

UNIVERSIDAD DE VIGO
DEPARTAMENTO DE FÍSICA APLICADA
Environmental Physics Laboratory

Universida_{de}Vigo

PhD Thesis

**DualSPHysics: Towards High Performance Computing
using SPH technique**

Memoria Presentada por
José Manuel Domínguez Alonso
para optar al título de DOCTOR POR LA UNIVERSIDAD DE VIGO
CON MENCIÓN INTERNACIONAL
Septiembre, 2014

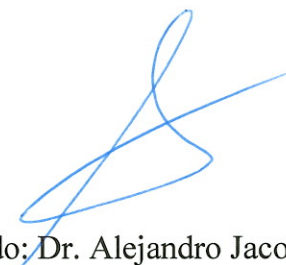
Informe del director

Dr. Alejandro Jacobo Cabrera Crespo, Contratado Juan de la Cierva de la Universidad de Vigo, y Dr. Ramón Gómez Gesteira, Catedrático del Departamento de Física Aplicada de la Universidad de Vigo:


CERTIFICAN

Que la presente memoria "*DualSPHysics: Towards High Performance Computing using SPH technique*", resume el trabajo de investigación realizado, bajo su dirección, por DON JOSÉ MANUEL DOMÍNGUEZ ALONSO en el departamento de Física Aplicada en el programa de doctorado de Ciencias del Clima: Meteorología, Oceanografía Física y Cambio Climático de la Facultad de Ciencias de Ourense para optar al título de "DOCTOR POR LA UNIVERSIDAD DE VIGO CON MENCIÓN INTERNACIONAL".

Y para que conste y en cumplimiento de la legislación vigente, firman el presente informe en Ourense, a 23 de Septiembre del 2014.



Fdo: Dr. Alejandro Jacobo
Cabrera Crespo



Fdo. Dr. Ramón Gómez Gesteira

Acknowledgements / Agradecimientos

En primer lugar me gustaría mostrar mi agradecimiento a Moncho por abrirme las puertas del apasionante mundo de la investigación, por su confianza y su tiempo. A él le debo gran parte de las nuevas experiencias vividas estos años, los lugares que he visitado, las cosas que he aprendido. Aunque también las noches sin dormir tratando de ir siempre un poco más allá.

A Alex, mi director, compañero y amigo. Gracias por su ayuda inestimable y todo su tiempo, tanto en el trabajo como fuera. Sin él, gran parte de lo conseguido no hubiera sido posible.

A todos mis compañeros de ahora y antes: Anxo, Orlando, Fran, Angel, Xurxo, Isabel, Alex y Maruxa, que siempre logran hacer del trabajo un lugar ameno. Sin olvidar a toda la gente del laboratorio.

To people in Manchester: Ben, Georgios, Athanasios and Abouzied, for their hospitality and help when I was away from home.

A mis amigos de Orense: Diego, Noe, Sandra, Noelia, Victor, Dani, Carlos, Edu y Marta, por estar siempre ahí y hacer imposible que me sintiese solo en esta ciudad.

A mis amigos de siempre: David, Hugo, Eva, Txatxo, Arancha, Nuria, Borja, Vero, Alberto, Mar, Dosy, Lucy, Dulci, Juan, Fernando, Ester y Toni, por los innumerables días y noches que he disfrutado de su compañía.

A Patricia por su cariño, su apoyo, su paciencia... por aguantarme que Dios sabe que no es fácil.

Muy en especial a mi madre, mi padre y mi hermana, que sin ellos no sería nada. Por ese amor incondicional que por mucho que lo intente, nunca podré devolverles en la misma medida en que lo recibo.

This work was partially supported by Xunta de Galicia under Axudas de apoio á etapa predoutoral do Plan Galego de Investigación, Innovación e Crecemento 2011-2015, Axudas a grupos de investigación do Campus de Ourense (INOUE2013), project Programa de Consolidación e Estructuración de Unidades de Investigación Competitivas (Grupos de Referencia Competitiva), funded by European Regional Development Fund (FEDER) and Ministerio de Economía y Competitividad under project BIA2012-38676-C03-03.

Abstract

Smoothed Particle Hydrodynamics (SPH) is a numerical method commonly used in Computational Fluid Dynamics (CFD). SPH is an ideal technique to simulate free-surface flows. Its range of application is very wide, including sloshing and flooding events, the design of coastal defences, dams or devices to generate renewable energies... The technique can also be used for engineering purposes in those problems involving the complex interaction between water and structures. In general, all these problems involve large domains that should be solved with fine resolution, which makes the model expensive in terms of computational requirements. This is the reason why these codes should be optimized and accelerated as much as possible.

The aim of this work is to use High Performance Computing to improve a Smoothed Particle Hydrodynamics model in order to develop a SPH code capable of performing simulations of real-life applications at a reasonable time.

The main goal is to develop an optimized version of the open-source code DualSPHysics (<http://dual.sphysics.org>), which can be used both on classic CPUs (Central Processing Unit) and novel GPUs (Graphics Processing Units). DualSPHysics has been designed to be run on multi-core CPUs, which is a relatively common resource, but also on GPUs. The GPU technology has experienced a rapid development during the last few years and constitutes a fast and cheap alternative to classical computation on CPUs. Nevertheless, a single GPU is not enough to run large domains due to memory requirements and huge execution times. Thus, a multi-GPU version of the code has also been developed. In addition, pre-processing and post-processing tools have been developed to take advantage of DualSPHysics capabilities.

SPH codes like DualSPHysics can be split into three main steps; (i) generation of a neighbour list, (ii) computation of forces between particles and (iii) integration in time of the physical quantities of all particles. The step devoted to compute forces consumes more than 90% of the total execution time, whereby it is the key step to be accelerated. However, its implementation and performance depends greatly on the previous step (neighbour list generation) therefore a study about different neighbour list approaches was first carried out. The use of Cell-linked list and Verlet list with several variations is compared,

being the Cell-linked list chosen to be implemented since it provides the best balance between performance and usage of memory.

Four optimizations are implemented for the CPU code in DualSPHysics. The first one applies symmetry in particle interactions, the second one divides the domain into smaller cells, the third one uses SSE instruction and the fourth one uses OpenMP to implement multi-core executions. Three different approaches of the multi-core implementation are presented. The most efficient OpenMP implementation outperforms the single-core by 4.6 using the available 8 logical cores provided by the CPU hardware used in this study.

CUDA (Compute Unified Device Architecture) is used to exploit the huge parallel power of present-day GPUs and several optimizations are presented for the GPU implementations; maximization of occupancy to hide memory latency, reduction of global memory accesses to avoid non-coalesced memory accesses, simplification of the neighbour search, optimization of the interaction kernel and division of the domain into smaller cells to reduce code divergence. The GPU parallel computing developed here can accelerate serial SPH codes with a speedup of 56.2x when using the Fermi GPU, but this speedup rises to 148.8x using the latest GPU GTX Titan. Finally, the speedup of the latest GPU over a multi-core CPU is more than 33x when using an optimised multi-threaded approach.

The multi-GPU approach includes CUDA and MPI (Message Passing Interface) programming languages to combine the parallel performance of several GPUs in a host machine or in multiple machines connected by a network. The multi-GPU implementation has shown an efficiency close to 100% using 128 GPUs of the Barcelona Supercomputing Center, when 8 million particles per GPU have been simulated. Moreover, an application with more than 10^9 particles is presented to show the capability of the code to handle simulations that would require large CPU clusters or supercomputers otherwise.

Finally, an efficient solution was implemented to avoid some problems of precision that can appear when the simulation involves a very large domain and very high resolution.

TABLE OF CONTENTS

TABLE OF CONTENTS.....	I
LIST OF FIGURES	V
LIST OF TABLES	XI
NOMENCLATURE	XIII
1. INTRODUCTION	1
1.1 NUMERICAL MODELING	1
1.2 SMOOTHED PARTICLE HYDRODYNAMICS.....	2
1.3 HIGH PERFORMANCE COMPUTING	3
1.3.1 OpenMP (Open Multi-Processing)	4
1.3.2 MPI (Message Passing Interface)	5
1.3.3 GPGPU (General-Purpose Computing on Graphics Processing Units).....	5
1.4 DUALSPHYSICS PROJECT	7
1.5 THESIS OUTLINE.....	10
2. SPH FORMULATION	13
2.1 THE SMOOTHING KERNEL.....	14
2.2 MOMENTUM EQUATION.....	16
2.2.1 Artificial Viscosity	16
2.2.2 Laminar viscosity and Sub-Particle Scale (SPS) Turbulence	16
2.3 CONTINUITY EQUATION	18
2.4 EQUATION OF STATE.....	19
2.5 PARTICLE MOTION	19
2.6 SHEPARD FILTER	20
2.7 TIME STEPPING.....	20
2.7.1 Verlet Scheme	21
2.7.2 Symplectic Scheme	21
2.7.3 Variable Time Step	22
2.8 BOUNDARY CONDITIONS	23
2.8.1 Dynamic Boundary Condition	23
2.8.2 Periodic Open Boundary Condition.....	23
2.8.3 Pre-imposed Boundary Motion	24
2.8.4 Fluid-driven Objects	24
3. NEIGHBOUR LIST IMPLEMENTATION.....	27

3.1	STEPS OF THE SPH CODE	28
3.2	TESTCASE	29
3.3	DIFFERENT APPROACHES OF NEIGHBOUR LIST	30
4.	CPU ACCELERATION	41
4.1	CPU OPTIMIZATIONS	41
4.1.1	Applying symmetry to particle-particle interaction	41
4.1.2	Splitting the domain into smaller cells.....	42
4.1.3	Using SSE instructions.....	43
4.2	OPENMP IMPLEMENTATION	44
4.3	RESULTS	46
5.	GPU ACCELERATION.....	49
5.1	CUDA PROGRAMMING MODEL	49
5.2	CUDA IMPLEMENTATION.....	51
5.3	GPU OPTIMIZATIONS.....	57
5.3.1	Maximizing the occupancy of GPU	57
5.3.2	Reducing global memory accesses	59
5.3.3	Simplifying the neighbor search	59
5.3.4	Adding a more specific CUDA kernel of interaction	60
5.3.5	Division of the domain into smaller cells	61
5.4	RESULTS	61
5.5	PERFORMANCE WITH THE LATEST GPU (AUGUST 2014).....	65
6.	MULTI-GPU ACCELERATION.....	69
6.1	MPI IMPLEMENTATION	71
6.1.1	Subdivision of the domain	73
6.1.2	Communication among processes.....	75
6.1.3	Dynamic load balancing.....	77
6.2	RESULTS	79
6.2.1	Testcases and hardware.....	79
6.2.2	Applying dynamic load balancing in a homogeneous cluster.....	81
6.2.3	Applying dynamic load balancing in a heterogeneous cluster.....	82
6.2.4	Efficiency and scalability.....	83
6.2.5	Bottlenecks: Loss of efficiency	86
6.2.6	Memory requirements	88
6.3	APPLICABILITY TO REALISTIC PROBLEMS	89
7.	DOUBLE PRECISION.....	93
7.1	THE PROBLEM OF PRECISION.....	93
7.2	SOLUTIONS USING DOUBLE PRECISION.....	96
7.2.1	Solution FullDouble	96
7.2.2	Solution PosDouble.....	96
7.2.3	Solution PosCell.....	96
7.2.4	Solution PosDoubleFast	98
7.3	PERFORMANCE.....	99
8.	CONCLUSIONS AND FUTURE WORK	103
8.1	CONCLUSIONS	103
8.1.1	Neighbour List	103

8.1.2 CPU Acceleration	104
8.1.3 GPU Acceleration	104
8.1.4 Multi-GPU Acceleration	105
8.1.5 Issue of precision	106
8.2 FUTURE WORK	106
A. DUALSPHYSICS DOCUMENTATION	107
A.1 SOURCE FILES.....	107
A.2 COMPILATION	110
A.3 FILES AND FORMAT	110
A.4 RUNNING DUALSPHYSICS	112
B. PRE-PROCESSING TOOLS.....	115
B.1 PARTICLE GENERATION	116
B.1.1 Predefined objects	118
B.1.2 External objects	118
B.1.3 Filling algorithm	119
B.1.4 Other design tools	120
B.2 FLOATING OBJECTS.....	122
B.3 INITIAL CONDITIONS	123
B.4 MOVEMENT DEFINITION	125
B.5 NORMAL VECTORS	126
B.6 EXAMPLES AND PERFORMANCE.....	127
B.6.1 Testcase Sink.....	128
B.6.2 Testcase Mixer	129
B.6.3 Testcase Pump.....	130
B.6.4 Testcase Mini Cooper	130
B.7 REMARKS	131
C. POST-PROCESSING TOOLS	133
C.1 PARTVTK	133
C.2 MEASURETOOL.....	134
C.3 ISO SURFACE	135
C.4 DECIMATE	136
C.5 BOUNDARYVTK.....	137
C.6 MEASUREBOXES	138
C.7 TRACERVTK.....	139
BIBLIOGRAPHY	141
LIST OF PUBLICATIONS.....	153

LIST OF FIGURES

Figure 1-1. Floating-Point Operations per Second for the CPU and GPU (source: CUDA Programming Guide v6.5).....	6
Figure 1-2. DualSPHysics website.	9
Figure 1-3. Number of code lines in the programs of DualSPHysics project.	10
Figure 1-4. Number of individual files in the programs of DualSPHysics project.	10
Figure 2-1 Cubic Spline kernel and its derivative divided by the dimensional factor α_D	15
Figure 2-2. Quintic kernel and its derivative divided by the dimensional factor α_D	15
Figure 3-1. Conceptual diagram summarising the implementation of a SPH code.	27
Figure 3-2. Different instants of the dam break evolution using 300,000 particles.	29
Figure 3-3. Sketch of the Cell-linked list (CLL).	32
Figure 3-4. Sketch of the Verlet list (VL).	34
Figure 3-5. Computational runtime of different approaches for neighbour list. ...	35
Figure 3-6. Memory requirements of different approaches for neighbour list.....	35
Figure 3-7. Improvement in time using VL_C and VL_X compared to CLL . All cases were calculated with $N=31,239$	36
Figure 3-8. Allocated memory in CLL , VL_X and VL_C . All cases were calculated with $N=31,239$	36
Figure 3-9. Improvement comparison between VL_X and VL_C referred to CLL	37
Figure 3-10. Comparison between VL_X with and without kernel gradient correction (KGC). The improvement is referred to CLL	38

Figure 4-1. Interaction cells in 3D without (left) and with (right) symmetry in particle interactions. Each cell interacts with 14 cells (right) instead of 27 (left).	42
Figure 4-2. Sketch of 3D interaction with close cells using symmetry. The volume searched using cells of side $2h$ (left panels) is bigger than using cells of side h (right panels).	43
Figure 4-3. Sketch Pseudocode in C++ showing the force computation between the particles of two cells without vectorial instructions (up) and grouping in blocks of 4 pair-wise of interaction using SSE instructions (down).	44
Figure 4-4. Example of dynamic distribution of cells (in blocks of 4) among 3 execution threads according to the execution time of each cell.	45
Figure 4-5. Speedup achieved on CPU for different number of particles (N) when applying symmetry, the use of SSE instructions. Two different cell sizes ($2h$ and $2h/2$) were considered.	47
Figure 4-6. Speedup achieved on CPU for different number of particles (N) with different OpenMP implementations (using 8 logical threads) in comparison with the most efficient single-core version that includes all the previous optimizations.	48
Figure 5-1. Grid of thread blocks in CUDA (source: CUDA Programming Guide v6.5)	50
Figure 5-2. Memory hierarchy (source: CUDA Programming Guide v6.5)	51
Figure 5-3. Conceptual diagram of the partial (left) and full (right) GPU implementation of the SPH code.	52
Figure 5-4. Example of the Neighbour list procedure.	54
Figure 5-5. Pseudocode of the System update procedure implemented on CPU and GPU.	55
Figure 5-6. Pseudocode of the Particle interaction procedure implemented on CPU and GPU.	56
Figure 5-7. Occupancy of the GPU for different number of registers with a variable and a fixed block size of 256 threads.	58
Figure 5-8. Interaction cells in 3D without symmetry but using 9 ranges of three consecutive cells (right) instead of 27 cells (left).	60
Figure 5-9. Computational runtimes (in seconds) using GTX 480 for different GPU implementations (partial, full and optimized) when simulating 500,000 particles.	63
Figure 5-10. Memory usage for different GPU versions implemented in DualSPHysics.	65
Figure 5-11. Runtimes for different CPU and GPU implementations.	65

Figure 5-12. Runtime for CPU and different GPU cards.	66
Figure 5-13. Speedups of GPU against CPU simulating 1 million particles.	67
Figure 5-14. Computational runtime distribution on CPU and GPU simulating 1 million particles. Neighbour List corresponds to blue bars, Particle Interaction to red bars and System Update to the green bars.	67
Figure 5-15. Maximum number of particles simulated with different GPU cards using DualSPHysics code.	68
Figure 6-1. Scheme of technologies and its scope of application.	70
Figure 6-2. Domain subdivision in four processes.	73
Figure 6-3. Example of subdivision of a domain (halos and edges).	74
Figure 6-4. Scheme of the communications among 3 MPI processes.	76
Figure 6-5. Example of the dynamic balancing scheme between 2 GPUs.	78
Figure 6-6. Testcase1: Dam break flow impacting on a structure.	79
Figure 6-7. Testcase2: Dam break flow.	80
Figure 6-8. Different instants of the simulation of testcase1 when using the dynamic load balancing according to the number of particles.	81
Figure 6-9. Distribution of the fluid particles and execution times of force computation among the 3 GPUs of system #1a using load balancing according to the number of particles.	82
Figure 6-10. Distribution of the fluid particles and execution times of force computation among the 3 different GPUs of system #1b using load balancing according to the number of particles.	82
Figure 6-11. Distribution of the fluid particles and execution times of force computation among the 3 different GPUs of system #1b using load balancing according to the computation time.	83
Figure 6-12. Execution times of the 3 GPUs of the system #1b used individually and together applying dynamic load balancing.	83
Figure 6-13. Speedup for different number of GPUs using strong and weak scaling with the hardware systems #1a, #2 and #3.	85
Figure 6-14. Percentage of time dedicated to tasks exclusive of the multi-GPU executions using the system #3.	86
Figure 6-15. Percentage of the computational time dedicated to specific MPI tasks simulating 16M particles using different number of Tesla M2050 GPUs (left) and simulating different number of particles with 16 Tesla M2050 (right).	87

Figure 6-16. Percentage of time dedicated to tasks exclusive of the multi-GPU executions including the latest improvements (using the system #2).	88
Figure 6-17. Maximum number of particles that can be simulated for the testcase2 with the systems #1a, #2 and #3.	89
Figure 6-18. Realistic dimensions of the oil rig simulated in the application.	90
Figure 6-19. Different instants (2.2s, 3.2s and 10s) of the simulation of a large wave interacting with an oil rig using more than 109 particles.	90
Figure 7-1. Testbed to study problems of precision.	93
Figure 7-2. Different instants of the simulation of the testbed.	95
Figure 7-3. Relative error in the distance between two particles interacting using double and single precision for different particle positions.	95
Figure 7-4. Different instants of the previous simulation improving precision in the position of the particles.	97
Figure 7-5. Relative error in the position of the particles for different distances to zero and using different approaches.	98
Figure 7-6. Loss of efficiency compared with simple precision simulations using a 3D dam-break with 4M particles.	99
Figure 7-7. Percentage of occupancy according to the number of registers and compute capability of GPU.	100
Figure B-1. Generation of a 2D triangle.	117
Figure B-2. Discretization accuracy for different number of particles. The absolute measures of the object are 0.39 x 0.46 x 0.42.	117
Figure B-3. Some predefined objects: box, sphere, cylinder, prism,	118
Figure B-4. Basic shapes “solid” and “face”	118
Figure B-5. Mixer: 3D model (left) and point distribution (right).	119
Figure B-6. Filling an irregular beach with fluid.	120
Figure B-7. Example of rotation and scaling of a 3D model.	121
Figure B-8. Creating a balustrade starting from a primitive element.	121
Figure B-9. Merging objects with different label.	121
Figure B-10. Gravity center and inertia (lower pannel) computed starting from different particle distributions (upper pannel).	122
Figure B-11. Different initial configurations depending on the value of lattice for fluid (blue points) and boundary (black points) particles.	123

Figure B-12. Initial density distribution.	124
Figure B-13. Mixing of two fluids.	124
Figure B-14. Different instants of a pendulum movement (rotational, circular and rectilinear sinusoidal).	125
Figure B-15. Mixer as an example of hierarchy of movements.	126
Figure B-16. Normal vector (n) computation for a triangle.	126
Figure B-17. Normal vector computation for a 3D object.	127
Figure B-18. Sink with floating object (polygons and particles).	128
Figure B-19. Execution runtimes for the Sink.	129
Figure B-20. Mixer (polygons and particles).	129
Figure B-21. Execution runtimes for the Mixer.	129
Figure B-22. Pump (polygons and particles).	130
Figure B-23. Execution runtimes for the Pump.	130
Figure B-24. Mini Cooper (polygons and wire).	131
Figure B-25. Execution runtimes for the Mini Cooper.	131
Figure C-1. Visualisation of density from a fluid block of particles.	133
Figure C-2. Example of graph with wave elevation at a specific position.	134
Figure C-3. Visualises the wave elevation for a slice of fluid.	134
Figure C-4. Conversion of points to surfaces, from particles to isosurface.	135
Figure C-5. Original isosurface of fluid (left) and simplified isosurface by Decimate program with a reduction to 10%.	136
Figure C-6. Floating body movement represented using a box.	137
Figure C-7. Application of MeasureBoxes to measure a flow at complex terrain.	138
Figure C-8. Waves interaction with a coastal structure consisting of antifers and trajectories of fluid particles between antifers.	139

LIST OF TABLES

Table 3-1. Comparison of Cell-linked list (<i>CLL</i>) and Verlet list (<i>VL</i>): percentage of the total runtime of the dam-break simulation using 300,000 particles.....	31
Table 4-1. Speedup achieved on CPU simulating 300,000 particles when using 4 and 8 threads compared to the single CPU version.....	48
Table 5-1. Technical specifications of GPUs according to the compute capability.	58
Table 5-2. List of variables needed to calculate forces.	59
Table 5-3. Improvement achieved on GPU simulating 1 million particles when applying the different GPU optimizations using GTX 480 and Tesla 1060.	62
Table 5-4. Results of the CPU and GPU simulations.....	64
Table 5-5. Specifications of different execution devices.	66
Table 6-1. Features of the different systems used.	80
Table 6-2. Formulae to measure efficiency and scalability.....	84
Table 7-1. Double precision implementations	98
 Table A-1. List of source files of DualSPHysics code.....	 107
Table A-2. List of source files of DualSPHysics code not related to the SPH solver.	108
Table A-3. List of source files of DualSPHysics code for the SPH execution. .	108
Table A-4. List of source files of DualSPHysics code for the SPH execution on CPU.	109
Table A-5. List of source files of DualSPHysics code for the SPH execution on GPU.	109
Table A-6. List of execution parameters of DualSPHysics.....	113

Table B-1. Features of the cases.....	127
---------------------------------------	-----

NOMENCLATURE

Symbol	Definition
F	Abitrary function.
c	Speed of sound.
dp	Particle spacing.
f	Force per unit mass.
g	Gravity force.
h	Smoothing length.
a	Particle where the interpolation is performed.
b	Neighbouring particle.
i,j,k	Unit vectors
m	Particle mass
M	Mass of the floating object
P	Particle pressure
q	Non-dimensional distance between particles (r/h)
r	Distance between particles
\mathbf{r}	Position vector
\mathbf{R}_0	Centre of mass of the floating object
t	Time
\mathbf{v}	Velocity
W	Smoothing kernel
\tilde{W}	Zeroth order corrected kernel
α	Artificial viscosity parameter
α_D	Kernel normalisation factor
γ	Polytropic index
δ	Dirac function
Δt	Time step
μ	Dynamic viscosity
ν_0	Laminar kinematic viscosity
Π	Artificial viscosity term
ρ	Particle density
ρ_0	Reference density
Ω	Rotational velocity of the floating object

1. INTRODUCTION

In this first chapter, a general overview of the numerical methods and, more specifically, of the Smoothed Particle Hydrodynamics (SPH) method is provided. The advantages and disadvantages of the SPH methods when compared with other methods are also described. Furthermore, different High Performance Computing techniques are presented to accelerate the SPH method. Finally, the DualSPHysics code is presented.

1.1 NUMERICAL MODELING

Nature can be modelled looking for analytical solutions of the equations that define a system (or mathematical model). Once the equations are validated, the behaviour of the system can be predicted tuning some parameters and imposing a set of initial conditions. The numerical modelling looks for solving these equations in a numerical way instead of analytically. So that, designing algorithms that use numbers and simple mathematical rules that can simulate complex processes of the real world. The numerical simulation is a powerful tool that allows for understanding the behaviour of complex systems and even for predicting their evolution starting from initial conditions. Numerical modelling becomes more important with the arrival of the computers since these machines can perform thousands of million mathematical operations per second. This allows for the simulation of very complex systems in few time using simple mathematical operations.

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that studies the behaviour of the fluids using numerical modelling. The main advantage of this technique is the capability to simulate complex scenarios and provide physical data that can be difficult, or even impossible, to measure in a real model. Despite of the accuracy of the numerical models, these cannot replace the

construction of scale models, but they can reduce significantly the number of physical tests. This leads to an important saving since the construction of physical models is very expensive and slow.

There are two numerical approaches to describe the fluid motion; Eulerian and Lagrangian. The Eulerian approach solves the equations at the fixed nodes of a mesh. In the Lagrangian description, the positions where equations are solved move with the fluid and a fixed mesh is not used. The meshbased methods (finite elements, finite differences and finite volumes) are currently very robust, well developed and have been applied to a wide range of applications providing highly accurate results. These meshbased methods are ideal for systems where the domain is perfectly defined and for simulations where the boundaries remain fixed. However the creation of the mesh can be very inefficient if the system is complex. In recent years, numerous meshless methods have appeared and grown in popularity as they can be applied to problems that are highly nonlinear in arbitrarily complex geometries and are difficult for mesh-based methods. Within the meshless methods now available, Smoothed Particle Hydrodynamics (SPH) is, possibly, the most popular and has attained the required level of maturity to be used for engineering purposes.

1.2 SMOOTHED PARTICLE HYDRODYNAMICS

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian meshless method that is increasingly used for an extensive range of applications within the field of Computational Fluid Dynamics (CFD). Originally invented for astrophysics during the seventies [Lucy, 1977; Gingold and Monaghan, 1977], it has been applied in many different fields including fluid dynamics and solid mechanics. The method uses particles to represent a fluid and these particles move according to the governing dynamics. More complete description of the SPH formulation is found in Chapter 2. When simulating free-surface flows, the Lagrangian nature of SPH allows the domain to be multiply-connected, with no need of a special treatment of the surface, making the technique ideal for studying violent free-surface motion.

SPH has been used to describe a variety of free-surface flows (wave propagation over a beach, plunging breakers, impact on structures and dam breaks). [Monaghan, 1994] presented the first attempt to study free-surface flows. Monaghan also studied the behaviour of gravity currents ([Monaghan, 1996]),

solitary waves ([Monaghan et al., 1999]) and wave arrival at a beach ([Monaghan and Kos, 1999]). Later on, the model was applied to the study of the wave-structure interaction such as in [Colagrossi and Landrini, 2003] that considered the study of interfacial flows. The classical dam-break problem was also studied in 3D by [Gómez-Gesteira and Dalrymple, 2004]. Within the area of coastal engineering, SPH was firstly employed to study wave-breakwater interaction in [Gotoh et al., 2004] and [Shao, 2005], and to predict wave impact pressure due to sloshing waves in [Khayyer and Gotoh, 2009].

However, the high computational cost is an important drawback of this technique. Thus, a short period of physical time applications requires a large execution time when running on a single Central Processing Unit (CPU) due to the large number of interactions for each particle at each timestep. This has hindered the development of SPH and its industrial use to solve real problems. Hence, the ability to perform computations involving millions of particles in a reasonable time is essential to perform simulations that are industrially relevant. However, this is only possible if some hardware acceleration techniques are employed.

1.3 HIGH PERFORMANCE COMPUTING

High Performance Computing (HPC) is a very dynamic field that deals with the study and usage of new computational resources and technologies. Its aim is to solve very complex problems that require high computational capacity so that cannot be solved with conventional computer systems, making necessary the use of clusters or supercomputers. A supercomputer is a computer with a very high computational speed dedicated on the execution of parallel operations and designed for intensive computation. These are extremely expensive machines. On the other hand, a cluster is a collection of computers connected through a high speed network and considered as a single machine. This is a cheaper option as it can be integrated by more conventional machines, which currently have high performance at very low prices. They also offer the possibility to extend their computing capacity, theoretically unlimited, by simply adding more computers.

HPC includes multiple techniques of parallel computing and distributed computing. In the main, parallel computing consists of executing several operations simultaneously.

This parallelism can be applied at instruction-level, since current processors divide the execution of an instruction in several stages, so they can keep running several instructions at different stages (instruction pipelines). In addition, the superscalar microprocessors can execute multiple instructions simultaneously when there is no data dependency among them. The task-level parallelism consists of dividing a volume of data into different computing nodes to perform the same set of operations. Finally, the task-level parallelism distributes the execution of different computations, on the same or different data, among multiple processing units.

Parallel computing can be applied with hardware of shared memory in which a machine has one or more processors that use the same memory space. In this case the more extended tools of programming are *pthread*s [Buttlar et al., 1996] and OpenMP [Chandra et al., 1996; Chandra et al., 2002] that can be considered as the standard for this kind of systems with shared memory, due to the advantages over other standard parallel-programming models [Dagum and Menon, 1998]. Parallel computing can be also applied with systems of distributed memory in which each processor is associated with a memory space and cannot directly access to the memory associated with other processors. In these systems, the data exchange between processors must be carried out explicitly using a message passing model. The most common options for this kind of programming are PVM [Geist et al., 1994], BSP [Bisseling, 2004] and MPI [Pacheco, 1996; Snir et al., 1998; Gropp et al., 1999] that is the standard one.

It is also important to note that in recent years, the use of special-purpose processors as general purpose parallel systems are becoming increasingly important in HPC. Hence, Processing Graphics Units (GPU), Digital Signal Processors (DSP), Field Programmable Gate Array (FPGA) and other systems are used as scientific computer systems rather than for its original purpose.

The following explains in more detail the main HPC techniques used to accelerate SPH.

1.3.1 OpenMP (Open Multi-Processing)

OpenMP [<http://www.openmp.org>] is a model of parallel programming for systems of shared memory. It provides an Application Program Interface (API) in C, C++ and Fortran applications. OpenMP is a portable and flexible programming interface where multiple threads of execution perform tasks

defined by OpenMP directives. Its implementation does not involve major changes in the code. Using OpenMP, multiple threads for a process can be easily created. These threads are distributed among all the cores of the CPU sharing the memory. Thus, there is no need to duplicate data or to transfer information among threads. For these reasons OpenMP is the best option to optimize the performance of the multiple cores of the current CPUs [Clark, 1998].

1.3.2 MPI (Message Passing Interface)

MPI is a message-passing library specification for parallel computers and clusters where a distributed memory system is used. MPI is not a language or a compiler or a specific implementation, it simply defines a library of functions that can be called from C, C++, and Fortran programs. In this parallel programming model, an execution consists of one or more processes that communicate by calling routines of a library to send and receive messages among processes. Although designed for distributed memory systems, its use with shared memory systems can lead to an improvement since MPI encourages memory locality. The use of MPI is typically combined with OpenMP in clusters by using a hybrid communication model. In this way, within each machine, the processors directly access the shared memory and the message exchange with MPI is used to share information among processes of different machines.

The first implementation of MPI standard was MPICH [<http://www-unix.mcs.anl.gov/mpi/mpich1>]. Other implementations are LAM-MPI [<http://www.lam-mpi.org/>] and more recently, OpenMPI [<http://www.open-mpi.org>] that is an open-source distribution of the MPI2 specification.

1.3.3 GPGPU (General-Purpose Computing on Graphics Processing Units)

GPGPU involves the study and use of parallel computing ability of a GPU to perform general purpose programs. Graphics Processing Units are powerful parallel processors originally designed for graphics rendering. Due to the development of the video games market and multimedia, their computing power has increased much faster than CPUs (see Figure 1-1). Therefore GPUs can be used for scientific applications achieving speedups of 100x or more. This joined to their very low cost and that GPUs can be used on a personal computer made GPGPU very popular in recent years [Owens et al., 2007; Nickolls and Dally, 2010]. In fact, new computation centres based on GPUs are emerging driven by

their computing power and comparatively low energy costs per FLOP (Floating-point Operations Per Second) [McInstosh-Smith et al., 2012]. Indeed, the current number two of the TOP500 List of the world's top supercomputers released in June 2014 [<http://www.top500.org/lists/2014/06>] is Titan, a Cray XK7 system that has 560,640 processors, including 18,688 Nvidia K20x accelerator GPU cards.

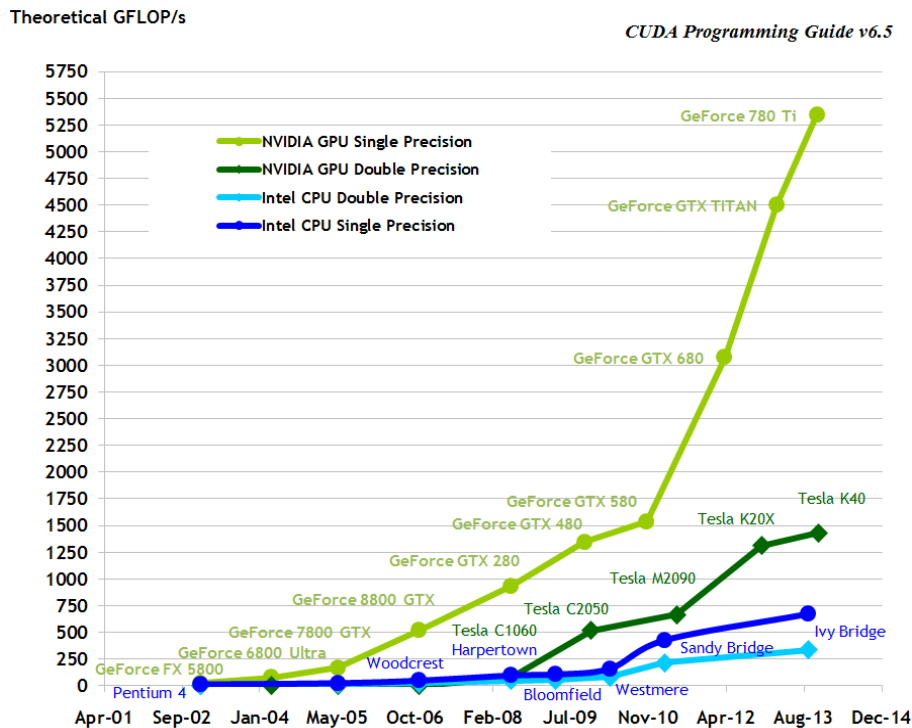


Figure 1-1. Floating-Point Operations per Second for the CPU and GPU (source: CUDA Programming Guide v6.5).

GPUs are optimized for floating-point parallel operations, it is important to note that not all applications are suitable for GPU, only those that exhibit a high degree of parallelism. In addition, the features of the GPU architecture need to be taken into account to obtain the maximum performance. While CPUs are designed for an efficient random memory access, GPUs provide a more restrictive memory access and a careful usage of the memory hierarchy is fundamental. This requires a new implementation of the algorithms used in CPU for an efficient use in GPUs.

Much of the success of GPGPU is the appearance of general purpose programming languages and APIs such as Brook and CUDA since they provided an easier access to the computing power of these devices. Brook was a compiler and runtime implementation of a stream programming language for modern graphics hardware of ATI Technologies. CUDA (Compute Unified Device

Architecture) is both a programming environment and language for parallel computing specifically for Nvidia GPUs [Nickolls et al., 2008; CUDA Programing Guide]. Currently CUDA is the most popular programming graphics model due to the large amount of documentation and utilities that can be found in the CUDA web (<https://developer.nvidia.com/cuda-zone>).

The framework called OpenCL (Open Computing Language) [Khronos, 2009] is becoming increasingly important in GPGPU. OpenCL is a framework to code programs that are executed across heterogeneous platforms including GPUs, CPUs, DSPs, FPGAs and other processors. It is an open standard maintained by Khronos Group and adopted by the most important technology companies such as Intel, AMD and Nvidia.

1.4 DUALSPHYSICS PROJECT

SPHysics was an open-source SPH model developed by researchers at the Johns Hopkins University (US), the University of Vigo (Spain), the University of Manchester (UK) and the University of Rome, La Sapienza. The software is available to download from www.sphysics.org, a complete guide of the FORTRAN code is found in [Gómez-Gesteira et al., 2012a; Gómez-Gesteira et al., 2012b]. The SPHysics code was validated for different problems of wave breaking [Dalrymple and Rogers, 2006], dam-break behaviour [Crespo et al., 2008], interaction with coastal structures [Gómez-Gesteira and Dalrymple, 2004] or with a moving breakwater [Rogers et al., 2010]. A shallow water version was also developed [Vacondio et al., 2012; Vacondio et al., 2013a]. Although SPHysics allows modelling problems with high resolution, the main problem for the application to real engineering problems is its high computational cost, therefore SPHysics is rarely applied to large domains. Hardware acceleration and parallel computing are required to make codes such as SPHysics more useful and versatile.

The code DualSPHysics has been developed by starting from the FORTRAN SPH formulation implemented in SPHysics, this code was considered robust and reliable but not optimised for large simulations. DualSPHysics is implemented in C++ and CUDA and is designed to launch simulations either on multiple CPUs using OpenMP or on a GPU. The GPU portion of DualSPHysics implements the most appropriate parallelisation to maximise speedup during particle interaction computation.

The code can be executed either on the CPU or on the GPU since all computations have been implemented both in C++ for CPU simulations and in CUDA for the GPU simulations. The philosophy underlying the development of DualSPHysics is that most of the source code is common to CPU and GPU which makes debugging straightforward as well as the code maintenance and new extensions. This allows the code to be run on workstations without a CUDA-enabled GPU, using only the CPU implementation. On the other hand, the resulting codes should be necessarily different since code developers have considered efficient approaches for every processing unit. As explained below, the same programming strategy can be efficient on a CPU but inefficient on a GPU (or vice versa). Thus, comparisons between the performances of both approaches are more reliable since appropriate optimisations have been considered for every case.

The first rigorous validation of the GPU implementation of DualSPHysics code was presented in [Crespo et al., 2011]. The code has been developed to simulate real-life engineering problems using SPH models such as the computation of forces exerted by large waves on the urban furniture of a realistic promenade ([Barreiro et al., 2013]) or the study of the run-up in an existing armour block sea breakwater ([Altomare et al., 2014a]). Other recent examples of the study of wave-structure interaction, by means of the DualSPHysics model, are the works of [Ren et al., 2014], where the SPH model is validated against other available numerical results and against experimental data for wave damping over porous seabed with different levels of permeability. Other recent example is the work of [St-Germain et al., 2014] to investigate the hydrodynamic forces induced by the impact of rapidly advancing tsunami like hydraulic bores.

DualSPHysics is an open-source code developed and redistributed under the terms of the GNU General Public License as published by the Free Software Foundation (www.gnu.org/licenses/). The software is available to free download at www.dual.sphysics.org (Figure 1-2). Along with the source code, documentation that describes the compilation and execution of the source files is also distributed. This documentation has been created using the documentation system Doxygen (www.doxygen.org). One of the purposes of this code is to encourage other researchers to try SPH. Most downloads to date have been registered by researchers and students that have conducted their research on fluid dynamics using SPH models. Furthermore, the code has been downloaded not

only by students and researchers from universities and institutes but also by companies with industrial interests.

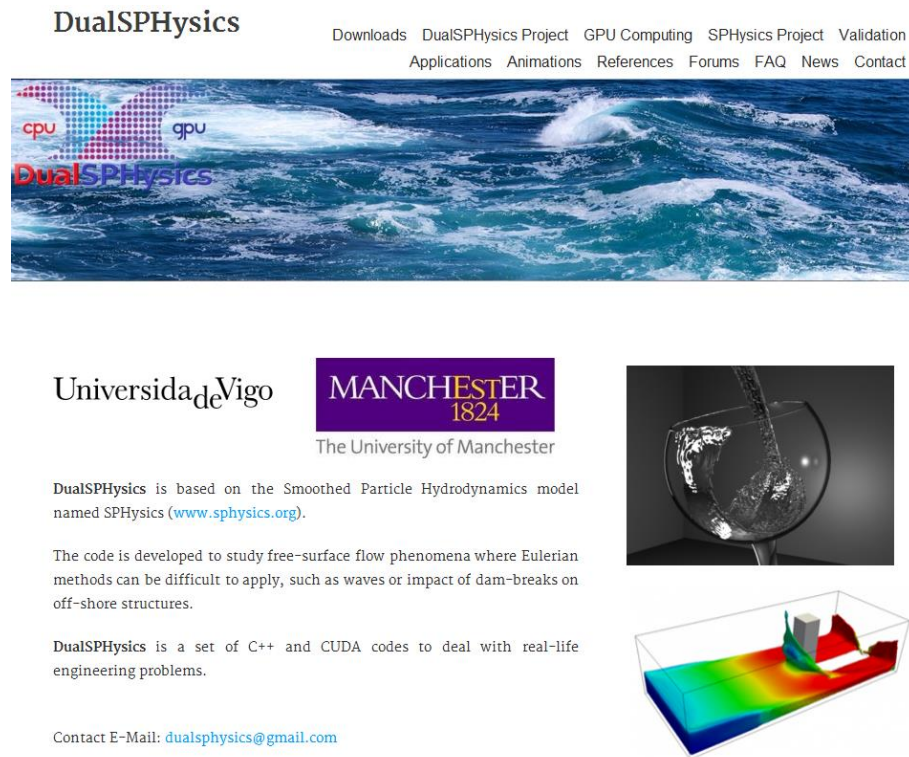


Figure 1-2. DualSPHysics website.

DualSPHysics package includes not only the source files of the SPH solver but also some advanced pre-processing tools to create more complex geometries and post-processing tools to analyse easily numerical results. Any complex geometry can be loaded from different format files such as .cad, .3ds, .stl, .ply, .dwg, .dxf, .shp, .igs, .vtk, .csv... and then converted into SPH particles. For example, a CAD file is converted into particles representing the boundary starting from a triangulation of the object's surface, followed by a filling algorithm. The post-processing tools allow the computation of magnitudes of interest such as vorticity at different planes, forces exerted on different objects, maximum wave heights or just plotting the different physical quantities of the particles.

In order to give an idea about the size of the DualSPHysics project, Figure 1-3 and Figure 1-4 shows the number of code lines and files (.cpp, .h, .cu) that are integrated in the DualSPHysics project. This includes the SPH solver (Appendix A) and pre-processing (Appendix B) and post-processing (Appendix C) tools. It can be noticed that most of the developed code is shared among several codes being 172 different files with around 80,500 code lines.

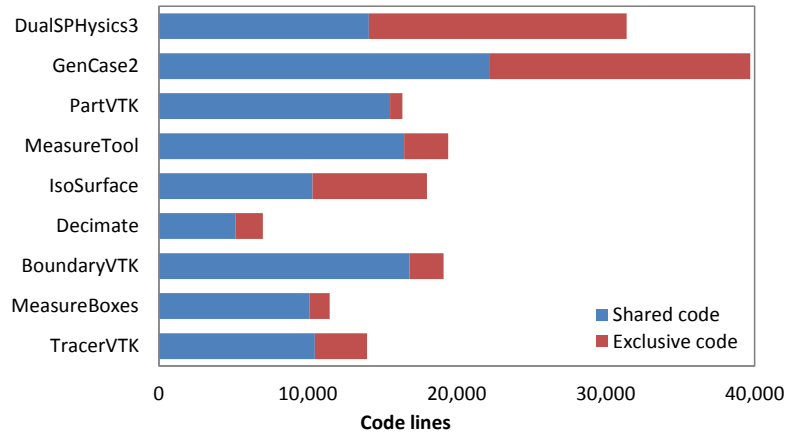


Figure 1-3. Number of code lines in the programs of DualSPHysics project.

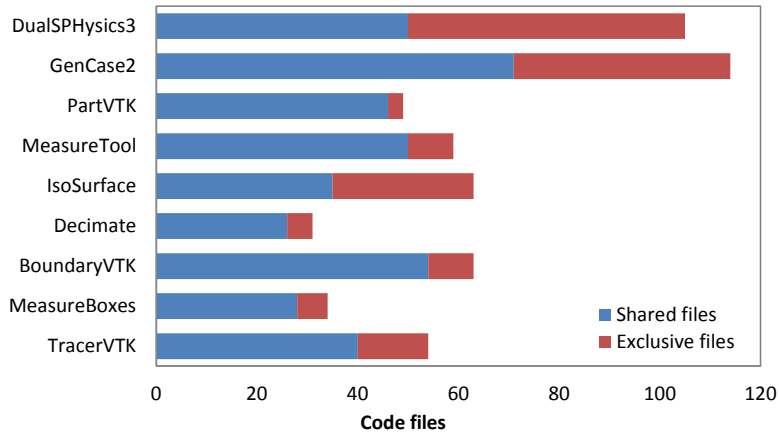


Figure 1-4. Number of individual files in the programs of DualSPHysics project.

1.5 THESIS OUTLINE

The thesis provides a description of the DualSPHysics code and its implementation using different acceleration approaches. It is organized in a total of 8 chapters that are briefed as follows:

Chapter 1 introduces background knowledge of numerical simulation. The main features of the SPH method are briefed. Some general ideas of HPC are described. DualSPHysics code associated with this thesis is introduced.

Chapter 2 provides fundamentals and basic concepts of the SPH method such as integral interpolants, smoothing kernels, the governing equations, time step algorithm and solid boundaries treatment.

Chapter 3 describes the main steps of the SPH simulation and its implementation in DualSPHysics. More detailed is focused on the creation of the neighbour list of the code, which is based on the journal paper [Domínguez et al., 2011a].

Chapter 4 deals with different strategies for CPU optimizations applied to DualSPHysics. Implementation following OpenMP is addressed and results of the performance are shown. This chapter is based on the journal paper [Domínguez et al., 2013a].

Chapter 5 deals with different strategies for GPU optimizations applied to DualSPHysics. Some of the GPU optimizations applied here present not only the suggested basic optimizations described in the CUDA manuals, but also other GPU optimizations intrinsic to the SPH method. Their impact on the efficiency achieved with different GPU architectures is also shown. GPU performance is also compared to CPU multi-core. Implementation with CUDA is also described. This chapter is based on the journal papers [Crespo et al., 2011] and [Domínguez et al., 2013a].

Chapter 6 presents a novel SPH implementation that utilizes MPI and CUDA to combine the power of different devices making possible the execution of SPH on heterogeneous clusters. Specifically, the proposed implementation enables communications and coordination among multiple CPUs, which can also host GPUs, making possible multi-GPU executions. This chapter is based on the journal paper [Domínguez et al., 2013b].

Chapter 7 addresses the issue of precision and solutions using double precision are presented for GPU computing looking for the minimum loss of performance. This chapter is based on the proceedings paper [Domínguez et al., 2014].

Chapter 8 draws together conclusions and ongoing research.

Appendix A contains all the DualSPHysics documentation with a summary of the source files, how to compile and run the code and description of the input and output files and their format. This appendix is based on the journal paper [Crespo et al., 2014].

Appendix B describes the pre-processing tool that creates the configuration that will be loaded by the SPH solver as initial condition for the simulation. This appendix is based on the proceedings paper [Domínguez et al., 2011b].

Appendix C describes the post-processing tools that help to analyse the numerical results and to visualise the simulation.

2. SPH FORMULATION

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian meshless method. The technique discretises a continuum using a set of material points or *particles*. When used for the simulation of fluid dynamics, the discretised Navier-Stokes equations are locally integrated at the location of each of these particles, according to the physical properties of surrounding particles. The set of neighbouring particles is determined by a distance based function, either circular (two-dimensional) or spherical (three-dimensional), with an associated characteristic length or *smoothing length* often denoted as h . At each time-step new physical quantities are calculated for each particle, and they then move according to the updated values.

The conservation laws of continuum fluid dynamics are transformed from their partial differential form to a form suitable for particle based simulation using integral equations based on an interpolation function, which gives an estimate of values at a specific point. Typically this function is referred to as the kernel function (W) and can take different forms, with the most common being cubic or quintic. Any function $F(\mathbf{r})$ is defined at \mathbf{r}' by the integral approximation

$$F(\mathbf{r}) = \int F(\mathbf{r}')W(\mathbf{r} - \mathbf{r}', h)d\mathbf{r}' \quad (2.1)$$

The smoothing kernel must fulfil several properties [Monaghan, 1992; Liu, 2003], such as positivity inside a defined zone of interaction, compact support, normalization and monotonically decreasing with distance and differentiability. For a more complete description of SPH, the reader is referred to [Monaghan, 2005; Violeau, 2012].

The function F in Eq. 2.1 can be approximated in a non-continuous, discrete form, based on the set of particles. In this case the function is interpolated at a

particle (a) where a summation is performed over all the particles that fall within the region of compact support, as defined by the smoothing length h

$$F(\mathbf{r}_a) \approx \sum_b F(\mathbf{r}_b) W(\mathbf{r}_a - \mathbf{r}_b, h) \Delta v_b \quad (2.2)$$

where Δv_b is the volume of a neighbouring particle (b). If $\Delta v_b = m_b / \rho_b$, with m and ρ being the mass and the density of particle b respectively then Eq. 2.2 becomes

$$F(\mathbf{r}_a) \approx \sum_b F(\mathbf{r}_b) \frac{m_b}{\rho_b} W(\mathbf{r}_a - \mathbf{r}_b, h) \quad (2.3)$$

2.1 THE SMOOTHING KERNEL

Performance of an SPH model depends heavily on the choice of the smoothing kernel. Kernels are expressed as a function of the non-dimensional distance between particles (q), given by $q = r/h$, where r is the distance between any two given particles a and b and the parameter h (the smoothing length) controls the size of the area around particle a in which neighbouring particles are considered. In the text that follows, only kernels with an influence domain of $2h$ ($q \leq 2$) will be considered. Within DualSPHysics, the user is able to choose from one of the following kernel definitions:

a) Cubic spline

$$W(r, h) = \alpha_D \begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & 0 \leq q \leq 1 \\ \frac{1}{4}(2-q)^3 & 1 \leq q \leq 2 \\ 0 & q \geq 2 \end{cases} \quad (2.4)$$

where by α_D is equal to $10/7\pi h^2$ in 2-D and $1/\pi h^3$ in 3-D.

The tensile correction method, proposed by [Monaghan, 2000], is only actively used in the cases kernels whose first derivative goes to zero with the particle

distance q . The shape of this function and its derivative can be observed in Figure 2-1.

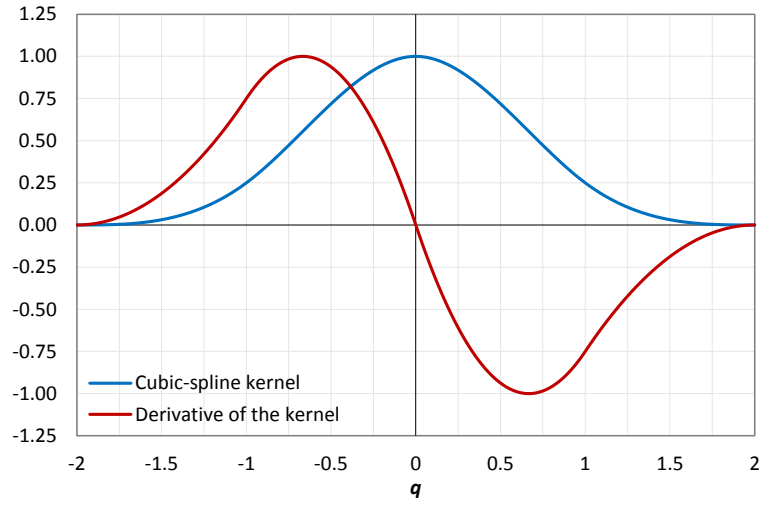


Figure 2-1 Cubic Spline kernel and its derivative divided by the dimensional factor α_D .

b) Quintic [Wendland, 1995]

$$W(r, h) = \alpha_D \left(1 - \frac{q}{2}\right)^4 (2q + 1) \quad 0 \leq q \leq 2 \quad (2.5)$$

where α_D is equal to $7/4\pi h^2$ in 2-D and $21/16\pi h^3$ in 3-D. The shape of this function and its derivative can be observed in Figure 2-2.

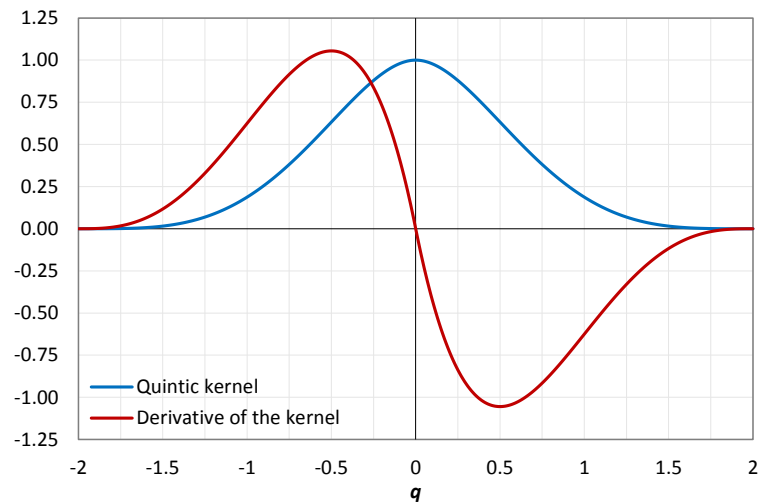


Figure 2-2. Quintic kernel and its derivative divided by the dimensional factor α_D .

2.2 MOMENTUM EQUATION

The momentum conservation equation in a continuum is

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho} \nabla P + \mathbf{g} + \mathbf{I} \quad (2.6)$$

where \mathbf{I} refers to dissipative terms and \mathbf{g} is gravitational acceleration. DualSPHysics offers different options for including the effects of dissipation.

2.2.1 Artificial Viscosity

The artificial viscosity scheme, proposed by [Monaghan, 1992], is a common method within fluid simulation using SPH due primarily to its simplicity. In SPH notation, Eq. (2.6) can be written as

$$\frac{d\mathbf{v}_a}{dt} = -\sum_b m_b \left(\frac{P_b}{\rho_b^2} + \frac{P_a}{\rho_a^2} + \Pi_{ab} \right) \nabla_a W_{ab} + \mathbf{g} \quad (2.7)$$

Where P_k and ρ_k are the pressure and density that correspond to particle k (as evaluated at a or b). The viscosity term Π_{ab} is given by

$$\Pi_{ab} = \begin{cases} \frac{-\alpha \overline{c_{ab}} \mu_{ab}}{\rho_{ab}} & \mathbf{v}_{ab} \cdot \mathbf{r}_{ab} < 0 \\ 0 & \mathbf{v}_{ab} \cdot \mathbf{r}_{ab} > 0 \end{cases} \quad (2.8)$$

where $\overline{\rho_{ab}} = 0.5(\rho_a + \rho_b)$, $\mathbf{r}_{ab} = \mathbf{r}_a - \mathbf{r}_b$ and $\mathbf{v}_{ab} = \mathbf{v}_a - \mathbf{v}_b$ with \mathbf{r}_k and \mathbf{v}_k being the particle position and velocity respectively. $\mu_{ab} = h \mathbf{v}_{ab} \cdot \mathbf{r}_{ab} / (r_{ab}^2 + \eta^2)$, $\overline{c_{ab}} = 0.5(c_a + c_b)$ is the mean speed of sound, α is a coefficient that needs to be tuned in order to introduce the proper dissipation and $\eta^2 = 0.01h^2$ avoids numerical divergence when the distance between particles tends to zero.

2.2.2 Laminar viscosity and Sub-Particle Scale (SPS) Turbulence

Laminar viscous stresses in the momentum equation can be expressed as [Lo and Shao, 2002]

$$(\nu_0 \nabla^2 \mathbf{v})_a = \sum_b m_b \left(\frac{4\nu_0 \mathbf{r}_{ab} \cdot \nabla_a W_{ab}}{(\rho_a + \rho_b)(r_{ab}^2 + \eta^2)} \right) \mathbf{v}_{ab} \quad (2.9)$$

where ν_0 is kinematic viscosity (typically $10^{-6} \text{ m}^2\text{s}$ for water). In SPH discrete notation this can be expressed as

$$\frac{d\mathbf{v}_a}{dt} = -\sum_b m_b \left(\frac{P_b}{\rho_b^2} + \frac{P_a}{\rho_a^2} \right) \nabla_a W_{ab} + \mathbf{g} + \sum_b m_b \left(\frac{4\nu_0 \mathbf{r}_{ab} \cdot \nabla_a W_{ab}}{(\rho_a + \rho_b)(r_{ab}^2 + \eta^2)} \right) \mathbf{v}_{ab} \quad (2.10)$$

The concept of the Sub-Particle Scale (SPS) was first described by [Gotoh et al., 2001] to represent the effects of turbulence in their Moving Particle Semi-implicit (MPS) model. The momentum conservation equation is defined as

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho} \nabla P + \mathbf{g} + \nu_0 \nabla^2 \mathbf{v} + \frac{1}{\rho} \nabla \cdot \bar{\boldsymbol{\tau}} \quad (2.11)$$

where the laminar term is treated as per Eq. 2.9 and $\bar{\boldsymbol{\tau}}$ represents the SPS stress tensor. Favre-averaging is needed to account for compressibility in weakly compressible SPH [Dalrymple and Rogers, 2006] where eddy viscosity assumption is used to model the SPS stress tensor with Einstein notation for the shear stress component in directions i and j

$$\bar{\tau}_{ij} = \nu_t \left(2S_{ij} - \frac{2}{3} k \delta_{ij} \right) - \frac{2}{3} C_I \Delta^2 \delta_{ij} |S_{ij}|^2,$$

where $\bar{\tau}_{ij}$ is the sub-particle stress tensor, $\nu_t = [C_s \Delta l]^2 |S|$ the turbulent eddy viscosity, k the SPS turbulence kinetic energy, C_s the Smagorinsky constant (0.12), $C_I = 0.0066$, Δl the particle to particle spacing and $|S| = 0.5(2S_{ij}S_{ij})$ where S_{ij} is an element of the SPS strain tensor. [Dalrymple and Rogers, 2006] introduced SPS into weakly compressible SPH using Favre averaging, Eq. 2.11 can be rewritten as

$$\begin{aligned} \frac{d\mathbf{v}_a}{dt} = & -\sum_b m_b \left(\frac{P_b}{\rho_b^2} + \frac{P_a}{\rho_a^2} \right) \nabla_a W_{ab} + \mathbf{g} \\ & + \sum_b m_b \left(\frac{4\nu_0 \mathbf{r}_{ab} \cdot \nabla_a W_{ab}}{(\rho_a + \rho_b)(r_{ab}^2 + \eta^2)} \right) \mathbf{v}_{ab} + \\ & + \sum_b m_b \left(\frac{\bar{\tau}_{ij}^b}{\rho_b^2} + \frac{\bar{\tau}_{ij}^a}{\rho_a^2} \right) \nabla_a W_{ab} \end{aligned} \quad (2.12)$$

where the superscripts refer to particles a and b .

2.3 CONTINUITY EQUATION

Throughout the duration of a weakly-compressible SPH simulation (as presented herein) the mass of each particle remains constant and only their associated density fluctuates. These density changes are computed by solving the conservation of mass, or continuity equation, in SPH form:

$$\frac{d\rho_a}{dt} = \sum_b m_b \mathbf{v}_{ab} \cdot \nabla_a W_{ab} \quad (2.13)$$

Within DualSPHysics it is also possible to apply a *delta-SPH* formulation, which introduces a diffusive term [Molteni and Colagrossi, 2009] to reduce density fluctuations

$$\frac{d\rho_a}{dt} = \sum_b m_b \mathbf{v}_{ab} \cdot \nabla_a W_{ab} + 2\delta h \sum_b m_b \overline{c_{ab}} \left(\frac{\rho_a}{\rho_b} - 1 \right) \frac{1}{\mathbf{r}_{ab}^2 + \eta^2} \cdot \nabla_a W_{ab} \quad (2.14)$$

where $\overline{c_{ab}} = 0.5(c_a + c_b)$ and δ is the *delta-SPH* coefficient. This technique is designed to filter relatively large wave numbers from the density field while solving for the conservation of mass of each particle, therefore reducing noise throughout the system of particles. The term can be expanded into a first and second order contributions, where the second order corresponds to its diffusive nature and the first order is approximately zero if the kernel is complete [Antuono et al., 2012]. However, at open boundaries, where a non-complete interpolation kernel is inevitably present, the first order term originates a net contribution. For this reason, it is advised that the *delta-SPH* scheme is disabled for cases that rely on hydrostatic equilibrium. If the case represents a very dynamic situation the term contributes with a force that may be several orders of magnitude smaller than the pressure and viscous terms, not contributing to a significant degradation of the solution. A *delta-SPH* (δ) coefficient of 0.1 is recommended for most applications.

2.4 EQUATION OF STATE

Following the work of [Monaghan, 1994], the fluid in the SPH formalism defined in DualSPHysics is treated as weakly compressible and an equation of state is used to determine fluid pressure based on particle density. The compressibility is adjusted so that the speed of sound can be artificially lowered; this means that the size of time step taken at any one moment (which is determined according to a Courant condition, based on the currently calculated speed of sound for all particles) can be maintained at a reasonable value. Such adjustment however, restricts the sound speed to be at least ten times faster than the maximum fluid velocity, keeping density variations to within less than 1%, and therefore not introducing major deviations from an incompressible approach. Following [Monaghan et al., 1999] and [Batchelor, 1974], the relationship between pressure and density follows the expression

$$P = B \left[\left(\frac{\rho}{\rho_0} \right)^\gamma - 1 \right] \quad (2.15)$$

where $\gamma=7$, $B=c_o^2\rho_0/\gamma$ where $\rho_0=1000\text{ kg m}^{-3}$ is the reference density and $c_o = c(\rho_o) = \sqrt{(\partial P/\partial \rho)|_{\rho_o}}$ which is the speed of sound at the reference density.

2.5 PARTICLE MOTION

Particles are moved according to a method proposed by Monaghan and referred to as XSPH [Monaghan, 1989]. This aims to move particles with a velocity close to the average of the velocity of all particles in their neighbourhood in order to assure a more ordered flow and to prevent penetration between continua, particles are therefore moved using

$$\frac{d\mathbf{r}_a}{dt} = \mathbf{v}_a + \varepsilon \sum_b \frac{m_b}{\rho_{ab}} \mathbf{v}_b W_{ab} \quad (2.16)$$

where ε is a problem specific parameter ranging from 0 to 1 and $\overline{\rho_{ab}} = 0.5(\rho_a + \rho_b)$.

2.6 SHEPARD FILTER

The Shepard filter is a correction to the density field that can be applied every M_s time steps according to the following procedure

$$\rho_a^{new} = \sum_b \rho_b \tilde{W}_{ab} \frac{m_b}{\rho_b} = \sum_b m_b \tilde{W}_{ab} \quad (2.17)$$

where the kernel has been corrected using a zeroth-order correction

$$\tilde{W}_{ab} = \frac{W_{ab}}{\sum_b W_{ab} \frac{m_b}{\rho_b}} \quad (2.18)$$

In cases where the *delta-SPH* method is in use, it may not be sensible to apply the Shepard density filter as well, however it is possible for both methods to be used simultaneously within DualSPHysics. The frequency M_s that the filter is applied is a free parameter that can be set to between 1 and an unbounded upper limit; however it is recommended that the value is set to a value ranging from 30 to 40 time steps.

2.7 TIME STEPPING

DualSPHysics includes a choice of numerical integration schemes, if the momentum (\mathbf{v}_a), density (ρ_a) and position (\mathbf{r}_a), equations are first written in the form

$$\frac{d\mathbf{v}_a}{dt} = \mathbf{F}_a \quad (2.19a)$$

$$\frac{d\rho_a}{dt} = D_a \quad (2.19b)$$

$$\frac{d\mathbf{r}_a}{dt} = \mathbf{v}_a \quad (2.19c)$$

Where \mathbf{v}_a may also include an XSPH correction when these equations are integrated in time using a computationally simple Verlet based scheme or a more numerically stable but computationally intensive two-stage Symplectic method.

2.7.1 Verlet Scheme

This algorithm, which is based on the common Verlet method [Verlet, 1967] is split into two parts and benefits from providing a low computational overhead compared to some other integration techniques, primarily as it does not require multiple (i.e. predictor and corrector) calculations for each step. The predictor step calculates the variables according to

$$\mathbf{v}_a^{n+1} = \mathbf{v}_a^{n-1} + 2\Delta t \mathbf{F}_a^n; \mathbf{r}_a^{n+1} = \mathbf{r}_a^n + \Delta t \mathbf{v}_a^n + 0.5\Delta t^2 \mathbf{F}_a^n; \rho_a^{n+1} = \rho_a^{n-1} + 2\Delta t D_a^n \quad (2.20)$$

where \mathbf{F}_a^n and D_a^n are calculated using Eq. 2.7 (or Eq. 2.12) and Eq. 2.13 (or Eq. 2.14) respectively.

However, once every N_s time steps (where $N_s \approx 50$ is suggested), variables are calculated according to

$$\mathbf{v}_a^{n+1} = \mathbf{v}_a^n + \Delta t \mathbf{F}_a^n; \mathbf{r}_a^{n+1} = \mathbf{r}_a^n + \Delta t \mathbf{v}_a^n + 0.5\Delta t^2 \mathbf{F}_a^n; \rho_a^{n+1} = \rho_a^n + \Delta t D_a^n \quad (2.21)$$

This second algorithm is designed to stop divergence of integrated values through time as the equations are no longer coupled. In cases where the Verlet scheme is used but it is found that numerical stability is an issue, it may be sensible to increase the frequency at which the second part of this scheme is applied, however if it should be necessary to increase this frequency beyond $N_s=10$ then this could indicate that the scheme is not able to capture the dynamics of the case in hand suitably and the Symplectic scheme should be used instead.

2.7.2 Symplectic Scheme

Symplectic integration algorithms are time reversible in the absence of friction or viscous effects [Leimkuhler et al., 1996]. They can also preserve geometric features, such as the energy time-reversal symmetry present in the equations of motion, leading to improved resolution of long term solution behaviour. The scheme used here is an explicit second-order Symplectic scheme with an accuracy in time of $O(\Delta t^2)$ and involves a predictor and corrector stage.

During the predictor stage the values of acceleration and density are estimated at the middle of the time step according to

$$\mathbf{r}_a^{n+\frac{1}{2}} = \mathbf{r}_a^n + \frac{\Delta t}{2} \mathbf{v}_a^n ; \quad \rho_a^{n+\frac{1}{2}} = \rho_a^n + \frac{\Delta t}{2} D_a^n \quad (2.22)$$

where the superscript n denotes the time step and $t = n\Delta t$.

During the corrector stage $d\mathbf{v}_a^{n+\frac{1}{2}}/dt$ is used to calculate the corrected velocity, and therefore position, of the particles at the end of the time step according to

$$\begin{aligned} \mathbf{v}_a^{n+1} &= \mathbf{v}_a^{n+\frac{1}{2}} + \frac{\Delta t}{2} \mathbf{F}_a^{n+\frac{1}{2}} \\ \mathbf{r}_a^{n+1} &= \mathbf{r}_a^{n+\frac{1}{2}} + \frac{\Delta t}{2} \mathbf{v}_a^{n+1} \end{aligned} \quad (2.23)$$

and finally the corrected value of density $d\rho_a^{n+1}/dt = D_a^{n+1}$ is calculated using the updated values of \mathbf{v}_a^{n+1} and \mathbf{r}_a^{n+1} [Monaghan, 2005].

2.7.3 Variable Time Step

With explicit time integration schemes the time step is dependent on the Courant-Friedrich-Levy (CFL) condition, the force terms and the viscous diffusion term. A variable time step Δt is calculated according to [Monaghan and Kos, 1999] using

$$\begin{aligned} \Delta t &= 0.2 \cdot \min(\Delta t_f, \Delta t_{cv}) \\ \Delta t_f &= \min_a \left(\sqrt{h/|f_a|} \right) \\ \Delta t_{cv} &= \min_a \frac{h}{c + \max_b \left| \frac{h\mathbf{v}_{ab} \cdot \mathbf{r}_{ab}}{(\mathbf{r}_{ab}^2 + \eta^2)} \right|} \end{aligned} \quad (2.24)$$

where Δt_f is based on the force per unit mass ($|f_a|$), and Δt_{cv} combines the Courant and the viscous time step controls.

2.8 BOUNDARY CONDITIONS

In DualSPHysics, the boundary is described by a set of particles that are considered as a separate set to the fluid particles. The software currently provides functionality for solid impermeable and periodic open boundaries. Methods to allow boundary particles to be moved according to fixed forcing functions are also present.

2.8.1 Dynamic Boundary Condition

The Dynamic Boundary Condition (DBC) is the default method provided by DualSPHysics [Crespo et al., 2007]. This method sees boundary particles that satisfy the same equations as fluid particles, however they do not move according to the forces exerted on them. Instead, they remain either fixed in position or move according to an imposed/assigned motion function (i.e. moving objects such as gates or wave-makers).

When a fluid particle approaches a boundary and the distance between the boundary particles and the fluid particles becomes smaller than twice the smoothing length (h), the density of the affected boundary particles increases, resulting in a pressure increase. In turn, this results in a repulsive force being exerted on the fluid particle due to the pressure term in the momentum equation.

Stability of this method relies on the length of time step taken being suitably short in order to handle the highest present velocity of any fluid particles currently interacting with boundary particles and it is therefore an important issue when considering how the variable time step is calculated.

2.8.2 Periodic Open Boundary Condition

DualSPHysics provides support for open boundaries in the form of a periodic boundary condition. This is achieved by allowing particles that are near an open lateral boundary to interact with the fluid particles near the complementary open lateral boundary on the other side of the domain.

In effect, the compact support kernel of a particle is clipped by the nearest open boundary that it is nearest to and the remainder of its clipped support applied at the complementary open boundary.

2.8.3 Pre-imposed Boundary Motion

Within DualSPHysics it is possible to define a pre-imposed movement for a set of boundary particles. Various predefined movement functions are available as well as the ability to assign a time-dependant input file containing kinematic details.

These boundary particles behave as DBC described in Section 2.8.1, however rather than being fixed, they move independently of the forces currently acting upon them. This provides the ability to define complex simulation scenarios (i.e. a wave-making paddle) as the boundaries influence the fluid particles appropriately as they move.

2.8.4 Fluid-driven Objects

It is also possible to derive the movement of an object by considering its interaction with fluid particles and using these forces to drive its motion. This can be achieved by summing the force contributions for an entire body. By assuming that the body is rigid, the net force on each boundary particle is computed according to the sum of the contributions of all surrounding fluid particles according to the designated kernel function and smoothing length. Each boundary particle k therefore experiences a force per unit mass given by

$$\mathbf{f}_k = \sum_{a \in WPs} \mathbf{f}_{ka} \quad (2.25)$$

where \mathbf{f}_{ka} is the force per unit mass exerted by the fluid particle a on the boundary particle k , which is given by

$$m_k \mathbf{f}_{ka} = -m_a \mathbf{f}_{ak} \quad (2.26)$$

For the motion of the moving body, the basic equations of rigid body dynamics can then be used

$$M \frac{d\mathbf{V}}{dt} = \sum_{k \in BPs} m_k \mathbf{f}_k \quad (2.27a)$$

$$I \frac{d\boldsymbol{\Omega}}{dt} = \sum_{k \in BPs} m_k (\mathbf{r}_k - \mathbf{R}_0) \times \mathbf{f}_k \quad (2.27b)$$

where M is the mass of the object, I the moment of inertia, \mathbf{V} the velocity, $\mathbf{\Omega}$ the rotational velocity and \mathbf{R}_0 the centre of mass. Equations 2.27a and 2.27b are integrated in time in order to predict the values of \mathbf{V} and $\mathbf{\Omega}$ for the beginning of the next time step. Each boundary particle within the body then has a velocity given by

$$\mathbf{u}_k = \mathbf{V} + \mathbf{\Omega} \times (\mathbf{r}_k - \mathbf{R}_0) \quad (2.28)$$

Finally, the boundary particles within the rigid body are moved by integrating Eq. 2.28 in time. Both [Monaghan et al., 2003] and [Monaghan, 2005] showed that this technique conserves both linear and angular momentum.

3. NEIGHBOUR LIST IMPLEMENTATION

The DualSPHysics code is the result of an optimised implementation that uses the best approaches for CPU and GPU computation of SPH, with simulation accuracy, reliability and numerical robustness given precedence over computational performance where necessary. SPH software frameworks (such as DualSPHysics) can be split into three main steps (Figure 3-1); (i) generation of a neighbour list (NL), (ii) computation of forces between particles and solving momentum and continuity equations (PI) and (iii) integrating in time to update all the physical properties of the particles in the system (SU). Running a simulation therefore means executing these steps in an iterative manner.

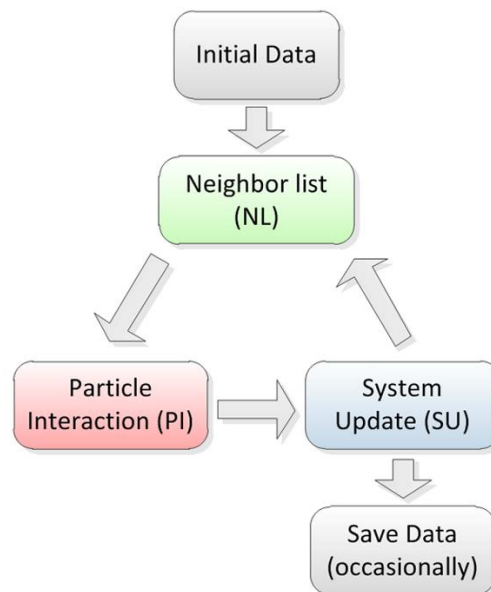


Figure 3-1. Conceptual diagram summarising the implementation of a SPH code.

3.1 STEPS OF THE SPH CODE

During the first step the neighbour list is generated. Particles only interact with neighbouring particles located at a distance less than $2h$. Thus, the domain is divided into cells of size $(2h \times 2h \times 2h)$ to reduce the neighbour search to only the adjacent cells and the cell itself. The Cell-linked list described in [Domínguez et al., 2011a] was implemented in DualSPHysics. Another traditional method to perform a neighbour search is creating an array with all the real neighbours of each particle of the system (named Verlet list), however the main drawback of this approach is its higher memory requirements compared to the Cell-linked list. In the DualSPHysics, two different cell lists were created; the first one with fluid particles and the second one with boundary particles. Therefore, this process can be divided into 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, (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. Note that an actual neighbour list is not created, but also a list of particles reordered according to the cell they belong to, which facilitates the identification of real neighbours during the next step. More details about the neighbour list implementation are provided in Section 3.3.

Secondly, the force computation is performed so that all particle interactions are solved according to the SPH equations. Each particle interacts with all neighbouring particles located at a distance less than $2h$. Only particles inside the same cell and adjacent cells are candidates to be neighbours. Kernel and kernel gradient symmetry, avoids unnecessary repetition of particle interactions leading to a minor improvement in performance. When the force interaction of one particle with a neighbour is calculated, the force of the neighbouring particle on the first one is known since they have the same magnitude but opposite direction. Thus, the number of adjacent cells to search for neighbours can be reduced if the symmetry in the particle interaction is considered, which reduces the computational time. The equations of conservation of momentum and mass (Eq. 2.7 and Eq. 2.13 respectively) are computed for the pair-wise interaction of particles.

Finally, the system is updated. New time step is computed (Eq. 2.24) and the physical quantities are updated in the next step starting from the values of physical variables at the present time step, the interaction forces and the new

time step value (Eq. 2.19b). In addition, particle information (position, velocity and density) are saved on local storage (the hard drive) at defined times.

3.2 TESTCASE

The experiment of Yeh and Petroff at the University of Washington is numerically reproduced using DualSPHysics in order to analyse the performance of the code. This experiment, also described in [Gómez-Gesteira and Dalrymple, 2004] for validation of their 3D SPH model, consists of a dam break problem confined within a rectangular box 160 cm long, 67 cm wide and 40 cm high. The volume of water initially contained behind a thin gate at one end of the box is 40 cm long x 67 cm x 30 cm high. A tall structure, which is 12 cm x 12 cm x 45 cm in size, is placed 50 cm downstream of the gate and 24 cm from the nearest sidewall of the tank. A physical time of 1.5 seconds is calculated. Different instants of the simulation can be observed in Figure 3-2.

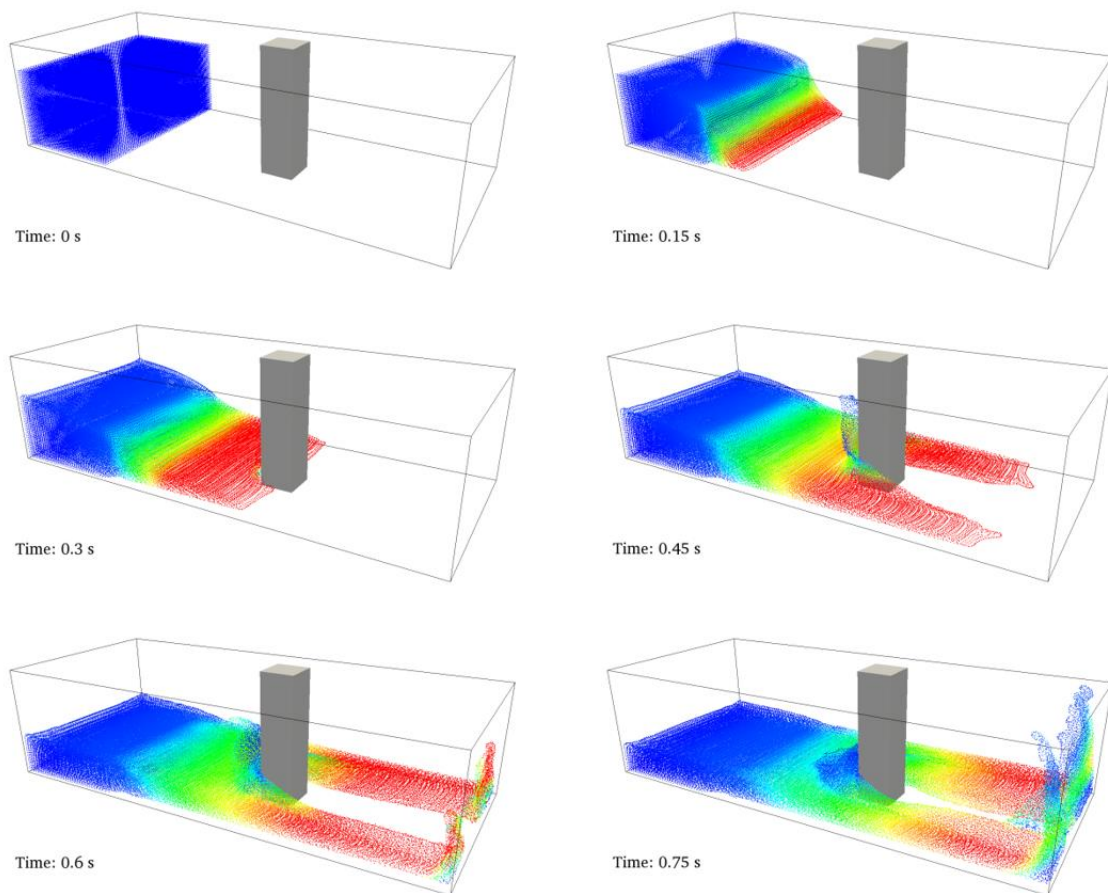


Figure 3-2. Different instants of the dam break evolution using 300,000 particles.

A validation of DualSPHysics using this testcase has already been shown in [Barreiro et al., 2013] where experimental forces exerted onto the structure were in good agreement with the numerical values.

As mentioned above, the SPH method is expensive in terms of computational time. For example, a simulation of this dam break evolution during 1.5s of physical time using 300,000 particles (Figure 3-2) takes more than 15 hours on a single-core machine. The first limitation is the small time step (10^{-6} - 10^{-5} s) imposed by forces and velocities [Monaghan et al., 1999]. Thus, in this case, more than 16,000 steps are needed to complete the 1.5s of physical time. On the other hand, each particle interacts with more than 250 neighbours, which implies a large number of interactions (operations) in comparison with the methods based on a mesh (Eulerian methods) where only a few grid nodes are taken into account. In this case, as it will be shown, the particle interaction takes more than 90% of the total computational time when executed on a single-core CPU. Thus, all the efforts to increase the performance of the code must be focused on reducing the execution time of the particle interaction stage.

3.3 DIFFERENT APPROACHES OF NEIGHBOUR LIST

As mentioned above, the particle interaction step is the most time consuming part of the algorithm in terms of computational time. Before the acceleration of this step, attention must be focused on the neighbour list. The approach used to create the neighbour list needs to be optimised as much as possible to achieve the best performance during the particle interaction.

The determination of which particles are inside the interaction range requires the computation of all pair-wise distances, a procedure with high requirements in terms of computational time for large domains. The efficiency of this procedure, which involves a number of interactions on the order of N^2 (being N the number of particles), is so poor that this brute force evaluation of interactions can only be used in academic exercises as pointed out in [Viccione et al., 2008].

Different approaches coexist in SPH to create a list of neighbours. Here we will focus on just two of them, the cell-linked list and the Verlet list. There are more methods, such as oct-tree methods that are used mostly in astrophysical problems [Stellingwerf and Wingate, 1994] where different variable time scales and long-range interactions like gravity take place.

In the cell-linked list (*CLL* from now on), the computational domain is divided in cells of side $2h$ (cut-off limit), then particles are stored according to the cell they belong to. Thus, an array of particles reordered through the cells is obtained. This array is used during the particle interaction stage, where each particle of interest only looks for its potential neighbours in the adjacent cells (the candidates to be neighbours). When the distance between two particles is less than $2h$, then a real neighbour of the particle of interest is found and forces will be computed. In this case, a list will be associated with each cell. In the simplest version of Verlet list (*VL* from now on), the domain is also divided in cells of size $2h$, and the particles are also allocated in an array where they are grouped according to the cell they belong to. However in this case, a new array is obtained from the previous one, the properly named neighbour list includes all the particles of the adjacent cells at a distance shorter than $2h$ for each particle of the domain. Thus, this so called Verlet list contains the real neighbours of each particle. This array of neighbours is used during the force computation where only the computation of the interaction forces between neighbouring particles is carried out. Percentages in Table 3-1 are referred to the 100% total runtime of the simulation of a dam break, which implies the execution of several time steps. These values are related to a given solution where particle forces are computed once. The creation of the *VL* is more complex since it involves all calculations needed to generate *CLL* and the additional construction of the Verlet list. However, this list can be kept during several time steps considering cells of size slightly higher than $2h$, as it will be shown.

Table 3-1. Comparison of Cell-linked list (*CLL*) and Verlet list (*VL*): percentage of the total runtime of the dam-break simulation using 300,000 particles.

SPH step	Cell-linked list (<i>CLL</i>)		Verlet list (<i>VL</i>)	
Neighbour List	cells division particles in cells	1.7%	cells division particles in cells search of neighbours in adjacent cells neighbour list construction	20.5%
Force Computation	neighbours search interaction forces	96.7%	load neighbour list interaction forces	78%
System Update	solve variables of next step	1.6%	solve variables of next step	1.5%

A cell-linked list (*CLL*) can be calculated by means of the following steps:

- i) The computational domain is divided into cells of side $2h$.
- ii) Particles are stored according to the cell they belong to.

The sketch of this method is shown in Figure 3-3. The possible neighbours (coloured dots) of a particle a are placed in the adjacent cells, but only those particles placed at a distance shorter than $2h$ (dark colour) interact with the particle a .

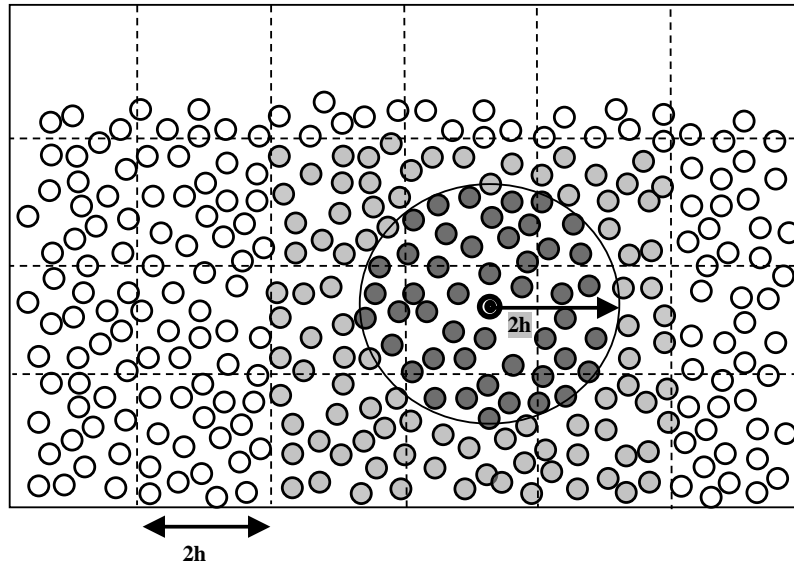


Figure 3-3. Sketch of the Cell-linked list (*CLL*).

On the other hand, the main advantage of a Verlet list is the possibility of keeping the same list during several consecutive time steps. Although, the technique is well known, it has some intrinsic limitations that must be eliminated to obtain a more efficient code. Here, we will first describe the classical method and then the possible improvements. Let us assume that in the classical Verlet list (VL_C from now on) the list is required to remain fixed for the next C time steps:

- a) The computational domain is divided in cells of side $2H=2h+\Delta h$, being $\Delta h=v(2\cdot V_{max}\cdot C\cdot dt)$, where V_{max} is the maximum velocity of any particle of the system, multiplied by 2 since the worst situation appears when two particles with the maximum velocity are moving apart and v is a parameter slightly higher than 1. C is the number of time steps the list is going to be kept. Note that this part of the method is common to *CLL* when $\Delta h=0$.
- b) Search for potential neighbours at the adjacent cells. When the distance between the particle of interest, a , and another particle, b , is less than $2H$

($r_{ab} < 2H$) that particle is added to the list of potential neighbours. Note that the particle is only a candidate to interact with a during the following C time steps but only particles with $r_{ab} < 2h$ will interact during the present time step.

- c) The list of potential neighbours is loaded and kept during the following C time steps. Only those particles with $r_{ab} < 2h$ will interact. Note that particles move in time, in such a way that from the initial set of candidates only a percentage of the total interacts each time step, and the interacting particles can change every time step.

The method presents several drawbacks. On the one hand, the list is not checked every time step and particles can leave or enter the neighbourhood without being detected. The imposed condition on Δh depends on V_{max} and dt , which do not remain constant during the C time steps. V_{max} can vary due to flow acceleration and dt is variable. This fact can give rise to inaccuracies in calculations. This effect can be prevented by using $v=1.2$ in the definition of Δh , although this implies higher memory requirements and will slow down the code since the number of “false” candidates increases. On the other hand, the method is inefficient in terms of computational time. An initial condition is imposed on Δh , but that condition considers the worst situation at the first step from the C steps the list is kept. However, velocity can decrease during the C time steps and the interaction between two particles with the maximum velocity does not necessarily take place. In summary, the list is likely to remain valid for more than C time steps.

The following Verlet list (VL_X from now on) is proposed. In this case Δh is calculated in the same way as in VL_C . However, the number of steps the list is kept (X instead of C) is only tentative, assumed at the first time step, but it can be longer or shorter depending on the calculation. The position of all particles in the domain is also recorded at the first time step. During the following time steps the position of the particles is checked. When the distance travelled by any particle from the first step is longer than $\Delta h/2$ the Verlet list is recalculated and assumed to last for X time steps. Note that the drawbacks mentioned above disappear with this approach. When a particle enters or leaves the neighbourhood of particle a the list is recalculated, even when the number of steps is less than X . Furthermore, the list can be kept even when the number of steps is higher than X if no particle has left or entered the neighbourhood of any particle a . Finally, $v=1.0$ is assumed in Δh calculation because no extra distance is added to $2H$ since the real position of the particles is checked every time step.

The sketch of Verlet List approach is shown in Figure 3-4. The possible neighbours (coloured dots) of the particle are placed in the adjacent cells. Only those particles placed at a distance shorter than $2H$ (dark colour) will potentially interact with particle a and will be included in the list. Note that during the first time step only the particles marked with black dots will interact with particle a .

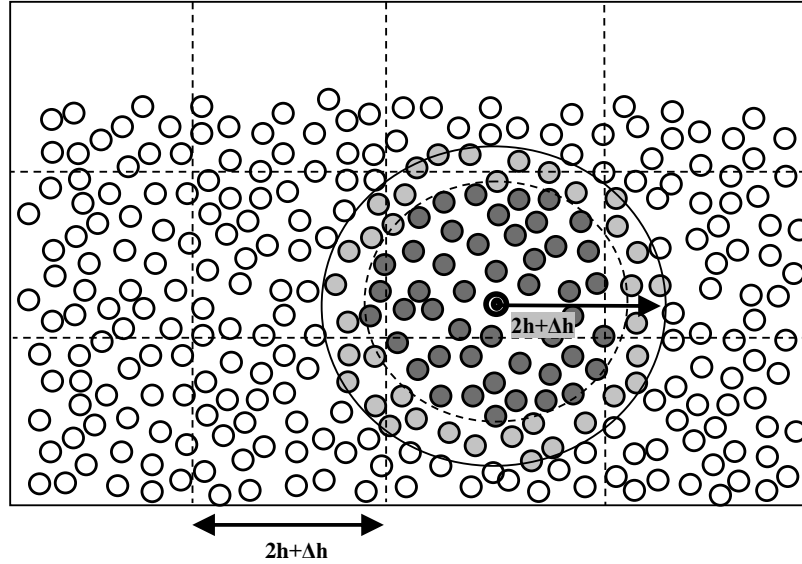


Figure 3-4. Sketch of the Verlet list (VL).

The case shown in Figure 3-2 is used here to compare both neighbour lists. Hence, differences in the computational runtime can be observed in Figure 3-5. In both simulations, particles have been sorted according to the cells. *VL* is slower than *CLL* for any number of particles. In particular, the method is about 13% slower for 150,000 particles. This difference is due to the time needed to create the real neighbour list in *VL*. However, the power of this method has not been properly exploited since the same list can be kept for several time steps, which alleviates the additional burden associated with the creation of the Verlet list.

From the point of view of memory requirements, *VL* is less efficient than *CLL* (Figure 3-6). Thus, for example, the allocated memory is 18 times higher in *VL* when using 150,000 particles. In addition, this ratio increases almost linearly with N .

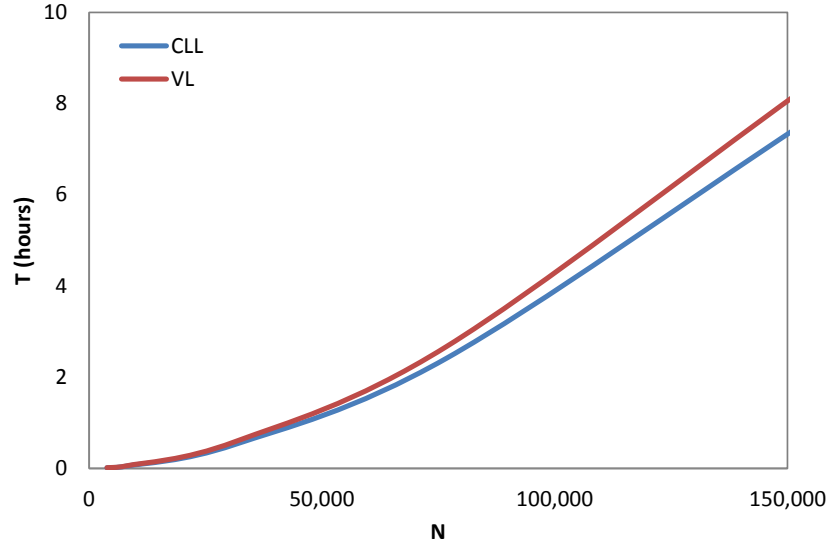


Figure 3-5. Computational runtime of different approaches for neighbour list.

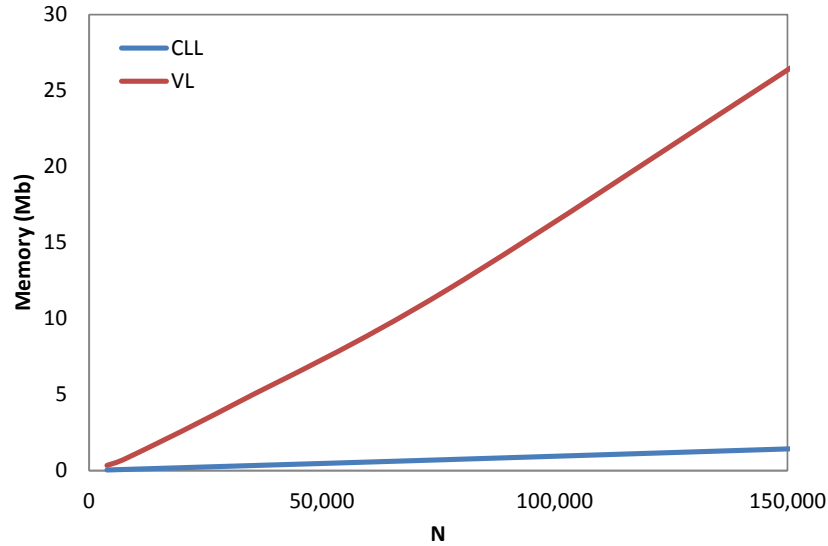


Figure 3-6. Memory requirements of different approaches for neighbour list.

Figure 3-7 shows the different performances of VL_C and VL_X in terms of runtime depending on the number of time steps (n_s) the list is kept (C or X depending on the approach). Both VL_C and VL_X showed to be less efficient than CLL for low and high n_s values. However, there is an intermediate region ($3 \leq n_s \leq 15$), where both methods showed to be faster than CLL . In particular, the most efficient region is obtained for $n_s \sim 7$ time steps, where VL_X is about 4% faster than CLL and VL_C 3% faster. In addition, the method VL_X has shown to be faster than VL_C for any n_s . Obviously, the particular location of the maximum and the interval where the methods based on the Verlet list are more efficient depend on the case under study, although other calculations with different test cases showed a similar behaviour. A similar figure can be obtained for different values of N .

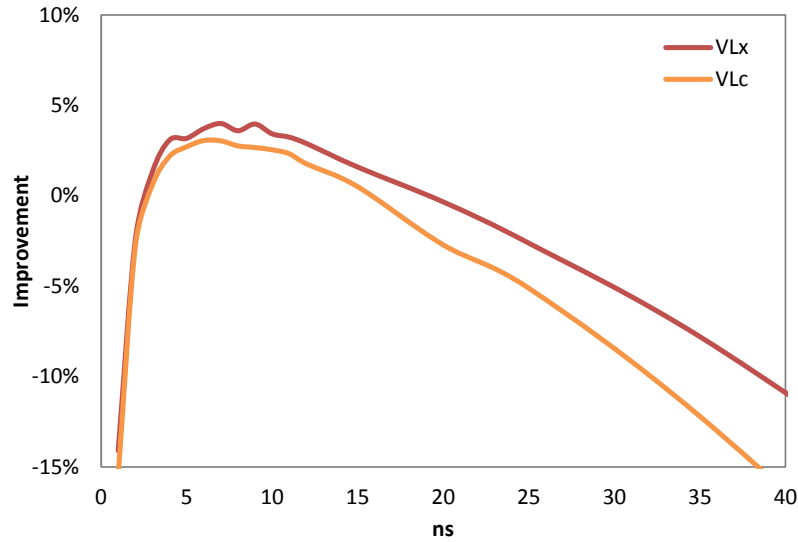


Figure 3-7. Improvement in time using VL_C and VL_X compared to CLL . All cases were calculated with $N=31,239$.

The memory requirements of the different methods are shown in Figure 3-8. Thus, while CLL always requires the same amount of memory, the increase is almost parabolic with n_s in VL_C and VL_X . In the present case, $N=31,239$ particles, the memory allocated for CLL is on the order of 0.25 Mb and it can even be on the order of 25 and 40 Mb for VL_X and VL_C respectively. In the region where both methods are efficient ($n_s \sim 7$) the allocated memory is about 30 times higher than in CLL . In addition, it should be noted that VL_C has higher memory requirements than VL_X for any n_s . This is a direct consequence of the different value of v considered in both approaches (see Δh definition).

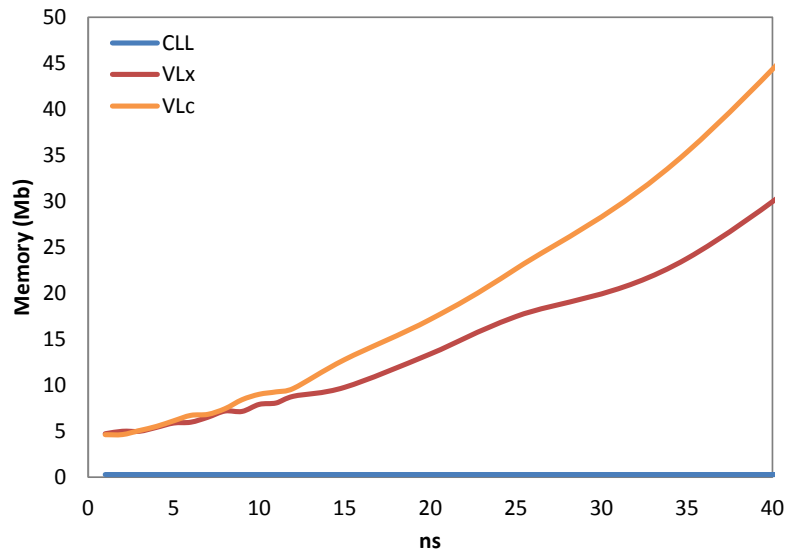


Figure 3-8. Allocated memory in CLL , VL_X and VL_C . All cases were calculated with $N=31,239$.

Figure 3-9 shows the comparison between VL_C and VL_X in terms of runtime improvement compared to CLL . This comparison was carried for $n_s = 7$ time steps which corresponds to the most efficient value for the methods based on a Verlet list as shown in Figure 3-7. For any number of particles, both methods have shown to be faster than CLL , with an improvement that tends to increase with N . In addition, VL_C was observed to be slower than VL_X for any N . The maximum improvement ($\sim 5.7\%$) was obtained for VL_X with 150,000 particles.

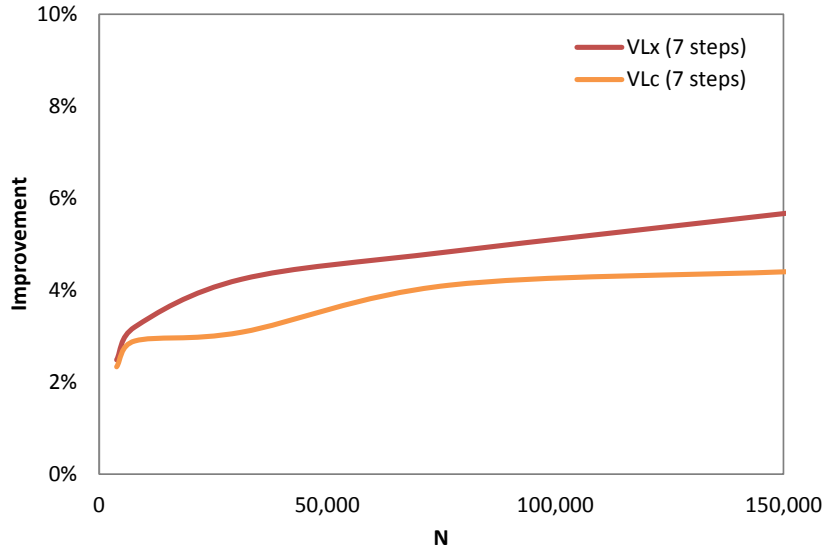


Figure 3-9. Improvement comparison between VL_X and VL_C referred to CLL .

The observed improvement in velocity is moderate, especially when the memory requirements of the Verlet list are on the order of 30 times higher than for CLL in the most efficient region (see Figure 3-7 and Figure 3-8). However, this result can be improved in those cases where the loop over particles should be carried out more than once per time step. This is the case, for example, of different improvements in classical SPH formulation as MLS filters ([Colagrossi and Landrini, 2003], [Dilts, 1999]), kernel and kernel gradient corrections ([Belytschko et al., 1998], [Bonet and Lok, 1999], [Vila, 1999], [Chen and Beraun, 2000]) or Riemann solvers ([Marongiu et al., 2010]). In CLL the potential neighbours placed in adjacent cells are checked several times every time step, while in the methods based on Verlet list the same list is loaded more than once but not recalculated several times every time step.

Figure 3-10 shows the comparison between VL_X in terms of runtime improvement compared to CLL . Two different approaches have been considered in this case. This line coincides with the red solid line shown in Figure 3-9. The red dashed line corresponds to the same model with the kernel gradient

correction described in [Bonet and Lok, 1999]. In the case with kernel gradient correction, the velocity improvement was calculated comparing the runtime using VL_x with the runtime using CLL and the same gradient correction. Comparison was carried out assuming the most favourable case ($n_s=7$, see Figure 3-7). Obviously, the improvement is higher in the corrected case, reaching a percentage higher than 8% for $N=150,000$ particles.

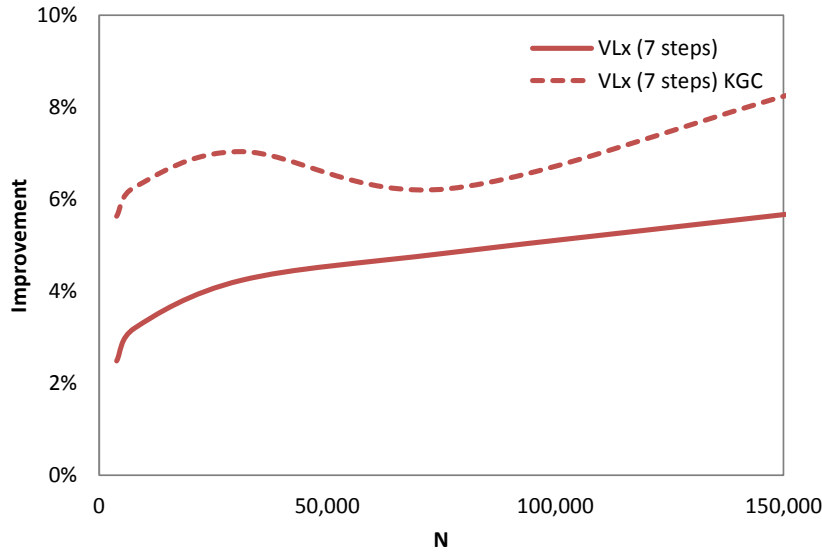


Figure 3-10. Comparison between VL_x with and without kernel gradient correction (KGC). The improvement is referred to CLL .

In general, VL needs much more memory than CLL , which is the main drawback of the method. In terms of runtime, the main advantage of VL is the possibility of keeping the same list during several consecutive time steps. The improved version (VL_x) of the classical Verlet list showed to be dependent on the number of steps, n_s , that the list is kept. For low and high values of n_s the CLL method was faster than VL_x . Only in an intermediate region VL_x was faster than CLL with a maximum improvement close to 6% for $n_s=7$ time steps. This runtime improvement is rather moderate, especially when considering the memory requirements of the method compared to CLL . A better improvement in terms of runtime can be achieved when SPH has to loop over the particles more than once per time step, since the same list is kept in VL_x but the code is forced to a new search when using CLL . An improvement in runtime higher than 8% was obtained when using a SPH formulation with a kernel gradient correction, which implies a double loop every time step. Further improvement is expected when the number of loops per time step increases. To sum up, the choice of the neighbour list approach (CLL or VL_x) depends on the specific simulation under study. CLL is suggested for use when running a serial code since the number of particles is

high and the memory requirements in VL_x are too expensive to be balanced by a runtime improvement on the order of 10%.

DualSPHysics is designed to simulate large number of particles. So that, the Cell-linked list is implemented since it provides the best balance between the performance and the memory usage. Once the neighbour list has been optimised with the most efficient algorithm, the force computation can be now accelerated with the best CPU and GPU strategies, as it will be presented in the following chapters.

4. CPU ACCELERATION

Some features intrinsically linked to the Lagrangian nature of SPH models should be mentioned before going into details about optimization strategies. The physical variables corresponding to each particle (position, velocity, density...) are stored in arrays. During the Neighbour List stage (see Chapter 3), the cell to which each particle belongs is determined. This makes possible to reorder the particles (and the arrays with particle data) following the order of the cells. Thus, if particle data are closer in the memory space, the access pattern is more regular and efficient [Ihmsen et al., 2011]. Another advantage is the ease to identify the particles that belongs to a cell by using a range since the first particle of each cell is known. In this way, the interaction between particles is carried out in terms of the interaction between cells. All the particles inside a cell interact with all the particles located in the same cell and in adjacent cells. Force computations between two particles will be carried out when they are closer than the interaction range ($2h$).

4.1 CPU OPTIMIZATIONS

Some standard and well-known CPU optimizations have been applied to DualSPHysics such as: applying symmetry to particle-particle interaction, splitting the domain into smaller cells, using SIMD instructions and multi-core programming with OpenMP.

4.1.1 Applying symmetry to particle-particle interaction

When the force, f_{ab} , exerted by a particle, a , on a neighbour particle, b , is computed, the force exerted by the neighbouring particle on the first one can be known since it has the same magnitude but opposite direction ($f_{ba} = -f_{ab}$). Note that $\nabla_a W_{ab} = -\nabla_b W_{ba}$ in Eq. 2.7 and Eq. 2.13. Thus, the number of interactions to

be evaluated can be reduced by two, which decreases the computational time. For this purpose, in 3D, each cell only interacts with 13 cells and, partially, with itself (symmetry is also applied for the particles inside the same cell), instead of 27 as shown in Figure 4-1.

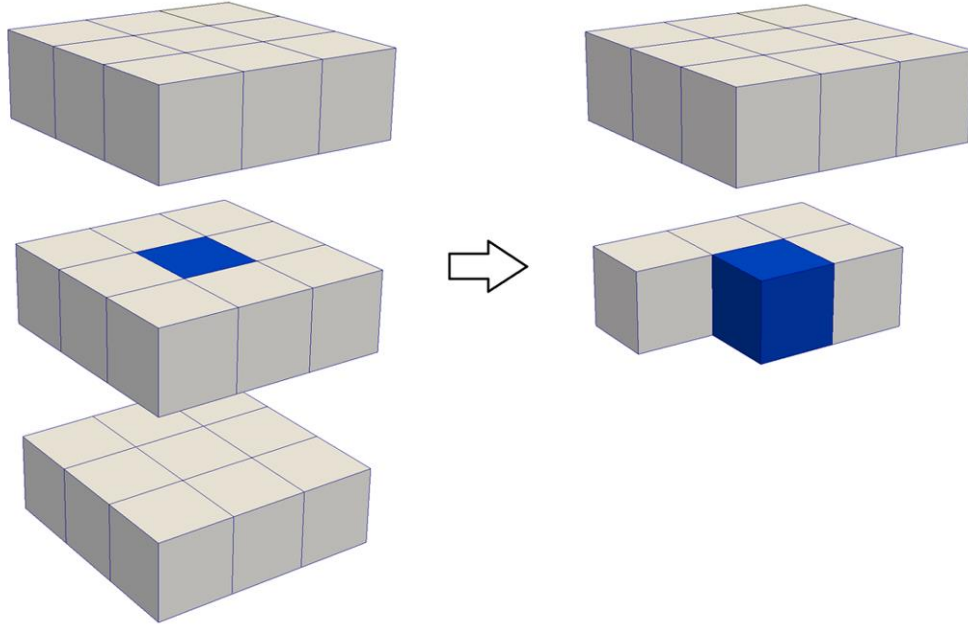


Figure 4-1. Interaction cells in 3D without (left) and with (right) symmetry in particle interactions. Each cell interacts with 14 cells (right) instead of 27 (left).

4.1.2 Splitting the domain into smaller cells

Usually, in particle methods, the domain is split into cells of size $(2h \times 2h \times 2h)$ to reduce the neighbour search to only the adjacent cells. Thus, in 3D and without considering symmetry, a volume of $27(2h)^3$ is searched for every cell to look for potential neighbours. This volume is considerably higher than the volume of the sphere of radius h around the target particle, a , where its real neighbours are placed ($V_{sphere} = (4/3)\pi(2h)^3 \sim 4.2 \cdot (2h)^3$). This can be generalized to any division of the computational domain into cells of side $2h/n$. Thus the ratio between the searched volume and the sphere volume becomes $(2 + (1/n))^3 / ((4/3)\pi)$, which tends asymptotically to $6/\pi$ when n goes to infinity. Thus, a suitable technique to diminish the number of *false* neighbours would be to reduce the volume of the cell. Unfortunately, each cell requires the storage of information to identify its beginning, end and number of particles, which prevent the use of large n values. A balance between decreasing the searching volume and limiting memory requirements should be found. According to our experience, n values on the order of 2 are recommended. In fact, the kernel support of the chosen kernel (Eq.

2.4 and Eq. 2.5) is $2h$ so the smaller cells will be of the size of $2h/2$ (h) in this case. Figure 4-2 shows the comparison between dividing the domain into cells of side $2h$ ($n=1$) and side $2h/2$ ($n=2$).

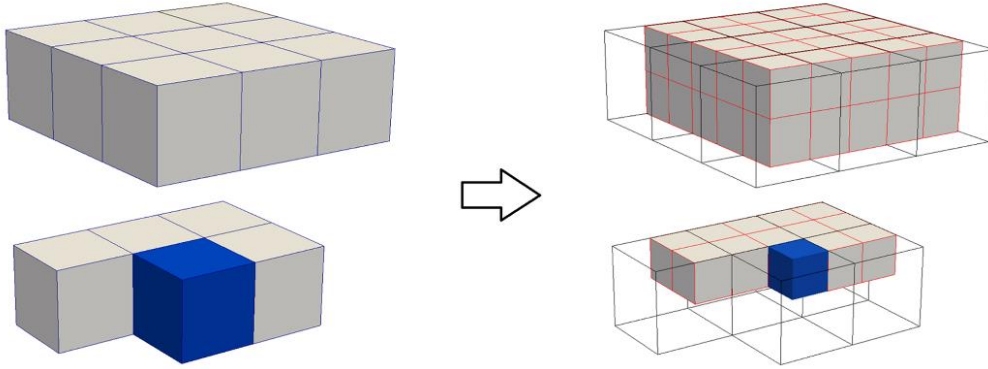


Figure 4-2. Sketch of 3D interaction with close cells using symmetry. The volume searched using cells of side $2h$ (left panels) is bigger than using cells of side h (right panels).

4.1.3 Using SSE instructions

The current CPUs have special instruction sets (SSE, SSE2, SSE3...) of SIMD type (Single Instruction, Multiple Data) that allow performing operations on data sets. A basic operation (addition, subtraction, multiplication, division, comparison...) of four real numbers (in single precision) can be executed simultaneously. Another advantage is the straightforward translation to machine-code providing a higher performance. However this optimization also presents two disadvantages: first, coding is quite cumbersome and, second, the technique can only be applied to specific cases where the calculations are performed in packs of 4 values. Although modern compilers implement the automatic use of these SIMD instructions, [Dickson et al., 2011] emphasize the need of making an explicit vectorisation of the computations to obtain the best performance on the CPU. Therefore, these instructions are applied to the interaction between particles that were previously grouped into packs of 4 to compute forces simultaneously. An example of a simplified pseudocode can be seen in Figure 4-3.

```

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]);

```

Figure 4-3. Sketch Pseudocode in C++ showing the force computation between the particles of two cells without vectorial instructions (up) and grouping in blocks of 4 pair-wise of interaction using SSE instructions (down).

4.2 OPENMP IMPLEMENTATION

The main CPU optimization described in this work is the implementation of a multi-core programming with OpenMP (as described in Section 1.3). Current CPUs have several cores or processing units, so it is essential to distribute the computation load among them to maximize the CPU performance and to accelerate the SPH code. There are two main options to implement a parallel code in CPU, namely MPI and OpenMP. MPI (also described in Section 1.3) is particularly suitable to distribute memory systems where each processing unit has only access to a portion of the system memory and the different processes need to exchange data by passing messages. However the architecture used in this work uses shared memory system, where each process can directly access to all memory without the extra cost of the message passing in MPI as shown in SPH in [Goozee and Jacobs, 2003]. As mentioned, OpenMP is portable and flexible whose implementation does not involve major changes in the code. All the cores of the CPU share the same memory space so data transfer is not required among threads. Therefore, OpenMP is used in DualSPHysics when executing the code in a multi-core CPU machine.

Several parts of the SPH code can be parallelised, which is especially important for force calculation that is the most expensive part of the code. The minimum execution unit of each thread is the cell, so that all particles of the same cell are

processed sequentially. Neighbouring particles are searched in the surrounding cells and the particle interaction is computed. However, it is not straightforward to apply symmetry to particle-particle interaction when several execution threads of a CPU are used in parallel since the concurrent access to the same memory positions for read-write particle forces can give rise to unexpected results due to race conditions. In addition, special attention should be paid to the load balancing to distribute equally the work among threads. Therefore, three different approaches were proposed to avoid concurrent accesses and obtain load balancing:

- a) **Asymmetric:** Concurrent access occurs in force computation when applying symmetry, since the thread that computes the summation of the forces on a given particle also computes the forces on the particles placed in the neighbourhood of the first one. Nevertheless, these neighbouring particles may be simultaneously processed by another thread. To avoid this conflict, symmetry is not applied in a first approach. The load balancing is achieved by using the dynamic scheduler of OpenMP. Cells can be assigned (usually in blocks of 10) to the threads as they run out of workload. Figure 4-4 shows an example of dynamic distribution of cells (in blocks of 4) among 3 execution threads according to the execution time of each cell, which depends on the number of neighbouring particles. The main advantage is the ease of implementation, being the main drawback the loss of symmetry.

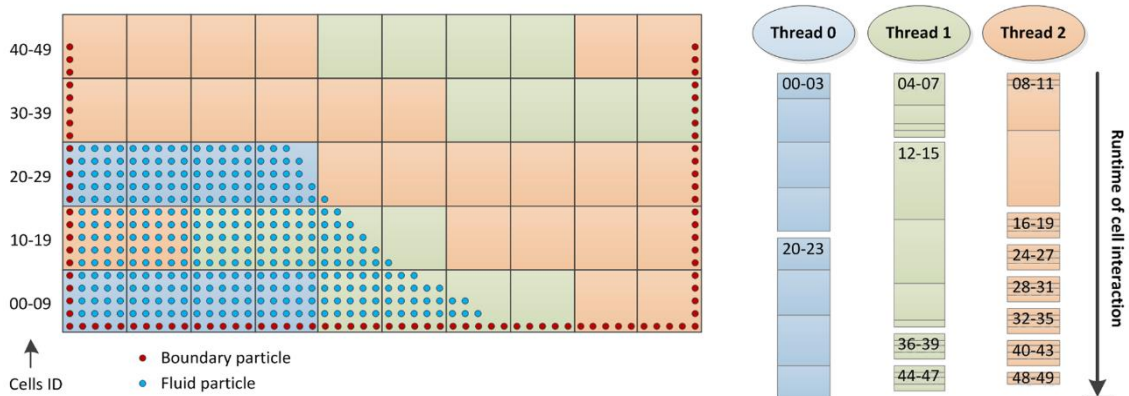


Figure 4-4. Example of dynamic distribution of cells (in blocks of 4) among 3 execution threads according to the execution time of each cell.

- b) **Symmetric:** In this approach, the dynamic scheduler of OpenMP is also employed distributing cells in blocks of 10 among different threads. The difference with the previous case lies in the use of the symmetry in the computation of the particle-particle interaction. Now the concurrent memory access is avoided since each thread has its own memory space to allocate

variables where the forces on each particle are accumulated. Thus, the final value of the interaction force for each particle is obtained by combining the results once all threads have finished. This final value is also computed by using multiple threads. The advantage of this approach is the use of the symmetry in all the interactions and the easy implementation of the load dynamic balancing. The main drawback is the increase in memory requirements, which depends on the number of threads. Note that memory duplication for each thread is efficient when using a system with a small number of threads (as the hardware available in this work), but it does not scale on a system with much wider CPUs which can execute much more threads. For example, memory requirement increases by a factor of 2 when passing from 1 to 8 threads in the testcase.

- c) **Slices:** The domain is split into slices, so that the number of slices is the number of available execution threads. Symmetry is applied to the interactions among cells that belong to the same slice, but not to the interactions with cells from other slices. Thus, symmetry is used in most of the interactions (depending on the width of the slices). The thickness of the slices is adjusted to distribute the runtime of the particle interactions within each slice (dynamic load balancing). The division is periodically updated to keep the slices as balanced as possible. This thickness is adjusted according to the computation time required for each slice during the last time steps, which allows a more correct dynamic load balancing. The main drawbacks are the higher complexity of the code and the higher runtime associated to the dynamic load balancing.

4.3 RESULTS

The DualSPHysics code will be used to run the dam-break simulation described in Section 3.2. The system used for the CPU performance testing is the following:

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

Figure 4-5 shows the achieved speedup on CPU for different number of particles (N) when applying the three first optimization strategies explained in Section 4.1;

symmetry in particle interaction, division of the domain into cells of size $2h/2$ and use of SSE instructions. The blue line in Figure 4-5 shows the speedup obtained using symmetry and the red line includes the speedup when using SSE instructions and symmetry, the value in parentheses is the cell size. Using 300,000 particles, a maximum speedup of 2.3x is obtained using these CPU optimizations when compared to the version of the code without optimizations.

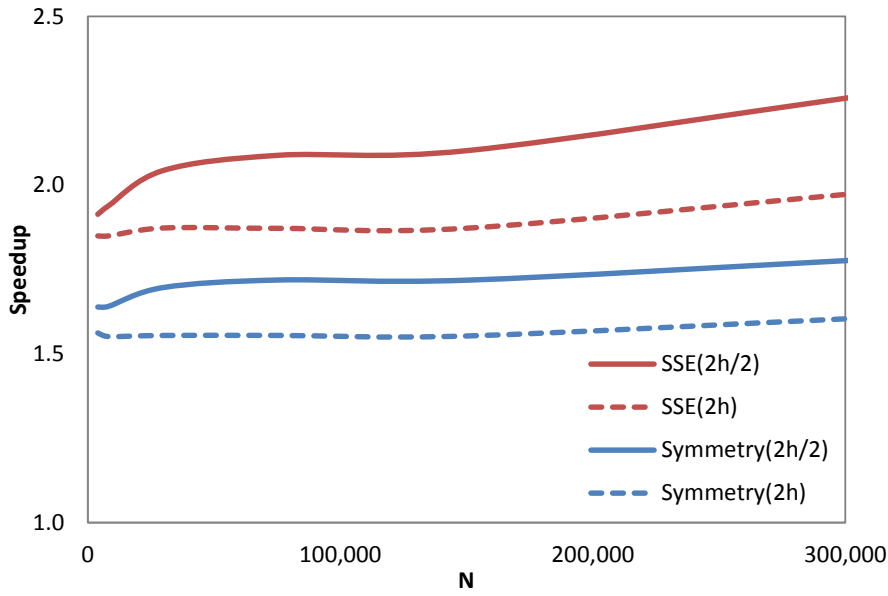


Figure 4-5. Speedup achieved on CPU for different number of particles (N) when applying symmetry, the use of SSE instructions. Two different cell sizes ($2h$ and $2h/2$) were considered.

The speedup obtained with the multi-core implementation on CPU of the SPH code for different number of particles is observed in Figure 4-6. In the figure, the performance of the different OpenMP implementations (using 8 threads) is compared with the most efficient single-core version (that includes symmetry, SIMD instructions and cell size equal to $2h/2$). The most (less) efficient implementation is *Symmetric* (*Asymmetric*). A speedup of 4.5x is obtained with *Symmetric* when using 8 threads. The approaches that divide the domain into slices (*Slices*) offer a higher performance when increasing the number of particles since the number of cells also increases, allowing a better distribution of the workload among the 8 execution threads. Using *Slices*, the efficiency does not depend on the direction of fluid movement. Similar performance is achieved when creating the slices in X or Y-direction, since the workload is distributed equally among the slices.

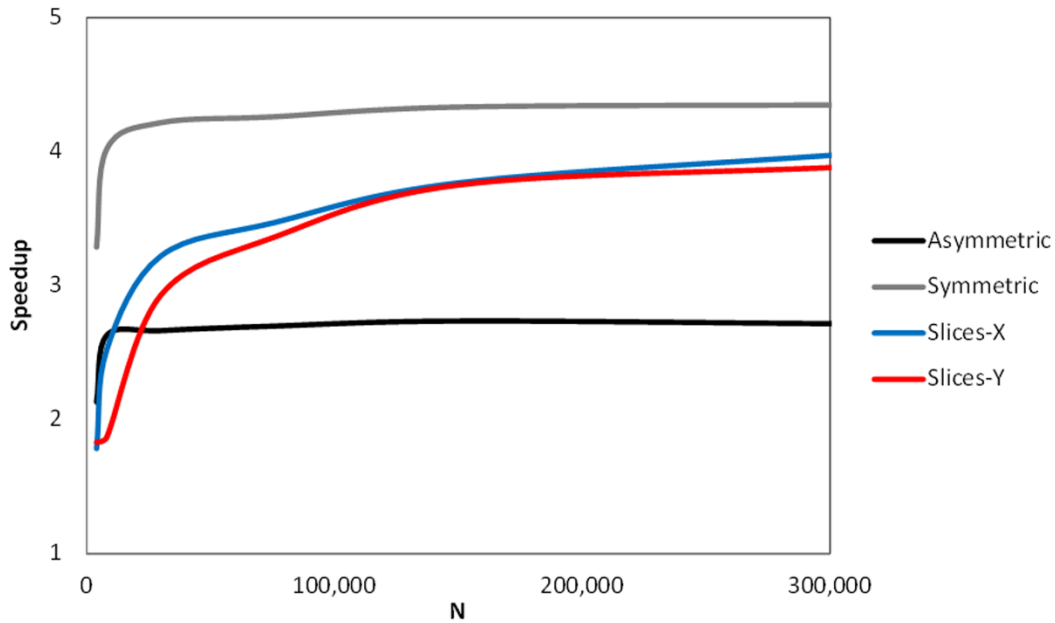


Figure 4-6. Speedup achieved on CPU for different number of particles (N) with different OpenMP implementations (using 8 logical threads) in comparison with the most efficient single-core version that includes all the previous optimizations.

Table 4-1 shows the computational times and speedups on CPU using the most efficient version of OpenMP (*Symmetric*) with 4 and 8 threads compared to the single-core CPU version. Note that the evaluation of the speedup is not expected to be linear with the number of threads since the available CPU hardware is the Intel® Core™ i7 with 4 physical cores and 8 logical cores with Hyper-threading and Table 4-1 shows results using logical cores instead of physical ones. Therefore, the parallel CPU version with 8 threads is 4.6 times faster than single-core version and the speedup is 3.9x using 4 threads.

Table 4-1. Speedup achieved on CPU simulating 300,000 particles when using 4 and 8 threads compared to the single CPU version.

Version	Total simulation time	Number of Steps	Computed steps per second	Speedup vs. CPU Single-core
CPU Single-core	24,520 s	16,282	0.66	1.0x
CPU 4 Threads	6,375 s	16,275	2.55	3.9x
CPU 8 Threads	5,414 s	16,284	3.01	4.6x

5. GPU ACCELERATION

Nowadays, GPUs can be used for general purpose applications, achieving important speedups in comparison with classical CPUs. However, an efficient and full use of the capabilities of the GPUs is not straightforward and it is necessary to know and to take into account the details of the GPU architecture and the CUDA programming model described in [CUDA Programing Guide]. On the other hand, SPH method is not very suitable to run on GPU because it presents several problems like divergence and irregular memory access. Hence, this kind of problems must be minimised to obtain good speedups.

5.1 CUDA PROGRAMMING MODEL

The GPU card is a specialized hardware to execute in parallel the same instruction on many data elements (SIMD parallelism). Therefore, it is especially well-suited to address problems with high arithmetic intensity and low flow control. CUDA (Compute Unified Device Architecture) is a programming environment for GPU computing. It includes a C/C++ language extension, a compiler called nvcc, libraries and tools to develop programs for Nvidia GPUs. A more complete description of CUDA programming model can be found in [CUDA Programing Guide], so only some basic concepts are introduced here.

A program implemented with CUDA contains a part that is executed on CPU (host) and another part executed on GPU (device). The code executed on GPU consists of a set of functions called kernels. The CPU memory and GPU memory are independent memory spaces, therefore an explicit memory transfer from CPU memory to GPU memory has to be carried out before running a GPU kernel. The same process has to be performed in the opposite direction to recover the results of a kernel execution. These data transfers can reduce the performance and should be minimised.

A kernel has a set of instructions which are executed with an element or data. Each element in CUDA is processed by an independent thread. The threads are grouped into blocks of threads and each block is executed in a SM (Streaming Multiprocessor). The maximum number of threads per block is 512 or 1024 depending on GPU model. The blocks are grouped into grids (see Figure 5-1) whose maximum size is 65535 x 65535 and higher in the most modern GPUs. In this way, a grid of 3907 blocks with 256 threads per block would be necessary to process 1 million elements. The size of the block and the grid can be defined using one or several dimensions to better suit the nature of the problem. During the kernel execution, the number in the block, block number in the grid, size of block and size of grid are known by each thread.

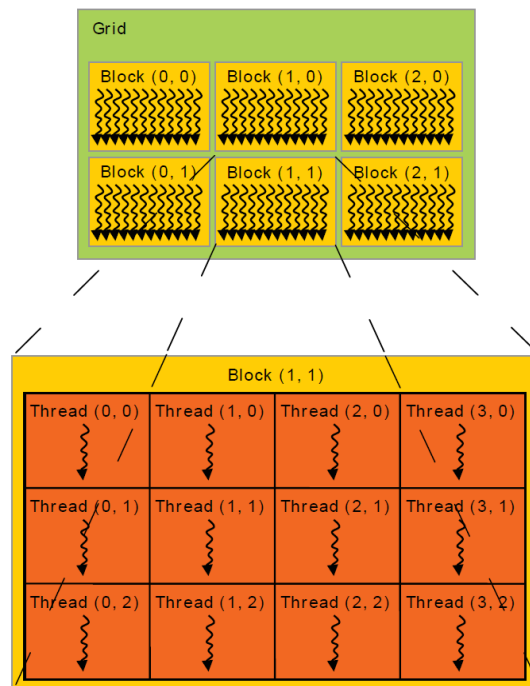


Figure 5-1. Grid of thread blocks in CUDA (source: CUDA Programming Guide v6.5)

Each thread has a private *local memory* and it cannot access to the *local memory* of other threads. The threads of the same block have a *shared memory* and they can use it to share or exchange data. All threads can access to the same *global memory*. The memory hierarchy of GPU is showed in the Figure 5-2. The speed of access is different for each kind of memory. *Global memory* is the largest (hundreds of megabytes or gigabytes), but also it is the slowest (two orders of magnitude slower). The *shared memory* can be as fast as the local memory

(registers) but its maximum size is 64 KB. The speed of access of the *shared memory* depends on the access pattern (regular or irregular) and GPU model. Two additional read-only memories are the *constant memory* and the *texture memory*. The first one is used to store constant values and the second one offers different addressing modes. Both memory spaces are accessible by all threads and a cache is used to improve its access time. The achieved performance depends greatly on how this memory hierarchy is used.

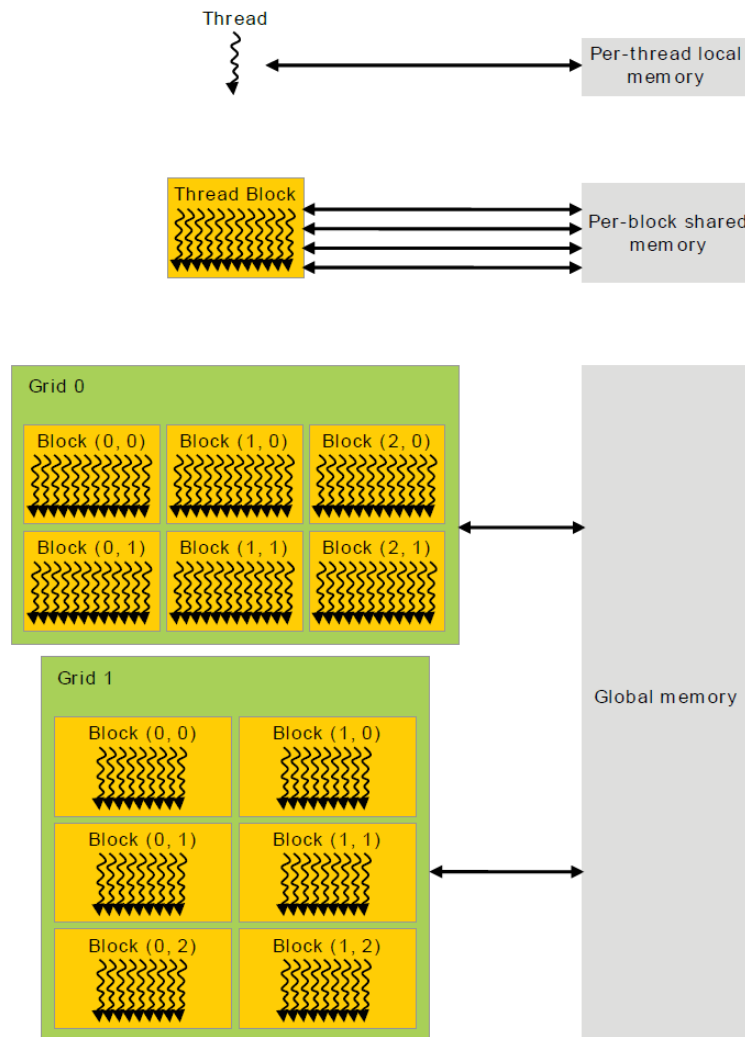


Figure 5-2. Memory hierarchy (source: CUDA Programming Guide v6.5)

5.2 CUDA IMPLEMENTATION

The work presented in [Crespo et al., 2009] introduced the framework to implement SPH codes using the best techniques and performance optimizations on GPU. That work focused on identifying suitable algorithms for efficient

parallelization since a proper and full use of all the capabilities of the GPU architecture is not easy. As an initial step, the implementation focused on solving the particle interactions on a GPU using CUDA and the next step was the implementation of the neighbour list and the time integration in order to develop an entire GPU-SPH model. Different neighbour lists were analysed in Section 3.3. Apart from a non-negligible improvement in the performance of the model, the work also showed that computing particle interactions is the most expensive SPH procedure in terms of computational runtime. This influences the development of a GPU code.

In a first approach (left panel of Figure 5-3), it is possible to keep the other two steps (neighbour list and system update) on the CPU. However, this is less efficient since particle data and neighbour list information must be transferred between both processing units each time step, which consumes hundreds of clock cycles. The most efficient option is keeping all data in the memory of the GPU where all processes are parallelised (right panel of Figure 5-3). Only output data requires transfer from GPU to CPU. This process is rarely carried out (one out of one hundred/thousand time steps) and only represents a low percentage of the total runtime.

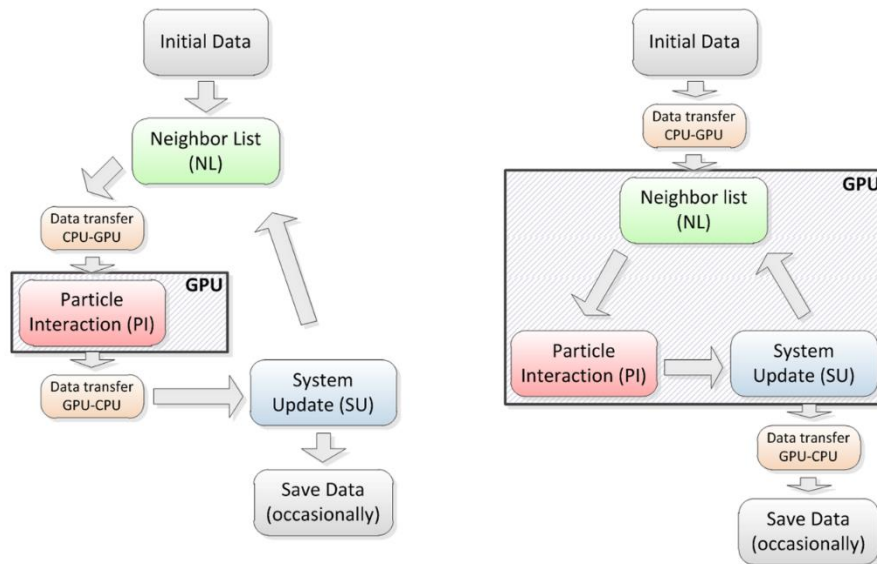


Figure 5-3. Conceptual diagram of the partial (left) and full (right) GPU implementation of the SPH code.

A preliminary version of the DualSPHysics code with a total GPU implementation was presented in [Crespo et al., 2010]. Initially, data is allocated on CPU, so there is a single memory transfer (from CPU to GPU). In all subsequent calculations, the three main steps are then performed on the GPU

device. All the sequential tasks and operations that involve a loop over all particles are performed using the parallel architecture of the GPU cores. To save (or output) data, a new memory transfer is needed (from GPU to CPU). If saving data is not required all particle information remains on the GPU memory and is only updated each time step.

The neighbour list creation follows the procedure used on a CPU, but with several differences. Reordering the particles according to the cells they belong is computed using the optimised *radixsort* algorithm provided by CUDA [Satish et al., 2009]. Figure 5-4 shows a simplified schematic diagram of the method used to generate an array of particle labels ordered according to cells and an array with the position index of the first particle in each cell. Four separate arrays are used: *Id*, *Cell*, *IdSort* and *CellBegin* with a superscript * denoting sorted arrays. The array *Id* (array of particle labels) is the starting point with particles randomly located in the domain, where the order of this array corresponds to the list of particles inherited from the previous timestep. The neighbour list is created according to the following steps:

- i) Particles are stored according to the cells, so the array *IdSort* is created.
- ii) The array *Cell* is also created where an entry gives the cell to which the particle of the same index in *Id* belongs, e.g. $Id(2) = \text{particle 3}$ which is located in Cell 6 \rightarrow hence $Cell(2) = 6$. *Cell* labels are depicted in green colour in Figure 5-4.
- iii) Using the *radixsort* algorithm from Nvidia [Satish et al., 2009], array *Cell* is reordered following the order of the six cells and *Cell*^{*} (reordered *Cell*) is used to reorder *IdSort* according to the cells the particles belong.
- iv) Once *IdSort*^{*} is generated, all the arrays with particle information (*Id*, *Position*, *Velocity*, *Density*...) are ordered giving rise to the new arrays (*Id_new*, *Pos_new*, *Vel_new*, *Dens_new*...) considering that $Id_new[i] = Id[IdSort^*[i]]$. For example, $Id_new[2] = Id[IdSort^*[2]] = Id[7] = 4$, in Figure 5-4 a blue circle marks the particle 4 and a red circle marks the 7th position.
- v) Finally, *CellBegin* is created with the indexes (position in data arrays) of the first particle of each cell. Indexes have been written in red colour in Figure 5-4. For example the first particle of the cell number 2 is the particle 7, whose position index is 3 in all particle property arrays, so the second value of *CellBegin*, which corresponds to cell number 2, will be 3. In this

way, the amount of particles in the cell k will be $CellBegin[k+1]-CellBegin[k]$.

In the latest version of the GPU code, $CellBegin$ has been replaced by $CellBeginEnd$, which not only includes the information of the first particle of the cell, but also the last particle of that cell. This present an advantage when this array is loaded in the GPU kernels.

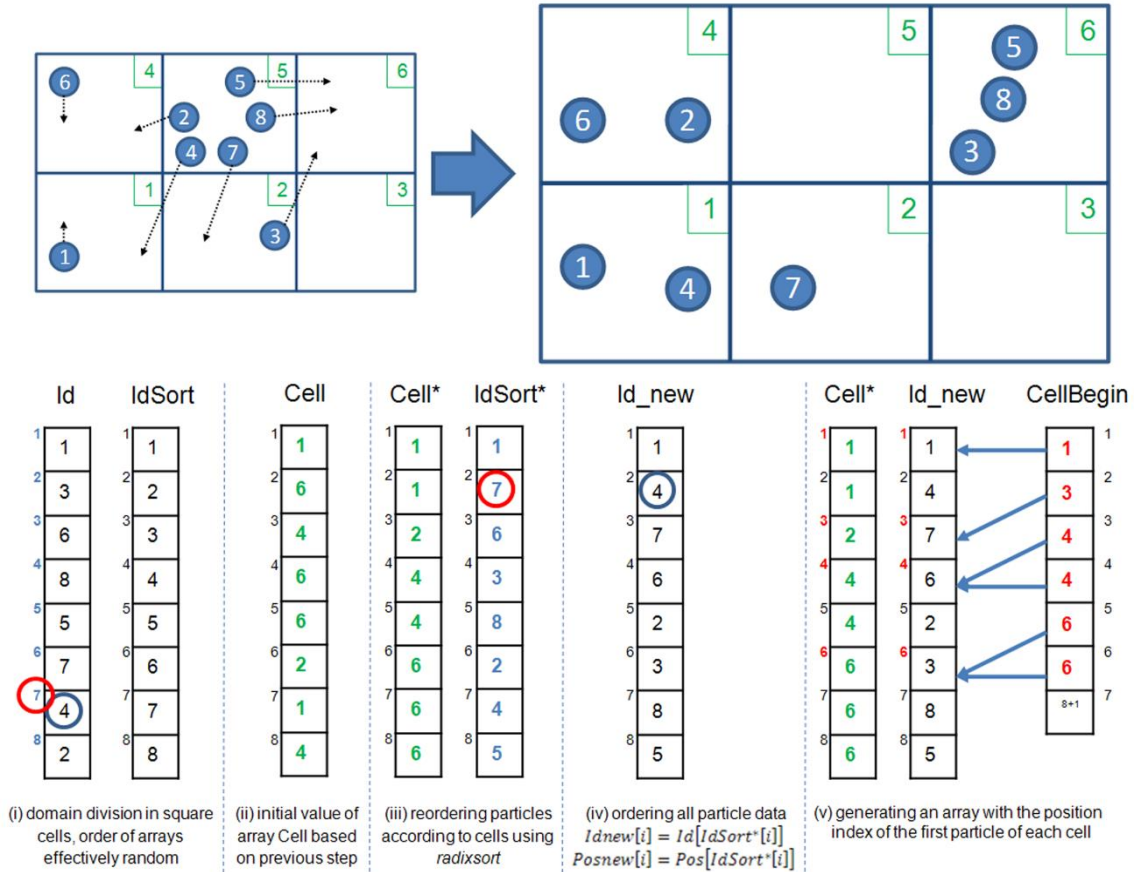


Figure 5-4. Example of the Neighbour list procedure.

The system update associated with time integration can be parallelised easily on a GPU. Example pseudocode is shown in Figure 5-5 where similarities between the CPU and GPU versions are clearly evident and demonstrates the advantages of a using C++ and CUDA when developing code. The new time step is computed according to Eq. 2.24 where the maximum and minimum values of different variables (force, velocity and sound speed) are calculated. This calculation is optimised using the *reduction* algorithm (also provided by CUDA). Reduction algorithm allows obtaining the maximum or minimum values of a huge data set taking advantage of the parallel programming in GPUs.

System update on C++	System update on CUDA
<pre> for(unsigned p=Npb;p<NpOk;p++){ Pos[p].x+=Vel[p].x*dt+Ace[p].x*dtsq_05; Pos[p].y+=Vel[p].y*dt+Ace[p].y*dtsq_05; Pos[p].z+=Vel[p].z*dt+Ace[p].z*dtsq_05; VelNew[p].x=VelM1[p].x+Ace[p].x*twodt; VelNew[p].y=VelM1[p].y+Ace[p].y*twodt; VelNew[p].z=VelM1[p].z+Ace[p].z*twodt; } </pre>	<pre> template <...> __global__ void KerCsComputeStepVerlet(...) { unsigned p=blockIdx.y*gridDim.x*blockDim.x+ +blockIdx.x*blockDim.x+threadIdx.x; if (p<n){ float3 race=ace[p]; rpos.x=rvel.x*dt + race.x*dt205; rpos.y=rvel.y*dt + race.y*dt205; rpos.z=rvel.z*dt + race.z*dt205; pos[p]=rpos; float3 rvelnew,rvelv=velv[p]; rvelnew.x=rvelv.x+race.x*dtdt; rvelnew.y=rvelv.y+race.y*dtdt; rvelnew.z=rvelv.z+race.z*dtdt; velnew[p]=rvelnew; } } </pre>

Figure 5-5. Pseudocode of the System update procedure implemented on CPU and GPU.

As mentioned above, the particle interactions of the force computation are a key process that must be implemented in parallel in order to improve the performance of the model. The use of the shared memory of the GPU was analysed to reduce the access to the global memory of the GPU. However, when the SPH code is implemented entirely on the GPU, this technique is not viable. For example, when the number of particles is large, the size of shared memory is not enough to allocate the properties of all the particles belonging to the same cell. Particle interactions can be implemented on the GPU for only one particle using one execution thread to compute the force resulting from the interaction with all its neighbours. This technique presents several limitations mainly due to the Lagrangian nature of the method. On the one hand, the workload of threads inside one block is not balanced since particles can have different numbers of neighbours. On the other hand, code divergence can appear since when the possible neighbours of a particle are evaluated, some of them are definite neighbours (interparticle distance less than $2h$) and the force computation is performed while other particles are not neighbours (at a distance higher than $2h$) and no computation is performed. Note that according to the link list described in Section 3.3, the potential neighbours are all particles located in adjacent cells. Nevertheless, only those particles at distances less than $2h$ from the target particle are real neighbours.

An important difference here from the CPU part of the DualSPHysics code is that the symmetry of the particle interaction cannot be applied efficiently on a GPU implementation since each thread is responsible for the interaction between a target particle and its neighbours, so that each thread must be the only one that computes the forces exerted on that particle. 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. If a second thread tried to modify those

forces, as could occur when considering particle kernel symmetry, it would generate erroneous results when both threads accessed simultaneously the same variable (race conditions). This effect can be removed by synchronising the threads but it would dramatically reduce the performance of the model. An example of the similarity of the C++ and CUDA codes for this illustrative point is shown in Figure 5-6.

Particle interaction on C++	Particle interaction on CUDA
<pre> for(int i=ibegin;i<=iend;i++){ for(int j=jbegin;j<=jend;j++){ float prs=Press[i]/(Rhop[i]*Rhop[i])+ +Press[j]/(Rhop[j]*Rhop[j]); Ace[i].x+=-massj*frx*(prs+pi_visc); Ace[i].y+=-massj*fry*(prs+pi_visc); Ace[i].z+=-massj*frz*(prs+pi_visc); Ace[j].x+=massi*frx*(prs+pi_visc); Ace[j].y+=massi*fry*(prs+pi_visc); Ace[j].z+=massi*frz*(prs+pi_visc); } } </pre>	<pre> template<...> __device__ void KerCsInteractionBox(...) { for(int j=jbegin;j<jend;j++) if (i!=j){ float prs=devspl.x+prrhop[j]; float p_vpm=-prs*massp2; Acei.x+=-massj*frx*(prs+pi_visc); Acei.y+=-massj*fry*(prs+pi_visc); Acei.z+=-massj*frz*(prs+pi_visc); } Ace[i]=Acei; } </pre>

Figure 5-6. Pseudocode of the Particle interaction procedure implemented on CPU and GPU.

The main difference between the full GPU implementation presented here and the works of [Kolb and Cuntz, 2005] and [Harada et al., 2007] is that they implemented a classical SPH approach on GPU before the appearance of CUDA in 2007 using shader programs written in C for Graphics. In this work, the full GPU implementation is performed using the parallel programming CUDA as described in Section 5.1. CUDA is more independent of the particular hardware. This allows the code to be run on new incoming GPU cards more efficiently. On the other hand, CUDA makes easy the maintenance and the updating of the code when including more complex algorithms and new SPH formulations. The codes developed by [Anderson et al., 2008] for MD and by [Herault et al., 2010] for SPH, also was developed entirely on the GPU but implementing a different approach for neighbor list, giving rise to different efficiencies in terms of performance and memory requirements. They implemented the Verlet list, so the number of particles that can be simulated in the memory space of one GPU card is much smaller than the number of particles presented here.

This implementation presents different problems to be solved:

- a) **Code divergence:** GPU threads are grouped into sets of 32 named *warps* in 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 sequentially, giving rise to a significant

loss of efficiency. This divergence problem appears during particle interaction since each thread has to evaluate which potential neighbors are real neighbors before computing the force.

- b) **No coalescent memory accesses:** The global memory of the GPU is accessed in blocks of 32, 64 or 128 bytes, so the number of accesses to satisfy a warp depends on how grouped data are. A regular memory access is not possible in particle interaction 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.
- c) **No balanced workload:** Warps are executed in *blocks* in the CUDA terminology. When a block is going to be executed, some resources are assigned and they will not be available for other blocks till the end of the execution. So, since each thread may have a different number of neighbors, a thread may need to perform more interactions than the rest. Thus, the warp can be under execution while the rest of threads of the same warp, or even of the block, can have finished. Thus, the performance is reduced due to the inefficient use of the GPU resources.

5.3 GPU OPTIMIZATIONS

Several optimizations have been developed to avoid or minimize the problems previously described. First of all, *maximizing the occupancy of GPU* and *reducing global memory accesses* are some of the well-known basic optimizations described in the CUDA manuals which must be always considered when porting a code to GPU. Then, more GPU optimizations intrinsic to the SPH method such as *simplifying the neighbor search*, *adding a more specific CUDA kernel of interaction* and *the division of the domain into smaller cells* will be described.

5.3.1 Maximizing the occupancy of GPU

Occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU or Streaming Multiprocessor (SM). Since the access to the GPU global memory is 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 (see Table 5-1) and the number of threads per block. The first option could be reducing the number of registers per thread, however this implies the increase of memory accesses and the number of computations in the interaction kernel. Another option is adjusting the block size in an automatic way according to the registers of the kernel and the hardware specifications. Figure 5-7 shows the obtained occupancy for different number of registers and for different computational capabilities of the GPU card when using 256 threads and using other block sizes. For example, the occupancy of a GPU sm13 (compilation with compute capability 1.3) for 35 registers is 25% (dashed blue line) using 256 threads, but it can be 44% (solid blue line) using 448 threads.

Table 5-1. Technical specifications of GPUs according to the compute capability.

Technical specifications	1.0	1.1	1.2	1.3	2.x	3.x
Max. of threads per block	512				1024	
Max. of resident blocks per SM	8					16
Max. of resident warps per SM	24		32		48	64
Max. of resident threads per SM	768		1024		1536	2048
Max. of 32-bit registers per SM	8 K		16 K		32 K	64 K

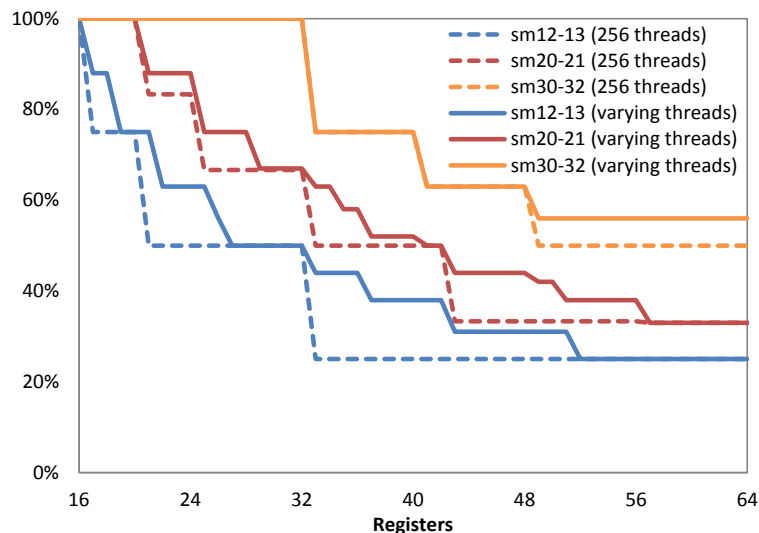


Figure 5-7. Occupancy of the GPU for different number of registers with a variable and a fixed block size of 256 threads.

5.3.2 Reducing global memory accesses

When computing the SPH forces during the particle interaction (PI) stage, the six arrays described in Table 5-2 are used. The arrays *csound*, *prrho* and *tensil* were previously calculated for each particle using *rho* to avoid calculating them for each interaction of the particle with all its neighbors. The number of memory accesses in the interaction kernel can be reduced by grouping part of these arrays (*pos+press* and *vel+rho* are combined to create two arrays of 16 bytes each one) and avoid reading values that can be calculated from other variables (*csound* and *tensil* are calculated from *press*). Thus, the number of accesses to the global memory of the GPU is reduced from 6 to 2 and the volume of data to be read from 40 to 32 bytes.

Table 5-2. List of variables needed to calculate forces.

Variable	Size (bytes)	Description
pos	3 x 4	Position in X,Y and Z
vel	3 x 4	Velocity in X,Y and Z
rho	4	Density
csound	4	Speed of sound
prrho	4	Ratio between pressure and density
tensil	4	Tensile correction following [Monaghan, 2000]

5.3.3 Simplifying the neighbor search

During the GPU execution of the interaction kernel, each thread has to look for the neighbors of its particle sweeping through the particles that belong to its own cell and to the surrounding cells, a total of 27 cells since symmetry cannot be applied. However, this procedure can be optimised when simplifying the neighbor search. This process can be removed from the interaction kernel when the range of particles that could interact with the target particle is previously known. Since particles are reordered according to the cells and cells follow the order of X, Y and Z axis, the range of particles of three consecutive cells in the X-axis ($\text{cell}_{x,y,z}$, $\text{cell}_{x+1,y,z}$ y $\text{cell}_{x+2,y,z}$) is equal to the range from the first particle of $\text{cell}_{x,y,z}$ to the last of $\text{cell}_{x+2,y,z}$. Thus, the 27 cells can be defined as 9 ranges of particles. The 9 ranges are colored in Figure 5-8. The interaction kernel is significantly simplified, when these ranges are known in advance. Thus, the memory accesses decrease and the number of divergent warps is reduced. In

addition, GPU occupancy increases since less registers are employed in the kernel. The main drawback is the higher memory requirements due to the extra 144 bytes needed per cell.

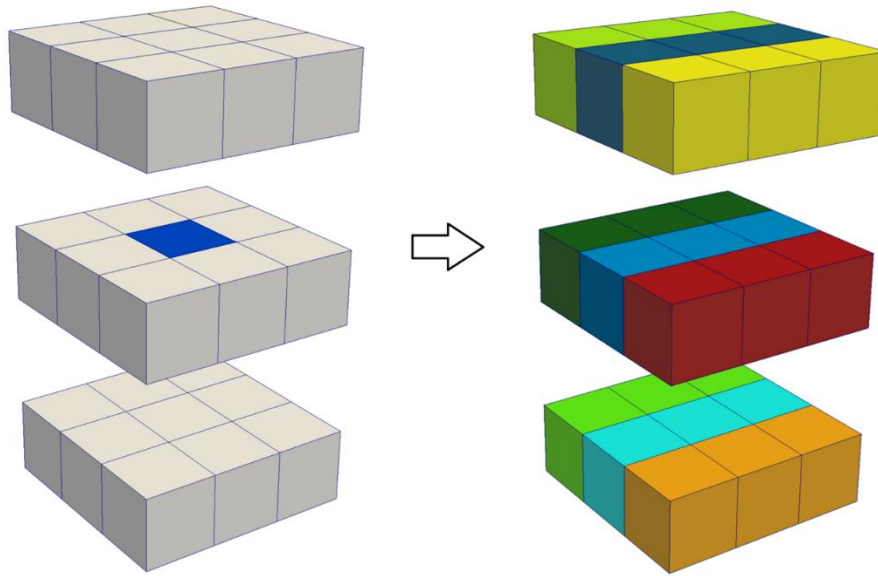


Figure 5-8. Interaction cells in 3D without symmetry but using 9 ranges of three consecutive cells (right) instead of 27 cells (left).

5.3.4 Adding a more specific CUDA kernel of interaction

Initially, the same CUDA kernel was used to calculate all interaction forces boundary-fluid (B-F), fluid-boundary (F-B) and fluid-fluid (F-F). However, symmetry in the force computation cannot be efficiently applied and the best option is implementing a specific kernel for the B-F interaction because only a subset of the fluid particles is required to be computed for the boundaries. The effect of this optimization on the overall performance is negligible when the number of boundary particles is small in comparison with the number of the fluid ones. On the other hand, the access to the global memory of the GPU is two orders of magnitude slower than the access to other registers. In order to minimize these accesses, each thread starts storing all its particle data in registers, so the thread only needs to read data corresponding to the neighbor particles. The same approach is applied to store the forces, which are accumulated in registers and written in global memory at the end. There are two types of particles (boundaries and fluids), so there are three interactions to calculate all the forces (F-F, F-B and B-F). Therefore, data of the fluid particles associated to the threads are read twice (when fluid particles interact with other fluid particles and when they interact with boundaries) and the same occurs when

writing results in the global memory. A way to avoid this problem is carrying out the interaction F-F and F-B in the same CUDA kernel with a single data load and a single final writing instead of two.

5.3.5 Division of the domain into smaller cells

As mentioned in the optimization applied in the CPU implementation (Section 4.1.2), the procedure consists in dividing the domain into cells of size $2h/2$ instead of size $2h$ in order to increase the percentage of real neighbors. Using cells of size $2h$ on the GPU implementation, the number of pair-wise interactions decreases. The disadvantage is the increase in memory requirements since the number of cells is 8 times higher and the number of ranges of particles to be evaluated in the neighbor search increases from 9 to 25 (using 400 bytes per cell).

5.4 RESULTS

The system used for the GPU performance testing:

- **Hardware1:** NVIDIA GTX 480 (15 Multiprocessors, 480 cores at 1.37 GHz with 1.5 GB of 1848 MHz GDDR5 RAM and compute capability 2.0).
- **Hardware2:** NVIDIA Tesla 1060 (30 Multiprocessors, 240 cores at 1.3 GHz with 4 GB of 1600 MHz GDDR3 RAM and compute capability 1.3).
- **Operating system:** Debian GNU/Linux 5.0 (Lenny) 64-bit.
- **Compiler:** CUDA 3.2 (compiling with the option `-use_fast_math`).

Table 5-3 summarizes the improvement achieved on the GPU cards GTX 480 and Tesla 1060 when using the different optimization strategies described before. All results were obtained simulating the testcase of Section 3.2 with 1 million particles. Two variables are shown: the percentage of improvement obtained when applying each individual optimization and the cumulative improvement achieved when including the present and the previous optimizations. It can be also observed the effect of optimizations on both GPU architectures; Tesla 1060 corresponds to the generation of GPUs with 240 cores and with compute capability 1.3 (see Table 5-1) and GTX 480 corresponds to the Fermi architecture with 480 cores and with compute capability 2.0 (see Table 5-1). In fact, this different behavior of both GPU cards is related not only to the compute capability and the number of cores but also to the number of registers and some

kind of cache memory available in the Fermi GPUs that reduces conflicts when accessing to the global memory. For example, *maximizing the occupancy of GPU* presents a better improvement with the Tesla card than with the GTX. Due to the lower occupancy provided by the compute capability sm13 in comparison to sm20, the margin of improvement is higher for the Tesla card (see Figure 5-7). In contrast, the impact of *dividing the domain into smaller cells* is more important with the GTX. The divergence diminishes when using smaller cells but the irregular accesses to memory increases and the GTX card presents that kind of cache memory that helps to mitigate the negative effect of the irregular accesses while the Tesla cannot. Considering the cumulative response of applying all the optimizations, the fully optimized GPU code for the GTX 480 is 1.65 times faster than the basic GPU version without optimizations and, in the case of Tesla 1060, the achieved speedup was 2.15x.

Table 5-3. Improvement achieved on GPU simulating 1 million particles when applying the different GPU optimizations using GTX 480 and Tesla 1060.

	GTX 480		Tesla 1060	
	Optimization	Cumulative	Optimization	Cumulative
Maximizing the occupancy of GPU	7.3%	7.3%	17.4%	17.4%
Reducing global memory accesses	18.9%	27.6%	28.9%	51.3%
Simplifying the neighbor search	3.1%	31.5%	12.9%	70.8%
Specific CUDA kernel of interaction	2.6%	34.9%	11.3%	90.1%
Division of the domain into smaller cells	22.7%	65.4%	12.8%	114.5%

The full implementation of the SPH code on GPU is basic since when neighbor list (NL), particle interaction (PI) and system update (SU) are implemented on GPU, the CPU-GPU data transfer is avoided in each time step. Figure 5-9 shows the computational runtimes using the GTX 480 for different GPU implementations (partial, full and optimized) simulating 500,000 particles of the testcase. *Partial GPU* implementation corresponds to a preliminary version where only the PI stage was implemented on GPU, in the *full GPU* version the three stages of the SPH code are executed on GPU and *optimized GPU* is the final version including all the proposed optimizations. It can be observed that the time dedicated to the CPU-GPU data transfer in the partial implementation is

9.4% of the total runtime. The CPU-GPU communications are not necessary at each time step when the SPH code is totally implemented on GPU. The runtimes of the NL and SU stages decrease when both parts of the code are also implemented on GPU. Finally, the computational time of the PI stage is reduced in about 40% when applying all the developed optimization strategies.

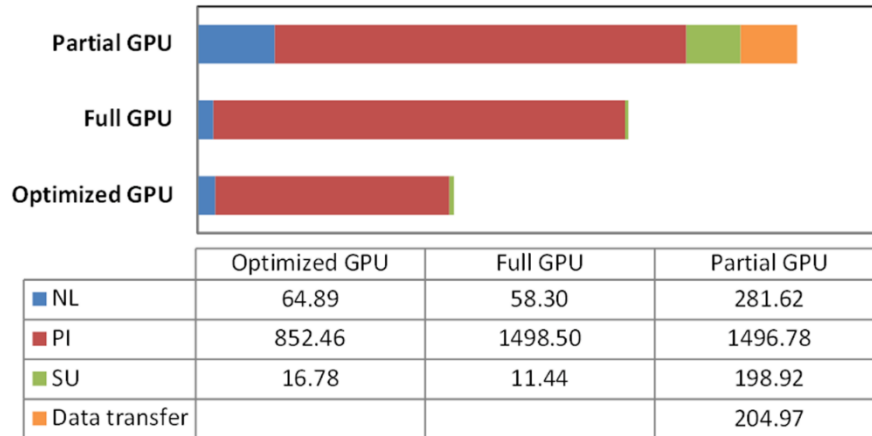


Figure 5-9. Computational runtimes (in seconds) using GTX 480 for different GPU implementations (partial, full and optimized) when simulating 500,000 particles.

In the last years, many performance comparisons between CPU and GPU have been reported achieving speedups over two or three orders of magnitude. However, many of these comparisons are not so fair since a highly optimised GPU code is compared against a basic CPU code, which does not take advantage of the real power of CPU [Lee et al., 2010]. This work shows a comparison once both codes (CPU and GPU) were optimised.

The comparison between CPU and GPU can be observed in Table 5-4. The table summarizes the execution runtimes, the number of computed steps and the achieved speedups. Note that the speedup has been measured here as the ratio between the number of time steps computed per second by the different versions. The data correspond to the most efficient implementation on GPU versus the multi-core implementation on CPU (*Symmetric* with 8 threads) and the single-core implementation. Thus, for example, for one million particles, the performance of the CPU is 0.2 time steps per second using the single-core version and 0.8 using the multi-core version, while 10.1 time steps per second can be computed with a GPU GTX 480. The whole simulation takes one day, 16 hours and 45 min on the Intel® Core™ i7 and only 42 min on the GTX 480, resulting in a speedup of 56.2x (vs. single-core CPU) and 12.5x (vs. CPU with 8 logical threads). It can be also observed that the speedups with GTX 480 (Fermi

technology) are twice those obtained with Tesla 1060, which belongs to a previous generation of GPU cards as mentioned above. Note that, usually, the works about parallel hardware to accelerate SPH published before the appearance of GPUs showed speed-ups considering CPU clusters versus a single core. When proving the capability of GPU computations for engineering applications, relative runtimes can be useful, so the speedup in comparison with a single CPU core is also shown to give an idea of the order of speedup that is possible when using GPU cards instead of large cluster machines.

Table 5-4. Results of the CPU and GPU simulations.

Version	Number of particles	Total simulation time	Number of Steps	Computed steps per second	Speedup vs. CPU Single-core	Speedup vs. CPU 8 Threads
CPU Single-core	503,492	14.6 h	19,855	0.4	1.0x	--
	1,011,354	40.7 h	26,493	0.2	1.0x	--
CPU 8 Threads	503,492	3.2 h	19,806	1.7	4.6x	1.0x
	1,011,354	9.1 h	26,511	0.8	4.5x	1.0x
GPU Tesla 1060	503,492	0.5 h	19,832	10.2	26.8x	5.8x
	1,011,354	1.5 h	26,509	4.9	27.3x	6.1x
GPU GTX 480	503,492	0.3 h	19,830	21.2	55.7x	12.2x
	1,011,354	0.7 h	26,480	10.1	56.2x	12.5x

The fastest GPU implementation uses all the GPU optimizations including *dividing the domain into smaller cells*, whose main disadvantage is the increase in memory requirements as mentioned above. Therefore, the maximum number of particles that can be simulated in a GTX 480 using the optimized GPU version of DualSPHysics is only 1.8 million. Accordingly, three different versions of the code are implemented to avoid this limitation. These different GPU versions are available in DualSPHysics and the fastest one is automatically selected by the code depending on the memory requirements of the simulation. The first version contains all the GPU optimizations and it is named *FastCells(2h/2)*, the second one, named *SlowCells(2h/2)*, is implemented without the optimization of *simplifying the neighbor search* and the third version, named *SlowCells(2h)*, is implemented without *simplifying the neighbor search* and without *dividing the domain into smaller cells*. The memory usage for these three different GPU versions can be seen in Figure 5-10. Note the black solid line represents the limit of memory that can be allocated on a GTX 480 (less than 1.4 GB) and the dotted line the limit for the Tesla 1060 (less than 4GB). Using all these different

versions, which will be automatically selected for each run depending on memory requirements, DualSPHysics allows simulating up to 9 million particles with a GTX 480 and more than 25 million with a Tesla 1060. The execution times corresponding to the three GPU versions (*FastCells(2h/2)*, *SlowCells(2h/2)* and *SlowCells(2h)*) and the times of the single-core and multi-core CPU versions are also summarized in Figure 5-11.

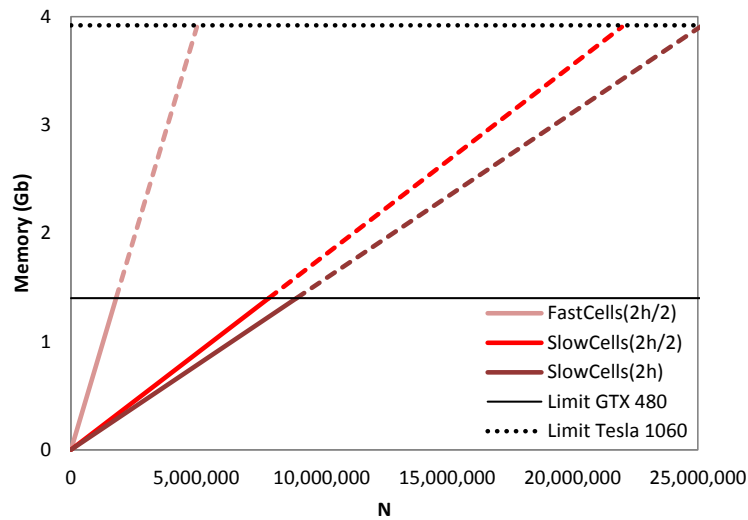


Figure 5-10. Memory usage for different GPU versions implemented in DualSPHysics.

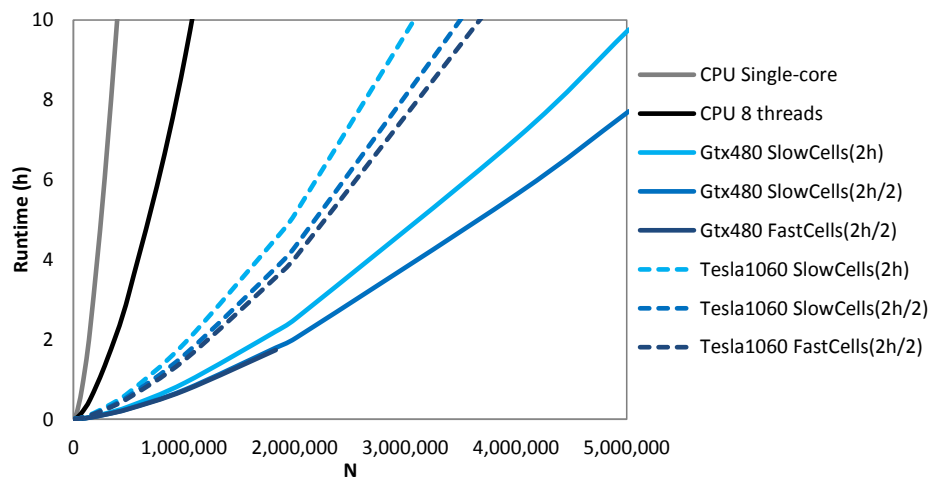


Figure 5-11. Runtimes for different CPU and GPU implementations.

5.5 PERFORMANCE WITH THE LATEST GPU (AUGUST 2014)

GPU technology is continuously improving, not only their performance increases but also the architecture is optimised. The results presented above were not

obtained with the latest GPUs in the market. Therefore, new results are presented in this section using novel and more powerful such as GTX 680, Tesla K20 and GTX Titan. The same testcase is executed now using the Intel Xeon X5500 CPU and the mentioned GPUs. Note that the previous GTX 480 is also included in the comparison to highlight the improvement achieved with newest cards. General specifications of the execution devices are summarised in Table 5-5.

Table 5-5. Specifications of different execution devices.

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

The performance of different simulations of the same case is presented for 1.5 seconds of physical time. The performance is analysed for different resolutions by running calculations with different numbers of particles. Computational times of the executions on CPU and GPU are shown in Figure 5-12 where it can be noticed that for a simulation of 3 million particles takes one hour using the GTX Titan GPU card while it takes almost 2 days using a CPU.

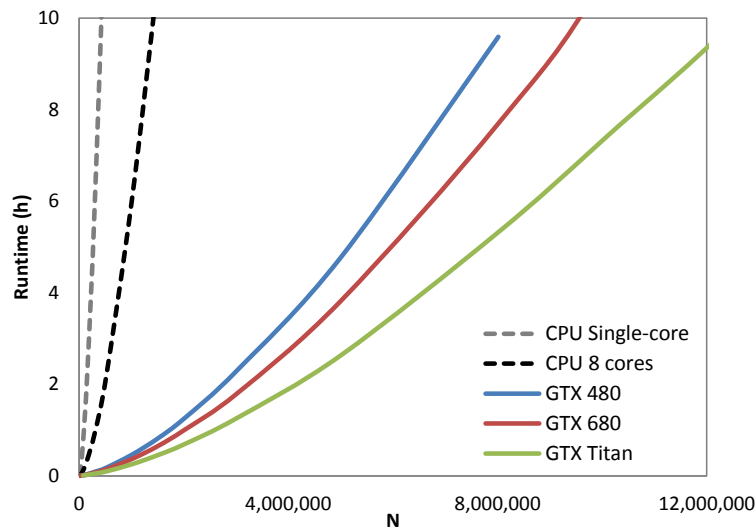


Figure 5-12. Runtime for CPU and different GPU cards.

This important acceleration of the code using the new GPU technology can also be observed in Figure 5-13, where the speedups of different GPUs are shown by

comparing their performance against the CPU device using a single core and also the full 8 cores of the Intel Xeon X5500. For the case chosen here, the use of a GPU can accelerate the SPH computations by almost two orders of magnitude, e.g. the GTX Titan card is 149 times faster than the single core CPU and 24 times faster than the CPU using all 8 cores.

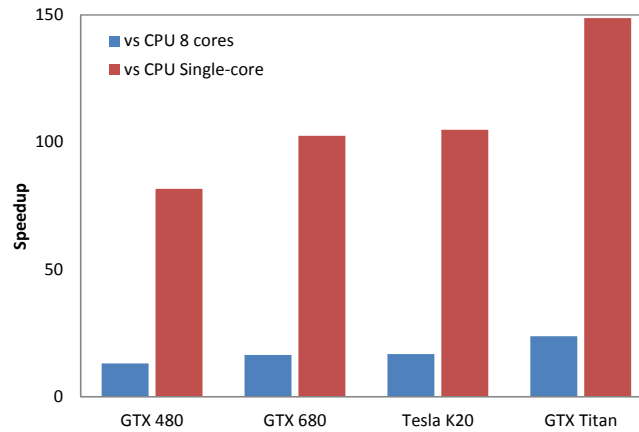


Figure 5-13. Speedups of GPU against CPU simulating 1 million particles.

Figure 5-14 shows the runtime distribution of the three main SPH steps; neighbour list (NL) creation, particle interaction (PI) and system update (SU) when simulating one million particles. The particle interaction takes 98.5% of the total computational time when using a CPU single-core and this percentage decreases when the code is parallelised. Hence PI takes 90.8% when using the 8 cores of the CPU and it is reduced to 88.3% and 85.7% when using GPU cards (GTX 480 and GTX Titan, respectively). On the other hand the percentages of NL and SU increase with the number of cores to parallelise over.

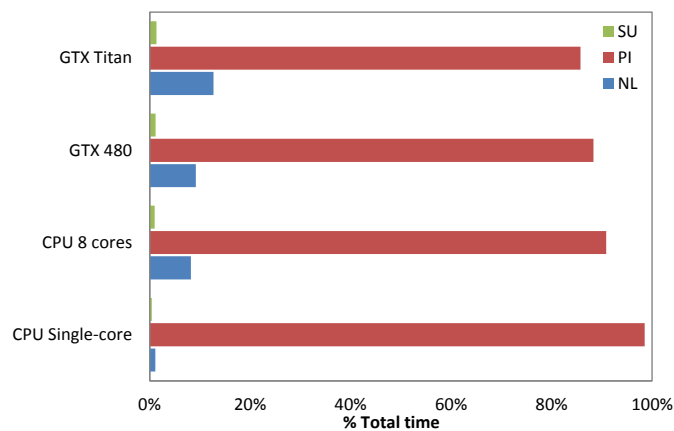


Figure 5-14. Computational runtime distribution on CPU and GPU simulating 1 million particles. Neighbour List corresponds to blue bars, Particle Interaction to red bars and System Update to the green bars.

Finally, Figure 5-15 gives an idea of how many particles can be simulated on the different GPU devices employed when using the DualSPHysics code. It can be observed that the difference in terms of speedup between GTX 680 and Tesla K20 is negligible (see Figure 5-13) and the main difference of using these two GPU cards lies in the memory space that allows simulating more than 28 million particles in one Tesla K20 while less than the half can be simulated with a GTX 680.

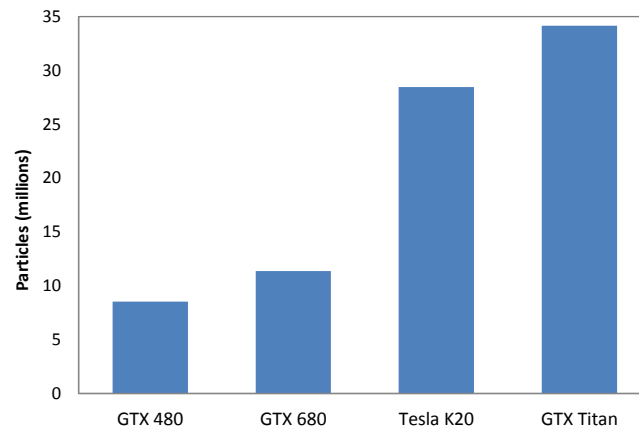


Figure 5-15. Maximum number of particles simulated with different GPU cards using DualSPHysics code.

6. MULTI-GPU ACCELERATION

In previous chapters, it has already been explained that SPH method presents a high computational cost and, hence, it is necessary to increase the velocity of the method. It is imperative to carry out real simulations where the number of particles is very high. The use of GPUs can provide large speedups compared to classical solutions based on CPUs. However, the use of a single GPU card is not sufficient for engineering applications that require several million particles that predict the desired physical processes: execution times are high and the available memory space is insufficient. Multiple spatial scales are present in most phenomena involving free-surface waves. Scales that range from hundreds of metres to centimetres are necessary to describe accurately the coastal hydrodynamics. Thus, most of the relevant phenomena in coastal engineering involve spatial scales over 4–5 orders of magnitude. For large simulations it is therefore essential to take advantage of the performance of multiple GPUs.

This section presents a novel SPH implementation that utilizes MPI and CUDA to combine the power of different devices making possible the execution of SPH on heterogeneous clusters (Figure 6-1). Specifically, the proposed implementation enables communications and coordination among multiple CPUs, which can also host GPUs, making possible multi-GPU executions.

A scheme for multi-GPU SPH simulations was presented by [Valdez-Balderas et al., 2012]. In that work, a spatial decomposition technique was described for dividing a physical system into fixed sub-domains, and then assigning a different GPU of a multi-GPU system to compute the dynamics of particles in each of those sub-domains. The Message Passing Interface (MPI) was used for communication between devices, i.e. when particles migrate from one sub-

domain to another, and to compute the forces exerted by particles on one sub-domain onto particles of a neighbouring sub-domain. The algorithm was only tested up to 32 million particles on 8 GPUs at a fraction of the computational cost of a conventional HPC cluster.

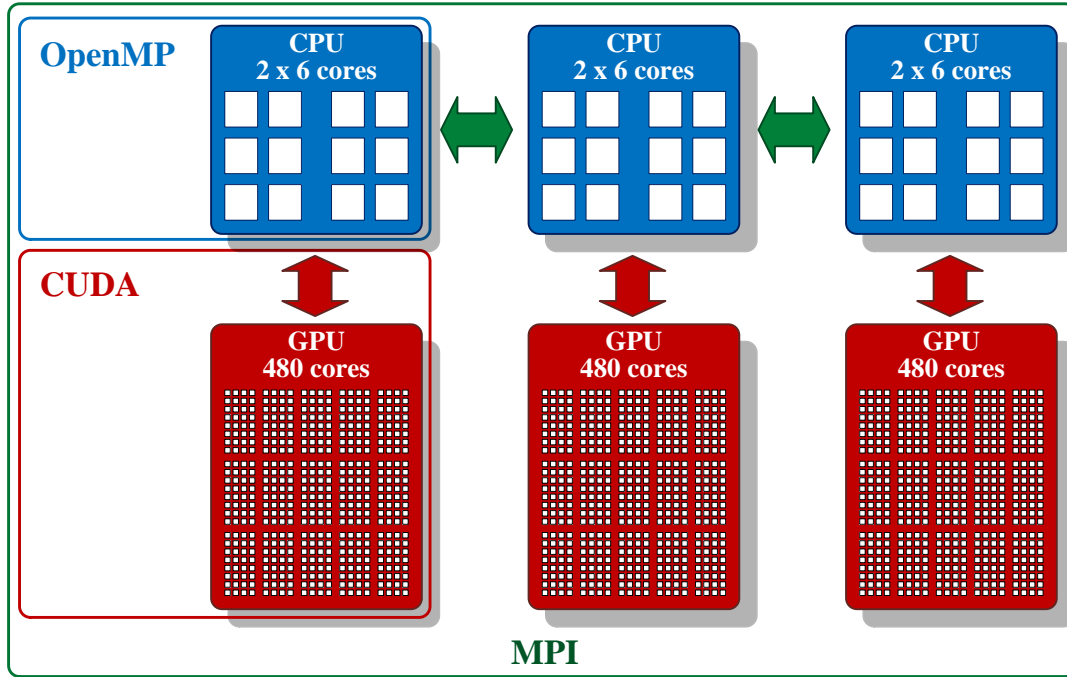


Figure 6-1. Scheme of technologies and its scope of application.

In the work of [Fleissner and Eberhard, 2007], [Ferrari et al., 2009] and more recently in [Maruzewski et al., 2010], MPI was also used to distribute the work load of SPH on multiple devices, although these studies used only CPUs. As in the case of [Valdez-Balderas et al., 2012], they applied a spatial decomposition of the domain into subdomains and each one was assigned to a processor but with a dynamic load balancing algorithm, which is not included in the scheme of [Valdez-Balderas et al., 2012]. However, in both cases the efficiency drops quickly by increasing the number of execution devices. A multi-GPU implementation of the SPH method was also presented in [Rustico et al., 2014]. In that work an asynchronous API (offered by CUDA) was used instead of MPI to execute the model in several GPUs hosted in one machine. Apart from the work of [Valdez-Balderas et al., 2012] little research has been published using multi-GPU schemes for SPH, but other types of particle-based simulations have already been the target of parallelization on multi-GPU systems. The field of classical molecular dynamics (MD) has perhaps seen the most extensive use of this technology, given the widespread use of this technique in the fields of physics, material science, and biology. For example, a series of publications focusing on the multi-GPU implementation of the code LAMMPS has been

written recently. Those include the work of [Brown et al., 2011] describing the implementation of a hybrid GPU-CPU code for MD systems of short-ranged interactions; [Trott et al., 2010] who presented more general capabilities added to LAMMPS to simulate a wide variety of materials; and the efforts of [Agarwal et al., 2012] on porting, optimizing, and tuning LAMMPS for biological simulations. Other significant works on MD on multi-GPU heterogeneous systems are [Qiang et al., 2012]. Although SPH and classical MD are both particle-based simulation techniques with strong similarities in their algorithms and data structures, they are intended for simulations of different types of systems. In MD the particles represent atoms, molecules, or coarse-grained model modelling of a material system, a continuum is discretized in SPH, and particles represent interpolation nodes. Consequently, interactions, boundaries and initial conditions are different. SPH in the form presented here is intended to simulate free-surface hydrodynamic flows, which inherently present abrupt variations of the density at the fluid surfaces, which, in turn, move rapidly during a simulation. MD, on the other hand, is typically tested on systems in which comparatively smaller fluctuations of density both in space and time, while at the same time tend to include a wider variety of particles and types of interactions. Consequently, one can expect that aspects of the problem, like the dynamic balancing of computing load among all available devices, be significantly different in SPH and in MD.

6.1 MPI IMPLEMENTATION

The parallel programming architecture Compute Unified Device Architecture (CUDA) developed by Nvidia is used to obtain an efficient and extensive use of the capabilities of the GPU architecture. In addition, a second level of parallelisation is applied by using MPI, where a set of directives enables communication between devices and allows combining the resources of several machines connected by a network. The execution power can therefore be increased easily by adding new machines. However, the division of the work load among different independent devices implies an extra computational cost. This extra runtime comes from; (i) the execution of new processes dedicated to manage the distribution of the work load, (ii) the time dedicated to data exchange and, (iii) the time consumed during synchronisations. These were investigated previously by [Valdez-Balderas et al., 2012].

The parallel implementation for SPH methods presented in this work use these parallelisation techniques with one or several machines connected in a network. This enables computations on heterogeneous clusters taking advantage of all available processing units. This is important since clusters can then be extended with new GPUs of different specifications. In the next section, we introduce a new implementation of this parallelism to obtain greater efficiencies from the additional hardware.

This section describes in more detail the proposed MPI implementation that will give rise to some interesting improvements but also to drawbacks (these will be assessed in Section 6.2.5).

The physical domain of the simulation is divided into subdomains among the different MPI processes. In this way, each process only needs to assign resources to manage a subset of the total amount of particles for each subdomain. Thus, the size of the simulation scales with the number of machines.

The two main sources of efficiency loss when the number of MPI processes is increased include data exchange between the devices managed by the processes and synchronisation. In the previous MPI implementation [Valdez-Balderas et al., 2012], it was observed that the time dedicated to communication constitutes a high percentage of the total execution time and that this percentage increases significantly with the number of processes. One option to reduce this time is to subdivide the calculation of forces on each subdomain so that communications and computation can overlap. When considering synchronisation of processes across devices, the SPH algorithm benefits from using a variable time step computed following Eq. 2.24. The new value is obtained from variables (force and viscous terms) that are known only after computing all particle interactions, that is, when all MPI processes have finished the force computation step. This synchronisation requires all processes to wait for the slowest process. Since the number of steps to complete a simulation is large, typically on the order of 10^4 - 10^6 , this implies a non-negligible loss of efficiency that also increases with the number of processes. To address this problem, the computational load or demand must be evenly divided among all processes, minimising the difference between the computation time needed for the fastest and the slowest process.

6.1.1 Subdivision of the domain

As mentioned above, the domain is divided into subdomains or blocks of particles that are assigned to the different MPI processes (Figure 6-2). This division can be performed in any direction (X, Y or Z) adapting to the nature of the simulation case. In this way, each subdomain has two neighbouring subdomains, one on either side, except those subdomains at the perimeter of the simulation box, which have only one neighbour. Each MPI process needs to obtain, at every time step, the data of neighbouring particles from the surrounding processes within the interaction distance ($2h$ here). Therefore, to calculate the forces exerted on the particles within its assigned subdomain, each process needs to know the data of particles from the neighbouring subdomains that are located within the interaction distance. We call this the *halo* of the process (or subdomain) existing on the *edge* of the neighbouring process (or subdomain).

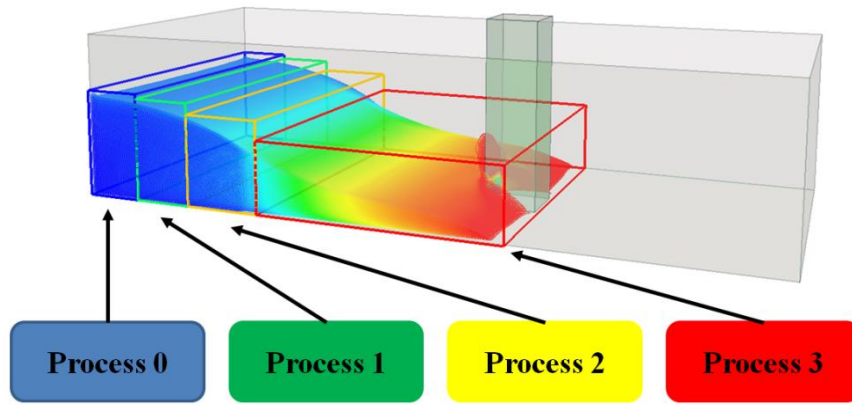


Figure 6-2. Domain subdivision in four processes.

Figure 6-3 shows the division of a domain into three subdomains (0, 1 and 2). Thus, grey particles belong to subdomain 1 but some of them, those that are in the left *edge* and at a distance less than $2h$ from domain 0, constitute the *halo* of subdomain 0 while the grey particles in the right *edge* constitute the *halo* of subdomain 2.

Unlike the scheme presented in [Valdez-Balderas et al., 2012], the data of *halo* particles of a given subdomain are not stored in the same data structure that holds the subdomain particles. Instead, in order to determine the *halo* of a given subdomain, information from a previous stage in the algorithm is used. That is, during the neighbour list stage, particles are sorted in cells of size $2h$ [Domínguez et al., 2013a] and the order can be XYZ, XZY or YZX according to the division axis Z, Y or X. As a consequence of this cell sorting, particles are

also sorted within slices $2h$ wide within each subdomain. The subdomain assigned to each process is chosen to have a minimum width of $6h$ to ensure a minimum size of $2h$ for the two *edges* of that domain ($2h$ from left *edge* + $2h$ inside the domain + $2h$ from right *edge*).

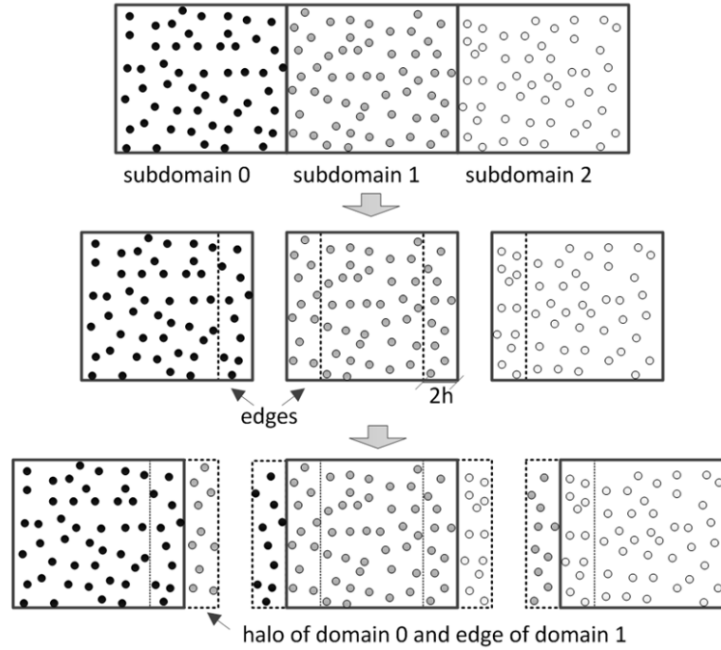


Figure 6-3. Example of subdivision of a domain (halos and edges).

This approach to divide the domain among the MPI processes where particles are reordered according to the direction of the domain subdivision gives rise to interesting advantages that increase performance:

- a) If particles of a subdomain are not merged with particles of the *halo*, no time is wasted in reordering all particles when receiving data from the *halo* before a force computation, and no time is wasted in separating them after the force computation. The memory usage is also reduced since only the basic properties of *halo* particles need to be stored (position, velocity and density).
- b) Each process can adjust the size of its subdomain with the limits of the fluid particles inside. With the number of cells minimised for each subdomain, the total number of cells over the entire is also minimised leading to a reduction of the execution time and memory requirements.
- c) Particle data of the subdomains is stored in slices. Data existing in the *edges* can be sent to the neighbouring processes much faster since all data is grouped in consecutive memory positions.
- d) This reordering system also enables automatic identification of particles contained in a subdomain needing to interact with the halo. Thus, task

overlapping is possible by computing particle interactions of particles not on an *edge*, while simultaneously performing the reception of the *halo* (needed only by *edge* particles), thereby inherently saving the communication time. For example, the grey particles in Figure 6-3 that belong to the left *edge* of the subdomain 1 also form the *halo* of subdomain 0 and force interactions between red particles not on the *edge* of subdomain 0 can be computed while they wait for the *halo*.

- e) Symmetry of pairwise interactions is not necessary for the particles of the *halo* since *halo* and *edge* particles only interact once in each process. This is relevant to the GPU implementation since symmetry is not applied for the pair-wise computations and does not represent any loss of performance in comparison to single-GPU version.

6.1.2 Communication among processes

Reducing time dedicated for exchanging data among MPI processes is essential to increase the number of processes without drastically decreasing efficiency. One method to achieve this is by overlapping the communication with the computation using asynchronous communications. Hence asynchronous send operations and synchronous receptions are used in the present algorithm. In this way, one process can send information to another while carrying out other tasks without waiting for the end of the transfer. This is an improvement over an algorithm that uses synchronous operations, in which an MPI process cannot continue execution of tasks until the operation is complete, implying a wait to receive data from another process or processes, thereby rendering computational resources idle, and consequently causing loss of efficiency.

Figure 6-4 shows the data exchanges that take place at each time step when using MPI. Three different processes are considered in this example. There are two important communications; the first one occurs during the neighbour list creation (solid arrows in Figure 6-4) and the second one in the force computation (dashed arrows in Figure 6-4). The dark arrows represent the submissions from process N to process $N+1$ and the light ones from process N to process $N-1$. Double-headed arrows show the synchronisation point after the force computation stage. Note that all the tasks corresponding to interactions among particles correspond to the boxes with grey background in the figure.

During the force computation, each process sends its *edges* to the surrounding processes (dashed arrows in Figure 6-4). While *edges* are being sent and the *halo* is received, computation of the force on the *interior* particles is performed. Once this is finished, the process waits for the reception of the first *halo* and computes forces of one *edge* with this *halo*. After that, the process waits to receive the second *halo* and computes the forces of the other *edge*. Thus the most expensive *halo-edge* data transfers are overlapped by calculating the forces between particles (also very time consuming). All particle data are allocated in the GPU memory, so data transfer is also needed between CPU and GPU memories. However, it should be noted that the cost is negligible since the volume of information is low and one of the advantages of the method proposed here is that the data to be transferred are stored in contiguous memory locations, which accelerates the process.

6.1.3 Dynamic load balancing

Due to the Lagrangian nature of SPH, particles move through space during the simulation so the number of particles must be redistributed after some time steps to maintain a balanced work load among the processes and minimise the synchronisation time. Most of the total execution time is spent on force computation, and this time depends mainly on the number of fluid particles. For an equal load per processor, the domain must be divided into subdomains with the same number of fluid particles (including particles of the *halos*) or with the number of particles appropriate to the computing power of the device assigned to it.

Two different dynamic load balancing algorithms are used. The first one assigns the same *number* of particles to each computing device, and is suitable when the simulation is executed on machines that present the same performance. The second load balancing algorithm is used when hardware of different specifications and features are combined, such as different models of GPU. This second approach takes into account the *execution time on each device*. In particular, a weighted average of the computing time per integration step over several steps (on the order of 30) is used, with a higher weight to the most recent steps. An average over many time steps is chosen because a single time step presents large fluctuations. This average time is used to distribute the number of particles so that the fastest devices can compute subdomains with more particles than the slowest devices.

The example depicted in Figure 6-5 can help to explain the second approach that takes into account the *execution time on each device* to balance the work load between GPUs. Thus, in the example, the first row in Figure 6-5 shows the distribution of the slices between two GPUs at a given step where the average time of the force computation during the last 30 steps was 9 ms in the first GPU and 6 ms in the second one. Therefore, a new distribution of the slices between GPUs is desired where the maximum computation time must be minimal. The second row shows the actual time dedicated to force computation for each device since this time is the summation of the times required to compute forces of particles within the slices plus the particles of the *halo*, i.e. $9\text{ ms} + 1\text{ ms} = 10\text{ ms}$ for the first GPU and $6\text{ ms} + 1\text{ ms} = 7\text{ ms}$ for the second one. That is, the maximum computation time in this case is 10 ms and it is therefore desirable to apply a new balancing if the current maximum time can be reduced in a given percentage. In the third row of the figure, it can be seen how a redistribution of the slices between the two GPUs is performed where the second GPU (GPU1) will compute particles within one extra slice originating from GPU0. So that the maximum time has been reduced from 10 ms to 9 ms leading to an improvement of a 10% can be achieved with this example distribution.

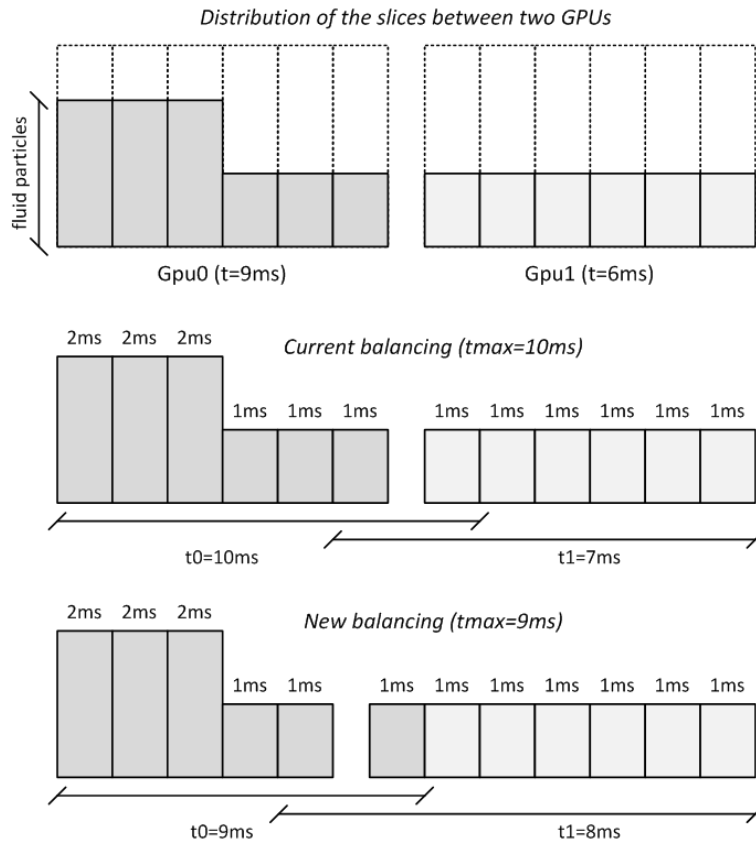


Figure 6-5. Example of the dynamic balancing scheme between 2 GPUs.

This second type of dynamic load balancing enables the adaptation of the code to the features of a heterogeneous cluster achieving a better performance. Thus although the balance is checked every few steps, it is only applied when it implies an improvement in the performance, and therefore its cost is minimal. In fact, the runtime consumed by this checking is usually higher than the computational time dedicated to balance since this is not carried out very often.

6.2 RESULTS

6.2.1 Testcases and hardware

Two testcases are used to analyse the performance of the new MPI-CUDA implementation. The first one is the testcase already used in Section 3.2. Figure 6-6 shows a sketch and several instants of the simulation of the testcase involving six million particles for a physical time of 1.5 seconds.

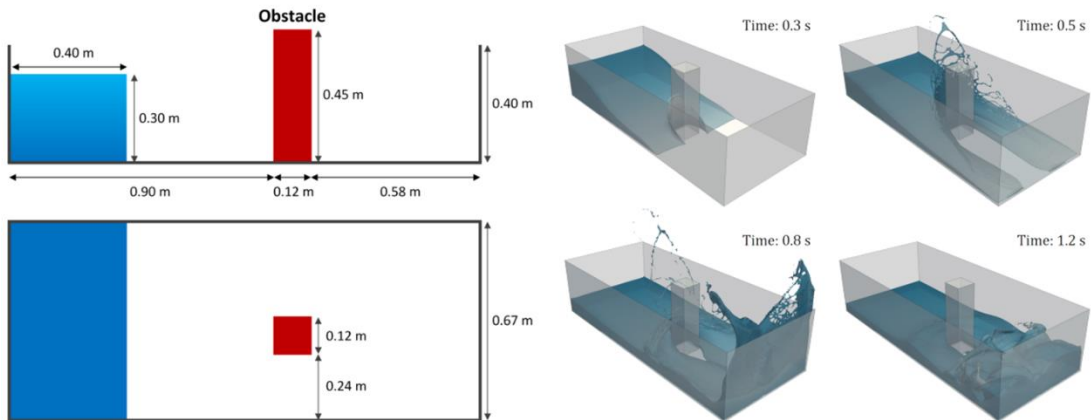


Figure 6-6. Testcase1: Dam break flow impacting on a structure.

The second testcase is also a dam break similar to the previous one, but the main differences are that there is no obstacle in the middle of the numerical tank and the width of the tank can be modified according to the number of particles to be simulated. Note that modifying the width, the number of particles can vary keeping the same number of steps to complete the simulation and the same number of neighbouring particles of each particle. Thus, this testcase, shown in Figure 6-7, is used to analyse the performance for different numbers of particles (from 1 to 1,024 million) to simulate 0.6 seconds of physical time.

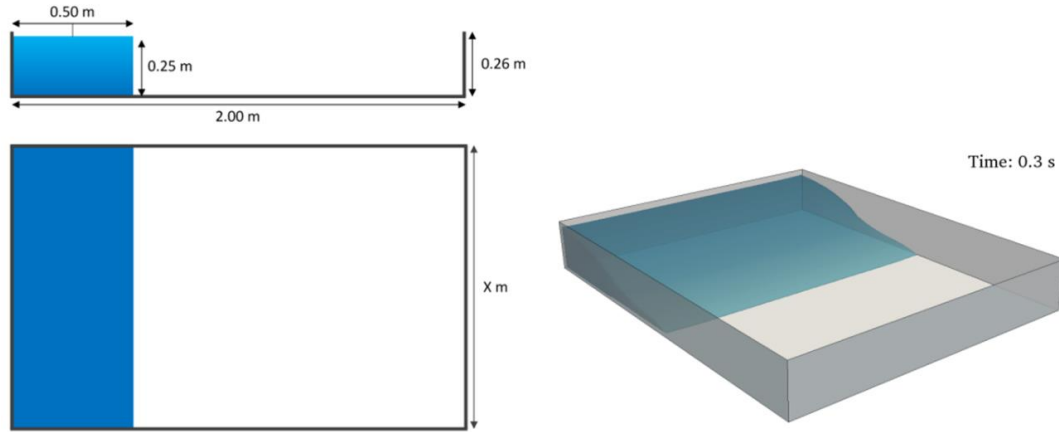


Figure 6-7. Testcase2: Dam break flow.

The simulations were carried out in four different systems at the University of Vigo (Spain), the University of Manchester (United Kingdom) and the Barcelona Supercomputing Center BSC-CNS (Spain). The specifications of each of those systems are summarised in Table 6-1. Two systems that belong to the University of Vigo (system#1a and system#1b) which have only one node were used to evaluate the different approaches of the dynamic load balancing (according to the number of particles and according to the time required for each machine). The system #2 (The University of Manchester) and system #3 (BSC-CNS) are built with several nodes (8 and 64 respectively) and they were used to analyse the performance and scalability (strong and weak scaling). The efficiency achieved using 8 nodes (16 GPUs) will be also confirmed analysing the efficiency with 64 nodes (128 GPUs). All the results presented in this work were obtained using CUDA 4.0, single precision and Error-correcting code memory (ECC) disabled.

Table 6-1. Features of the different systems used.

System #1	Software	<ul style="list-style-type: none"> - CentOS 5.5 - MPICH2 1.2.1 - CUDA 4.0 - gcc 4.1.2
	Hardware	<p>System #1a: 1 <i>homogenous</i> node with:</p> <ul style="list-style-type: none"> - 2 x Intel Xeon E5620 (4 cores at 2.4 GHz with 16 GB RAM) - 4 x GTX 480 Fermi (15 Multiprocessors, 480 cores at 1.40 GHz, 1.5 GB GDDR5, Compute capability 2.0) <p>System #1b: 1 <i>heterogeneous</i> node with:</p> <ul style="list-style-type: none"> - 2 x Intel Xeon E5620 (4 cores at 2.4 GHz with 16 GB RAM) - 1 x GTX 680 Kepler (8 Multiprocessors, 1536 cores at 1.14 GHz, 2 GB GDDR5, Compute capability 3.0) - 1 x GTX 480 Fermi (15 Multiprocessors, 480 cores at 1.40 GHz, 1.5 GB GDDR5, Compute capability 2.0) - 1 x GTX 285 (30 Multiprocessors, 240 cores at 1.48 GHz, 1 GB GDDR5, Compute capability 1.3)
System #2	Software	<ul style="list-style-type: none"> - Red Hat Enterprise Linux Server 6.2 - Open MPI 1.5.4 - CUDA 4.0 - gcc 4.4.6
	Hardware	<p>8 nodes connected via QDR Infiniband with:</p> <ul style="list-style-type: none"> - 2 x Intel Xeon L5640 (6 cores at 2.27 GHz with 24GB RAM) - 2 x Tesla M2050 (14 Multiprocessors, 448 cores at 1.15 GHz, 3 GB GDDR5, Compute capability 2.0)

System #3	Software	<ul style="list-style-type: none"> - Red Hat Enterprise Linux Server 6.0 - BullxMPI 1.1.11 - CUDA 4.0 - Intel C++ Compiler XE 12.0
	Hardware	128 nodes connected via QDR Infiniband with: <ul style="list-style-type: none"> - 2 x Intel Xeon E5649 (6 cores at 2.53 GHz with 24GB RAM) - 2 x Tesla M2090 (16 Multiprocessors, 512 cores at 1.30 GHz, 6 GB GDDR5, Compute capability 2.0)

First, the difference of the different load balancing schemes are compared for homogeneous and heterogeneous clusters

6.2.2 Applying dynamic load balancing in a homogeneous cluster

This section presents the results when using the dynamic load balancing according to the number of particles. Testcase1 (Figure 6-6) is simulated using system #1a (3 x GTX 480) so the domain is divided in 3 processes (3 GPUs) along the x -direction. Different instants of the simulations are shown in Figure 6-8. The limits of the three different subdomains are depicted using coloured boxes. The size of the different subdomains changes with time to keep the work load evenly distributed among processes (similar number of particles per process).

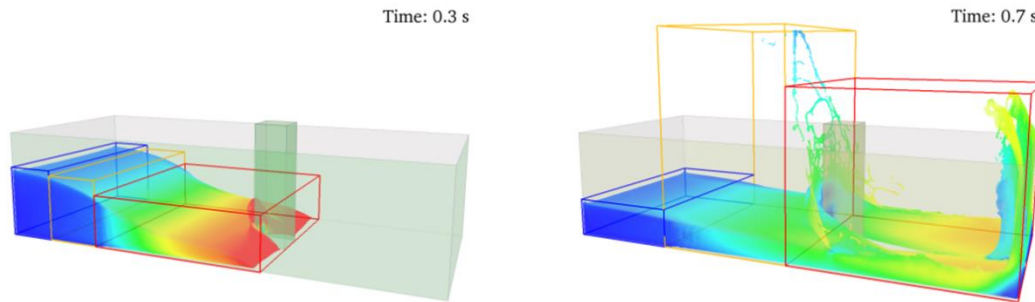


Figure 6-8. Different instants of the simulation of testcase1 when using the dynamic load balancing according to the number of particles.

The left panel of Figure 6-9 shows the distribution of the fluid particles among the 3 processes showing how the balancing is achieved since about the 33.33% of the particles are always computed for each process during the simulation. Since system #1a is homogeneous with the same three GPUs, the time dedicated to the force computation step for each GPU is also balanced as seen in the right panel of Figure 6-9. A total amount of 42,624 steps were performed to complete this simulation, the balancing was checked every 50 steps, so 852 times (0.04% of the total simulation time) but it was performed only 94 times (0.03% of the total simulation time).

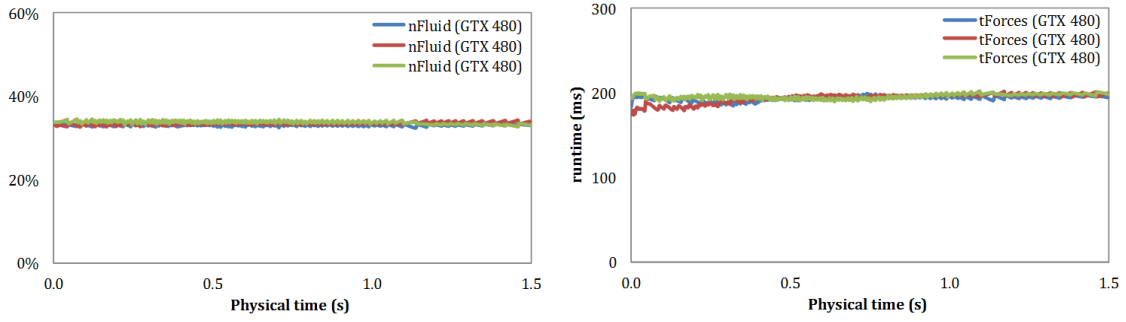


Figure 6-9. Distribution of the fluid particles and execution times of force computation among the 3 GPUs of system #1a using load balancing according to the number of particles.

6.2.3 Applying dynamic load balancing in a heterogeneous cluster

The dynamic load balancing scheme was also applied in the same testcase1 but using system #1b, which is a heterogeneous system since the 3 GPUs present different specifications and performances. From Figure 6-10, it can be concluded that the approach of the balancing according to the number of particles is not suitable in this case since despite the even distribution of the number of particles among the processes, the computation times are not balanced at all. The GPU card GTX 285 is much slower than the other two cards and the time required to compute the same number of particles is considerably higher than needed for the other two cards.

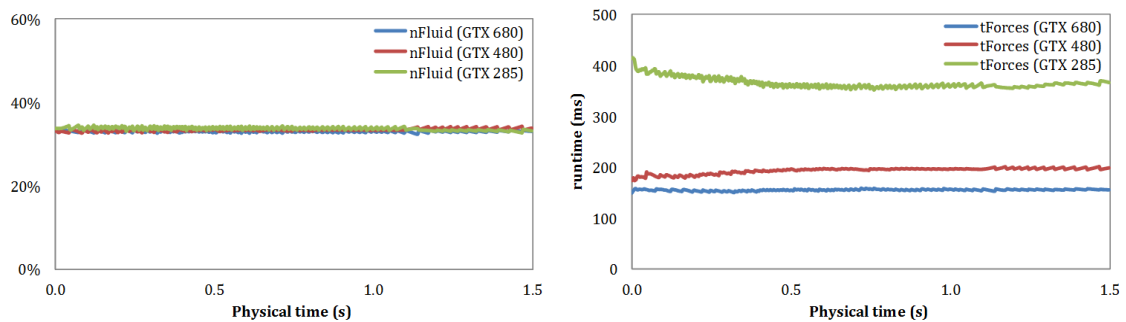


Figure 6-10. Distribution of the fluid particles and execution times of force computation among the 3 different GPUs of system #1b using load balancing according to the number of particles.

The second proposed option consists of the algorithm described in Section 6.1.3 based on the computation time required to compute the forces. In this way, a number of particles is assigned to each GPU card according to its performance to get a correct balance of the work load among the different GPUs. Figure 6-11

shows the different distribution of particles assigned to the three GPUs of the system #1b and the execution times to compute the particle interactions, which are very similar. Thus, the slowest card no longer represents a bottleneck.

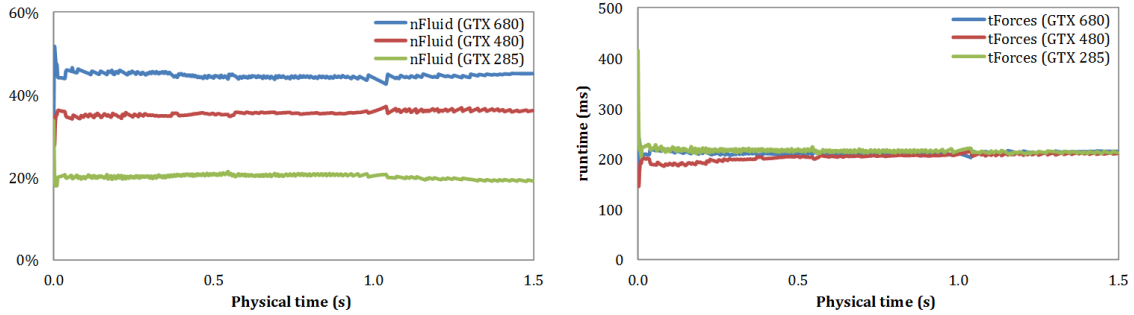


Figure 6-11. Distribution of the fluid particles and execution times of force computation among the 3 different GPUs of system #1b using load balancing according to the computation time.

The execution of testcase1 on one GTX 680 card takes 5.8 hours; combining this GPU with GTX 285 and GTX 480, whose performance characteristics are lower, the run takes 4.6 hours applying the dynamic load balancing according to the number of fluid particles, and only 2.8 hours applying the balancing based on the computation time of each device. Figure 6-12 summarises the execution times of the 3 GPUs of the system #1b when used individually and together.

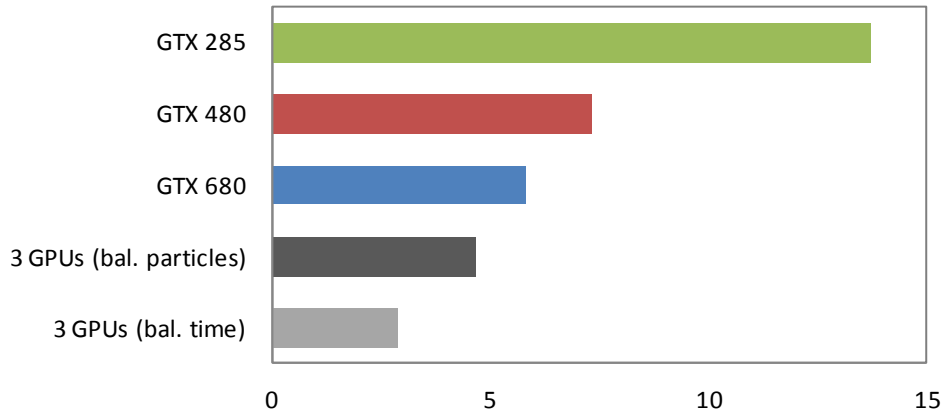


Figure 6-12. Execution times of the 3 GPUs of the system #1b used individually and together applying dynamic load balancing.

6.2.4 Efficiency and scalability

One of the main objectives of the proposed multi-GPU implementation using MPI is the possibility of simulating large systems (10^7 - 10^9 particles) at

reasonable computational times, which is imperative to use the model in real-life applications that require high resolutions. An efficient use of the resources to minimise the computational and economical cost will make these large scale simulations viable. Therefore, a study of the efficiency and scalability of the multi-GPU implementation is shown in this section.

The performance is measured as the number of steps computed per second using two approaches; (i) strong scaling $S(N)$ that determines how the solution time T varies with the number of processors N for a fixed total problem size; and (ii) weak scaling $s(N)$ that defines how the solution time varies with the number of processors for a fixed problem size per processor. Table 6-2 shows the formulae used to measure the speedups and efficiency using these two measures.

Table 6-2. Formulae to measure efficiency and scalability.

Strong scaling	$S(N) = \frac{T(N_{ref})}{T(N)}$
Weak scaling	$s(N) = \frac{T(N_{ref}) \cdot N}{T(N) \cdot N_{ref}}$
Efficiency	$E(N) = S(N)/N$

Testcase2 (Figure 6-7) is used here to evaluate the performance using the different number of GPUs of the systems #1a, #2 and #3. The achieved speedups are shown in Figure 6-13 analysing the strong scaling (left) and the weak scaling (right) for the different hardware systems.

For system #1a with only 4 Fermi GTX 480, the speedup is shown for 2, 3 and 4 GPUs simulating the testcase2 with sizes ranging from 1M to 8M particles for strong scaling and from 1M to 8M particles per GPU to quantify the weak scaling. As expected, the efficiency decreases with the number of GPUs. Thereby, using 4 GPUs and analysing the strong scaling, an efficiency of only 66% is achieved simulating 1M particles but 94% is achieved when simulating 8M because the proportion of time spent on communication is far smaller. Examining the weak scaling, an efficiency of 85.6% is obtained simulating 1M particles per GPU, but this value increases to 99.9% computing 8M per GPU.

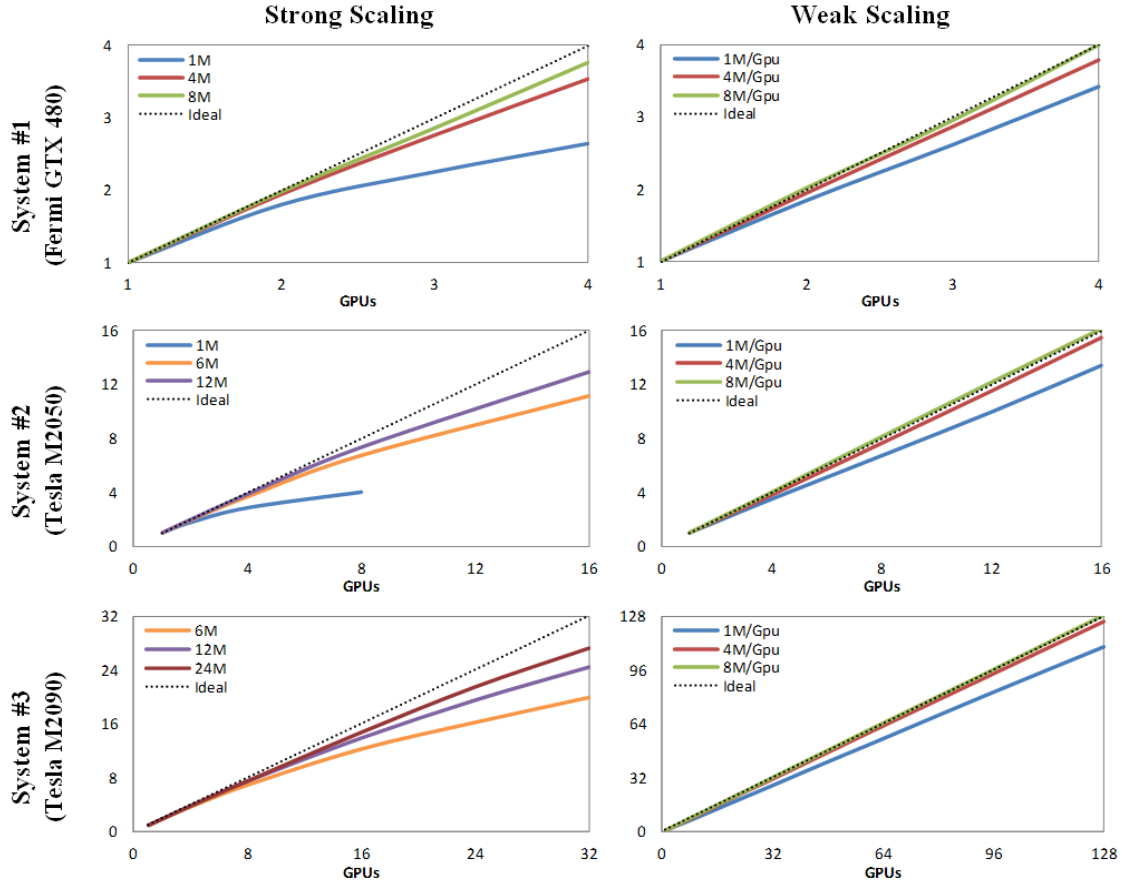


Figure 6-13. Speedup for different number of GPUs using strong and weak scaling with the hardware systems #1a, #2 and #3.

In the case of the system #2 with 16 Tesla M2050, the efficiency analysing the strong scaling is significantly reduced when the number of GPUs increases for a small number of particles. For example, an efficiency of 50% is obtained simulating 1M particles with 8 GPUs, but this amount cannot be simulated with more than 8 GPUs. As mentioned before, a minimum width of $6h$ is assigned to the subdomain of each process, so the maximum number of GPUs is restricted. The simulation of 12M particles using the 16 GPUs of the system #2 provides an efficiency of 81%. The results for the weak scaling are 96.8% simulating 4M particles per GPU and higher than 99.9% when 8M per GPU are performed.

Finally, for the system #3 using a maximum of 128 Tesla M2090, an efficiency of 97.4% is achieved simulating 4M per GPU and higher than 99.9% with 8M per GPU. Note that the highest execution simulated with this system simulates 1,024M (128 GPUs x 8M) to study the weak scaling.

Values of efficiency higher than 99.9% are obtained since the testcase2 does not scale perfectly. In spite of the efforts to choose a case where the execution time

per number of particles was the same for different sizes, this was not possible due to different factors such as the ratio between fluid and boundary particles. Thus, the execution time of one step per million particles is slightly higher with the case of 8M than in the cases with more than 32M. Therefore, when using the case of 8M as reference to compute the results of weak scaling in comparison to bigger cases, the efficiency is slightly higher than the expected. Figure 6-14 shows the percentage of time dedicated to tasks exclusive of the multi-GPU simulations, which represent the overcost compared to single-GPU. It can be observed how these tasks take less than 1.9% when simulating 8M/GPU with 128 GPUs and increases to 3% and 9% with 4M/GPU and 1M/GPU respectively.

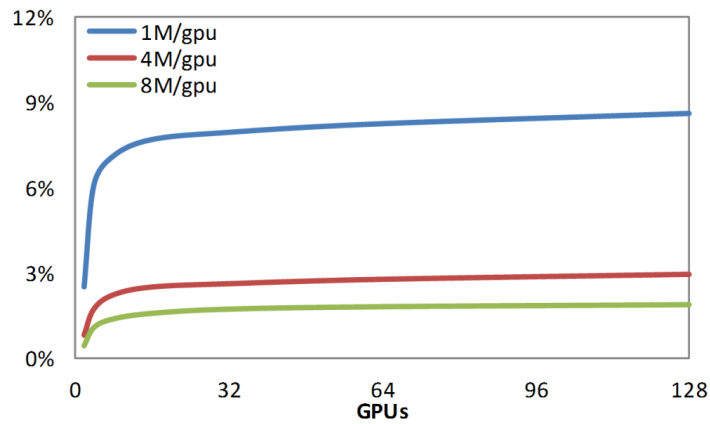


Figure 6-14. Percentage of time dedicated to tasks exclusive of the multi-GPU executions using the system #3.

6.2.5 Bottlenecks: Loss of efficiency

This section discusses the origin of the loss of efficiency when the number of particles per GPU is low. The MPI implementation requires tasks that represent an extra overhead thereby reducing efficiency. Synchronisation tasks and communication between processes are unavoidable and their cost increases with the number of processes. The only way to minimise their impact is to increase the number of particles per GPU. In the previous section, it was shown how efficiency about 99.9% is achieved by simulating 8M particles per GPU, but the efficiency drops significantly when simulating 1M per GPU or less. The main causes of this loss of efficiency can be analysed using Figure 6-14 which shows the percentage of computational time dedicated to the synchronisation tasks (*SynchroDt*), reception and transmission of the *halos* (*RecvHalo* and *SendHalo*) and reception and submission of the particles that have moved among domains (*RecvDisplaced* and *SendDisplaced*). Results of the simulations of 16M in 8, 12

and 16 GPUs are shown in the left panel, while results of the simulation of 8M, 16M and 24M in 16 GPUs are shown in the right panel. These plots reflect how the synchronisation is the first cause of loss of efficiency and the second one is the data exchange of the *halos*. Hence, this loss of efficiency increases with the number of GPUs (left panel of Figure 6-15) but decreases with the number of particles (right panel of Figure 6-15), so the loss of efficiency increases by reducing the number of particles per GPU.

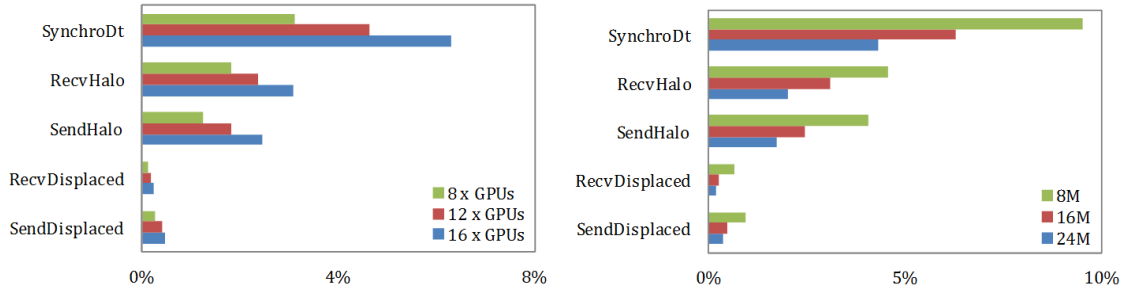


Figure 6-15. Percentage of the computational time dedicated to specific MPI tasks simulating 16M particles using different number of Tesla M2050 GPUs (left) and simulating different number of particles with 16 Tesla M2050 (right).

The time devoted to communications between the devices is significant since the overlapping between computation and data exchange between the MPI processes is not perfect. This data exchange implies four steps: (i) transfer from GPU to CPU, (ii) data submission through MPI, (iii) data reception and, (iv) transfer from CPU to GPU. Therefore, new improvements have been applied to reduce these times of communication. Transfers CPU \leftrightarrow GPU can be reduced to the half using pinned memory. Thus, asynchronous transfers can be used to overlap these times with other tasks of computation. The use of an intermediate buffer, employed in the submission and reception with MPI, can be removed so time of communication and required CPU memory are reduced. The use of streams of CUDA and the asynchronous memory transfers improve the overlap between computation and communication. Using these improvements, the force computation of the particles of each subdomain (very expensive in time) overlaps with the entire process of sending and receiving the two halos. Before this improvement, the overlap with the computation of these forces only occurred with the MPI reception of the first halo, while receiving the second halo overlapped with the force computation of particles of the first halo (no expensive in time).

Figure 6-16 shows the percentage of total time dedicated only to tasks exclusive of the multi-GPU simulations such as synchronisation to compute new time step,

data communication and balancing operations, which represent the overcost comparing to single-GPU. It can be observed how the latest improvements have reduced this percentage to the half for different number of GPUs and different number of particles.

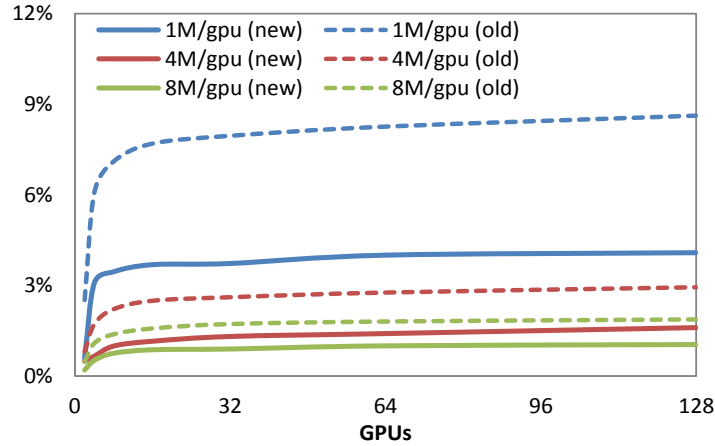


Figure 6-16. Percentage of time dedicated to tasks exclusive of the multi-GPU executions including the latest improvements (using the system #2).

6.2.6 Memory requirements

At the beginning, it was mentioned that the use of GPUs is an attractive alternative to accelerate the SPH simulations, but the limited GPU memory can be a serious drawback for very large cases. Therefore, one of the objectives of this multi-GPU implementation is to eliminate this limitation, so it is necessary to ensure an efficient use of memory for all the GPUs used in the simulation.

In the DualSPHysics version without MPI, all the required memory is allocated at the beginning of the execution since the maximum number of cells and particles is known *a priori*. However, in the multi-GPU version, this information is unknown for each GPU and the number of particles and cells allocated in each GPU change during the simulation. To solve this problem, the amount of memory needed for the particles and cells and an extra 10% is allocated initially, and only when this memory is no longer enough, a new allocation is performed. The maximum number of particles that can be simulated with this multi-GPU implementation for the testcase2 is about 7.14 million particles per GB of memory using single precision. As it can be observed in Figure 6-17, a maximum of 40M can be simulated with the 4 GPUs of the system #1a, 300M with the 16 GPUs of the system #2 and more than 1,370M with 32 GPUs of system #3.

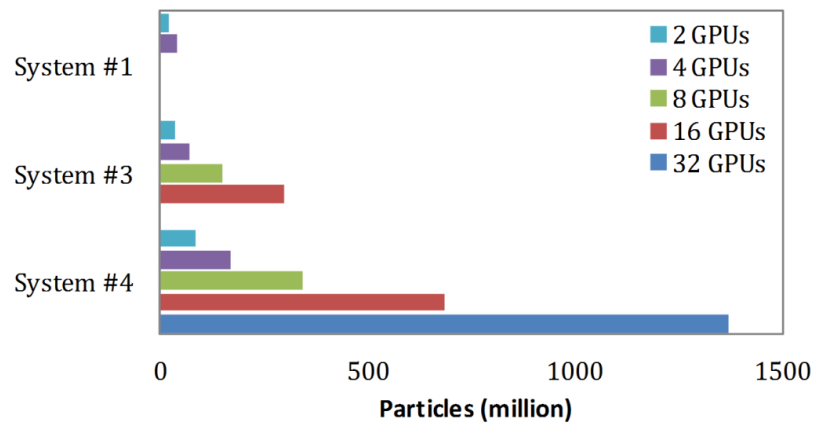


Figure 6-17. Maximum number of particles that can be simulated for the testcase2 with the systems #1a, #2 and #3.

6.3 APPLICABILITY TO REALISTIC PROBLEMS

As mentioned above, one of the main objectives of the multi-GPU code DualSPHysics is simulating real-life applications that require high resolution over a large domain. Thus, once the different algorithms have been described and their efficiency and main drawbacks have been discussed, the code is now applied to perform a huge simulation with more than 10^9 particles. This application consists of the interaction of a large wave with an oil rig using realistic dimensions and simulating 12 seconds of physical time. The fluid domain is 170m x 114m x 68m and the dimensions of the platform can be seen in Figure 6-18. The initial inter-particle distance is 6 cm which implies a simulation of 1,015,896,172 particles (1,004,375,142 fluid particles). This real application has been chosen since a huge number of particles is required to represent with very high resolution the smallest spatial scales in some objects of the oil platform (on the order of centimeters) and also need to describe properly the propagation of large waves (with wavelengths on the order of one hundred meters).

The simulation was carried out using 64 GPUs Tesla M2090 of the hardware system #3. Different instants of the simulation can see in Figure 6-19. A total number of 237,065 steps have been carried out in 79.1 hours. Data were saved every 0.04 seconds of physical time, which represents more than 8980 GB of output data.

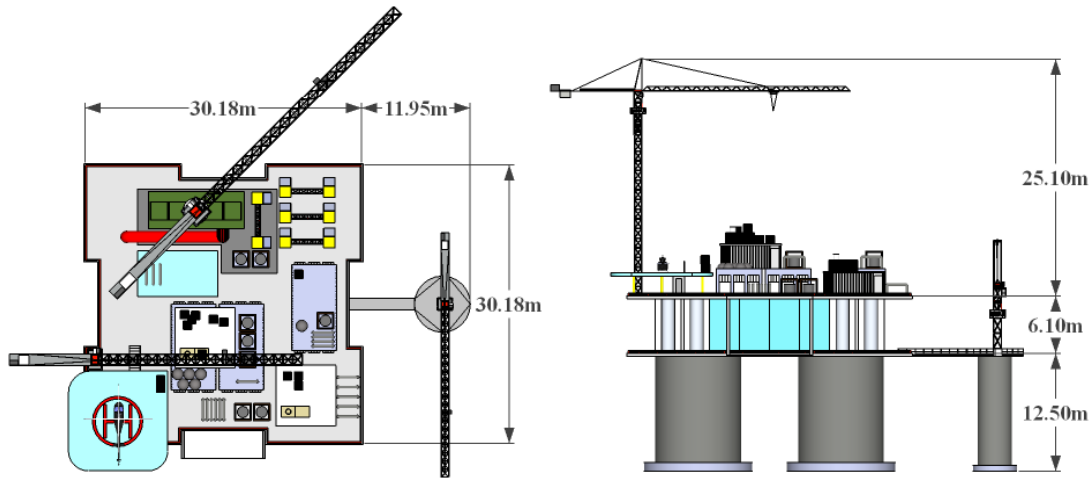


Figure 6-18. Realistic dimensions of the oil rig simulated in the application.

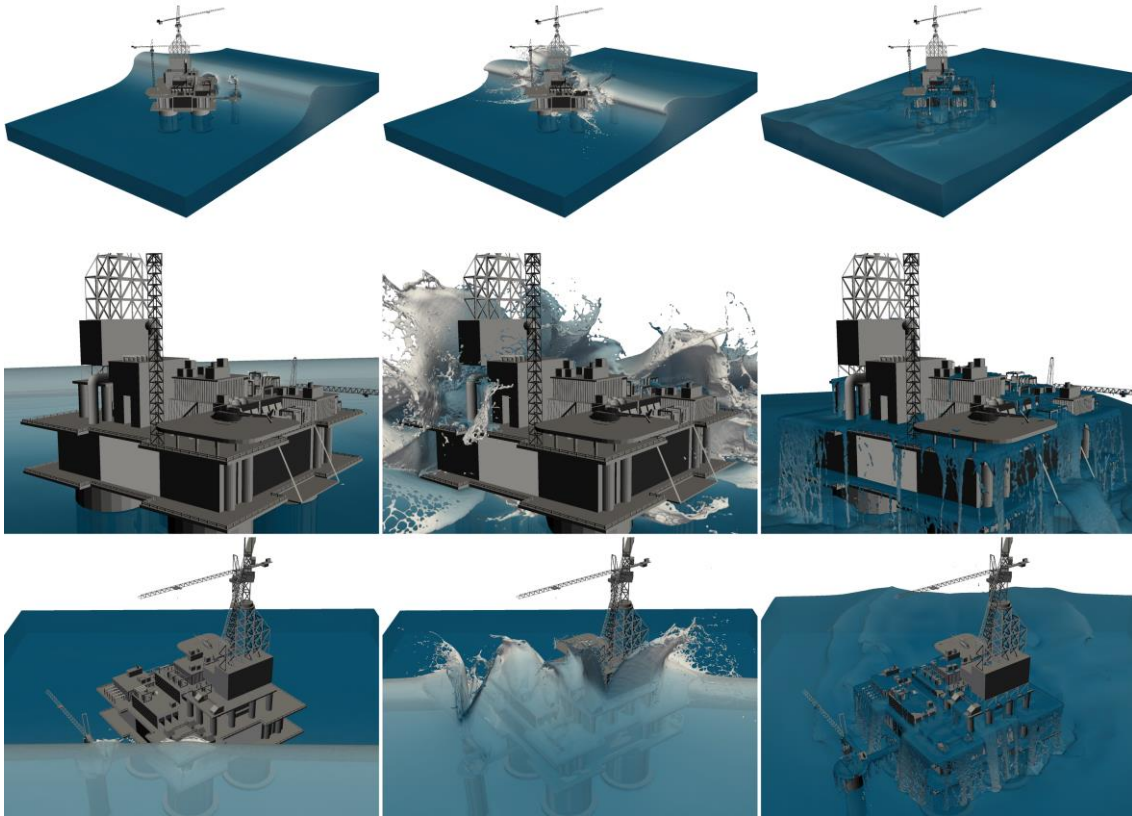


Figure 6-19. Different instants (2.2s, 3.2s and 10s) of the simulation of a large wave interacting with an oil rig using more than 109 particles.

In summary, the efficiency and performance of the new MPI-CUDA implementation of DualSPHysics were presented and analysed in this chapter. The main contributions can be summarised as follows:

- A dynamic load balancing is implemented to distribute work load across the multiple processes to achieve optimal resource utilization and minimise response time.
- Overlapping between data communications and computations tasks is introduced to balance latency and to reduce computational times.
- The proposed multi-GPU code can be executed on different GPUs with identical specifications or old and new cards can be exploited together. Thus, the heterogeneous version allows a more efficient use of different machines with different GPU cards.
- The scalability is analysed in terms of strong and weak scaling, indicating how the runtime varies with the number of processes for a fixed total problem size and how varies with the number of processes for a fixed problem size per processor.
- The simulation of billions particles is possible in medium-size clusters of GPUs.

7. DOUBLE PRECISION

The parallel computing power of Graphics Processing Units (GPUs) has led to an important increase in the size of the simulations but problems of precision can appear when simulating large domains with high resolution.

The goal of this section is to address the problem of the lack of precision and to develop the best solutions increasing the precision but keeping the current efficiency of the GPU codes. Single precision has been used in most of the cases presented in this work, showing to be sufficiently accurate. However, single precision is not enough for some special cases. The GPU implementation of double precision allows simulating real problems where single precision is not enough. This is especially well suited for problems where very different spatial scales are involved.

7.1 THE PROBLEM OF PRECISION

The problems of precision mainly appear when the domain is huge in comparison to the distance of interaction between particles. In Figure 7-1, a testbed is presented where the length of the domain ($L=18\text{m}$) is much higher than the initial depth of the fluid ($D=0.18\text{m}$) and huge comparing with inter-particle distance ($dp=0.01\text{m}$). The difference between the maximum and minimum spatial scale is bigger than three orders of magnitude in this case.

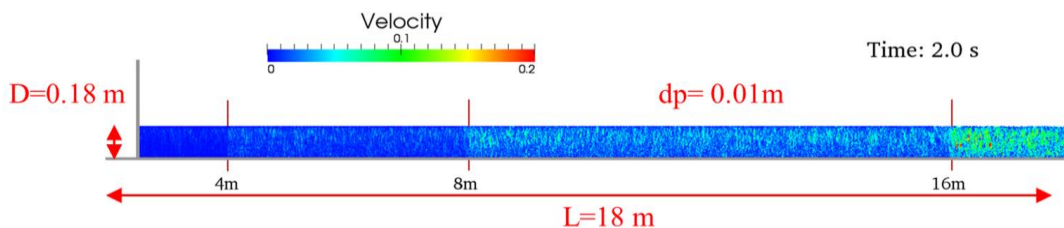


Figure 7-1. Testbed to study problems of precision.

The origin of the problem comes from the use of single precision for the variables to compute and store the position of the particles. The format of real data in single precision has a size of 32 bits; 1 bit for the sign, 8 bits for the exponent and the remaining 23 bits for the mantissa. This allows representing values from $1.175494351 \times 10^{-38}$ a $3.402823466 \times 10^{38}$. Thus, the mantissa has a precision of 23 bits which in decimal representation means 7 digits. A more detailed description of floating-point encodings and functionality can be found in [IEEE 754 Standard]. Therefore, real numbers are stored in a finite representation and their value has to be rounded. This rounding error is a characteristic feature of floating-point computation. This is usually not a problem, but a fatal error can appear when two numbers of very different magnitude are operated. More information about rounding error can be found in [Goldberg 1991] and related to GPU computation in [Whitehead and Fit-Florea 2011].

The use of single precision for the variables of the position of the particles presents problems in different computations:

- a) When two particles (a and b) are close to zero ($0.xxxxxx$), a precision of 7 decimal places is achieved when computing the distance between particles ($r_{ab}=r_a-r_b$) to obtain the value of the kernel (W_{ab}). But when the same two particles are located in the position $1000.xxx$ only a precision of 3 decimal places is achieved when computing r_{ab} .
- b) The same problem appears when updating the new position of the particles at the next time step $r_a(t+dt)$ adding a very small value to $0.xxxxxx$ (more precision) or $1000.xxx$ (lack of precision).

The effect of lack of precision in the position of the particles is shown in Figure 7-2 considering the numerical tank addressed in Figure 7-1. The simulation consists of a 2D numerical tank with a piston-like wavemaker in the left edge. The physical time is 25 seconds and the piston starts to move at 4.5s.

Difference among velocity values of the particles according to their position can be observed in the first frame (time=2.0s). This difference is marked at $x>8m$ and $x>16m$ since the internal representation of values is binary and therefore the precision jumps occur in powers of 2. In time, that inaccuracy (registered in $x>2^4$) affects the rest of the domain ($x>2^3$ & $x>2^2$). Thus, at time=5.0s, particles with $x>16m$ are being excluded from the simulation (black points) since their values of density are assumed as not valid. At time=10s, the excluded particles

appear starting from 8m and finally, at time=25s, the 72% of the fluid particles have been removed from the simulation.

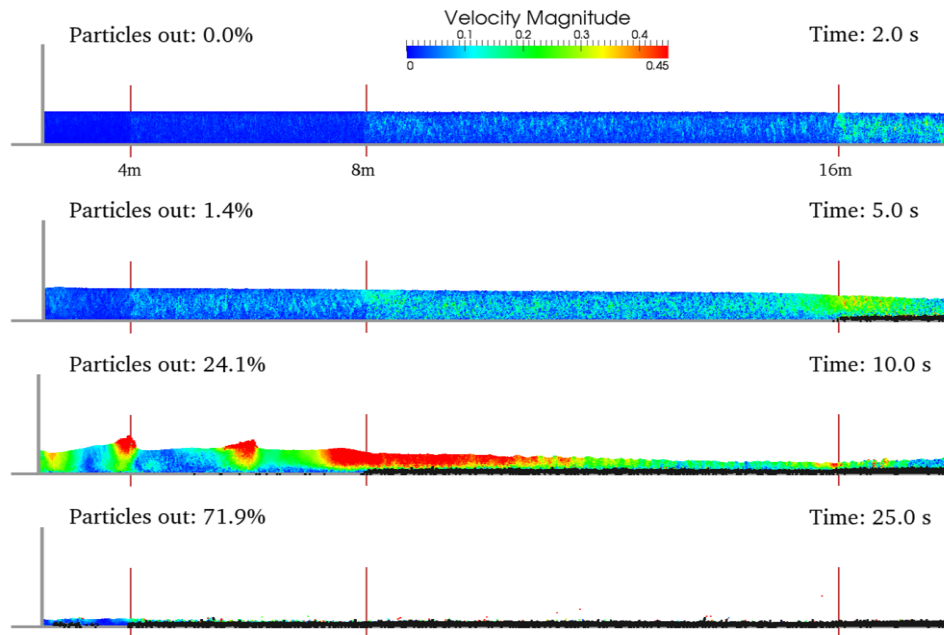


Figure 7-2. Different instants of the simulation of the testbed.

As mentioned above, the internal representation of values is binary and therefore the precision jumps occur in powers of 2. The distance between two neighbouring particles was computed using double and single precision and the difference between these approaches is represented for the different positions of the two interacting particles in Figure 7-3. For each position, this calculation is carried out by changing the distance between particles from 0 to $2h$ (kernel domain). The blue line represents the maximum error and red line is the mean error. It can be observed how this difference is higher when the positions of both particles are farther from the origin.

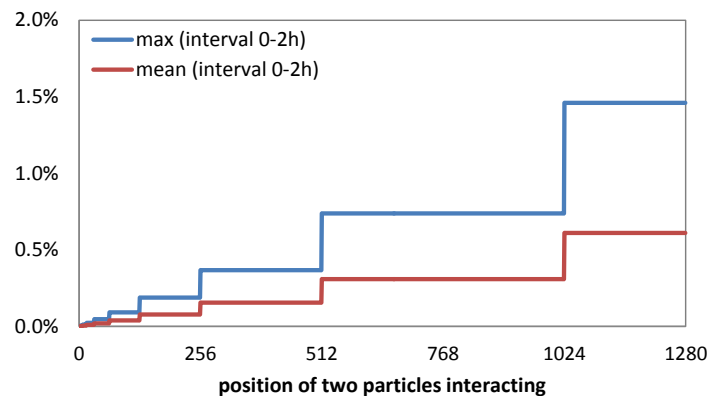


Figure 7-3. Relative error in the distance between two particles interacting using double and single precision for different particle positions.

7.2 SOLUTIONS USING DOUBLE PRECISION

The trivial solution to increase the accuracy is to increase the number of digits used to store the position of the particles, which can reduce significantly the performance. Different approaches have been considered.

7.2.1 Solution FullDouble

The solution to obtain correct results is to use double precision for all variables of the system and perform all computations (creating the neighbour list, computing forces and updating variables) in double precision. The format of data in double precision has a size of 64 bits (8 bytes), so 15 decimal digits. However this approach presents several limitations. The first one is that the executions would be over 7 times slower than using single precision. In terms of GPU implementation, the loss of performance is due to the increase in the number of registers in the CUDA kernel of particle interaction which implies a reduction in the occupancy of the GPU, so that, a reduction in the GPU performance. On the other hand, the capacity of calculation in single and double precision of the GPUs is not balanced. Depending on the model of the GPU, the computation capability in double precision can be several times slower. Another drawback is the loss of efficiency in multi-GPU since there will be more data to be exchanged and overlapping computation-communication is never perfect.

7.2.2 Solution PosDouble

The first feasible and affordable option proposed in this work is implementing only the variable position r_a in double precision since the lack of precision in the simulations seems to come from these variable. Therefore, the size of position ($pos.x$, $pos.y$, $pos.z$) changes from 3x4 bytes to 3x8 bytes. The creation of the neighbour list is performed using the position of the particle in double precision, force computation where distance between particles needs to be computed is also performed in double precision and the variables of position are updated using double precision, too. This approach is still slow (but not as much as *FullDouble*) and implies a higher memory usage on GPU.

7.2.3 Solution PosCell

An alternative is maintaining single precision for position, but instead of storing the real position of the particle r_a , the relative position to the cell the particle

belongs to is stored. Thus, one advantage is that the size of the position changes from 3x4 bytes to 4x4 bytes (*relative pos.x, pos.y, pos.z + cell*). This also implies that the value of the position (*pos.x, pos.y, pos.z*) is no longer higher than the interaction distance, independently on the dimension of the case to be simulated. The use of double precision is not needed in the force computation with this approach, which is remarkable since computing forces takes more than 90% of the total time in the GPU executions (even higher in CPU executions as shown in Figure 5-14). Updating position is still performed using double precision, however this task takes a minimal percentage of the total execution time. The disadvantage is the complexity of the code.

The benefits of using the *PosCell* approach are observed in Figure 7-4. It can be noticed how the problems shown in Figure 7-2 are now solved. After 25 seconds of simulation, only 0.3% of fluid particles were removed from the simulation. In fact, this low percentage of particles is removed due to instabilities created by the wavemaker and not due to precision issues.

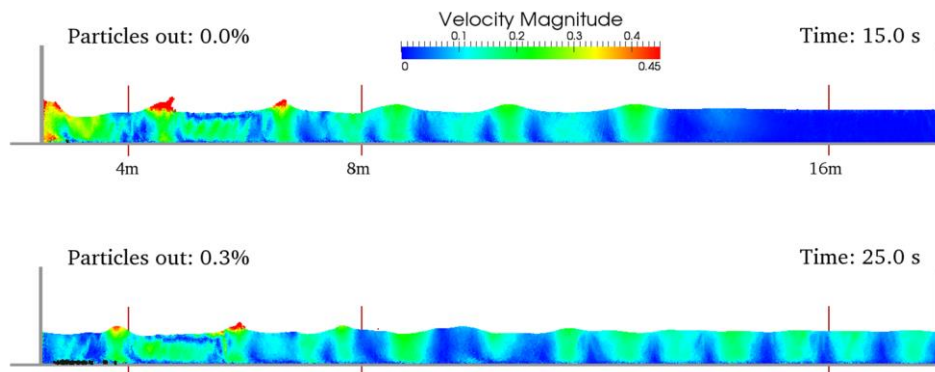


Figure 7-4. Different instants of the previous simulation improving precision in the position of the particles.

Figure 7-5 shows the error in the position of the particles (of the testbed) for different distances from zero. The error is represented for the different approaches; position in single precision (*PosSimple*), position in double precision (*PosDouble*) and position in single precision plus the relative position to the cell (*PosCell*). This error is computed as the difference between the value of the position using each of the mentioned approaches and the value of the position using *FullDouble* (full implementation in double precision). And the resulting difference is expressed as percentage of the interaction distance. The error using *PosDouble* and *PosCell* is constant (values in left axis of the figure) even although the initial position of the computational domain has been shifted in more than 8,000 meters from zero. The error achieved when using *PosSimple*

(values right axis) is several orders of magnitude higher and increases with the distance to zero.

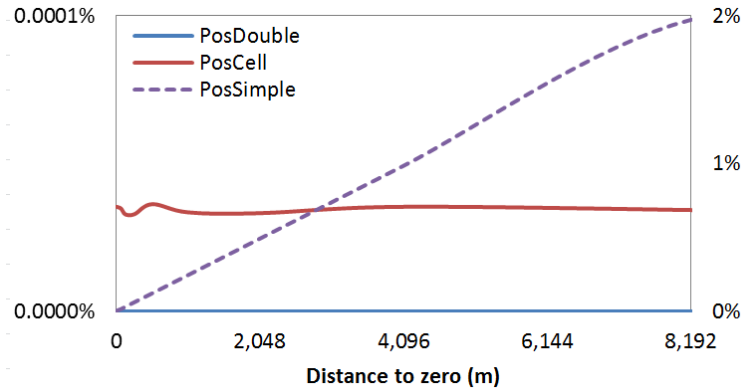


Figure 7-5. Relative error in the position of the particles for different distances to zero and using different approaches.

7.2.4 Solution PosDoubleFast

It is basically a modification of *PosDouble*. The only difference lies in the force computation stage where before computing the distance between particles $r_{ab}=r_a-r_b$, the variables r_a and r_b which are stored in double precision in *PosDouble* are converted here to single precision and the rest of computations during particle interactions are performed in single precision. The approach presents the same benefits solving the issue of precision as proven by *PosDouble* and *PosCell* (Figure 7-4), problems will only appear for domains with lengths higher than 2 km using fine resolution (problems beyond scope with SPH). However this approach will present important advantages in terms of performance since the force computation stage is the most time consuming step in the SPH execution and we can avoid the use of double precision during the computing forces with this new implementation.

The main features of the different approaches implemented to solve the precision issue are presented in Table 7-1.

Table 7-1. Double precision implementations

	PosDouble	PosDoubleFast	PosCell
Position Variable	3 arrays in DOUBLE precision (pos.x, pos.y, pos.z)	3 arrays in DOUBLE precision (pos.x, pos.y, pos.z)	4 arrays in SINGLE precision (relative pos.x, pos.y, pos.z + cell)

	PosDouble	PosDoubleFast	PosCell
Neighbour list	Creating cells list in DOUBLE precision	Creating cells list in DOUBLE precision	Creating cells list in SINGLE precision
Force computation	Computing r_{ab} (to use W_{ab}) with r_a and r_b in DOUBLE precision but stored in SINGLE precision	Computing r_{ab} (to use W_{ab}) with r_a and r_b in SINGLE precision	Computing r_{ab} (to use W_{ab}) with r_a and r_b in SINGLE precision
System update	Updating $r_a(t+dt)$ in DOUBLE precision	Updating $r_a(t+dt)$ in DOUBLE precision	Updating $r_a(t+dt)$ in SINGLE precision

7.3 PERFORMANCE

The implementations *PosDouble* and *PosCell* solve the lack of precision but also imply a high cost in the execution time. Here the loss of performance is analysed and the computational runtime of both approaches are compared against the single precision implementation. The 3D dam-break shown in Figure 6-7 is also used here as testbed, where 4 million particles are simulated to perform 0.6 seconds of physical time. Figure 7-6 represents the loss of efficiency of *PosDouble* and *PosCell* comparing to the single precision implementation for different GPU models.

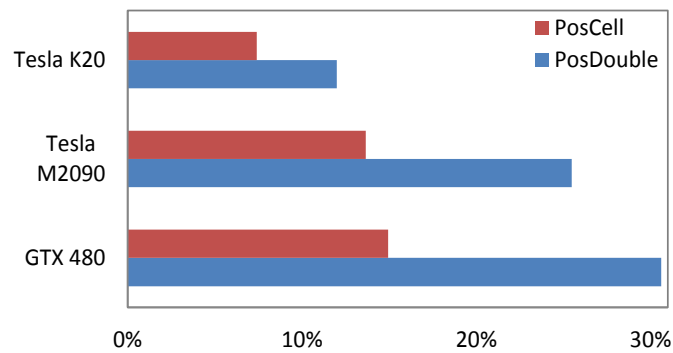


Figure 7-6. Loss of efficiency compared with simple precision simulations using a 3D dam-break with 4M particles.

The loss of efficiency depends on the GPU model. Using *PosDouble*, a loss of 20% is registered with the Tesla K20 (30% with GTX 480) and using *PosCell*, less than 8% in Tesla K20 (less than 15% in GTX 480).

The approach *PosCell* was discarded despite being much faster than *PosDouble*, since the complexity of the code increases significantly. This is a key factor since the code is developed to be latter released as open source for the whole scientific community.

The latest option *PosDoubleFast* (using single precision in the force computation) allows to obtain the same results without loss of performance. That is the reason why this approach was not shown in Figure 7-6. In fact, there is no loss of performance even for particles whose position is moved more than 1km from the origin.

There is no loss of performance using *PosDoubleFast* due to different reasons:

- a) Force computation represents the 92% (94%) of the execution time using Tesla K20 (GTX 480). Therefore, using the variables of position in single precision when computing forces, the impact on the total execution time is very limited (compared with the original version in single precision), although double precision is used in other parts of the code.
- b) The rest of the SPH execution, where variables of position are used in double precision, can also lead to a loss of performance. However, the task of determining the cell where the particle belongs to during the creation the neighbour list and updating the new value of position in the system update stage are only the 0.7% (0.6%) and 1.1% (1.0%) of the total execution time using Tesla K20 (GTX 480). Thus, there is no impact on runtime reduction compared with single precision implementation.
- c) The use of double precision implies an increase in the number of registers that are used in CUDA kernels, which can give rise to a loss of occupancy of the GPU, so that, the performance decreases. Nevertheless, this only occurs for high number of registers. When using double precision in the position, the number of registers increases in 5. This increase in the force computation stage means using 53 registers instead of 48 with a loss of occupancy of 11%. However, there is no loss of occupancy in the system update stage, where 26 registers are used instead of 21. Figure 7-7 shows the occupancy using 256 threads according to the number of registers to better understand this last point.

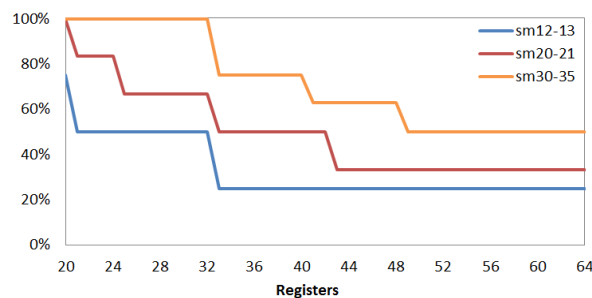


Figure 7-7. Percentage of occupancy according to the number of registers and compute capability of GPU.

- d) The use of double precision also implies an increase in the volume of data to be read/written in memory. However, this increase of the volume of data does not represent a loss of performance in the CUDA kernels used during the neighbour list creation and system update since the memory accesses are coalescent in those kernels and there is not divergence. In the case of CUDA kernels for computing forces among particles, the increase of data to be loaded in the memory gives rise to a significant loss of performance since these kernels present problems of coalescence and divergence.

8. CONCLUSIONS AND FUTURE WORK

8.1 CONCLUSIONS

SPH is an ideal technique to simulate free surface flows, in particular violent collisions between water and structures. Its range of application is very wide, including sloshing and flooding event; design of coastal defenses, dams, devices to generate renewable energies... Actually, the technique can be used for engineering purposes in those problems involving the complex interaction between water and structures. In general, all these problems involve large domains that should be solved with fine resolution, which makes the model expensive in terms of computational requirements. This is the reason why codes should be optimized and accelerated.

The main goal of this work was to develop an optimized version of the open-source code DualSPHysics, which can be used both on CPUs and GPUs. DualSPHysics has been designed to be run on multi-core CPUs, which is a relatively common resource, but also on GPUs. The GPU technology has experienced a rapid development during the last few years and constitutes a fast and cheap alternative to classical computation on CPUs. Nevertheless, a single GPU is not enough to run large domains due to memory requirements. Thus, a multi-GPU version of the code has also been developed. In addition, pre-processing and post-processing tools have been developed to take advantage of DualSPHysics capabilities.

The main findings of this research are summarised in the following subsections.

8.1.1 Neighbour List

SPH software frameworks (such as DualSPHysics) can be split into three main steps; (i) generation of a neighbour list, (ii) computation of forces between

particles by solving momentum and continuity equations and (iii) integrating in time to update all the physical properties of the particles in the system. Running a simulation therefore means executing these steps in an iterative manner. The step devoted to compute forces consumes more than 90% of the execution time, whereby it is the most important part to be accelerated. However, its implementation and performance depends greatly on the previous step (neighbour list generation) therefore a study about different neighbour list approaches was carried out. The use of Cell-linked list and Verlet list with several variations was compared, being the Cell-linked list chosen to be implemented since it provides the best balance between performance and usage of memory.

8.1.2 CPU Acceleration

Four optimizations are implemented for the CPU code in DualSPHysics. The first one applies symmetry in particle interactions, the second one divides the domain into smaller cells, the third one uses SSE instruction and the fourth one uses OpenMP to implement multi-core executions. Three different approaches of the multi-core implementation are presented. The most efficient version uses the dynamic scheduler of OpenMP to achieve the load dynamic balancing and applies symmetry to particle interaction. Thus, the most efficient OpenMP implementation outperforms the single-core by 4.6 using the available 8 logical cores provided by the CPU hardware used in this study.

8.1.3 GPU Acceleration

CUDA is used to exploit the huge parallel power of present-day Graphics Processing Units for general purpose applications such as DualSPHysics. However, an efficient and full use of the capabilities of the GPUs is not straightforward.

Several optimizations are presented for the GPU implementations; maximization of occupancy to hide memory latency, reduction of global memory accesses to avoid non-coalesced memory accesses, simplification of the neighbour search, optimization of the interaction kernel and division of the domain into smaller cells to reduce code divergence. The optimized GPU version of the code outperforms the GPU implementation without optimizations by a factor on the order of 1.65 using a GTX 480 (Fermi architecture) and 2.15 using a Tesla 1060. In general, the designing improvements included in the new Fermi GPUs make

these cards less sensitive to the programming task. The GPU parallel computing developed here can accelerate serial SPH codes with a speedup of 56.2x when using the Fermi card. Finally, the speedup of the GPU over a multi-core CPU is 12.5x when using a multi-threaded approach.

In addition, an evaluation of performance using the latest GPUs is also included. Thus, the new GPUs with Kepler architecture, GTX 680 and Tesla K20 achieved a speedup of one hundred over single core CPU. This speedup rises to 148.8x using a GPU GTX Titan.

8.1.4 Multi-GPU Acceleration

The multi-GPU approach includes CUDA and MPI programming languages to combine the parallel performance of several GPUs in a host machine or in multiple machines connected by a network.

Dynamic load balancing was implemented to distribute work load across the multiple processes to achieve optimal resource utilization and minimise response time. It enables the adaptation of the code to the features of homogeneous and heterogeneous clusters achieving the best performance.

The multi-GPU implementation has shown a high efficiency using a significant number of GPUs. Thus, using 128 GPUs of the Barcelona Supercomputing Center, efficiencies of 85.9%, 97.4% and close to 100% have been achieved simulating 1M/GPU, 4M/GPU and 8M/GPU respectively.

The possibility of combining the resources of several GPUs and the efficient use of the memory enables simulations with a huge number of particles. For example, 40M particles can be simulated with 4 GPUs GTX 480, more than 300M with 16 GPUs Tesla M2050 and more than 2000M with 64 GPUs Tesla M2090.

To show the capabilities of the code, a realistic interaction of a large wave with an oil rig using more than 10^9 particles have been carried out. A total number of 237,065 steps have been carried out in 79.1 hours using 64 GPUs M2090.

8.1.5 Issue of precision

Problems of precision in DualSPHysics can appear in simulations involving very large domains at a very high resolution. It has been shown that the source of the problem comes from the lack of precision to represent the position of the particles. Several implementations have been proposed to solve the issue of precision measuring the accuracy of the results and the loss of performance for each approach. Finally, the best solution avoids problems of precision without loss of performance and without increasing significantly the complexity of the code.

8.2 FUTURE WORK

The aim of DualSPHysics is two-fold. Firstly the code is a user-friendly platform designed to encourage other researchers to use the SPH technique to investigate a large number of novel CFD problems. Secondly, the method can be used by industry to simulate real problems that are beyond the scope of classical models.

New features are constantly being integrated into the DualSPHysics code or are planned to be carried out in the near future. Some of them are mentioned here:

- Variable particle resolution [Vacondio et al., 2013b].
- Multiphase cases (gas-soil-water) [Fourtakas et al., 2013; Mokos et al., 2014].
- New boundary conditions [Fourtakas et al., 2014].
- Coupling with the Discrete Element Method (DEM) [Canelas et al., 2014].
- Coupling with the SWASH Wave Propagation Model [Altomare et al., 2014b].
- Coupling with IBER model (<http://iberaula.es/modelo-iber/modelo>).

A. DUALSPHYSICS DOCUMENTATION

A.1 SOURCE FILES

A set of C++ and CUDA files need to be compiled to generate the DualSPHysics binary. Here all the source files are listed, however each file contains more detailed comments describing the SPH formulation and the algorithms. As mentioned before, the same application can be run using either a CPU or GPU implementation; therefore some files are common for the SPH solver while others are specific to CPU or GPU executions. Table A-1 shows a general overview of the different source files integrated in the project.

Table A-1. List of source files of DualSPHysics code.

No SPH	SPH on CPU & GPU	
Functions (.h .cpp) JException (.h .cpp) JFloatingData (.h .cpp) JLog2 (.h .cpp) JObject (.h .cpp) JObjectGpu (.h .cpp) JPartData (.h .cpp) JPtXasInfo (.h .cpp) JSpaceCtes (.h .cpp) JSpaceEParms (.h .cpp) JSpaceParts (.h .cpp) JSpaceProperties (.h .cpp) JRangeFilter (.h .cpp) JTimer.h JTimerCuda.h JVarsAscii (.h .cpp) TypesDef.h JFormatFiles2.h <i>JFormatFiles2.lib / libjformatfiles2.a</i> JSphMotion.h <i>JSphMotion.lib / libjsphmotion.a</i> JXml.h <i>JXml.lib / libjxml.a</i>	main.cpp JCfgRun (.h .cpp) JSph (.h .cpp) JPartsLoad (.h .cpp) JPartsOut (.h .cpp) JSphDtFixed (.h .cpp) JSphVarAcc (.h .cpp) Types.h	
	SPH on CPU	SPH on GPU
	JSphCpu (.h .cpp)	JSphGpu (.h .cpp) JSphGpu_ker (.h .cu)
	JSphCpuSingle (.h .cpp)	JSphGpuSingle (.h .cpp)
	JSphTimersCpu.h	JSphTimersGpu.h
	JCellDivCpu (.h .cpp)	JCellDivGpu (.h .cpp) JCellDivGpu_ker (.h .cu)
	JCellDivCpuSingle (.h .cpp)	JCellDivGpuSingle (.h .cpp) JCellDivGpuSingle_ker (.h .cu)
	JPeriodicCpu (.h .cpp)	JPeriodicGpu (.h .cpp) JPeriodicGpu_ker (.h .cu)
		JGpuArrays (.h .cpp)

The following tables show the goal of each individual file; Table A-2 describes the files not related to the SPH solver; Table A-3 describes the files of the SPH solver common to CPU and GPU implementations; and Table A-4 and Table A-5 describe the files for the specific execution on CPU and GPU, respectively.

Please note that both the C++ and CUDA version of the code contain the same features and options. Most of the source code is common to CPU and GPU (files in Table A-2 and Table A-3).

Table A-2. List of source files of DualSPHysics code not related to the SPH solver.

No SPH FILES	
Functions (.h .cpp)	Declares/implements basic/general functions for the entire application
JException (.h .cpp)	Declares/implements the class that defines exceptions with the information of the class and method
JFloatingData (.h .cpp)	Declares/implements the class that allows reading/writing files with data of floating bodies
JLog2 (.h .cpp)	Declares/implements the class that manages the output of information in the file Run.out and on screen
JObject (.h .cpp)	Declares/implements the class that defines objects with methods that throws exceptions
JObjectGpu (.h .cpp)	Declares/implements the class that defines objects with methods that throws exceptions about tasks in GPU
JPartData (.h .cpp)	Declares/implements the class that allows reading/writing files with data of particles in formats binx2, ascii...
JPtXasInfo (.h .cpp)	Declares/implements the class that returns the number of registers of each CUDA kernel.
JSpaceCtes (.h .cpp)	Declares/implements the class that manages the info of constants from the input XML file
JSpaceEParms (.h .cpp)	Declares/implements the class that manages the info of execution parameters from the input XML file
JSpaceParts (.h .cpp)	Declares/implements the class that manages the info of particles from the input XML file
JSpaceProperties (.h .cpp)	Declares/implements the class that manages the properties assigned to the particles in the XML file
JRangeFilter (.h .cpp)	Declares/implements the class that facilitates filtering values within a list
JTimer.h	Declares the class that defines a class to measure short time intervals
JTimerCuda.h	Declares the class that defines a class to measure short time intervals in GPU using cudaEvent
JVarsAscii (.h .cpp)	Declares/implements the class that reads variables from a text file in ASCII format
TypesDef.h	Declares general types and functions for the entire application
JFormatFiles2.h	Declares the class that provides functions to store particle data in formats VTK, CSV, ASCII
JSpHMotion.h	Declares the class that provides the displacement of moving objects during a time interval
JXml.h	Declares the class that helps to manage the XML document using library TinyXML

Table A-3. List of source files of DualSPHysics code for the SPH execution.

SPH SOLVER	
main.cpp	Main file of the project that executes the code on CPU or GPU
JCfgRun (.h .cpp)	Declares/implements the class that defines the class responsible of collecting the execution parameters by command line
JSph (.h .cpp)	Declares/implements the class that defines all the attributes and functions that CPU and GPU simulations share
JPartsLoad (.h .cpp)	Declares/implements the class that manages the initial load of particle data
JPartsOut (.h .cpp)	Declares/implements the class that stores excluded particles at each instant till writing the output file
JSphDtFixed (.h .cpp)	Declares/implements the class that manages the use of prefixed values of DT loaded from an input file
JSphVarAcc (.h .cpp)	Declares/implements the class that manages the application of external forces to different blocks of particles (with the same MK)
Types.h	Defines specific types for the SPH application

Table A-4. List of source files of DualSPHysics code for the SPH execution on CPU.

SPH SOLVER ONLY FOR CPU EXECUTIONS	
JSphCpu (.h .cpp)	Declares/implements the class that defines the attributes and functions used only in CPU simulations
JSphCpuSingle (.h .cpp)	Declares/implements the class that defines the attributes and functions used only in Single-CPU
JSphTimersCpu.h	Measures time intervals during CPU execution
JCellDivCpu (.h .cpp)	Declares/implements the class responsible of computing the Neighbour List in CPU
JCellDivCpuSingle (.h .cpp)	Declares/implements the class responsible of computing the Neighbour List in Single-CPU
JPeriodicCpu (.h .cpp)	Declares/implements the class that manages the interactions between periodic edges in CPU

Table A-5. List of source files of DualSPHysics code for the SPH execution on GPU.

SPH SOLVER ONLY FOR GPU EXECUTIONS	
JSphGpu (.h .cpp)	Declares/implements the class that defines the attributes and functions used only in GPU simulations
JSphGpu_ker (.h .cu)	Declares/implements functions and CUDA kernels for the particle interaction and system update
JSphGpuSingle (.h .cpp)	Declares/implements the class that defines the attributes and functions used only in Single-GPU
JSphTimersGpu.h	Measures time intervals during GPU execution
JCellDivGpu (.h .cpp)	Declares/implements the class that defines the class responsible of computing the Neighbour List in GPU
JCellDivGpu_ker (.h .cu)	Declares/implements functions and CUDA kernels to compute operations of the Neighbour List
JCellDivGpuSingle (.h .cpp)	Declares/implements the class that defines the class responsible of computing the Neighbour List in Single-GPU
JCellDivGpuSingle_ker (.h .cu)	Declares/implements functions and CUDA kernels to compute operations of the Neighbour List
JPeriodicGpu (.h .cpp)	Declares/implements the class that manages the interactions between periodic edges in GPU
JPeriodicGpu_ker (.h .cu)	Declares/implements functions and CUDA kernels to obtain particles that interact with periodic edges
JGpuArrays (.h .cpp)	Declares/implements the class that manages arrays with memory allocated in GPU

A.2 COMPILATION

The code can be compiled for either CPU or GPU execution. In order to compile the code for CPU execution, only a C++ compiler (for example GNU's g++) is needed with the resultant binary allowing the code to be run on workstations without a CUDA-enabled GPU.

To run DualSPHysics on GPU, an Nvidia CUDA-enabled GPU is needed and the latest version of the GPU driver must be installed. However, to compile the source code, the GPU programming language CUDA and NVCC compiler must be installed on the computer. The CUDA Toolkits can be downloaded directly from Nvidia (<https://developer.nvidia.com/cuda-downloads>). CUDA versions 4.0, 4.1, 4.2, 5.0, and 5.5 have been tested (the same numerical results are obtained with different CUDA versions).

Makefiles can be used to compile the code:

- i) Make `-f Makefile_cpu` only for CPU compilation (files of Table A-5 are not included in the compilation) leading to the binary *DualSPHysicsCPU_linux64*,
- ii) Make `-f Makefile` for a full compilation creating a binary for CPU-GPU and the result of the compilation is the binary *DualSPHysics_linux64*.

The user can modify the compilation options such as the path of the CUDA toolkit directory or the GPU architecture. By default the GPU code is compiled for "sm_12,compute_12" and "sm_20,compute_20" using CUDA v5.0, the log file generated by the compiler is stored in the file *DualSPHysics_ptxasinfo*. For example, any possible error in the compilation of *JSphGpu_ker.cu* can be identified in this *ptxasinfo* file. This file is also parsed by the executable on initial startup in order to perform hardware specific kernel optimisation.

The same code can be compiled for Windows platform and in that sense a file with Microsoft Visual Studio project and libraries for Windows are included.

A.3 FILES AND FORMAT

Different files for the input and the output data are involved in the DualSPHysics execution: .xml, .bi2 and .vtk.

The XML (EXtensible Markup Language) is a textual data format that can easily be read or written using any platform and operating system. It is based on a set of labels (tags) that organise the information and can be loaded or written easily using any standard text or dedicated XML editor. This format is used for input files for the code.

Data stored in text format (ASCII) consumes at least six times more memory than the same data stored in binary format. Values stored in text format in the memory cannot always be recorded accurately due to rounding error introduced by I/O routines and data truncation. Reading and writing data in ASCII is computationally more expensive than using binary (this can be as high as two orders of magnitude). As DualSPHysics allows simulations to be performed with a large number of particles, a binary file format is necessary to avoid these problems. The use of a binary format reduces the stored size of the files and also the time dedicated to generating them. The format used in DualSPHysics is named BINX2 (.bi2), these files contain only the meaningful information of particle properties. Some variables are removed, e.g. the pressure is not stored since it can be calculated starting from the density using the equation of state as a pre-processing step. The value for mass is constant for fluid and boundary particles and so only two values are used instead of an array. The position of fixed boundary particles is only stored in the first file since they remain unchanged throughout the simulation. Data for particles that leave the limits of the domain are stored in an independent file which leads to an additional saving. Hence, the advantages of BINX2 can be summarised as: (i) memory storage reduction, (ii) fast access, (iii) no precision lost and (iv) portability (i.e. to different architectures or different operating systems).

VTK (Visualisation ToolKit) files are used for final visualisation of the results and can either be generated as a pre-processing step or output directly by DualSPHysics instead of the standard BINX format (albeit at the expense of computational overhead). VTK not only supports the particle positions, but also physical quantities that are obtained numerically for the particles involved in the simulations. VTK supports many data types, such as scalar, vector, tensor, texture, and also supports different algorithms such as polygon reduction, mesh smoothing, cutting, contouring and Delaunay triangulation. The VTK file format consists of a header that describes the data and includes any other useful information, the dataset structure with the geometry and topology of the dataset and its attributes. Here VTK files of POLYDATA type with legacy-binary format is used. This format is also easy for read-write operations.

A.4 RUNNING DUALSPHYSICS

The input files to run the DualSPHysics code include one XML file (*Case.xml*) and a binary file (*Case.bi2*). *Case.xml* contains all the parameters of the system configuration and its execution, such as key variables (i.e. smoothing length, reference density, gravity, coefficient to calculate pressure, speed of sound), the number of particles in the system, movement definition of moving boundaries and properties of moving bodies. The binary file *Case.bi2* contains the initial particle data; arrays of position, velocity and density and headers. The output files of DualSPHysics consist of binary format files (by default) with the particle information at different instants of the simulation: *Part0000.bi2*, *Part0001.bi2*, *Part0002.bi2* ..., *PartOut.bi2* with excluded particles and *Run.out* with a brief description of the execution.

Different execution parameters can be changed in the XML file: time stepping algorithm specifying Symplectic or Verlet, choice of kernel function which can be Cubic or Wendland, the value for artificial viscosity or laminar+SPS viscosity treatment, activation of the Shepard density filter and how often it is applied, activation of the delta-SPH correction, the maximum time of simulation and time intervals to save the output data. To run the code, it is also necessary to specify whether the simulation is going to run in CPU or GPU mode, the format of the output files, files that summarise the execution process with the computational time of each individual process. For CPU executions, a multi-core implementation using OpenMP enables executions in parallel using the different cores of the machine. It takes the maximum number of cores of the device by default or users can specify the number used. In addition, the parallel execution with OpenMP can use dynamic or static load balancing.

To run the program, type the command `./DualSPHysics_linux64 Case [options]`, where *Case* is the name of the input files (*Case.xml* and *Case.bi2*). The configuration of the execution is mostly defined in the XML file, but it can be also defined or changed using execution parameters. Furthermore, new options and possibilities for the execution can be imposed using [options] as seen in Table A-6. For example:

```
$dualsphysics $dirout/$name $dirout -svres -cpu
```

enables the simulation on the cpu, where *\$dirout* is the directory with the file *\$name.bi2*

`$dualsphysics $dirout/$name $dirout -svres -gpu`
enables the same simulation on the gpu.

`$dualsphysics $dirout/$name $dirout -svres -gpu -partbegin:69 $dirdata`
restarts the simulation from the time corresponding to files output Part0069.bi2
in \$dirdata directory.

Table A-6. List of execution parameters of DualSPHysics.

PARAMETER	DESCRIPTION
-h	Shows information about parameters
-opt <file>	Loads configuration from a file
-cpu	Execution on Cpu (option by default)
-gpu[:id]	Execution on Gpu and id of the device
-stable	Ensures the same results when repeated a simulation since operations are always carried out in the same order
-ompthreads:<int>	Only for Cpu. Indicates the number of threads by host for parallel execution, it takes the number of cores of the device by default (or using zero value)
-ompdynamic	Only for Cpu. Parallel execution with symmetry in interaction and dynamic load balancing. Not compatible with -stable
-ompstatic	Only for Cpu. Parallel execution with symmetry in interaction and static load balancing
-cellorder:<axis>	Indicates the order of the axis. (xyz/xzy/yxz/yzx/zxy/zyx)
-cellmode:<mode>	Specifies the cell division mode, by default, the fastest mode is chosen h fastest and the most expensive in memory 2h lowest and the least expensive in memory
-symplectic	Symplectic algorithm as time step algorithm
-verlet[:steps]	Verlet algorithm as time step algorithm and number of time steps to switch equations
-cubic	Cubic spline kernel
-wendland	Wendland kernel
-viscoart:<float>	Artificial viscosity [0-1]
-viscolamsp:<float>	Laminar+SPS viscosity [order of 1E-6]
-shepard:steps	Shepard filter and number of steps to be applied
-deltasph:<float>	Constant for DeltaSPH. By default 0.1 and 0 to disable
-sv:[formats,...]	Specifies the output formats: none No files with particle data are generated binx Bynary files (option by default) vtk VTK files ascii ASCII files (PART_xxxx of SPHysics) csv CSV files
-svres:<0/1>	Generates file that summarizes the execution process
-svtimers:<0/1>	Obtains timing for each individual process
-svdomainvtk:<0/1>	Generates VTK file with domain limits
-name <string>	Specifies path and name of the case
-runname <string>	Specifies name for case execution
-dirout <dir>	Specifies the output directory

-partbegin:begin[:first] dir	RESTART option. Specifies the beginning of the simulation starting from a given PART (begin) and located in the directory (dir), (first) indicates the number of the first PART to be generated
-incz:<float>	Allowable increase in Z+ direction. Case domain is fixed as function of the initial particles, however the maximum Z position can be increased with this option in case particles reach higher positions
-rhopout:min:max	Excludes fluid particles out of these density limits
-ftpause:<float>	Time to start floating bodies movement. By default 0
-tmax:<float>	Maximum time of simulation
-tout:<float>	Time between output files
-ptxasfile <file>	Indicates the file with information about the compilation kernels in CUDA to adjust the size of the blocks depending on the needed registers for each kernel (only for gpu). By default, it takes the path and the name of the executable + _ptxasinfo

B. PRE-PROCESSING TOOLS

The process of generating the geometry of an experiment based on particles is not trivial and can give rise to a significant computational cost. Generating the initial configuration of particles for a SPH simulation requires filling volumes of irregular shapes using particles that must be spaced equidistant. Depending on the treatment of the boundary conditions, computation of the normal vectors of the boundary points might be required.

To perform this task, a code named GenCase was developed. GenCase is a tool implemented in C++ that works independently without the need for other design software. This code combines the simplicity of defining the case using basic geometrical shapes with the capacity of including 3D models. Thus, starting from the case description and the 3D external objects, the code is able to generate very complex geometries using millions of particles not only in an easy way but also almost instantaneously.

At its core, GenCase is a drawing application that creates points that will be converted into particles which carry physical quantities (position, velocity, density...). It creates the configuration that will be loaded by the SPH solver as initial condition for the simulation. The central feature of the code is its capability to convert a wide variety of geometrical shapes into their respective particle representation. In fact it is possible to convert any shape that consists of a mesh with edges and faces. The procedure is based on a simple algorithm. GenCase employs a 3D mesh to locate points which represent possible particle positions. The main idea is to build an object by placing particles only at those points which are required to generate the desired geometry.

The input file of GenCase is a XML file. The XML (eXtensible Markup Language) format consists of an extensible meta-programming language that allows a structured representation of data. In order to represent all the

information required to define a case, the best and clearest option is using this format due to its simplicity, generality and usability. The output file is a new XML file and a binary file containing the data of all the particles of the domain. In addition, VTK files with particles or VTK files with the planes of the geometry can be used for visualisation.

B.1 PARTICLE GENERATION

A 3D mesh is used to construct the points that will be used to define the particle positions. The mesh is implemented as a matrix where each element represents a possible point. A label that identifies the point is stored on the elements or positions of matrix. The location of the points is implicit in the given structure of the matrix. These labels allow marking out the different types of points; fluids (fluid), boundaries (bound) or empty points (void). The type “void” is the initial state of all points of the mesh.

The use of the mesh has several advantages. On one hand, all points will be placed maintaining an equidistant distribution independently of how complicated the case geometry is. On the other hand, the performance of read-write tasks is improved. This accelerates the algorithms of creating points but restricts the size of the case, though the maximum number of points that can be created is $2 \cdot 10^9$.

In order to represent 3D objects in a mesh, only the points that compose the shape of the object will be marked. Thus, when a 3D object is drawn, a set of points with a specific label are marked in the mesh. Generally, 3D models are composed of polygons that can be decomposed into triangles. Thus, Figure B-1 illustrates how this algorithm is employed to create a triangle in 2D. Firstly, the points of a mesh are defined covering the desired triangle, then the three lines with the three vertices of the triangle are defined and finally, particles in the available points under the three lines are created. A similar procedure is applied for other shapes such as spheres, ellipsoids, cylinders...

The geometry of the case is defined following absolute measures independently on the inter-particle distance. This allows varying the number of particles by just defining a different distance among particles. The complexity of the object will be better represented if the number of particles is higher. Figure B-2 shows how the detail and the accuracy of the object changes when the inter-particle distance

is modified. For a better visualisation of the figure, particles are represented by cubes.

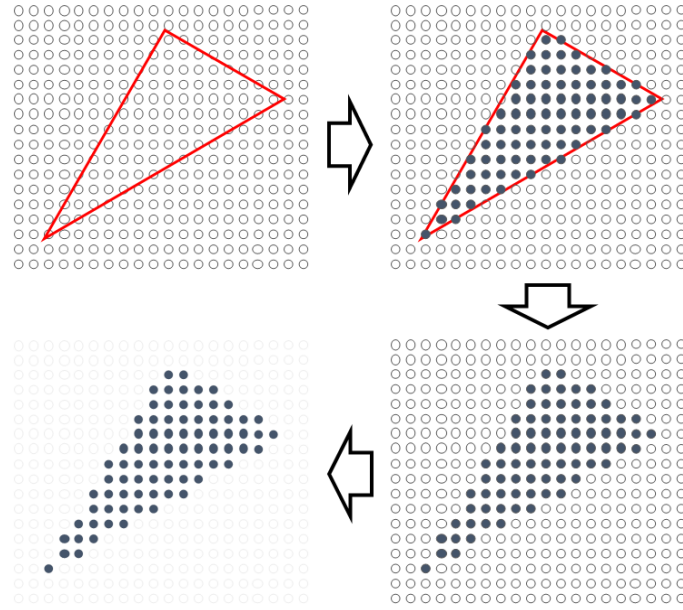


Figure B-1. Generation of a 2D triangle.

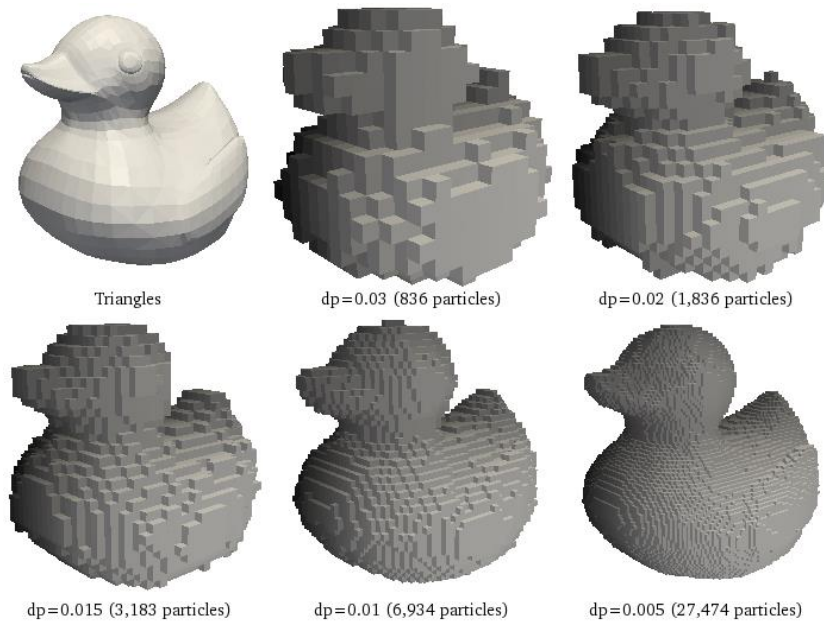


Figure B-2. Discretization accuracy for different number of particles. The absolute measures of the object are $0.39 \times 0.46 \times 0.42$.

As mentioned above, some points are marked in the mesh to draw a 3D object. These points are stored with a label that indicates what type of particles will be created, i.e. fluid particles or boundary particles.

B.1.1 Predefined objects

A wide variety of predefined shapes can be added to the simulation just by setting up some configuration parameters. For instance, a corner and the size are required to create a box, the centre and radius are needed to plot the sphere, two points and radius for the cylinder... Figure B-3 shows some examples.

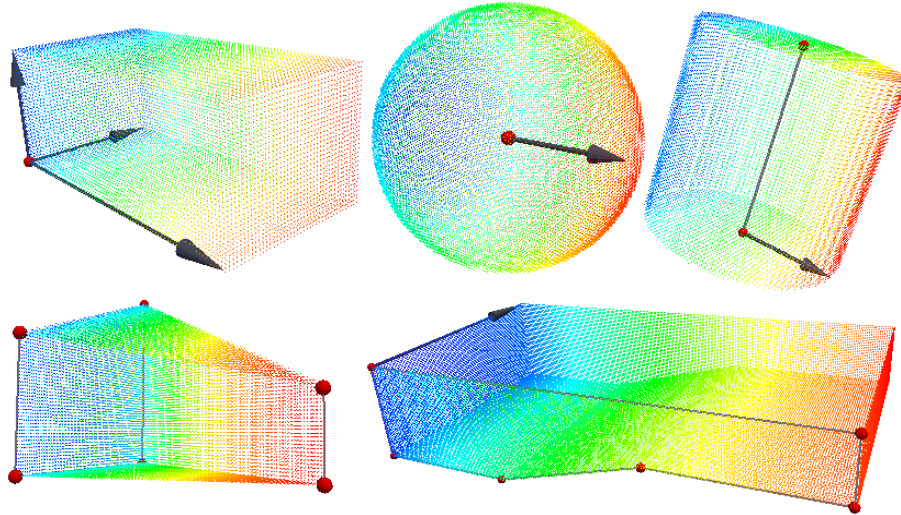


Figure B-3. Some predefined objects: box, sphere, cylinder, prism,...

Particles can be built in different ways starting from a mesh. The “face” mode creates particles along the boundaries of the object, the “solid” mode only uses internal points and the “full” mode creates particles according to the combination of both “face” and “solid”. Furthermore, the “face” mode allows selecting edges to be hidden. Figure B-4 represents a solid ellipsoid, a box without top and front face and a cylinder without covers.

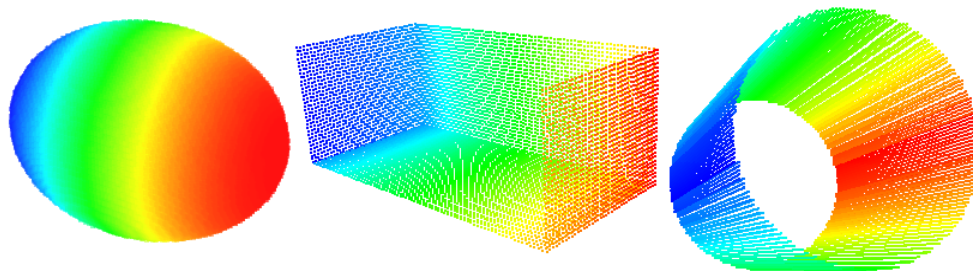


Figure B-4. Basic shapes “solid” and “face”.

B.1.2 External objects

Design software such as AutoCAD, Blender or 3D Studio Max is suggested to be used to generate complex 3D models in an easier way. The model can be then

exported to the formats: STL, PLY or VTK. These formats can then be loaded by GenCase and the geometry is then converted to points and particles. This option allows the use of pre-existing 3D models, available for example on Internet. One such example is the mixer from the Google SketchUp Gallery shown in Figure B-5.

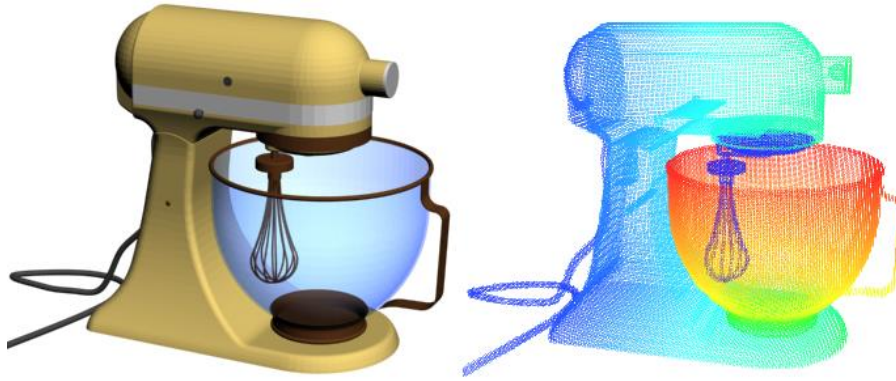


Figure B-5. Mixer: 3D model (left) and point distribution (right).

B.1.3 Filling algorithm

Since SPH is used to study free-surface flow applications, the treatment of boundary conditions is intrinsic to the problem. In case of complex boundaries, a tool to fill areas with fluid particles is required. GenCase is able to perform this task independently of how irregular the shape is. The code is also efficient since it can create configurations that require several million of particles within a few seconds.

GenCase presents several options for the filling of areas and can be adapted to any problem. First, a seed point must be defined, this point is marked with the label or the type of point chosen for the filling (fluid, bound, void). Starting from this seed point the procedure is extended to the surrounding points according to their labels. The algorithm is configured to fill when the surrounding particles fulfil different criteria; when they have the required label or type of point (filling with fluid while points are void) or when they do not have the required label or type (filling with fluid while points are not bound). Finally the area to be filled can be limited defining different shapes (box, prism...).

The procedure of the filling algorithm consists of; (i) identifying the point of the mesh that is closest to the seed point; (ii) if the criteria to mark a new point are fulfilled, the filling algorithm marks the first point at this location, then (iii)

neighbouring points (6 adjacent points) are analysed to check if they fulfil the criteria, if so new points are marked, (iv) the procedure ends when no more points fulfil the criteria or when the positions of the points reach the limits that can be defined jointly with the seed point.

An example of how the filling algorithm works is depicted in Figure B-6. The case consists of a 2D beach with an irregular bottom and two floating objects. The geometry must be filled from the bottom to a given height. The colour of the particles represents the order followed during the filling procedure (from blue to red) starting from the seed point (the large red dot).

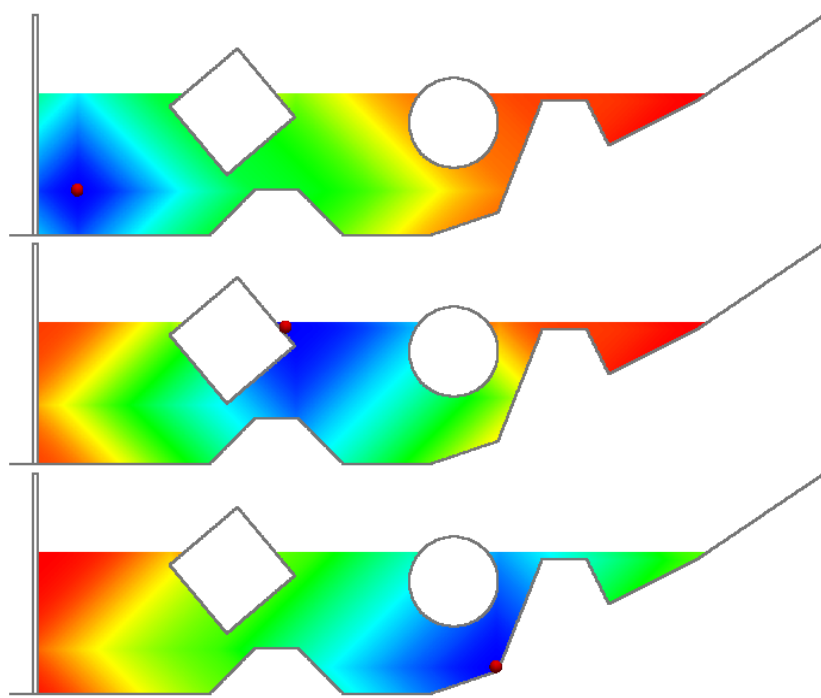


Figure B-6. Filling an irregular beach with fluid.

B.1.4 Other design tools

GenCase presents several options to transform the objects (predefined or external) to make the design of the case easier. The basic transformation operations are shifting, scaling and rotation over an arbitrary axis. Note that all these transformations are cumulative so when one is applied, the following objects and operations will also be affected. A transformation matrix is used and the procedure consists of multiplying this matrix with each vertex of the object. Figure B-7 shows an example of different transformations. Rotation and scaling operations are applied to the vertices of the triangles of a 3D object.

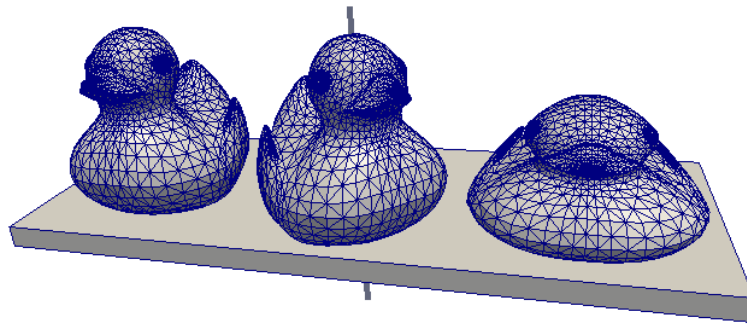


Figure B-7. Example of rotation and scaling of a 3D model.

Different operations such as constructing an object and the transformations can be grouped in lists. This makes it easy to repeat a sequence of operations. An example of how this can be used to create a model starting from a primitive element is demonstrated in Figure B-8.

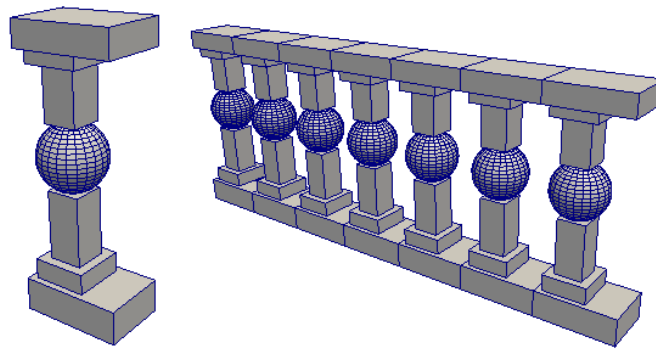


Figure B-8. Creating a balustrade starting from a primitive element.

There is the possibility to merge objects in order to create new ones. When an object is drawn at the same location as a previous one, all the points whose positions coincide will be replaced with the label of the new object. In Figure B-9, a sphere with label void is drawn over a box with label bound.

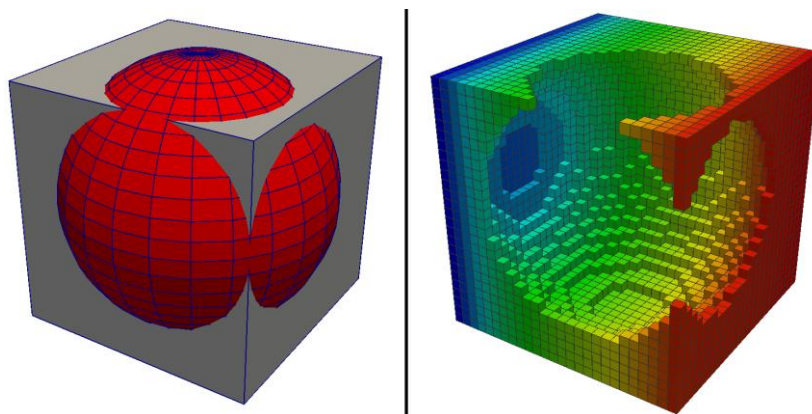


Figure B-9. Merging objects with different label.

B.2 FLOATING OBJECTS

Including floating bodies in SPH simulations can be important for certain applications. GenCase also offers the possibility of using an external 3D model and label the points that formed the object as “floating”. In order to simulate the rigid motion of a floating body, the centre of gravity (GC), the moment of inertia (I) and the mass of the body (M) must be calculated. These properties are easily computed when basic shapes are considered (boxes, spheres ...). However, this task becomes more difficult in case of complex geometries. There are different algorithms to compute these three variables starting from any polyhedron. However, these algorithms cannot be applied when the object consists of an open mesh. Another issue is that a 3D object does not always have homogeneous density and some parts can have higher density than the rest. For example, the front part of a car with the engine is heavier than the part containing the passengers.

GenCase allows setting up the properties for each floating object, but it is also able to obtain the mentioned variables (GC, I, M) based on a point cloud. Thus, the method can compute these magnitudes of any 3D object using its point representation. Defining parts of the object with higher density can be achieved by placing more particles at the desired location. Figure B-10 shows how GC changes due to the distribution of particles.

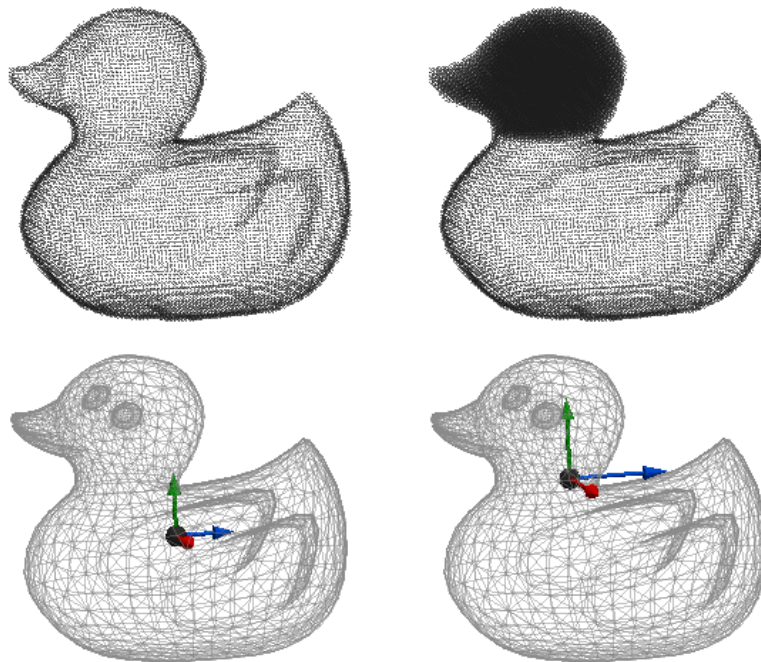


Figure B-10. Gravity center and inertia (lower pannel) computed starting from different particle distributions (upper pannel).

B.3 INITIAL CONDITIONS

Once particles are created based on the marked points of the mesh, the values of different variables and physical quantities must be assigned to each particle: *id*, *position*, *velocity* and *density*. Different options of GenCase can be used to compute the values of these quantities.

The *position* of each particle is calculated by multiplying the position in the mesh with the distance among particles and adding the coordinates of the based point of the mesh. A variable “lattice” can be defined as 1 or 2; the value 2 means that two particles will be generated for each point. Thus, starting from the position, a quarter of the inter-particle distance is subtracted to determine the final position of the first particle and a quarter is added to calculate the position of the second one. Different initial configurations are represented in Figure B-11 with different values of “lattice” to create fluid and boundary particles.

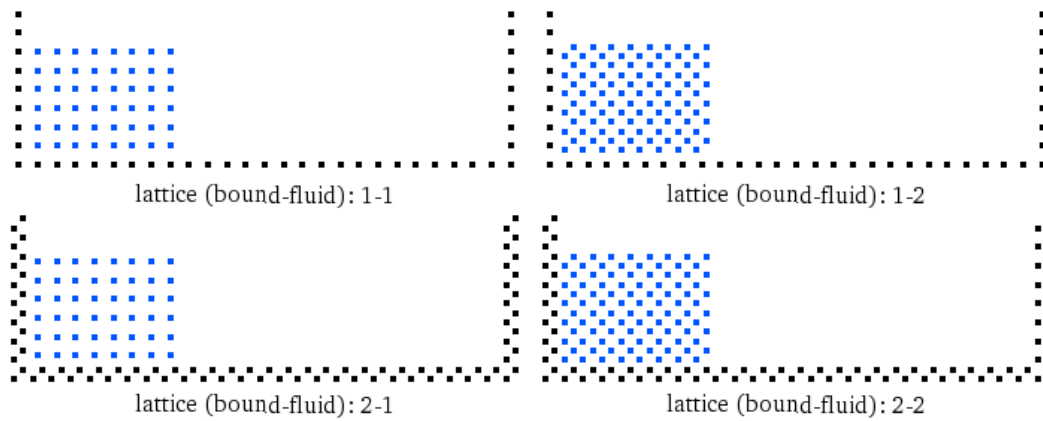


Figure B-11. Different initial configurations depending on the value of lattice for fluid (blue points) and boundary (black points) particles.

The initial values of *velocity* for all particles are zero. However in the case of fluid particles, a different initial velocity can be defined for a subset of particles with the same label. The value of this initial velocity can be the same for all the particles or the velocity profile of solitary wave.

The *density* is computed automatically in the code depending on the depth of each particle in relation with the rest of fluid particles. An example of the density distribution according to the depth can be seen in Figure B-12.

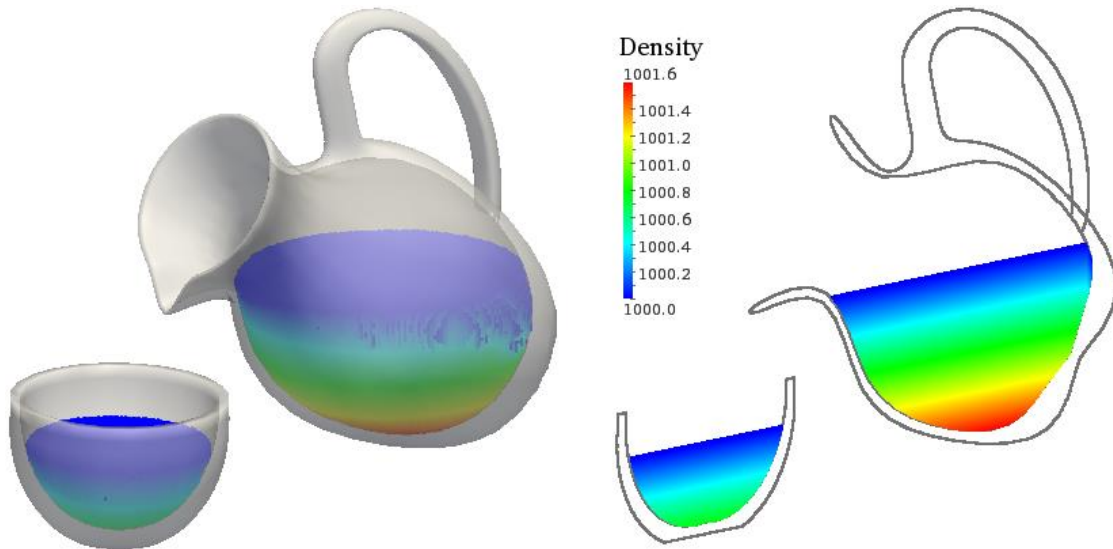


Figure B-12. Initial density distribution.

The value of *id* allows the identification of each particle using a unique number. This value is set for each particle according to the order of its creation. Particles are created following the order of the labels. For each label, the subset of particles is created sweeping the mesh in the direction Z^+ , then Y^+ and finally X^+ . However this order can be changed and defined as desired. This feature is very useful for visualisation and for tracking of the SPH particles during the simulation. The mixing between two different volumes of fluid can be observed in the Figure B-13.

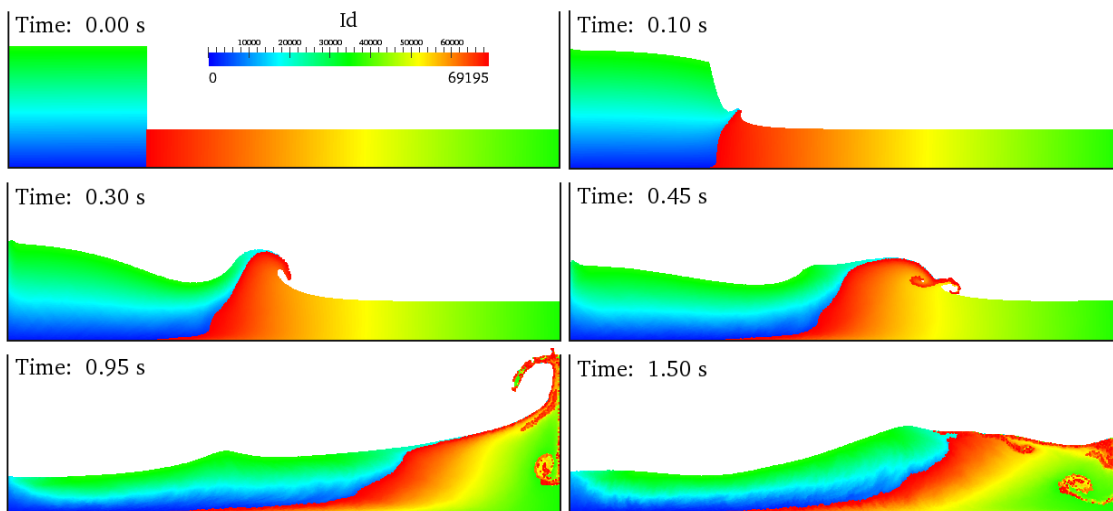


Figure B-13. Mixing of two fluids.

B.4 MOVEMENT DEFINITION

At this point we have the ability of representing any complex geometry by particles. Describing any kind of movement that mimics the behaviour of the real problem is imperative when engineering or industrial situations are going to be analysed.

Different kinds of movements can be imposed to a set of particles; rectilinear motion, rotational motion, circular motion and sinusoidal motion. Additionally, predefined motion can be imposed with data from an external file. Different instants of the movement of a pendulum are depicted in Figure B-14. The green piece follows a sinusoidal rotational motion, the yellow one follows a sinusoidal circular motion and the red one represents a sinusoidal rectilinear movement.

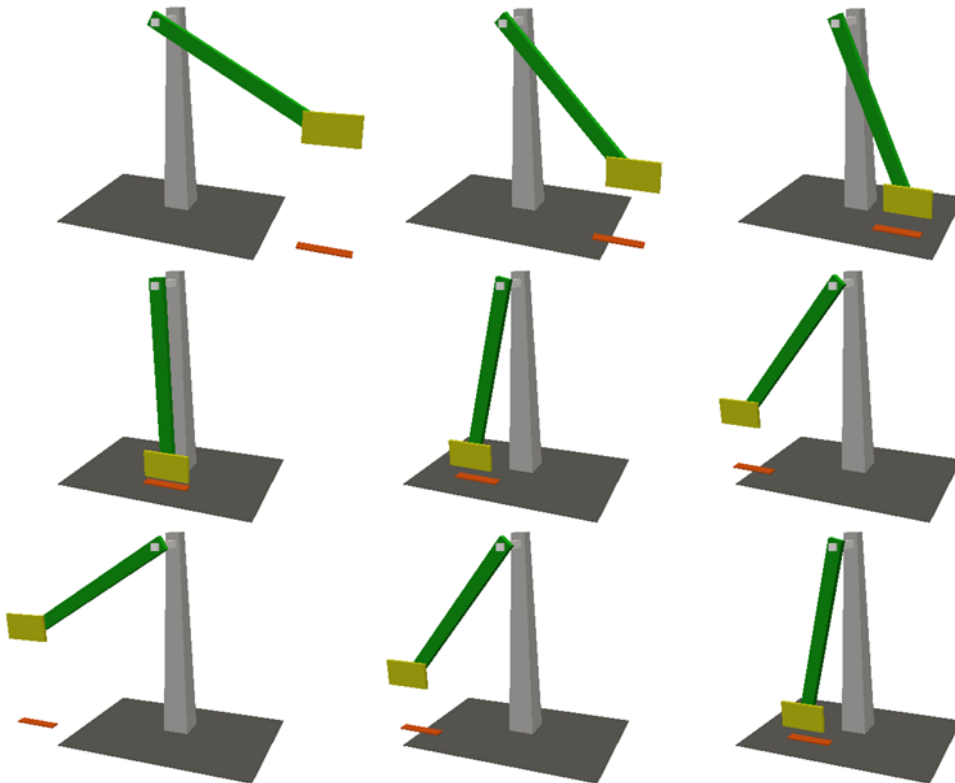


Figure B-14. Different instants of a pendulum movement (rotational, circular and rectilinear sinusoidal).

All movements are associated with a given duration and they are identified with a specific code. This code allows the linking of several movements in order to be executed one after another. The specific parameters for each kind of movement must be given. For example, the initial velocity and the acceleration values are required to define the accelerated rectilinear motion while frequency, amplitude, phase and an axis are required to define sinusoidal rotation. On the other hand, a

movement can be applied to an object (a set of particles with the same label) or a set of objects. Thus, a hierarchy of movements is created when an object has its own movement and a movement associated with its set at the same time. An example of hierarchy of movements is shown in Figure B-15 where the two mobile pieces of the mixer share a rotational movement while the red piece additionally has its own rotation.



Figure B-15. Mixer as an example of hierarchy of movements.

B.5 NORMAL VECTORS

Boundary conditions such as the repulsive forces need to compute the normal vectors at the position of each boundary particle. Using GenCase, normals are calculated for a triangle according to the order of the three vertices of each one as shown in Figure B-16. The right panel of the figure shows the result of computing normals for the given triangle. In this way, all particles that belong to this triangle have the same normal vector. When a particle belongs to different triangles, its normal vector is the result of averaging the different vectors.

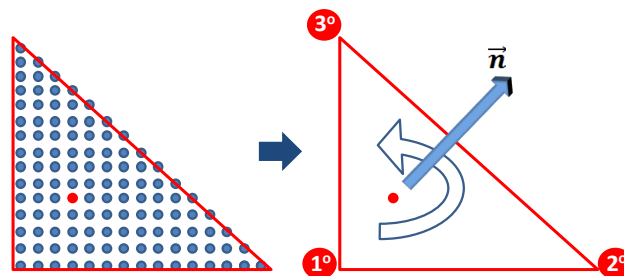


Figure B-16. Normal vector (\vec{n}) computation for a triangle.

Therefore, normal vectors can be computed for any complex object since it consists of triangles as shown in Figure B-17.

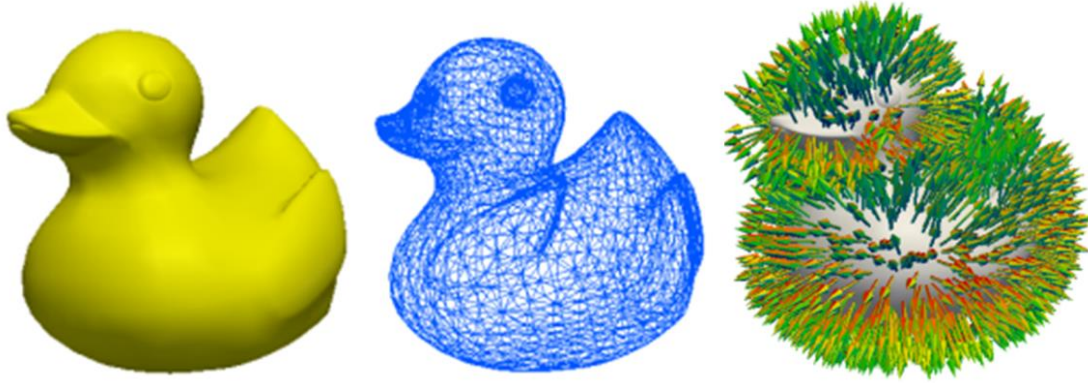


Figure B-17. Normal vector computation for a 3D object.

Figure B-17 shows a 3D object (left frame) that is formed by triangles (centre frame) so normal vectors of each triangle (right frame) can be calculated following the mentioned procedure.

B.6 EXAMPLES AND PERFORMANCE

Four testcases are described in this section to prove the capability and the performance of the GenCase code. A brief description of the case and computational times are presented for each case. Execution runtime is divided in three parts; representing the initial setup with points (*DrawPoints*), creating the particles starting from the points (*ToParticles*) and saving data in the output files (*SaveData*).

Table B-1 shows all the achieved results. These computational times are obtained with the same execution device: an Intel Core i7 at 2.93GHz, 6GB of RAM DDR3 at 1333 MHz and using Ubuntu 10.10 (64 bits).

Table B-1. Features of the cases.

Case	Dp	Results with GenCase			
		Particles	Fluid(%)	Time	Data Size
Sink 11,664 polygons	0.006	54,665,246	81.8%	91.9 s	1460 Mb
	0.007	35,366,936	79.2%	33.7 s	944 Mb
	0.01	13,102,483	72.6%	9.8 s	350 Mb
	0.015	4,399,652	64.0%	2.4 s	117 Mb
	0.02	2,060,729	56.4%	0.7 s	55 Mb

Mixer 28,879 polygons	0.0016	76,111,196	90.1%	193.1 s	2032 Mb
	0.0018	54,010,704	89.0%	88.5 s	1442 Mb
	0.002	39,814,547	87.9%	38.8 s	1063 Mb
	0.0025	20,948,274	85.3%	17.3 s	559 Mb
	0.003	12,461,843	82.8%	9.0 s	333 Mb
	0.004	5,488,221	78.1%	3.1 s	147 Mb
	0.005	2,967,685	74.3%	1.2 s	79 Mb
Pump 57,879 polygons	0.00085	81,006,785	93.0%	171.3 s	2163 Mb
	0.001	50,269,756	91.9%	59.2 s	1342 Mb
	0.0015	15,348,958	88.2%	11.9 s	410 Mb
	0.002	6,693,996	84.9%	5.0 s	179 Mb
	0.0025	3,523,610	81.6%	1.9 s	94 Mb
MiniCooper 3,848,388 polygons	0.00135	17,427,772	0.0%	39.8 s	465 Mb
	0.00145	15,047,528	0.0%	31.0 s	402 Mb
	0.0016	12,312,028	0.0%	23.2 s	329 Mb
	0.002	7,777,736	0.0%	13.0 s	208 Mb
	0.003	3,376,230	0.0%	4.5 s	90 Mb

B.6.1 Testcase Sink

The first example consists of a sink with water and with a floating duck. The geometry of the sink and the model of duck are created starting from external VTK files. The duck is a floating object, where the centre of gravity, inertia and mass are computed. The water is placed inside the sink using the filling algorithm. A representation of the case using polygons and particles is depicted in Figure B-18. The time taken by the three different parts mentioned above is shown in Figure B-19 for different number of particles. It can be observed that the highest cost in terms of computational time is the procedure to create particles from points and how the time dedicated to save data becomes the most expensive part for very large number of particles.

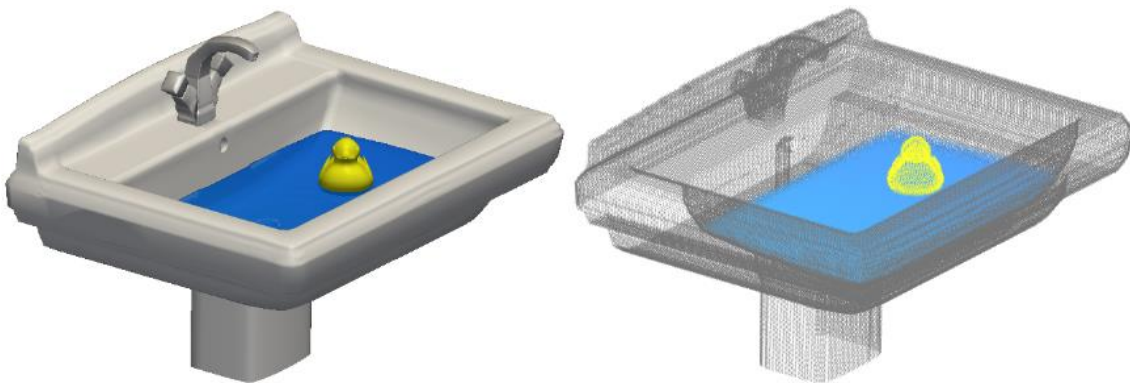


Figure B-18. Sink with floating object (polygons and particles).

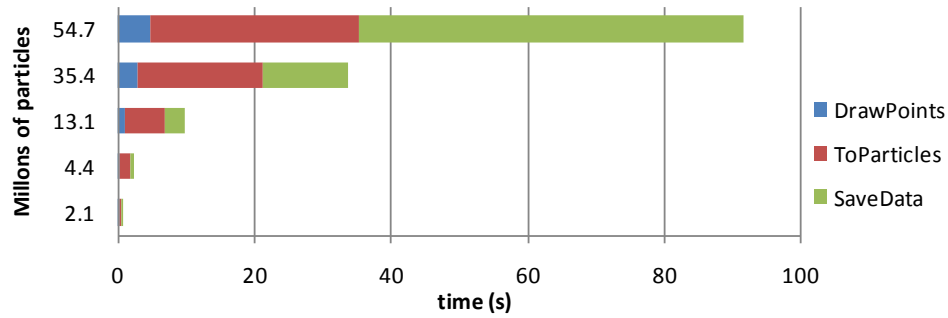


Figure B-19. Execution runtimes for the Sink.

B.6.2 Testcase Mixer

The second example is a mixer created from an external VTK file and fluid particles are introduced using the filling algorithm. The different types of rotational movements allow reproducing the motion of the pieces of the mixer. The geometry of the case is depicted in Figure B-20 and the execution runtimes are shown in Figure B-21.

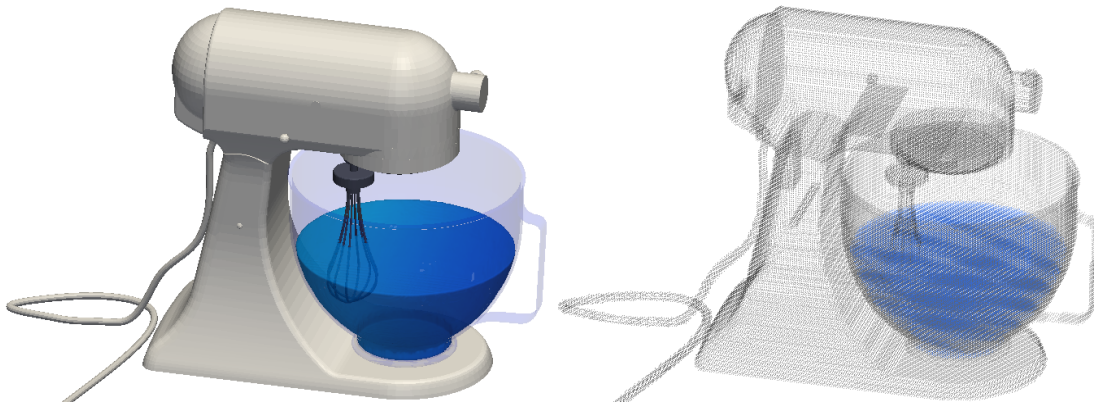


Figure B-20. Mixer (polygons and particles).

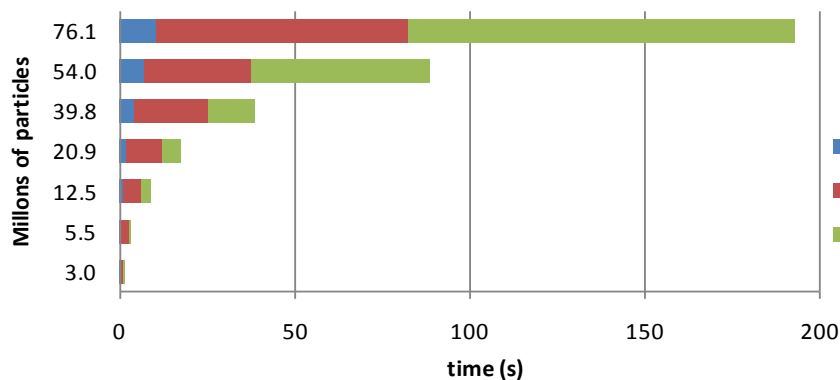


Figure B-21. Execution runtimes for the Mixer.

B.6.3 Testcase Pump

The third example consists of a water pump. The geometry is also loaded from an external VTK file which originally comes from a CAD geometry. Once again, the fluid is easily introduced using the filling algorithm. Figure B-22 shows the initial configuration of the case and the execution runtimes for different number of particles are represented in Figure B-23.

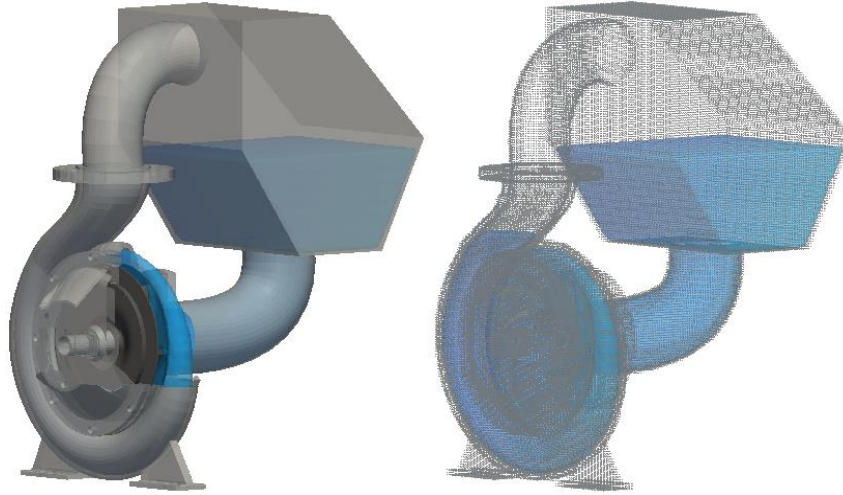


Figure B-22. Pump (polygons and particles).

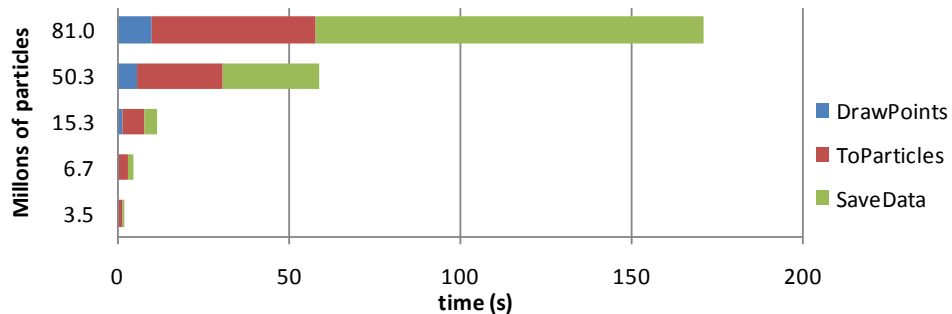


Figure B-23. Execution runtimes for the Pump.

B.6.4 Testcase Mini Cooper

There is no fluid in this case, only a Mini Cooper is represented using boundary particles. The geometry of the car is generated using an STL file with a lot of detail (3.8 million triangles). Figure B-24 shows the 3D model using polygons and using the wire mode the details of the model can be appreciated. The different execution times to generate the boundary particles are presented in Figure B-25.



Figure B-24. Mini Cooper (polygons and wire).

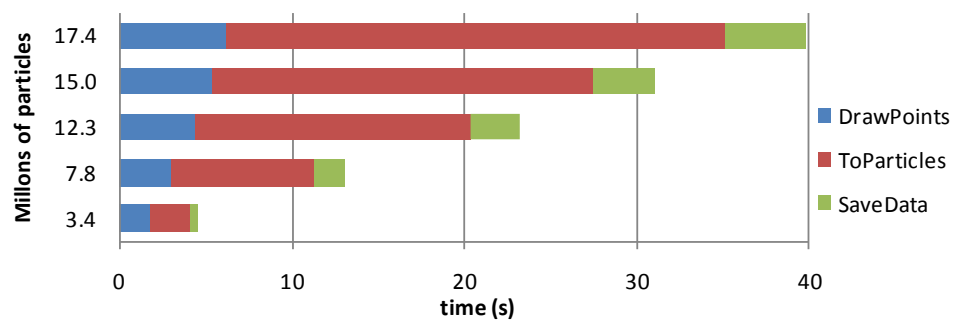


Figure B-25. Execution runtimes for the Mini Cooper.

B.7 REMARKS

A powerful tool named GenCase has been developed to generate the initial configuration of the system using particles for an SPH simulation. The use of external geometries, the filling of irregular shapes, the definition of different movements, the characterization of the floating objects and the normal vectors computation are the main features of this code. All these capabilities can be easily defined using an XML file.

The use of a 3D mesh does not only increase the performance of the code, but also simplifies the algorithms. This allows implementing new functionalities in an easy way.

GenCase has been proven to be efficient. It is fast enough to generate complex cases such as the Pump case with 80 million particles in less than 200 seconds. Furthermore, most of the time is consumed by saving data as it needs to save more than 2GB of data. In the case of the Mini Cooper, the conversion of 3.8 million triangles to particles takes less than 40 seconds.

C. POST-PROCESSING TOOLS

As we mentioned above, DualSPHysics is a powerful model that allows the analysis of complex flows, which make it ideal for engineering purposes. The final goal of the technique is to provide results to help designers and decision makers. As a consequence, it is mandatory to develop a full set of tools to analyse the obtained results. The main tools are described in this appendix.

C.1 PARTVTK

This code is used to convert the output binary files of DualSPHysics into different formats that can be visualised and /or analysed. Mainly the VTK format is used to show information about particles using the software *Paraview*. Paraview is an open-source and multi-platform program to visualise and to analyse scientific data. This package also supports other output formats like CSV (comma-separated values) or ASCII (American Standard Code for Information Interchange). PartVTK can get data of particles (position, velocity, density, mass) or calculate other values (press, acceleration, vorticity...), using all particles of simulation or only a selected part of them. The Figure C-1 illustrates how output of PartVTK is employed to visualise density of particles.

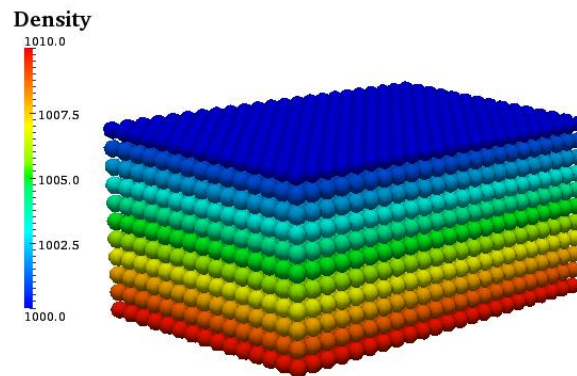


Figure C-1. Visualisation of density from a fluid block of particles.

C.2 MEASURETOOL

A tool is needed to analyse these numerical measurements to be compared with experiments. We must note that information in DualSPHysics is generated at the particles, whose position varies in time. Thus, information should be spatially averaged when the time evolution of a property is calculated. The MeasureTool code allows computing different physical quantities at a set of given points. MeasureTool calculates multiple physical quantities at any position. The binary files (.bi2) generated by DualSPHysics are the input files of the MeasureTool code and the output files can be VTK-binary or CSV or ASCII. The numerical values at a given position are computed by means of a SPH interpolation. This information depends on the values of the neighbouring particles averaged in terms of a kernel. An example of output MeasureTool is shown in Figure C-2 and Figure C-3.

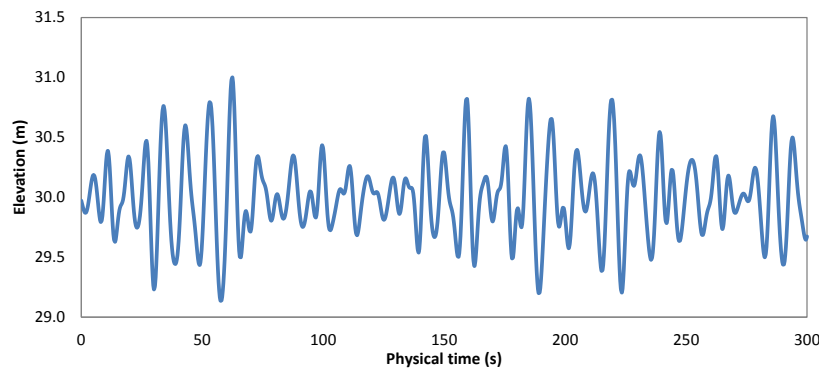


Figure C-2. Example of graph with wave elevation at a specific position.

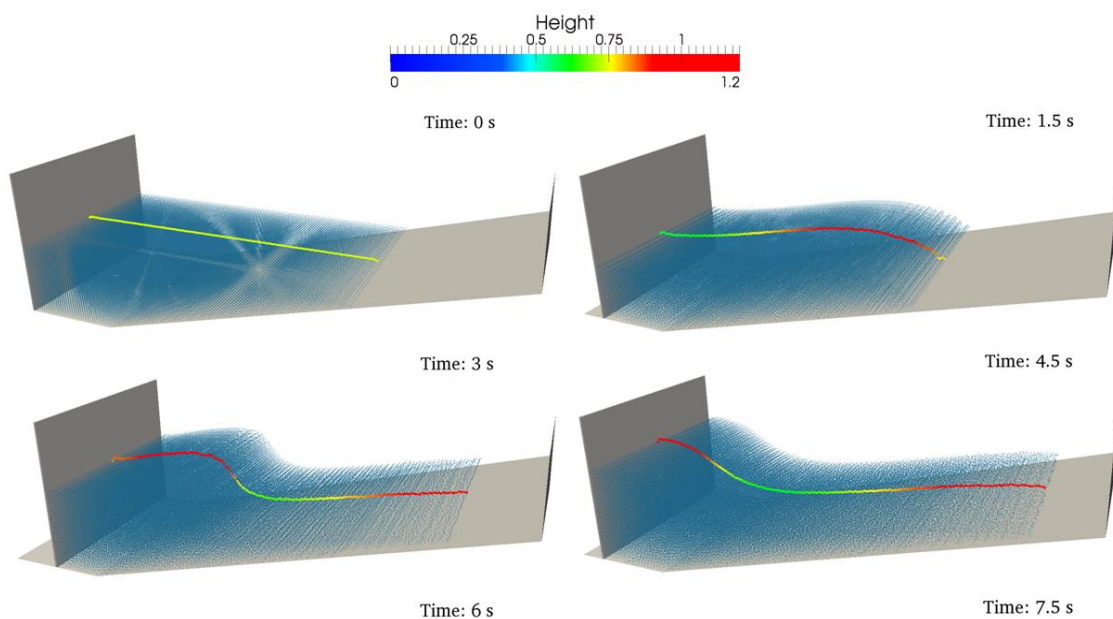


Figure C-3. Visualises the wave elevation for a slice of fluid.

C.3 ISO SURFACE

IsoSurface tool generates the isosurface of fluid to improve the visualisation when the number of particles is very high. In that case, the visualisation can be improved by representing surfaces instead of particles. To create the surfaces, the marching cubes algorithm is used [Lorensen and Cline, 1987]. This computer graphics technique extracts a polygonal mesh (set of triangles) of an isosurface from a 3-D scalar field.

Figure C-4, represents a 3D dam-break simulation using 300,000 particles. The first snapshot shows the particle representation. Values of mass are interpolated at the nodes of a 3-D Cartesian mesh that covers the entire domain using an SPH interpolation. Thus a 3-D mesh vertex that belongs to the free surface can be identified. The triangles of this surface (generated by means of the marching cubes algorithm) are represented in the second frame of the figure. The last snapshots correspond to the surface representation, where the colour corresponds to the interpolated velocity at the position of the triangles.

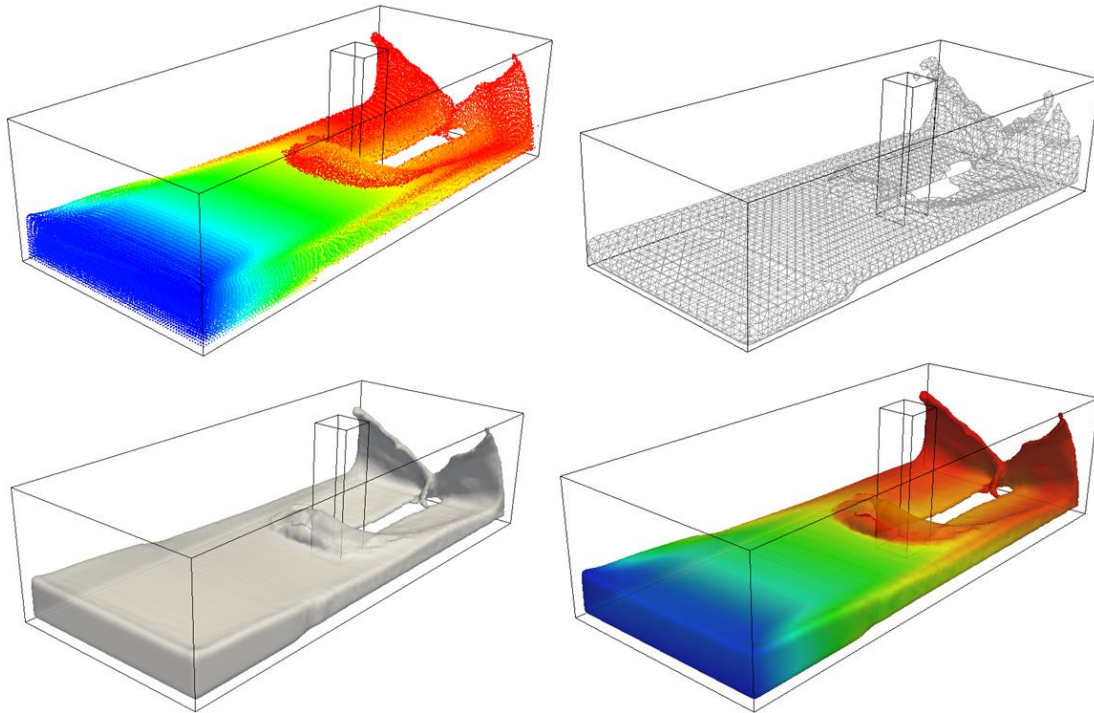


Figure C-4. Conversion of points to surfaces, from particles to isosurface.

C.4 DECIMATE

The use of the isosurface is a good option to represent the fluid when the number of particles is too high (more than 5 million particles) to visualise the particles in a standard personal computer. However, the isosurface can still be too heavy in some applications where the domain is huge and the resolution is very high (for example, the application shown in Figure 6-19). Thus, a method is needed to simplify the geometry of the isosurface and to reduce the number of triangles. The algorithm Decimation based on [Schroeder et al. 1992; Schroeder, 1997] is applied to reduce the number of triangles in a mesh but preserving the original topology of the mesh, and also considering the data associated with the vertices like velocity or density. Decimation technique was necessary in the case of the interaction of a large wave with an oil rig using more than one billion particles (described at Section 6.3). In that application the number of triangles of the isosurface reaches 180 million and Decimation was used to reduce this number to 10%. Another example can be also seen in Figure C-5 where the original isosurface contains 540,668 triangles and only 54,056 when applying Decimation.

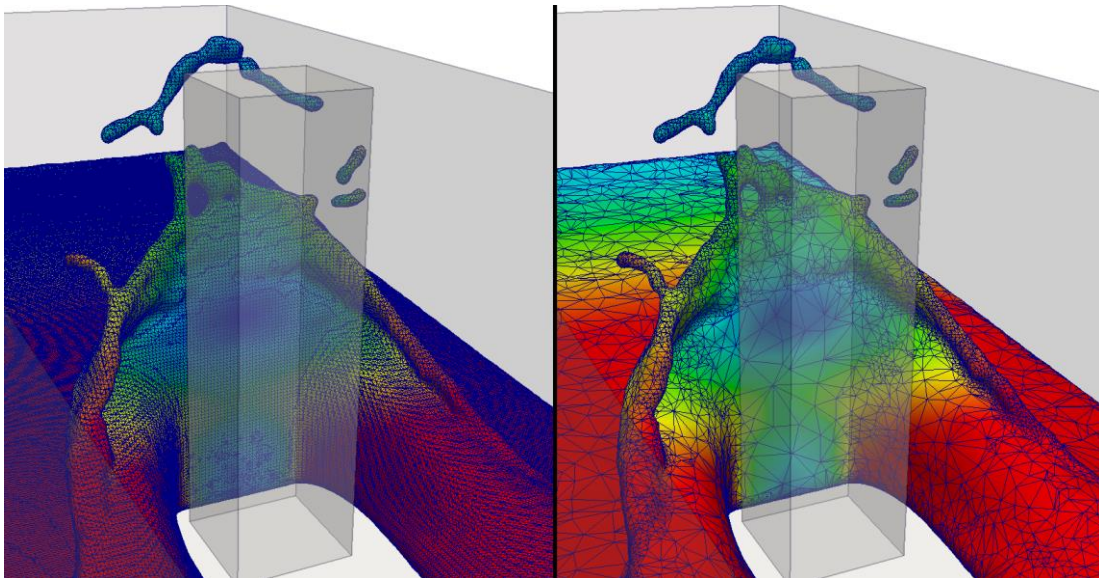


Figure C-5. Original isosurface of fluid (left) and simplified isosurface by Decimate program with a reduction to 10%.

C.5 BOUNDARYVTK

In order to visualise the boundary shapes formed by the boundary particles, different geometry files can be generated using the BoundaryVTK code. The code creates triangles or planes to represent the boundaries. This tool extracts the motion from boundary particles (moving or floating) to create a better visualisation of the moving objects using shapes instead of particles. This tool is also very useful for display purposes and to check the predefined movement of the boundary before starting the simulation. Figure C-6 shows the floating body movement using a box to clarify the visualisation.

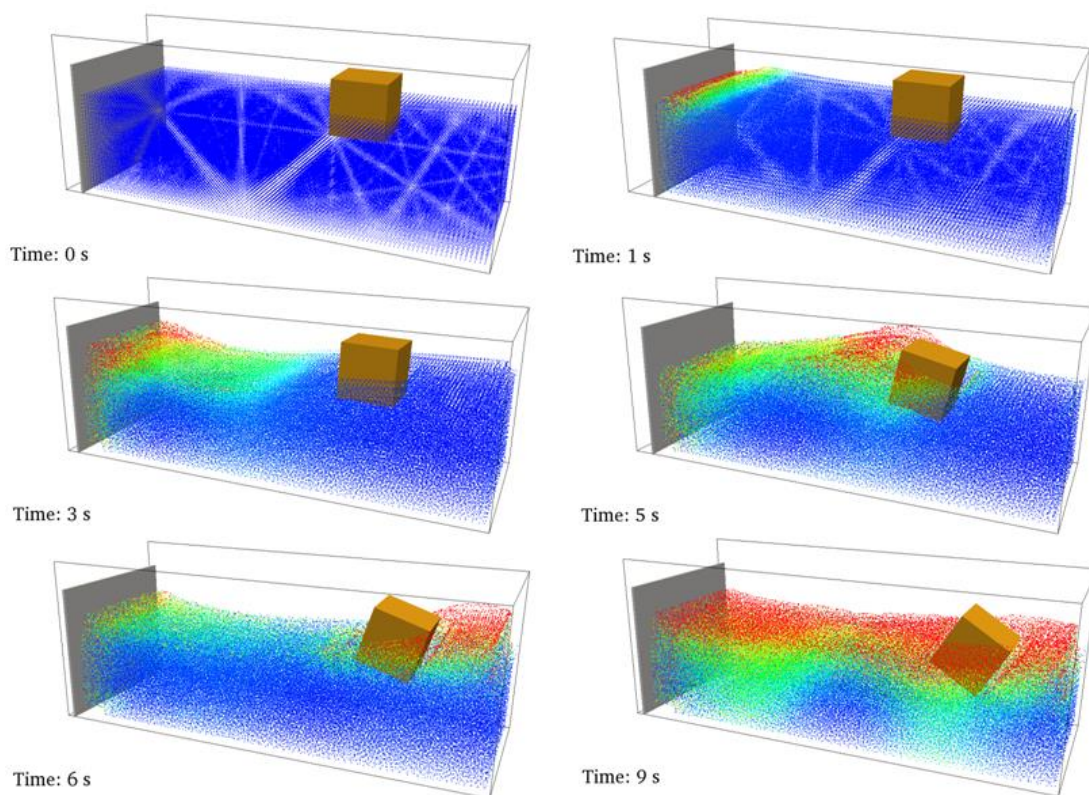


Figure C-6. Floating body movement represented using a box.

C.6 MEASUREBOXES

MeasureBoxes program calculates the volume of fluid and its velocity in any volume of the simulation. Thus, any volume can be delimited by triangles to measure the amount of fluid inside and the mean value of different properties like fluid. This tool is very useful to measure flows on complex terrains. Figure C-7 illustrates an example (presented in [Barreiro et al., 2014]) where MeasureBoxes is used to study the runoff on a real terrain. The rain water is collected in the dark area (top of Figure C-7). MeasureBoxes is used to study the effect of the ditch to avoid water arrival at the road (red area in the bottom of Figure C-7). Thus volume of fluid is measured at each time step.

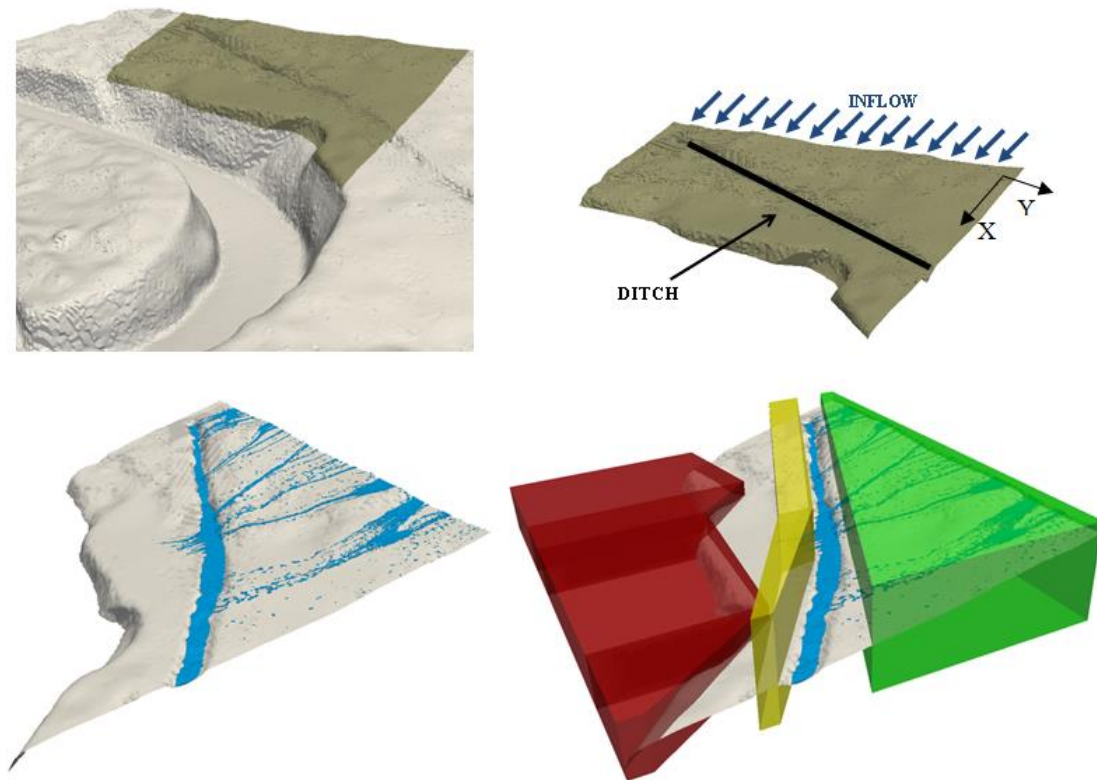


Figure C-7. Application of MeasureBoxes to measure a flow at complex terrain.

C.7 TRACERVTK

To observe the movement of fluid particles can be very complicated, especially in 3D simulations. This tool plots the trajectory of a set of selected particles to show clearly how these particles have moved during some interval of time. Figure C-8 shows an example where this tool is useful to visualise how fluid particles move inside the gaps among the blocks (antifers) of a coastal protection structure presented in [Altomare et al., 2014a].

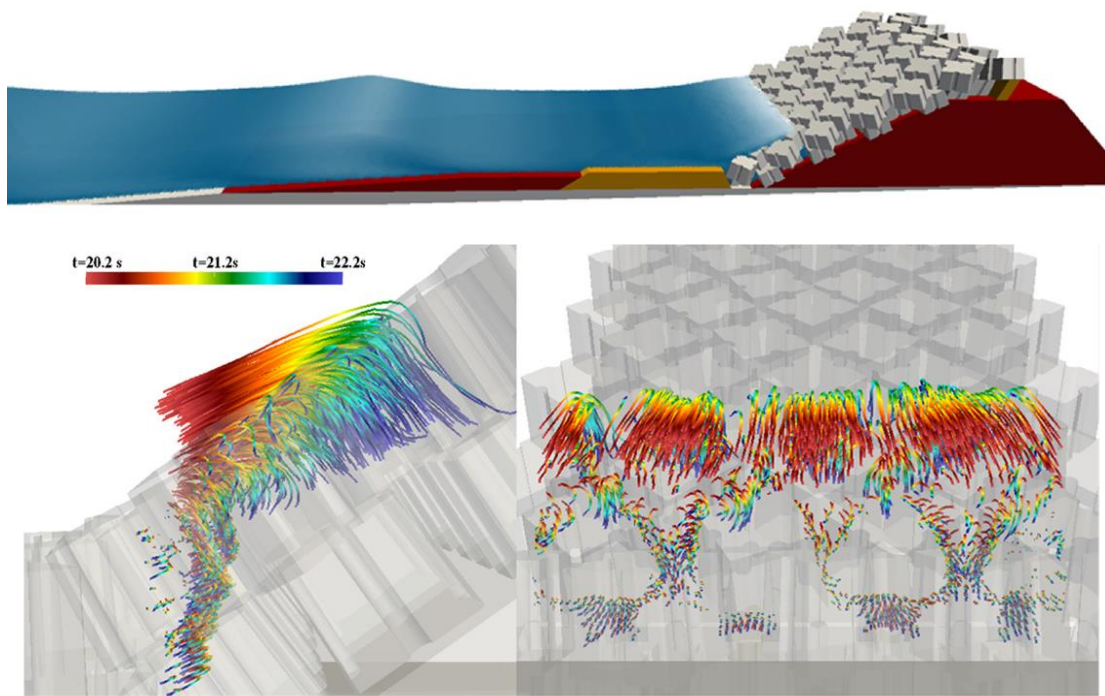


Figure C-8. Waves interaction with a coastal structure consisting of antifers and trajectories of fluid particles between antifers.

BIBLIOGRAPHY

- [Agarwal et al., 2012] P.K. Agarwal, S. Hampton, J. Poznanovic, A. Ramanathan, S.R. Alam and P.S. Crozier (2012). Performance modeling of microsecond scale biological molecular dynamics simulations on heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, DOI: 10.1002/cpe.2943.
- [Altomare et al., 2014a] C. Altomare, A.J.C. Crespo, B.D. Rogers, J.M. Domínguez, X. Gironella and M. Gómez-Gesteira (2014). Numerical modelling of armour block sea breakwater with Smoothed Particle Hydrodynamics. *Computers and Structures* 130, 34-45.
- [Altomare et al., 2014b] C. Altomare, T. Suzuki, J.M. Domínguez, A.J.C. Crespo and M. Gómez-Gesteira (2014). Coupling Between SWASH and SPH for Real Coastal Problems. *Proceedings of the 9th SPHERIC*, 254-259.
- [Anderson et al., 2008] J.A. Anderson, C.D. Lorenz and A. Travesset (2008). General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. *Journal of Computational Physics* 227, 5342-5359.
- [Antuono et al., 2012] M. Antuono, A. Colagrossi and S. Marrone (2012). Numerical diffusive terms in weakly-compressible SPH schemes. *Computer Physics Communications* 183.
- [Barreiro et al., 2013] A. Barreiro, A.J.C. Crespo, J.M. Domínguez and M. Gómez-Gesteira (2013). Smoothed Particle Hydrodynamics for coastal engineering problems. *Computers and Structures* 120(15), 96-106.
- [Barreiro et al., 2014] A. Barreiro, J.M. Domínguez, A.J.C. Crespo, H. González-Jorge, D. Roca and M. Gómez-Gesteira (2014). Integration of UAV

photogrammetry and SPH modelling of fluids to study runoff on real terrains. PLoS ONE, DOI: 10.1371/journal.pone.0111031.

[Batchelor, 1974] G.K. Batchelor (1974). Introduction to fluid dynamics. Cambridge University Press.

[Belytschko et al., 1998] T. Belytschko, Y. Krongauz, J. Dolbow and C. Gerlach (1998). On the completeness of meshfree particle methods. International Journal for Numerical Methods in Engineering 43, 785–819.

[Bisseling, 2004] R.H. Bisseling (2004). Parallel Scientific Computation: A Structured Approach using BSP and MPI. Oxford University Press, ISBN: 978-0-19-852939-2.

[Bonet and Lok, 1999] J. Bonet and T.-S.L. Lok (1999). Variational and momentum preservation aspects of Smoothed Particle Hydrodynamic formulations. Computer Methods in Applied Mechanics and Engineering 180, 97-115.

[Brown et al., 2011] W.M. Brown, P. Wang, S.J. Plimpton and A.N. Tharrington (2011). Implementing molecular dynamics on hybrid high performance computers - short range forces. Computer Physics Communications 182, 898-911.

[Buttlar et al., 1996] D. Buttlar, J. Farrell and B. Nichols (1996). PThreads Programming: A POSIX Standard for Better Multiprocessing. O'Reilly Media, ISBN: 978-1565921153.

[Canelas et al., 2014] R. Canelas, R.M.L. Ferreira, J.M. Domínguez and A.J.C. Crespo (2014). Modelling of Wave Impacts on Harbour Structures and Objects with SPH and DEM, Proceedings of the 9th SPHERIC, 313-320.

[Chandra et al., 1996] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan and J. McDonald (1996). Parallel Programming in OpenMP. Morgan Kauffman Publishers Inc., ISBN: 978-1558606715.

[Chandra et al., 2002] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan and J. McDonald (2002). OpenMP C and C++ Application Program Interface. OpenMP Architecture Review Board. <http://www.openmp.org/drupal/mp-documents/cspec20.pdf>.

- [Chen and Beraun, 2000] J.K. Chen and J.E. Beraun (2000). A generalized smoothed particle hydrodynamics method for nonlinear dynamic problems. *Computer Methods in Applied Mechanics and Engineering* 190, 225-239.
- [Clark, 1998] D. Clark (1998). OpenMP: A Parallel Standard for the Masses. *IEEE Concurrency* 6(1), 10-12, DOI: 10.1109/4434.656771
- [Colagrossi and Landrini, 2003] A. Colagrossi and M. Landrini (2003). Numerical simulation of interfacial flows by smoothed particle hydrodynamics. *Journal of Computational Physics* 191, 448-475.
- [Crespo et al., 2007] A.J.C. Crespo, M. Gómez-Gesteira and R.A. Dalrymple (2007). Boundary Conditions Generated by Dynamic Particles in SPH Methods. *CMC: Computers, Materials, & Continua* 5 (3), 173-184.
- [Crespo et al., 2008] A.J.C. Crespo, M. Gómez-Gesteira and R.A. Dalrymple (2008). Modeling Dam Break Behavior over a Wet Bed by a SPH Technique. *Journal of Waterway, Port, Coastal and Ocean Engineering* 134 (6), 313-320.
- [Crespo et al., 2009] A.J.C. Crespo, J.C. Marongiu, E. Parkinson, M. Gómez-Gesteira and J.M. Domínguez (2009). High Performance of SPH Codes: Best approaches for efficient parallelization on GPU computing. *Proc IVth Int SPHERIC Workshop (Nantes)*, 69-76.
- [Crespo et al., 2010] A.J.C. Crespo, J.M. Domínguez, A. Barreiro and M. Gómez-Gesteira (2010). Development of a Dual CPU-GPU SPH model. *Proc 5th Int SPHERIC Workshop (Manchester)*, 401-407.
- [Crespo et al., 2011] A.J.C. Crespo, J.M. Domínguez, A. Barreiro, M. Gómez-Gesteira and B.D. Rogers (2011). GPUs, a new tool of acceleration in CFD: Efficiency and reliability on Smoothed Particle Hydrodynamics methods. *PLoS ONE* 6 (6), e20685, DOI: 10.1371/journal.pone.0020685.
- [Crespo et al., 2014] A.J.C. Crespo, J.M. Domínguez, B.D. Rogers, M. Gómez-Gesteira, S. Longshaw, R. Canelas, R. Vacondio, A. Barreiro and O. García-Feal (2014). DualSPHysics: open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH). *Computer Physics Communications*, DOI: 10.1016/j.cpc.2014.10.004.
- [CUDA Programing Guide] Nvidia Corporation (2014). CUDA Programming Guide, <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>

- [Dagum and Menon, 1998] L. Dagum and R. Menon (1998). OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering* 5(1), 46-55, DOI: 10.1109/99.660313
- [Dalrymple and Rogers, 2006] R.A. Dalrymple and B.D. Rogers (2006). Numerical modeling of water waves with the SPH method. *Coastal Engineering* 53, 141–147.
- [Dickson et al., 2011] N.G. Dickson, K. Karimi and F. Hamze (2011). Importance of explicit vectorization for CPU and GPU software performance. *Journal of Computational Physics*, DOI: 10.1016/j.jcp.2011.03.041.
- [Dilts, 1999] G. Dilts (1999). Moving-least-square-particle hydrodynamics-I. Consistency and stability. *Internacional Journal for Numerical Methods in Engineering* 44, 1115–1155.
- [Domínguez et al., 2011a] J.M. Domínguez, A.J.C. Crespo, M. Gómez-Gesteira and J.C. Marongiu (2011). Neighbour lists in Smoothed Particle Hydrodynamics. *International Journal for Numerical Methods in Fluids* 67, 2026-2042.
- [Domínguez et al., 2011b] J.M. Domínguez, A.J.C. Crespo, A. Barreiro, M. Gómez-Gesteira and A. Mayrhofer (2011). Development of a new pre-processing tool for SPH models with complex geometries. *Schriftenreihe Schiffbau 6th SPHERIC*. Edited by Hamburg University of Technology, 117-124
- [Domínguez et al., 2013a] J.M. Domínguez, A.J.C. Crespo and M. Gómez-Gesteira (2013). Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method. *Computer Physics Communications* 184(3), 617-627, DOI: 10.1016/j.cpc.2012.10.015.
- [Domínguez et al., 2013b] J.M. Domínguez, A.J.C. Crespo, D. Valdez-Balderas, B.D. Rogers and M. Gómez-Gesteira (2013). New multi-GPU implementation for Smoothed Particle Hydrodynamics on heterogeneous clusters. *Computer Physics Communications* 184(8), 1848-1860, DOI: 10.1016/j.cpc.2013.03.008
- [Domínguez et al., 2014] J.M. Domínguez, A.J.C. Crespo, A. Barreiro, M. Gómez-Gesteira and B.D. Rogers (2014). Efficient implementation of double precision in GPU computing to simulate realistic cases with high resolution, *Proceedings of the 9th SPHERIC*, 140-145.

- [Ferrari et al., 2009] A. Ferrari, M. Dumbser, E.F. Toro and A. Armanini (2009). A new 3D parallel SPH scheme for free surface flows. *Computers & Fluids* 38, 1203–1217.
- [Fleissner and Eberhard, 2007] F. Fleissner and P. Eberhard (2007). Load balanced parallel simulation of particle-fluid DEM-SPH systems with moving boundaries. *John Von Neumann Institute for Computing* 48, 37-44.
- [Fourtakas et al., 2013] G. Fourtakas, B.D. Rogers and D. Laurence (2013). Modelling sediment suspension in industrial tanks using SPH. *La Houille Blanche* 2, 39-45.
- [Fourtakas et al., 2014] G. Fourtakas, J.M. Domínguez, R. Vacondio, A. Nasar and B.D. Rogers (2014). Local Uniform STencil (LUST) Boundary Conditions for 3-D Irregular Boundaries in DualSPHysics. *Proceedings of the 9th SPHERIC*, 103-110.
- [Geist et al., 1994] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam (1994). *Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, ISBN: 978-0262571081
- [Gingold and Monaghan, 1977] R.A. Gingold and J.J. Monaghan (1977). Smoothed particle hydrodynamics: theory and application to non- spherical stars. *Monthly Notices of the Royal Astronomical Society* 181, 375–389.
- [Goldberg 1991] D. Goldberg (1991). What every computer scientist should know about oating point arithmetic. *ACM Computing Surveys* 23(1), 5-48.
- [Gómez-Gesteira and Dalrymple, 2004] M. Gómez-Gesteira and R. Dalrymple (2004). Using a 3D SPH method for wave impact on a tall structure. *Journal of Waterway, Port, Coastal and Ocean Engineering* 130 (2), 63-69.
- [Gómez-Gesteira et al., 2012a] M. Gómez-Gesteira, B.D. Rogers, A.J.C. Crespo, R.A. Dalrymple, M. Narayanaswamy and J.M. Domínguez (2012). SPHysics - development of a free-surface fluid solver- Part 1: Theory and Formulations. *Computers & Geosciences* 48, 289-299.
- [Gómez-Gesteira et al., 2012b] M. Gómez-Gesteira, A.J.C. Crespo, B.D. Rogers, R.A. Dalrymple, J.M. Domínguez and A. Barreiro (2012). SPHysics - development of a free-surface fluid solver- Part 2: Efficiency and test cases. *Computers & Geosciences* 48, 300-307.

- [Goozee and Jacobs, 2003] R.J. Goozee and P.A. Jacobs (2003) Distributed and shared memory parallelism with a smoothed particle hydrodynamics code. Australian and New Zealand Industrial and Applied Mathematics Journal 44, C202–C228.
- [Gotoh et al., 2001] H. Gotoh, T. Shibihara and M. Hayashii (2001). Subparticle-scale model for the MPS method-lagrangian flow model for hydraulic engineering. Computational Fluid Dynamics Journal 9, 339–347.
- [Gotoh et al., 2004] H. Gotoh, S. Shao and T. Memita (2004). SPH-LES model for numerical investigation of wave interaction with partially immersed breakwater. Coastal Engineering Journal 46(1), 39–63.
- [Gropp et al., 1999] W. Gropp, E. Lusk and A. Skjellum (1999). Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press, ISBN: 978-0262571326
- [Harada et al., 2007] T. Harada, S. Koshizuka and Y. Kawaguchi (2007). Smoothed particle hydrodynamics on GPUs. Computer Graphics International, 63–70.
- [Herault et al., 2010] A. Herault, G. Bilotta and R.A. Dalrymple (2010). SPH on GPU with CUDA. Journal of Hydraulic Research 48, 74–79.
- [IEEE 754 Standard] IEEE 754-2008 (2008). IEEE 754-2008 Standard for Floating-Point Arithmetic.
- [Ihmsen et al., 2011] M. Ihmsen, N. Akinci, M. Becker and M. Teschner (2011). A parallel SPH implementation on multi-core CPUs. Computer Graphics Forum 30(1), 99–112, DOI: 10.1111/j.1467-8659.2010.01832.x
- [Khayyer and Gotoh, 2009] A. Khayyer and H. Gotoh (2009). Wave impact pressure calculations by improved SPH methods. International Journal of Offshore and Polar Engineering 19, 300–307.
- [Khronos, 2009] Khronos (2009), The OpenCL Specification, <https://www.khronos.org/opencl>
- [Kolb and Cuntz, 2005] A. Kolb and N. Cuntz (2005). Dynamic particle coupling for GPU-based fluid simulation. Proceedings of the 18th Symposium on Simulation Technique, 722–727.

- [Qiang et al., 2012] W. Qiang, Y. Canqun, T. Tao and L. Kai (2012). Fast parallel cutoff pair interactions for molecular dynamics on heterogeneous systems. *Tsinghua Science and Technology* 17, 265-277.
- [Lee et al., 2010] V.W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A.D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupati, P. Hammarlund, R. Singhal, and P. Dubey (2010). Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News* 38, 451-460
- [Leimkuhler et al., 1996] B.J. Leimkuhler, S. Reich and R.D. Skeel (1996). *Integration Methods for Molecular dynamic* IMA Volume in Mathematics and its application. Springer.
- [Lo and Shao, 2002] E.Y.M. Lo and S. Shao (2002). Simulation of near-shore solitary wave mechanics by an incompressible SPH method. *Applied Ocean Research* 24, 275-286.
- [Lorensen and Cline, 1987] W.E. Lorensen and H.E. Cline (1987). Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 163-170, ISBN: 0-89791-227-6, DOI: 10.1145/37401.37422.
- [Lucy, 1977] L.B. Lucy (1977). A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal* 82, 1013–1024, DOI: 10.1086/112164.
- [Marongiu et al., 2010] J.C. Marongiu, F. Leboeuf, J. Caro and E. Parkinson (2010). Free surface flows simulations in Pelton turbines using an hybrid SPH-ALE method. *Journal of Hydraulic Research* 48 Extra Issue, 40–49.
- [Maruzewski et al., 2010] P. Maruzewski, D. Le Touzé, G. Oger and F. Avellan (2010). SPH high-performance computing simulations of rigid solids impacting the free-surface of water, *Journal of Hydraulic Research* 48, 126–134.
- [McInstosh-Smith et al., 2012] S. McInstosh-Smith, T. Wilson, A.A. Ibarra, J. Crisp and R.B. Sessions (2012). Benchmarking Energy Efficiency, Power Costs and Carbon Emissions on Heterogeneous Systems. *The Computer Journal* 55(2), 192-205, DOI: 10.1093/comjnl/bxr091

- [Mokos et al., 2014] A. Mokos, B.D. Rogers, P.K. Stansby and J.M. Domínguez (2014). A multi-phase particle shifting algorithm for SPH simulations for violent hydrodynamics on a GPU. *Proceedings of the 9th SPHERIC*, 1-8.
- [Molteni and Colagrossi, 2009] D. Molteni and A. Colagrossi (2009). A simple procedure to improve the pressure evaluation in hydrodynamic context using the SPH. *Computer Physics Communications* 180(6), 861–872, DOI: 10.1016/j.cpc.2008.12.004
- [Monaghan, 1989] J.J. Monaghan (1989), On the problem of penetration in particle methods. *Journal of Computational Physics* 82(1), 1-15, DOI: 10.1016/0021-9991(89)90032-6.
- [Monaghan, 1992] J.J. Monaghan (1992). Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics* 30(1), 543-574, DOI: 10.1146/annurev.aa.30.090192.002551.
- [Monaghan, 1994] J.J. Monaghan (1994). Simulating free surface flows with SPH. *Journal of Computational Physics* 110, 399- 406, DOI: 10.1006/jcph.1994.1034.
- [Monaghan, 1996] J.J. Monaghan (1996). Gravity currents and solitary waves. *Physica D: Nonlinear Phenomena* 98(2-4), 523–533.
- [Monaghan, 2000] J.J. Monaghan (2000). SPH without Tensile Instability. *Journal Computational Physics* 159(2), 290-311, DOI: 10.1006/jcph.2000.6439.
- [Monaghan, 2005] J.J. Monaghan (2005), Smoothed Particle Hydrodynamics. *Reports on Progress in Physics* 68(8), 1703-1759, DOI: 10.1088/0034-4885/68/8/R01.
- [Monaghan and Kos, 1999] J.J. Monaghan and A. Kos (1999). Solitary waves on a Cretan beach. *Journal of Waterway, Port, Coastal and Ocean Engineering* 125(3), 145-154.
- [Monaghan et al., 1999] J.J. Monaghan, R.A.F. Cas, A.M. Kos and M. Hallworth (1999). Gravity currents descending a ramp in a stratified tank. *Journal of Fluid Mechanics* 379, 39–70.

- [Monaghan et al., 2003] J.J. Monaghan, A. Kos and N. Issa (2003). Fluid motion generated by impact. *Journal of Waterway, Port, Coastal and Ocean Engineering* 129, 250-259.
- [Nickolls et al., 2008] J. Nickolls, I. Buck, M. Garland, and K. Skadron (2008). Scalable parallel programming with CUDA. *ACM Queue* 6(2), 40-53.
- [Nickolls and Dally, 2010] J. Nickolls and W.J. Dally (2010). The GPU computing era. *IEEE Micro* 30, 56–69.
- [Owens et al., 2007] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn and T.J. Purcell (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26(1), 80–113.
- [Pacheco, 1996] P. Pacheco (1996). *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., ISBN: 978-1558603394.
- [Ren et al., 2014] B. Ren, H. Wen, P. Dong and Y. Wang (2014). Numerical simulation of wave interaction with porous structures using an improved smoothed particle hydrodynamic method. *Coastal Engineering* 88, 88–100.
- [Rogers et al., 2010] B.D. Rogers, R.A. Dalrymple and P.K. Stansby (2010). Simulation of caisson breakwater movement using SPH. *Journal of Hydraulic Research* 48, 135-141.
- [Rustico et al., 2014] E. Rustico, G. Bilotta, A. Hérault, C. Del Negro and G. Gallo (2014). Advances in multi-GPU Smoothed Particle Hydrodynamics simulations. *IEEE Transactions on Parallel and Distributed Systems* 25(1), 43-52
- [Satish et al., 2009] N. Satish, M. Harris and M. Garland (2009). Designing Efficient Sorting Algorithms for Manycore GPUs. *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 1-10, DOI: 10.1109/IPDPS.2009.5161005.
- [Schroeder, 1997] W.J. Schroeder (1997). A Topology Modifying Progressive Decimation Algorithm. *VIS '97 Proceedings of the 8th conference on Visualization '97*, ISBN: 1-58113-011-2
- [Schroeder et al. 1992] W.J. Schroeder, J.A. Zarge, W.E. Lorensen (1992). Decimation of Triangle Meshes. *ACM SIGGRAPH Computer Graphics* 26(2), 65-70, DOI: 10.1145/142920.134010

- [Shao, 2005] S.D. Shao (2005). SPH simulation of solitary wave interaction with a curtain-type breakwater. *Journal of Hydraulic Research* 43(4), 366–375, DOI: 10.1080/00221680509500132.
- [Snir et al., 1998] M. Snir, J. Dongarra, J.S. Kowalik, S. Huss-Lederman, S.W. Otto and D.W. Walker (1998). *MPI: The Complete Reference*. MIT Press, ISBN: 978-0262692168.
- [St-Germain et al., 2014] P. St-Germain, I. Nistor, R. Townsend and T. Shibayama (2014). Smoothed-Particle Hydrodynamics Numerical Modeling of Structures Impacted by Tsunami Bores. *Journal of Waterway, Port, Coastal, and Ocean Engineering* 140(1), 66-81.
- [Stellingwerf and Wingate, 1994] R.F. Stellingwerf and C.A. Wingate (1994). Impact Modelling with SPH. *Memorie della Societa Astronomia Italiana* 65(4), 1117-1128.
- [Trott et al., 2010] C.R. Trott, L. Winterfeld and P.S. Crozier (2010). General-purpose molecular dynamics simulations on GPU-based clusters. *Computer Physics Communications*, arXiv:1009.4330 (2010).
- [Liu, 2003] G.R. Liu (2003). *Mesh Free methods: Moving beyond the finite element method*. CRC Press, ISBN: 978-0849312380.
- [Vacondio et al., 2012] R. Vacondio, B.D. Rogers, P.K. Stansby and P. Mignosa (2012). A correction for balancing discontinuous bed slopes in two-dimensional smoothed particle hydrodynamics shallow water modelling. *International Journal for Numerical Methods in Fluids* 71, 850–872.
- [Vacondio et al., 2013a] R. Vacondio, B.D. Rogers, P.K. Stansby and P. Mignosa (2013). Shallow water SPH for flooding with dynamic particle coalescing and splitting. *Advances in Water Resources* 58, 10-23.
- [Vacondio et al., 2013b] R. Vacondio, B.D. Rogers, P.K. Stansby, P. Mignosa and J. Feldman (2013). Variable resolution for SPH: a dynamic particle coalescing and splitting scheme. *Computer Methods in Applied Mechanics and Engineering* 256, 132-148.
- [Valdez-Balderas et al., 2012] D. Valdez-Balderas, J.M. Domínguez, A.J.C. Crespo and B.D. Rogers (2012). Towards accelerating Smoothed Particle Hydrodynamics simulations for free-surface flows on multi-GPU clusters.

Journal of Parallel and Distributed Computing, DOI:
10.1016/j.jpdc.2012.07.010.

- [Verlet, 1967] L. Verlet (1967), Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review* 159, 98-103.
- [Viccione et al., 2008] G. Viccione, V. Bovolin and E.P. Carratelli (2008). Defining and optimizing algorithms for neighbouring particle identification in SPH fluid simulations. *International Journal for Numerical Methods in Fluids* 58, 625–638, DOI: 10.1002/flid.1761.
- [Vila, 1999] J.P. Vila (1999). On particle weighted methods and SPH. *Mathematical Models and Methods in Applied Sciences* 9, 161–210, DOI: 10.1142/S0218202599000117.
- [Violeau, 2012] D. Violeau (2012). *Fluid Mechanics and the SPH Method: Theory and Applications*, Oxford University Press, ISBN: 0199655529.
- [Wendland, 1995] H. Wendland (1995). Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics* 4, 389-396.
- [Whitehead and Fit-Florea 2011] N. Whitehead and A. Fit-Florea (2011). Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs, NVIDIA Technical White Paper

LIST OF PUBLICATIONS

- [Domínguez et al., 2011a] **J.M. Domínguez**, A.J.C. Crespo, M. Gómez-Gesteira and J.C. Marongiu (2011). Neighbour lists in Smoothed Particle Hydrodynamics. *International Journal for Numerical Methods in Fluids* 67, 2026-2042.
- [Crespo et al., 2011] A.J.C. Crespo, **J.M. Domínguez**, A.Barreiro, M. Gómez-Gesteira and B.D. Rogers (2011). GPUs, a new tool of acceleration in CFD: Efficiency and reliability on Smoothed Particle Hydrodynamics methods. *PLoS ONE* 6 (6), e20685, DOI: 10.1371/journal.pone.0020685.
- [Gómez-Gesteira et al., 2012a] M. Gómez-Gesteira, B.D. Rogers, A.J.C. Crespo, R.A. Dalrymple, M. Narayanaswamy and **J.M. Domínguez** (2012). SPHysics - development of a free-surface fluid solver- Part 1: Theory and Formulations. *Computers & Geosciences* 48, 289-299.
- [Gómez-Gesteira et al., 2012b] M. Gómez-Gesteira, A.J.C. Crespo, B.D. Rogers, R.A. Dalrymple, **J.M. Domínguez** and A. Barreiro (2012). SPHysics - development of a free-surface fluid solver- Part 2: Efficiency and test cases. *Computers & Geosciences* 48, 300-307.
- [Valdez-Balderas et al., 2012] D. Valdez-Balderas, **J.M. Domínguez**, A.J.C. Crespo and B.D. Rogers (2012). Towards accelerating Smoothed Particle Hydrodynamics simulations for free-surface flows on multi-GPU clusters. *Journal of Parallel and Distributed Computing*, DOI: 10.1016/j.jpdc.2012.07.010.
- [Domínguez et al., 2013a] **J.M. Domínguez**, A.J.C. Crespo and M. Gómez-Gesteira (2013). Optimization strategies for CPU and GPU implementations of

- a smoothed particle hydrodynamics method. *Computer Physics Communications* 184(3), 617-627. DOI: 10.1016/j.cpc.2012.10.015.
- [Domínguez et al., 2013b] **J.M. Domínguez**, A.J.C. Crespo, D. Valdez-Balderas, B.D. Rogers and M. Gómez-Gesteira (2013). New multi-GPU implementation for Smoothed Particle Hydrodynamics on heterogeneous clusters. *Computer Physics Communications* 184(8), 1848-1860, DOI: 10.1016/j.cpc.2013.03.008
- [Barreiro et al., 2013] A. Barreiro, A.J.C. Crespo, **J.M. Domínguez** and M. Gómez-Gesteira (2013), Smoothed Particle Hydrodynamics for coastal engineering problems. *Computers and Structures* 120(15), 96-106.
- [Altomare et al., 2014a] C. Altomare, A.J.C. Crespo, B.D. Rogers, **J.M. Domínguez**, X. Gironella and M. Gómez-Gesteira (2014). Numerical modelling of armour block sea breakwater with Smoothed Particle Hydrodynamics. *Computers and Structures* 130, 34-45.
- [Barreiro et al., 2014] A. Barreiro, **J.M. Domínguez**, A.J.C. Crespo, H. González-Jorge, D. Roca and M. Gómez-Gesteira (2014). Integration of UAV photogrammetry and SPH modelling of fluids to study runoff on real terrains. *PLoS ONE*, DOI: 10.1371/journal.pone.0111031.
- [Crespo et al., 2014] A.J.C. Crespo, **J.M. Domínguez**, B.D. Rogers, M. Gómez-Gesteira, S. Longshaw, R. Canelas, R. Vacondio, A. Barreiro and O. García-Feal (2014). DualSPHysics: open-source parallel CFD solver based on Smoothed Particle Hydrodynamics (SPH). *Computer Physics Communications*, DOI: 10.1016/j.cpc.2014.10.004.
- [Altomare et al., 2014c] C. Altomare, A.J.C. Crespo, **J.M. Domínguez**, M. Gómez-Gesteira, T. Suzuki and T. Verwaest (2014). Applicability of Smoothed Particle Hydrodynamics for estimation of sea wave impact on coastal structures. *Coastal Engineering*, submitted 2014.
- [Canelas et al., 2014] R. Canelas, **J.M. Domínguez**, A.J.C. Crespo, M. Gómez-Gesteira, R.M.L. Ferreira (2014). A Smooth Particle Hydrodynamics discretization for the modelling of free surface flows and rigid body dynamics. *International Journal for Numerical Methods in Fluids*, submitted 2014.