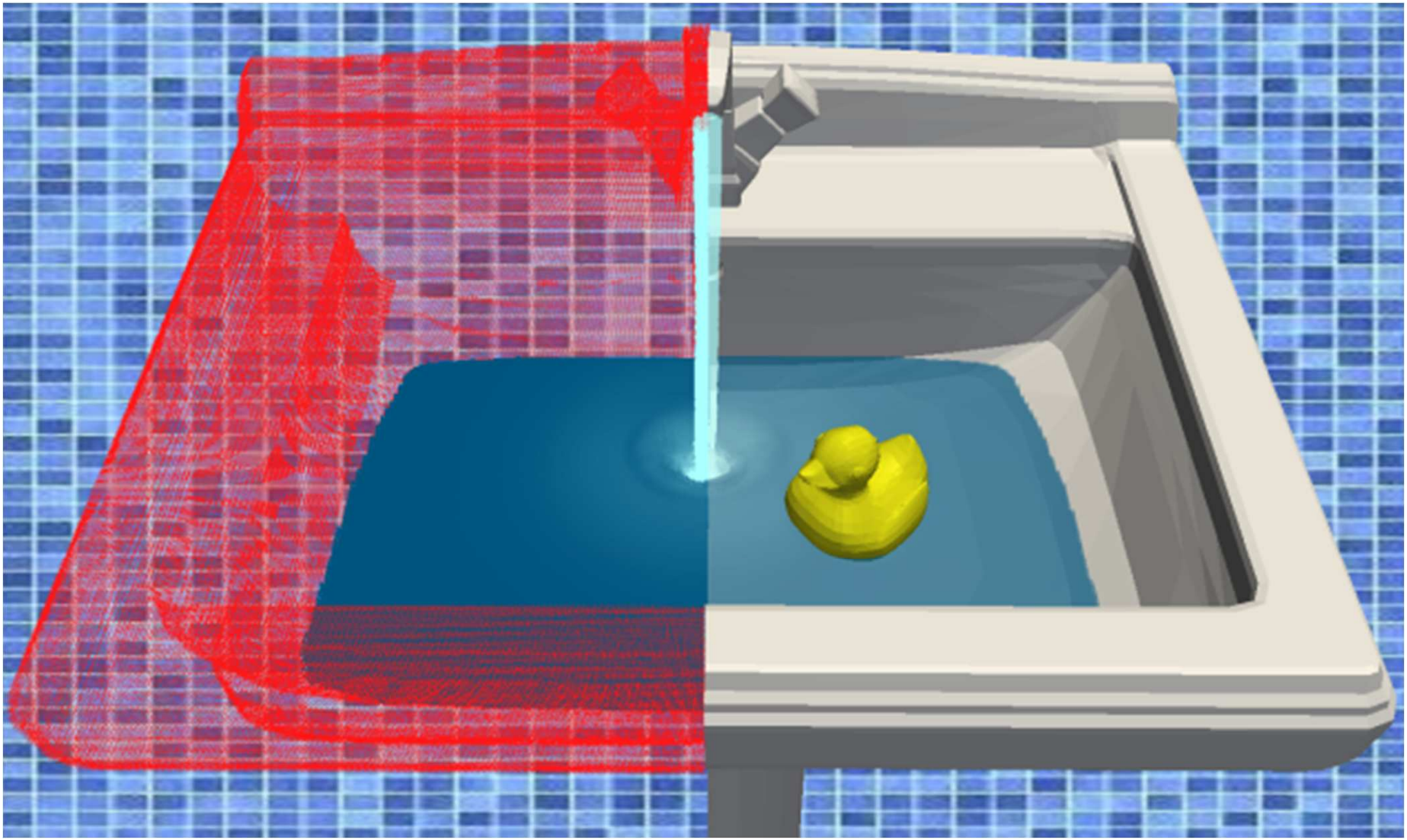
PLY, STL and VTK can be loaded by GenCase
The file is actually a set of triangles, each of these triangles are converted to particles

PRE-PROCESSING

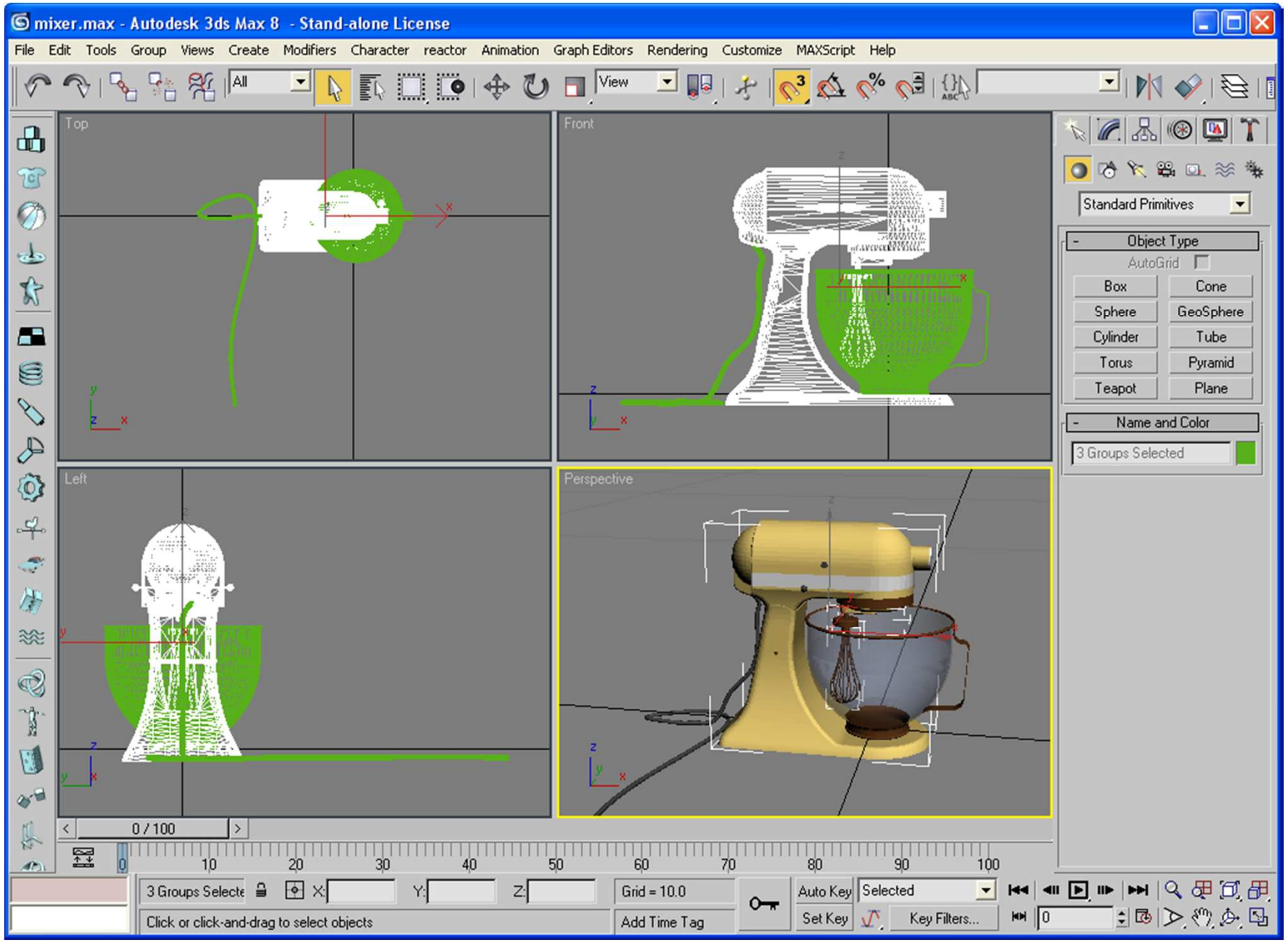


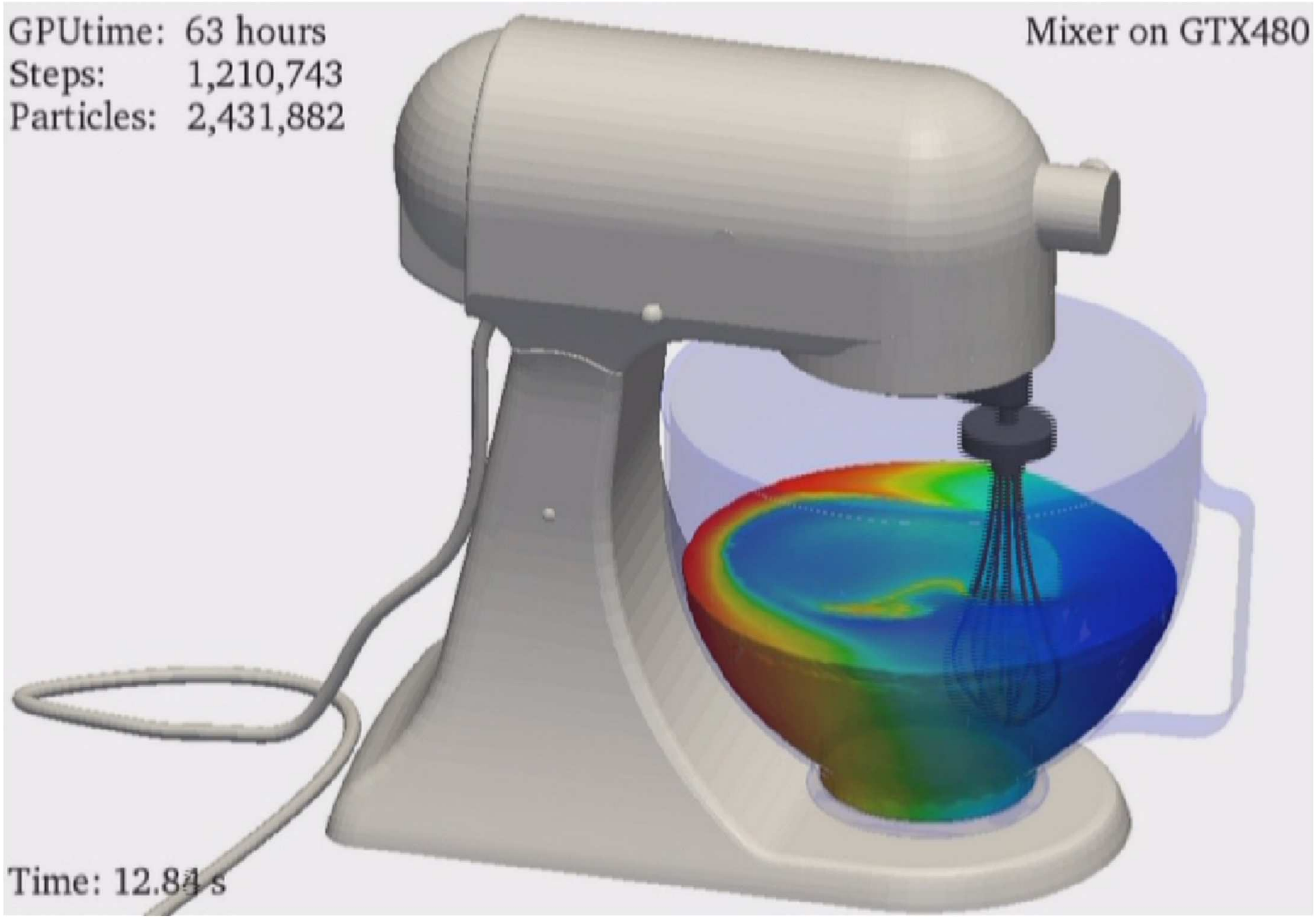
PRE-PROCESSING

Importing 3DStudio objects



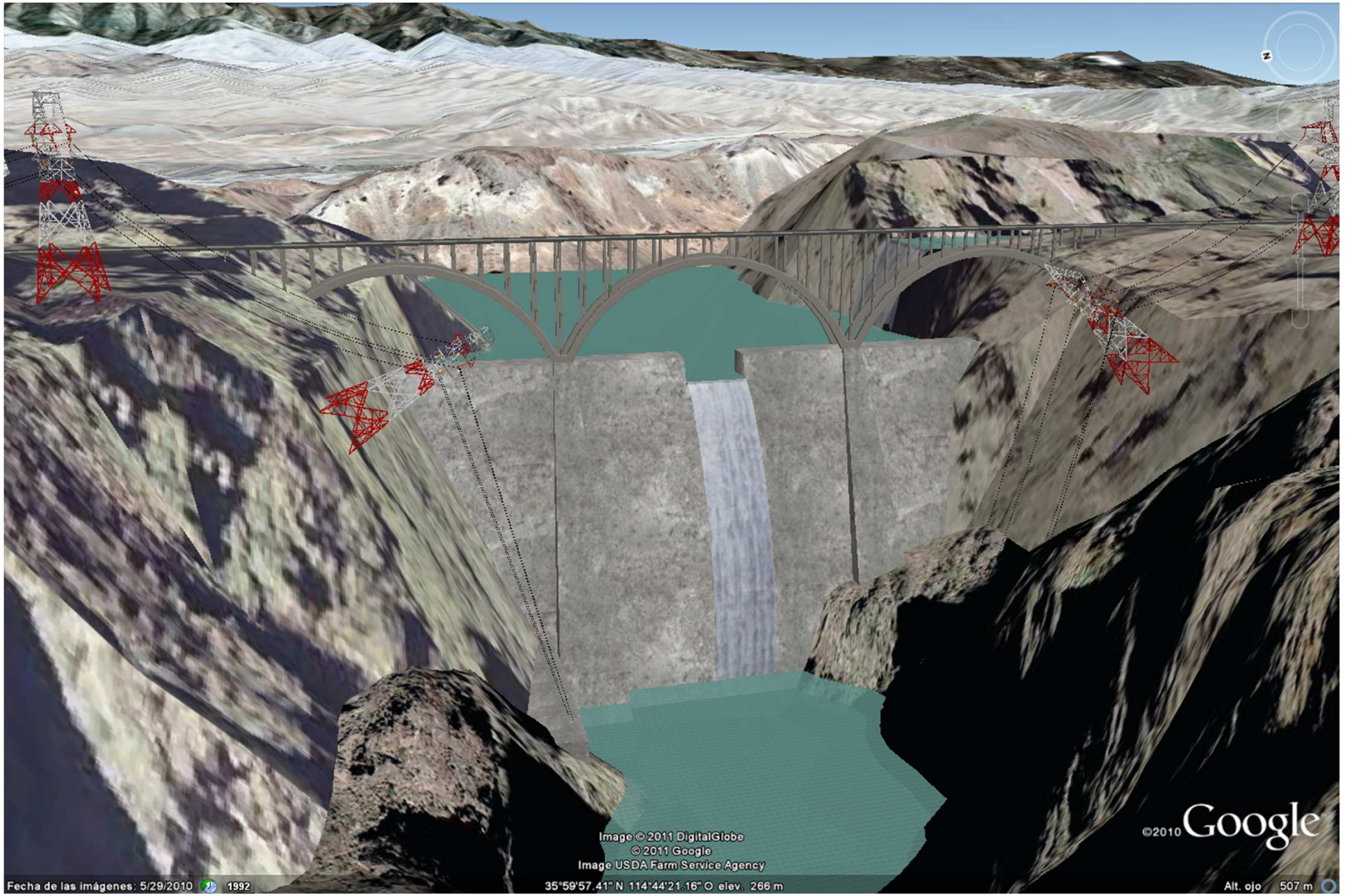
PRE-PROCESSING




GPUtime: 63 hours
Steps: 1,210,743
Particles: 2,431,882
Mixer on GTX480
Time: 12.84 s

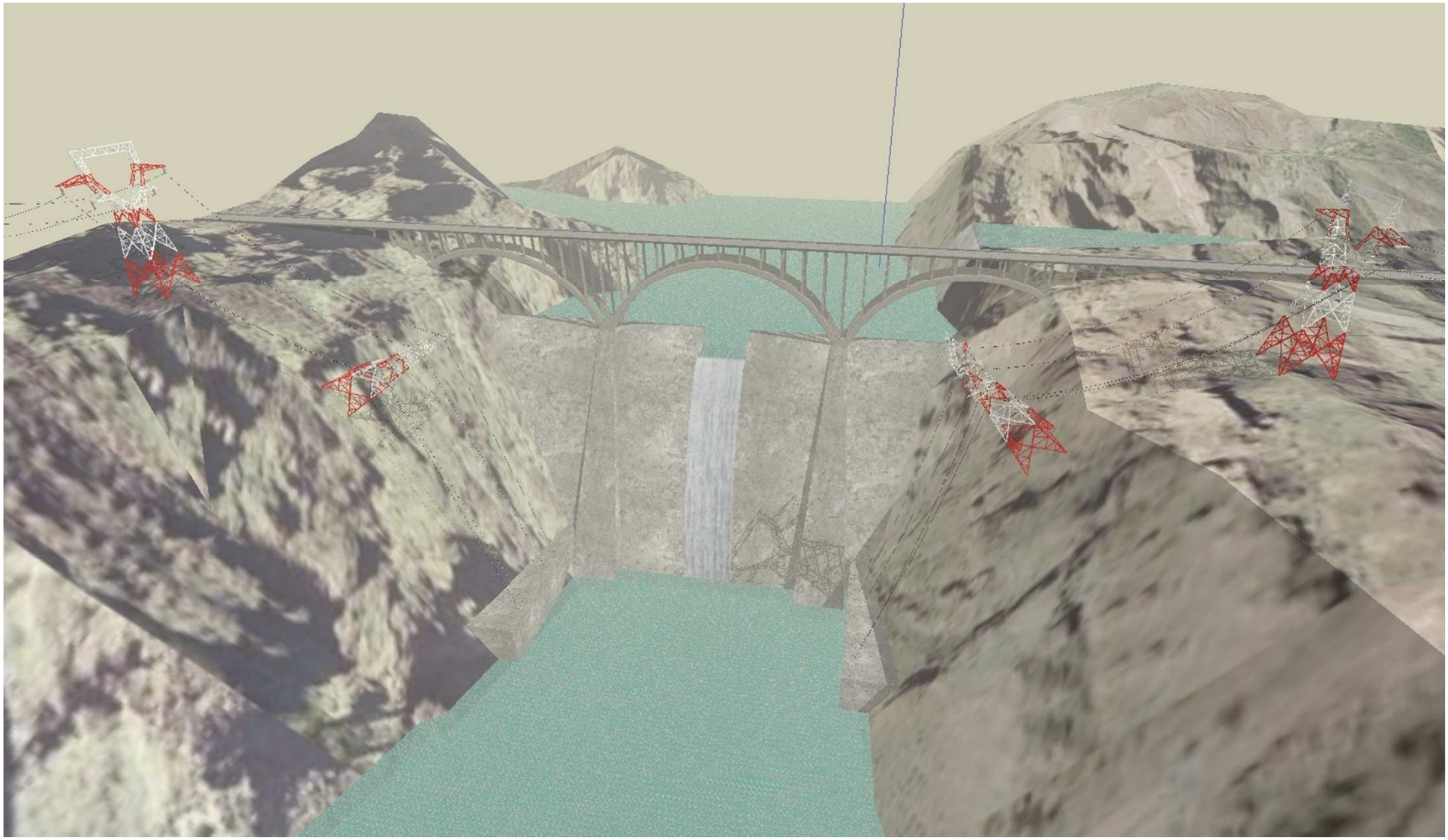
PRE-PROCESSING

KMZ from GOOGLE EARTH



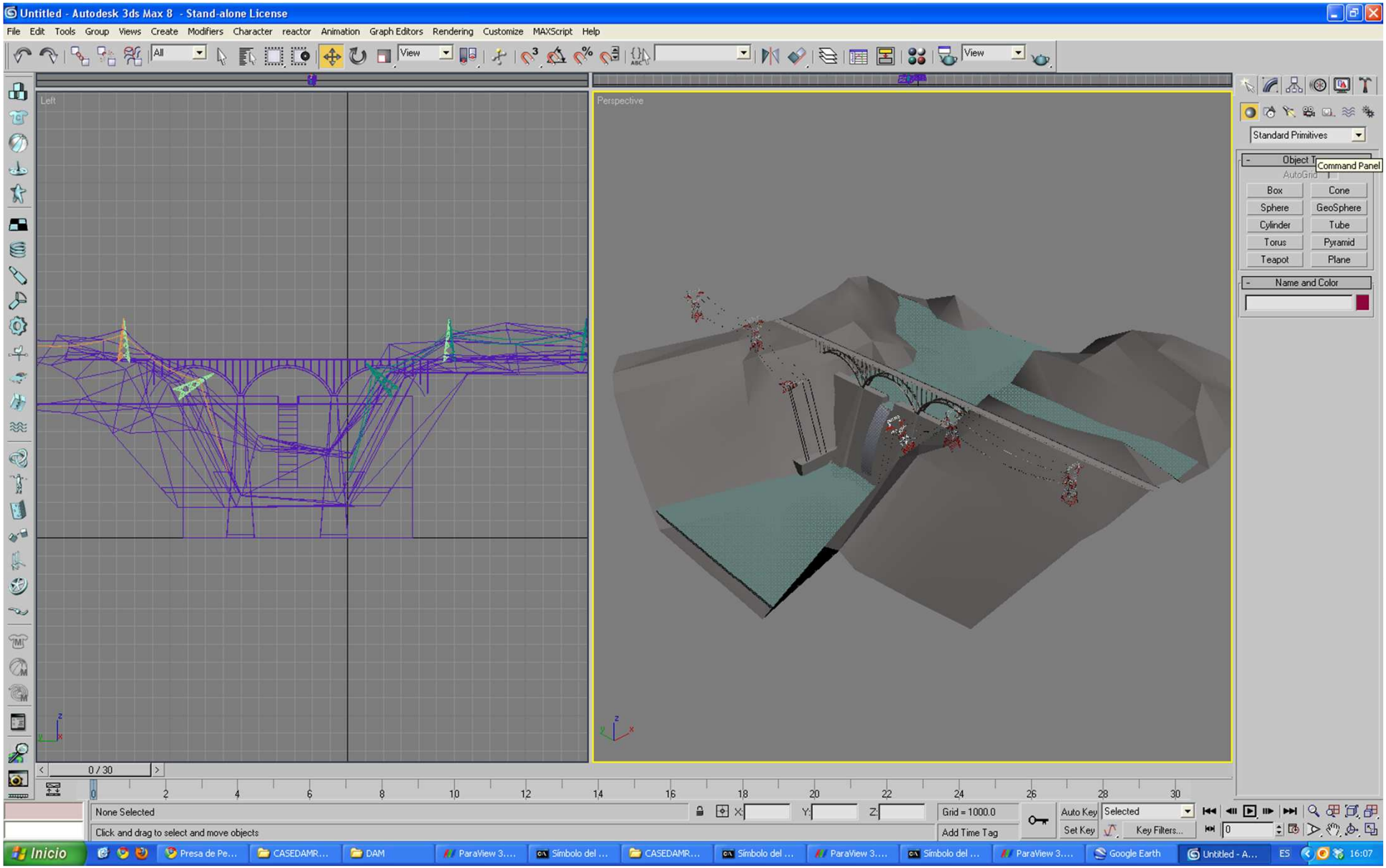
PRE-PROCESSING

SKP from GOOGLE Sketchup 8



PRE-PROCESSING

3DS from Autodesk 3ds MAX 8

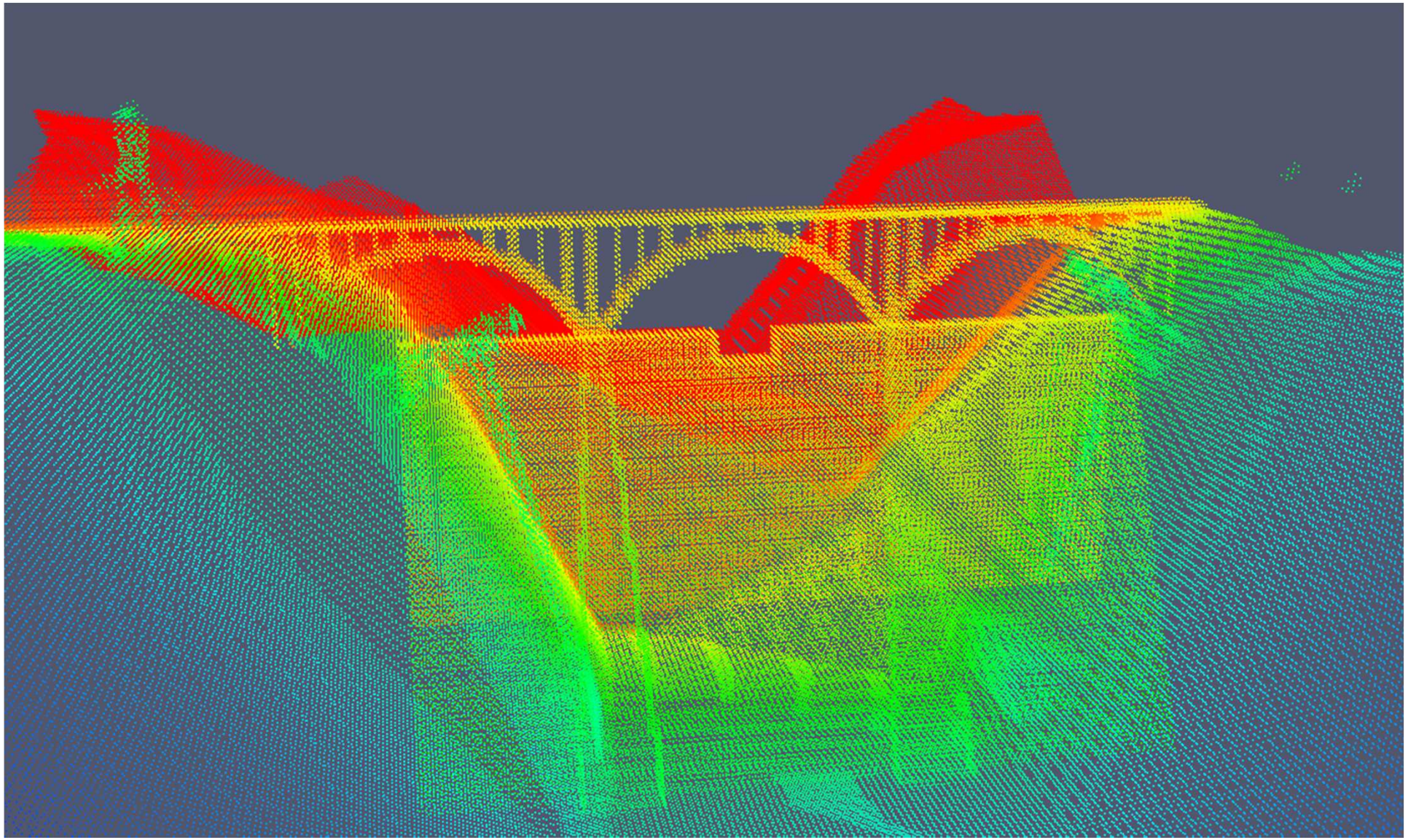


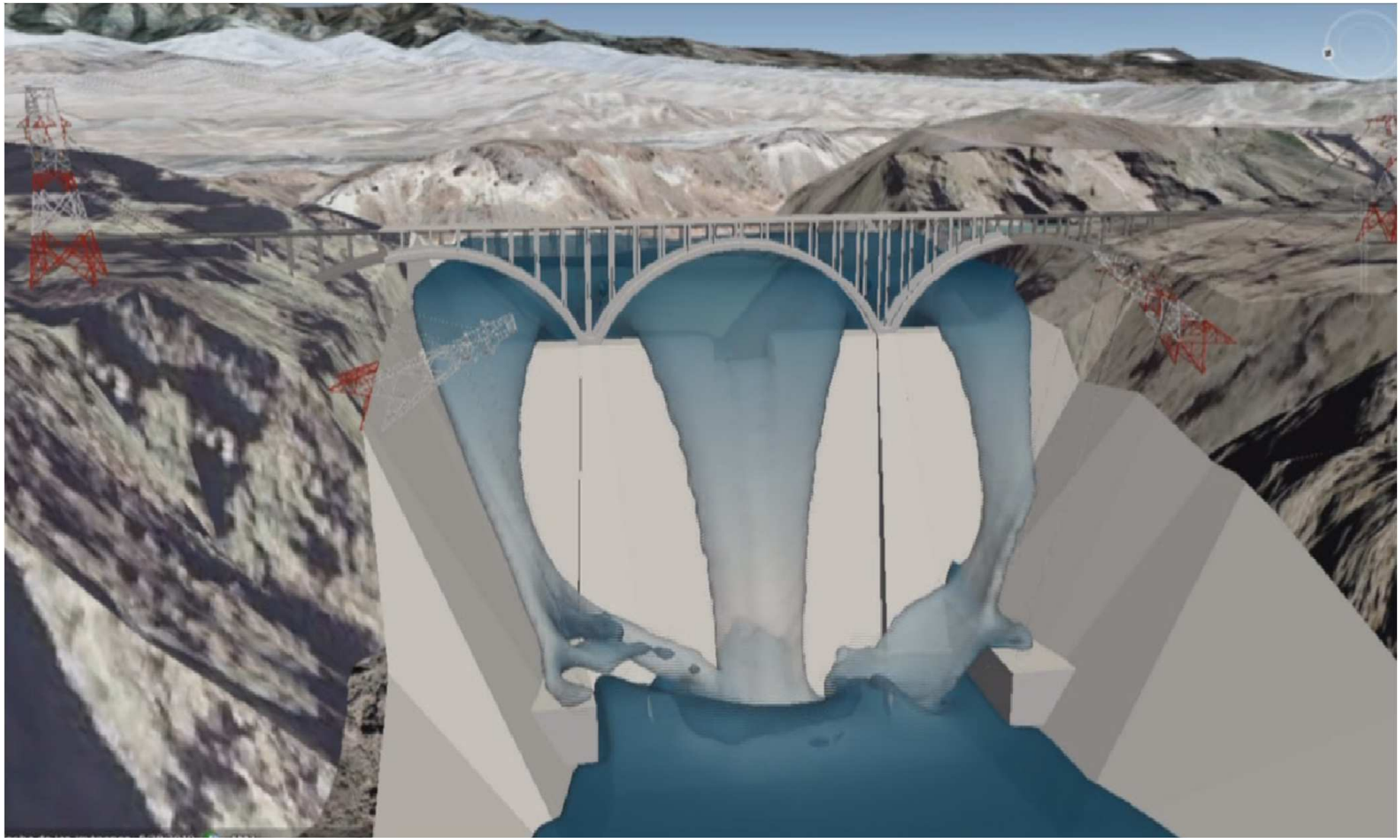
PRE-PROCESSING

VTK(polygons) from Paraview

PRE-PROCESSING

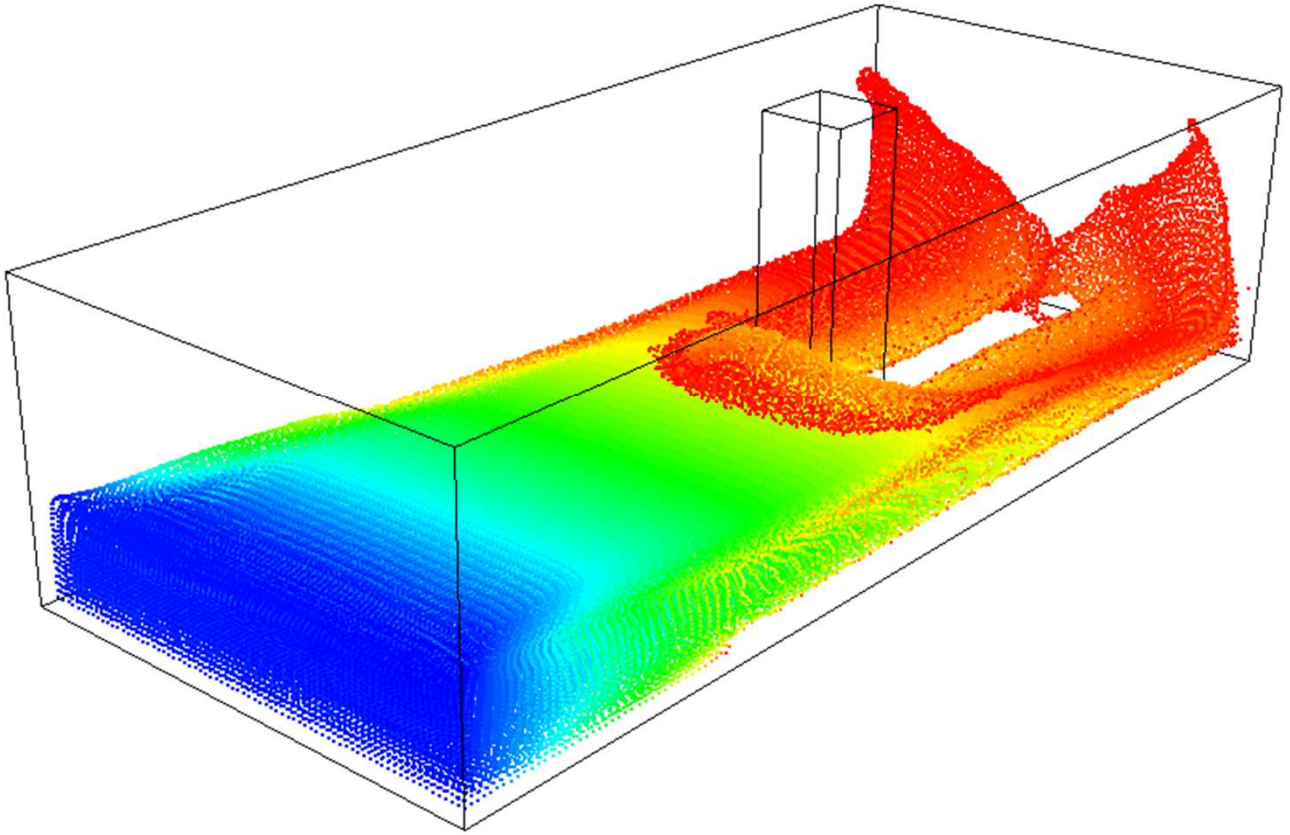
VTK(points) from Paraview





POST-PROCESSING

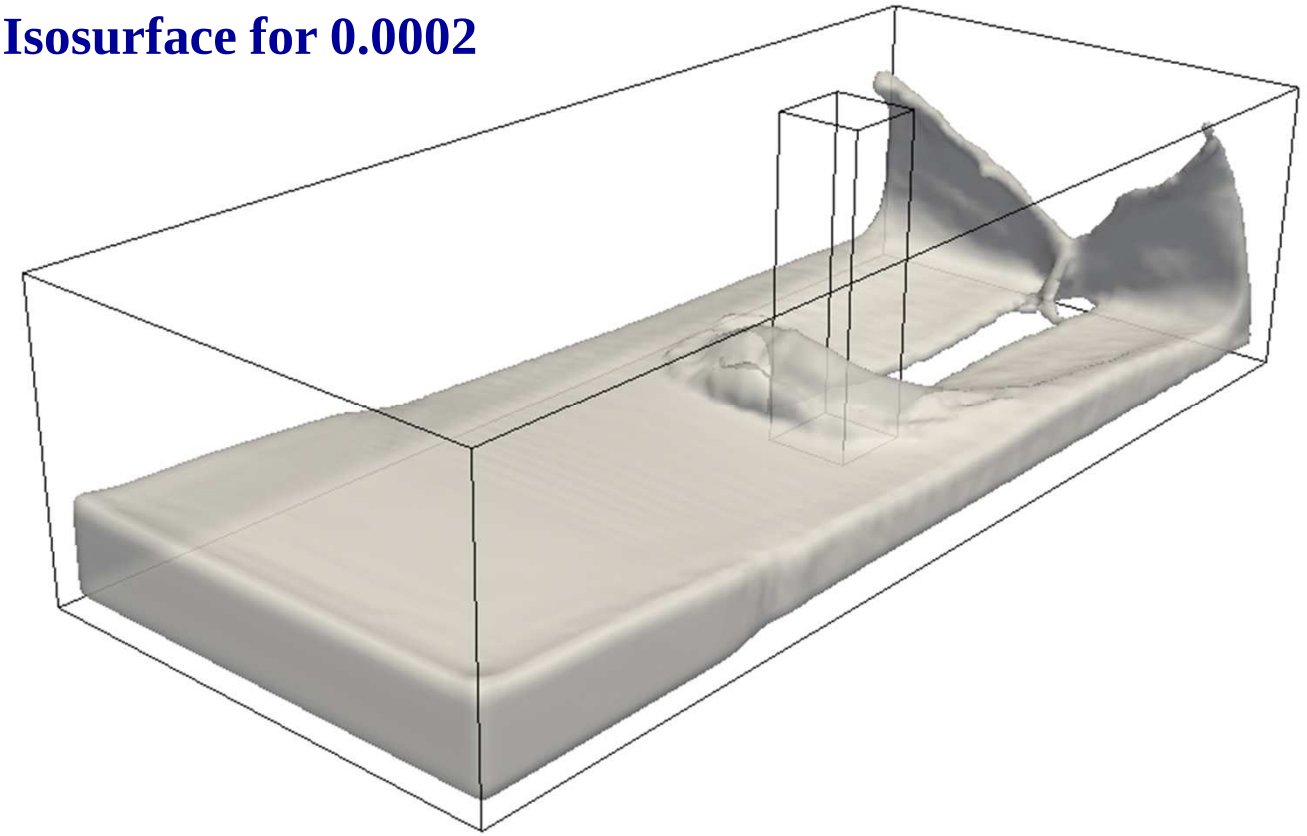
305,252 particles



POST-PROCESSING

Mass: [0 , 0.0004]

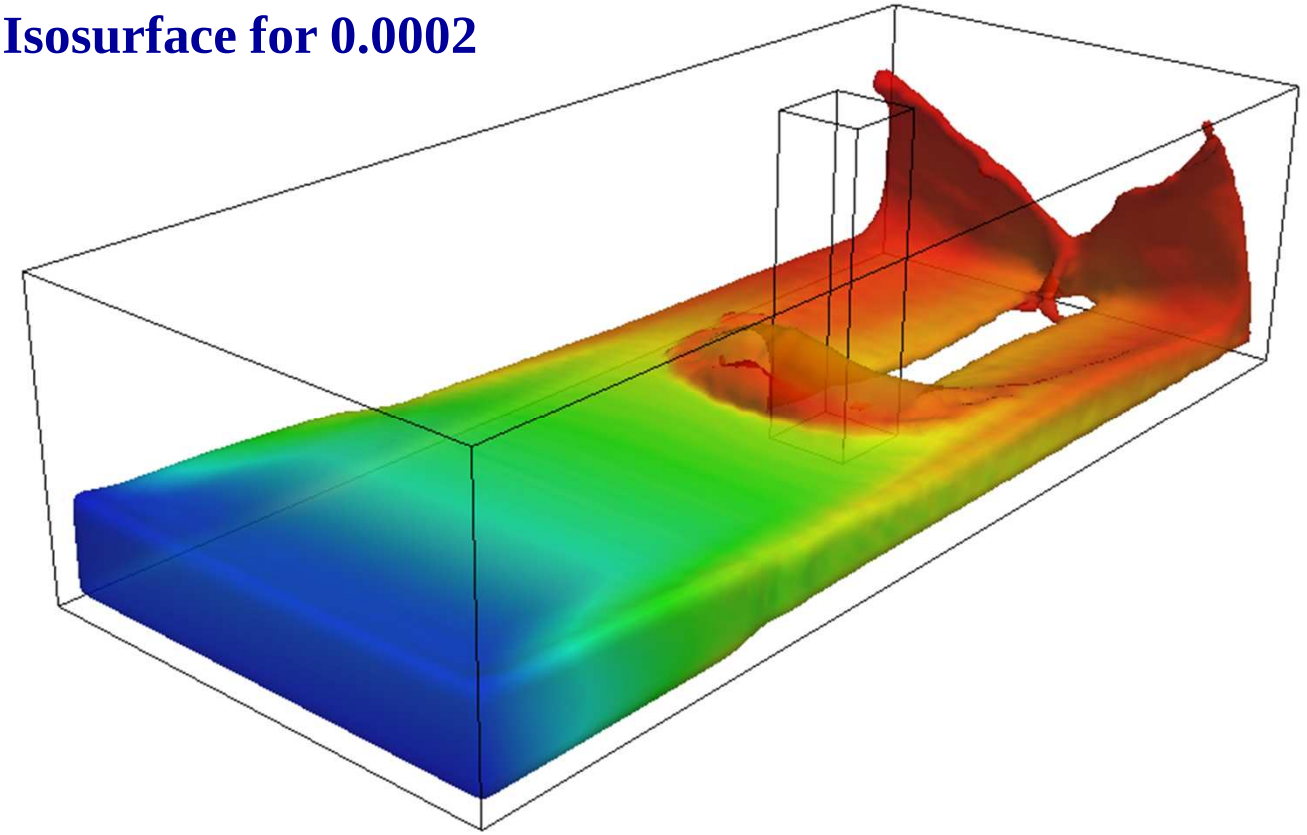
Isosurface for 0.0002



POST-PROCESSING

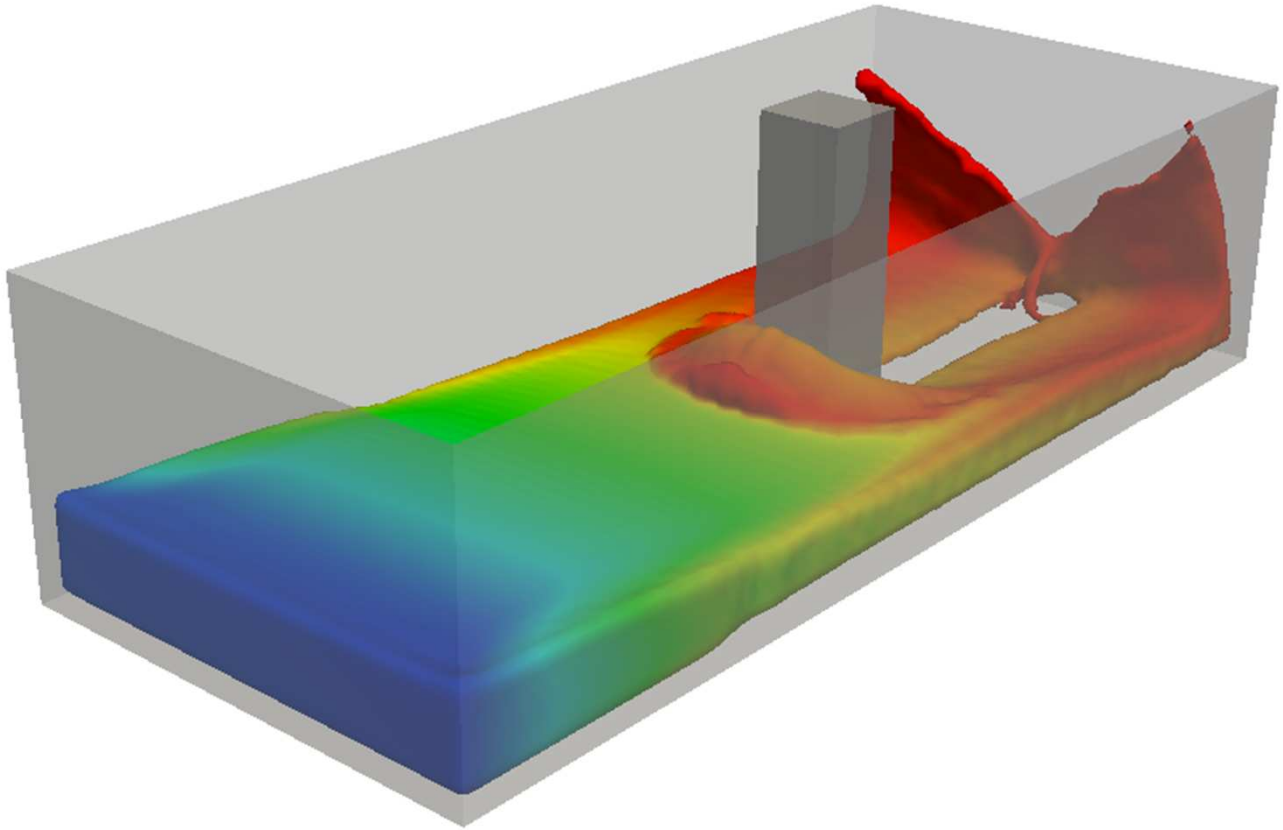
Mass: [0 , 0.0004]

Isosurface for 0.0002

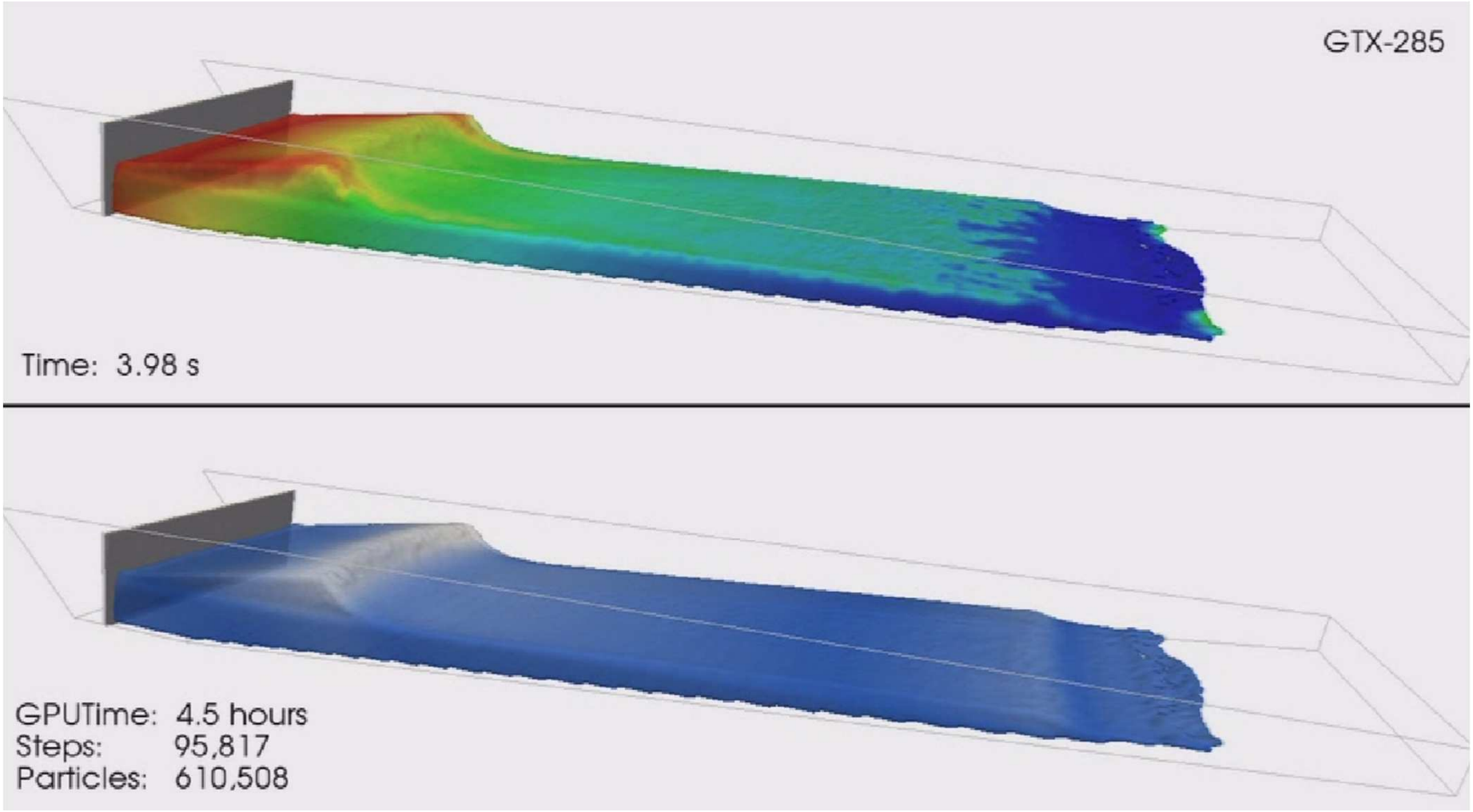


POST-PROCESSING

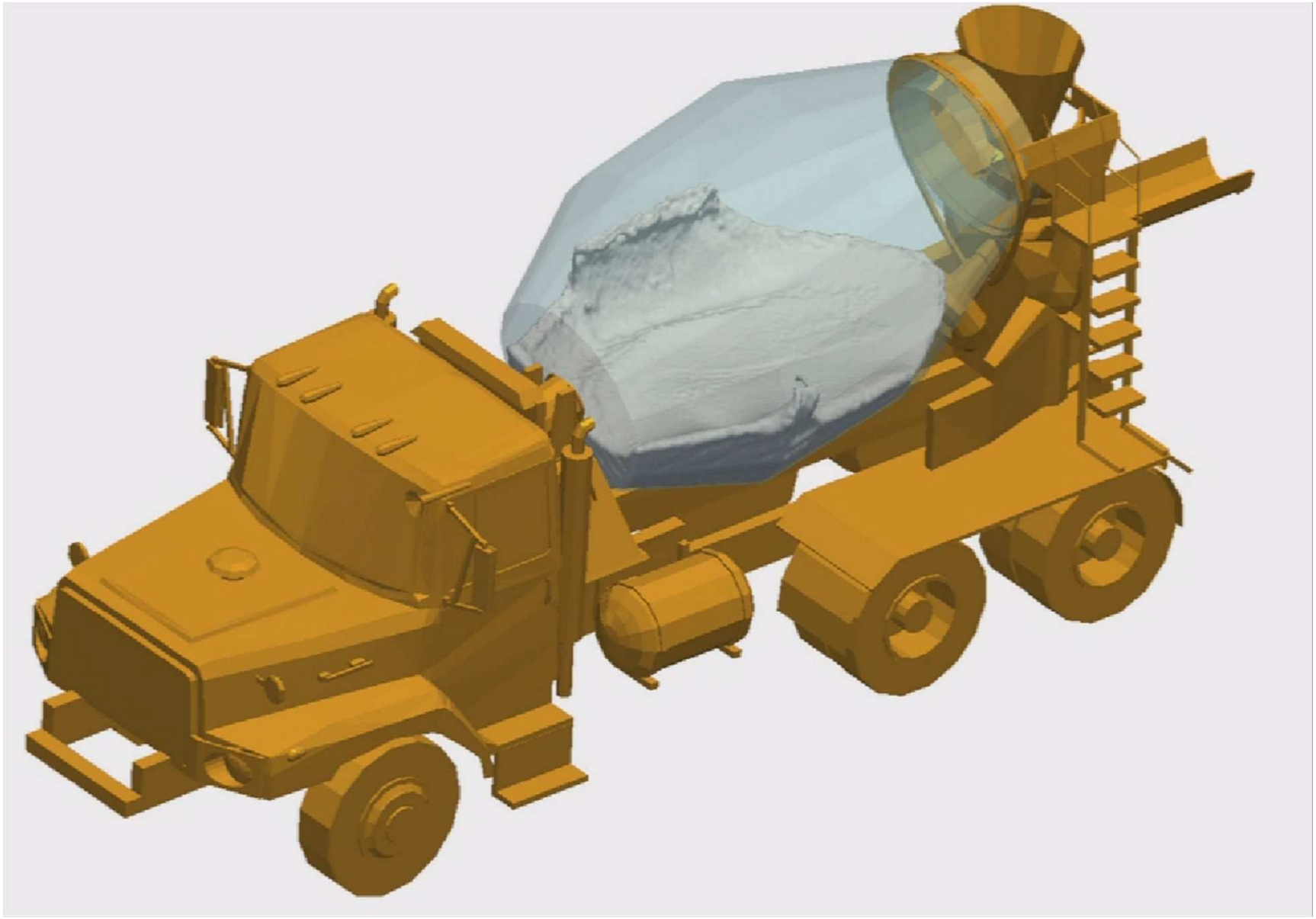
Import VTK objects



POST-PROCESSING



POST-PROCESSING



POST-PROCESSING

Corridor of the department at University of Vigo

POST-PROCESSING

POST-PROCESSING

Outline

- Numerical methods

- SPH method and computational runtimes

- SPHysics and DualSPHysics project

- How to accelerate SPH

- Multi-CPU implementation

- GPU-implementation

- Multi-GPU implementation

- Applications

- Needs when accelerating the code: format files, pre/post-processing

- **DualSPHysics code**

DualSPHysics code

www.sphysics.org

DualSPHysics code

www.dual.sphysics.org

DUALSPHYSICS DOCUMENTATION

DUALSPHYSICS DOCUMENTATION v1.0

Filename	Date	Size	D/L
DualSPHysics_v1.0_GUIDE.pdf	2011-01-11 12:53	5.34 MB	1,037

DUALSPHYSICS PACKAGE

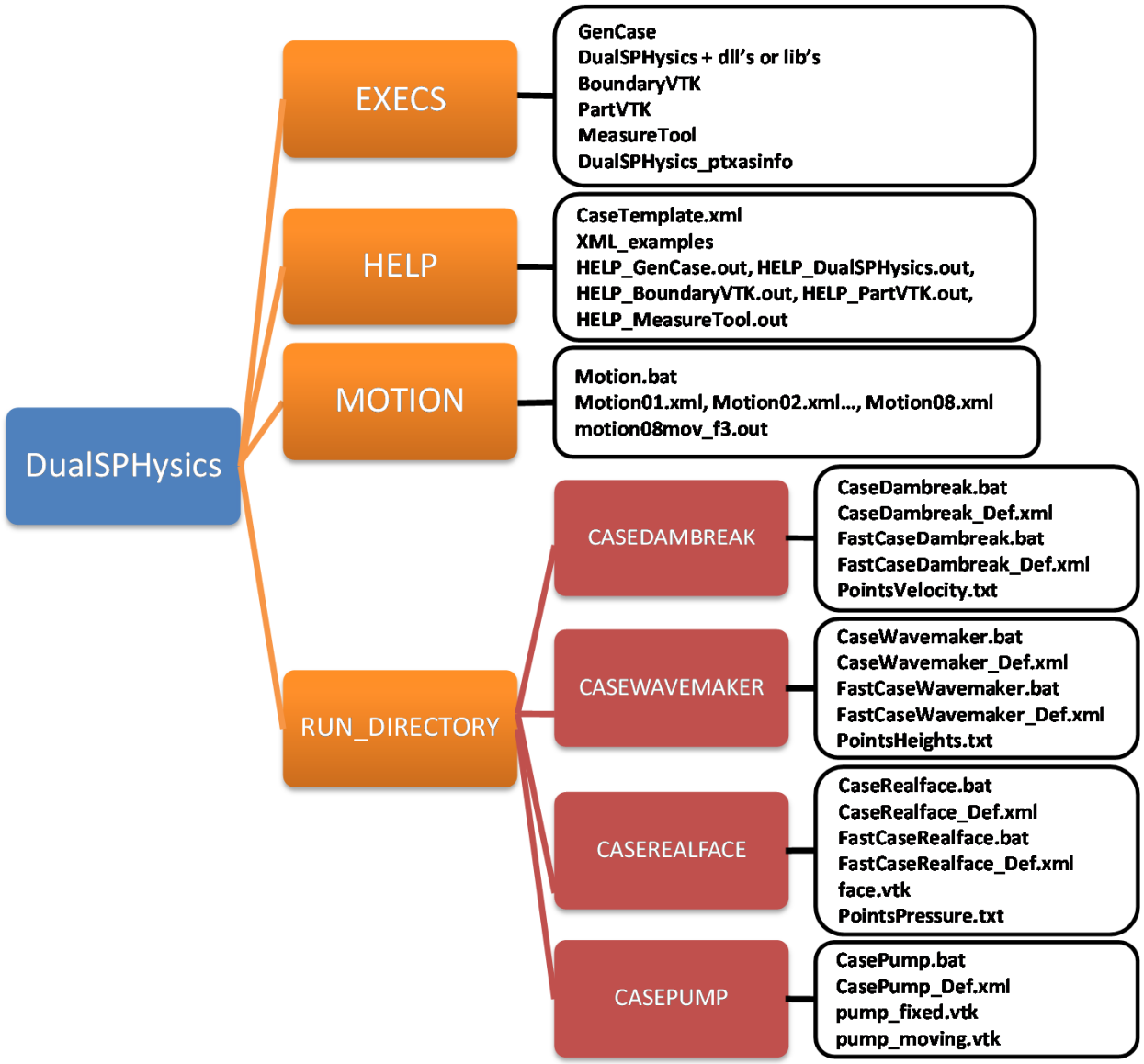
DUALSPHYSICS PACKAGE v1.0

Filename	Date	Size	D/L
DualSPHysics_v1.0_linux_32bit.zip	2011-01-11 12:41	14.68 MB	170
DualSPHysics_v1.0_linux_64bit.zip	2011-01-11 12:41	15.61 MB	241
DualSPHysics_v1.0_windows_32bit.zip	2011-01-11 12:41	12.44 MB	363
DualSPHysics_v1.0_windows_64bit.zip	2011-01-11 12:41	12.24 MB	312

More than 1,000 downloads of v1.0 during the first 90 days
Available now v1.2 with Multi-core implementation.

DualSPHysics code

www.dual.sphysics.org



GenCase
Pre-processing

DualSPHysics
SPH solver

BoundaryVTK
PartVTK
Measutool
Post-processing

DualSPHysics code

HELP:

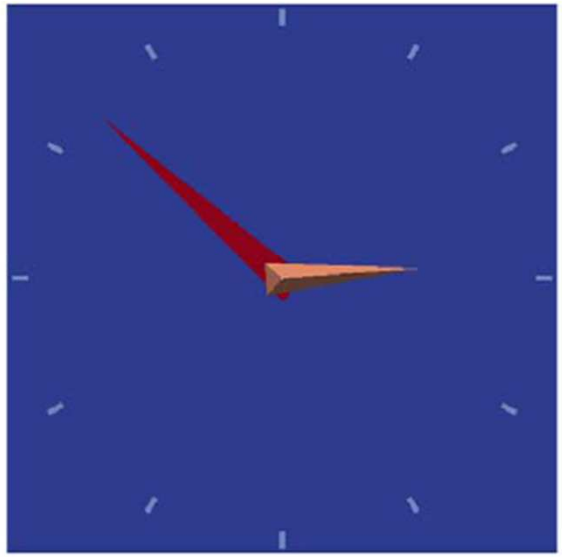
•Contains *CaseTemplate.xml*, a XML example with all the different labels and formats that can be used in the input XML file.

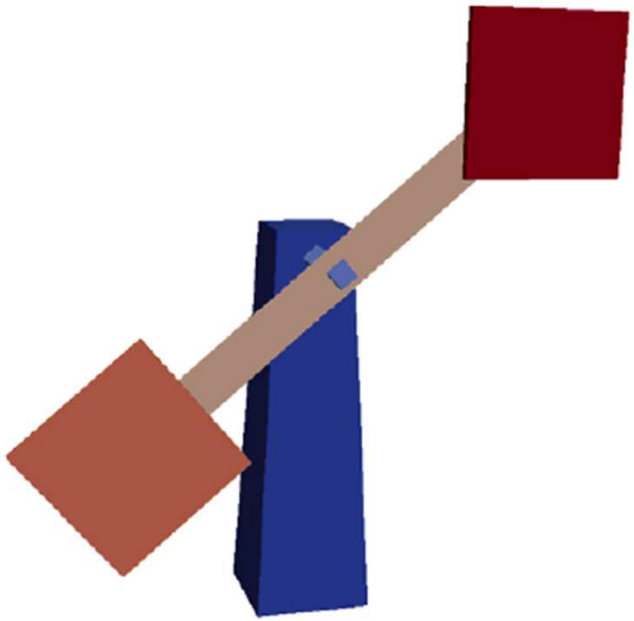
•*HELP_NameCode.out* includes the HELP about the execution parameters of the different codes.

MOTION:

•Contains the bat file *Motion.bat* to perform the examples with the different type of movements that can be described with DualSPHysics. Eight examples can be carried out (*Motion01.xml…*, *Motion08.xml*).

•The text file motion08mov_f3.out describes the predefined motion used in the eighth example.

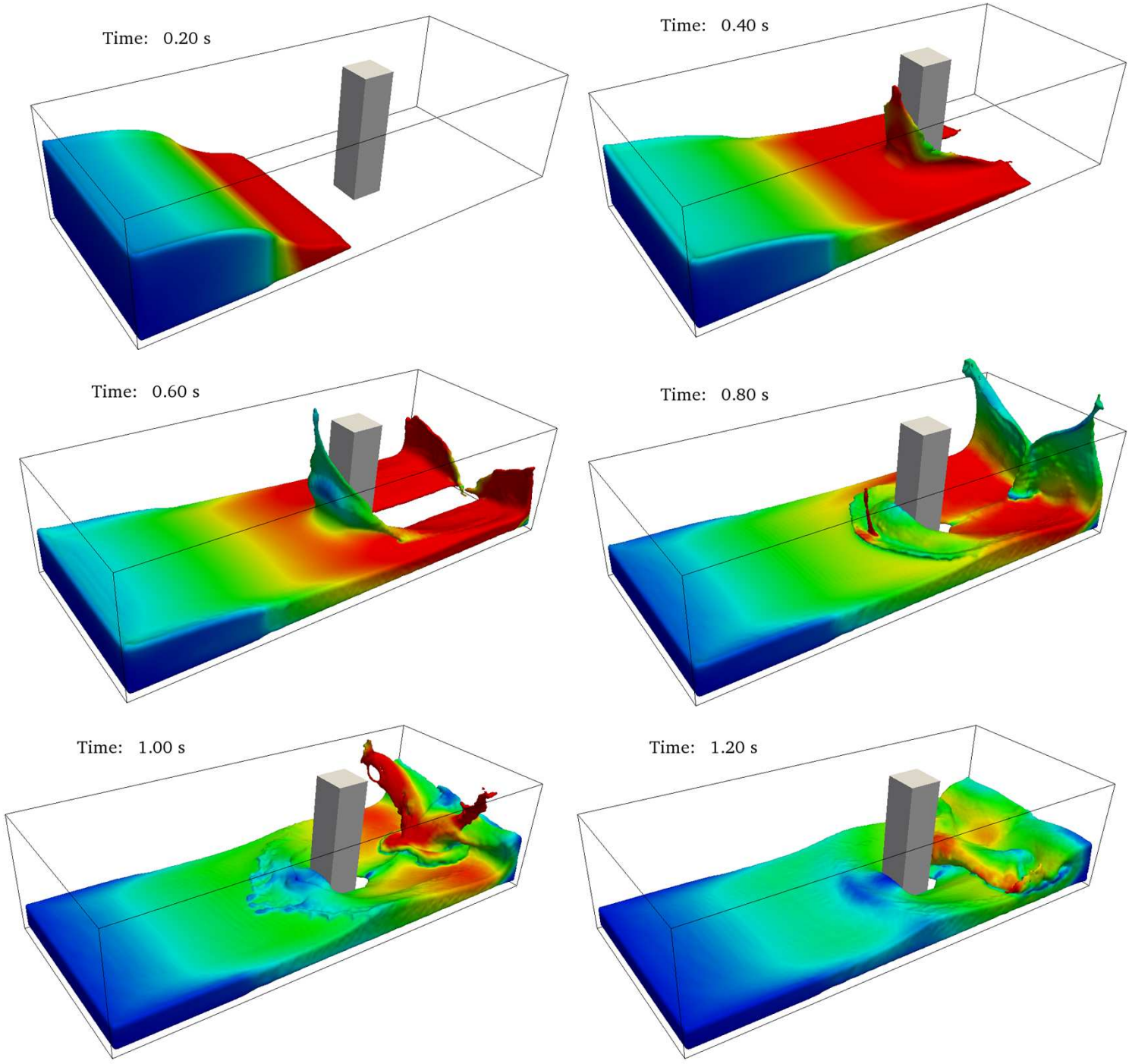




DualSPHysics code

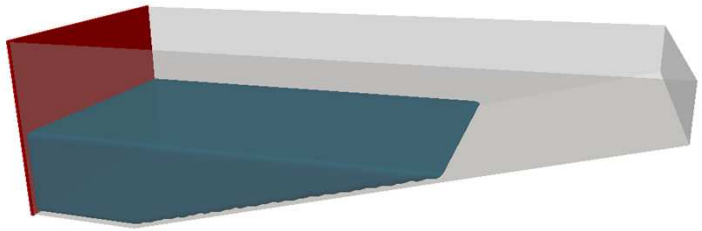
TESTCASES

1. Dambreak

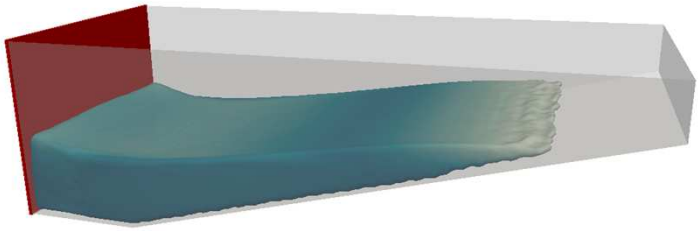


TESTCASES

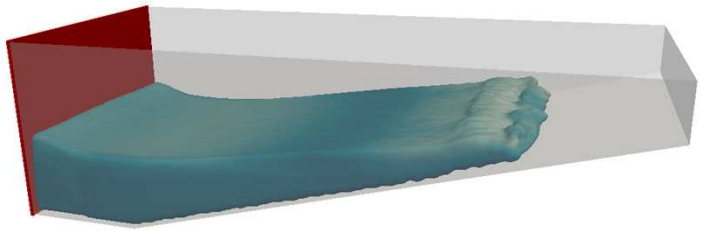
2. Wavemaker



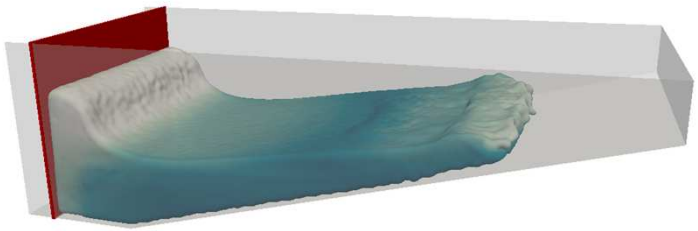
Time: 0.00 s



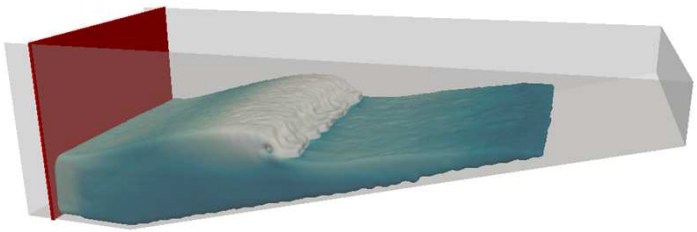
Time: 2.00 s



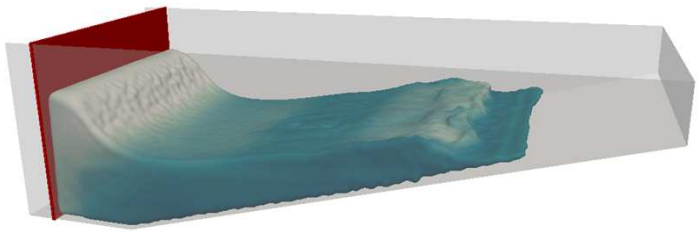
Time: 4.00 s



Time: 6.00 s



Time: 8.00 s

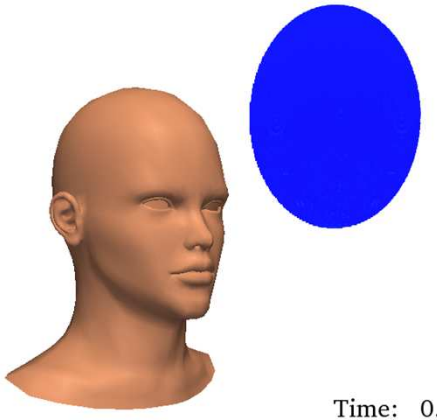


Time: 10.00 s

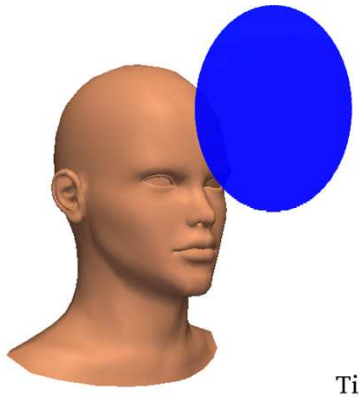
DualSPHysics code

TESTCASES

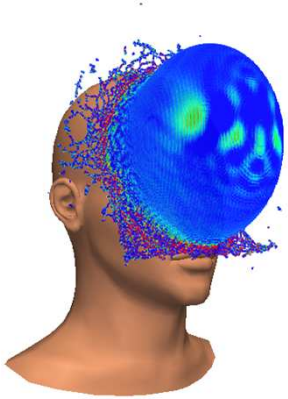
3. RealFace



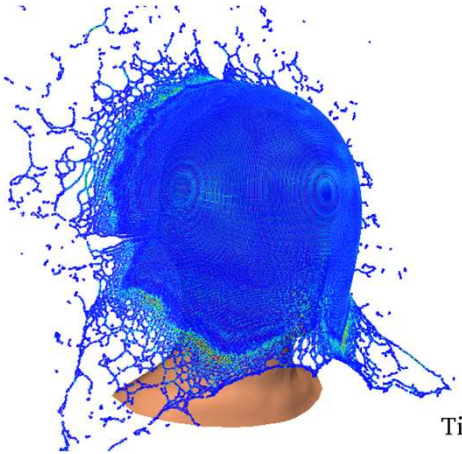
Time: 0.05 s



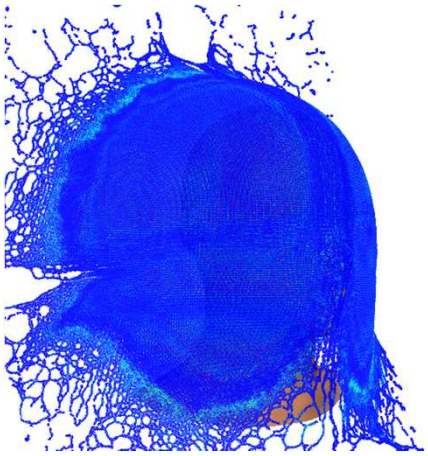
Time: 0.45 s



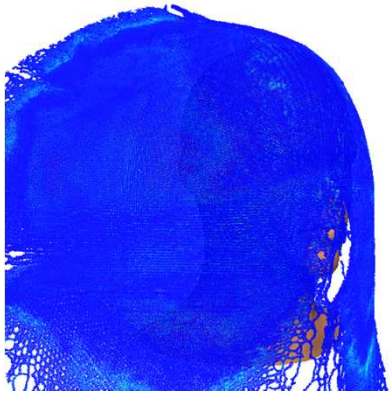
Time: 0.90 s



Time: 1.35 s



Time: 1.80 s



Time: 2.25 s

DualSPHysics code

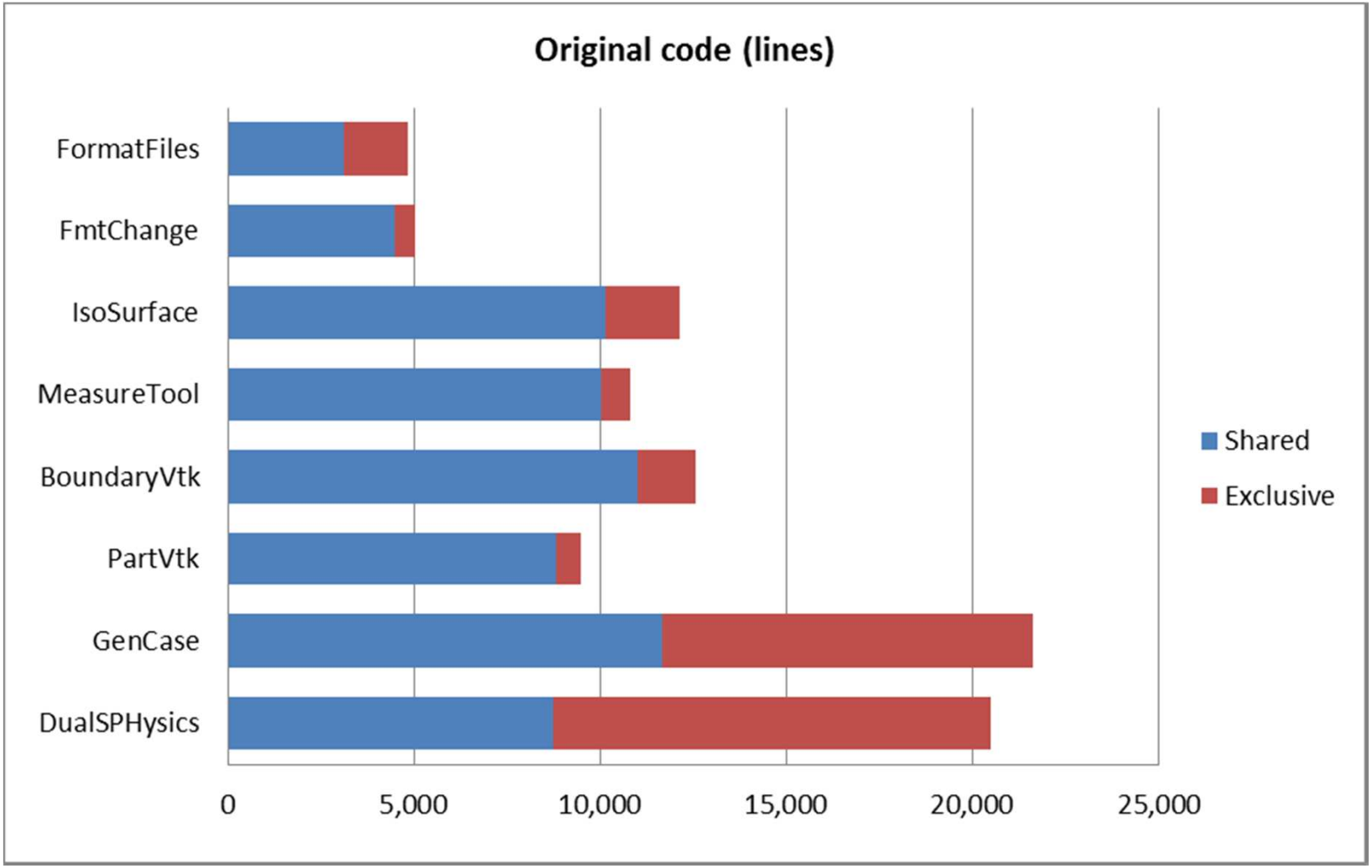
EXECS:

•Contains all the executables codes.

- ❑ GenCase - pre-processing tool
- ❑ DualSPHysics - SPH solver
- ❑ BoundaryVTK - post-processing tool
- ❑ PartVTK - post-processing tool
- ❑ MeasureTool - post-processing tool

•The text file *ptxas_info.out* is used to optimise the block size for the different CUDA kernels on GPU executions, to maximise the occupancy.

DualSPHysics code



DualSPHysics code

SOFTWARE:

Codes

- **SPHysics from www.sphysics.org**
- **DualSPHysics from www.dual.sphysics.org**

Pre-processing

- **FreeCad**
- **Qgis**
- **EveryDWG**

Post-processing

- **Paraview from www.paraview.org**
- **Blender from www.blender.org**

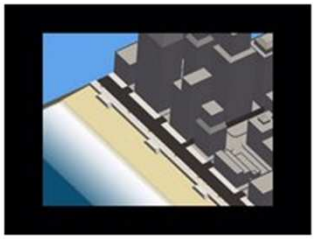
Compilers

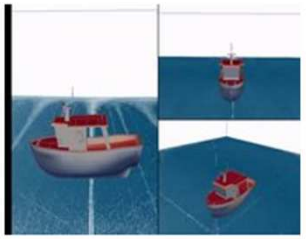
- **CUDA from www.nvidia.com**
- **GCC**

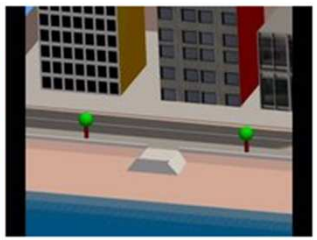
DualSPHysics code

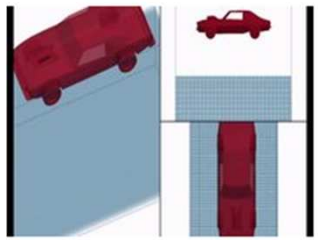
www.vimeo.com/dualsphysics

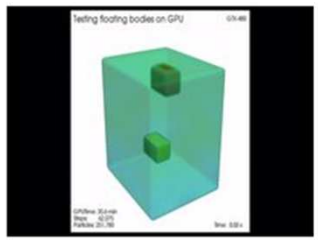

Chess game with DualSPHysics (SPH on GPU)

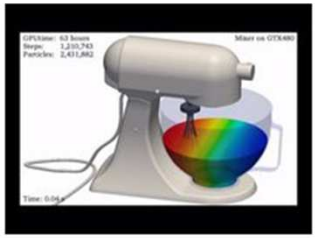

Promenade-wave interaction with DualSPHysics (SPH on GPU)

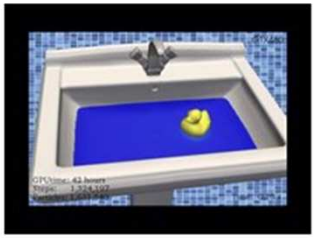

Floating ship with DualSPHysics (SPH on GPU)

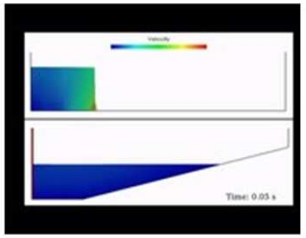

Wave-seawalk interaction with DualSPHysics (SPH on GPU)

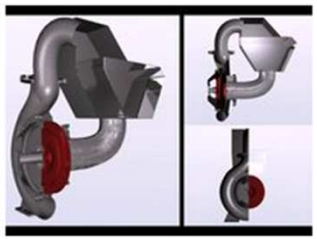

Real car falling down in a pool with DualSPHysics (SPH on GPU)

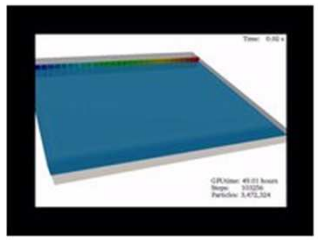

Testing floating bodies with DualSPHysics (SPH on GPU)


Mixer machine mechanism using DualSPHysics (SPH on GPU)

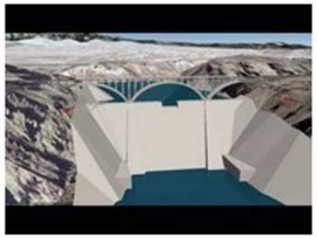

Sink with DualSPHysics (SPH on GPU)

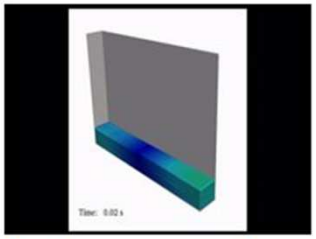

2D simulations with DualSPHysics (SPH on GPU)

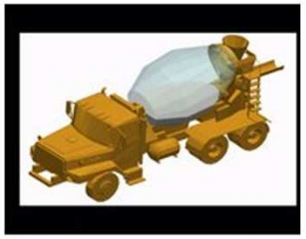

Pump mechanism with DualSPHysics (SPH on GPU) BETTER VISUALISATION

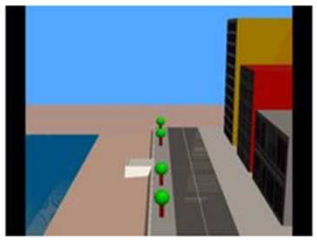

Wavetank with DualSPHysics (SPH on GPU)

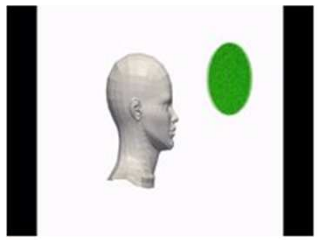

New visualisation tools with DualSPHysics (SPH on GPU)

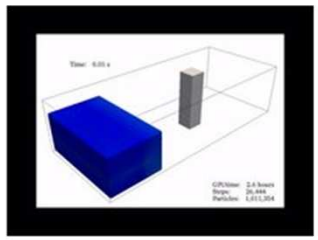

Dam release using a KMZ file from GOOGLE EARTH with DualSPHysics (SPH on GPU)


Sloshing movement of a wave tank with DualSPHysics (SPH on GPU)


Sloshing movement of a concrete mixer with DualSPHysics (SPH on GPU)


Wave-seawalk interaction with DualSPHysics (SPH on GPU) LATERAL VIEW


Fluid impact on a real face with DualSPHysics (SPH on GPU)

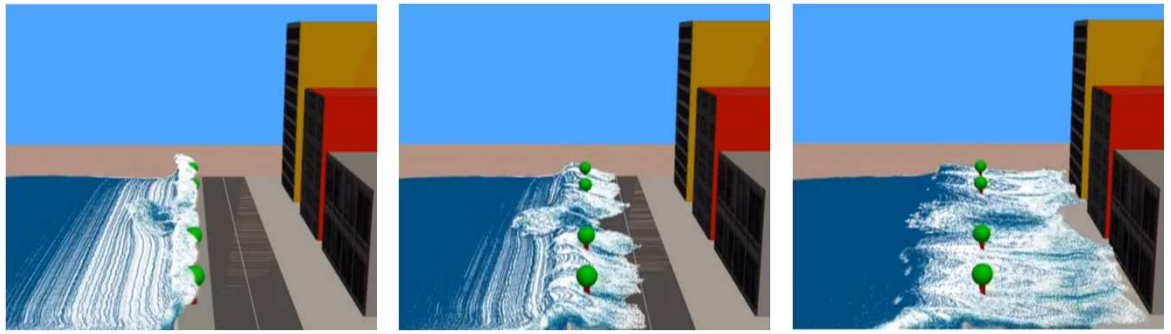

Wave-structure interaction with DualSPHysics (SPH on GPU)

Conclusions

We have been able to simulate 45 million particles on the nVidia TESLA S2050 with 448 cores and 3GB memory. (9 million per GPU in 4 GPUs)

This would be impossible to fit on a single GPU and it was possible without the need of large, expensive cluster of CPUs.

Our new SPH models are capable to deal with real-life engineering CFD problems.



Future developments

Multi-core

-Combination of OpenMP with MPI

GPU

-Include more SPH formulations expensive in time
-Double precision
-New strategies to optimize particle interaction
-New capabilities of CUDA 4.0

Multi-GPU

-Dynamic load balancing
-2D domain decomposition
-Use of Infiniband to decrease CPU-CPU communication
-Test pinned memory to decrease GPU-CPU communication
-CUDA 4.0 to fully investigate inter-GPU communications

- -One thread per GPU limitation removed
- -GPUDirect v2.0
- -Unified virtual addressing UVA