# SMOOTHED PARTICLE HYDRODYNAMICS ON GRAPHICS PROCESSING UNITS

CHRISTOPHER MCCABE

A thesis submitted in partial fulfilment of the requirements

of the

Manchester Metropolitan University for the degree of

Doctor of Philosophy

School of Computing, Mathematics & Digital Technology

the Manchester Metropolitan University

July 2012

# Abstract

A recent development in Computational Fluid Dynamics (CFD) has been the meshless method called Weakly Compressible Smoothed Particle Hydrodynamics (WCSPH), which is a Lagrangian method that tracks physical quantities of a fluid as it moves in time and space. One disadvantage of WCSPH is the small time steps required due to the use of the weakly compressible Tait equation of state, so large scale simulations using WCSPH have so far been rare and only performed on very expensive CPU-based supercomputers. As CFD simulations grow larger and more detailed, the need to use high performance computing also grows. There is therefore great interest in any computer technology that can provide the equivalent computational power of the CPU-based supercomputer for a fraction of the cost. Hence the excitement aroused in the SPH community by the Graphics Processing Unit (GPU).

The GPU offers great potential for providing significant increases in computational performance due to its much smaller size and power consumption relative to the more established and traditional high performance computers comprising hundreds or thousands of CPUs. However, there are some disadvantages in programming GPUs. The memory structure of the GPU is more complex and more variable in speed, and there are other factors that can seriously affect performance, such as the thread grid dimensions which drives the occupancy of the GPU.

The aim of this thesis is to describe how WCSPH can be efficiently implemented on multiple GPUs.

First, some CFD methods and their success or otherwise in simulating free surfaces are discussed, and examples of previous attempts at implementing CFD algorithms on GPUs are given. The mathematical theory of WCSPH is then presented, followed by a detailed examination of the architecture of a GPU and how to program a GPU. Two different implementations of the same WCSPH algorithm are then described to simulate a well known experiment of a collapse of a column of water to highlight two possible uses of the GPU memory. The first method uses the fast shared memory of the GPU, which is recommended by the GPU manufacturer, while the second method uses the texture memory of the GPU, which acts as a cache. It is shown that due to the theory of WCSPH, which allows particles to only interact with other particles a short distance apart, that despite the speed of the shared memory and the power of coalescing data into the shared memory, the texture memory method is currently the most efficient, but that this method of implementing WCSPH on a single GPU requires a much higher degree of complexity of programming than the shared memory method. It is also shown that the size of the thread block can have a significant effect on performance.

Riemann solvers add more computational effort but can provide more accuracy. The use of Riemann solvers in WCSPH and their success or otherwise is then examined, and the results and performance of one particular WCSPH algorithm that uses an approximate Riemann solver when executed on a GPU are reported.

The treatment of boundaries has been and continues to be a problem in WCSPH, and there are a number of creative proposals for boundary treatments. Some of these are described in detail before a new boundary treatment is proposed that builds upon a boundary treatment that was recently proposed, and improves its performance in execution time on a GPU by using the registers and not the slower memories of the GPU. This new boundary treatment builds a unique private grid of boundary particles for each fluid particle close to the boundary. All computation is performed in the registers, the properties of the boundary particles depend on the fluid particle only, and there is no requirement to recall data from the slower global or texture memories of the GPU. The new boundary treatment is also shown to propagate a solitary wave further, preserves the wave height more and takes less execution time to compute than the original boundary treatment this new treatment builds on.

A unique and simple implementation of WCSPH on multiple GPUs is then described, and the results of a simulation of a collapse of a column of water in 3D are reported and compared against the results from a simulation of the same problem with the same WCSPH algorithm executed on a large cluster of multi core CPUs. The conclusion is that simulations on a small cluster of GPUs can achieve greater performance than from a cluster of multi core CPUs, but to achieve this the slow GPU memories, including the texture memory, must be avoided by using the registers as much as possible, and the architecture of the network linking the GPUs together must be exploited. The former was achieved by using the new boundary treatment proposed in this thesis and discussed above, and the latter was achieved by the use of the MPI Group functionality. The GPUs used for this thesis were already connected together in boxes of 4 by the manufacturer. The cluster used for this thesis consisted of 8 of these boxes, giving a total of 32 GPUs. These boxes of 4 GPUs were connected together through a common host, but the communication speed over the connection between the box and the host is much slower than that between the GPUs inside the box. The total communication time was minimized by grouping the GPUs inside a box together with their private unique MPI communicator, and a communication procedure was created to minimize communication over the relatively slow connection between the boxes of GPUs and the host.

Finally, some conclusions are drawn and suggestions for further work are made.

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for any other degree or qualification at this or any other institution.

Apart from those parts of this thesis which contain citations to the work of others and apart from the assistance mentioned in the acknowledgements, this thesis is my own work.

Christopher McCabe

# Acknowledgments

I would like to thank the following.

- my Director of Studies Professor Derek Causon and Second Supervisor Professor Clive Mingham for their support, encouragement and generosity in providing the computational resources required for this work,
- the EPSRC for the Doctoral Training Account research grant that financed this research,
- the UK STFC Daresbury Laboratory for their permission to use the NVIDIA S1070 cluster based there,
- Igor Kozin formerly at UK STFC Daresbury Laboratory for his guidance on how to use the NVIDIA S1070 cluster at Daresbury Laboratory,
- and finally my family and friends for helping and supporting me during challenging times.

# Contents

# List of Figures

# List of Tables

# Nomenclature

**Operations**

| | |
|---|---|
| $\nabla$ | Gradient |
| $\Omega$ | Support domain |

**Scalars**

| | |
|---|---|
| $\mu$ | Absolute or dynamic viscosity |
| $\rho$ | Density |
| $c$ | Speed of sound |
| $h$ | Smoothing length |
| $m$ | Mass |
| $P$ | Pressure |
| $W$ | Smoothing function |

**Vectors**

| | |
|---|---|
| $\underline{v}$ | Velocity |
| $\underline{x}$ | Position |

# Chapter 1

# Introduction

Modeling of free surfaces in fluid dynamics is notoriously difficult due to rapidly changing structures of fluids and/or moving boundaries which interact with those fluids. Finite difference, volume and element methods have been used to model this family of complex problems with reasonable success. These methods require a mesh of nodes at which physical properties are calculated at each time step, but also require that mesh to be altered to account for the rapidly changing structure of the problem. This mesh alteration adds to the computation time. However, recently a method of modeling fluids called Smoothed Particle Hydrodynamics (SPH) has been the focus of investigation in free surface problems. SPH requires no mesh of nodes to both update and restructure, but instead tracks a set of particles as they move and interact with each other and any surrounding boundaries.

Previous attempts at modeling free surfaces have been made, some of which involve the idea of particles but do not use particles explicitly.

The Finite Difference Method was advanced by Harlow & Welch[1] for free surfaces with their Marker-and-Cell method, or MAC, in which the free surface is tracked by a set of marker particles, or cells, in a background mesh. Each cell then simply identifies if it contains fluid or not.

The Finite Volume Method was used by Mingham & Causon[2] with the HLL Riemann solver to model bore waves and dam breaks. Building on the Finite Volume method the Cartesian Cut Cell Method has been successfully developed at Manchester Metropolitan University, where this method has been implemented in their AMAZON codes. Regarding the application to free surface hydrodynamic problems, the method was proved by Qian *et al.*[3] to be accurate in simulating a collapse of a water column against a rectangular obstacle. Later the method was extended by Qian *et al.*[4] to accurately model flows involving two fluids with moving bodies.

The Volume of Fluid (VoF) Method was first proposed by Hirt & Nichols[5] as an improvement on the MAC method to track the free surface and they compared their numerical results with the experimental results of Martin & Moyce[6] who had earlier studied the collapse of a water column on perspex. The European Research Community on Flow, Turbulence and Combustion (ERCOFTAC) Special Interest Group for Smoothed Particle Hydrodynamics, called SPHERIC, use a study by Kleefsman *et al.*[7] as a test case involving free surfaces which studied the collapse of a column of water against an object. Kleefsman *et al.*[7] had used a VoF method to simulate that collapse. Greaves[8] combined both the Finite Volume and Volume of Fluid methods to model breaking waves. Wang *et al.*[9] built on the work of Greaves and Borthwick[10] in using Finite Volume, VoF and quadtrees to refine the background mesh close to the free surface.

The Moving Particle Semi-implicit Method was developed by Koshizuka and Oka[11] and successfully simulated the collapse of a small water column in a tank.

Archibald[12] recently implemented the Boundary Element Method on a single GPU and accurately simulated a variety of wave interactions with structures.

To decrease the execution time of code simulating a particular free surface problem the Computational Fluid Dynamics community has been using High Performance Computing facilities across the world. In the UK The Meteorological Office uses several high performance computers connected in a cluster to predict the weather and climate. Such high performance computers are relatively large and expensive, both to purchase and maintain. But recently a type of processor that was initially developed only for graphics processing has been receiving interest from the scientific community. High Performance Computers made from such processors are much cheaper and smaller than the high performance computers that The Meteorological Office, for example, use, and initial studies of implementing scientific software on graphics processing units (GPUs) indicates that they offer great potential, both in computing power and in reducing the financial costs of initial purchase, maintenance, storage and energy consumption.

In the UK some of the first academics to look at using GPUs were in CFD and related applications. Steve Gratton has been looking at implementing codes for cosmological simulations[13], while his colleagues in the Engineering Department at Cambridge Tobias Brandvik and Graham Pullan have been implementing mesh-based CFD codes on GPUs[14][15]. GPUs are now being taken seriously by the CFD community. A particle-based CFD algorithm on GPUs has been proposed by Westphal[16], while Griebel *et al.*[17] have reported the implementation of a 3D two-phase solver for the incompressible Navier-Stokes Equations on multiple GPUs. Astrophysical implementations of SPH on GPUs include those of Berentzen[18] and the award-winning work of Spurzem *et al.*[19] in which a simulation was run using upto 170 GPUs over 3 continents.

This thesis will try to answer such questions as

- how can a computer initially designed for graphics processing be used for CFD, and in particular for SPH?
- can SPH code be implemented and executed efficiently on GPUs?
- how can SPH be implemented on multiple GPUs?
- how does GPU performance compare with a cluster of multicore CPUs?

In trying to answer these questions this thesis will look at the NVIDIA Tesla C1060 multiprocessor, its architecture and how its architecture can be used for scientific programming, and how SPH can be implemented on such multiprocessors.

Riemann solvers have also been a topic of interest in CFD due to their ability to accurately resolve shocks. The application of Riemann solvers in SPH will also be examined in this thesis.

This thesis is concerned primarily with the implementation of SPH on multiple GPUs, i.e. it focuses more on the high performance computing aspect of the subject so the novel elements of this thesis are more concerned with the computational engineering implementation of SPH on GPUs. Therefore SPH will be described in some but not great detail when compared to other PhD theses in the field of SPH. Consequently SPH is briefly introduced in one chapter, introducing the basic mathematics and components of SPH algorithms so that readers can then understand how to eventually implement a simple SPH scheme on multiple GPUs. The open source code SPHysics by Gomez-Gesteira *et al.*[20] has been a reference, and also a valuable aid to understanding SPH and its implementation on a CPU, and as such some but not all aspects of SPHysics is covered in the chapter on SPH.

The NVIDIA GPU language CUDA is based on the computing language C, so it is assumed that readers wishing to understand code snippets in this thesis and implement or amend codes using CUDA have a sound knowledge of C.

# Chapter 2

# Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics is a relatively new method in Computational Fluid Dynamics that has its roots in astrophysics. Since its first introduction variants of SPH have been developed, two being Weakly Compressible SPH (WCSPH) and Incompressible SPH (ISPH).

ISPH attempts to solve the Poisson equation for pressure assuming constant density. WCSPH uses an equation of state to calculate pressure from a fluctuating particle density instead of solving the Poisson equation. Lee *et al.*[21] and Lee *et al.*[22] compare the accuracy and performance of these two variants of SPH, describing the differences between the two, particularly the calculation of pressure.

The first section of this chapter will address the mathematical theory of WCSPH, explaining how mathematical expressions, such as divergence, can be manipulated for particles so that differential equations can be written in terms of particles. The remainder of this chapter will then address the components of a SPH algorithm.

## 2.1 What is a Particle in SPH?

This is perhaps the most important question, and is answered by Liu & Liu[23].

A particle is a point in space with properties defined by a smoothing function $W$ centred on that point. A particle can have

- a support domain

- an influence domain

A particle's support domain is defined to be the set of particles that have influence on the value of the properties at that point in space. A particle's influence domain is the set of particles that it interacts with to influence their properties.

### 2.1.1 The Continuous Approximation

A field variable $f(\underline{x})$, such as density, of a particle with position $\underline{x}$ can be written as

$$f(x) = \int_{\Omega} f(\underline{x}^{'})\delta(\underline{x} - \underline{x}^{'})d\underline{x} \tag{2.1}$$

where the Dirac delta function is defined as

$$\delta(\underline{x} - \underline{x}^{'}) = \begin{cases} 1, & \underline{x} = \underline{x}^{'} \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

and $\Omega$ is the support domain of the particle at $\underline{x}$.

If the Dirac delta function is replaced by a smoothing function $W(\underline{x} - \underline{x}^{'}, h)$ then the integral of $f(\underline{x})$ becomes

$$f(\underline{x}) = \int_{\Omega} f(\underline{x}^{'})W(\underline{x} - \underline{x}^{'}, h)d\underline{x} \tag{2.3}$$

where $h$ is the smoothing length defining the influence domain of the smoothing function $W$.

This is the continuous representation of a particle, and is written in the SPH convention

$$< f(\underline{x}) >= \int_{\Omega} f(\underline{x}^{'})W(\underline{x} - \underline{x}^{'}, h)d\underline{x} \tag{2.4}$$

The smoothing function $W$ should satisfy the following seven conditions.

1. unity

2. compact support

3. positivity
4. decay
5. Delta function
6. symmetry
7. smoothness

The unity condition states that

$$\int_{\Omega} W(\underline{x} - \underline{x}^{'}, h) d\underline{x} = 1 \tag{2.5}$$

The compact condition states that

$$W(\underline{x} - \underline{x}^{'}, h) = 0 \text{ when } |\underline{x} - \underline{x}^{'}| > kh \tag{2.6}$$

The positivity condition states that for any particle at position $\underline{x}^{'}$ in the support domain of the particle at position $\underline{x}$

$$W(\underline{x} - \underline{x}^{'}, h) \geq 0 \tag{2.7}$$

The decay condition states that the smoothing function should be monotonically decreasing.

The Delta function condition states that

$$\lim_{h \to 0} W(\underline{x} - \underline{x}^{'}, h) = \delta(\underline{x} - \underline{x}^{'}) \tag{2.8}$$

where the Dirac delta function $\delta$ is defined in Equation (2.2).

The symmetry condition states that the smoothing function should be even.

The smoothness condition states that the smoothing function should be sufficiently smooth.

### 2.1.2 Error in Particle Approximation

The error in the particle approximation can be found by using a Taylor series expansion of $f(\underline{x})$ around $\underline{x}^{'}$. The truncated Taylor expansion gives

$$f(\underline{x}) = f(\underline{x}') + (\underline{x} - \underline{x}^{'})f^{'}(\underline{x}^{'}) + O(\underline{x} - \underline{x}^{'})^2 \tag{2.9}$$

Substituting this into Equation (2.4) gives

$$< f(\underline{x}) >= \int_{\Omega} [f(\underline{x}') + (\underline{x} - \underline{x}^{'})f^{'}(\underline{x}^{'}) + O(\underline{x} - \underline{x}^{'})^2]W(\underline{x} - \underline{x}^{'}, h)d\underline{x} \tag{2.10}$$

When Equation (2.10) is expanded, the second term becomes

$$f^{'}(\underline{x}^{'}) \int_{\Omega} (\underline{x} - \underline{x}^{'})W(\underline{x} - \underline{x}^{'}, h)d\underline{x} \tag{2.11}$$

Assuming that the smoothing function $W$ is even with respect to $\underline{x}$ then the integral in this term equals zero because $(\underline{x} - \underline{x}^{'})W(\underline{x} - \underline{x}^{'}, h)$ will be odd. So removing this term, and using the unity condition Equation (2.5) and the property that

$$\underline{x} - \underline{x}^{'} = O(h) \tag{2.12}$$

the approximation error becomes

$$< f(\underline{x}) >= f(\underline{x}) + O(h^2) \tag{2.13}$$

### 2.1.3 The Summation Approximation

The particle approximation, which can be made by replacing the continuous integral by a summation over a finite set of $N$ particles in a support domain and replacing the infinitesimal volume $d\underline{x}^{'}$ of each particle in the support domain with the finite volume of the particle, is

$$f(\underline{x}) = \sum_{j=1}^{N} f(\underline{x}^{'})W(\underline{x} - \underline{x}^{'}, h)V_j \tag{2.14}$$

But $V_j = m_j/\rho_j$, where $m$ is the mass and $\rho$ is the density, so the particle is then approximated by

$$f(\underline{x}) = \sum_{j=1}^{N} \frac{m_j}{\rho_j} f(\underline{x}^{'})W(\underline{x} - \underline{x}^{'}, h) \tag{2.15}$$

If $W$ is a differentiable function then the particle approximation can be differentiated exactly to give

$$\frac{\partial f(\underline{x})}{\partial x} = \sum_{j=1}^{N} \frac{m_j}{\rho_j} f(\underline{x}^{'}) \frac{\partial W(\underline{x} - \underline{x}^{'}, h)}{\partial x} \tag{2.16}$$

Thus

$$\nabla f_i = \sum_{j=1}^{N} \frac{m_j}{\rho_j} f_j \nabla W_{ij} \tag{2.17}$$

where $\nabla$ is the Gradient operator.

But as Monaghan[24] pointed out, this does not vanish if $f$ is constant. To guarantee that this does vanish the first derivative of the particle approximation can be found from using the identity

$$\frac{\partial f}{\partial x} = \frac{1}{\Phi}\Big(\frac{\partial(\Phi f)}{\partial x} - f\frac{\partial \Phi}{\partial x}\Big) \tag{2.18}$$

where $\Phi$ is any differentiable function. Then for any particle

$$\frac{\partial f_i}{\partial x} = \frac{1}{\Phi_i}\Big(\frac{\partial(\Phi_i f_i)}{\partial x} - f_i\frac{\partial \Phi_i}{\partial x}\Big) \tag{2.19}$$

Substituting for the partial derivatives with Equation (2.16) gives

$$\frac{\partial f_i}{\partial x} = \frac{1}{\Phi_i}\Big(\sum_{j=1}^{N}\frac{m_j}{\rho_j}\Phi_j f_j \frac{\partial W}{\partial x} - f_i \sum_{j=1}^{N}\frac{m_j}{\rho_j}\Phi_j \frac{\partial W}{\partial x}\Big) \tag{2.20}$$

which simplifies to

$$\frac{\partial f_i}{\partial x} = \frac{1}{\Phi_i}\sum_{j=1}^{N}\frac{m_j}{\rho_j}\Phi_j (f_j - f_i)\frac{\partial W}{\partial x} \tag{2.21}$$

from which follows

$$\nabla f_i = \frac{1}{\Phi_i}\sum_{j=1}^{N}\frac{m_j}{\rho_j}\Phi_j (f_j - f_i)\nabla W_{ij} \tag{2.22}$$

## 2.2 The Particle Approximation of the Navier-Stokes Equations

This thesis examines the implementation of the weakly compressible SPH method. The compressible Navier-Stokes Equations are thus the governing equations, but for simplicity they shall be reduced to the Euler equations plus a viscous term $\Theta$, which will be discussed later, and an external forces vector $\underline{F}$.

$$\frac{d\rho}{dt} = -\rho\nabla\cdot\underline{v} \tag{2.23}$$

$$\frac{d\underline{v}}{dt} = -\frac{1}{\rho}\nabla P + \underline{\theta} + \underline{F} \tag{2.24}$$

The particle approximation of Equation (2.23), the continuity equation, can be found by using Equation (2.21) with $\Phi = \rho$, the density, and the variable $f$ taken to be the components of the velocity $\underline{v} = (u, v, w)$ then

$$\frac{\partial u_i}{\partial x} = \frac{1}{\rho_i}\sum_{j=1}^{N}\frac{m_j}{\rho_j}\rho_j (u_j - u_i)\frac{\partial W}{\partial x} \tag{2.25}$$

$$\frac{\partial v_i}{\partial y} = \frac{1}{\rho_i}\sum_{j=1}^{N}\frac{m_j}{\rho_j}\rho_j(v_j - v_i)\frac{\partial W}{\partial y} \tag{2.26}$$

$$\frac{\partial w_i}{\partial z} = \frac{1}{\rho_i}\sum_{j=1}^{N}\frac{m_j}{\rho_j}\rho_j(w_j - w_i)\frac{\partial W}{\partial z} \tag{2.27}$$

Adding these three equations together gives

$$\nabla \cdot \underline{v}_i = \frac{1}{\rho_i}\sum_{j=1}^{N} m_j\Big((u_j - u_i)\frac{\partial W}{\partial x} + (v_j - v_i)\frac{\partial W}{\partial y} + (w_j - w_i)\frac{\partial W}{\partial z}\Big) \tag{2.28}$$

The continuity equation, Equation (2.23), then becomes

$$\frac{d\rho_i}{dt} = \sum_{j=1}^{N} m_j \underline{v}_{ij} \cdot \nabla_i W_{ij} \tag{2.29}$$

where $\underline{v}_{ij} = \underline{v}_i - \underline{v}_j$.

Similarly, by setting $\Phi = 1$ in Equation (2.21) the continuity equation becomes

$$\frac{d\rho_i}{dt} = \rho_i \sum_{j=1}^{N} \frac{m_j}{\rho_j}\underline{v}_{ij} \cdot \nabla_i W_{ij} \tag{2.30}$$

Monaghan[24] argues that in multiphase problems if the density ratios are $\leq 2$ then either Equation (2.29) or Equation (2.30) can be used, otherwise Equation (2.30) is more accurate.

The momentum equation, Equation (2.24), can be written in particle form, ignoring the viscosity term $\underline{\Theta}$ and external forces $\underline{F}$ for now, by using Equation (2.17) substituting pressure $P$ to give

$$\nabla P_i = \sum_{j=1}^{N} \frac{m_j}{\rho_j} P_j \nabla W_{ij} \tag{2.31}$$

This then gives the momentum equation

$$\frac{d\underline{v}_i}{dt} = -\frac{1}{\rho_i}\sum_{j=1}^{N} \frac{m_j}{\rho_j} P_j \nabla W_{ij} \tag{2.32}$$

But this does not conserve linear or angular momentum. Conservation was ensured by

Gingold and Monaghan[25] using

$$\frac{d\underline{v}_i}{dt} = -\sum_{j=1}^{N} m_j \Big(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\Big) \nabla W_{ij} \tag{2.33}$$

This equation can also be derived by substituting pressure $f = P$ and $\Phi = 1/\rho$ in Equation (2.18) to give

$$\nabla P = \rho\Big(\nabla\Big(\frac{P}{\rho}\Big) - P\nabla\Big(\frac{1}{\rho}\Big)\Big) \tag{2.34}$$

Applying the quotient rule to the second term gives

$$\nabla P = \rho\Big(\nabla\Big(\frac{P}{\rho}\Big) + \frac{P}{\rho^2}\nabla\rho\Big) \tag{2.35}$$

Then using Equation (2.22) again, first for $f = P/\rho$ and then for $f = \rho$ the particle momentum equation, Equation (2.33), is obtained.

So the final particle approximation of the momentum equation is

$$\frac{d\underline{v}_i}{dt} = -\sum_{j=1}^{N} m_j \Big(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2}\Big) \nabla W_{ij} + \underline{\Theta} + \underline{F} \tag{2.36}$$

The viscous term $\underline{\Theta}$ has been modelled with particles by several authors. Monaghan & Gingold[26] proposed the simple viscous term $\Pi_{ij}$ which conserves linear and angular momentum and works well under most circumstances.

$$\Pi_{ij} = -\nu_{ij}\phi_{ij} \tag{2.37}$$

where

$$\nu_{ij} = \frac{\alpha \overline{h}_{ij} \overline{c}_{ij}}{\overline{\rho}_{ij}} \tag{2.38}$$

where $c$ is the speed of sound,

$$\phi_{ij} = \frac{\underline{v}_{ij} \cdot \underline{x}_{ij}}{\left|\underline{x}_{ij}\right|^2 + \epsilon \overline{h}_{ij}^2} \tag{2.39}$$

$$\overline{c}_{ij} = \frac{1}{2}(c_i + c_j) \tag{2.40}$$

$$\overline{\rho}_{ij} = \frac{1}{2}(\rho_i + \rho_j) \tag{2.41}$$

$$\overline{h}_{ij} = \frac{1}{2}(h_i + h_j) \tag{2.42}$$

$$\underline{v}_{ij} = \underline{v}_i - \underline{v}_j \tag{2.43}$$

$$\underline{x}_{ij} = \underline{x}_i - \underline{x}_j \tag{2.44}$$

The parameter $\alpha$ is typically set to equal 1 but must be tuned for each individual simulation, and $\epsilon$ is typically set to equal 0.01

This was later amended by Monaghan[27] such that

$$\nu_{ij} = \frac{\overline{h}_{ij}}{\overline{\rho}_{ij}}\Big(\alpha\overline{c}_{ij} - \beta\overline{h}_{ij}\phi_{ij}\Big) \tag{2.45}$$

Note that this artificial viscosity has two parameters to tune, $\alpha$ and $\beta$.

A similar type of artificial viscosity was proposed by Cleary[28] with

$$\Pi_{ij} = -\frac{\xi}{\rho_i\rho_j}\frac{4\mu_i\mu_j}{(\mu_i + \mu_j)}\frac{\underline{v}_{ij}\cdot\underline{x}_{ij}}{(|\underline{x}_{ij}|^2 + \epsilon\overline{h}_{ij}^2)} \tag{2.46}$$

where $\mu$ is the dynamic visosity of the particle. This allows multiphase problems to be simulated. Cleary showed that the parameter $\xi$ should be approximately 5 and is independent of $\mu$.

When this type of artificial viscosity is used the momentum equation becomes

$$\frac{d\underline{v}_i}{dt} = -\sum_{j=1}^{N} m_j\Big(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij}\Big)\nabla W_{ij} + \underline{F} \tag{2.47}$$

where in Equation (2.36) the viscosity term

$$\underline{\Theta} = -\sum_{j=1}^{N} m_j\Pi_{ij}\nabla W_{ij} \tag{2.48}$$

## 2.3 Density

The density can be calculated using the rate of change of density, as in Equation (2.29) or Equation (2.30). Another method is to use the summation approach, in which the density is found by substituting $\rho$ for $f$ in Equation (2.15), to give

$$\rho_i = \sum_j m_j W_{ij} \tag{2.49}$$

## 2.4 Smoothing Functions

There are a number of smoothing functions proposed in the SPH literature, and Liu & Liu[23] dedicate a whole chapter in their book on deriving smoothing functions. The smoothing functions that have been used here are now given. In this section the term $q = |\underline{x}_i - \underline{x}_j|/h$.

### 2.4.1 Gaussian

In their seminal paper, Gingold & Monaghan[29] proposed the following smoothing function, the Gaussian.

$$W(q,h) = \alpha e^{-q^2} \tag{2.50}$$

where $\alpha = 1/(h\sqrt{\pi})^D$ for dimension $D = 2, 3$.

This smoothing function does not satisfy the compactness condition, but because it approaches zero very quickly is virtually compact.

### 2.4.2 Parshikov Cubic Spline

Parshikov *et al.*[30] used a cubic spline for their work using approximate Riemann solvers in SPH. This smoothing function was used when looking at implementing the Parshikov model.

$$W_{ij} = \alpha \begin{cases} 1 - 1.5q^2 + 0.75q^3 & \text{if } 0 \le q < 1, \\ 0.25(2-q)^3 & \text{if } 1 \le q < 2, \\ 0 & \text{otherwise} \end{cases} \tag{2.51}$$

where $\alpha = 1/(0.7\pi h^2), 1/(\pi h^3)$ for 2 and 3 dimensions respectively.

### 2.4.3 Wendland Quintic Spline

The Wendland quintic spline was used for the implementation of the Vila SPH algorithm on the GPU, and takes the simple form

$$W = \alpha(1 - \frac{q}{2})^4(2q+1) \tag{2.52}$$

where $\alpha = 7/(4\pi h^2), 21/(16\pi h^3)$ for 2 and 3 dimensions respectively. This smoothing function was examined in some detail by Robinson[31] in a chapter on particle clumping.

## 2.5 Smoothing Length

The smoothing length of the particle can be either constant or variable. The purpose of varying the smoothing length is to maintain the number of particles in the support domain.

A very simple method to vary the smoothing length *h* is

$$h_i = \sqrt[\nu]{\frac{m_i}{\rho_i}} \tag{2.53}$$

where $\nu$ is the number of dimensions.

Sigalotti *et al.*[32] proposed a more complex method to vary the smoothing length which uses the summation density, as opposed to the continuity density, in a three phase procedure.

## 2.6 Boundary Conditions

The modeling of the physical boundary continues to be a problem in SPH. Particle penetration of the boundaries can also completely destroy the credibility of a SPH simulation. Due to this a number of boundary treatments have been proposed, and shall be covered briefly here. A later chapter focusing explicitly on boundary treatment will explore these boundary treatments in more detail, and will propose a new boundary treatment that when implemented on the GPU can significantly accelerate computation.

### 2.6.1 On-Boundary Particles

Virtual particles can be used to model solid boundaries. One method of implementing virtual particles is to use a Lennard-Jones repulsive force introduced by Monaghan[33] in which a virtual particle placed on the boundary exerts a repulsive force similar to that in electrodynamics on any fluid particle within a defined radius of the virtual particle. Monaghan & Kos[34] proposed another form of virtual particle on the boundary using the normal and tangential velocity of a particle close to the boundary. This was amended later by Monaghan *et al.*[35]. More recently yet another boundary particle method was proposed by Monaghan & Kajtar[36].

Other methods of modeling the boundary with particles on the boundary have been proposed by Morris *et al.*[37], and by Dalrymple & Knio[38] with their properties examined by Crespo *et al.*[39].

### 2.6.2 Ghost Particles

Another popular method of implementing virtual particles is the ghost particle, of which there are several variants. Ghost particles do not exist on the boundary as described above, but are created with a temporary existence when required to create a virtual boundary. Ghost boundary particle methods have been proposed or used by Colagrossi[40], Cummins & Rudman[41], and Ferrari *et al.*[42].

### 2.6.3 Hybrid Boundary Treatment

Liu & Liu[23] have proposed a hybrid of ghost and repulsive virtual particles, in which a layer of particles on the boundary is reinforced by ghost particles which mirror the fluid particles. A similar approach was taken by Lo & Shao [43].

A different hybrid boundary treatment has been used by Violeau & Issa[44] and Lee *et al.*[21] in which the boundary is modeled by four layers of particles which do not move but assume properties that are dependent on the fluid particle they are interacting with.

### 2.6.4 Wall Functions

Harada *et al.*[45] proposed a weighted wall function, in which for a set of perpendicular distances from the wall the contribution to the fluid particle density and viscosity from the wall is precomputed. During the simulation this contribution to the fluid particle from the wall is calculated as a linear interpolation between two of the precomputed values depending only on perpendicular distance of the fluid particle from the wall.

## 2.7 Corrections in SPH

Several proposals have been made to improve the accuracy of WCSPH. These corrections include the following.

### 2.7.1 Density Correction

There are two ways to calculate density in SPH. The continuity approach, that uses a rate of change continuity equation, an example of which is Equation (2.29). The second is the summation approach, an example of which is Equation (2.49), which respects the essence of SPH, but requires more computation than the continuity approach. So to accelerate a simulation the continuity approach has been preferred. However, the continuity approach does not conserve mass, unlike the summation approach. So when using the continuity approach it is useful to occasionally correct the density to conserve mass. Two density corrections have been proposed.

With the Shepard Filter[46], after every N iterations the density for each fluid particle is reset as

$$\rho_i = \sum_j m_j \tilde{W}_{ij} \tag{2.54}$$

where

$$\tilde{W}_{ij} = \frac{W_{ij}}{\sum_j \frac{m_j}{\rho_j} \tilde{W}_{ij}} \tag{2.55}$$

The Moving Least Squares[47] density correction is 1st order accurate and is much more complex than the Shepard Filter, requiring the inversion of a matrix for every particle every N time steps. This correction has been implemented by Colagrossi[40]. As with the Shepard filter, with MLS after every N time steps the density is reset with Equation (2.54), but with MLS, and for simplicity in 2 dimensions,

$$\tilde{W}_j(\underline{x}_i) = [\beta_0(\underline{x}_i) + \beta_1(\underline{x}_i)(x_i - x_j) + \beta_2(\underline{x}_i)(y_i - j_j)]W_{ij} \tag{2.56}$$

where

$$\beta(\underline{\mathbf{x}}_{\mathbf{i}}) = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{pmatrix} = \mathbf{A}^{-1}(\underline{\mathbf{x}}_{\mathbf{i}}) \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.57}$$

$$A(\underline{x}_i) = \sum_j W_j(\underline{x}_i) \tilde{A}_{ij} \tag{2.58}$$

$$\tilde{A}_{ij} = \begin{bmatrix} 1 & (x_i - x_j) & (y_i - y_j) \\ (x_i - x_j) & (x_i - x_j)^2 & (x_i - x_j)(y_i - y_j) \\ (y_i - y_j) & (y_i - y_j)(x_i - x_j) & (y_i - y_j)^2 \end{bmatrix} \tag{2.59}$$

### 2.7.2 Kernel Correction

Bonet & Lok[48] described two methods for correcting particle deficiency, particularly at boundaries and free surfaces where the support domain of a fluid particle is truncated, before proposing combining both.

The first method involves correcting the kernel gradient via a correction matrix $L$. The corrected kernel gradient $\tilde{\nabla} W$ is used in the SPH equations where

$$\tilde{\nabla} W_j(\underline{x}_i) = L_i \nabla W_j(\underline{x}_i) \tag{2.60}$$

and

$$L_i = \Big(\sum_{j=1}^{N} \frac{m_j}{\rho_j} W_j(\underline{x}_i) \otimes (\underline{x}_j - \underline{x}_i)\Big)^{-1} \tag{2.61}$$

The second method involves correcting the kernel itself, but they dismiss this method as computationally expensive and instead suggest a simpler linear correction for the velocity

$$\underline{v}(\underline{x}) = \frac{\sum_{j=1}^{N} V_j \underline{v}_j W_j(\underline{x})}{\sum_{j=1}^{N} V_j W_j(\underline{x})} \tag{2.62}$$

where $V_j = m_j/\rho_j$, but with the caveat that this does not preserve angular momentum.

### 2.7.3 Tensile Correction

In SPH Tensile Instability leads to particles forming clumps. Swegle[49] identified a tensile instability in SPH for solid mechanics. Monaghan[50] proposed a simple correction designed to create a repulsive force between two particles very close together which is added to the momentum equation. This Monaghan tensile correction takes the form of $Rf_{ij}^n$, and if using one of the artificial viscosities discussed above, the momentum equation becomes

$$\frac{d\underline{v}_i}{dt} = -\sum_{j=1}^{N} m_j \Big(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij} + Rf_{ij}^n\Big) \nabla W_{ij} + \underline{F} \tag{2.63}$$

For a cubic spline $n = 4$. The function $f_{ij}$ has the form

$$f_{ij} = \frac{W(\underline{r}_{ij})}{W(\Delta P)} \tag{2.64}$$

where $\underline{r}_{ij}$ is the vector between two particles, and $\Delta P$ is the initial particle spacing. The variable $R = R_i + R_j$, where

$$R_k = \begin{cases} 0.01P_k/\rho_k^2, & P_k > 0 \\ 0.2|P_k|/\rho_k^2, & P_k < 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.65}$$

for $k = i, j$.

### 2.7.4 XSPH

Monaghan[51] proposed a term to add to the velocity during integration which helps to keep a particle moving with a speed close to average speed of those particles close to it, which should reduce particle penetration. The XSPH correction is

$$\underline{XSPH}_i = \epsilon \sum_j \frac{m_j}{(\rho_i + \rho_j)} (\underline{v}_j - \underline{v}_i) W_{ij} \tag{2.66}$$

with a typical value of $\epsilon$=0.5, though Liu & Liu[23] recommend $\epsilon$=0.3. The integration of the position then becomes

$$\frac{d\underline{x}_i}{dt} = \underline{v}_i + \underline{XSPH}_i \tag{2.67}$$

### 2.7.5 Hughes and Graham Correction

Hughes & Graham[52] proposed a correction for the density of boundary particles when using boundary treatments in which the density of a boundary particle can evolve, such as in the Dalrymple & Knio boundary treatment. This correction involves calculating the density of the boundary paricles every 20th time step and either resetting the density of the boundary particle to the reference density if it is found to be less than the reference density, or renormalizing the density using either the Shephard or MLS density correction method described above.

## 2.8 Equations of State

To calculate pressure an equation of state (EoS), a function of the density, is used. The two most common equations of state for water are the Tait and Morris equations of state. If using air particles in a SPH simulation, the ideal gas equation of state is frequently used for the air particles.

### 2.8.1 Tait EoS

The Tait equation of state[53] takes the form

$$P_i = B\Big(\Big(\frac{\rho_i}{\rho_0}\Big)^{\gamma} - 1\Big) \tag{2.68}$$

The speed of sound for each particle is then

$$c_i = c_o\Big(\frac{\rho_i}{\rho_0}\Big)^{3} \tag{2.69}$$

where $\gamma = 7$, $\rho_0$ is the reference density of water, and $B = c_0^2\rho_0/\gamma$ where $c_0$ is the speed of sound at the reference density.

### 2.8.2 Morris EoS

The Morris equation of state[37] is

$$P_i = c_0^2(\rho_i - \rho_0) \tag{2.70}$$

where $\rho_o$ is the reference density of water, and $c_0$ is the speed of sound at the reference density $\rho_o$. Morris *et al.* state that the parameter $c_o$ can require some degree of tuning, but once tuned is constant for all particles.

## 2.9 Time Step and Integration Schemes

The following integration and time step schemes have been used in the implementations of SPH algorithms used for this thesis.

### 2.9.1 Leapfrog

Leapfrog integration is a second order scheme, but also has the interesting and useful property of time reversibility, which guarantees conservation of energy and momentum. The term leapfrog derives from the first integration in which the position and velocity integrate out of synchronisation, with all variables except position integrating a half time step while position integrates a full time step. All subsequent integrations then use the full time step for all variables. So from the open source code in the book by Liu & Liu[23], the scheme is written as, for the first integration

$$\rho_i^{1/2} = \rho^0 + \frac{\Delta t}{2}\frac{d\rho_i}{dt} \tag{2.71}$$

$$\underline{v}_i^{1/2} = \underline{v}_i^0 + \frac{\Delta t}{2}\frac{d\underline{v}_i}{dt} + \underline{XSPH}_i^0 \tag{2.72}$$

$$\underline{x}_i^1 = \underline{x}_i^0 + \Delta t \underline{v}_i^{1/2} \tag{2.73}$$

For all subsequent integrations, the scheme is written as

$$\rho_i^{t+1/2} = \rho^{t-1/2} + \Delta t\frac{d\rho_i}{dt} \tag{2.74}$$

$$\underline{v}_i^{t+1/2} = \underline{v}_i^{t-1/2} + \Delta t\frac{d\underline{v}_i}{dt} + \underline{XSPH}_i^{t-1/2} \tag{2.75}$$

$$\underline{x}_i^{t+1} = \underline{x}_i^t + \Delta t \underline{v}_i^{t+1/2} \tag{2.76}$$

When this integration scheme is used for this thesis the time step is constant.

### 2.9.2 Predictor-Corrector

The following 2nd order Predictor-Corrector scheme has been implemented for this thesis from the SPHysics[20] open source code, with the vector $Q = (\rho, \underline{x}, \underline{v})$. The prediction uses values at time step *n*.

$$Q^{n+1/2} = Q^n + \frac{\Delta t}{2}\frac{dQ^n}{dt} \tag{2.77}$$

and if using an equation of state for pressure, $P^{n+1/2} = P(\rho^{n+1/2})$. The predictions are then corrected using the values just calculated for time step $n + 1/2$.

$$Q^{n+1/2} = Q^n + \frac{\Delta t}{2}\frac{dQ^{n+1/2}}{dt} \tag{2.78}$$

The final integration to time step $n + 1$ is

$$Q^{n+1} = 2Q^{n+1/2} - Q^n \tag{2.79}$$

and $P^{n+1} = P(\rho^{n+1})$ with $P(\rho)$ being the Tait or Morris Equation of State.

The size of the time step $\delta t$ itself is calculated as

$$\delta t = \text{CFL} \times \min(\delta t_f, \delta t_{visc}) \tag{2.80}$$

where *CFL* is the CFL number, and

$$\delta t_f = \min_i(\sqrt{h/|f_i|}) \tag{2.81}$$

in which *h* is the smoothing length, $f_i$ is the total force per unit mass exerted on particle *i*, and

$$\delta t_{visc} = \min_i \left( h/(c_{max} + \max_j \frac{h\underline{r}_{ij} \cdot \underline{v}_{ij}}{r_{ij}^2}) \right) \tag{2.82}$$

in which *h* is the smoothing length, $c_{max}$ is the maximum speed of sound over all fluid particles, and the term involving the dot product is the viscous term proposed in Equation (2.39) and for each fluid particle the maximum value of this term is calculated from each interaction and the maximum of these is used in the calculation of the time step.

### 2.9.3 Runge-Kutta Schemes

This integration scheme is used by Ferrari *et al.*[42] in which the governing SPH equations are a system of ODEs of the form

$$\frac{dQ}{dt} = S(Q, t) \tag{2.83}$$

A 3rd Order TVD Runge-Kutta scheme has been implemented in which the vector $Q = (\rho, \underline{x}, \underline{v})$, and

$$Q^{n+1} = Q^n + \frac{\Delta t}{4}(k_1 + 3k_3) \tag{2.84}$$

where

$$k_1 = S(Q^n, t^n) \tag{2.85}$$

$$k_2 = S(Q^n + \frac{1}{3}\Delta t k_1, t^n + \frac{1}{3}\Delta t) \tag{2.86}$$

$$k_3 = S(Q^n + \frac{2}{3}\Delta t k_2, t^n + \frac{2}{3}\Delta t) \tag{2.87}$$

The time step is calculated by

$$\Delta t = \sigma \min_i \left(\frac{h_i}{\alpha}\right), \alpha = \max_i(c_i, |\underline{v}_i|) \tag{2.88}$$

where $\sigma$ is the CFL number, the smoothing length *h* is calculated by Equation (2.53), and

$$c_i = \sqrt{\gamma \frac{B}{\rho_0}\left(\frac{\rho_i}{\rho_0}\right)^{(\gamma-1)}} \tag{2.89}$$

where $\gamma$=7, and $\rho_0$ is the reference density of water.

A 4th Order Runge-Kutta (RK4) scheme has also been implemented in SPH. Oger *et al.*[54] proposed an SPH algorithm and used different integration schemes, and found that there can be advantages in using schemes of order greater than two, with the larger time step dominating the extra computation. Colagrossi & Landrini[40] also implemented a RK4 scheme but also used the Moving-Least-Square density reinitialization described above.

## 2.10 Finding Particle Interactions

A naive method of finding particle interactions is to consider every particle and investigate if it interacts with all other particles in the simulation. The computational effort for this is $O(N^2)$, where N is the number of particles.

However, the definition of a particle can be used to reduce the computational effort in finding particle interactions. Depending on the smoothing function being used, the particles can be assigned to cells in a simple background grid. For example, if the smoothing function states that its value is zero beyond $\beta h$ from the centre of the particles then when looking for a particle's support domain only neighbouring cells of the cell containing that particle need be searched if the cells are set to be of size $\beta h$ x $\beta h$ in 2 dimensions. Usually $\beta = 2$, but some definitions for smoothing functions have $\beta > 2$. The computational effort for this method is $O(N)$. Variable smoothing lengths can be easily accounted for with this background mesh by resizing the cells so that they are of size $\beta h_{max}$, where $h_{max}$ is the maximum smoothing length of all particles. This will lead to some particles with smaller smoothing lengths checking for interactions with particles in neighbouring cells that are beyond a distance $\beta h_i$, but this method will still be more efficient.

# Chapter 3

# GPU Programming

The main manufacturer of graphics processing units is NVIDIA. Early scientific programs executed on these GPUs used graphics programming languages such as OpenGL and DirectX, and the program had to be written such that the problem being solved could be written using the functionality of these languages. But then in 2006 NVIDIA released a language to run on some but not all of their latest processors. These later NVIDIA processors did not have the capability to produce graphics, but their architecture was derived from their graphics processors, and they were designed specifically for high performance computing. The language NVIDIA created is called Compute Unified Device Architecture, or CUDA. CUDA allows parallel computation of highly parallel algorithms using multiple blocks of lightweight threads. Some but not all NVIDIA graphics processors can execute hybrid graphics/CUDA code to run interactive applications. Processors that can run CUDA code only are the Tesla and more recently the Fermi groups of NVIDIA processors. This chapter will introduce the architecture of the NVIDIA GPU, and also address the functionality of CUDA that has been used in the SPH codes developed for this thesis.

## 3.1 The Architecture of the NVIDIA Graphics Processing Unit

### 3.1.1 Memory Hierarchy

The NVIDIA processor used for the work in this thesis is the NVIDIA Tesla 10 series (T10). This processor consists of 240 cores, divided into 30 blocks of 8 cores. Such a block of 8 cores is called a thread processor array (TPA), but is also called a multiprocessor in some NVIDIA literature. Each core in a TPA is connected to a block of 16 KB shared memory which is shared between the 8 cores in the TPA. The clock speed of each core is 1.3 GHz. This processor forms the basis for the NVIDIA C1060 which has 4 GB of hard memory operating at 800 Mhz. An illustration of a general NVIDIA Tesla processor is shown in Figure 3.1[55]. For the T10, in Figure 3.1 the variables M = 8 and N = 30.

The T10 was one of the first NVIDIA Tesla GPUs with double precision, but all the work done for this thesis was done using single precision.

Also shown in Figure 3.1 are the different kinds of memory on a NVIDIA GPU; device, texture, shared, constant and registers. Each has a different purpose, speed and size. For the NVIDIA C1060, which incorporates the T10,

- the device memory is 4 GB and its purpose is to hold the bulk of the data to be processed.
- the texture memory is cached device memory and is between 6 KB and 8 KB.
- the shared memory is 16 KB for each TPA, and its purpose is to hold reusable data close to the registers for fast recycling of data.
- the constant memory is for data that is constant for the lifetime of the application and is 64 KB.
- each core has 2048 registers.

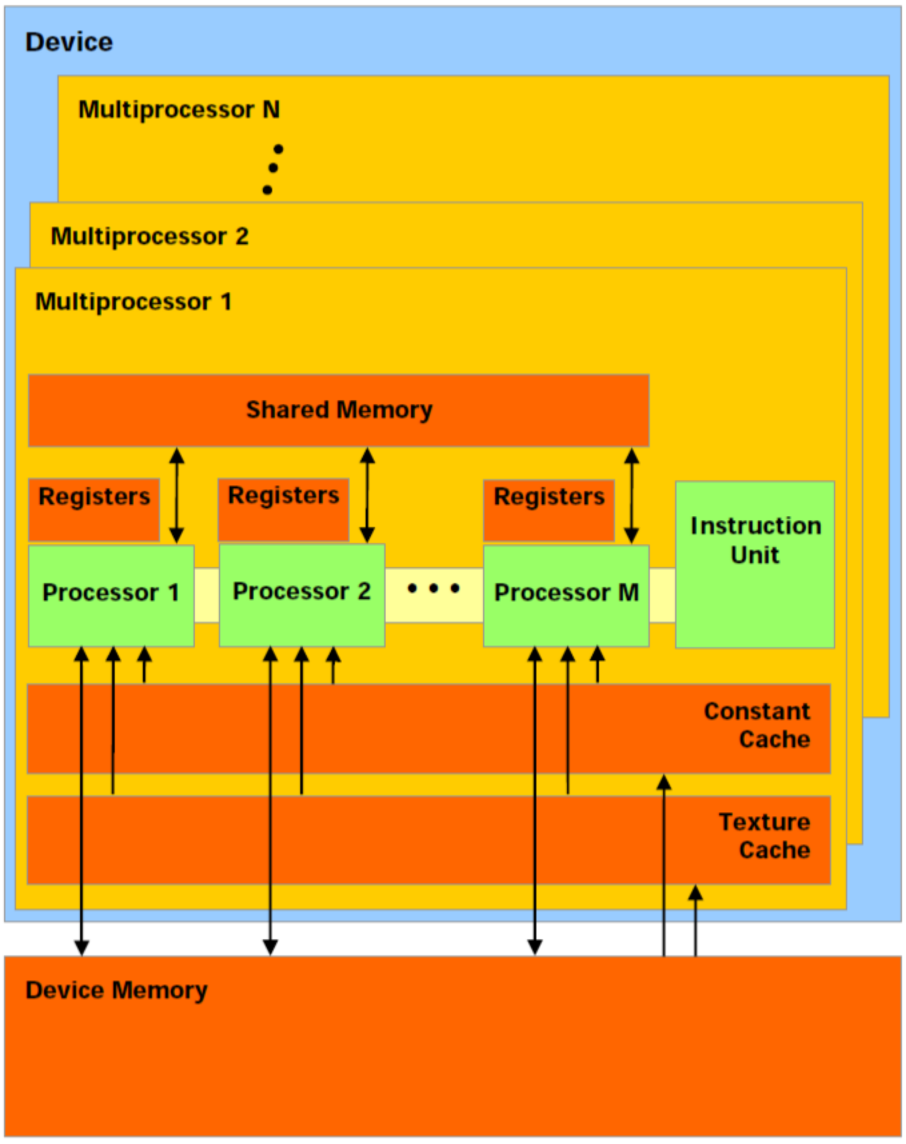


Figure 3.1: The architecture of the NVIDIA Tesla GPU

The shared memory is arranged into 16 banks. This number 16 is important because it is the size of a half warp. A warp is 32 threads. The parallelism of the GPU comes from the warp. All threads in a warp execute in parallel if they all execute the same code, so if there is a logical statement, such as an *if-else* or *switch* statement which could force threads in a warp to execute different paths of the same piece of code, then those threads in that warp will execute in serial, thus losing their parallelism. This execution of the warp in serial is called Warp Divergence. And with the core frequency of 1.3 GHz on the GPU and modern CPUs operating at 3 GHz, it is advisable to avoid Warp Divergence as much as possible, but sometimes this is impossible.

### 3.1.2 Coalescing

NVIDIA recommend that CUDA programs use the shared memory together with the concept of coalescing to achieve maximum efficiency[55]. The efficiency of the shared memory is due to its proximity to the registers. But to transfer data into the shared memory that data must first be read from the large and slow global memory. Coalescing allows multiple items of data to be read from global memory in one instruction. NVIDIA state that one read instruction from global memory can take between 400 and 600 clock cycles, which is slow, so any opportunity to read multiple items of data from global memory into shared memory in one instruction would be very efficient and should be used as often as possible.

NVIDIA have introduced a concept called Compute Capability to describe the functionality of their GPUs. The first Tesla GPUs, C870 and D870, were of Compute Capability 1.0. The C1060 is of Compute Capability of 1.3. Coalescing rules for Compute Capability 1.0 were strict. For coalescing to occur for Compute Capability 1.0 the data in the global memory had to be contiguous and linear, i.e. for threads in a half warp to coalesce a read of data from array $x$ in global memory then thread $i$ must address $x[i]$, thread $i+1$ must address $x[i+1]$, etc. For devices of Compute Capability 1.3 this rule has been relaxed slightly so that threads in a half warp can coalesce a read of data from array $x$ in global memory if all the data required by the half warp is in a block of the array $x$ that is

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| X1 | Y1 | Z1 | RHO1 | MASS1 | X2 | Y2 | Z2 | RHO2 | MASS2 |

Table 3.1: Memory alignment for a simple particle structure

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| X1 | X2 | ... | Y1 | Y2 | ... | Z1 | Z2 | ... | RHO1 |

Table 3.2: Memory alignment using arrays of variables

of a certain size, depending on the data type being read. This means that if thread *i* wants to read *x[i+i]* and thread *i+1* wants to read *x[i]* then on a device of Compute Capability 1.3 this read could be coalesced if *x[i]* and *x[i+1]* are in the same segment of array *x*. This subtle difference is shown in Figure 3.2[55] in which the left diagram shows that for items of data to be coalesced for Compute Capability 1.0 the data had to be contiguous and linear, while for Compute Capability 1.3 the pattern of access can be more complex but so long as the required data is in the same segment of global memory the access can be coalesced in one or two instructions, depending on the data type being used. CUDA code should also be written to exploit coalescing by using arrays of variables instead of arrays of structures. For example, and because this thesis examines Smoothed Particle Hydrodynamics, a particle could be defined in C as proposed in the structure in Figure 3.3. where *x*, *y*, *z* are the coordinates of the particle, *mass* is the particle mass and *rho* is the density of the particle.

In memory this would be allocated as shown in Table 3.1. When data is coalesced neighbouring data elements are accessed together in one read or write. The structure in Figure 3.3 does not lend itself well to coalescing because the data for a particular parameter is not sequential, i.e. the *x* coordinates are not immediate neighbours but are 5 float variables apart in memory. The particle data could be rearranged in memory and instead written as a list of separate arrays for *x*, *y*, *z*, *rho*, and *mass*. With the data arranged in this format, the memory would look like that suggested in Table 3.2.

Using the second data structure in Table 3.2 a call could be issued in device code to read the variable array *x* in a kernel as shown in Figure 3.4. Because the data elements in the linear array *x* are contiguous, then when considering neighbouring particles in memory, the reading (or writing) of elements of *x* for these neighbouring particles can be coalesced. However, if using the data structure *particle* declared in Figure 3.3, the *x*

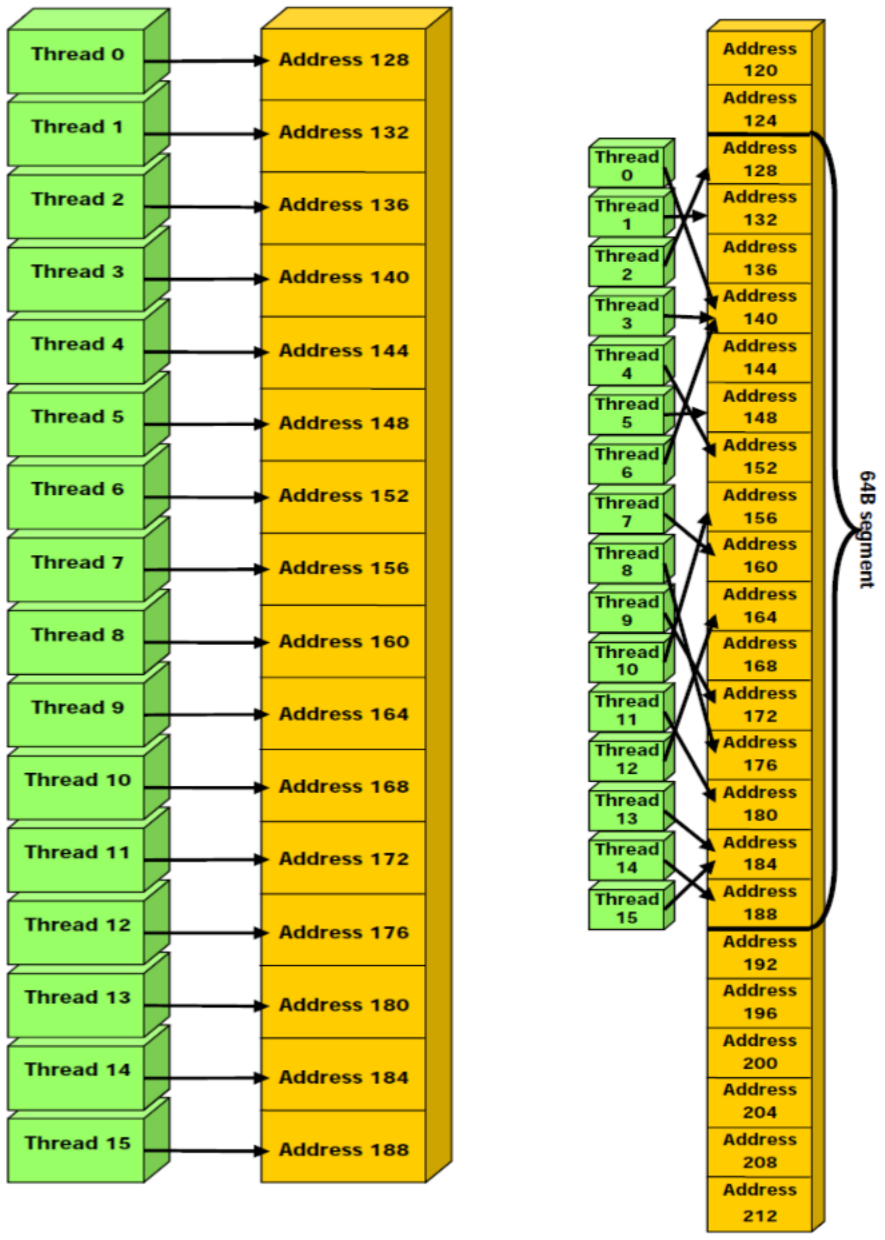


Figure 3.2: Valid coalescing rules

```
typedef struct
{
        float       x;
        float       y;
        float       z;
        float       rho;
        float       mass;
}particle;
```

Figure 3.3: A simple structure in C to define a particle

```
__global__ ReadX()
{
  int        tid = threadIdx.x;
  int        myX = x[tid];
}
```

Figure 3.4: Reading a linear array

```
__global__ ReadX()
{
  int        tid = threadIdx.x;
  int        myX = ParticleArray[tid].x;
}
```

Figure 3.5: Reading a structure to access a component

components of that structure are not contiguous so may not necessarily be coalesced. In order to access the same component *x* of the structure the code in Figure 3.5 would be required; Such a call over a half-warp could take up to 16 times longer, if the structure had many components, because reads would be required, not just the one read as when the data is coalesced using the second memory allocation of arrays of variables.

## 3.2 The Fundamentals of CUDA

CUDA is a complex language, and a significant knowledge of computer science is required to understand it. Not all the functionality of CUDA has been explored in this thesis, but that which has been employed in the source codes used for this thesis will now be described.

### 3.2.1 Data types

CUDA builds on the standard data types of C, such as *int* and *float*, to provide vector types. For example the CUDA data type *float4* is a vector consisting of 4 components, each of type float, and its components are referenced as *x*, *y*, *z* and *w*, as shown in Figure 3.6 The fourth component *w* could be used for a scalar quantity, such as pressure or density. Similar data types and methods of component reference exist for the data types *char* and *uchar*, *short* and *ushort*, *int* and *uint*, *long* and *ulong*, and *double*, and these vectors can have one, two, three or four components which are referenced *(x)*, *(x,y)*, *(x,y,z)* and

```
float4        position;
float        x,y,z,density;
x = position.x;
y = position.y;
z = position.z;
density = position.w;
```

Figure 3.6: Accessing components of a float4 variable

*(x,y,z,w)* respectively.

### 3.2.2 A CUDA Program

NVIDIA have employed a system in which data is passed between the host, which also acts as the master controlling the program and data flow, and the GPU, also called the device which also acts as the servant doing most of the computational work. The host, usually with much more memory than the device, holds the data and passes it to the device as and when required. A typical CUDA program would involve

1. allocating memory and initializing data on the host
2. allocating memory and initializing data on the device
3. transferring data from the host to the device
4. the device performing computation
5. transferring computed data from the device back to the host
6. the host writing the returned data to file or standard output
7. deallocation of memory on both host and device

Until recently the device could not write to file or standard output but the latest NVIDIA GPU, the Fermi, can now do this. Before this welcome addition to the functionality of CUDA, in order for the user to read the result of a calculation performed on the GPU the data must have been transferred back to the host from where a call to the standard C functions *printf* or *fprintf* would have been made. The data can be initialised either on the host and transferred to the device, which would occur if the initial data was first read from a file, or initialized on the device itself using a CUDA function *cuMemset* or initializing by thread ID if testing, for example.

The structure of a CUDA program takes a very similar form to that of a C program, and the source code can be written in modules so that C code to be executed on the host can be kept separate from the CUDA functions to be executed on the device. These CUDA functions that execute on the device only are called kernels. The *main* function is still required and is executed on the host only, but the *main* function can be written using C and CUDA functions, such as *cudaMemcpy*, which transfers data between the host and device. The extension for a CUDA file is .cu and a header file for that CUDA file has the extension .cuh, and any module calling a CUDA function must have the extension .cu and also include the CUDA header file $< cuda.h >$.

Functions executed on the host only are declared in the usual fashion for C functions. Functions that are executed on the device are declared as either

- void __global__
- void __device__

Any function declared as __global__ is a kernel and all threads declared in the thread grid specified for that kernel execute that kernel. A function declared as __device__ is not necessarily executed by all threads, but can only be executed on the device, and can only be called from a __global__ function or another __device__ function.

When a kernel is executed a thread grid for that kernel must be specified. This grid determines the total number of threads and the sequence in which the threads execute that kernel. The method of specification of this thread grid is flexible and can consist of two, three or four variables, either integers or of a special CUDA datatype called *dim3* designed specifically for thread grid specification.

Kernels are limited in size, e.g. by register use, so it cannot be guaranteed that an algorithm can be written in just one kernel, unless the programmer is fortunate enough that the algorithm is small enough or the problem size is small enough to allow this. A large and/or long algorithm will very probably need to be partitioned into several kernels, but this partitioning must also be done taking into account any natural synchronisation points in the algorithm. These restrictions are of great importance, because thread synchronisation is not that straightforward, as will be described later.

The NVIDIA CUDA compiler is called nvcc. To compile a CUDA file called *cudafile.cu* the command is *nvcc cudafile.cu*, and the directories containing the CUDA libraries and header files such as *cuda.h* must be supplied, either with the command or set in the environment paths.

There are a number of options that can be added to this command to compile. These options are

- -ptx which will compile the CUDA code into a ptx file, which is what the GPU actually reads.
- –ptxas-options=”-v” which will provide information about the memory and register use of each of the kernels declared. This is important because if a kernel is too big, i.e. uses too many registers, then the code will not execute properly.
- -cubin which will compile the CUDA code as cubin, which is a machine code language that can be manually altered to improve performance.
- -deviceemu which will compile the CUDA code to emulate the GPU for debugging, to run on the CPU and in serial.

Besides these options it is also possible to specify a level of optimization similar to most Fortran and C compilers such as that supplied by Gnu and Intel. The default is -O0, and the others are -O1, -O2 and -O3, each providing increasing levels of optimization and of similar levels as that provided by other compilers providing different levels of optimization, but with optimization applied to the CUDA code as well as the C. As stated above, it is possible to amend the machine code called cubin that nvcc produces in order to optimize the executable code even further.

To execute a kernel on the GPU, memory must first be allocated on the GPU to contain the data to be used in the kernel. This can be done in two ways and both can be used in an application;

```
int* d_array;
cudaMalloc((void**)&d_array,N*sizeof(int));
```

Figure 3.7: Allocating memory on the device with cudaMalloc

1. statically, by declaring a variable as __device__ in the pre-processor. Such memory is deallocated by the compiler.

2. dynamically, by allocating the memory in host code through a call to *cudaMalloc*, but the programmer must remember to deallocate such memory through a call to *cudaFree*.

As with static and dynamic memory, there are the usual advantages and disadvantages. Static memory persists throughout the lifetime of the application, so if a program is running short of memory such static memory cannot be deallocated to free memory for reallocation for use for another variable. On the other hand, dynamic memory can be allocated and deallocated at any point during execution to minimize memory use during the lifetime of an application. This implies that data can be transferred between the device and host more frequently than if the data variables were __device__ variables and remain on the GPU at all times, thus increasing execution time due to the transfer time but increasing the effective memory of the device because memory is used only when and as it is required.

Memory cannot be dynamically allocated in device code, but can be allocated dynamically in host code. This implies that a linked list cannot be created in device code. However there is a subtle way in which a linked list could be created dynamically, but it would involve returning data to the host and the host then dynamically allocating memory and transferring data back to the device, which is not exactly dynamic in the usual sense of the word in computer science and is probably highly inefficient. It may well be faster to manipulate the device architecture.

To allocate memory dynamically in host code a call to the function *cudaMalloc* is made. The code in Figure 3.7 allocates memory on the device for an array called *d_array* of N integers. This returns an address on the device pointing to the first element of

```
cudaFree(d_array);
```

Figure 3.8: Deallocating memory on the device with cudaFree

```
kernelname<<<NUMBLOCKS,NUMTHREADS>>>(parameters);
```

Figure 3.9: Specifying the thread grid for a kernel

*d_array*.

The code in Figure 3.8 deallocates memory on the device. A kernel is important for synchronisation of threads, for it is only at the end of a kernel that a guarantee can be made that all threads have written their data to global memory. There is a CUDA function called *__syncthreads*, but this only synchronises threads in a block, and it is very rare that only one block is used in a kernel, and it is in fact discouraged, so organising synchronisation through calls to *__synchthreads* and by partitioning the algorithm into kernels to achieve global synchronisation is very important.

A kernel is executed in blocks of threads, which themselves execute as warps, and the arrangement of these blocks and their threads is specified in the call to a kernel. This call to a kernel takes the form shown in Figure 3.9 in which

- *kernelname* is the name of the kernel
- *NUMBLOCKS* is the number of blocks of threads
- *NUMTHREADS* is the number of threads per block
- *(parameters)* is the list of parameters passed to the kernel, and must refer to data on the device only unless data from the host is being passed by value

The values of *NUMBLOCKS* and *NUMTHREADS* do not have to be the same for all calls to all kernels in an application, and can be tuned to some degree by using the occupancy calculator that comes with the CUDA Software Development Kit.

When this kernel is launched on the device a thread grid is created consisting of *NUMBLOCKS* blocks of *NUMTHREADS* threads. The blocks will initially execute in numerical order, but the order may be changed during execution by the block scheduler. An illustration of a thread grid is shown in Figure 3.10[55].

Variable arrays declared on the device and passed to the kernel are passed by reference. The parameter list of a kernel will usually consist of pointers to device variables, single variables being passed explicitly by value, and possibly structures of device variables if the number of variable arrays being passed to the kernel is excessive. This concept of passing variables by structures is addressed later in this chapter.

For example, to execute a kernel called *inckernel*, which will simply increment elements in variable array *d_array* declared on the device, with 3 blocks each of 32 threads, the statement in Figure 3.11 is made. When this statement is executed control of execution of the program passes from the host to the device, which will create 3 blocks of 32 threads, giving a total of 96 threads. To enable each thread to calculate its unique ID each thread makes the statement given in Figure 3.12, which is usually the first statement in any kernel. The block scheduler on the device attempts to optimize occupancy of the multiprocessor by scheduling the blocks of threads. It is possible, due to the latency in reading from and writing to global memory, that one block of threads wishes to write to global memory, but due to the slowness of the access to global memory the block scheduler could suspend the execution of that particular block of threads and execute another block of threads while the data from the previous block is being read from global memory at the same time. This allows different blocks of threads to be executing different parts of a kernel, i.e. not all threads are executing the same piece of code at any one time. The block scheduler estimates how long a sequence of statements for a block of threads takes, and decides if one block of threads can use the processors and its registers while a different block of threads currently using the registers executes a second set of statements that could take a relatively longer time, e.g. a read from global memory. If this occurs then the data in the registers is saved in on-chip memory ready for fast return of control to the multiprocessor to the initial block of threads. This is called context switching and has virtually zero overhead. The aim of this is to maximize occupancy of the multiprocessor.

Once a thread has calculated its unique ID it can then refer to data using that ID. In the example above the array *d_array* was passed to *inckernel*. For each thread to set the

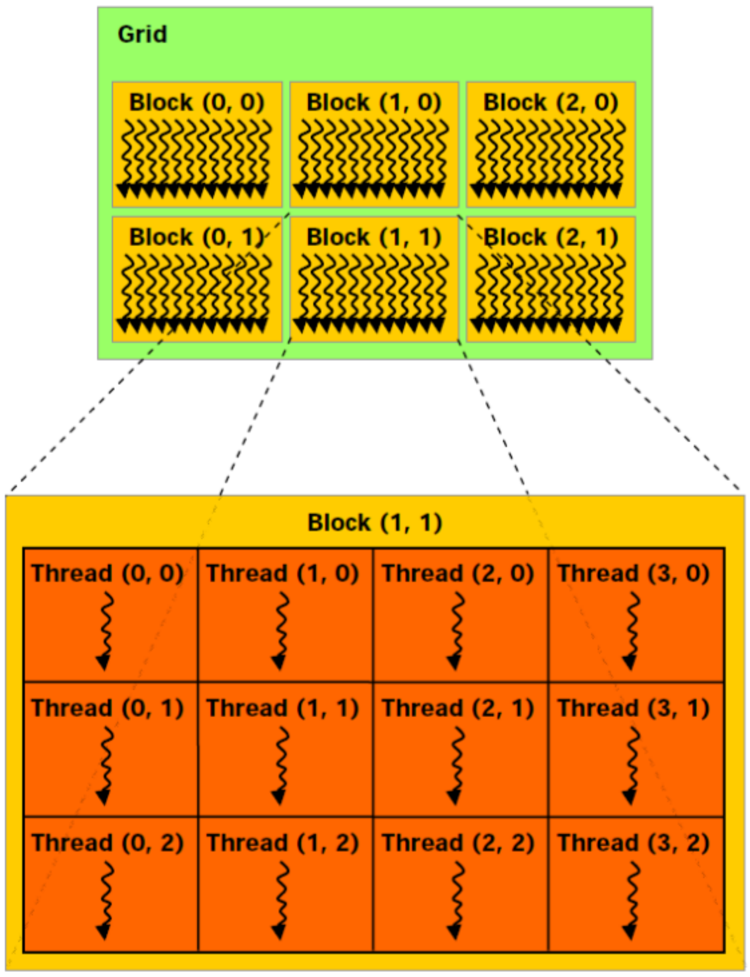


Figure 3.10: A Simple thread grid

```
inckernel<<<3,32>>>(d_array);
```

Figure 3.11: Specifying the thread grid for the inckernel

```
int         threadid = blockDim.x*blockIdx.x + threadIdx.x;
```

Figure 3.12: Each thread calculates its unique thread ID

```
d_array[threadid] = threadid;
```

Figure 3.13: Assigning thread ID

```
cudaMemcpy(h_array,
           d_array,
           N*sizeof(int),
           cudaMemcpyDeviceToHost);
```

Figure 3.14: Transfer of data from device to host

values of *d_array* to the ID of that thread the statement in Figure 3.13 is made using the variable *threadid* calculated by the thread in Figure 3.12. Every thread in the kernel's thread grid will execute this statement. This statement in Figure 3.13 has the effect that thread 0 will write 0 into d_array[0], thread 1 will write 1 into *d_array[1]*, thread 2 will write 2 into *d_array[2]*, etc, and this will be done more or less concurrently due to the size of the kernel because there are only 96 threads when a maximum of 128 can execute concurrently.

So there is now an array on the device initialized by thread ID.

But there is another piece of functionality that is generally not available on the NVIDIA GPU, except on the Fermi, which is to write to standard output or to file while in device code, i.e. in a kernel or device function. To perform either of these the data on the GPU must be transferred from the device back to the host, and once on the host the data can be written using the standard *printf* or *fprintf* C functions, but the latest NVIDIA Fermi GPUs do allow writing directly from the kernel code. To write data to or read data from the device a call to the function *cudaMemcpy* is made. The statement in Figure 3.14 reads data from the array *d_array* previously allocated on the device into an array *h_array* residing on the host.

The form of this function is given in Figure 3.15.

```
cudaMemcpy(destination,
           origin,
           number of bytes,
           direction);
```

Figure 3.15: Specification of cudaMemcpy

So what can a basic CUDA program look like? As an example, using the simple concepts above an array of 96 integers on the device will be initialised with the thread ID and the values transferred back to the host to print the values. This will show

1. static and dynamic memory allocation.
2. data transfer between host and device.
3. how to calculate the thread ID and use it.
4. the difference between a device function and a kernel.
5. the difference between host code and device code.

The code in Appendix A should be saved as *DeviceTest1.cu* and can be compiled with the command *nvcc -o DeviceTest1 DeviceTest1.cu* and executed with the command *./DeviceTest1*

An illustration of this program is shown in Figure 3.16 (adapted from NVIDIA CUDA Programming Guide 2.3.1[55]).

At line 7 an array of integers *device_array* is statically declared on the device (GPU).

At line 10 the kernel *device_kernel* to be executed on the GPU is declared. This kernel is passed an array which will store the values in *device_array*.

At line 11 the device function *SetDeviceArray*, which will be called from the kernel to initialize *device_array* with the thread IDs, is declared.

At line 13 the host function *allocateArray*, which will dynamically allocate memory on the device, is declared.

At line 14 the host function *freeArray*, which will dynamically deallocate memory on the device, is declared.

At line 18 the host array *host_array*, which will reside on the host, is declared.

At line 19 the device array *d_host_array*, which will reside on the device, is declared (but not allocated).

At line 25 the values of *host_array* after initializing them to 0 are printed out.

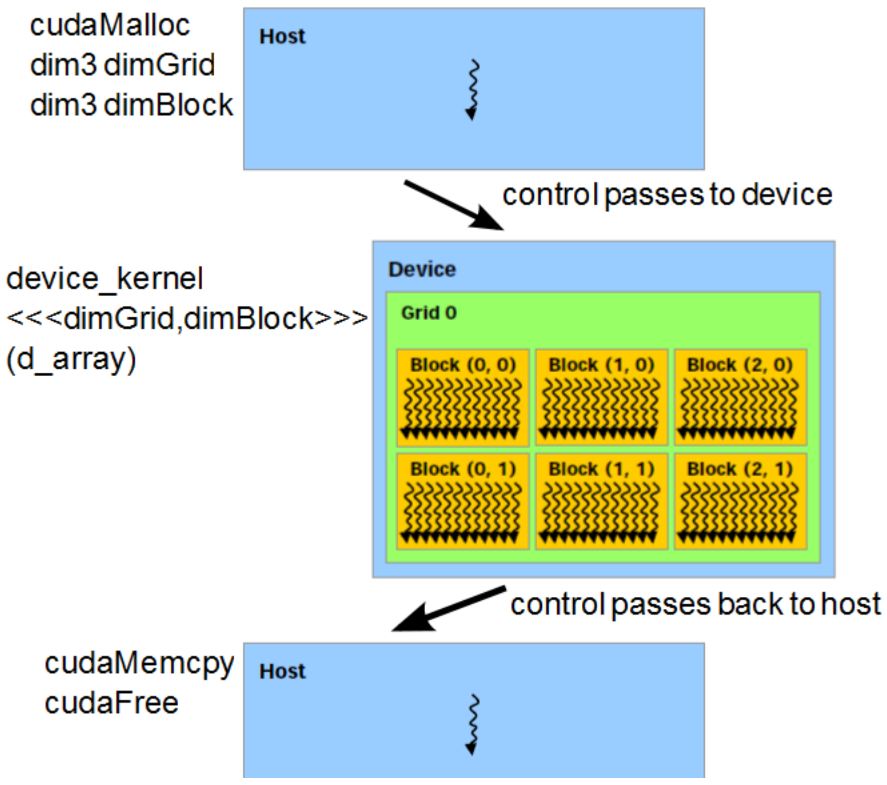


Figure 3.16: A simple CUDA program

At line 29 *d_host_array* is allocated on the device through a call to *allocateArray*, which calls *cudaMalloc*. The size of *d_host_array* must be provided, which is *N* x *sizeof(int)*. So at this point in the program the following memory has been allocated;

1. *device_array* on the device statically allocated in the preprocessor
2. *host_array* on the host
3. *d_host_array* on the device dynamically allocated

This is three arrays required to read one array. This is excessive, but this is a simple example to show how memory can be allocated on the device and how communication between the host and device can be achieved.

At line 31 the number of blocks of threads in the *dim3* variable *dimGrid* is defined, which has three parameters *dimGrid.x*, *dimGrid.y* and *dimGrid.z*, each of which has the default value of 1. The total number of blocks in this example $= dimGrid.x$ x $dimGrid.y$ x $dimGrid.z$, so with this declaration 3 blocks are being

```
kernelname<<<grid_dimensions, block_dimensions>>>(parameters)
```

Figure 3.17: General kernel specification

```
device_kernel<<<3,32>>>(d_array);
```

Figure 3.18: Simple kernel specification

created.

At line 32 the number of threads per block is defined in the *dim3* variable *dimBlock*, which has three parameters, *dimBlock.x*, *dimBlock.y* and *dimBlock.z*. The number of threads per block $= dimBlock.x$ x $dimBlock.y$ x $dimBlock.z$, so with this declaration each of the three blocks will contain 32 threads each, giving a total of 96 threads.

At line 36 the kernel *device_kernel* is called and control of the program is passed to the kernel. The call to a kernel takes the form given in Figure 3.17. Both *grid_dimensions* and *block_dimensions* can simply be integers, such as *G* and *B*, where *G* would be the number of blocks, while *B* would be the number of threads per block. The statement given in Figure 3.18 will have the same effect as declaring and using the *dim3* variables as above in lines 31 and 32. Note that although *d_host_array* is actually memory on the device it still needs to be passed by reference to the kernel from the host code because the kernel is going to read values from *d_host_array*. It should also be noted that the second array that has been declared on the device, *device_array* declared in the preprocessor, does not need to be passed because the compiler allocates this memory at compile time.

At line 39 the CUDA function *cudaMemcpy* is called to copy data between the device and the host, and takes four parameters, which are in order from left to right in the parameter list

1. the memory address to write to
2. the memory address to write from
3. the total size of the data being transferred
4. the direction of the data transfer

The direction takes one of three values

- cudaMemcpyDeviceToHost
- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToDevice

In this instance the data is initially held in *d_host_array* (which is on the device) and is to be transeferred into *host_array* (which is on the host) so the direction is *cudaMemcpyDeviceToHost*.

At line 4 the memory on the device which was dynamically allocated for *d_host_array* is deallocated because it is no longer need it. Note that there is no need to deallocate the array *device_array* that was declared in the preprocessor.

At line 43 the values in the array *host_array* are printed.

All the code above between lines 1 to 44 inclusive is called host code, because it executes on the host only, and not the device.

Lines 47 to 78 inclusive are called device code because they are executed on the device only, and are made to do so by the qualifiers __global__ (which declares a kernel function) and __device__ (which declares a device function).

All threads created in the kernel thread grid, defined by the parameters declared between the chevrons, execute the kernel. There is a possibility that not all threads execute the device function. A device function can be called from a kernel or another device function. A kernel cannot be called from a kernel or device function, only host code. Only one kernel executes at any one time.

The grid specification described by the *dim3* parameters *dimGrid* and *dimBlock* defines the number of threads in total, and the arrangement and partitioning of those threads. For this simple example, in Figure 3.10, there would be 3 blocks of 32 threads. If the number of threads is excessive then not all threads can be executed at once, so they are broken up into and executed as blocks which in turn execute in warps. This can hide latency in which for example a sequence of reads from global memory taking a long time can be 'parked' and another block of threads executes instead, in order to increase processor occupancy.

```
void __global__ kernelname(parameter list)
```

Figure 3.19: Simple kernel specification

At line 47 the kernel which will execute on the device is defined. A kernel is declared as given in Figure 3.19.

A kernel cannot return a value directly, hence the *void* in the kernel declaration, but it can return a value if a variable is passed to it by reference, then a value or set of values can be written into that variable, and a *cudaMemcpy* ican be executed to copy that value or values into a variable declared on the host. In this example the kernel is passed by reference an integer array which resides on the device but was initially created and allocated in the host code using *cudaMalloc*.

At line 50 the thread ID is calculated by all threads created in the thread grid by accessing the *x* component of the *threadIdx* variable that is calculated from the grid specification, and is implicit.

At line 52 the kernel calls the device function *SetDeviceArray*. As there is no conditional code, in other words no *if* or *switch* statement on the thread ID, in the kernel then every thread executes this device function. It is possible to partition the threads so that a subset of the thread grid executes this device function with the use of the thread ID, or perhaps a computed variable. When this occurs the threads in the warp execute in serial, which decreases performance.

At line 54 the value per thread ID from *device_array* is copied into *d_host_array*. As each thread executes the kernel and the thread ID is known, each thread will assign the value in *device_array[id]* to *d_host_array[id]*.

At line 57 the kernel definition ends. This is the only point in the kernel at which it is guaranteed that all threads in the thread grid for the kernel have written their index of *device_array* into *d_host_array*. Blocks of threads can be synchronised through the call to *__synchthreads()*, but this guarantees synchronicity between threads in a block only, not all threads in all blocks. This is important if the algorithm being implemented has synchronization points at which it is crucial that a variable array has been written to by all threads. In SPH this could be the array for particle pressure.

At line 59 the function to dynamically allocate memory on the device is defined. This uses *cudaMalloc*.

At line 65 the function to deallocate dynamically allocated memory on the device is defined. This uses *cudaFree*.

At line 71 the __device__ function to initialize the array *device_array* that was statically allocated on the device is defined. All threads will execute this particular device function because there is no logical condition on its execution and it is called directly from a kernel.

At line 75 the ID of every thread is found by the call to *threadIdx*.

At line 77 the values of *device_array* are initialized by each thread to that thread's ID.

This program is a very basic CUDA program that

- statically allocates memory on the GPU in the preprocessor
- dynamically allocates memory on the GPU in host code using *cudaMalloc*
- defines a kernel grid using *dim3* variables, named *dimGrid* and *dimBlock*
- executes a kernel, defines a simple thread grid for that kernel, and passes a data structure residing on the device to that kernel
- initializes a statically allocated array in the kernel with thread IDs
- copies that statically allocated array on the GPU into the dynamically allocated array on the GPU passed to the kernel
- reads the data from the GPU back to the host via *cudaMemcpy*
- prints out the data from the host

These are CUDA essentials, in that memory allocation is required to hold data, and data transfer between host and device is required to read results obtained from execution of a kernel on the GPU, because it is generally not possible to write to standard output or file in device code, i.e. directly from the GPU, except if the compute capability of the device is 3.2. The CUDA SDK also provides a basic performance analysis suite called *cudaprof* that can report the coalescing in an application.

## 3.3 A Closer Look at Memory Hierarchy

As stated above the GPU has several types of memory. These are

1. global memory - large and slow
2. texture - cached global memory
3. shared memory - small and very fast
4. constant - small and fast, but not as fast as either registers or shared memory
5. registers - small and fast but very limited

How to use each type of memory will now be described.

### 3.3.1 Global Memory

Memory allocated using *cudaMalloc* is in global memory. This memory is large but slow, and is usually where most data on the GPU initially resides. The challenge of a CUDA program is to minimize data reads from and writes to global memory because data transfer between the registers, where all the computation takes place, and global memory is of the order of 400 to 600 clock cycles. One useful piece of functionality of global memory is that coalescing of reads and writes can be performed in which neighbouring addresses can be accessed simultaneously so that multiple reads from and writes to global memory can be executed in one instruction. The details of how this is achieved in hardware are not given in the CUDA programming guides.

Coalescing can be achieved for data types that are 32 bit, 64 bit, or 128 bit words, and are aligned as linear memory. The most obvious declaration for which this would work is to statically declare an integer array on the device. For the C870 processor integers are 32 bit words. CUDA has a number of data types that are 4, 8 and 16 bytes, i.e. 32 bit, 64 bit and 128 bit words, the most common being *int*, *int2* and *int4*, and *float*, *float2* and *float4*. To achieve coalescing of data the data must be

1. of a type that is a 32 bit, 64 bit or 128 bit word, and can also be a structure of those sizes, e.g. a structure of two 32 bit integers

```
texture<float, 1, cudaReadModeElementType> d_rhoTex;
```

Figure 3.20: The declaration of a linear 1D texture

2. aligned so that the hardware can execute the coalescing, e.g. declared as a linear array

An array of structures that is not of a size that is 32, 64 or 128, such as the structure defined in Figure 3.3, will not be coalesced, but the compiler will minimize the required memory accesses.

The disadvantage of global memory is that it is not automatically cached, though this is now the case with the Fermi processors. Most CPU-based computers have a memory structure that uses at least one cache to use the principles of spatial and temporal locality. Spatial locality implies that if an item of data is required from global memory then items of data that are physically close to that item of data will also be required. Temporal locality implies that if an item of data is required from global memory then that item of data will very soon be required again. This principle can be implemented on the GPU by declaring texture memory.

### 3.3.2 Texture Memory

Texture memory is cached global memory. NVIDIA do not go into too much detail into how global memory is cached when a variable is declared as a texture, so how this is implemented is unknown, though an investigation has been carried out by Wong *et al.*[56]. A cached memory is relatively good for random access of data in global memory, but NVIDIA suggest that the use of coalescing and shared memory is preferable to textures if the algorithm can be written to exploit the multiple reads and writes in one instruction that coalescing and shared memory offer. Data transfer between registers and texture memory is of the order of 40 clock cycles.

A simple linear texture of type float called *d_rhoTex* can be declared as shown in Figure 3.20. To use this texture it must be bound to a device variable. Binding allows the device variable to be cached. This binding is performed before the kernel using the

```
cudaBindTexture(0,
                d_rhoTex,
                d_rho,
                num*sizeof(float4));

kernel<<<Numblocks,Numthreads>>>(parameter list);

cudaUnbindTexture(d_rhoTex);
```

Figure 3.21: The binding, use and unbinding of a texture

```
__shared__ float shX[DATASIZE];
```

Figure 3.22: The declaration of a shared memory variable

texture is executed. Once bound to a device variable the texture variable does not need to be passed to the kernel in the kernel parameter list. Any reference to the device variable must instead be through reference to the texture variable in the kernel code. The texture must be unbound after kernel execution. The code in Figure 3.21 shows how to bind, use in a kernel and then subsequently unbind a device variable *d_rho* representing density to a texture *d_rhoTex*. NB There is no need to pass the texture variable, or the device variable to which it is bound, to the kernel in the kernel parameter list. Any reference to the device variable must be made through reference to the texture. For this to occur the texture declarations, such as that in Figure 3.20, must be made in the same file or module as the kernel code using the textures. For example, if a large CUDA project has been written in modules such that the kernels are in their unique .cu module then the textures are declared in that module.

### 3.3.3 Shared Memory

Reading from and writing to shared memory is very fast, due to its proximity to the registers, and should be used as much as possible, but it is very small compared to global and texture memories. To allocate a variable array called *shMass*, representing mass, of type *float* and of size *DATASIZE* the declaration given in Figure 3.22. As with global memory, if the data in shared memory is not aligned properly then a phenomenon called a bank conflict can reduce the efficiency of using shared memory. The shared memory is divided into 16 banks, and when data is written to the shared memory from global

```
int        MyMass = shMass[id];
```

Figure 3.23: Assigning a shared memory variable to a register

memory a misalignment of data can take place. If 16 items of data are transferred from global memory into shared memory then those items will occupy 16 locations in different banks. If two or more items of data from the same bank are required in an instruction then the access to those items must be serialized, so that if the maximum number of items of data on any one bank is *N* then the number of reads/writes required will be *N*. Hence if all 16 items are on 16 different banks then only one instruction is required.

For example, take the single simple instruction in Figure 3.23 to read the variable *shMass* declared in shared memory in Figure 3.22 into a register variable *MyMass*. This instruction reads the $id^{th}$ item of *shMass* in shared memory and copies it to a register called *MyMass* allocated by each of the $id^{th}$ threads. If 16 threads in a half-warp execute this instruction and the 16 items of data are in 16 banks then those 16 threads will each have their data in the register *MyMass* in one instruction only. But if just two items of data are in the same bank then two instructions will be required, i.e. it would take double the time than if the 16 items were in 16 banks.

So if shared memory is to be used it would make sense to load data into shared memory in blocks of 16 threads and in a way such that the data is aligned in global memory. Any larger than that and bank conflicts could arise, particularly when executing in blocks of more than 16 threads. This is the significance of a half-warp. If 32 items of data were written into shared memory from linear global memory via coalescing then items 0 to 15 would be in banks 0 to 15, but so also would be items 16 to 31. If an instruction is subsequently made by a block of 32 threads to read those 32 items then threads 0 and 16 would require access to the same bank 0, but different items of data in that bank. The access to the bank would be serialized and thread 0 would access the bank first followed by thread 16.

This should suggest why the GPU is so efficient; multiple items of data from **global** memory can be coalesced into **shared** memory in one read/write instruction, and from shared memory multiple items of data very quickly colaesced into registers if the data is on different banks in the shared memory. The kind of mathematical computations in which this kind of data transfer pattern is required are in matrix manipulations, in which one element interacts with a lot of other elements in a well defined process. In these particular cases blocks of data can be coalesced in to shared memory, then each thread can very quickly process all that data. This is not the case in SPH, where a particle interacts with only a very small and quickly changing subset of all the particles. It will be shown in the next chapter, which shows how SPH can be implemented on GPUs, that using coalescing and shared memory for small datasets can be efficient, but for large datasets using texture memory is more efficient.

So an algorithm that when implemented on the GPU uses both coalescing and shared memory will have one of the fastest possible memory allocations. Using data in shared memory is the fastest way of processing recycled data, and coalescing the data reads from global memory into shared memory and writes from shared memory to global memory is a highly efficient method of data transfer between those two memories.

So to recap on data transfer between the registers and the different memories on the device, the typical time it takes to access the memories from the registers is as follows;

- global memory - 400 clock cycles
- texture memory - 40 clock cycles
- shared memory - 4 clock cycles

### 3.3.4 Constant Memory

Constant memory is initialized at the start of an application and cannot be written to again once initialization has taken place. This memory is useful for storing constant values that will be used very frequently.

### 3.3.5 Registers

Register use is important because if a kernel uses too many registers then it will not launch and the application will fail, usually with the run time error message ”too many resources requested”. Information about register use before execution can be obtained by compiling with the –ptxas-options=”-v” option. The output from the compiler when using this flag reports the following for each declared kernel;

- the estimated number of registers used
- the estimated amount of shared memory used
- the estimated amount of constant memory used

The most important of these is the register use, followed by shared memory. This includes the registers required for any __device__ functions called from that kernel. Suggestions on how the programmer can reduce register use, or increase the number of available regsiters permitted, are

- more and smaller kernels
- use constant and/or shared memory to hold variables that occupy registers
- reduce the number of register variables by recycling those that have been declared
- use –maxregcount=X in the compile command to allocate X registers per thread

Any combination of these can maximize the number of registers to a level such that the kernel will execute.

## 3.4 Pinned Memory and Streams

It is possible to accelerate the process of transferring data between device and host and executing a kernel by using pinned memory and streams. This adds another layer of parallelism to the algorithm but can improve efficiency significantly if the algorithm requires both a significant amount of data transfer between host and device and can be manipulated to facilitate streaming.

```
cudaMemcpy(d_data,
           h_data,
           datasize,
           cudaMemcpyHostToDevice);

incKernel<<<numblocks,numthreads>>>(d_data);

cudaMemcpy(h_data,
           d_data,
           datasize,
           cudaMemcpyDeviceToHost);
```

Figure 3.24: A simple data transfer to and from the device

A stream enables the overlap of kernel execution and data transfer by partitioning the thread grid into separate processes which manage a portion of the data being calculated or manipulated. A simple example of this would be to write a block of data from host to device, then for a kernel to increment that data, and then transfer the manipulated data back from device to host. Code for this simple exercise is given in Figure 3.24. Under normal circumstances in the code in Figure 3.24 the kernel *incKernel* could not be executed until all the data has been transferred from the host to the device, and the transfer of the incremented data from the device back to host cannot occur until all threads in the thread grid have executed the *incKernel*. With streams it is possible to overlap writing and kernel execution by creating processes that are not in complete synchronisation, so that one stream can begin processing its data in the kernel while another stream is still receiving data from the host. However, there is a caveat. This can only be done if the actual purpose of the kernel permits this, i.e. a thread can execute the whole kernel without reference to any other thread that may not yet have received its data from the host, so that no thread synchronisation is required. In the simple example above in Figure 3.24 of a kernel that only increments data there is no dependence between any threads so streaming can be used.

A set of streams called *stream* can be declared and allocated with the procedure shown in Figure 3.25 and can be destroyed with the code given in Figure 3.26. To partition the kernel execution and data transfer the streams are executed in a loop as shown in Figure 3.27. The code in Figure 3.27 partitions data so that CUDA streams can be used. For Figure 3.27 there are *datasize* items of data and both *d_data* and *h_data* are of this size.

```
cudaStream_t *stream =
     (cudaStream_t*) malloc(NSTREAMS*sizeof(cudaStream_t));
for(int i = 0 ; i < NSTREAMS; i++)
     cudaStreamCreate(&(stream[i])) ;
```

Figure 3.25: Declaring and creating an array of streams

```
for(int i = 0; i < NSTREAMS; i++)
       cudaStreamDestroy(stream[i]);
```

Figure 3.26: Destroying an array of streams

```
for(int i = 0; i < NSTREAMS; i++)
{
  int streamstart=i*datasize/NSTREAMS;
  cudaMemcpyAsync(&d_data[streamstart],
                  &h_data[streamstart],
                  datasize/NSTREAMS,
                  cudaMemcpyHostToDevice,
                  stream[i]);

  incKernel<<<numblocks/NSTREAMS,
            numthreads,0,
            stream[i]>>>(&d_data[streamstart]);

  cudaMemcpyAsync(&h_data[streamstart],
                  &d_data[streamstart],
                  datasize/NSTREAMS,
                  cudaMemcpyDeviceToHost,
                  stream[i]);
}
```

Figure 3.27: Using an array of streams for concurrent kernel execution and data transfer

```
int N;
int* h_data;
cudaMallocHost((void**)&h_data,N*sizeof(int));
```

Figure 3.28: Declaring pinned memory

Each stream is responsible for processing a chunk of data of size *datasize/NSTREAMS*, where *NSTREAMS* is the number of streams, beginning at index *i* x *datasize/NSTREAMS* in both *d_data* and *h_data*, where *i* is the ID of the stream. NB when using CUDA streams the data on the **host**, in this case the array *h_data*, must be declared as pinned memory, which is achieved by using the CUDA function *cudaMallocHost* as shown in Figure 3.28.

## 3.5 Limit on Kernel Parameter List

There is a limit on the size of the parameter list passed to a kernel. That limit is 256 bytes. This can cause trouble when an algorithm requires a lot of parameters, which was required in implementing the Vila SPH algorithm on GPUs. One way of working around this is to declare on the device a structure of device pointers pointing to memory on the device, and pass that device structure to the kernel in the kernel's parameter list instead of the long list of device pointers themselves.

The code in Figure 3.29 shows how this can be done with a small structure. In the code in Figure 3.29, once the structures have been allocated on **both** the host and device, and the device arrays have been allocated with *cudaMalloc*, the trick is to

1. assign the device arrays to the **host** structure
2. copy the **host** structure to the **device** structure

With this process complete, the device arrays *d_rhoudx*, *d_rhoudy* and *d_rhoudz* can be passed to a kernel by passing the single device structure instead of the three device arrays as shown in Figure 3.30. and the components of the structure can be referenced in the kernel code as shown in Figure 3.31. This procedure adds another layer of reference and

```
//declare the structure
typedef struct
{
  float*       drhoudx;
  float*       drhoudy;
  float*       drhoudz;
}params;

//declare the structure parameters
params* h_params;
params* d_params;

//allocate memory for the structures
h_params = new params;
cudaMalloc((void**)&d_params,
            sizeof( CSPHparameters));

//declare and allocate device arrays
float* d_drhoudx;
cudaMalloc((void**)&d_drhoudx,
            N*sizeof(float));
float* d_drhoudy;
cudaMalloc((void**)&d_drhoudy,
            N*sizeof(float));
float* d_drhoudz;
cudaMalloc((void**)&d_drhoudz,
            N*sizeof(float));

//assign device pointers to host structure
h_params->drhoudx = d_drhoudx;
h_params->drhoudy = d_drhoudy;
h_params->drhoudz = d_drhoudz;

//copy host structure to device structure
cudaMemcpy(d_params,
           h_params,
           sizeof(params),
           cudaMemcpyHostToDevice);
```

Figure 3.29: Creating a structure on the device to pass a list of variables to a kernel

```
incKernel<<<BLOCKS,BLOCKSIZE>>>(d_params);
```

Figure 3.30: Passing a structure on the device to pass a list of variables to a kernel

```
int ThreadID = blockDim.x*blockIdx.x + threadIdx.x;
float mydrhodux = d_params->drhoudx[ThreadID];
```

Figure 3.31: Dereferencing variables passed in a structure in a kernel

degrades performance a little, but is the only way to pass a large number of variables to a kernel.

## 3.6 Warp Divergence

When the threads in a warp encounter a logical statement, such as an *if* statement, the possible paths are evaluated and the threads in that warp are executed in batches such that the threads taking branch 1 execute first, threads taking branch 2 execute second etc. In other words the paths are executed in serial, but the threads taking those paths are executed in parallel, with all other threads disabled. The threads in the warp converge when all threads in that warp have all executed their respective branches, and the whole warp executes in parallel again.

## 3.7 Padding

Padding is related to warp divergence. It is possible that the number of particles in a simulation is not a multiple of a warp. One possible way around this is to execute a kernel only if the particle ID is less than a constant, e.g. the total number of fluid particles. But this adds warp divergence, because the *if* statement to determine the value of the particle ID needs to be executed by all threads in the warp. Another way around this is to add dummy particles which execute all the code in the kernel but do not have any effect, thus avoiding the potential for warp divergence and the cost of all threads executing a *if* statement on the particle ID.

To illustrate how padding can be easily implemented in SPH assume that a simulation involves *P* particles, where *P* is not a multiple of the warp size and $P = N$ x $warpsize{+}x$, where *1¡x¡warpsize*. The variable arrays can be slightly increased in size to hold data for $P = (N + 1)$ x $warpsize$. To guarantee that these extra particles in the padding have no effect on the simulation these extra padding particles can be given positions which are significantly but not ridiculously outside the domain of interest. These particles will not interact with any particles within the domain of interest but their acceleration etc. can be calculated as part of a warp without having to make any distinction between the padded particles and the real particles of interest in the simulation. Similarly for their integration. With this method a warp can execute in full without any warp divergence and without any effect on the simulation.

## 3.8 Error Reporting

When a kernel executes it returns messages to inform the user of the success or otherwise of the execution of the kernel. The following statement can be inserted after each kernel call to report the outcome of kernel execution.

```
printf("\n\n%s\n", cudaGetErrorString(cudaGetLastError()));
```

This will return any error that occured during a kernel execution. One error message that relates to a shortage of registers is ”unspecified launch failure”. One problem with this function is that it is possible that an error being reported is due to a kernel that was executed before the kernel that is being reported as causing the error.

# Chapter 4

# SPH on a GPU

The previous two chapters described Smoothed Particle Hydrodynamics and the NVIDIA GPU architecture together with some but not all of the functionality of CUDA, presenting only the CUDA functionality that has been used in the code to implement SPH on GPU as presented in this thesis.

This chapter will describe how an SPH algorithm can be implemented on a GPU. NVIDIA recommend that algorithms executed on their GPUs should use the coalescing and shared memory approach to exploit the fast data transfer from the slow global memory to the fast shared memory offered by coalescing, and the close proximity of the shared memory to the registers for fast data recycling. This approach will be examined and described. Another approach that has been proposed is to use the texture memory as a cache. This method will also be implemented. The results from implementing both methods for the same SPH algorithm for the same problem will be compared for execution time. The theoretical computational effort to process particle interactions for the shared memory method is $O(N^2)$, and is $O(N)$ for the texture memory method, where $N$ is the number of particles. But the different speeds of the memories of the GPU could make the shared memory method faster than the texture memory method.

The CUDA Software Development Kit comes with a profiler and occupancy calculator. The profiler shows some performance counters such as the number of coalesced reads. The occupancy calculator suggests the optimal size of the thread block for a given kernel in a CUDA program. This chapter will also show the effect that the thread block size can have on GPU occupancy and thus performance.

Previous work on comparing different implementations of the same SPH algorithm was done by McCabe *et al.*[57] who made a brief study of SPH on GPUs by implementing the SPH algorithm proposed by Monaghan[33], but with some amendments, on a NVIDIA C870 by using the two different methods of implementation to be described in this chapter.

## 4.1 The SPH Algorithm

This thesis looks at implementing the SPH algorithm proposed by Ferrari *et al.*[42][58] so the reader will benefit greatly if the implementation of this algorithm on a GPU is explained in some detail. The governing equations are

$$\frac{d\rho_i}{dt} = -\sum_{j=1}^{N} m_j \Big( (\underline{v}_j - \underline{v}_i) \cdot \nabla_i W_{ij} - \underline{n}_{ij} \cdot \nabla_i W_{ij} \Big( \frac{c_{ij}}{\rho_j} (\rho_j - \rho_i) \Big) \Big) \tag{4.1}$$

$$\frac{d\underline{v}_i}{dt} = -\sum_{j=1}^{N} \underline{F}_{ij}^{I} - \sum_{j=1}^{N} \underline{F}_{ij}^{V} + \underline{S}_i \tag{4.2}$$

where

$$\underline{F}_{ij}^{I} = m_j \Big( \frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \Big) \nabla_i W_{ij} \tag{4.3}$$

$$\underline{F}_{ij}^{V} = \Theta_{ij} \Big( \frac{7}{3} \frac{\mu}{\rho_i} \frac{m_j}{\rho_j} + \frac{5}{3} \frac{\mu}{\rho_i} \frac{m_j}{\rho_j} \underline{n}_{ij} \cdot (\underline{v}_j - \underline{v}_i) \Big) (\underline{v}_j - \underline{v}_i) \tag{4.4}$$

$$\Theta_{ij} = -\frac{\underline{n}_{ij}}{|\underline{x}_j - \underline{x}_j|} \cdot \nabla_i W_{ij} \tag{4.5}$$

$$\underline{n}_{ij} = \frac{\underline{x}_j - \underline{x}_i}{|\underline{x}_j - \underline{x}_i|} \tag{4.6}$$

$$c_{ij} = max(c_i, c_j) \tag{4.7}$$

$$c_i = \sqrt{\gamma \frac{B}{\rho_0} \Big( \frac{\rho_i}{\rho_0} \Big)^{(\gamma - 1)}} \tag{4.8}$$

$$P_i = B\Big( \Big( \frac{\rho_i}{\rho_0} \Big)^{\gamma} - 1 \Big) \tag{4.9}$$

$$\mu = 0.0013 \tag{4.10}$$

In their paper, Ferrari *et al.*[42] state that their results are from an inviscid fluid to show that their scheme is stable without the need for a viscous term.

The time step is calculated by Equations (2.88) - (2.89).

In their work Ferrari *et al.* use the third order Runge Kutta TVD integration scheme Equations (2.84) - (2.87), but for simplicity in the work presented in this chapter the predictor-corrector integration scheme Equations (2.77) - (2.79) is used to move the particles. The boundary treatment does not use that proposed by Ferrari *et al.*. Instead the on-boundary particle treatment as proposed by Dalrymple & Knio[38] is used. The smoothing function is the Wendland quintic given by Equation (4.11).

$$W_{ij} = \begin{cases} \frac{7}{4\pi h^2}(1+2q_{ij})(1-0.5q_{ij})^4 & \text{if } 0 < q_{ij} \leq 2, \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

where $q_{ij} = (|\underline{x}_i - \underline{x}_j|)/h_{ij}$.

To the best of this authors knowledge this particular combination of these components for a SPH algorithm has not been reported, not even on a CPU.

The problem being simulated is that investigated by Koshizuka & Oka[11] for which experimental data is available to verify the accuracy of the algorithm and the code.

To show the difference in program structure and performance in using the shared memory and coalescing approach, as recommended by NVIDIA, and the texture memory approach, there now follows a detailed description of the implementation of both approaches.

## 4.2 Shared Memory and Coalesced Implementation

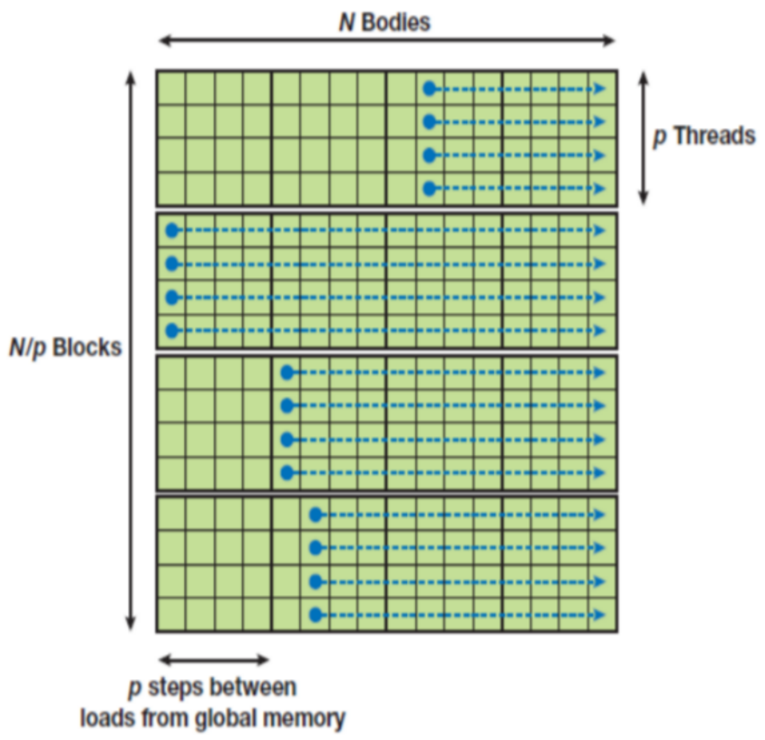
As described in the previous chapter on GPUs and CUDA programming, NVIDIA advise that using coalescing and shared memory should be used as much as possible. Coalescing of data from global memory can be achieved in certain circumstances, and these were described in the previous chapter. Once in shared memory that data can be reused very efficiently from the fast shared memory. This data re-use is essential for efficient implementations of matrix operations, but may not be so good for SPH in which one particle interacts with only a small subset of all the particles in the simulation.

| Name | Datatype | Description |
|---|---|---|
| d_x | float2 | the position |
| d_v | float2 | the velocity |
| d_dvdt | float2 | the acceleration |
| d_x0 | float2 | the saved position at start of timestep |
| d_v0 | float2 | the saved velocity at start of timestep |
| d_mass | float | the mass |
| d_rho | float | the density |
| d_rho0 | float | the saved density at start of timestep |
| d_p | float | the pressure |
| d_hsml | float | the variable smoothing length |
| d_drhodt | float | the rate of change of density |
| d_celerity | float | the celerity |
| d_type | integer | the particle type |

Table 4.1: The device variables required for shared memory implementation

This method is based on the work on N body simulation by Nyland *et al.*[59] which examines the calculation of gravitational forces in an N body system. The idea in this implementation is to process $N$x$N$ interactions as efficiently as possible, which can be represented as a $N$x$N$ matrix as shown in the top image in Figure 4.1[59]. This matrix can be partitioned into blocks of threads, as would occur on the GPU. The bottom image in Figure 4.1 shows the process for one simple block of four threads, called block *A*. This block *A* of threads, with each thread representing a particle, has to process the data from all particles to find interactions. The bottom image indicates that the plan is to load blocks of particles, which in this simple case would be four particles at a time, into shared memory and then process the data of those particles. So the data for particles in block *A* is first to be loaded into shared memory by coalescing, and then processed by particles in block *A*. Then the data for particles in block *B* is loaded into shared memory by coalescing, and processed by particles in block *A*. Then block *C*. Then block *D* etc. Once all blocks have been processed by block *A* the particles in block *B* process the data of all particles in the same way by loading, or coalescing, data from blocks *A*, *B*, *C*, *D* etc. into shared memory. Then block *C* processes all the blocks. Then block *D* etc.

The *main* function for this implementation has the structure as shown in Figure 4.2.



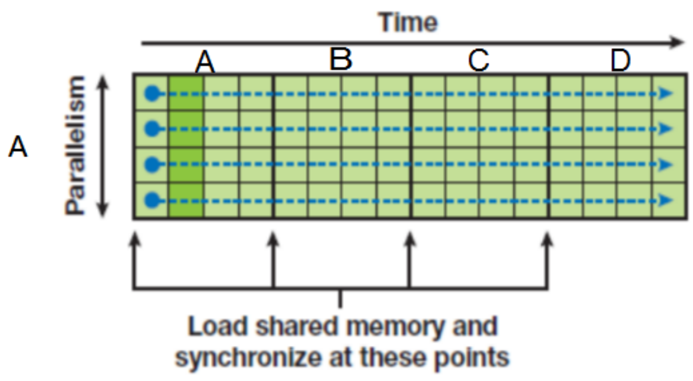


Figure 4.1: N Body simulation

```
allocate host variables
allocate device variables
read initial data from file into host variables
transfer host data to the device
for maxiterations
{
    save x, v and rho
    SPH calculations
    integrate prediction
    calculate maxH
    calculate dtpred
    SPH calculations
    integrate correction
    calculate maxH
    calculate dtcorr
    calculate dt for next time step
}

free device variables
```

Figure 4.2: Pseudo code for implementation of main

### 4.2.1 Calculation of Rates of Change

The function, or kernel, to calculate the acceleration and rate of change of density is called from the *main* function via an intermediate function called *CalcForces*. This intermediate function as shown in Figure 4.3 is quite straightforward in that it calculates the thread grid dimensions for the kernel that will execute on the device. The variables that are passed to this function which are described in Table 4.1 are passed on to the kernel for the device to refer to.

NB in this code there is a variable *d_cell* which is passed on to the kernel. This variable is the mesh of background cells which particles can be assigned to, but is not used in this example.

The kernel requires the following to occur

1. transfer data from global memory to shared memory

2. each thread processes the data in shared memory

In the code provided in Appendix B this kernel is called *calculateforceskernel*. The initialisation of this kernel is shown in Figure 4.4. The shared memory arrays to hold the particle position, velocity, smoothing length, mass, density, pressure and type are declared. These will be filled with data a block at a time, hence the size of the arrays is declared as *NUMTHREADS*, the thread block size. The thread ID *threadid* within the block of threads is required because each thread within the block will coalesce one item of data

```
int numThreads, numBlocks;

computeGridSize(numBodies, NUMTHREADS, numBlocks, numThreads);

calculateforceskernel<<<numBlocks,numThreads>>>
(d_x,
d_v,
d_mass,
d_hsml,
d_rho,
d_p,
d_type,
d_drhodt,
d_dvdt,
d_cell,
numBodies);

cudaThreadSynchronize();
```

Figure 4.3: The Intermediate CalcForces function

from the global memory into shared memory, and the global thread ID *gtid* is required for reading the correct items of data into the thread's own private variables *myX*, *myHsml* etc. The variables *mydvdtx*, *mydvdty* and *mydrhodt* will accumulate the acceleration in the *x* direction, the acceleration in the *y* direction and rate of change of density respectively as particle interactions are found.

Once initialised the thread can begin its main loop to load, or coalesce, particle data from global memory into shared memory a block of particles at a time. This main loop is shown in Figure 4.5 with supporting snippets for coalescing global data into shared memory in Figure 4.6 and processing particle interactions in Figure 4.7.

The first call to the CUDA function *__syncthreads()* is made to synchronise the threads in the block so that no thread in the block can progress until all threads in the block have read their designated items of data from global in to shared memory.

As stated above the idea is to coalesce data from the global memory into the shared memory. For the outer loop the variable *idx* increases with a stride of size
*tile* x *NUMTHREADS* each iteration of the loop, where *tile* represents the block ID and *NUMTHREADS* is the thread block size. In the first iteration threads coalesce data from global memory into shared memory for particles *0* to *NUMTHREADS-1*, with thread 0 of the block loading particle 0, thread 1 loading particle 1, etc. In the next loop threads coalesce data from global memory into shared memory for particles *NUMTHREADS* to *2NUMTHREADS-1*, with thread 0 of the block loading particle NUMTHREADS, thread

```
//calculate thread id
int        threadid = threadIdx.x;
int        blockid = blockIdx.x;
int gtid = blockid* blockDim.x + threadid;

//declare shared memory variables
__shared__ float2 shX[NUMTHREADS];
__shared__ float2 shV[NUMTHREADS];
__shared__ float  shHsml[NUMTHREADS];
__shared__ float  shMass[NUMTHREADS];
__shared__ float  shRho[NUMTHREADS];
__shared__ float  shP[NUMTHREADS];
__shared__ int    shType[NUMTHREADS];

//load local register variables
float2 myX = d_x[gtid];
float2 myV = d_v[gtid];
float  myHsml = d_hsml[gtid];
float  myRho = d_rho[gtid];
float  myP = d_p[gtid];
int    myType = d_type[gtid];

//set accumulators to zero
float  mydvdtx = 0.0f;
float  mydvdty = -GRAVITY;
float  mydrhodt = 0.0f;
```

Figure 4.4: The Initialisation of calculateforceskernel kernel

```
float2  D,V;
float          r, mhsml;
float          xdwdx,ydwdx;
float          K,Kxdwdx,Kydwdx;
float          A,C;
float          Ci,Cj,Cij;
float2        n;

int        i,j,tile,diff,jparticleid;
__syncthreads();

for (i=0, tile=0; i<BLOCKSIZE; i+=NUMTHREADS, tile++)
{
  //load global data into shared memory

  __syncthreads();

  for (j = 0; j < NUMTHREADS ; j++)
  {
    jparticleid = tile * NUMTHREADS + j;
    diff = gtid - jparticleid;
    switch(diff)
    {
      case        0  :  break;

      default   :  D.x = myX.x-shX[j].x;
                   D.y = myX.y-shX[j].y;
                   mhsml = (myHsml + shHsml[j])/2.0;
                   r = sqrt(D.x*D.x + D.y*D.y);

                   if(r<SCALE*mhsml)
                   {
                     //process interaction
                   }
                   break;
    }
  }
  __syncthreads();
}
```

Figure 4.5: The calculateforceskernel kernel main loop

```
//load global data into shared memory
int idx = tile * NUMTHREADS + threadid;
shX[threadid] = d_x[idx];
shV[threadid] = d_v[idx];
shHsml[threadid] = d_hsml[idx];
shMass[threadid] = d_mass[idx];
shRho[threadid] = d_rho[idx];
shP[threadid] = d_p[idx];
shType[threadid] = d_type[idx];
shCell[threadid] = d_cell[idx];
```

Figure 4.6: Coalescing global data into shared memory

```
kerneldw(&xdwdx,&ydwdx,r,mhsml,D);
D = shX[j] - myX;
V = shV[j] - myV;
n = D/r;

//calculate maximum celerity
Ci = sqrt(GAMMA*B*pow((myRho/RHO0),GAMMA-1)/RHO0);
Cj = sqrt(GAMMA*B*pow((shRho[j]/RHO0),GAMMA-1)/RHO0);
Cij = max(Ci,Cj);

////////////////////////////////////////////////
//  DENSITY
////////////////////////////////////////////////
mydrhodt+= -shMass[j]*(V.x*xdwdx + V.y*ydwdx);
mydrhodt+= shMass[j]*(n.x*xdwdx + n.y*ydwdx)*Cij*
                (shRho[j] - myRho)/shRho[j];

////////////////////////////////////////////////
//        MOMENTUM
////////////////////////////////////////////////
////////////////////////////////////////////////
//  FI
////////////////////////////////////////////////
K = myP/(myRho*myRho) + shP[j]/(shRho[j]*shRho[j]);
K*= shMass[j];

Kxdwdx = K*xdwdx;
Kydwdx = K*ydwdx;
mydvdtx+= -Kxdwdx;
mydvdty+= -Kydwdx;

////////////////////////////////////////////////
//  FV
////////////////////////////////////////////////
A = MU*shMass[j]/(3.0f*myRho*shRho[j]);
C = (n.x*V.x + n.y*V.y)*(n.x*xdwdx + n.y*ydwdx)/r;

mydvdtx+= A*(7.0*V.x + 5.0*C*n.x);
mydvdty+= A*(7.0*V.y + 5.0*C*n.y);
```

Figure 4.7: Process the interaction between two distinct particles

```
d_dvdt[gtid].x = mydvdtx;
d_dvdt[gtid].y = mydvdty;
d_drhodt[gtid] = mydrhodt;
```

Figure 4.8: The Coalesced Write of Acceleration and Rate of Change of Density to Global Memory

1 loading particle *NUMTHREADS+1*, etc. This is the purpose of the statements in Figure 4.6, which coalesces the positions, velocities etc into shared memory.

The inner loop checks for interactions between the particles by each thread reading the contents of the shared memory in sequence. Before checking for an interaction each thread must check that the ID of the particle being checked is not the same as the particle it is representing, hence the statement *diff = gtid - jparticleid;*. The variable *jparticleid* is the ID of the particle being checked and is incremented in the inner loop by the statement *jparticleid = tile * NUMTHREADS + j;*.

If the particles are found to be distinct then the distance between them is calculated. If that distance is less than twice the mean of the particle smoothing lengths then there is an interaction which is processed as shown in Figure 4.7. The function *kerneldw* calculates the kernel gradient. The maximum celerity $C_{ij}$ of the two interacting particles is first calculated, which is needed for the calculation of density. The rate of change of density and the accelerations in the x and y directions are then found and accumulated. First the rate of change of density is calculated according to the Ferrari algorithm. This is done for all particles, fluid and boundary, because the density of boundary particles changes for boundary particles proposed by Dalrymple & Knio[38]. The Ferrari algorithm splits the acceleration into two components, *FI* and *FV*, as indicated in the governing equations above.

The second call to *__syncthreads()* in the main loop synchronises the threads in the block so that no thread can begin loading the next block of data into shared memory before the other threads in the block have processed the current particle data in the shared memory, and thus corrupt the computation.

The main loop is then advanced one step, the value of *tile* is incremented, the data from the next block of particles is coalesced into shared memory, overwriting the data from the previous iteration of the main loop, and the threads then process the data in shared memory in the same way, accumulating change in density and acceleration from any interactions.

Once all the particles, fluid and boundary, have been processed by the threads in the current block the total *x* and *y* acceleration and rate of change of density can be coalesced

```
int         i;
float         speed,celerity;

i = blockDim.x * blockIdx.x + threadIdx.x;

d_rho[i]+= (d_drhodt[i])*dt/2.0f;
d_p[i] = B*(pow((d_rho[i]/RHO0),7) - 1);

if(d_type[i]==WATER)
{
  //integrate acceleration and velocity
  //////////////
  d_x[i].x+= d_v[i].x*dt/2.0f;
  d_x[i].y+= d_v[i].y*dt/2.0f;

  d_v[i].x+= d_dvdt[i].x*dt/2.0f;
  d_v[i].y+= d_dvdt[i].y*dt/2.0f;

  d_hsml[i] = sqrt(d_mass[i]/d_rho[i]);
  celerity = sqrt(GAMMA*B*pow((d_rho[i]/RHO0),GAMMA-1)/RHO0);
  speed = sqrt(d_v[i].x*d_v[i].x + d_v[i].y*d_v[i].y);
  d_celerity[i] = d_hsml[i]/max(celerity,speed);
}
```

Figure 4.9: The predictor phase integration

to global memory, as shown in Figure 4.8.

### 4.2.2 Integration

The Predictor-Corrector integration scheme is straightforward to implement and described in Chapter 2. The integration at the end of the prediction phase is shown in Figure 4.9. When using the boundary treatment of Dalrymple & Knio the density and thus pressure are calculated for all particles, fluid and boundary. The momentum is calculated for fluid particles only, hence the requirement to check if the particle is water. The second integration, performed at the end of the correction phase, is given in Figure 4.10.

## 4.3 Texture Memory Implementation

The texture approach to implementing SPH involves assigning the particles to cells in a background grid, as described in the chapter on SPH, but the data required for the SPH calculations is sorted so that each particle has two IDs; the first being its *original* ID and the second its *sorted* ID. In this implementation, during the SPH calculations the data for each particle is referred to by its *sorted* ID, but the accumulated acceleration and rate of change of density and the integrated values of position, velocity and density are written

```
int        i;
float        speed,celerity;

i = blockDim.x * blockIdx.x + threadIdx.x;

d_rho[i] = d_rho0[i] + d_drhodt[i]*dt/2.0f;
d_rho[i] = 2*d_rho[i] - d_rho0[i];
d_p[i] = B*(pow((d_rho[i]/RHO0),7) - 1);

if(d_type[i]==WATER)
{
  //integrate acceleration and velocity
  //////////////
  d_x[i].x = d_x0[i].x + d_v0[i].x*dt/2.0f;
  d_x[i].y = d_x0[i].y + d_v0[i].y*dt/2.0f;

  d_v[i].x = d_v0[i].x + d_dvdt[i].x*dt/2.0f;
  d_v[i].y = d_v0[i].y + d_dvdt[i].y*dt/2.0f;

  ///////////////////////////////////////////////
  //        FINAL INTEGRATION
  ///////////////////////////////////////////////
  d_x[i] = 2*d_x[i] - d_x0[i];
  d_v[i] = 2*d_v[i] - d_v0[i];

  celerity = sqrt(GAMMA*B*pow((d_rho[i]/RHO0),GAMMA-1)/RHO0);
  speed = sqrt(d_v[i].x*d_v[i].x + d_v[i].y*d_v[i].y);
  d_hsml[i] = sqrt(d_mass[i]/d_rho[i]);
  d_celerity[i] = d_hsml[i]/max(celerity,speed);
}
```

Figure 4.10: The corrector phase integration

```
calculate particle hash (cell ID)
sort particles based on cell ID
reorder data and record cell starts
calculate acceleration and rate of change
```

Figure 4.11: The pseudo code for SPH calculations

to the particle's *original* ID. Therefore besides the device variables given in Table 4.1 as used in the shared memory approach this method requires the extra variables given in Table 4.2.

The *main* function for this implementation has the same structure as that for the shared memory approach, as shown in Figure 4.2, but the *SPH calculations* are more complex and have the structure as shown in Figure 4.11. The procedure to implement the SPH calculations as given in Figure 4.11 is as follows;

1. find the ID of the cell each particle is in

2. sort the particle positions by increasing cell ID

3. record the index of the first particle in each cell

| Name | Datatype | Description |
|---|---|---|
| d_sorted_x | float2 | the sorted position |
| d_sorted_v | float2 | the sorted velocity |
| d_sorted_hsml | float | the sorted smoothing length |
| d_sorted_mass | float | the sorted mass |
| d_sorted_rho | float | the sorted density |
| d_sorted_p | float | the sorted pressure |
| d_sorted_type | integer | the sorted particle type |
| d_cellStart | uint | stores the index of first particle in each cell |
| d_particleHash[0] | uint2 | stores the particles sorted by cell ID |
| d_particleHash[1] | uint2 | intermediate array required for sorting |

Table 4.2: The extra device variables required for texture implementation

4. reorder the required data arrays by using the sorted particle IDs
5. calculate acceleration and density change by binding the sorted data to textures
6. integrating using the original arrays

This sequence will now be described in much greater detail and will refer to the extra variables given in Table 4.2.

### 4.3.1 Cell ID

This uses an array of datatype *uint2* called *d_particleHash*. The first, or *x*, component of *d_particleHash* is the cell ID and the second, or *y*, component is the original particle ID. Four constants are also defined; *GRIDSIZEX* and *GRIDSIZEY* define how many cells there are in the *x* and *y* directions in the background grid respectively, and *MINX* and *MINY* are the minimum values of *x* and *y* in the grid.

The cell to which each particle belongs is found for each particle in parallel through a call to the intermediate function *calcHash* called from *main* which is given in Figure 4.12. This intermediate function first calculates the thread grid for the kernel through the call to the function *computeGridSize*, which simply sets the thread grid size to a multiple of the thread block size, and then calls the kernel *calcHashD*.

NB All particles execute this kernel.

```
void
calcHash(float2* pos,
uint2* particleHash,
int numBodies,
FILE* outfile,
float cellsize)
{
  int numThreads, numBlocks;

  computeGridSize(numBodies, NUMTHREADS, numBlocks, numThreads);

  calcHashD<<< numBlocks, numThreads >>>(pos,
                                         particleHash,
                                         cellsize);

  cudaThreadSynchronize();
}
```

Figure 4.12: The calcHash function

```
__global__ void calcHashD(float2* pos,uint2*  particleHash,float cellsize)
{
  int index = __mul24(blockIdx.x, blockDim.x) + threadIdx.x;

  float2 p = pos[index];

  // get address in grid
  int2 gridPos = calcGridPos(p,cellsize);
  uint gridHash = calcGridHash(gridPos);

  particleHash[index].x = gridHash;
  particleHash[index].y = index;
}
```

Figure 4.13: The calcHashD kernel

The kernel *calcHashD* is given in Figure 4.13. The cell ID is found with the following *device* functions. First the cell coordinates in the background grid are found with the device function *calcGridPos*, which simply calculates how many cells up and to the right the cell containing the particle is, as shown in Figure 4.14. The cell coordinates are then passed to a second device function *calcGridHash* which multiplies the $y$ component of the cell coordinate by the number of cells in the $x$ direction and then adds the $x$ component of

```
__device__ int2 calcGridPos(float2 p,float cellsize)
{
  int2 gridPos;

  gridPos.x = floor((p.x - XMIN) / cellsize);
  gridPos.y = floor((p.y - YMIN) / cellsize);

  return gridPos;
}
```

Figure 4.14: The calcGridPos device function

```
__device__ uint calcGridHash(int2 gridPos)
{
  gridPos.x = max(0, min(gridPos.x, GRIDSIZEX-1));
  gridPos.y = max(0, min(gridPos.y, GRIDSIZEY-1));

  return __mul24(gridPos.y, GRIDSIZEX) + gridPos.x;
}
```

Figure 4.15: The calcGridHash device function

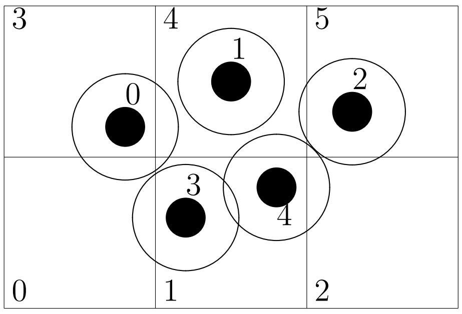
the cell coordinate, as shown in Figure 4.15. For example, in Figure 4.16 *GRIDSIZEX*=3, *GRIDSIZEY*=2, *CELLSIZEX*=1, *CELLSIZEY*=1, *MINX*=0 and *MINY*=0. For particle 1 the grid coordinates of the cell containing particle 1 are *(1,1)*, and the function *calcGridHash* would calculate the ID of this cell as 4.

This cell ID, or grid hash, is then stored in the array *d_particleHash* with *d_particleHash[index].x=gridHash* and *d_particleHash[index].y=index*, as shown in the unsorted particleHash array Figure 4.16, in which the top row in the cell ID and the bottom row is the particle ID. Note that this array is ordered by ascending particle ID.

### 4.3.2 Sorting

The sorting of the data is performed by a radix sort as proposed by LeGrand[60], but any sorting procedure can be used. However, the radix sort has already been written in CUDA, and the code for this sort procedure comes with the CUDA Software Development Kit, or it can be downloaded from the NVIDIA website, and can be easily added to a CUDA project. Figure 4.16 shows 5 particles contained in a simple 3x2 grid. The array *d_particleHash* would look like that shown in Figure 4.16, in which for each column the top row, component *x*, is the cell ID and the bottom row, component *y*, is the particle ID. After sorting the *d_particleHash* array by ascending cell ID it would look like that shown in Figure 4.16. Note that the unsorted *d_particleHash* array is ascending in particle ID, while the sorted *d_particleHash* array is ascending in cell ID. Another array *d_cellStart* records the lowest index in the sorted *d_particleHash* array at which the particles contained in each cell begin. In this example, looking at the array *d_cellStart* the entry for cell 0 is *ff*, which means there are no particles in that cell, as is the entry for cell 2. Particles in cell 1 begin at index 0 of the sorted *d_particleHash* array (there are two particles in that cell), the value, the particles in cell 3 begin at index 2 of the sorted

A simple background grid containing particles



The unsorted particleHash array

| | | | | |
|---|---|---|---|---|
| 3 | 4 | 5 | 1 | 1 |
| 0 | 1 | 2 | 3 | 4 |

The sorted particleHash array

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 3 | 4 | 5 |
| 3 | 4 | 0 | 1 | 2 |

The cell start array

| | | | | | |
|---|---|---|---|---|---|
| $ff$ | 0 | $ff$ | 2 | 3 | 4 |

Figure 4.16: The sorting and reordering of particles in a grid

*d_particleHash* array (there is only one particle in that cell), the particles in cell 4 begin at index 3 of the sorted *d_particleHash* array (there is only one particle in that cell) and finally the particles in cell 5 begin at index 4 of the sorted *d_particleHash* array (there is only one particle in that cell). The number of elements in the *d_particleHash* array is *NTOTAL*, for all particles, while the number of elements in *d_cellStart* is *GRIDSIZEX* x *GRIDSIZEY=numGridCells*.

## 4.3.3 Reordering

Once the particles have been reordered by cell ID then all their data can be reordered in the same order. For example, in Figure 4.16 the particle with original ID of 2 had all its data at the index 2 of all the variable arrays, such as position, density, etc. But after

```
cudaBindTexture(0, d_xTex, d_x, numparticles*sizeof(float2));
```

Figure 4.17: Binding a texture

```
texture<float2, 1, cudaReadModeElementType> d_xTex;
```

Figure 4.18: Declaring a texture

sorting, that particle with original ID of 2 is now at index 4 in the sorted arrays, i.e. its new sorted ID is 4. So all the data could be reordered so that the position, velocity, smoothing length, mass, etc for particle with original ID of 2 is now at index 4 of a family of *d_sorted* arrays. This explains the use of the *d_sorted* family of variables declared in Table 4.2.

The reordering is done by a call from *main* to the intermediate function *reorder-DataAndFindCellStart*, passing the <u>device</u> variables to be sorted, together with the variables to hold the sorted versions of these variables. Along with these the *d_cellStart* array is also passed, because it holds the indices of the sorted *d_particleHash* at which the cells start. The number of cells in the grid *numGridCells* and the total number of particles to be sorted *ntotal* are also passed, because these are required to bind the original arrays to textures.

As stated in the previous chapter on GPUs and CUDA, to bind a texture for position *d_x*, for example, a statement similar to that given in Figure 4.17 is made. The textures should be defined in the module/file in which they are to be referenced, i.e. the file or files containing the kernel code. Textures are declared with a statement similar to that given in Figure 4.18, which declares a linear one dimensional texture of datatype *float2*.

The variable arrays, for original and sorted values, with the sorted *d_particleHash* and *d_cellStart* arrays are then passed to the kernel *reorderDataAndFindCellStartD*, the *D* indicating it is being executed on the device.

All particles have their data reordered in parallel by this kernel.

Once the kernel has been executed the textures are unbound with a simple statement similar to that given in Figure 4.19.

```
cudaUnbindTexture(d_xTex);
```

Figure 4.19: Unbinding a texture

### 4.3.4 Calculation of Acceleration and Density Change

With the data sorted by cell ID, and a record of where the particles in each cell are in those sorted arrays (stored in the array *d_cellStart*), it is a relatively straightforward procedure to now iterate through the sorted arrays and find the particles and their data for all particles in any cell. SPH requires that only particles within a small specified distance should make up the support domain for a particular particle. This can be quickly achieved by searching for particles in cells that are immediate neighbours of the cell in which a specific particle is contained.

To calculate the acceleration and density change the function intermediate function *CalcForces* is called from *main*, passing the *sorted*, not the original, arrays along with the *d_particleHash* and *d_cellStart* arrays. These are then bound to textures and a call to the kernel *CalculateForces* is made. This is given in Appendix C. Note that not all variables bound to textures are passed in the kernel parameter list. The sorted arrays for position, velocity, smoothing length, mass, pressure, density and type are not in the parameter list for the kernel *CalculateForces*. This is because the textures, not the variables bound to the textures, can be referenced directly in the kernel code.

The *CalculateForces* kernel uses two indices to refer to variable arrays. The first index *mySortedIndex* is the thread ID. This index is used to access the sorted data to create a family of variables beginning with *my*, such as *myX* for position, from the textures. It is possible to reference the original data, but this is cumbersome and unnecessary in this context, but is essential for multiple GPU use. Note that because this data is read from a variable that has been bound to a texture then the values are now stored in texture, i.e. have been cached, so if required by any other particle can be read from cache.

The second index *myTrueIndex* is found from the *y* component of the *d_particleHash* array at index *mySortedIndex*, and is used to write the accumulated acceleration and density change to the correct index of the *original* arrays in the global memory at the end of the kernel.

```
// examine only neighbouring cells
for(int y=LOWCELL; y<HICELL; y++)
{
  gridPos2y = myGridPos.y + y;
  for(int x=LOWCELL; x<HICELL; x++)
  {
    gridPos2x = myGridPos.x + x;
    {
      gridPos2.x = gridPos2x;
      gridPos2.y = gridPos2y;

      AccumulateForces(...);
    }
  }
}
```

Figure 4.20: The nested loop to investigate neighbouring cells

To search for particle interactions in neighbouring cells only, the grid coordinates *gridPos* of the cell which contains the particle with *ID=mySortedIndex* is found. Then a nested loop is entered in which the values -1,0,1 are added to both the *x* and *y* components of the *int2* variable *myGridPos*. In 2D this gives 9 cells to check for interactions. The purpose of the nested loop is to find all interactions involving the particle with *ID=mySortedIndex*, finding the contribution to the acceleration and density change from all interactions in each cell and accumulating the contributions on a cell by cell basis. This nested loop takes the form shown in Figure 4.20. In the nested loop in Figure 4.20 the *device* function *AccumulateForces* is called. Note again that neither the sorted variables nor the textures to which they are currently bound, are passed in the parameter list for *AccumulateForces*. The textures are referenced directly in the code of *AccumulateForces*.

The purpose of the *device* function *AccumulateForces* is to find the interactions between one particular particle being managed by the thread and all particles in a particular cell and accumulate the acceleration and density change from any interactions found. Each thread, or particle, executes the kernel *CalculateForces* in parallel, so each thread is calling *AccumulateForces* in parallel. A single thread, or particle, has its data passed down in the *my* family of values. The grid coordinates of the cell being examined in *AccumulateForces* are passed down in the variable *gridPos2*. The ID, or Hash, for this cell is found with the call to *calcGridHash*. With the cell ID, or *cellgridHash*, for the current cell under investigation for interactions, the particles in that cell can be found from finding the entry at *d_cellStart[cellgridHash]*, which is the lowest index of *d_particleHash* where the particles are listed in sequence, i.e. the first particle in the cell can be found

```
// get start of bucket for this cell
uint bucketStart = FETCH(d_cellStart, cellgridHash);

if (bucketStart == 0xffffffff) // cell is empty
return density;

// iterate over particles in this cell
for(uint i=0; i<MAXPARTICLESPERCELL; i++)
{
  uint index2 = bucketStart + i;
  uint2 cellData = FETCH(d_particleHash, index2)

  // no longer in same bucket
  if (cellData.x != cellgridHash) break;

  // particle i is not particle j
  if (index2 != mySortedIndex)
  {
    // if interaction
    // calculate acceleration
    // and density change
  }
}
```

Figure 4.21: The loop to step through particles

at *d_cellStart[cellgridHash]*, which is the new index of each particle, because they have been sorted by ascending cell ID. Recall that in Figure 4.16, the particles were sorted by ascending cell ID, and the array *d_cellStart* held the index in the sorted *d_particleHash* at which the particles in a particular cell started.

So to iterate over all the particles in the current cell being investigated for interactions, the loop shown in Figure 4.21 is executed.

In the code snippet in Figure 4.21, the index *bucketStart* in the sorted arrays and the *d_particleHash* for the cell with *ID=cellgridHash* is found, and if this value *=0xffffffff* the cell is empty, and the function is left returning zero contributions from that cell. So with a cell containing at least one particle all the particles in that cell, up to a maximum number of particles *MAXPARTICLESPERCELL*, are checked sequentially for interations. To check that the particle being examined in the loop is contained in the cell being examined, the variables *index2*, which is the sorted ID, and *cellData*, are used to examine the contents of *d_particleHash[index2]*, because this contains the cell ID to which the particle with *sorted* index *index2* belongs. If the *x* component of that entry, the cell ID, is not the same as *cellgridHash* then something is wrong or the loop has run through all the particles in the cell and has moved on to the next cell, in which case the function is left. The third and final check is that the two particles are not the same particle, which is possible. This

```
if (index2 != mySortedIndex)
{
  // calculate acceleration and
  // density change...
}
```

Figure 4.22: The statement to check that two particles are not the same

```
//find distance between two particles
xj = FETCH(d_sorted_x,index2);
hsmlj = FETCH(d_sorted_hsml,index2);
D = myX - xj;
r = sqrt(D.x*D.x + D.y*D.y);
mhsml = 0.5f*(myHsml + hsmlj);

//if interaction is occuring
if(r<2.0f*mhsml)
{
  // calculate acceleration and
  // density change...
}
```

Figure 4.23: Checking for interaction

is made with the statement in Figure 4.22, which checks that the *sorted* IDs are not the same. If the particle being examined is not the same particle being managed by the thread executing the code then a check is made to find if an interaction is occuring by the snippet of source code shown in Figure 4.23, which simply finds the distance between the particle centres. In the snippet in Figure 4.23, the positions and smoothing lengths of the particles in the cell are brought from texture memory via a call to the function *FETCH*, which takes the texture as the first parameter and the index of the particle as the second parameter. NB the texture was not passed to the function in the parameter list. This *FETCH* function is defined in a file *definitions.cuh* and is short hand for the *CUDA* function *tex1Dfetch*.

If there is an interaction then variable values are first brought from texture to register variables, as shown in the snippet in Figure 4.24, to make the SPH computations more efficient. NB this data is brought from memory by the *FETCH* function, which brings the data first from global memory, and subsequent requests for that data from other threads

```
xj = FETCH(d_sorted_x,index2);
vj = FETCH(d_sorted_v,index2);
rhoj = FETCH(d_sorted_rho,index2);
pj = FETCH(d_sorted_p,index2);
massj = FETCH(d_sorted_mass,index2);
hsmlj = FETCH(d_sorted_hsml,index2);
```

Figure 4.24: Fetch texture values into registers for use in SPH calculations

```
kerneldw(&xdwdx,&ydwdx,r,mhsml,D);

D = xj - myX;
V = vj - myV;
n = D/r;

Ci = sqrt(GAMMA*B*pow((myRho/RHO0),GAMMA-1)/RHO0);
Cj = sqrt(GAMMA*B*pow((rhoj/RHO0),GAMMA-1)/RHO0);
Cij = max(Ci,Cj);

////////////////////////////////////////////////
//  DENSITY
////////////////////////////////////////////////
//Ferrari Riemann
  (*mydrhodt)+= -massj*(V.x*xdwdx + V.y*ydwdx);
  (*mydrhodt)+= massj*(n.x*xdwdx + n.y*ydwdx)*
                Cij*(rhoj - myRho)/rhoj;

////////////////////////////////////////////////
//  MOMENTUM
////////////////////////////////////////////////
////////////////////////////////////////////////
//        FI
////////////////////////////////////////////////
K = myP/(myRho*myRho) + pj/(rhoj*rhoj);
K*= massj;

Kxdwdx = K*xdwdx;
Kydwdx = K*ydwdx;
*(mydvdtx)+= -Kxdwdx;
*(mydvdty)+= -Kydwdx;

////////////////////////////////////////////////
//  FV
////////////////////////////////////////////////
A = MU*massj/(3.0f*myRho*rhoj);
C = (n.x*V.x + n.y*V.y)*
      (n.x*xdwdx + n.y*ydwdx)/r;

*(mydvdtx)+= A*(7.0*V.x + 5.0*C*n.x);
*(mydvdty)+= A*(7.0*V.y + 5.0*C*n.y);
```

Figure 4.25: Calculate the acceleration and density change for the texture implementation

```
// Save the result in global memory
volatile uint2 sortedData = particleHash[mySortedIndex];
uint myTrueIndex = sortedData.y;

mydvdt.x = mydvdtx;
mydvdt.y = mydvdty;

d_drhodt[myTrueIndex] = mydrhodt;
d_dvdt[myTrueIndex] = mydvdt;
d_dvdt[myTrueIndex].y+= -GRAVITY;
```

Figure 4.26: The coalesced write of accumulation of acceleration and density change to global memory

are read from the cache. Then the SPH computations are made using these register variables. First the change of density is calculated, followed by the *FI* and *FV* components of the acceleration are calculated, just as with the shared memory approach described above, and is shown in Figure 4.25.

This is performed for all particles in the cell. The loop is terminated if it is found that a particle's *gridHash*, found in *d_particleHash[index2].x*, does not equal *cellgridHash*.

When all neighbouring cells have been checked for particle interactions, the accumulated acceleration *mydvdt* and accumulated density change *mydrhodt* are then written to the *d_drhodt* and *d_dvdt* arrays respectively at index *myTrueIndex*. Note that the index used here is *myTrueIndex* not *mySortedIndex*, because *myTrueIndex* is the particle's true original index before sorting and reordering and the data is being written to the original arrays and not the sorted arrays. This value of *myTrueIndex* is in
*d_particleHash[mySortedIndex].y*, which is shown in Figure 4.16. The code which performs this operation is in Figure 4.26.

With the accumulated acceleration and density change in the original arrays, the same integrations as for the shared memory approach can take place and are given in Figures 4.9 and 4.10 .

## 4.4 Performance of the two implementations

Both the texture and shared memory implementations described above were run for a collapse of a water column 0.146 m x 0.292 m in a tank of 0.6 m x 0.6 m as described in the experiment performed by Koshizuka & Oka[11] with three different resolutions. The

```
X0 = 1.01f*DXL;
Y0 = DYL;
fac = 0.5f;
x = X0;
do
{
  y = Y0;
  do
  {
    //initialise particle
    y+=DYL;
  }while(y<HEIGHT);
  x+= 0.5f*DXL;
  fac = -fac;
  Y0 = Y0 + fac*DYL;
}while(x<(LENGTH));
```

Figure 4.27: The set up of fluid particles

parameters, the particle spacings, *DXL* and *DYL* were 8 mm, 4 mm and 2 mm, giving a total of 468, 2628 and 10440 fluid particles respectively. The execution times taken to simulate 1 second of real time, including saving outputs for printing every 0.1 seconds, with the number of iterations required to reach 1 second real time are given in minutes in Table 4.3. The thread block size was 64. The particles were initially set up on a lattice as shown by the code snippet in Figure 4.27. The reason why *X0* is multiplied by 1.012 is because for the smallest particle smoothing length when *X0=DXL* the simulation did not work for the full 1 second, but did for the other two particle resolutions. This slight shift produced simulations for the full 1 second for each of the three particle smoothing lengths, except for the smallest for the shared memory approach. The codes producing these results were executed on the NVIDIA S1070 cluster at UK STFC Daresbury Laboratory using just one GPU per simulation.

A snapshot of the simulation at 0.9 seconds with 2628 fluid particles from the texture memory approach is given in Figure 4.28, and a snapshot of the simulation at 0.9 seconds with 2628 fluid particles from the shared memory approach is given in Figure 4.29. Though the profiles look similar, they are slightly different and the particle densities are different.

The results in Table 4.3 show that the texture memory method is faster than the shared memory approach, and that this difference increases with problem size. But they also show that a different number of iterations was required, implying different time steps. So what could be happening? In an effort to resolve this problem the computation to

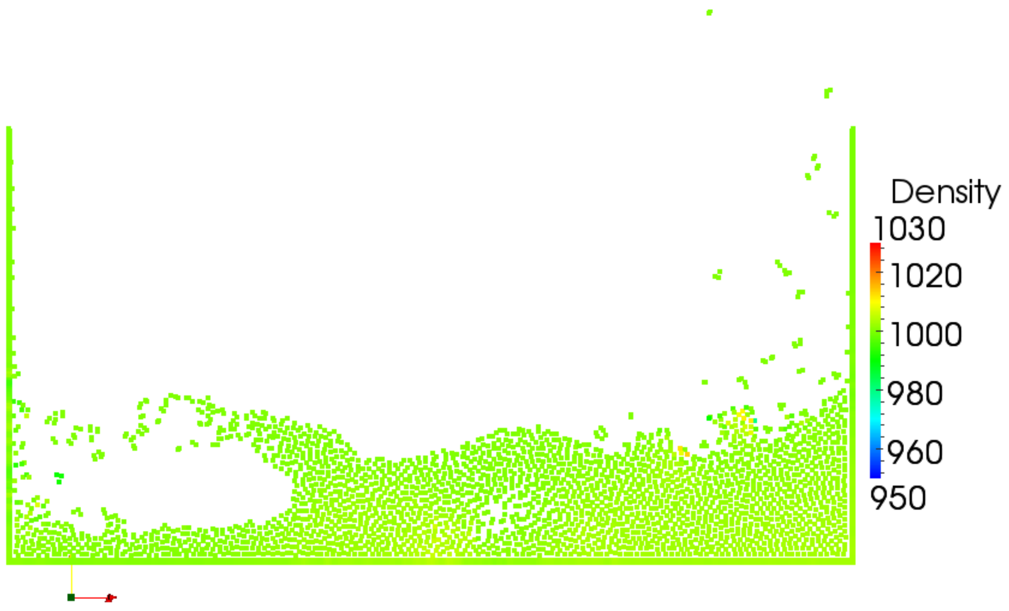


Figure 4.28: The collapse at t=0.9 s from the texture memory method with 2628 fluid particles

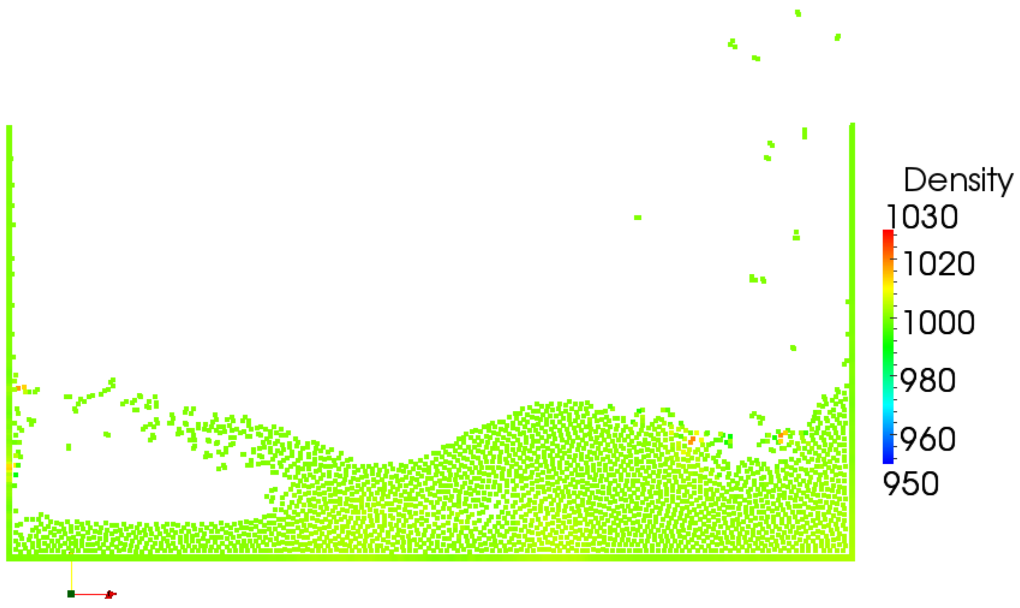


Figure 4.29: The collapse at t=0.9 s from the shared memory method with 2628 fluid particles

| | | Method | | | |
|---|---|---|---|---|---|
| Problem Spec | | Texture | | Shared | |
| DXL (mm) | Particles | Time | Iterations | Time | Iterations |
| 8 | 648 | 0.39 | 14816 | 0.69 | 14813 |
| 4 | 2628 | 0.89 | 29695 | 2.73 | 29663 |
| 2 | 10440 | 3.18 | 59911 | 29.96 | 59830 |

Table 4.3: The execution times in minutes for Koshizuka & Oka simulation for 1 second of real time for three different particle resolutions

```
rhocont = -massj*(V.x*xdwdx + V.y*ydwdx)
        + massj*(n.x*xdwdx + n.y*ydwdx)*Cij*(rhoj - myRho)/rhoj;
```

Figure 4.30: The computation of rate of change of density for each particle interaction

find the rate of change of density, as shown in Figure 4.30, was examined while the program ran in emulation mode on the host, i.e. CPU. Emulation mode emulates the GPU and allows the *printf* statement to print output to file from the kernel which is not allowed for a kernel executing on the NVIDIA C1060. It was found that even though the data in the registers was the same the outcome could be slightly different for the two methods, as shown in Table 4.4. The contribution to the rate of change of density from this particular interaction is different. Is this occuring on the GPU too? The difference between the number of iterations is a small percentage of the total number of iterations, and the visual results imply that the cause of the difference in the number of iterations is not algorithmic. For example, it could be possible that the synchronisation points in the shared memory method are in the wrong place, but this sort of error would produce a much greater difference. And besides, the shared memory code was adapted from the N body simulation code published by Nyland[59], while the texture method was adapted from the CUDA SDK Particles method.

## 4.5 Verification

The accuracy of the results from these implementation can be verified by comparing them with the results obtained by Koshizuka & Oka.

The collapse of the water column over 1 second is shown every 0.2 seconds from Figure 4.31 through to Figure 4.36. The thread blocksize was 64. For comparison pho-

| Component | Texture | Shared |
|---|---|---|
| V.x | -0.00000536 | -0.00000536 |
| V.y | -0.00003271 | -0.00003271 |
| n.x | 0.70710677 | 0.70710677 |
| n.y | -0.70710677 | -0.70710677 |
| xdwdx | 6760297.50000000 | 6760297.50000000 |
| ydwdx | -6760297.50000000 | -6760297.50000000 |
| Ci | 17.60555649 | 17.60555649 |
| Cj | 17.60929871 | 17.60929871 |
| Cij | 17.60929871 | 17.60929871 |
| massj | 0.01108247 | 0.01108247 |
| myRho | 1008.67572021 | 1008.67572021 |
| rhoj | 1008.74707031 | 1008.74707031 |
| rhocont | 129.91999817 | 129.91952515 |

Table 4.4: The components of a contribution to the rate of change of density

tographs taken by Koshizuka & Oka of the collapse are given in Figure 4.37[61]. For the simulation giving these results, the initial particle spacings DXL=DYL=8 mm, and X0=DXL, Y0=DYL for the fluid particle set up as given in Figure 4.27 giving a total of 648 particles, the same number that Koshizuka & Oka reported using. The texture memory method was used to produce the results.

The same particle resolutions as given in Table 4.3 were also used to measure the column front and height as it collapsed. The column front is shown in Figure 4.38 and compared with the experimental data obtained by Martin & Moyce[6]. The readings from the simulation were taken every 0.01 seconds until 0.3 seconds of real time had elapsed. There is no obvious convergence towards the experimental data with increasing number of particles. There also appears to be a slightly increased acceleration of the front at the beginning of the simulation which then follows approximately the same speed as experimental data.

## 4.6 The Effect of Thread Block Size on Execution Time

The CUDA SDK comes with an occupancy calculator which acts as a guide to suggest the optimum thread block size for a kernel. The information the calculator needs is obtained by using the flag *–ptxas-options="-v"* when compiling. When this flag is used the following information is provided by the compiler

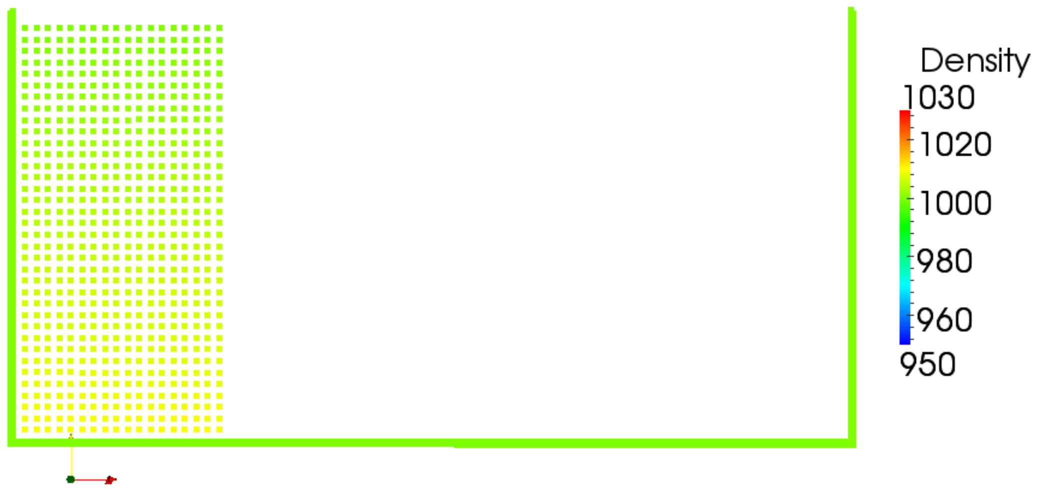


Figure 4.31: The simulation of column collapse at t=0s

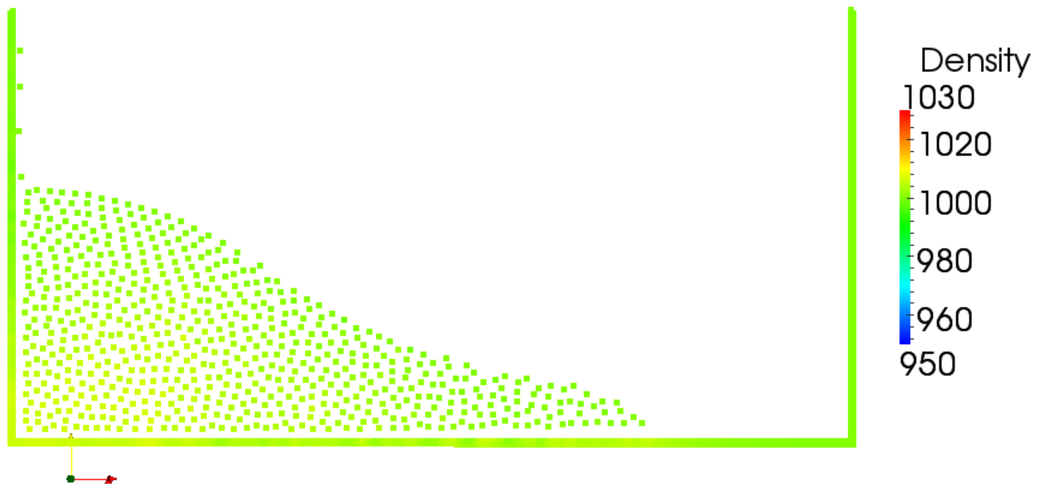


Figure 4.32: The simulation of column collapse at t=0.2s

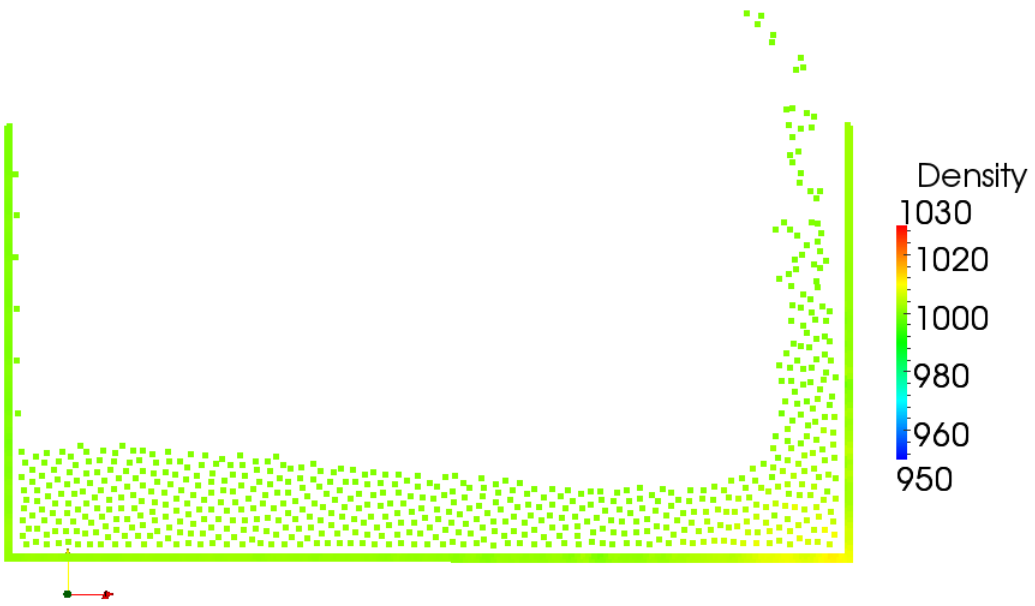


Figure 4.33: The simulation of column collapse at t=0.4s

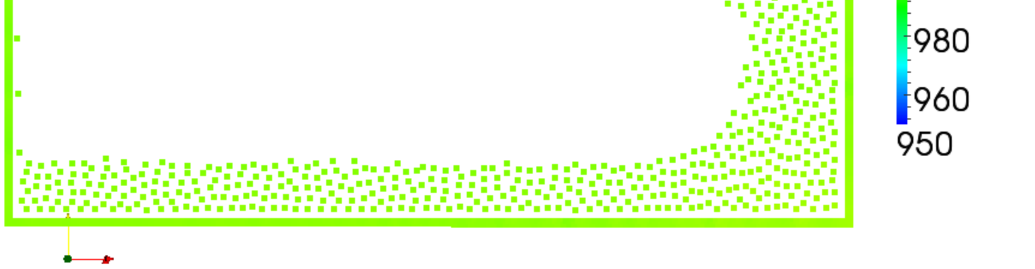


Figure 4.34: The simulation of column collapse at t=0.6s

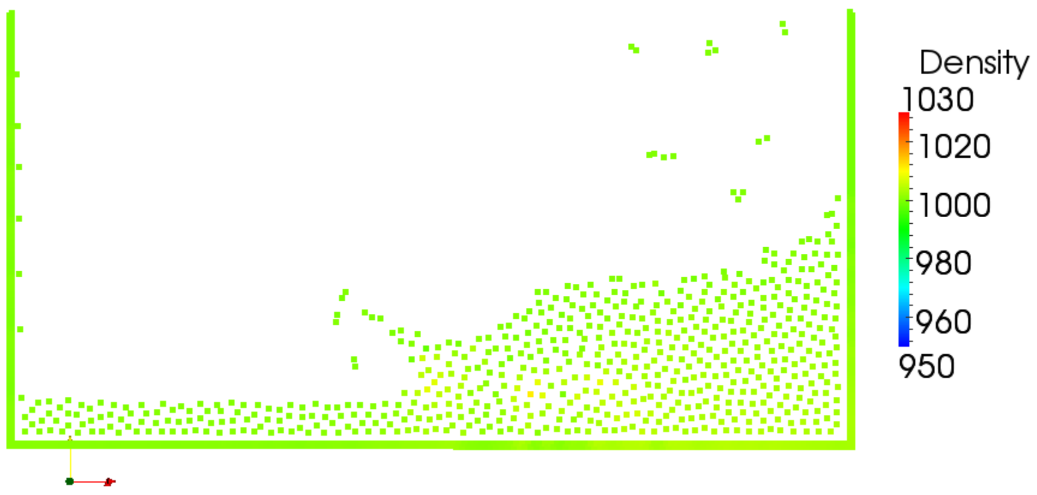


Figure 4.35: The simulation of column collapse at t=0.8s

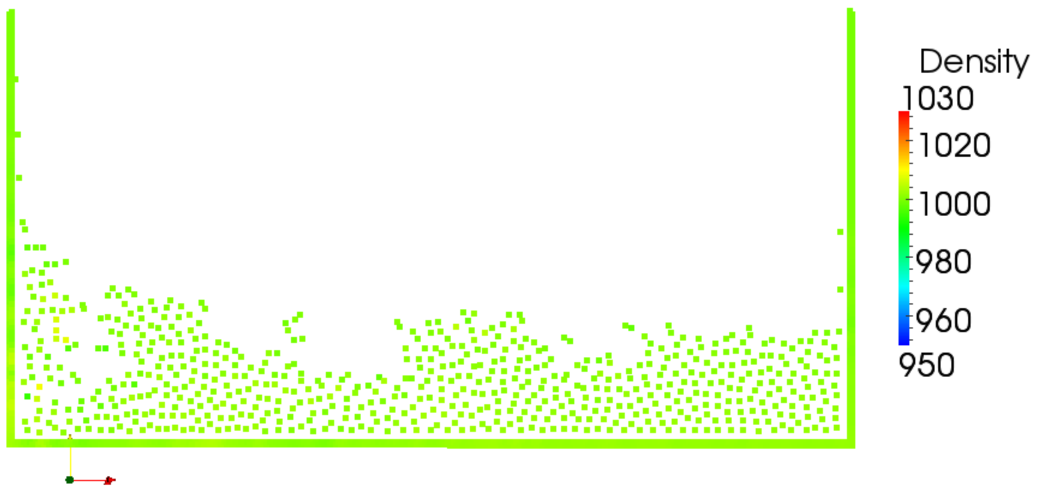


Figure 4.36: The simulation of column collapse at t=1.0s

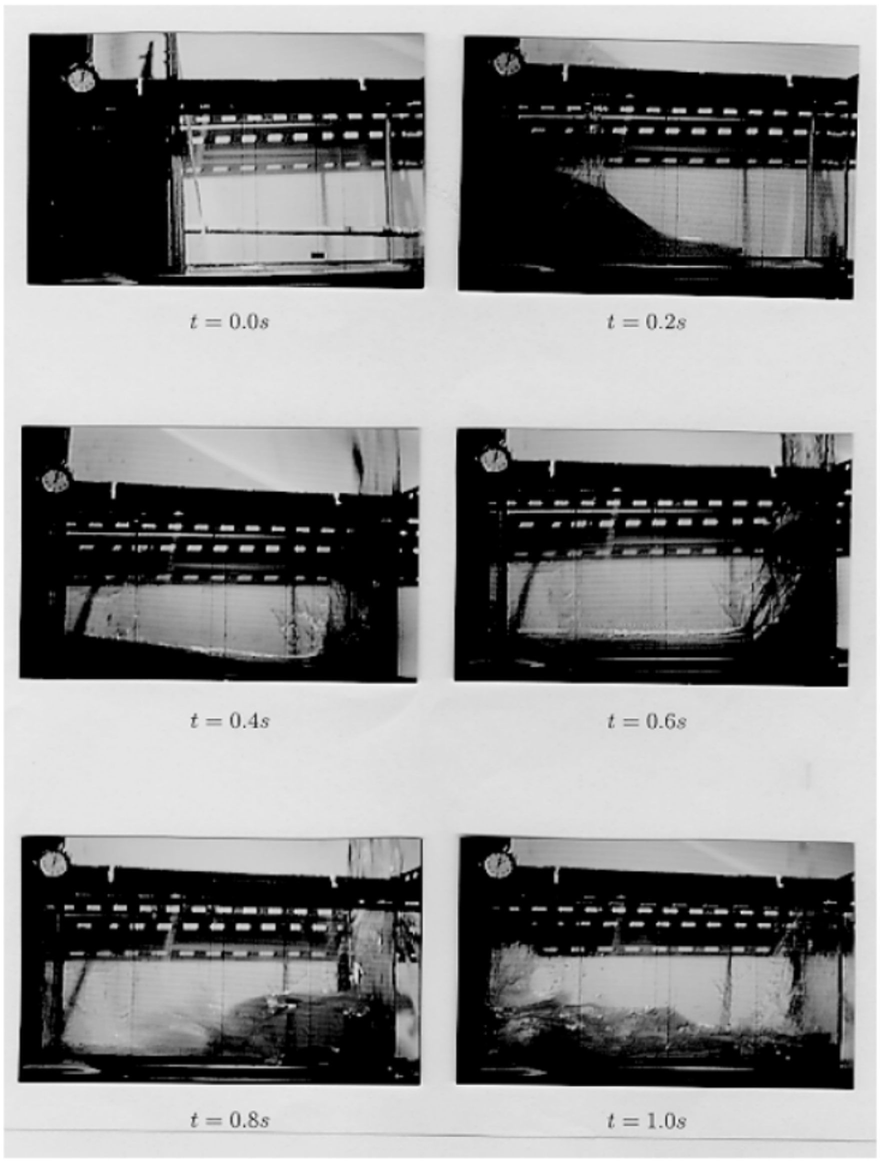


Figure 4.37: Photographs taken by Koshizuka & Oka of the collapse

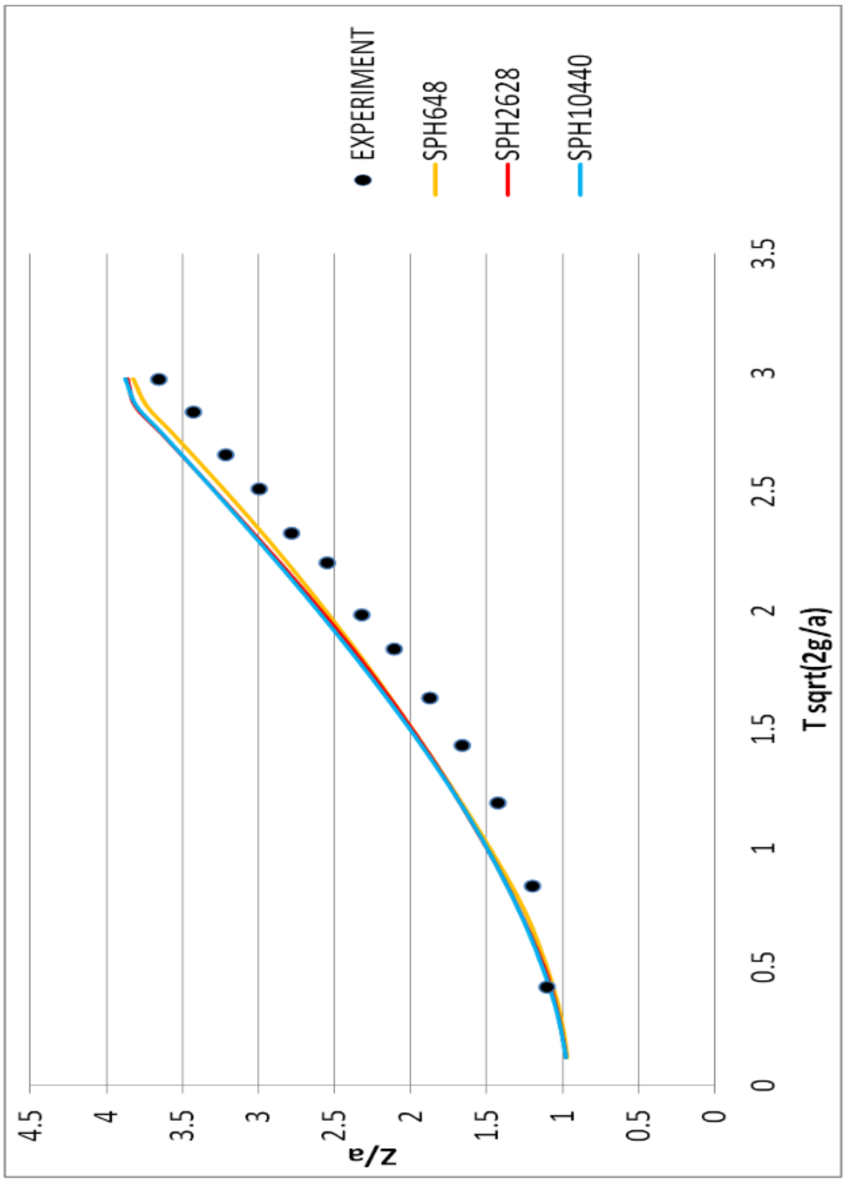


Figure 4.38: The location of the water column front in the simulation of column collapse

```
ptxas info    : Compiling entry function
'_Z15CalculateForcesP5uint2PjPfP6float2ff'
ptxas info    : Used 37 registers, 56+52 bytes smem, 40 bytes cmem[1]

ptxas info    : Compiling entry function
'_Z28reorderDataAndFindCellStartDP5uint2P6float2S2_PfS3_S3_S3_PiS2_S2_S3_S3_S3_S3_S4_Pj'
ptxas info    : Used 9 registers, 1172+144 bytes smem, 8 bytes cmem[1]

ptxas info    : Compiling entry function '_Z9calcHashDP6float2P5uint2f'
ptxas info    : Used 6 registers, 36+32 bytes smem, 16 bytes cmem[1]

ptxas info    : Compiling entry function
'_Z16integratekernel2PfS_S_P6float2S1_S1_S1_S1_S_fS_S_S_Pi'
ptxas info    : Used 10 registers, 128+120 bytes smem, 20 bytes cmem[1]

ptxas info    : Compiling entry function
'_Z16integratekernel1PfS_P6float2S1_S1_S_fS_S_S_Pi'
ptxas info    : Used 9 registers, 104+96 bytes smem, 20 bytes cmem[1]

ptxas info    : Compiling entry function
'_Z10savekernelPfS_P6float2S1_S1_S1_f'
ptxas info    : Used 5 registers, 68+64 bytes smem
```

Figure 4.39: The resources required for each kernel

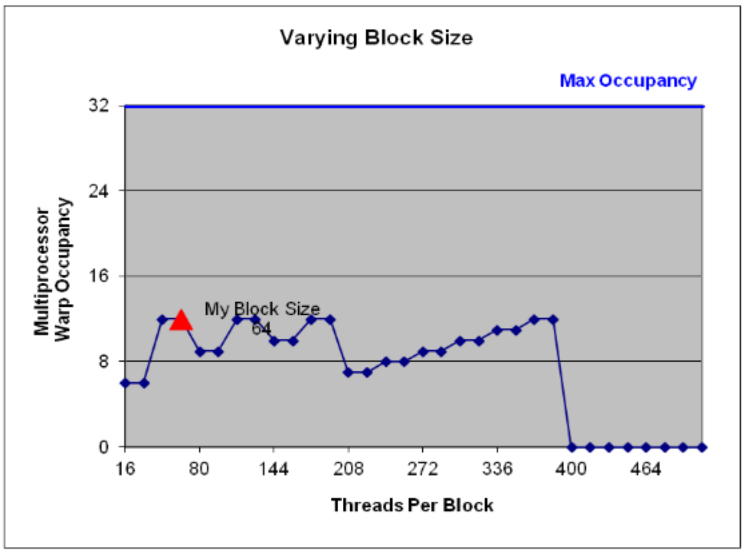
- the number of registers required per thread
- the size of shared memory required per thread
- the size of constant memory required per thread

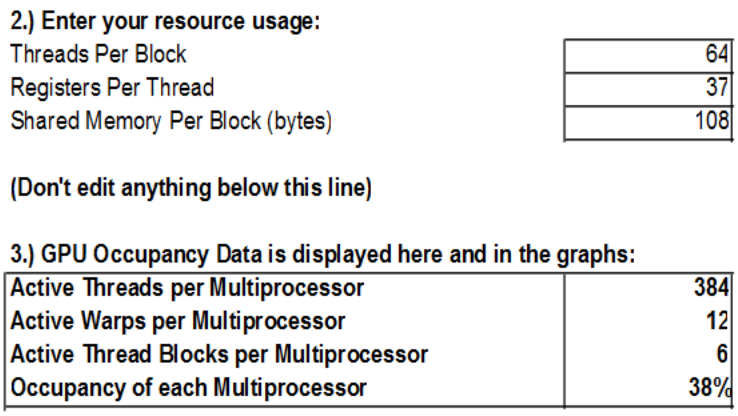
To show the effect of block size on this problem the flag to report register use etc. by the compiler will be used to build the project, and the results from the CUDA occupancy calculator shown. Then different block sizes will be used and their execution times reported to show how block size can affect performance.

## 4.6.1 Occupancy

When the flag *–ptxas-options="-v"* is used to compile the source code the compiler reports how many registers and how much shared and constant memory is required for each kernel. For this source code these are shown in Figure 4.39. From the output produced by the compiler shown in Figure 4.39, the kernel *CalculateForces*, the largest of the kernels, requires 37 registers, 108 bytes of shared memory and 40 bytes of constant memory. The results of using the CUDA occupancy calculator with these values is shown in Figure 4.40.

The occupancy calculator is designed to give a clue to the optimal number of registers and thread block size for each kernel. For example, it may be possible to reduce the





**2.) Enter your resource usage:**

| | |
|---|---|
| Threads Per Block | 64 |
| Registers Per Thread | 37 |
| Shared Memory Per Block (bytes) | 108 |

**(Don't edit anything below this line)**

**3.) GPU Occupancy Data is displayed here and in the graphs:**

| | |
|---|---|
| **Active Threads per Multiprocessor** | **384** |
| **Active Warps per Multiprocessor** | **12** |
| **Active Thread Blocks per Multiprocessor** | **6** |
| **Occupancy of each Multiprocessor** | **38%** |

Figure 4.40: The occupancy for the CalculateForces kernel with variable block size

| NUMTHREADS | Execution Time (ms) | ntotal | numblocks |
|---|---|---|---|
| 32 | 52451.42 | 3840 | 120 |
| 64 | 52408.82 | 3840 | 60 |
| 128 | 52522.25 | 3840 | 30 |
| 256 | 54561.86 | 3840 | 15 |

Table 4.5: The execution times for variable block size with 2628 fluid particles

number of registers in a kernel by splitting the kernel into more than one kernel, but this can add overhead in kernel set up. For the *CalculateForces* kernel the occupancy is 38%. This is calculated as

$$occupancy = 100 \text{ x } ActiveWarps/MaxActiveWarps \qquad (4.12)$$

where *ActiveWarps* is calculated as 12 by the occupancy calculator using the threads per block (defined by the user), and the registers per thread and the shared memory per block produced from the ptxas at compile time. For the T10 the maximum number of active warps *MaxActiveWarps* is 32. To illustrate the effect of thread block size on performance the code described for the texture memory implementation for 4001 fluid particles was used with thread block sizes of 32, 64, 128 and 256 threads per block. The timing results from this are given in Table 4.5. In this simulation there are 2628 fluid particles. The total number of particles, including padding particles, is given in the column *ntotal*, takes the total number of particles in the simulation, fluid, boundary and padding, to an integer multiple of the thread block size given in the column *NUMTHREADS*.

## 4.7 Conclusions

The particular SPH algorithm implemented in this chapter successfully simulates the experiment of Koshizuka & Oka with a good degree of accuracy, even with just 648 fluid particles.

The extra effort of implementing the texture memory approach is justified by the performance results, with the texture memory approach being much faster than the shared memory approach, and that gain in performance increases as the number of particle increases.

The slight difference in the results from the two implementations, using shared memory and texture memory, as shown in Figure 4.28 and Figure 4.29 could be due to the compiler producing different sequences of instructions, or the order of execution of blocks of threads decided by the block scheduler is different, or both. Both of these could lead to the sequence of instructions and their associated data being processed in a different order from each other, and machine precision propagating any rounding or truncation errors during the computation of the expressions would more than likely produce sightly different outcomes, as observed.

The thread block size can have a significant effect on performance, and help on calculating the optimum size can be found from the CUDA occupancy calculator which suggests the optimal block size for a particular kernel when the user inputs data, such as the number of registers, which are produced by the compiler when the *–ptxas-options="-v"* flag is used in the compile command.

# Chapter 5

# Riemann Solvers in SPH

One of the main disadvantages of WCSPH can be the distinct fluctuations in the pressure field. The general flow may be satisfactorily described by the solution but the pressure field could be significantly smoothed and improved. One idea that has been suggested to smooth the pressure field is the use of density filters which reset the density after a fixed number of time steps, but this is computationally expensive. The reputation of Riemann solvers in Finite Volume applications led to the SPH community investigating the potential of using Riemann solvers in SPH.

There is still some debate about the use of Riemann solvers in SPH. Research into Riemann solvers in SPH has shown that solutions can be significantly improved and very accurate, but most if not all of this research has been based on using SPH for gases without the use of boundaries.

Ferrari *et al.*[42] looked at using one formulation of a SPH algorithm using one Riemann solver in a simple test looking at a drop of water and concluded that Riemann solvers are too diffusive, even when using an exact Riemann solver. Antuono *et al.*[62] came to the same conclusion. However this conclusion is based on only one SPH algorithm using just one Riemann solver in one simple test case involving water. There are a number of formulations of SPH using Riemann solvers, and a number of Riemann solvers, approximate and exact, combinations of which have yet to be investigated, in violent and non-violent simulations, and involving the use of boundaries.

On the other side of the debate Rogers *et al.*[63] praise the SPH formulation of Vila[64] with the HLLC ARS[65] concluding that water waves can be more accurately simulated over a distance due to the accuracy provided by the HLLC ARS. Roubtsova & Kahawita[66] applied the ARS proposed by Parshikov *et al.*[30] to a sloshing problem, a dam break problem and a simulation of The Vaiont Disaster and concluded that the technique is easy to implement and gives satisfactory results. However Omidvar *et al.*[67] report on the sensitivity of MUSCL-based $\beta$-limiters for wave generation in a numerical wave tank.

The success of Riemann solvers in Finite Volume applications should not be quickly dismissed. The use of a Riemann solver eliminates the need for a viscous term in the SPH equations, and Riemann solvers require little or no tuning. A disadvantage of Riemann solvers is their complexity and potential for requiring massive computation, but with the GPU now offering the potential for low cost high performance computing a thorough investigation of Riemann solvers, both approximate and exact, in SPH for hydrodynamic applications involving boundaries is now available.

This chapter looks at the use and success or otherwise of ARS in SPH.

## 5.1 SPH Algorithms using Approximate Riemann Solvers

There have been several attempts at implementing SPH with an approximate Riemann solver (ARS).

Monaghan [68] does not explicitly use a Riemann solver but assumes that the artificial viscosity term $\Pi_{ab}$ introduced in Chapter 2 on SPH can be approximated with a signal velocity $v_{sig}(a,b)$ between two interacting particles, such that the artificial viscosity

$$\Pi_{ab} = -\frac{K v_{sig}(a,b)\underline{v}_{ab}\cdot\underline{j}}{\overline{\rho}_{ab}} \tag{5.1}$$

where

$$v_{sig}(a,b) = \left(c_a^2 + \beta(\underline{v}_{ab}\cdot\underline{j})^2\right)^{1/2} + \left(c_b^2 + \beta(\underline{v}_{ab}\cdot\underline{j})^2\right)^{1/2} - \underline{v}_{ab}\cdot\underline{j} \tag{5.2}$$

in which $c_k$ is the speed of sound for particle $k$, $\underline{v}_{ab} = \underline{v}_a - \underline{v}_b$ where $\underline{v}_k$ is the velocity of particle $k$, and $\beta$ is a parameter that needs tuning for each simulation.

There are a number of sets of governing equations in Smoothed Particle Hydrodynamics. One particular set of SPH equations was used and extended by Parshikov *et al.*[69][30][70] by solving a Riemann problem at the contact point between two particles to give the following approximate Riemann solution

$$U_{ij}^{*R} = \frac{U_j^R \rho_j C_j + U_i^R \rho_i C_i - P_j + P_i}{\rho_j C_j + \rho_i C_i} \tag{5.3}$$

$$P_{ij}^{*} = \frac{P_j \rho_i C_i + P_i \rho_j C_j - \rho_j C_j \rho_i C_i \left(U_j^R - U_i^R\right)}{\rho_j C_j + \rho_i C_i} \tag{5.4}$$

These approximate solutions are then substituted into the governing SPH equations.

Vila[64] derives a set of SPH equations that describe the Euler equations with an intrinsic viscosity, and are accepted by particular sections of the SPH community as being accurate. In their study of a Caisson breakwater Rogers *et al.*[63], who make up part of the SPHysics group, express their preference for this algorithm in their examination of using the Vila formulation of SPH for that study. The implementation of this algorithm on a single GPU using the HLLC solver is discussed and compared with the SPH algorithm proposed by Ferrari *et al.* and another but simpler SPH formulation later in this chapter. The derivation of the Vila SPH equations is too long and complex for this thesis, but because this algorithm is examined later in this chapter to show how Riemann solvers can reduce dissipation, the SPH formulation of Vila is

$$\frac{d\underline{x}_i}{dt} = \underline{v}(\underline{x}_i, t) \tag{5.5}$$

$$\frac{d\omega_i}{dt} = \omega_i \nabla \cdot \underline{v}_i \tag{5.6}$$

$$\frac{d\omega_i\rho_i}{dt} = -\omega_i \sum_{j\in\Omega} 2\omega_j \rho^* (\underline{v}_{ij}^* - \underline{v}^0(\underline{x}_{ij}, t)) \cdot \nabla_i W_{ij} \tag{5.7}$$

$$\frac{d\omega_i\rho_i\underline{v}_i}{dt} = \omega_i f_i - \omega_i \sum_{j\in\Omega} 2\omega_j [P^* + \rho^* \underline{v}_{ij}^* \otimes (\underline{v}_{ij}^* - \underline{v}^0(\underline{x}_{ij}, t))] \nabla_i W_{ij} \tag{5.8}$$

where the superscript $*$ denotes the solution from the Riemann problem between the interacting particles, and the superscript $0$ denotes the mean of the velocities of the interacting particles.

Similar to Vila, Inutsuka[71] derives his own version of the SPH equations which employ the solution to the Riemann problem between each pair of interacting particles, suggesting that the Riemann solution should be found from using either the Riemann solver of either Colella & Woodward[72], or that of van Leer[73].

Cha & Whitworth[74] take a similar but slightly different approach to Parshikov in inserting a Riemann solution into the SPH equations, and call their algorithm Godunov-type Particle Hydrodynamics (GPH). They examine four different sets of GPH equations and use the iterative Riemann solver of van Leer[73] on shock tube and blast wave tests and achieve very good agreement with the analytic solutions.

Molteni & Bilello[75] use the work of Parshikov to compute terms in the Euler equations in a complex and computationally expensive method.

All show very good agreement with exact solutions to well known shock tube problems involving gases without boundaries.

But this thesis looks at problems involving water and boundaries. So how does the use of a Riemann solver in SPH behave when simulating water and boundaries?

## 5.2 Dissipation in SPH with Riemann Solver

As mentioned above the Ferrari group[42] with support from Antuono[62] argue that Riemann solvers are too dissipative, while Rogers *et al.*[63] supported by Roubtsova & Kahawita[66] are in favour of Riemann solvers in SPH for simulating water. This section will examine the performance of a Riemann-based SPH algorithm.

The Vila[64] algorithm using the HLLC ARS has been coded in Fortran for a single CPU and released as open source software by the SPHysics group[20], which this author has translated into CUDA for this work and amended by implementing the Lennard-Jones type boundary conditions of Monaghan[33] and those proposed by Dalrymple & Knio [38]. The Vila SPH algorithm with HLLC ARS was then used to simulate the Martin & Moyce experiment[6] with the three boundary conditions of Monaghan & Kos[34], Monaghan Lennard-Jones[33] and Dalrymple & Knio[38]. This is to test

- how does the Vila SPH algorithm with the HLLC ARS behave in this experiment?

- do the boundary conditions used have any significant influence?

The experiment run by Koshizuka & Oka[11] is quite a violent test but with an initial gentle collapse so it should test the algorithm under both violent and non-violent conditions simultaneously and possibly help to resolve the question surrounding the use of Riemann solvers in SPH for hydrodynamics with boundaries. For each test a fluid particle with constant smoothing length 6 mm was used. A constant distance *DXL* and *DYL* of 4.6875 mm was used to set up the initial particle configuration, with fluid particles arranged on a lattice a distance $d_0$ apart, where

$$d_0 = \sqrt{(DXL^2 + DYL^2)} \tag{5.9}$$

For the HLLC Riemann solver the $\beta$ limiter is used with $\beta = 0.5$.

Boundary particles are spaced a distance $bd_0 = bpf$ x $DXL$ apart.

For each variant of the algorithm below the parameters *X0* and *Y0* are given. These two parameters specify the initial position *(X0,Y0)* of the fluid particles. The fluid particles are set up as shown in the code snippet in Figure 4.27. Also given are the CFL number and *bpf*. For the Lennard Jones boundary treatment the parameter *R0* is also given. This is the radius of the boundary particle inside which it begins to exert a repulsive force on a fluid particle.

For each simulation the front and height of the column were recorded every 0.01 seconds for the first three tenths of a second of the collapse. The front locations are shown in Figure 5.1, and the heights of the columns are shown in Figure 5.2. Experimental data from Martin & Moyce[6] are also shown for comparison.

### 5.2.1 Monaghan & Kos Boundary Treatment

The initial conditions for this simulation were that

- *bpf*=0.5
- X0=3.5DXL, Y0=3.4DYL
- CFL=0.025

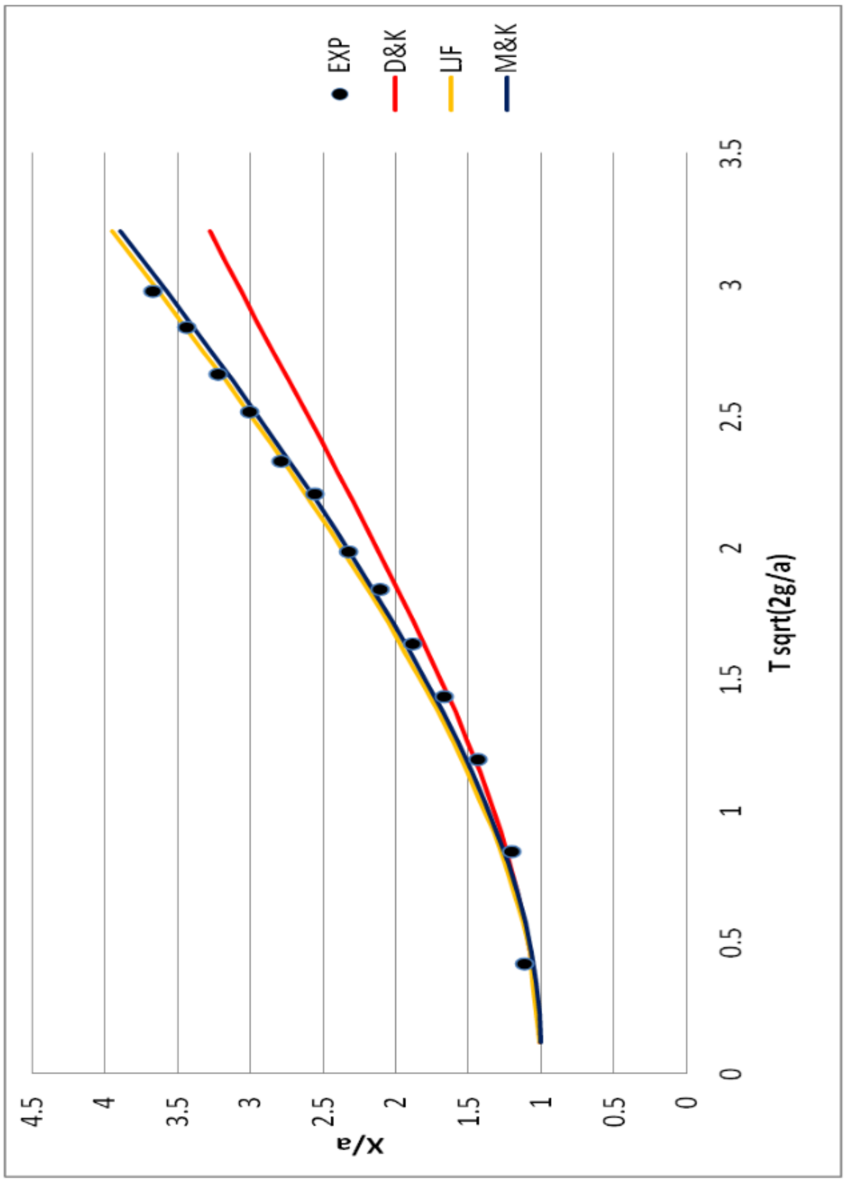


Figure 5.1: The front of the column for the Vila SPH algorithm with different boundary treatments

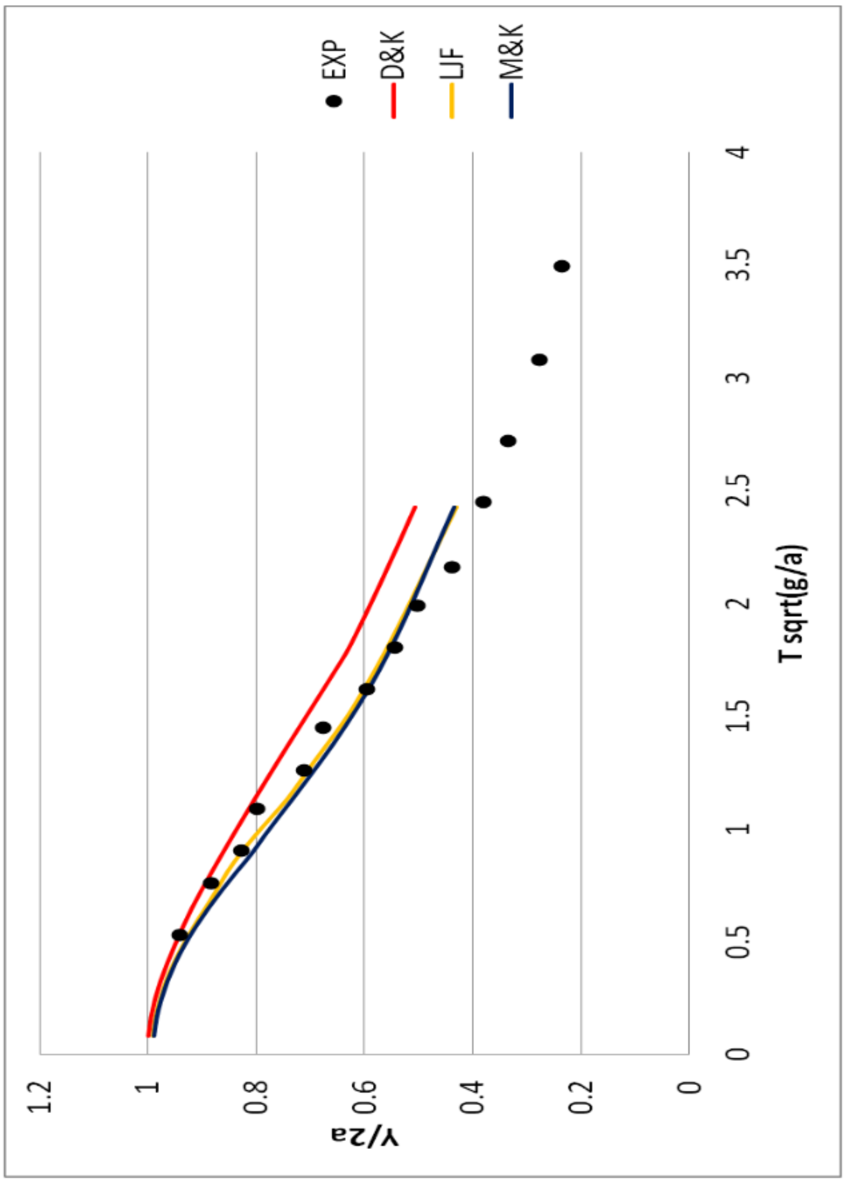


Figure 5.2: The height of the column for the Vila SPH algorithm with different boundary treatments

Snapshots of the collapse at 0 s, 0.2 s, 0.4 s, 0.6 s, 0.8 s and 1 s in Figures 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8 indicate that there is no fragmentation or spray, unlike that provided by the Ferrari algorithm examined in the previous chapter and shown in the photographs taken by Koshizuka & Oka[11].

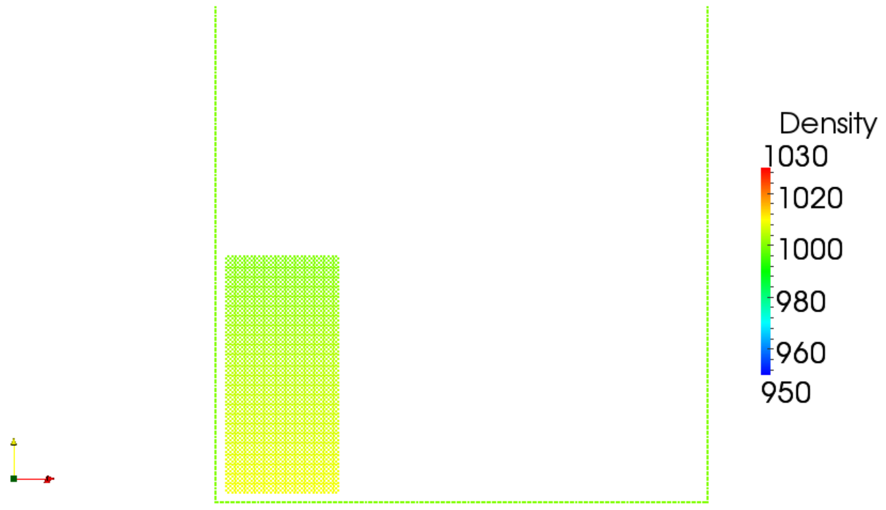


Figure 5.3: Snapshot of collapse of Vila SPH algorithm with Monaghan & Kos boundary treatment at 0 s

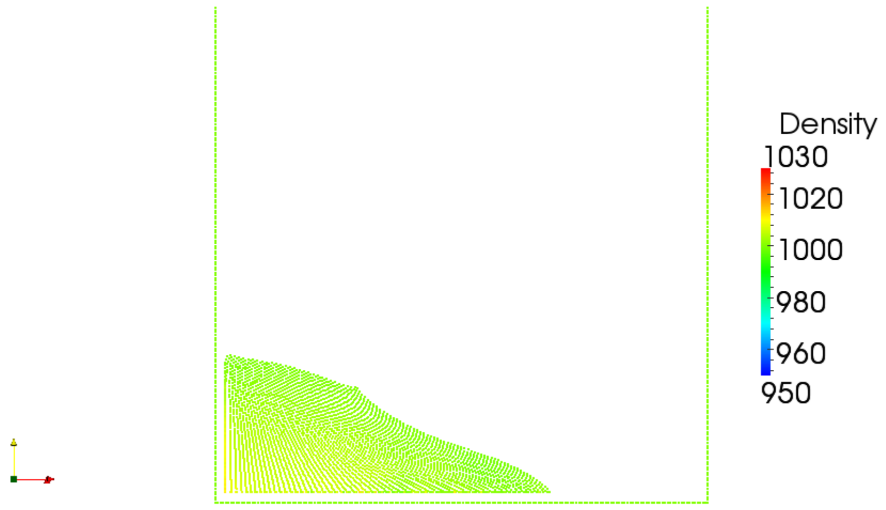


Figure 5.4: Snapshot of collapse of Vila SPH algorithm with Monaghan & Kos boundary treatment at 0.2 s

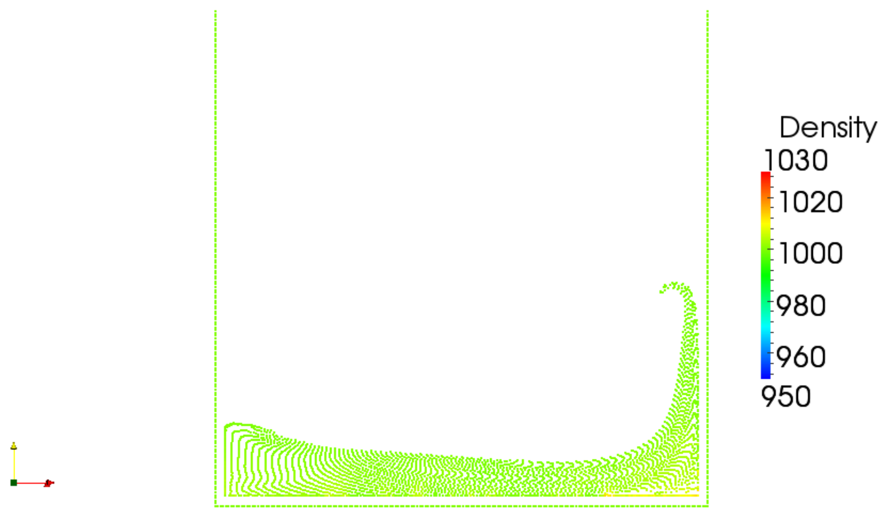


Figure 5.5: Snapshot of collapse of Vila SPH algorithm with Monaghan & Kos boundary treatment at 0.4 s

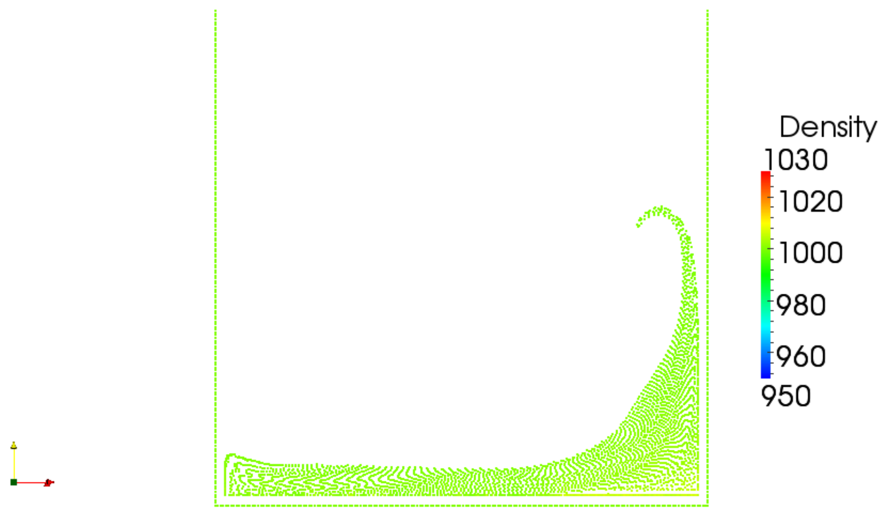


Figure 5.6: Snapshot of collapse of Vila SPH algorithm with Monaghan & Kos boundary treatment at 0.6 s

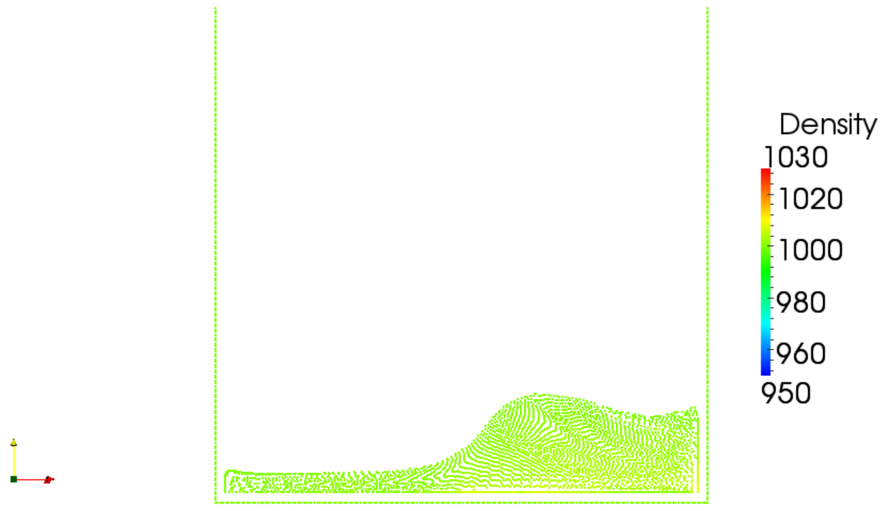


Figure 5.7: Snapshot of collapse of Vila SPH algorithm with Monaghan & Kos boundary treatment at 0.8 s

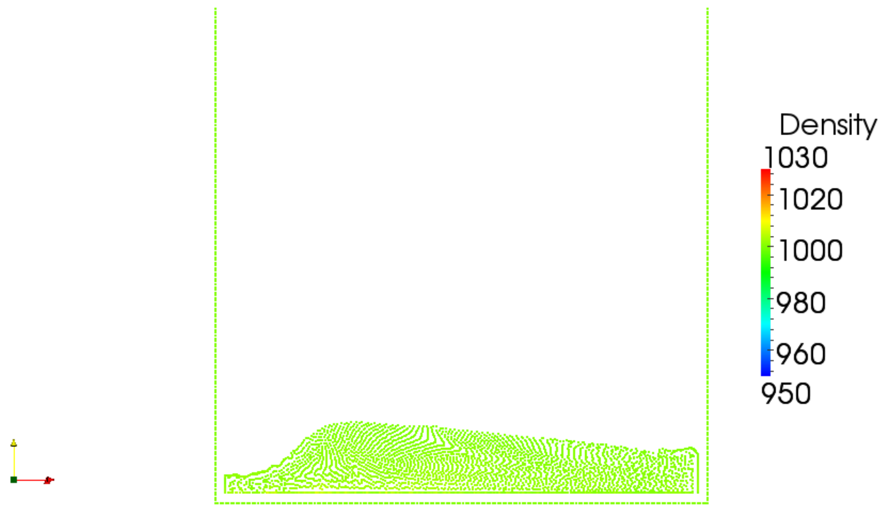


Figure 5.8: Snapshot of collapse of Vila SPH algorithm with Monaghan & Kos boundary treatment at 1 s

### 5.2.2 Monaghan Lennard-Jones Boundary Treatment

The initial conditions for this simulation were that

- *bpf*=0.65
- R0=1.5 DXL

- X0=1.1R0, Y0=R0
- CFL=0.025

Snapshots of the collapse at 0 s, 0.2 s, 0.4 s, 0.6 s, 0.8 s and 1 s in Figures 5.9, 5.10, 5.11, 5.12, 5.13 and 5.14 indicate that, as with the boundary treatment of Monaghan & Kos, there is no fragmentation of the fluid and no spray which is provided by the Ferrari algorithm examined in the previous chapter and shown in the photographs taken by Koshizuka & Oka[11].

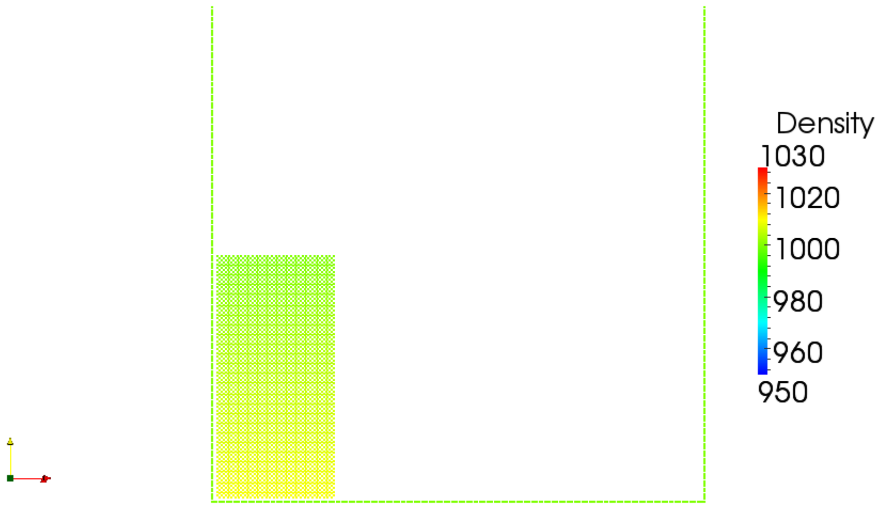


Figure 5.9: Snapshot of collapse of Vila SPH algorithm with Lennard Jones boundary treatment at 0 s

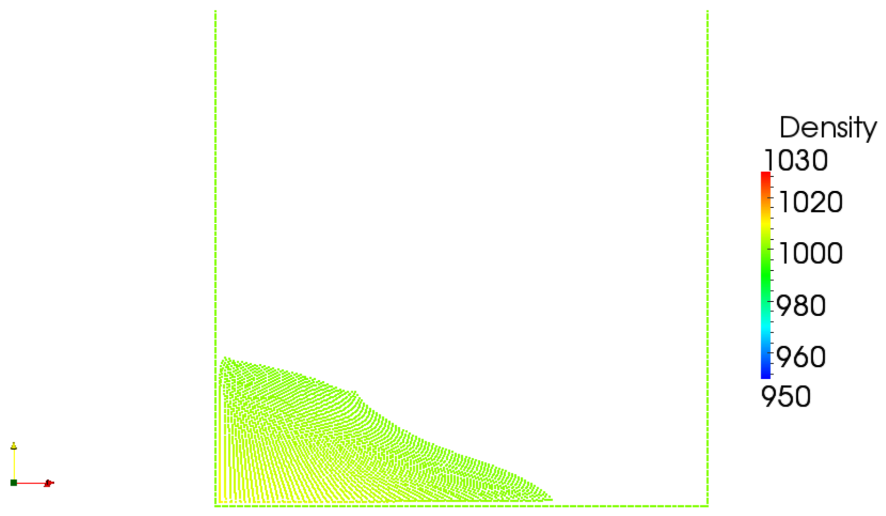


Figure 5.10: Snapshot of collapse of Vila SPH algorithm with Lennard Jones boundary treatment at 0.2 s

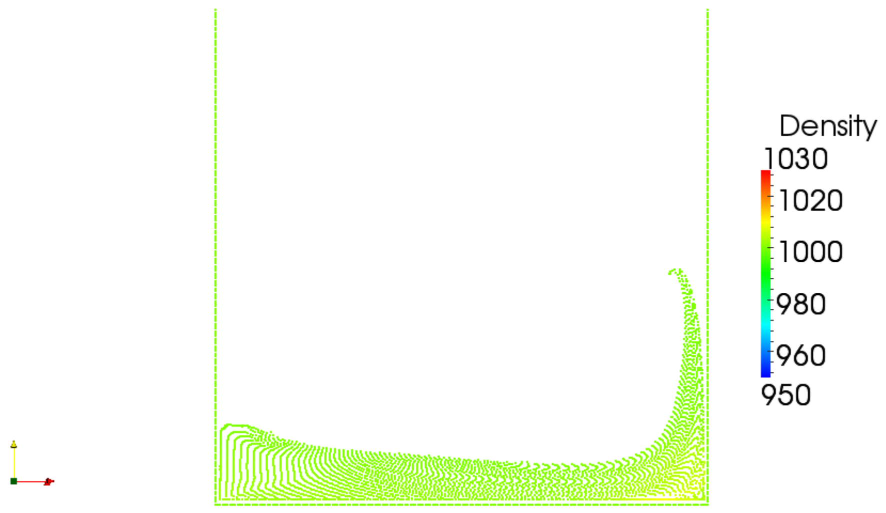


Figure 5.11: Snapshot of collapse of Vila SPH algorithm with Lennard Jones boundary treatment at 0.4 s

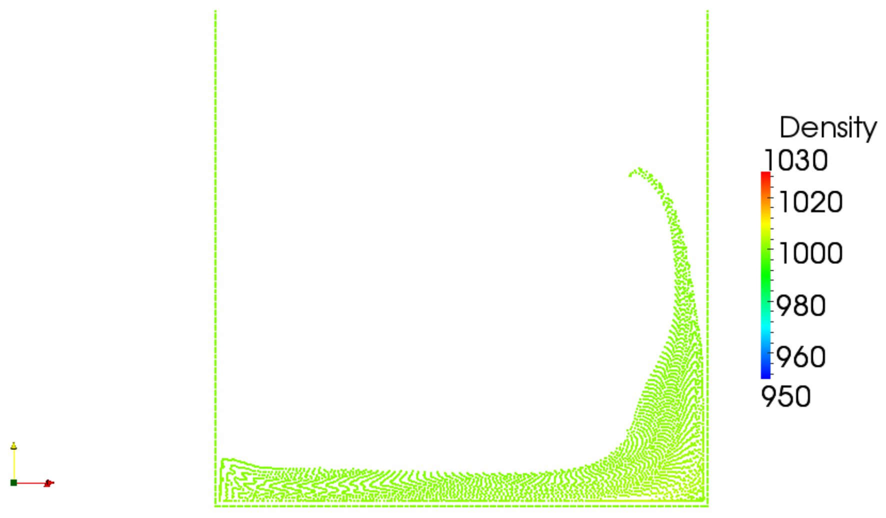


Figure 5.12: Snapshot of collapse of Vila SPH algorithm with Lennard Jones boundary treatment at 0.6 s

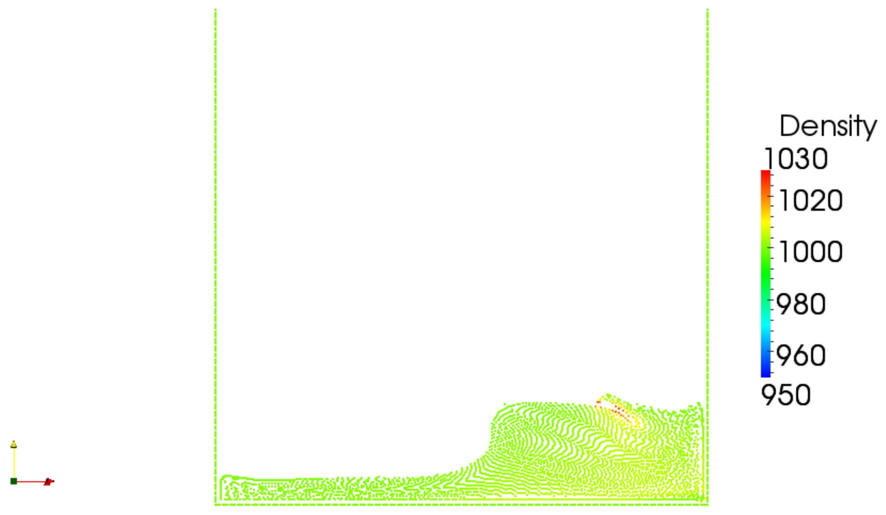


Figure 5.13: Snapshot of collapse of Vila SPH algorithm with Lennard Jones boundary treatment at 0.8 s

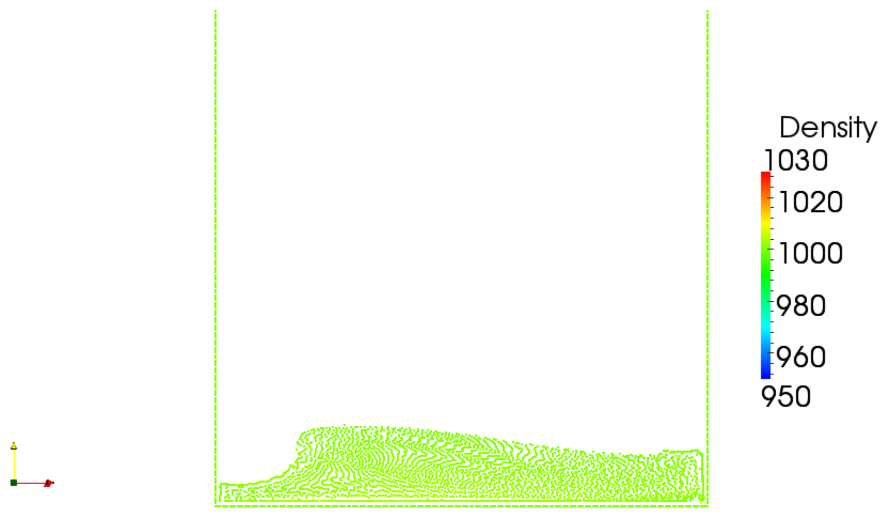


Figure 5.14: Snapshot of collapse of Vila SPH algorithm with Lennard Jones boundary treatment at 1 s

### 5.2.3 Dalrymple & Knio Boundary Treatment

The initial conditions for this simulation were that

- *bpf*=0.65
- R0=1.5 DXL
- X0=1.1R0, Y0=R0
- CFL=0.025

Snapshots of the collapse at t=0 s, 0.2 s, 0.4 s, 0.6 s, 0.8 s and 1 s in Figures 5.15, 5.16, 5.17, 5.18, 5.19 and 5.20 indicate that not only is there no fragmentation or spray with this boundary treatment, but the initial collapse also appears more viscous, which is supported by the results of the front and height of the column as it collapses.

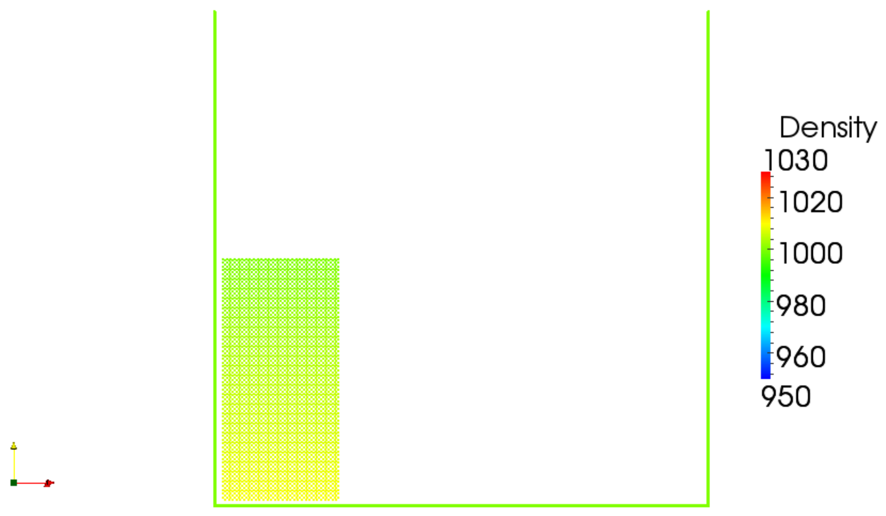


Figure 5.15: Snapshot of collapse of Vila SPH algorithm with Dalrymple & Knio boundary treatment at 0 s

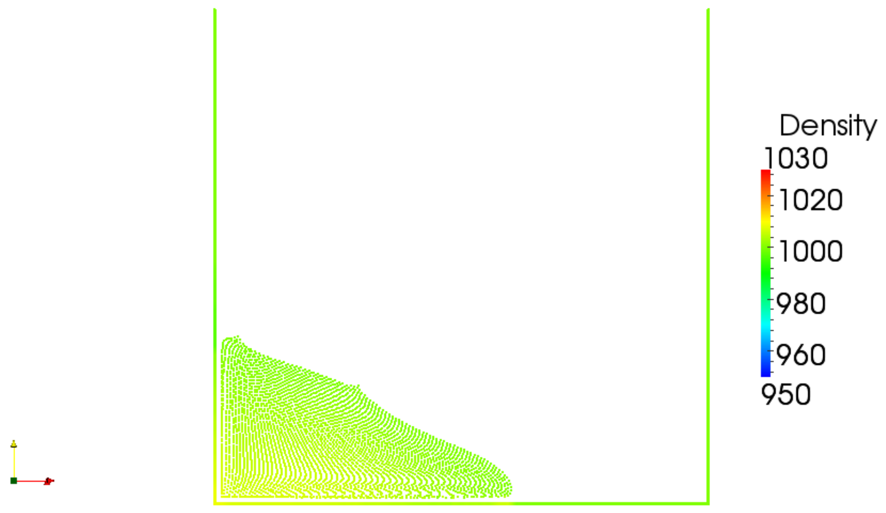


Figure 5.16: Snapshot of collapse of Vila SPH algorithm with Dalrymple & Knio boundary treatment at 0.2 s

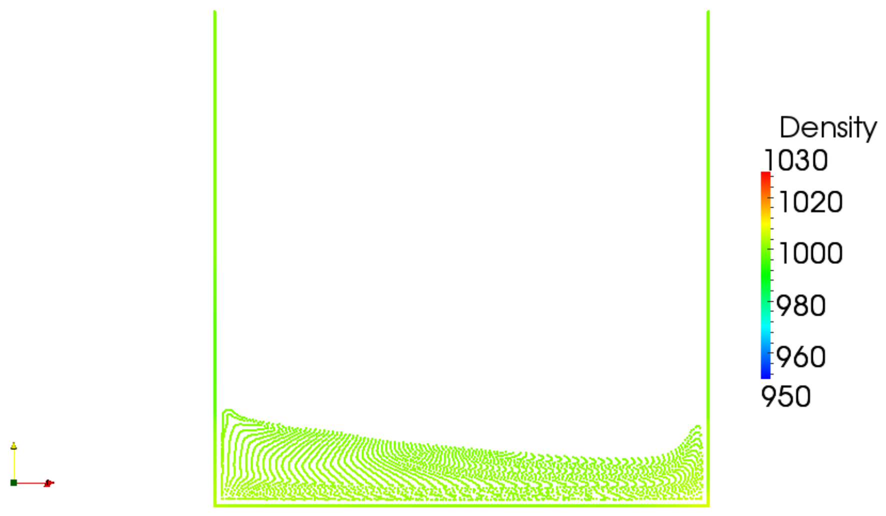


Figure 5.17: Snapshot of collapse of Vila SPH algorithm with Dalrymple & Knio boundary treatment at 0.4 s

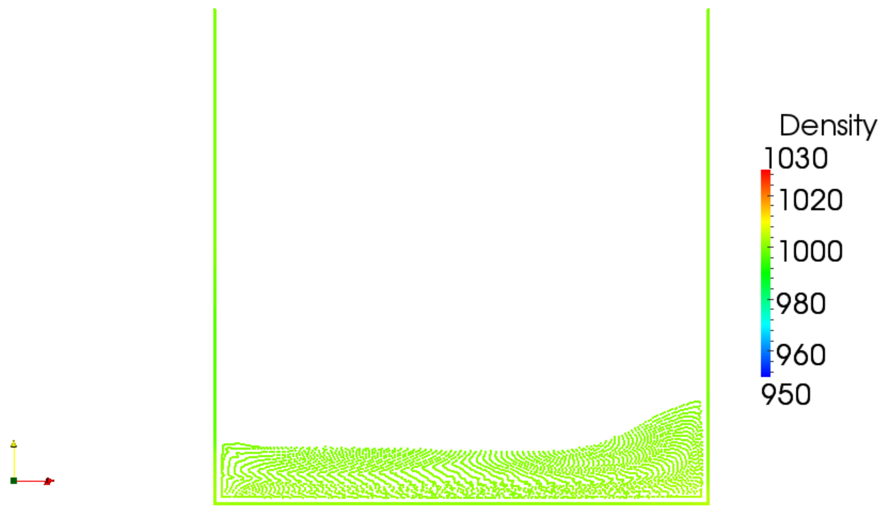


Figure 5.18: Snapshot of collapse of Vila SPH algorithm with Dalrymple & Knio boundary treatment at 0.6 s

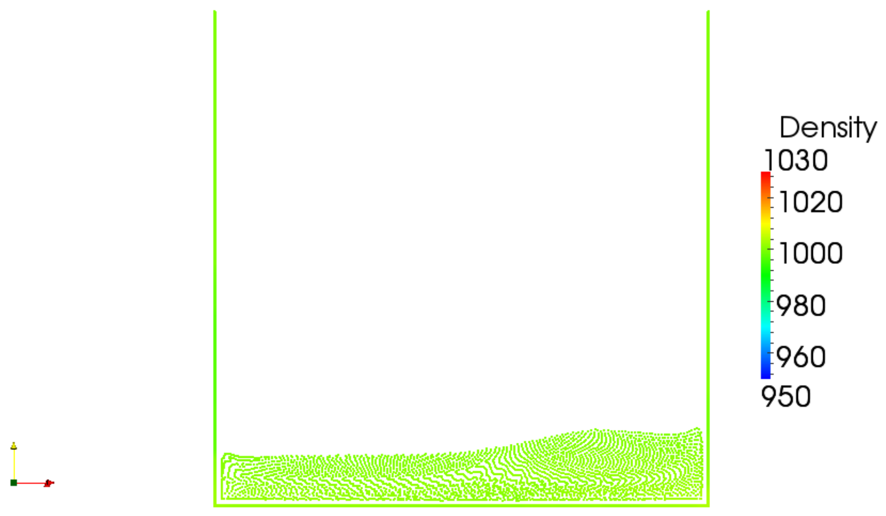


Figure 5.19: Snapshot of collapse of Vila SPH algorithm with Dalrymple & Knio boundary treatment at 0.8 s

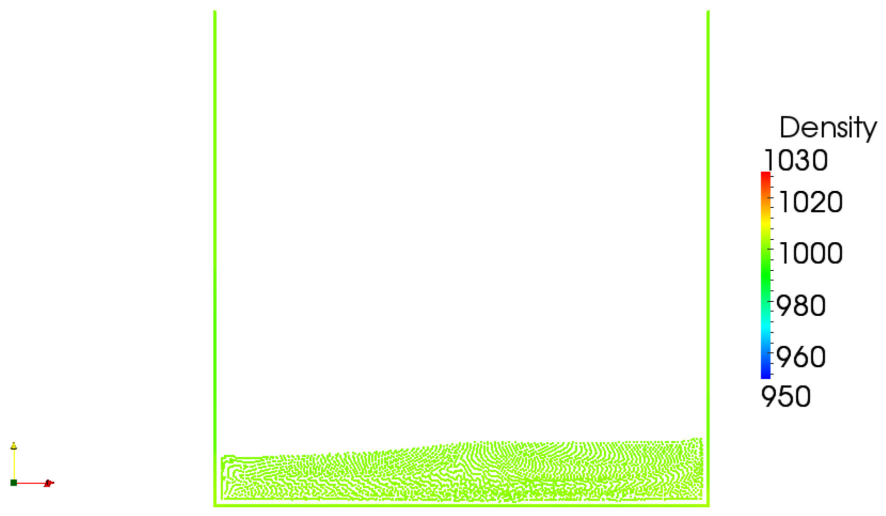


Figure 5.20: Snapshot of collapse of Vila SPH algorithm with Dalrymple & Knio boundary treatment at 1 s

## 5.3 A Solitary Wave

A solitary wave can be set up as the solution to the Boussinesq equations[43] which should travel with constant speed and height. The wave has amplitude $A$, and elevation $\eta$ above a constant depth $D$, where

$$\eta = A\,\mathrm{sech}^2\left[\sqrt{\frac{3A}{4D^3}}(x-ct)\right] \tag{5.10}$$

and the wave celerity $c$ is given by

$$c = \sqrt{g(D+A)} \tag{5.11}$$

The particles initially have zero vertical velocity, and their horizontal velocity $u$ is given by

$$u = \eta\sqrt{g/D} \tag{5.12}$$

A solitary wave with a constant depth *D*=0.21 m and amplitude *A*=0.088 m was set up in a tank of length 10 m with the crest at *x*=0 m, the tank running from *x*=-2 m to *x*=8 m. The particles were initially placed on a regular grid and were 0.01 m apart. Two SPH algorithms were used to simulate this solitary wave as it travelled down the tank. The first was the Vila SPH algorithm with the Monaghan & Kos boundary treatment and $\beta$=0.5 for the limiter. The second was the Ferrari SPH algorithm including the boundary treatment that the Ferrari group proposed. For both algorithms the CFL number was 0.1.

The result of simulating this solitary wave using the Vila SPH algorithm at 0 s, 1 s and 4 s are shown in Figure 5.21, Figure 5.22 and Figure 5.23 respectively.

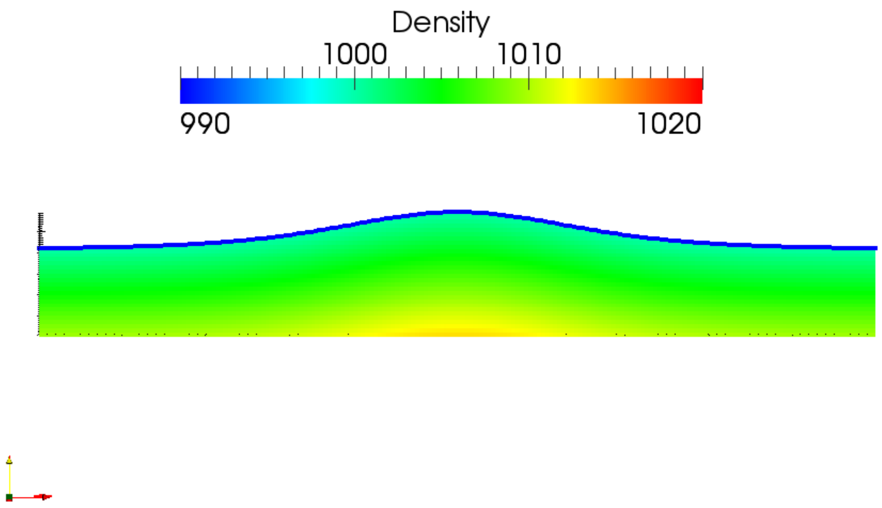


Figure 5.21: The solitary wave at 0 s with the Vila SPH algorithm

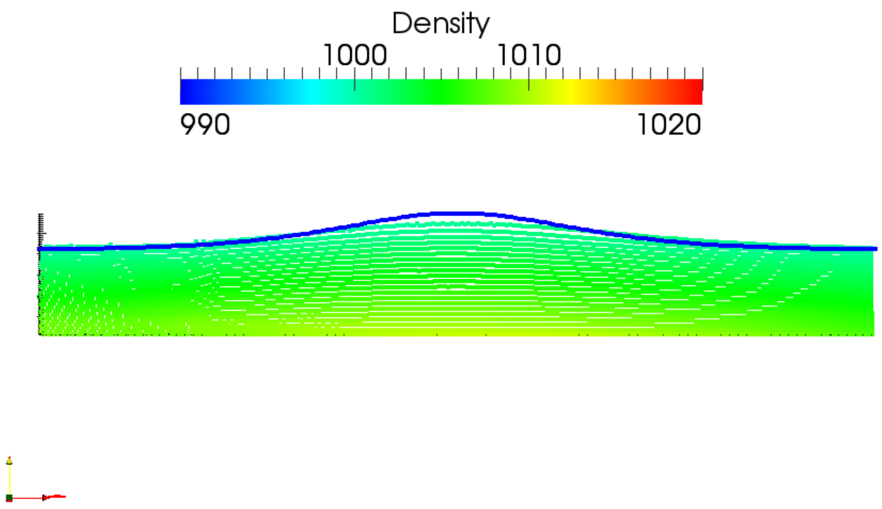


Figure 5.22: The solitary wave at 1 s with the Vila SPH algorithm

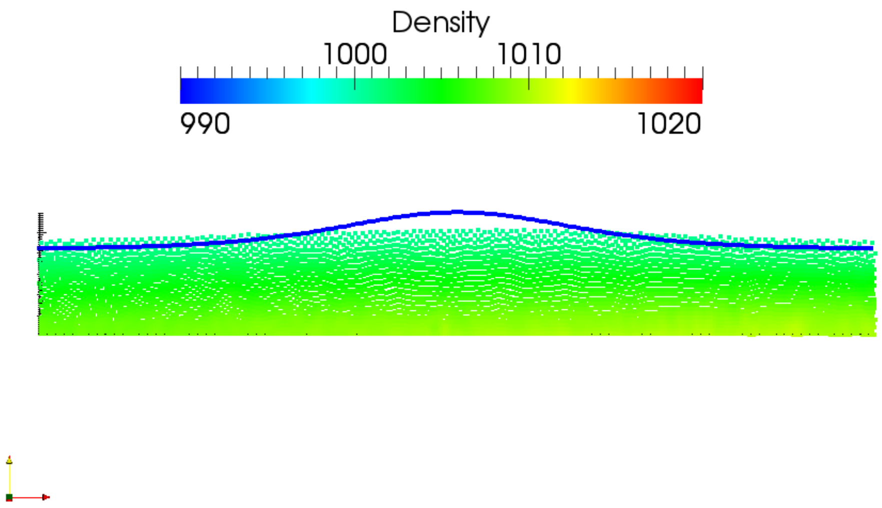


Figure 5.23: The solitary wave at 4 s with the Vila SPH algorithm

The result of simulating this solitary wave with the Ferrari SPH algorithm at 0 s, 1 s and 4 s are shown in Figure 5.24, Figure 5.25 and Figure 5.26 respectively.

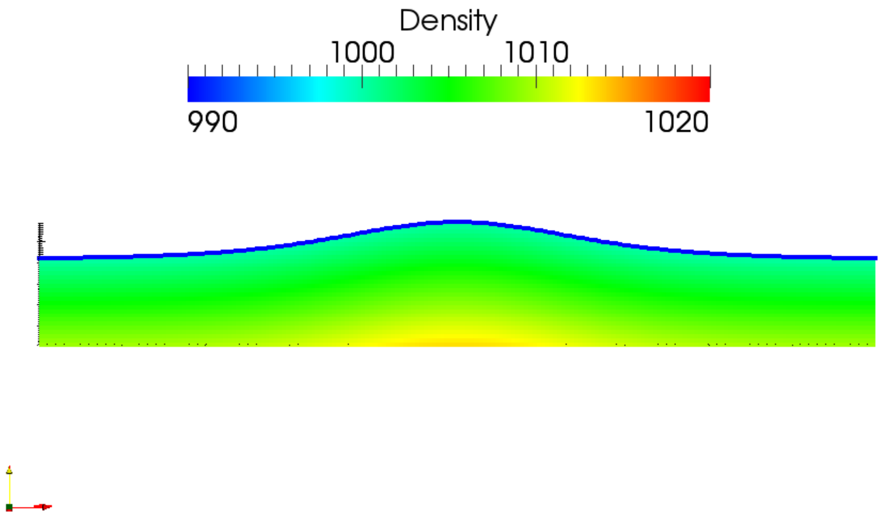


Figure 5.24: The solitary wave at 0 s with the Ferrari SPH algorithm

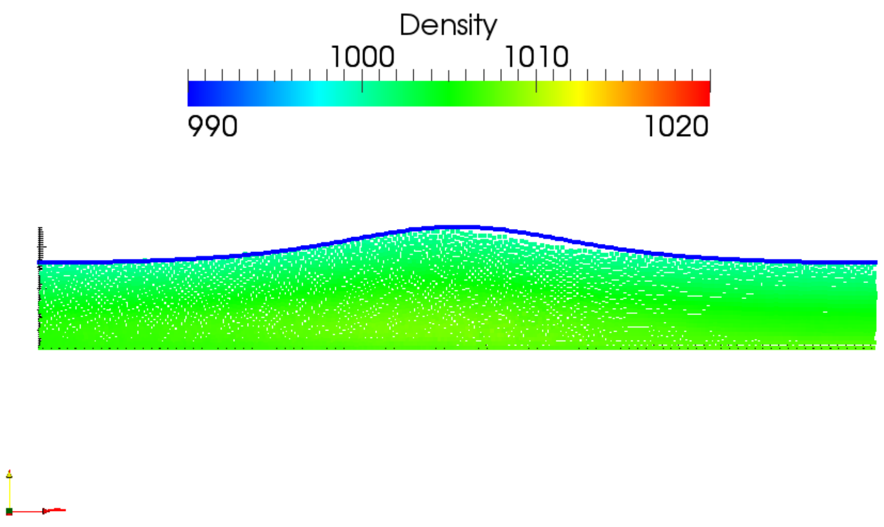


Figure 5.25: The solitary wave at 1 s with the Ferrari SPH algorithm

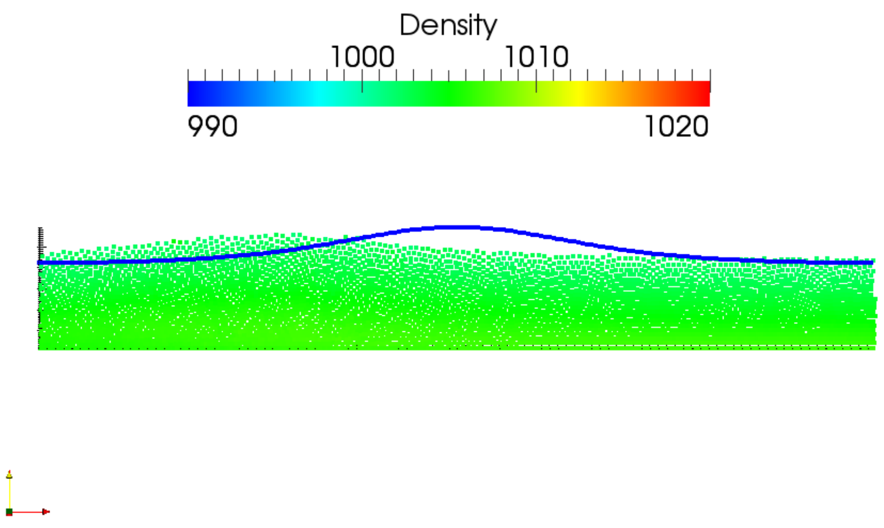


Figure 5.26: The solitary wave at 4 s with the Ferrari SPH algorithm

In both figures the blue line indicates where the solitary wave should be.

Table 5.1 shows the number of fluid particles involved, the execution times to simulate 4 seconds of real time, including times to write output every 1 second of real time, the difference between where the wave crest should be and where the particle with the largest height is, and the loss of height from an initial maximum height of 0.295 m.

| **Algorithm** | **Fluid Particles** | **Time (mins)** | **Crest** | **Height Loss (m)** |
|---|---|---|---|---|
| Vila | 21648 | 8.00 | 0.09 | 0.04 |
| Ferrari | 21648 | 18.43 | -0.38 | 0.01 |

Table 5.1: The execution times for the Vila and Ferrari algorithms to simulate 4 s of a solitary wave

The results in Table 5.1 indicate that the Ferrari algorithm preserves the wave height more than the Vila algorithm does, but that the highest particle is some distance behind where it should be theoretically, and this is obvious as shown in Figure 5.26. On the other hand the Vila algorithm although losing more height than with the Ferrari algorithm appears to have accelerated the wave because the highest particle is in front of where the crest should theoretically be, in this case 9 cm ahead, while the Ferrari algorithm gives a crest 38 cm behind where the crest should be. But because the wave from the Vila algorithm has lost more height the wave is flatter, so the fact that the highest particle in this case is in front of the wave could be an anomaly due to the particle nature of SPH. Figure 5.23 shows that the wave is where it should be but has lost more height than with the Ferrari algorithm.

## 5.4 Conclusions

For the Koshizuka & Oka experiment the Vila SPH algorithm with HLLC ARS does appear to be too dissipative despite the satisfactory results for the front and height of the column with Monaghan & Kos and Monaghan Lennard-Jones boundary treatments during the initial collapse phase. These results do tend to confirm the statements of Ferrari *et al.* and Antuono *et al.* that Riemann solvers in SPH can provide more accuracy but can also be too dissipative in violent simulations. This may be resolved with the use of a different equation of state and a different approximate Riemann solver to the HLLC used here, or an exact Riemann solver.

The boundary conditions are also of great importance. The boundary treatment of Dalrymple & Knio appears to add viscosity when used with the Vila SPH algorithm. The idea of the Dalrymple & Knio boundary treatment is to consider the boundary particles as water particles that obey the SPH equations being used, so that the pressure exerted by the boundary particles is dynamic. But the use of the other two boundary treatments, the Monaghan & Kos and Monaghan Lennard-Jones treatments, shows that in violent conditions the algorithm has too much intrinsic viscosity. With the Dalrymple & Knio boundary treatment, in which the boundary particles are treated as water particles, this intrinsic viscosity is also present at the boundary, which may be contributing to the apparent excessive viscosity of the water during the initial collapse phase of the simulation.

The simulation of the solitary wave with the Vila algorithm and the Ferrari algorithm indicate that these algorithms are not that good at solitary wave generation, even with a Riemann solver, and may need considerably more particles.

# Chapter 6

# Boundary Treatment

The treatment of boundaries in SPH continues to be a problem, despite the number of approaches to model the boundary that have been suggested. Some boundary treatments were briefly discussed in Chapter 2, the introduction to SPH. This chapter will examine these boundary treatments in more detail, and then propose a new boundary treatment that when implemented on the GPU significantly accelerates computation.

## 6.1 Known Boundary Treatments

As briefly discussed in Chapter 2, there is a wealth of boundary treatments available in SPH, some of which will now be covered in some detail to help explain the new boundary treatment proposed in this chapter.

1. On-Boundary Particles
2. Ghost Particles
3. Mixed, or Hybrid, Boundary Particles

### 6.1.1 On-Boundary Particles

On-boundary particles exist on the boundary and reside in computer memory for the duration of the simulation.

The classic example is that proposed by Monaghan[33] which models the boundary with a set of particles on the surface of the boundary only which exert a repulsive force

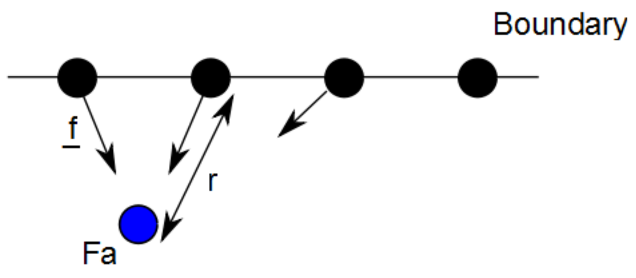


Figure 6.1: The On-Boundary Treatment of Monaghan

dependent on the distance between a fluid particle and a boundary particle. This is shown in Figure 6.1 in which on-boundary particles exert a repulsive force $\underline{f}$ on fluid particle *Fa* if the distance between particle centres *r* is less than a user-specified distance *R*, in which case the repulsive force $\underline{f}$ on fluid particle *Fa* takes the form

$$\underline{f} = D\Big[\Big(\frac{R}{r}\Big)^{P_1} - \Big(\frac{R}{r}\Big)^{P_2}\Big]\frac{\underline{r}}{r^2} \tag{6.1}$$

Another example of this type of boundary particle is that proposed by Dalrymple & Knio[38] and is shown in Figure 6.2. These boundary particles differ from those proposed by Monaghan in two ways.

- a second layer of particles reinforces the particles on the boundary surface and these are placed in a staggered grid structure
- these particles are treated as fluid particles, interacting with fluid particles in their influence domain, with a density and pressure which evolve as fluid particles, i.e. the SPH continuity equation is applied to them and integrated as if they were fluid particles, but the momentum equation is not, so the only velocity these boundary

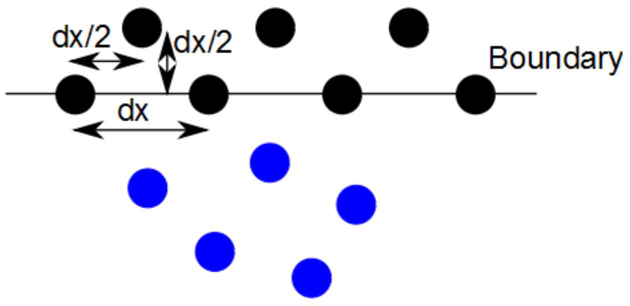


Figure 6.2: The On-Boundary Treatment of Dalrymple & Knio

particles have is that of the boundary they are representing, e.g. a piston

This idea was extended by Violeau & Issa[44] who use four layers of such particles which are aligned in a regular lattice structure.

### 6.1.2 Ghost Particles

Ghost particles do not either exist on the boundary or reside in computer memory for the full duration of a simulation, but are created whenever they are required. Their properties usually depend on the fluid particles and the method of generation being used. There are several proposals as to what properties the ghost particles have, and when they should be created.

### 6.1.3 Mixed or Hybrid Boundary Particles

A suggestion has been made by Liu & Liu to mix the two above models, with one set of on-boundary particles on the surface of the boundary only, with ghost particles to reinforce the on-boundary particles, and they report much less penetration of the boundary by fluid particles.

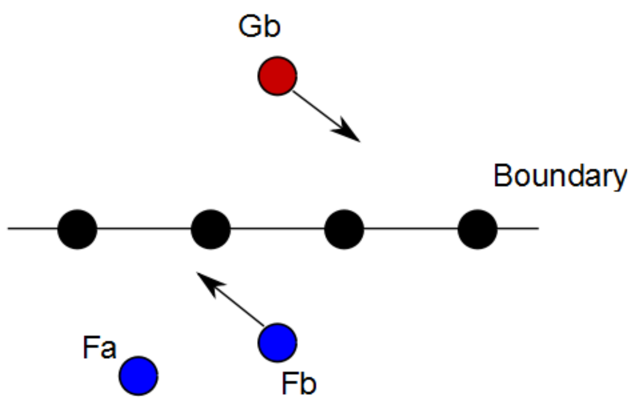


Figure 6.3: The Hybrid Boundary Treatment of Lo & Shao

Lo & Shao[43] implemented a hybrid boundary method. They placed fixed particles on the wall which were spaced according to the initial fluid particle spacing, and the Poisson equation is solved on these particles. But to prevent particle penetration they also create a ghost particle if there is an interaction between two fluid particles if both are close to the boundary. This hybrid treatment is shown in Figure 6.3, in which for two fluid particles *Fa* and *Fb*

1. the position of the ghost particle Gb is a direct reflection of the fluid particle Fb across the boundary
2. the velocity of Gb is the opposite of the fluid particle Fb, i.e. $\underline{v}_{Gb} = -\underline{v}_{Fb}$
3. the pressure of Gb equals the pressure Fb

## 6.2 The SPH Algorithm

The new boundary treatment is a proposal to improve the SPH algorithm as proposed by Ferrari *et al.*[42] when it is implemented on GPUs. The Ferrari group themselves proposed a new ghost particle method to model the boundary. This algorithm has also

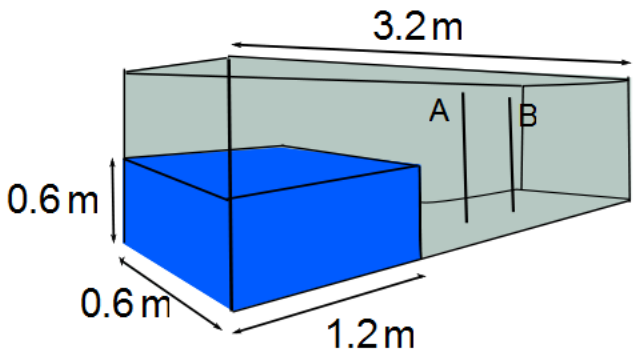


Figure 6.4: The initial set up of the water collapse

been implemented on a CPU cluster by Cherfils *et al.*[76] but with a boundary treatment based on the Immersed Boundary Method of Su *et al.*[77] which is significantly different to the boundary treatment proposed by the Ferrari group.

The Ferrari SPH algorithm used for the work in this chapter is the same as that of Ferrari *et al.*, with the governing equations given by Equations (4.1) - (4.10), which are integrated using the RK3 TVD integration scheme Equations (2.84) - (2.87) and the time step calculated by Equations (2.88) - (2.89). The particle is modeled by the Wendland Quintic smoothing function (4.11).

The problem simulated is that reported by Zhou *et al.*[78] and shown in Figure 6.4. A block of water of dimensions 1.2 m x 0.6 m x 0.6 m is allowed to collapse in a tank with dimensions 3.2 m x 1.2 m x 0.6 m.

The fluid particles were set up as follows. Particle spacings DXL, DYL and DZL along the Cartesian axes were all 0.012 m and the particles were initially aligned in a lattice structure so that particles were initially a distance

$$dr = \frac{1}{2}\sqrt{(DXL^2 + DYL^2 + DZL^2)} \tag{6.2}$$

with the particles closest to the boundary a distance 0.006 m from the boundary. This gave a total number of approximately 477k fluid particles. The initial smoothing length was 0.02 m.

Unfortunately the positioning of the boundary particles is not described in either Ferrari paper so an element of intuition has been applied to build a boundary strong enough to prevent penetration by the fluid particles. During the experimentation in creating a suitable boundary it was found that some boundaries required more boundary particles than others, in particular the floor required much more particles than most of the side walls. It was also found that a large number of boundary particles was required when compared to the number of fluid particles. This becomes a problem for the performance of the simulation on a GPU if the data for those many boundary particles is stored in the slow GPU global memory, and although the Ferrari algorithm does indeed use ghost particles those ghost particles are generated not just from the fluid particle but also from data for on-boundary particles which is usually stored in memory. To accelerate the computation/simulation the data for the on-boundary particles should preferably reside in the on-chip memory, in registers. There are at least two methods to implement this.

The first method is to generate the boundary particle positions as they would be if they were being stored in global memory from the known position of the fluid particle. For example, if the boundary particles were simply generated on the boundary with particle spacing *dx* from the origin and the fluid particle had position *(x,y)* then the boundary particle positions, at least, could be generated in the registers. However, if at least one property of the boundary particles evolves with time, such as pressure, then this will have to be stored in the global memory and recalled from there. The ideal situation is if the properties of the boundary particles depend only on the fluid particles at the time of computation. This method computes a boundary that would behave exactly as if the particles were recalled from global or texture memory but would be computed faster.

The second method is to generate a unique private grid of boundary particles for each fluid particle. In this case a fluid particle close to the boundary creates its own unique private grid of boundary particles so that the grid for one fluid particle is different to that of all the other particles.

This second method will now be implemented to measure its effect not only on the performance but also on the physical flow of the fluid which is being simulated.

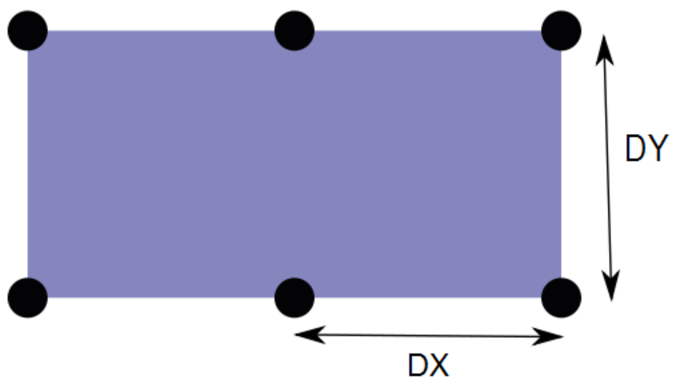


Figure 6.5: Boundary particles on a regular grid structure

## 6.3 A New Boundary Treatment

As stated above the description of the locations of the boundary particles is not specified in either Ferrari paper, but some success was found with the following arrangement of boundary particles.

The simulation is of a static column of water in a tank collapsing under gravity. The water is initially in the left of the tank and will collapse to flow towards and be reflected off the right hand wall. The near and far walls contain the water in the tank from spilling out of the side of the tank. For the left, near and far walls the boundary particles are placed on horizontal lines on a regular grid, as shown in Figure 6.5, a distance $DY = sf$ x $DYL$ apart in the y direction and $DX = sf$ x $DXL$ or $DZ = sf$ x $DZL$ apart in the $x$ or $z$ direction, where $sf$ is the space factor and is equal to $0.25$. In addition, on the near and far walls for x> 2.4 extra boundary particles were placed on the boundary so the boundary particles formed a lattice structure, as shown in Figure 6.6.

All the particles on the right wall were placed in a lattice structure with $DY = 0.5$ x $sf$ x $DYL$ and $DZ = 0.5$ x $sf$ x $DZL$

The particles for the floor were also all in a lattice but $sf = 0.125$, i.e. half that for the other boundaries. This is due to the pressure applied to the floor from the mass of the water.

There is evidence that this is possibly not the same arrangement as used by the Ferrari

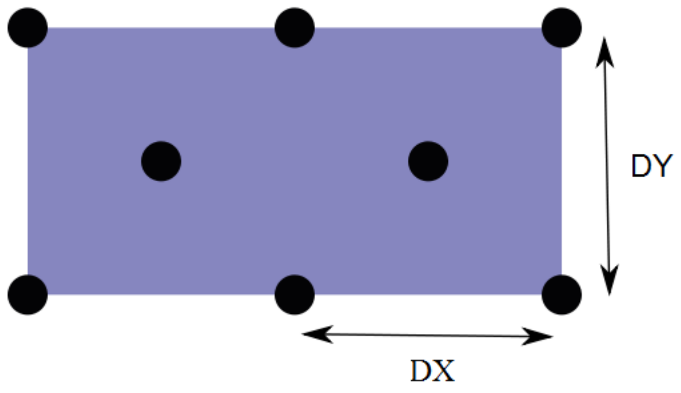


Figure 6.6: Boundary particles on a lattice structure

group because in the second paper that has been published based on this algorithm the CFL number used is quoted as 0.9, while the author with this particular arrangement of boundary particles found that a CFL number $> 0.2$ gave penetration of the boundary. Ferrari *et al.* do not comment on particle penetration of the boundary in either paper.

The proposed new boundary treatment is shown in Figure 6.7. Instead of using static boundary particles, a private and unique grid of boundary particles is created for each fluid particle *i* whose perpendicular distance from the boundary is less than the smoothing length of that particle. The centre of this grid depends on the fluid particle itself. The size of this private grid is $2h_i$ x $2h_i$, where $h_i$ is the variable smoothing length of fluid particle *i* close to the boundary, and the grid consists of 81 particles with a regular structure. As with the Ferrari boundary treatment the fluid particle does not interact with the boundary particles in this grid but does interact with ghost particles that are created by local point symmetry.

The result of running the amended Ferrari algorithm as described above on one NVIDIA S1070, i.e. 4 GPUs, at times 0 s, 0.6 s, 1.2 s, 1.5 s and 2.0s, with the new boundary treatment just discussed are shown in Figure 6.8, Figure 6.9, Figure 6.10, Figure 6.11 and Figure 6.12 respectively. Execution times are provided in Table 6.3 and show an approximate 5x speed up, despite the large number of boundary particles in the fluid particle's private grid with the new boundary treatment. Figure 6.12 shows the water at

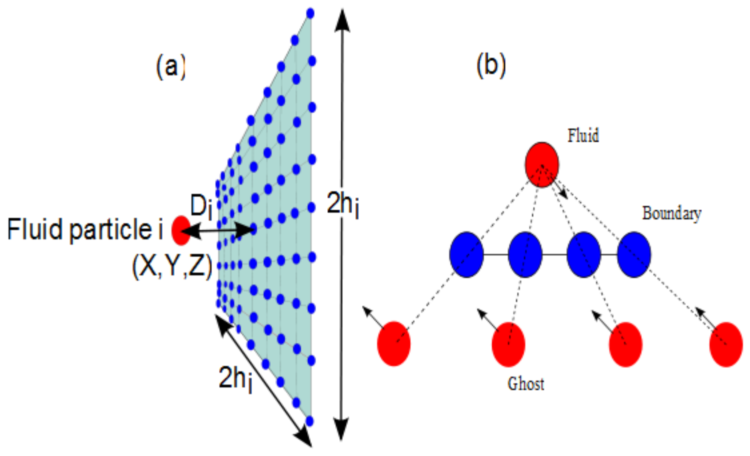


Figure 6.7: (a) shows the private boundary cell created for fluid particle i at position (X,Y,Z), and (b) shows how the ghost particles are created by local point symmetry for each fictitious boundary particle in the cell created in (a).

2.0 s with the proposed new boundary treatment. Note there is no particle penetration of the boundaries. Discussion of how this algorithm can be implemented on multiple GPUs is left for Chapter 7.

Zhou *et al.*[78] recorded the height of the water at four locations, but the Ferrari group compare the heights from their simulation at just two locations, which are shown as probe A at $x$=2.228 m and probe B at $x$=2.725 m in Figure 6.4. These are shown in Figure 6.13 and Figure 6.14. The heights of the water at these locations show quite good agreement, but there appears to be an over prediction of the height after the water has been reflected. This may be due to the particle nature of SPH and the method of deciding where the main body of water is. In grid based methods it is relatively straightforward to place a node of the grid at a specific location, for example where a probe in an experiment was located, and calculate properties, such as water height, at that node. In particle based methods this is not that straightforward, particularly when the water fragments and creates spray, as occurs with the Ferrari SPH method. The questions then become

1. does a specific particle belong to the main body of water or is it spray?

2. and if a particle is spray then should it be included in the estimation of water height?

To highlight this problem, Figure 6.15 shows a portion of the water around where the

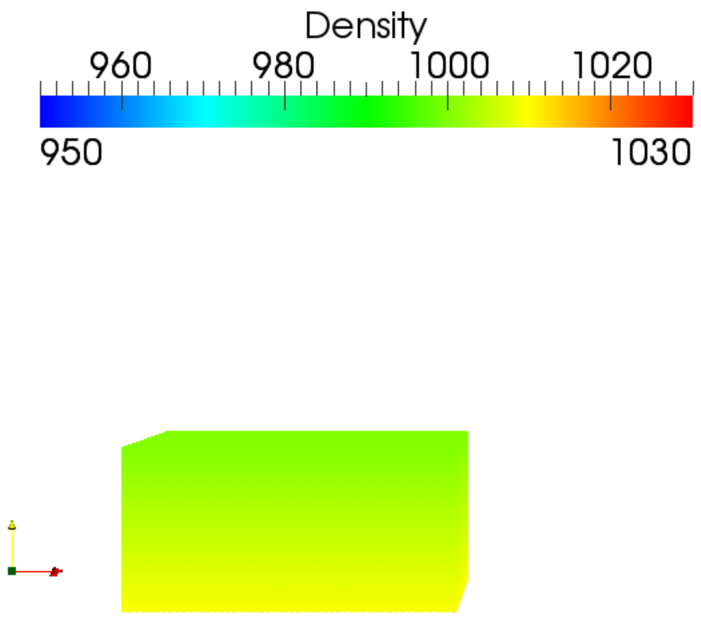


Figure 6.8: Snapshot of a 3D dambreak using the Ferrari SPH algorithm at 0 s with new boundary treatment

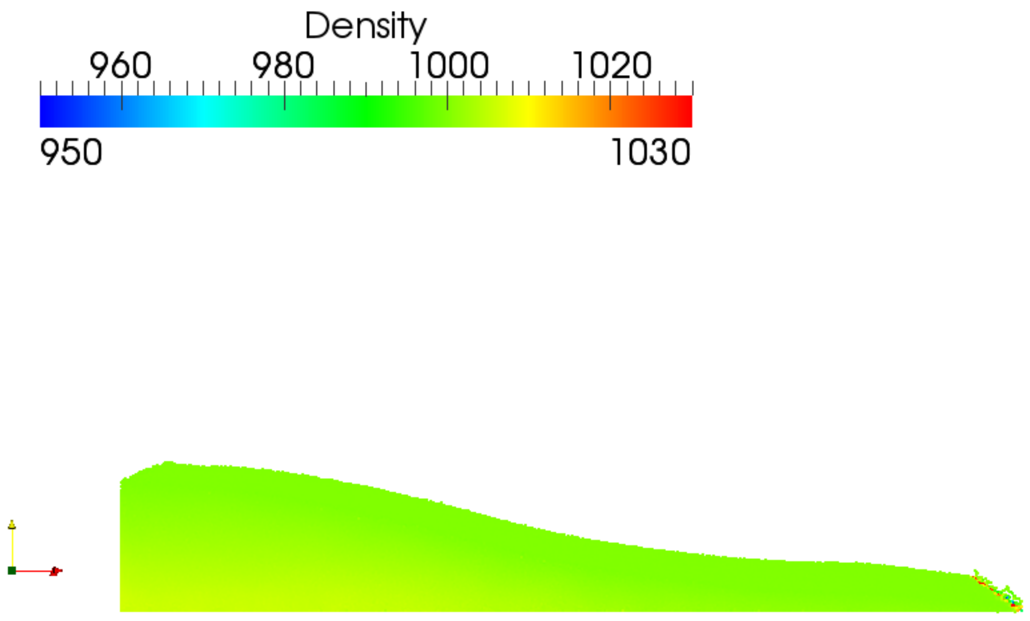


Figure 6.9: Snapshot of a 3D dambreak using the Ferrari SPH algorithm at 0.6 s with new boundary treatment

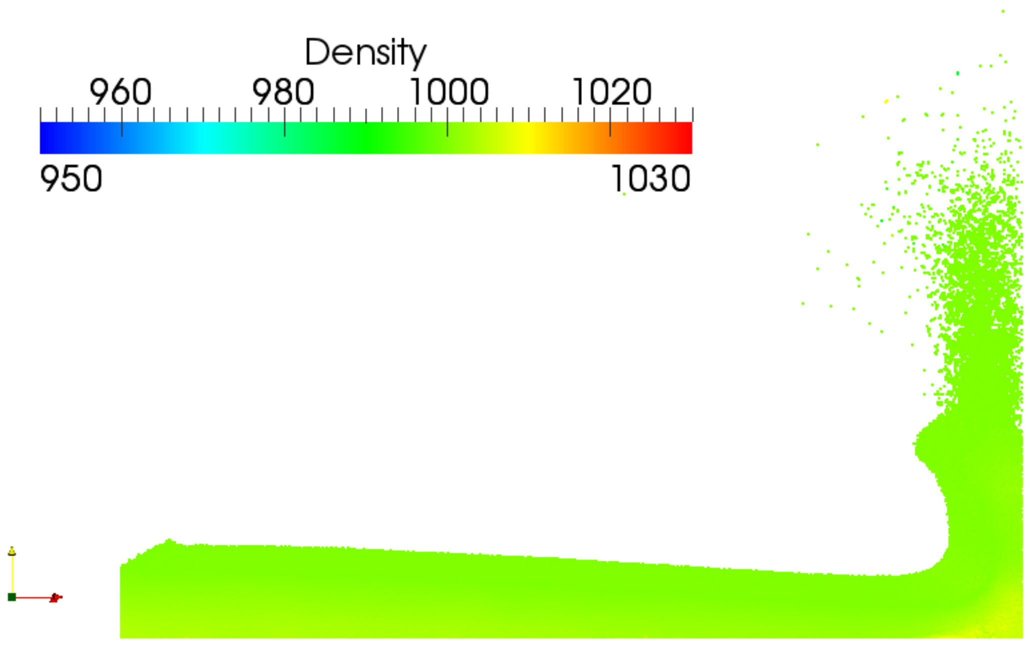


Figure 6.10: Snapshot of a 3D dambreak using the Ferrari SPH algorithm at 1.2 s with new boundary treatment

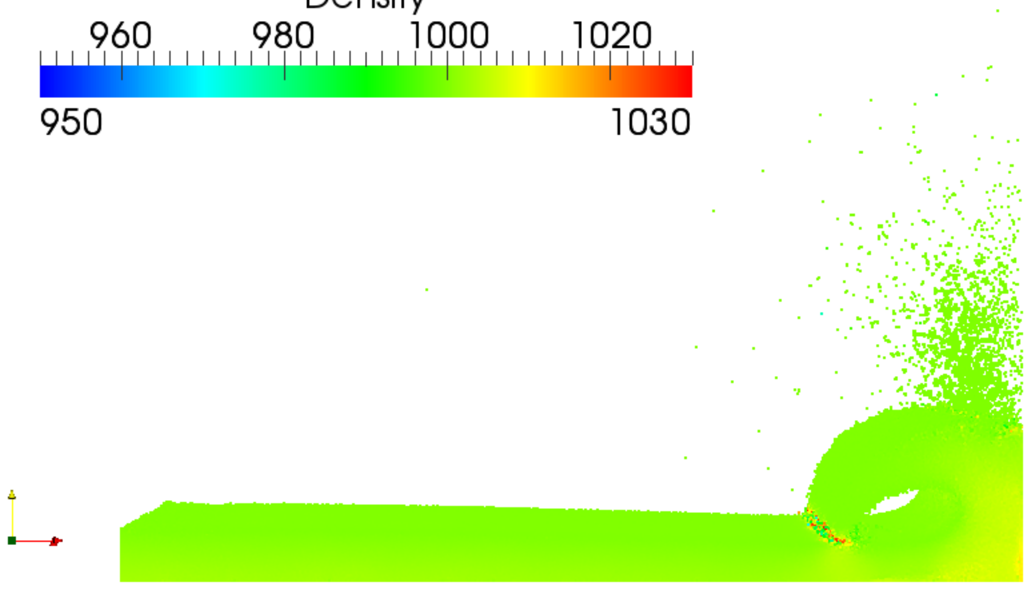


Figure 6.11: Snapshot of a 3D dambreak using the Ferrari SPH algorithm at 1.5 s with new boundary treatment

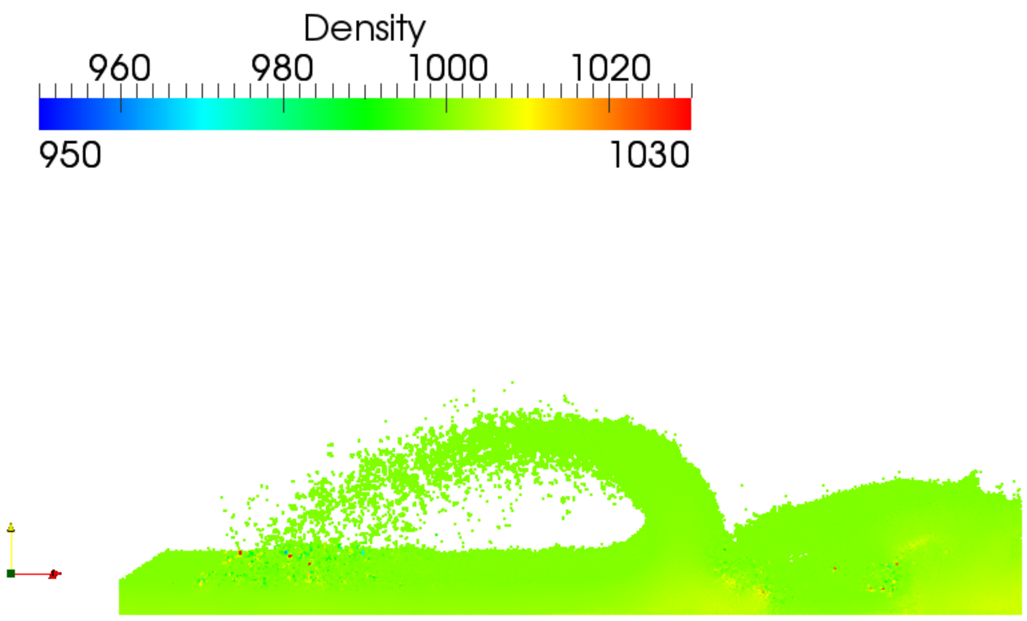


Figure 6.12: Snapshot of a 3D dambreak using the Ferrari SPH algorithm at 2.0 s with new boundary treatment

| **Real Time (s)** | **Original BC (mins)** | **New BC (mins)** |
|---|---|---|
| 0 | 0.00 | 0.00 |
| 0.1 | 42.83 | 8.01 |
| 0.2 | 87.53 | 16.48 |
| 0.3 | 136.09 | 26.19 |
| 0.4 | 189.57 | 36.99 |
| 0.5 | 248.26 | 48.79 |
| 0.6 | 315.04 | 61.57 |
| 0.7 | 398.08 | 76.43 |
| 0.8 | 476.33 | 90.55 |
| 0.9 | 557.02 | 104.86 |
| 1 | 640.10 | 119.54 |
| 1.1 | 725.98 | 134.58 |
| 1.2 | 813.51 | 150.05 |
| 1.3 | 903.29 | 165.95 |
| 1.4 | 994.08 | 182.19 |
| 1.5 | 1091.56 | 199.60 |
| 1.6 | - | 218.27 |
| 1.7 | - | 237.83 |
| 1.8 | - | 256.92 |
| 1.9 | - | 275.97 |
| 2 | - | 295.84 |

Table 6.1: The simulation times to reach the given real times

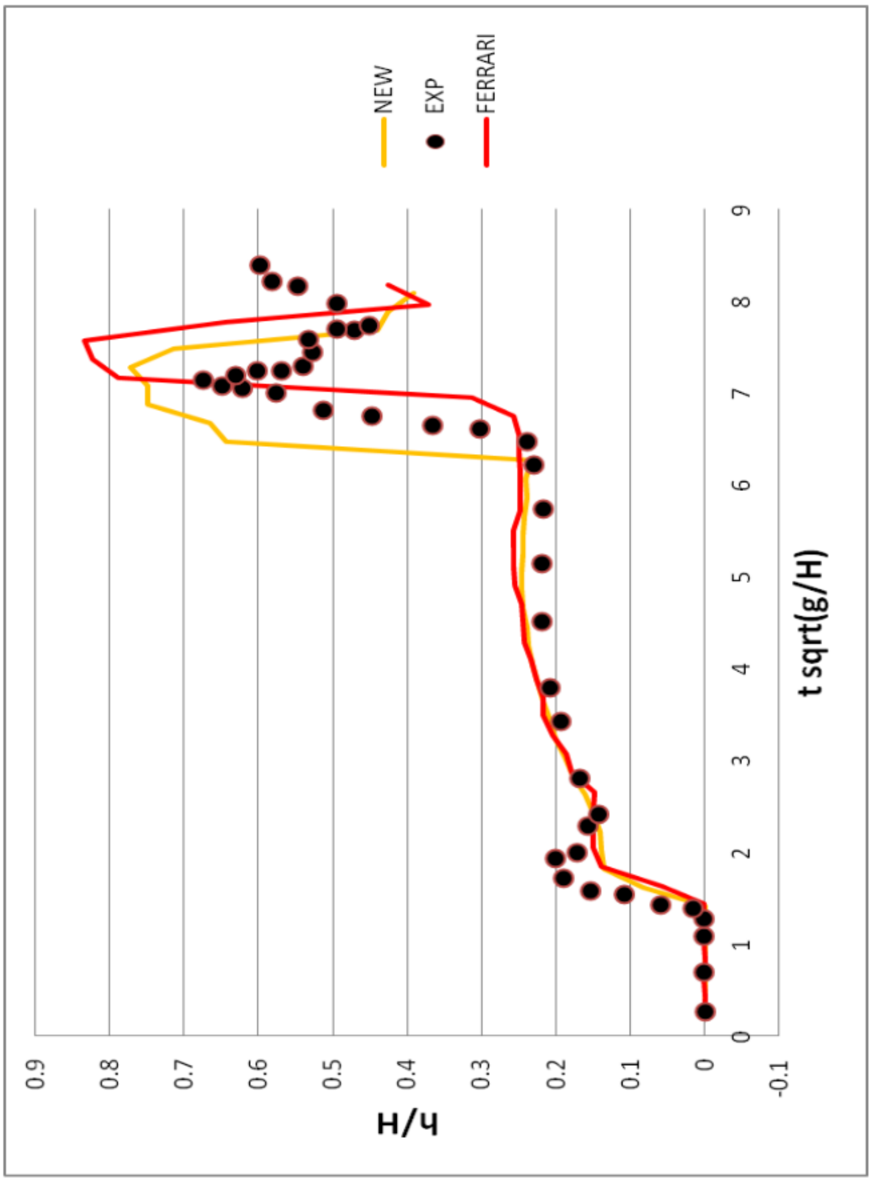


Figure 6.13: The height of water at probe A

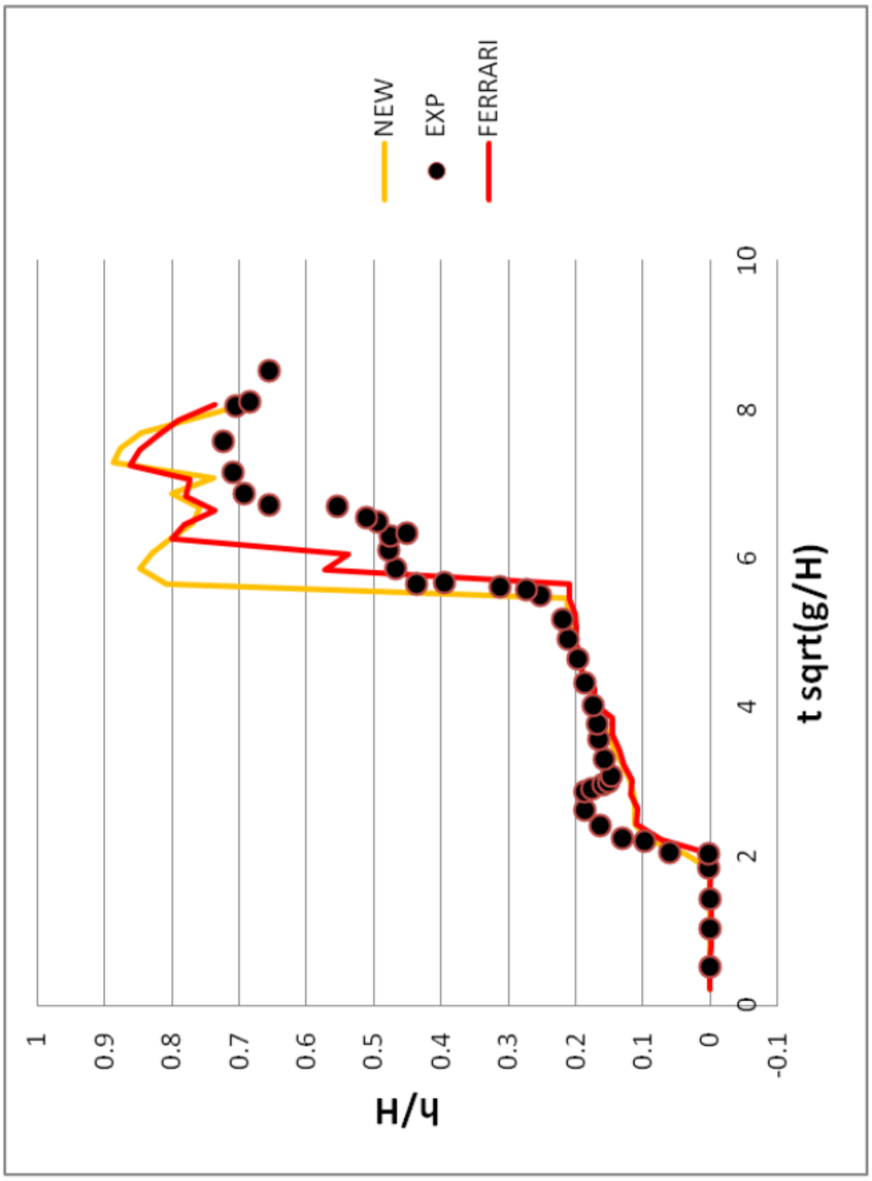


Figure 6.14: The height of water at probe B

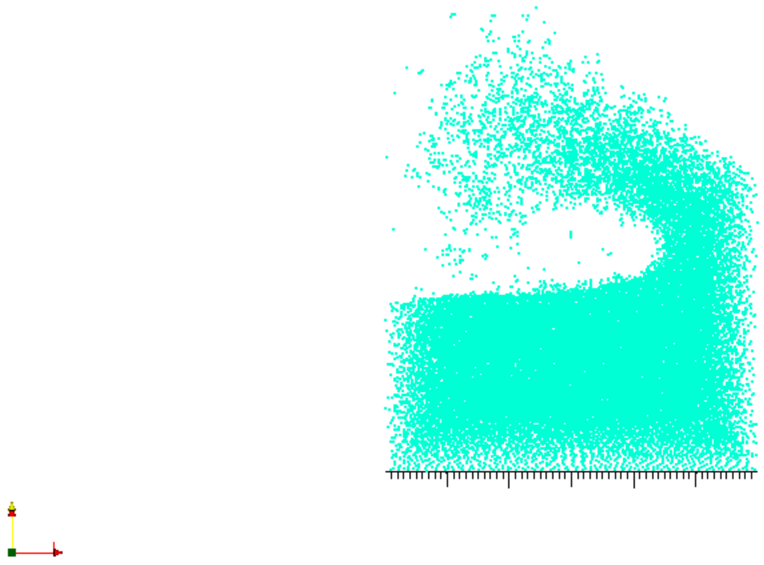


Figure 6.15: The height of water at probe B

probe A should be at $x$=2.228 m. So which particles make up the main body of water, and which are spray? The method used in this chapter to determine which was which was to assume that all particles belonged to the main body of water, except those which were obviously spray and were over 20 cm away from what appears to be the main body of water. Using this assumption probably leads to particles that are probably spray being considered as belonging to the main body of water, which leads to the over prediction of the height of the water at the probe after the water has been reflected. Before the water is reflected the height of the water predicted by the simulation using the new boundary treatment is in good agreement with experimental results so the new boundary treatment can be considered as valid.

The result of using this new boundary treatment is that

1. execution time is significantly decreased, with an approximate 5x speed up,
2. there is no penetration of the boundaries with the new boundary treatment,
3. results from simulations give good agreement with experimental results.

## 6.4 A Solitary Wave

A solitary wave with the same profile and dimensions as given in Chapter 5 was set up to examine how the new boundary treatment as proposed in this chapter would perform. The results of the simulation at 0 s, 1 s and 4 s of real time are given in Figure 6.16, Figure 6.17 and Figure 6.18 respectively. The solid blue line indicates where the wave should be. The new boundary treatment appears to preserve the solitary wave more than with the original Ferrari boundary treatment, but the wave has still lost horizontal speed.

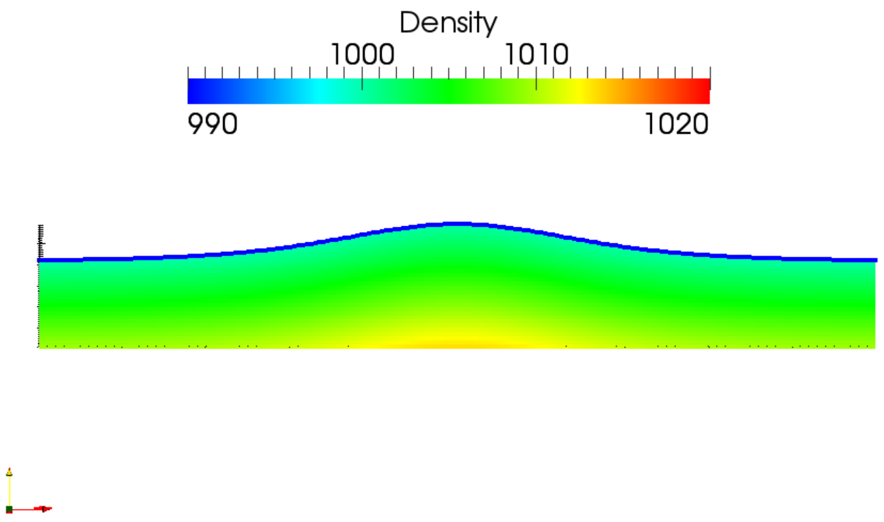


Figure 6.16: The solitary wave at 0 s with the Ferrari SPH algorithm and new boundary treatment

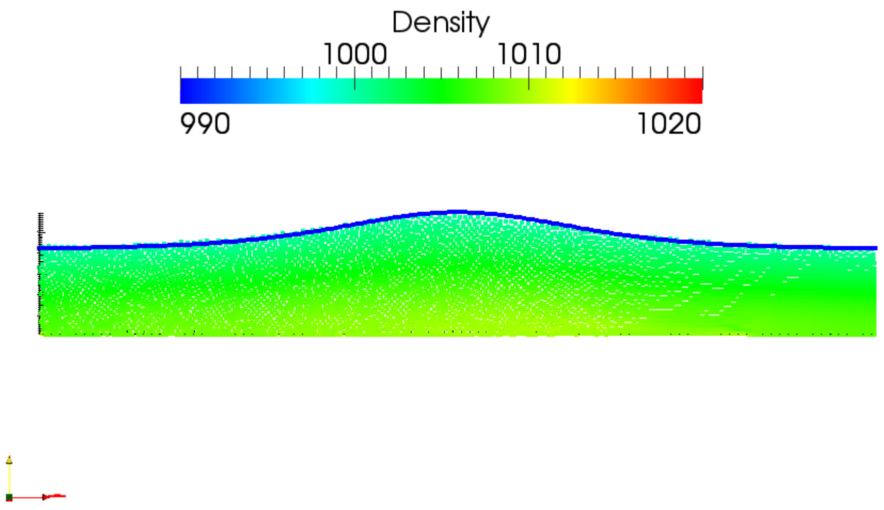


Figure 6.17: The solitary wave at 1 s with the Ferrari SPH algorithm and new boundary treatment

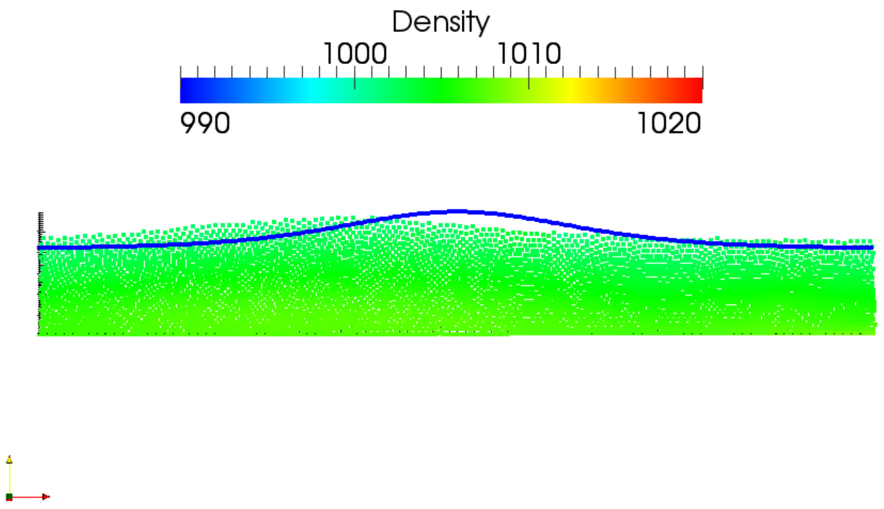


Figure 6.18: The solitary wave at 4 s with the Ferrari SPH algorithm and new boundary treatment

Table 6.2 reflects Table 5.1 given in Chapter 5 and contains the same data but with extra provided by the solitary wave simulation described in this section. The new boundary treatment improves the performance of the Ferrari algorithm in execution time, wave height and crest location. However the location of the crest is still disappointing when compared to that obtained from the Vila algorithm.

| **Algorithm** | **Fluid Particles** | **Time (mins)** | **Crest** | **Height Loss (m)** |
|---|---|---|---|---|
| Vila | 21648 | 8.00 | 0.09 | 0.038 |
| Ferrari | 21648 | 18.43 | -0.38 | 0.015 |
| New BC | 21648 | 8.25 | -0.25 | 0.011 |

Table 6.2: The execution times for the Vila and Ferrari algorithms to simulate 4 s of a solitary wave

## 6.5 Conclusion

This chapter has shown that the way the boundary is handled in the computer code itself may need to be amended for execution of a SPH simulation on a GPU so that boundary treatments that use significant numbers of static boundary particles, such as in the Ferrari algorithm above, or as used by Violeau & Issa[44], can achieve better performance and utilise the speed of the GPU. The speed up from the boundary treatment proposed in this chapter comes from two sources; the generation of boundary data in the registers, and a larger CFL number that does not give particle penetration of the boundaries.

The new boundary treatment proposed in this chapter helps to preserve a solitary wave simulated with the Ferrari SPH algorithm better than when the Ferrari boundary treatment is used, but the wave crest is still significantly behind its theoretical position.

# Chapter 7

# SPH on Multiple GPUs

The NVIDIA S1070 consists of four NVIDIA C1060 processors which can be programmed to run concurrently on the same problem with Message Passing Interface, also known as MPI. The physical structure of the S1070 makes it a distributed memory network, for which MPI is more suited. It is possible to use Open Multi-Processing, or OpenMP, but this is more suitable for multi-processor shared memory computers.

To this author's knowledge only one study of SPH on multiple GPUs for water flow has been made. In that study Valdez-Balderas *et al.*[79] partition the physical space into volume domains and assign a particular domain to a particular GPU which then manages all the particles in that domain, i.e. finds all interactions for those particles and integrates rates of change of density, velocity etc. At the end of each time step each GPU calculates which of the particles it is currently managing need to be transferred to another GPU due to the new positions of the particles, and which particles it is currently managing lie at its domain edge and could thus interact with particles in neighbouring domains. The GPUs managing the neighbouring domains are then informed of these particles. This algorithm was implemented on a cluster comprising the latest NVIDIA GPU S2050 which features the Fermi architecture, and inter GPU communication was facilitated with MPI.

This chapter will describe how the SPH algorithm as proposed by Ferrari and as modified in the previous chapter with a new boundary treatment can be implemented on a cluster of NVIDIA S1070s using a different strategy to that proposed by Valdez-Balderas *et al.*[79].

## 7.1 Message Passing Interface

The Message Passing Interface (MPI) permits the communication of data across a distributed network, which is essentially what a NVIDIA S1070 is, and thus a cluster of NVIDIA S1070s are, with each device having its own private device global memory of size 4 GB. A MPI program creates a set of *N* processes of rank 0,..,N-1. Each process usually contains the same variables of the same size, but is usually responsible for managing only a portion of those variables. For example, process *p* will be responsible for indices *pW* to *(p+1)W-1* of an array when each process is responsible for *W* items. NB each process has an instance of that array of size *nW*, where *n* is the number of processes, and not just one smaller array of size *W*. This is expensive in memory but communication between the processes allows the processes to act independently and process their portions of data until a synchronisation point is reached in the program at which the processes can easily refer portions of each others arrays, as will be shown with the MPI function *MPI_Allgather*. Figure 7.1 shows a MPI program running 4 processes on a single NVIDIA S1070. The system consists of a server with host memory and four devices each with their own private device global memory of size 4 GB, i.e. four blocks of distributed memory. Each process contains the same variables of the same size on both the host memory and the device memory. One of these processes is usually allocated as a master or root process to facilitate communication.

As a simple example assume that in a four process program on a NVIDIA S1070, with one process per device, each of the four processes is responsible for calculating the positions of a subset of particles, and then communicates these to all other processes so that each process holds the positions of all particles and not just those of the subset for which it is responsible. This is a communication that is required in the SPH implementation on multiple GPUs described in this chapter, in which at the end of one time step each device integrates the velocity to find the positions for a subset of particles, and communicates these to all other processes for the next time step to find interactions between the particles for which it is responsible and all particles.

For this we shall use the variables *d_X* of size *W* and *d_Xall* of size *4W* on the device, and *h_Xall* on the host, so each process is responsible for *W* particles. Each process

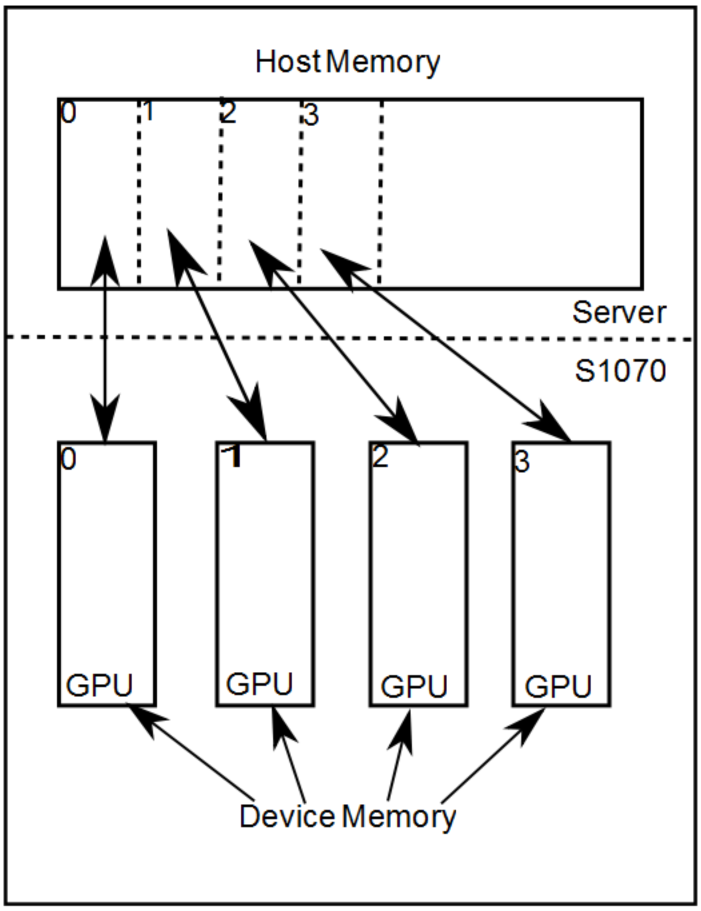


Figure 7.1: MPI Processes on a single node

calculates the contents of *d_X* on the device by integrating particle velocity, and then transfers those values to the host array *h_Xall* via the CUDA function *cudaMemcpy* with the direction device to host. On the host side each process holds a copy of the array *h_Xall* of size *4W*, so when transferring their data from the device to their process image on the host each process would write *W* items to the index *h_Xall[rank x W]* in their process on the host. Once each process has copied their data from the device to the host MPI can be used to communicate these values with the function *MPI_Gather*, which gathers all portions of *h_Xall* onto a root process, usually rank 0, followed by a call to the function *MPI_Bcast* from that root process to all processes, so that each process has a full copy of the array *h_Xall* containing the positions of all particles. Each process can then transfer the whole array *h_Xall* from host memory to *d_Xall* on its device by a call to the CUDA function *cudaMemcpy* with direction host to device. These MPI functions *MPI_Gather* and *MPI_Bcast*, with other MPI functionality used in this multiple GPU implementation, will be discussed later.

The model used for the implementation of the SPH algorithm on multiple GPUs consists of

- partitioning the fluid particles into the same number of blocks as there are devices, so that each device then manages the same subset of the fluid particles throughout the simulation

- each device also holds a copy of all particles, both fluid and boundary, if boundary particles are being used

- each device calculates the acceleration and rate of change of density for the set of fluid particles they are managing, by finding interactions with all particles, fluid and boundary, which have been sorted as described in Chapter 4

- each device integrates the acceleration and rate of change of density of the set of fluid particles they are managing to give position, velocity, density, pressure and smoothing length

- each device then communicates these integrated values to all other devices.

This last step requires MPI, and its use to communicate these values was just described. The MPI functionality used to implement this communication is *MPI_Gather*, *MPI_Allgather*, *MPI_Bcast* and *MPI_Group*, which will now be discussed. In the following description of the algorithm assume that each process is responsible for managing *WIDTH* particles.

### 7.1.1 MPI_Gather

This function assigns a process as a root to which all processes send their private block of data in an array to the root so that the root holds a copy of all the data from each process and in rank order. As stated before, each process contains an array on the host, for example *h_Xall*, but may only have copied its private data from its device into the block of size *WIDTH* beginning at index *rank x WIDTH*, including the root process. *MPI_Gather* reads the block of data in *h_Xall* of size *WIDTH* beginning at index *rankj x WIDTH*, where *rankj* is the rank of the process it is reading, and copies that block of data into the root's copy of *h_Xall* beginning at index *rankj x WIDTH*.

### 7.1.2 MPI_Bcast

This function also assigns a process as a root and broadcasts a whole array to each process. Thus a *MPI_Gather* followed by a *MPI_Bcast*, using the same root process, allows each process to eventually have a full copy of an array which was initially partitioned across all the processes. Taking the example above, before *MPI_Gather* was executed each process held a copy of its data in *h_Xall* only, of size *WIDTH* beginning at index *rank x WIDTH*. *MPI_Gather* copies all blocks of size *WIDTH* to the correct index of the root's copy, and *MPI_Bcast* then broadcasts the whole of *h_Xall* from that root process to each process so that each process has a copy of all the data in *h_Xall*, not just its own. The combined use of *MPI_Gather* and *MPI_Bcast* to communicate distributed data is shown in Figure 7.2. During the gather phase each process sends its block of data of size *WIDTH* beginning at index *rank x WIDTH* to the designated root process, which places the blocks of data of size *WIDTH* at index *rankj x WIDTH*, where *rankj* is the rank of the sending process. During the broadcast phase the root process sends the full array of size *N x WIDTH*,

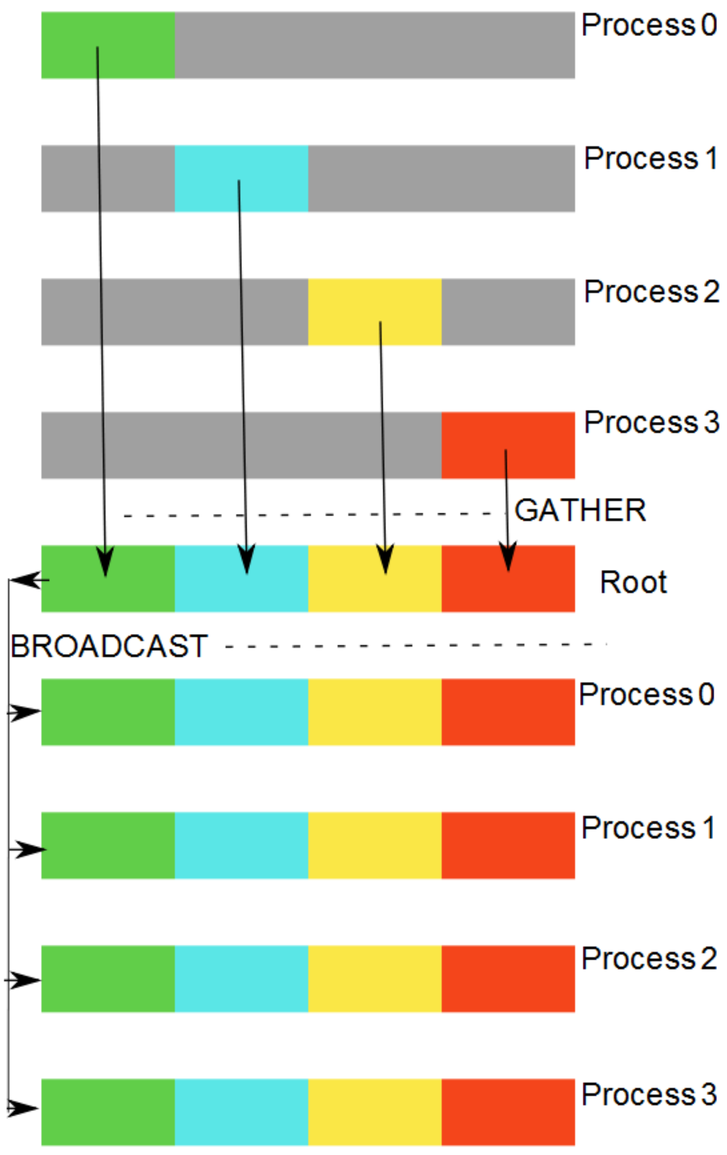


Figure 7.2: The gather and broadcast functionality for four processes

where *N* is the number of processes, to each process. Note that this occurs on the host or server side and not between devices or GPUs. To transfer data from the GPU to the host before a call to *MPI_Gather*, and to transfer data from the host to the GPU after a call to MPI_Bcast, a call to the CUDA function *cudaMemcpy* is required, setting the direction of transfer to device to host and host to device respectively. On the Fermi GPUs it is now possible to communicate directly between devices rather than transfer data via the host.

### 7.1.3 MPI_Allgather

This function is an extension of *MPI_Gather* and *MPI_Bcast* in which there is no root process but each process sends blocks of data of size *WIDTH* to and receives blocks of data of size *WIDTH* from all other processes in one call, and puts them into the block beginning at index *rankj x WIDTH* where *rankj* is the rank of the sending process. This has the same effect as using the *MPI_Gather* and *MPI_Bcast* method described above. This is shown in Figure 7.3. Initially each process contains data in a block of size *WIDTH* beginning at index *rank x WIDTH* where *rank* is the rank of the process. After the call to *MPI_Allgather* each process sends its block of data to all process and all processes receive that data and place it in a block of size *WIDTH* at index *rankj x WIDTH*, where *rankj* is the rank of the sending process. As with *MPI_Gather* and *MPI_Bcast* above, this communication occurs on the host or server, and calls to the CUDA function *cudaMemcpy* with the correct direction of transfer are required to transfer data from the device to the host before the call to *MPI_Allgather*, and back from the host to the device after the call to *MPI_Allgather*.

### 7.1.4 MPI_Group

As the name suggests MPI allows the user to define a subset of all processes as a group. In the implementation on multiple GPUs for this chapter, the groups were called *NodeGroup* and *PrimaryGroup*. Each group has a communicator which allows communication only between those processes assigned to the group.

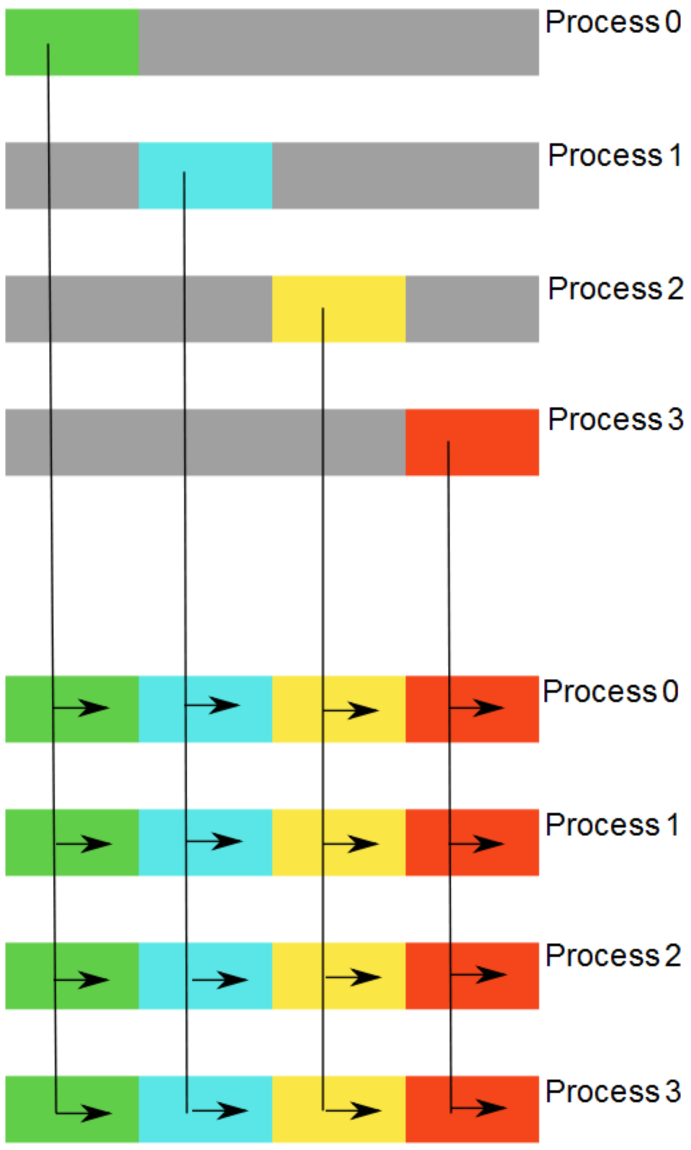


Figure 7.3: The allgather functionality for four processes

| Global rank | Internal rank | Node |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 2 | 1 | 0 |
| 3 | 1 | 1 |
| 4 | 2 | 0 |
| 5 | 2 | 1 |
| 6 | 3 | 0 |
| 7 | 3 | 1 |

Table 7.1: The MPI global ranks for a 8 process MPI program with Round Robin allocation

## 7.2 Rank Allocation

Devices can be allocated rank in MPI in a number of ways. The two most common are Round-Robin and Sequential. This is important because of the network topology of a cluster of NVIDIA S1070s in which the communication between nodes, i.e. S1070, is much slower than the internal communication between devices within a node.

The default method of rank allocation on the NVIDIA S1070 cluster at STFC Daresbury Laboratory, on which this method was implemented, is Round-Robin. In Round-Robin each node is considered in sequence, and the importance and difference between this and sequential allocation will soon become clear. There are four devices in a NVIDIA S1070 each with an internal rank of 0, 1, 2 or 3. A program requiring just four devices can be run on one S1070 and it does not matter which method of rank allocation is used. However, if more than four devices are required, for example eight, the Round-Robin method would allocate the devices with a MPI global rank as shown in Table 7.1. Assume we have two nodes each containing four devices. The MPI global rank allocation for a two node program is made by alternating between nodes 0 and 1. Similarly for a four node program, i.e. 16 devices, the MPI global ranks are allocated by assigning device 0 from node 0, node 1, node 2 and node 3, then device 1 from node 0, node 1, node 2 and node 3, etc, until all devices have been allocated a MPI global rank. In sequential allocation the devices within a node are all assigned a MPI global rank before any other device in any other node is assigned a MPI global rank. This is shown in Table 7.2. This becomes significant when the MPI_Group functionality is used.

| Global rank | Internal rank | Node |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 0 |
| 2 | 2 | 0 |
| 3 | 3 | 0 |
| 4 | 0 | 1 |
| 5 | 1 | 1 |
| 6 | 2 | 1 |
| 7 | 3 | 1 |

Table 7.2: The MPI global ranks for a 8 process MPI program with Sequential allocation

## 7.3 Implementation of a SPH Algorithm on Multiple GPUs

The SPH algorithm that has been implemented on multiple GPUs is based on that proposed by Ferrari *et al.*[42] and Ferrari[58] consisting of the governing equations given by Equations (4.1) - (4.10), which are integrated using the RK3 TVD integration scheme Equations (2.84) - (2.87) and the time step calculated by Equations (2.88) - (2.89). The particle is modeled by the Wendland Quintic smoothing function (4.11). The boundary treatment is that proposed in the previous chapter on boundary treatment.

To implement SPH on multiple devices the method of implementation of the radix sort as described in Chapter 4 needs a subtle amendment to account for the distributed nature of the implementation. In Chapter 4 all the data was held on just one GPU, all the particles were sorted in one sort and all particles were considered by the specification of the thread grid for the kernels. In this chapter, for multiple GPUs, each GPU, or device, is managing a subset of all the fluid particles. All the particles still need to be sorted as one set, but because only a subset of the particles is being managed by each device then after the sorting and reordering when the kernels for calculating acceleration and rate of change of density are executed the thread ID to extract the data for the *my* set of variables, e.g. *myMass*, needs to refer to real particle ID and not the sorted ID as in Chapter 4. Hence the real particle data is required, but just for the particles being managed by a device.

To guarantee that a process is responsible for the same subset of particles throughout a simulation we can use another set of arrays that are based on the real particle IDs. This involves slightly more communication than the previous suggestion.

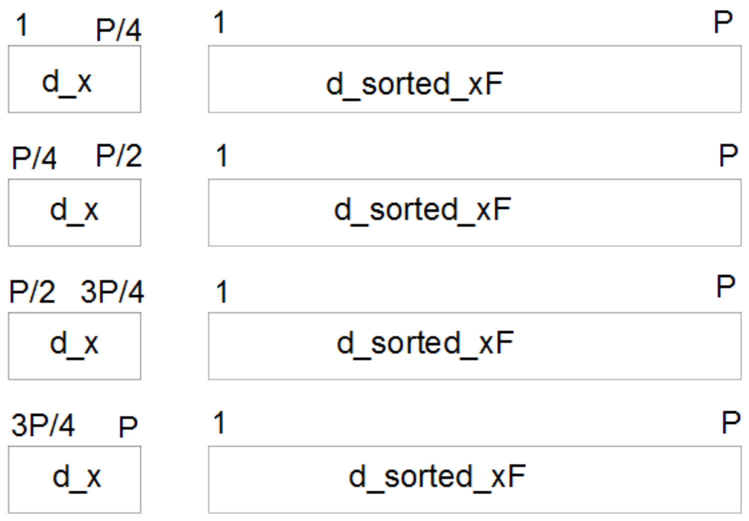


Figure 7.4: Each process has a copy of all particle data but manages only a fraction of the particles

Assume that the positions of all particles on all devices are held in an array called *d_xF*, and that this array is ordered by real particle ID. In addition assume that this data is also partitioned across the *N* devices being used in the simulation and that this partitioned data is held in the array *d_x*, i.e. if there are a total of *P* particles in the simulation then *d_xF* is of size *P* and the size of *d_x* is $P/N$. Thus each device is managing $P/N$ particles. As in chapter 4 we then sort *d_xF* to give the positions ordered by cell ID.

In chapter 4 the kernel to find the acceleration and rate of change of density would be given a thread grid definition large enough so that the acceleration and rate of change of density of all particles would be calculated. However, this is not the case with multiple GPUs in this method, in which a device is responsible for finding the acceleration and rate of change of density for only a subset of the particles. There is a need to limit the number of threads created for the thread grid for this kernel to just $P/N$ threads, one thread per particle. When all particles are on just one device, as in chapter 4, the thread ID was calculated as

```
ThreadID = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
```

and the properties of the particle that the thread represented were then assigned from the *sorted* arrays at the index *ThreadID*. But if we assign particle data to a thread from un-sorted arrays, which allows the same particles to be managed by the same device through-

```
float4 myX = d_x[GPUindex];
float4 myV = d_v[GPUindex];
float  myMass = d_mass[GPUindex];
float  myHsml = d_hsml[GPUindex];
float  myRho = d_rho[GPUindex];
float  myP = d_p[GPUindex];
```

Figure 7.5: The use of variable GPUindex to assign the properties of the particle being managed by the device

out the simulation, we need only use the *ThreadID* as calculated above, but this needs to be shifted depending on the rank of the device. This is shown in Figure 7.4 in which each of the four devices in this example has a copy of all data for *P* particles but also data from a subset of size $P/4$ of the particles which it is managing, with process 0 always managing particles 0 to $P/4$, process 1 always managing particles $P/4$ to $P/2$, etc. With this arrangement the kernel to calculate acceleration and rate of change of density need create $P/4$ threads only, with an ID called *GPUindex* just for use on the device only

```
GPUindex = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;
```

but also the real particle ID which is the *GPUindex* shifted by the following statement

```
realindex = GPUindex + rank*GPUBLOCKSIZE;
```

where *GPUBLOCKSIZE* is the number of particles being managed per device, which in this example is $P/4$.

The *GPUindex* is used to assign particle data to the thread from the *unsorted* data arrays of size $P/4$, as shown in Figure 7.5. Recall that in Chapter 4 the corresponding statements in Figure 7.5 referred to sorted arrays of size $P$, while here they refer to unsorted arrays of size $P/4$.

The $realindex$, or real particle ID, can then be used to check if the particles in the neighbouring cells are not the same particle that the thread is representing, with the statement as shown in Figure 7.6. where the array *particleHashF* is the sorted array of datatype *uint2* containing all particles, with cell ID as the *x* component and the real particle ID as the *y* component.

```
// get start of bucket for this cell
uint bucketStart = FETCH(cellStartF, gridHash);
if (bucketStart == 0xffffffff) return;   // cell empty

// iterate over particles in this cell
for(uint i=0; i<MAXPARTICLESPERCELL; i++)
{
  uint indexj = bucketStart + i;
  uint2 cellData = FETCH(particleHashF, indexj);
  if (cellData.x != gridHash) break;   // no longer in same cell

  // check not colliding with self
  if(cellData.y != realindex)
  {
    ...
  }
}
```

Figure 7.6: The use of variable realindex

### 7.3.1 The Amended Ferrari SPH on Multiple GPUs

The amended Ferrari SPH algorithm as discussed in Chapter 6 was used to simulate the exact same problem as described in that chapter, and executed on the cluster of NVIDIA S1070s at UK STFC Daresbury Laboratory with the MPI group, Sequential Allocation and CUDA stream functionalities discussed above. The codes were initially executed for 1000 iterations only. The results of the additions of these functionalities now follows.

To create a benchmark set of results the amended Ferrari SPH algorithm was executed for 1000 iterations only using 1, 2, 4, 8, 16 and 32 GPUs on the Daresbury cluster. This code had the following initial properties.

1. Used the new boundary treatment proposed in the previous chapter
2. Allocated global MPI rank by Round Robin
3. Did not use MPI group
4. Did not use CUDA streams

The initial fluid particle spacing was 0.012 m and they were organised as a regular lattice, giving approximately 478000 fluid particles. NB because the new boundary treatment proposed in the previous chapter was used there was no need to create boundary particles.

The results of these runs are given in Table 7.3 and are ordered by increasing number of GPUs. The total execution times in this table are found from the addition of the three

| GPUs | COMP(ms) | COMM(ms) | T(ms) | Total Time(ms) | Total Time (mins) |
|---|---|---|---|---|---|
| 1 | 1584332 | 70919 | 4952 | 1660203 | 27.67 |
| 2 | 807816 | 75736 | 2670 | 886222 | 14.77 |
| 4 | 390593 | 119050 | 2007 | 511650 | 8.52 |
| 8 | 189382 | 547107 | 1909 | 738398 | 12.31 |
| 16 | 100645 | 621239 | 5409 | 727293 | 12.12 |
| 32 | 59065 | 686493 | 4492 | 750050 | 12.5 |

Table 7.3: The benchmark times for 1000 iterations

components *COMP*, *COMM* and *T* which are the times for the SPH computations, communications (MPI and cudaMemcpy) and calculation of the next time step respectively.

NB just one node was used for the timings from 1,2, and 4 GPUs.

The first fact to note is that the total execution time does not necessarily decrease as the number of GPUs increases. There is a decrease with up to 4 GPUs, but when using more than 4 GPUs the execution time generally increases with the number of GPUs used. The component times indicate why this is so. The *COMP* times show an approximate halving of SPH computation time when the number of GPUs doubles, but it is the communication time *COMM* that dominates, with *COMM* approximately five times larger when more than one node is used. The MPI communication was done with the MPI function *MPI_Allgather* only. On just one node, with four devices, this communication mode is sufficient because all data transfer is via the relatively fast internal communication network of the node. But when using more than one node the *MPI_Allgather* is being executed by all processes so there is much more data transfer on the relatively slow communication network external to the nodes. For example for an 8 process program on two nodes one device in node 0 would need to communicate with all devices in node 1, both sending and receiving, and this would be multiplied by eight, for each device. So we can begin to see why when using more than one node the total execution time increases with increasing numbers of GPUs.

### 7.3.2 The Application of MPI Groups and Sequential Allocation

The MPI group facility allows us to group processes so that only processes in a group can communicate with each other. Recalling Table 7.1 when using Round Robin allocation for a program using two nodes the processes with even MPI global rank are all on one node, and the processes with odd MPI global rank are all on the other node. For an eight process program the groups would then be defined as $Group0 = 0, 2, 4, 6$ and $Group1 = 1, 3, 5, 7$. However, with Sequential Allocation, processes on node K can belong to group K and their ranks would be sequential. The groups would then be $Group0 = 0, 1, 2, 3$ and $Group1 = 4, 5, 6, 7$.

In the particular algorithm being implemented here all processes need to communicate their data with all other processes, but they do not have to do this communication directly with each other. To minimize the data transfer across the relatively slow external communication network, i.e. between nodes, it is possible for all processes to have a full copy of all required data via the following sequence of communications. A diagram of this communication sequence for a two node eight device scheme is shown in Figure 7.7. The processes on each node are grouped into a MPI_Group called *NodeGroup*, with $Group0 = 0, 1, 2, 3$ and $Group1 = 4, 5, 6, 7$, and each node has one process allocated as the primary process for that node and these primary processes are placed into a MPI_Group called *PrimaryGroup*. In Figure 7.7 the primary ranks are 0 and 4. The flow of communication in Figure 7.7 is as follows.

1. All processes on one node send their data to an allocated process, called the primary, on that node. This can be done with *MPI_Gather* onto the primary processes but using *NodeGroup* for communication and setting the primary for each group process as the root, and is represented by arrows from processes 1, 2 and 3 to process 0, and arrows from processes 5, 6 and 7 to process 4.

2. The primary processes on each node then communicate across the relatively slow external network using the *PrimaryGroup* for communication to exchange data so that only the primary processes have full copies of all the data. This can be done with textitMPI_Allgather and is represented by the double headed arrow between processes 0 and 4.

3. The primary processes then communicate all the data with the other processes on their node so that all processes have a full copy of all the data. This can be done with *MPI_Bcast* with *NodeGroup* for communication with the primary process as

| GPUs | COMP(ms) | COMM(ms) | T(ms) | Total Time(ms) | Total Time (mins) |
|---|---|---|---|---|---|
| 8 | 189485 | 207798 | 5591 | 402874 | 6.71 |
| 16 | 100853 | 236991 | 10578 | 348422 | 5.81 |
| 32 | 58114 | 248651 | 6526 | 313291 | 5.22 |

Table 7.4: The execution times for 1000 iterations with MPI groups and sequential allocation

the root and communicating to the node group, and is represented by the branched arrow from process 0 to processes 1, 2 and 3, and by the branched arrow from process 4 to processes 5, 6 and 7.

NB the devices first need to communicate their data to their hosts by the CUDA function *cudaMemcpy* before the call to *MPI_Gather*, and *cudaMemcpy* must also be called after the call to *MPI_Bcast* to transfer the data from the host to the device to complete the communication cycle so that each device has a copy of the full data. This is a lot of communication.

For the work quoted in here the primary processes were 0 and 4, but it should not matter which processes are designated as primary just as long as there is just one primary process on each node.

The results of applying both MPI Groups and Sequential Allocation when using more than one node are shown in Table 7.4. The results in Table 7.4 show that with the addition of MPI groups and sequential allocation

1. the total execution times decrease with increasing number of GPUs.
2. time required for SPH computation approximately halves when the number of GPUs doubles.
3. time required for communication is approximately constant and approximately one third of the communication time as the benchmark results.

### 7.3.3 The Application of CUDA Streams

CUDA streams are designed to facilitate concurrent host/device communication and kernel execution. Recall that a kernel is executed by blocks of threads, with the blocks specified in a grid by parameters supplied to the kernel in the kernel's parameter list. If

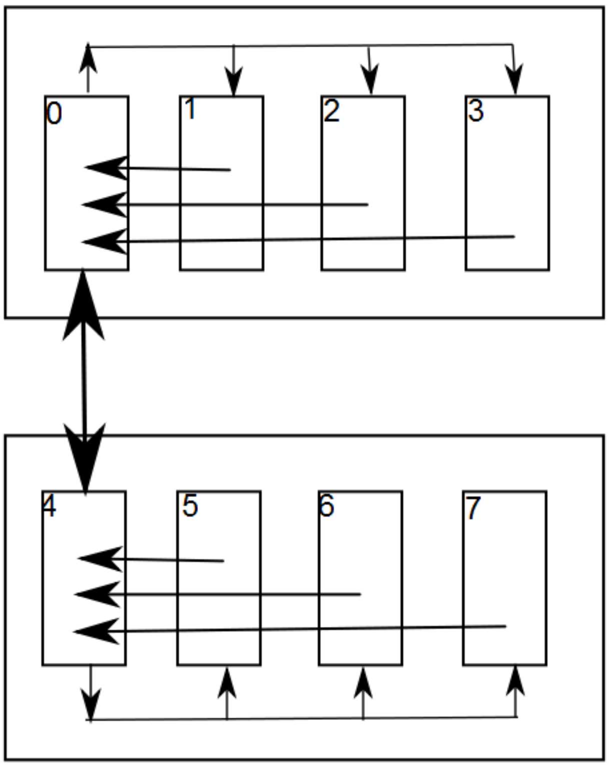


Figure 7.7: The communication flow for two NVIDIA S1070s with sequential allocation

| Streams | Total Time (mins) | % change |
|---|---|---|
| 16 | 7.36 | 9.68 |
| 4 | 6.58 | -2.01 |
| 8 | 6.50 | -3.20 |
| 2 | 6.28 | -6.52 |

Table 7.5: The execution times for 1000 iterations with MPI groups and sequential allocation for 8 GPUs and variable number of streams

blocks of threads can execute independently then it could be possible that the first few blocks to execute could complete their computation of the kernel and then have to wait for the remaining blocks of threads to complete their computation before any data transfer via *cudaMemcpy* to the host can occur. CUDA streams allow blocks that have completed their computation of a kernel to transfer data to the host while allowing other blocks of threads to complete their computation of the kernel. In previous examples memory on the device was allocated on the host with the standard C function *malloc*. When using CUDA streams the memory on the *host* should be pinned and allocated with the CUDA function *cudaMallocHost*.

The kernel taking the most time to execute in this amended Ferrari algorithm is that which calculates the acceleration and rate of change of density of the fluid particles. In the source code this is immediately followed by a kernel to integrate, which is simple in comparison, which in turn is followed by a group of statements to transfer several variables from the device to the host via the CUDA function *cudaMemcpy*. CUDA streams could accelerate this sequence of statements because the first few blocks of threads could execute the kernel to calculate acceleration and rate of change of density, then integrate those rates of change and then transfer the integrated variables to the host while the remaining blocks of threads execute the kernel to calculate the acceleration and rate of change of density.

CUDA streams were implemented for 8 GPUS for 2, 4, 8 and 16 streams and the results, including the percentage increase or decrease in execution time relative to that for 8 GPUs without streams from Table 7.4, are shown in Table 7.5. Similar experiments were performed for 16 and 32 devices and the results are shown in Table 7.6 and Table 7.7 respectively.

| Streams | Total Time (mins) | % change |
|---|---|---|
| 16 | 7.41 | 27.63 |
| 8 | 5.59 | -3.71 |
| 2 | 5.46 | -5.96 |
| 4 | 5.46 | -5.97 |

Table 7.6: The execution times for 1000 iterations with MPI groups and sequential allocation for 16 GPUs and variable number of streams

| Streams | Total Time (mins) | % change |
|---|---|---|
| 16 | 7.43 | 42.23 |
| 8 | 5.85 | 12.11 |
| 4 | 5.24 | 0.45 |
| 2 | 4.90 | -6.10 |

Table 7.7: The execution times for 1000 iterations with MPI groups and sequential allocation for 32 GPUs and variable number of streams

| Block size | Total Time (mins) | % change |
|---|---|---|
| 32 | 5.11 | -2.06 |
| 128 | 5.02 | -3.93 |
| 64 | 4.90 | -6.10 |

Table 7.8: The execution times for 1000 iterations with MPI groups and sequential allocation for 32 GPUs, 2 streams and a variable thread block size

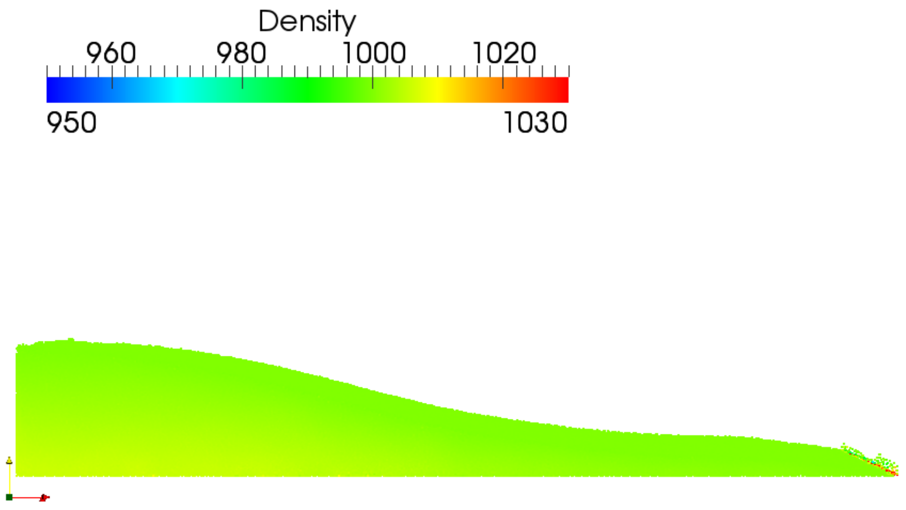


Figure 7.8: 3D view of simulation at 0.6 s

## 7.4 Full Simulation of 3D Dambreak

From the above timings from 1000 iterations Table 7.8 indicates that the fastest configuration of number of GPUs, the use of MPI groups as proposed above, sequential allocation, thread block size and number of CUDA streams is to use 32 GPUs with 2 streams and a thread block size of 64. This optimal configuration was run for a real time of 2 s on the cluster of 8 NVIDIA S1070s at STFC Daresbury Laboratory. The execution time for this simulation was approximately 1 hour 49 minutes. Images from the simulation at 0.6 s, 1.2 s, 1.5 s and 2 s, the particular times selected by Ferrari *et al.*[42], are shown in Figure 7.8, Figure 7.9, Figure 7.10 and Figure 7.11 respectively. The images from the Ferrari simulation at the same times are shown in Figure 7.12

## 7.5 Conclusion

This chapter examined the implementation of the amended Ferrari SPH algorithm on multiple GPUs to estimate the performance of the algorithm on a GPU cluster. The functionality of MPI used in the implementation was also discussed, as was the streaming functionality of CUDA. Experiments were first run over 1000 iterations to assess the optimal combination of number of GPUs, thread block size and CUDA streams, and it was found that with the current MPI algorithm described in this chapter in which particles

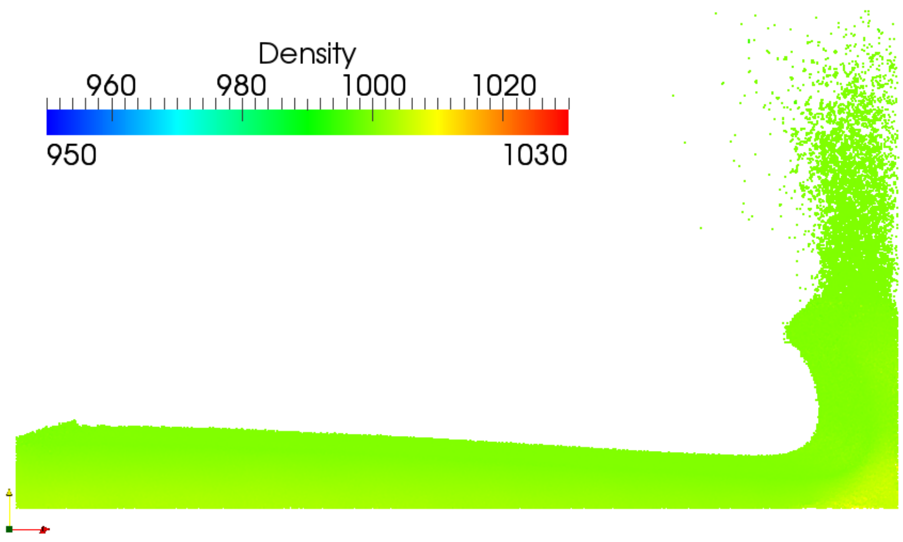


Figure 7.9: 3D view of simulation at 1.2 s

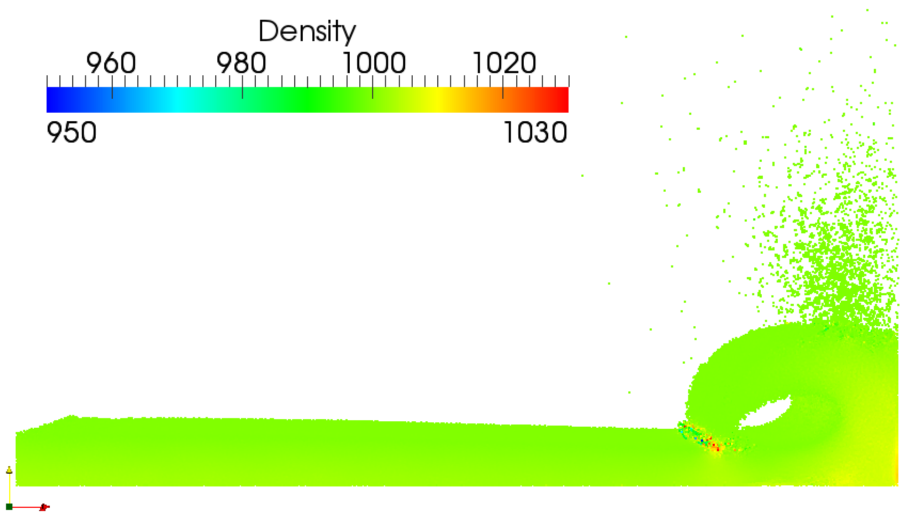


Figure 7.10: 3D view of simulation at 1.5 s

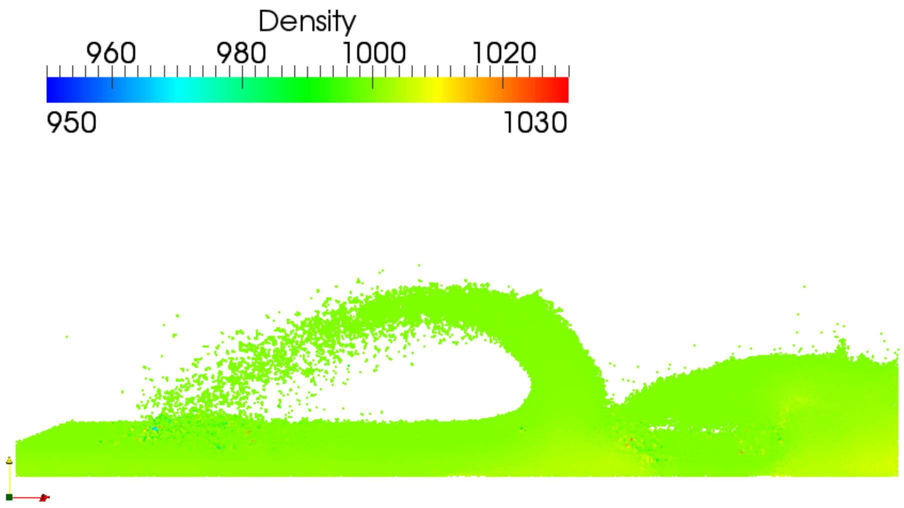


Figure 7.11: 3D view of simulation at 2.0s

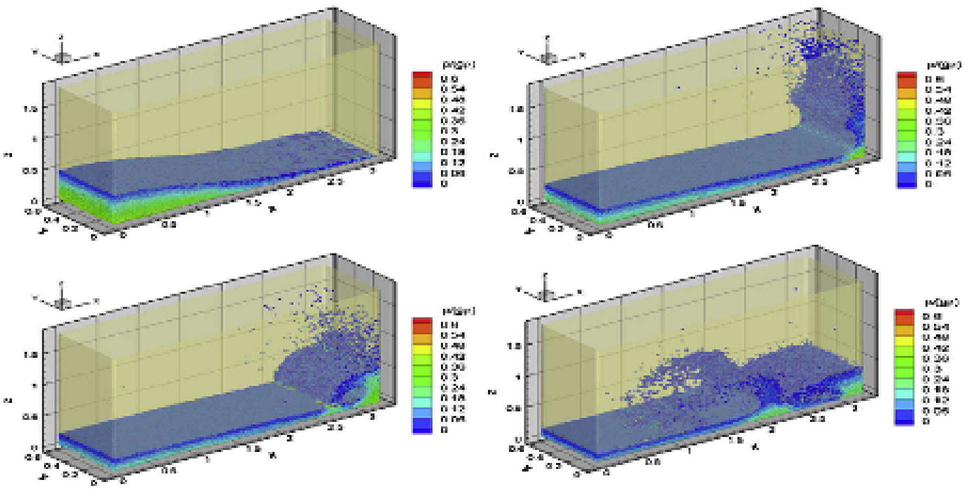
remain on a specified device throughout the simulation,

1. the computation time for the SPH algorithm approximately halved as the number of GPUs doubled, which is to be expected

2. MPI Groups reduce the total communication time and this time is approximately constant

3. CUDA streams can improve performance but they can also degrade it

With the optimal combination found to be 32 GPUs with 2 CUDA streams and a thread block size of 64 a simulation of 8 seconds real time was run which took approximately 9 hours 1 minute, and taking approximately 1 hour and 49 minutes for 2 seconds real time. The Ferrari group quote a simulation time of approximately 5 hours for a real time of 8 seconds on a 128 quad core CPU cluster. However, no CFL number was provided in the first Ferrari paper[42] and it is assumed that this was 0.9 because this is the CFL number quoted in the second Ferrari paper[58] which uses the same algorithm. In the simulations quoted in this chapter the CFL number was 0.3. The method of boundary treatment was addressed in the first Ferrari paper[42] but the actual structure or arrangement of the boundary particles was not, so as described in Chapter 6 a degree of trial and error was required to find an arrangement of boundary particles that worked. However a CFL number of greater than 0.2 led to particle penetration of the boundary, which was also not dis-



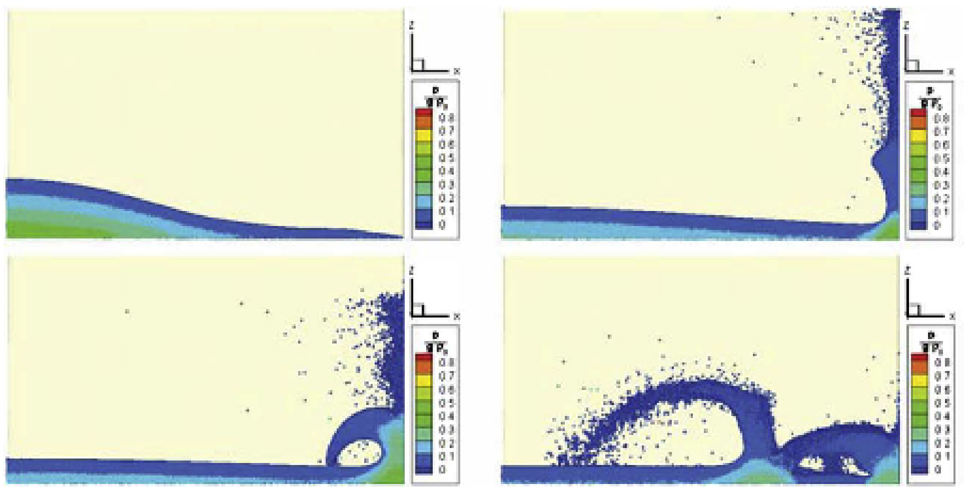


Figure 7.12: Images from the Ferrari simulation at 0.6 s, 1.2 s, 1.5 s and 2 s

cussed in either Ferrari paper. To increase the efficiency of handling the boundary on the GPU a new boundary treatment was proposed in Chapter 6 which permitted a CFL number of 0.3 without particle penetration of the boundary. It is possible to employ that same methodology, of calculating the position of the boundary particles in the registers, but this would still lead to the same CFL number of 0.2. If the Ferrari group could describe the arrangement of the boundary particles in greater detail which allows a CFL number of 0.9 then the positions of the boundary particles could also be calculated in the registers instead of being read from the much slower global and texture memories. This would then permit a CFL number of 0.9 leading to an approximate 3x speed up of the simulation on multiple GPUs discussed here, which would give a simulation time of approximately 3 hours for 8 seconds of real time, which would be significantly faster than the approximate 5 hours quoted by the Ferrari group. It should also be noted that the execution times for 1000 iterations on 16 GPUs are only a few seconds slower than those on 32 GPUs so it is very possible that this same simulation could be run on just 4 NVIDIA S1070s rather than 8 NVIDIA S1070s and it would still be faster than the 128 quad core CPU cluster.

Multiple GPUs have been used for SPH by Valdez-Balderas *et al.*[79] who partitioned the particles by volume or space. In their algorithm at the end of each time step there is then the requirement for

1. each particle to calculate on which device it should reside for the next time step
2. to transfer particle data between devices, if required
3. then calculate which particles are not in the volume controlled by a device but are close enough to interact with the particles on another device and then also transfer that data between devices
4. calculate if there is enough memory to hold all the data for the number of particles required on a particular device controlling a volume in space, and take appropriate measures if the data is excessive, i.e. load balancing

For problems in which particle speed is slow and/or problems with a high degree of symmetry, such as the 3D dambreak problem simulated for this chapter, this method of partition is suitable because there is little data to communicate between devices and the number of particles on each device remains approximately constant. However, in problems in which particle speeds are relatively high and/or problems with a low degree of symmetry, for example if the 3D dambreak collapsed into a large irregular object (most simulations collapse into a symmetrical block) causing reflected waves in all directions, implying much more communication of particle data between devices, this method of partitioning may degrade in performance significantly, particularly if a degree of load balancing is required when it is found that a significant fraction of the particles have been assigned to one device because of the positions of the particles. In this case even more communication is required from that overloaded device to the other devices so that the space containing the particles is more equally partitioned across the devices and algorithm execution can proceed with a more even distribution of particles across the available devices. This problem will never arise in the method of particle partition proposed in this chapter because particles always remain on the same device throughout the simulation. The problem then becomes one of simply minimizing data communication time between devices. A simple use of the MPI Group facility, as addressed in this chapter, showed that there is potential to accelerate this communication time by assigning only a small subset of the total number of devices to handle communication over the whole cluster.

The most efficient use of multiple GPUs for a SPH simulation may require both methods of partition as proposed in this chapter and by Valdez-Balderas *et al.*[79]. It may be that for relatively steadier flows the volume partition method of Valdez-Balderas *et al.*[79] may be more efficient, but when a simulation becomes violent then the particle partition method as proposed in this chapter may be more efficient. The problem would then depend on how violent a simulation has become, and how to evaluate that violence so that the program can switch from one partition method to the other and back again.

There is also potential to accelerate the data communication in the method proposed in this chapter. In the simulations giving the results quoted in this chapter the CUDA data type *float4* was used for the particle position and velocity, but only the $x$, $y$, and $z$ component of that data type were referenced in the source code for the SPH calculation but the whole data type was communicated, i.e. the $w$ component was redundant but communicated. This is a waste of communication. So it may be slightly faster to communicate and use in the SPH calculations two of density, pressure and smoothing length as the w component of the position and velocity.

Another amendment to the algorithm that would probably lead to an acceleration would be to process particle data as it is passed up the communication chain. Currently all the devices wait for a full and complete copy of the particle data as each process passes its particular portion of particle data up and down the communication chain. This waiting time could be used to process particle interactions from particles on the node, i.e. in the node group, and in the next MPI group level, and so on.

# Chapter 8

# Conclusions and Suggestions for Further Work

## 8.1 Conclusions

This thesis has shown

1. the basics of Smoothed Particle Hydrodynamics including how the particle approximations to the Euler equations are derived, and the components of a SPH algorithm including boundary treatments and corrections,
2. the architecture of the NVIDIA T10 Tesla Graphics Processing Unit and how it can be programmed for general purpose GPU computing,
3. how SPH can be implemented using two distinct methods on the NVIDIA T10 Tesla GPU; one method using coalescing and the shared memory as advised by NVIDIA, and the second method using a sorting procedure and the texture memory, with the latter currently the most efficient for large scale simulations,
4. how Riemann solvers have so far been implemented in SPH, the two opposing views of using Riemann solvers in SPH, the accuracy of one approximate Riemann solver in a special SPH formulation designed for using Riemann solvers,
5. some of the many different boundary treatments in SPH with a proposal for a new boundary treatment which implemented on the GPU can accelerate computation

when used in conjunction with one particular SPH algorithm,

6. how SPH can be implemented on multiple GPUs relatively easily giving the potential to run simulations much faster than on large expensive CPU clusters.

The novel elements of this thesis are

1. **the proposal of a new boundary treatment which implemented on the GPU can accelerate computation when used in conjunction with one particular SPH algorithm**, though the method can be used with any SPH algorithm,

2. **the proposal of a unique implementation of SPH on multiple GPUs giving the potential to run simulations much faster than on large expensive CPU clusters**, and this method can be used to implement any SPH algorithm on multiple GPUs,

The use of Riemann solvers in SPH was examined in this thesis, and despite the promising results from the use of Riemann solvers in SPH for gases without boundaries, the results of simulating a solitary wave in water with boundaries with the Vila SPH algorithm, which was proposed specifically for the use of a Riemann solver, with the HLLC approximate Riemann solver are disappointing. However, the results presented in this thesis were produced from just one SPH algorithm with the use of just one approximate Riemann solver. The simulations of the experiment of Koshizuka & Oka[11] from the Vila and Ferrari SPH algorithms indicate that the Vila algorithm produces little fragmentation and spray, while the Ferrari algorithm could be considered to produce too much spray. The photographs taken by Koshizuka & Oka of their experiment do show some spray, as seen in Figure 4.37.

The Riemann solver used for this thesis was difficult to code and debug, but gave superior results to those produced by the Ferrari SPH algorithm, even with the new boundary treatment proposed in this thesis, in the simulation of a solitary wave. On the other hand, the Ferrari SPH algorithm is relatively simple to code and uses a 3rd order integration scheme. The integration scheme used with the Vila SPH algorithm was a 2nd order Predictor-Corrector, so perhaps an integration scheme of higher order would produce a more accurate solitary wave. The conclusion on the use of Riemann solvers in SPH considering water with boundaries is that more research needs to be done into the use of exact and different approximate Riemann solvers other than the HLLC approximate Riemann solver used in this thesis, and with different equations of state other than the Tait equation of state used here. This research will be computationally expensive, but the GPU as shown in this thesis can provide the computing power for this research for relatively much lower cost than a large cluster of CPUs.

## 8.2 Further Work

The work described in this thesis can be extended with

### 8.2.1 Use of Riemann Solvers in Different SPH Equations in Different Situations

Ivings *et al.*[80] propose exact and approximate Riemann solvers specifically for compressible liquids, and showed that the solution to the problem being simulated can be significantly dependent on the Riemann solver being used. The Riemann solvers proposed by Ivings *et al.* may be able to produce much more accurate solutions to hydrodynamic simulations than has been achieved through using the HLLC ARS with the Vila SPH algorithm.

### 8.2.2 Use of GPU Shared Memory

A hybrid algorithm of first sorting particle data and then coalescing only a small number of blocks of particle data, as opposed to coalescing **all** blocks of particle data as described in Chapter 4, into the shared memory to process particle interactions can be implemented to evaluate its ease of implementation and performance as compared to the use of sorting the particles by cell and processing the particle interactions on a particle basis. Figure 4.1 from Chapter 4 shows how **all** particles are coalesced into shared memory to find interactions between **all** particles. Figure 8.1 shows how if the particles are first sorted by cell ID then only a small subset of particles need be coalesced into shared memory, with the blacked out blocks not coalesced into shared memory because the particles in those blocks would not be in neighbouring cells. The success or otherwise of this hybrid method will depend on the processor being used; a processor with fast texture/cache access is more likely to be more efficient with the sorting/texture approach. This method could also be used in 3D.

### 8.2.3 Volume Domain Decomposition

The volume domain decomposition technique as proposed by Valdez-Balderas *et al*[79] could be implemented for the Ferrari SPH algorithm to compare its performance against the technique described in Chapter 7 on multiple GPUs. In scenarios with rapidly moving particles the volume domain decomposition technique requires load balancing to evenly distribute the particles across the available GPUs. Such load balancing is not required in the technique described in the chapter on multiple GPUs because the particles are evenly distributed across the available GPUs at the start of the simulation and remain on those GPUs for the duration of the simulation.

### 8.2.4 Improving Communication on Multiple GPUs

The chapter on multiple GPUs proposed a method of communicating the data between all GPUs by a first communication to a designated GPU on a node followed by communication between those designated GPUs only, then a final communication from the designated GPU to the other GPUs on the node. For two nodes this is optimal. For greater than two nodes this may not be optimal. It may be more efficient to partition the nodes further so that for example with 4 nodes there is a designated GPU for two nodes so that only two GPUs communicate at the highest level instead of four as in the current method. Similarly for 32 GPUs and greater. The aim would be to have just two GPUs communicating with a MPI_Allgather at the highest level with all data being transferred up the chain to this highest level. This could improve communication efficiency and thus accelerate the computation further.

The MPI functions used and described in Chapter 7 on SPH on multiple GPUs are perhaps the most basic MPI functions available. More sophisticated MPI functions are available and could produce better communication performance to reduce total execution time. Similarly the underlying network hardware and topology also influences the

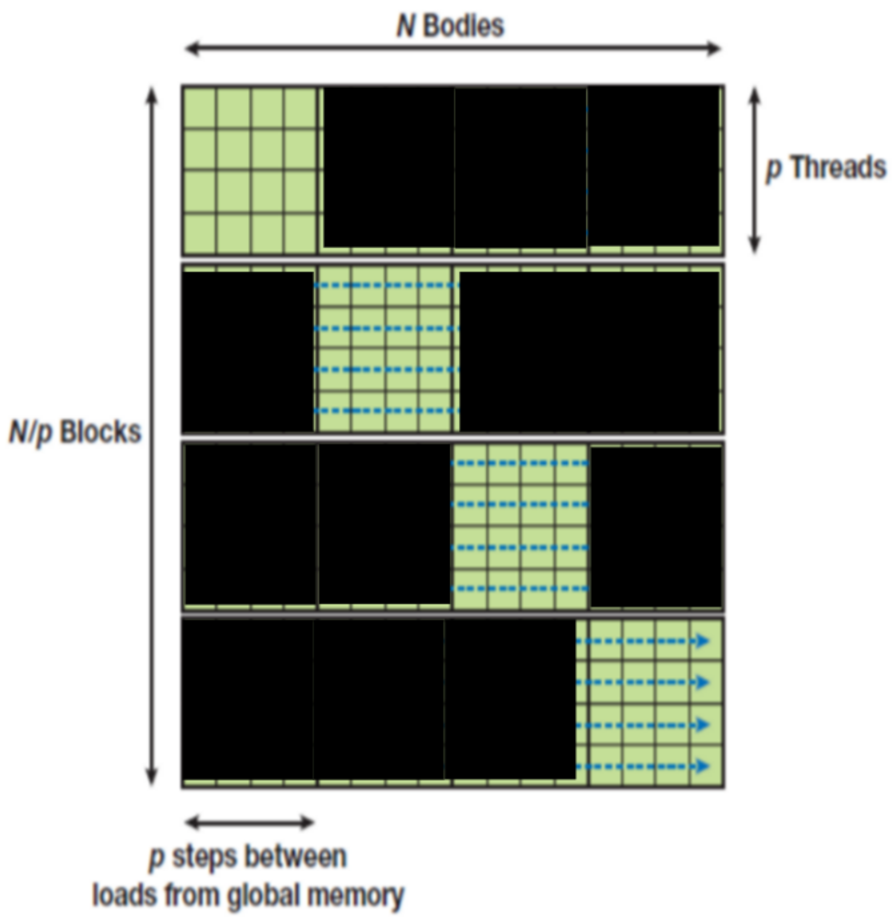


Figure 8.1: The hybrid method processing only a few blocks of sorted data

performance of the algorithm so it is possible that faster communications will make the technique more efficient.

### 8.2.5 Incompressible SPH on GPUs

As briefly mentioned in the chapter on SPH there is a variant of SPH called Incompressible SPH (ISPH) which solves the Poisson equation for pressure instead of using an equation of state. ISPH permits larger time steps than WCSPH so offers the potential for faster simulations. However, while WCSPH has been implemented on multiple GPUs it is unknown how ISPH can be implemented on multiple GPUs, including solving a Poisson equation using one of the linear algebra techniques that have been used for this, including the Bi-CGSTAB method without pre-conditioning as used by Lee *et al.*[21].

### 8.2.6 Implementation on NVIDIA Fermi GPUs

All computation for this thesis was executed on a NVIDIA T10. At the time of writing the T10 is three years old while the NVIDIA Fermi class of GPU has just been released. The Fermi has several advantages over the T10 which are of relevance to this thesis.

1. The Fermi has a different and larger cache structure, which should accelerate SPH computations when using the sorting and texture memory method.

2. The Fermi permits peer to peer memory access for multiple GPUs which should reduce the amount of communication required between GPUs when using the multiple GPU algorithm proposed in this thesis. It may also be possible to increase the number of particles that can be used for a simulation because the particle data can be distributed across all GPUs blocks of which can be transferred to a particular GPU when required rather than being restricted to each GPU requiring a copy of all particle data at all times.

So it would be of great interest to implement the multiple GPU algorithm on a cluster of NVIDIA Fermi GPUs to evaluate the performance and compare it to the results presented in this thesis.

# Appendix A

# A Simple CUDA Program

```
//this code should be saved in a file DeviceTest1.cu

#define N 96

#include <stdio.h>

__device__ int device_array[N];
int size;

__global__ void device_kernel(int* d_host_array);
__device__ void SetDeviceArray();

void allocateArray(void **devPtr,size_t size);
void freeArray(void *devPtr);

main()
{
 int  host_array[N];
 int* d_host_array;
 int i;
 int size_d_host_array;

 size_d_host_array = N*sizeof(int);

 for(i=0 ; i<N ; i++) host_array[i] = 0;

 for(i=0 ; i<N ; i++) printf("\n%i\t%i",i,host_array[i]);

 allocateArray((void**)&d_host_array,size_d_host_array);

 dim3 dimGrid(3,1);
 dim3 dimBlock(32,1,1);
```

```


// Launch the device computation
device_kernel<<<dimGrid, dimBlock>>>(d_host_array);


cudaMemcpy(host_array,
            d_host_array,
            size_d_host_array,
            cudaMemcpyDeviceToHost);

freeArray(d_host_array);

for(i=0 ; i<N ; i++) printf("\n%i\t%i",i,host_array[i]);
}

//////////////////////////////
__global__ void device_kernel(int* d_host_array)
{
int id;
id = blockDim.x * blockIdx.x + threadIdx.x;

SetDeviceArray();

d_host_array[id] = device_array[id];

}

///////////////////////////////
void allocateArray(void **devPtr,size_t size)
{
cudaMalloc(devPtr,size);
}

//////////////////////////////
void freeArray(void *devPtr)
{
cudaFree(devPtr);
}

////////////////////////////////
__device__ void SetDeviceArray()
{
int id;

id = blockDim.x * blockIdx.x + threadIdx.x;
```

```

device_array[id] = id;
}
```

# Appendix B

# The Calculation of Forces for the Shared Memory Implementation

The *main* function calls the following intermediate function *CalcForces* which simply calculates the thread grid for the kernel *calculateforceskernel*. The thread grid is defined by the parameters *numBlocks* and *numThreads*.

```
void CalcForces(float2* d_x,float2* d_v,float* d_mass,
float* d_hsml,float* d_rho,float* d_p,int* d_type,int numBodies,
float* d_drhodt,float2* d_dvdt,int2* d_cell)
{
  int numThreads, numBlocks;

  computeGridSize(numBodies, NUMTHREADS,
                  numBlocks, numThreads);

  #ifdef CUDAGETLASTERROR
  printf("\n\nBefore CalcForces %s\n", cudaGetErrorString(cudaGetLastError()));
  #endif

  calculateforceskernel<<<numBlocks,numThreads>>>
  (d_x,d_v,d_mass,d_hsml,d_rho,d_p,d_type,d_drhodt,
   d_dvdt,d_cell,numBodies);

  cudaThreadSynchronize();

  #ifdef CUDAGETLASTERROR
  printf("\n\nAfter CalcForces %s\n", cudaGetErrorString(cudaGetLastError()));
  #endif

  CUT_CHECK_ERROR("Kernel execution failed");
}
```

This function then calls the kernel *calculateforceskernel* which is executed on the device.

```
__global__ void calculateforceskernel(float2* d_x,float2* d_v,
float* d_mass,float* d_hsml,float* d_rho,float* d_p,int* d_type,
float* d_drhodt,float2* d_dvdt,int2* d_cell,int BLOCKSIZE)
{
  int  threadid = threadIdx.x;
  int  blockid = blockIdx.x;
  int  gtid = blockid* blockDim.x + threadid;

  //declare shared memory variables
  __shared__ float2  shX[NUMTHREADS];
  __shared__ float2  shV[NUMTHREADS];
  __shared__ float  shHsml[NUMTHREADS];
  __shared__ float  shMass[NUMTHREADS];
  __shared__ float  shRho[NUMTHREADS];
  __shared__ float shP[NUMTHREADS];
  __shared__ int shType[NUMTHREADS];
  //__shared__ int2  shCell[NUMTHREADS];


  //local register variables
  float2 myX = d_x[gtid];
  float myHsml = d_hsml[gtid];
  float2 myV = d_v[gtid];
  float myRho = d_rho[gtid];
  float myP = d_p[gtid];
  float mydvdtx = 0.0f;
  float mydvdty = -GRAVITY; //gravity
  float mydrhodt = 0.0f;
  //int2 myCell = d_cell[gtid];
  int myType = d_type[gtid];


  float2 D,V;
  float r, mhsml;
  float xdwdx,ydwdx;
  float K,Kxdwdx,Kydwdx;
  float A,C;
  float Ci,Cj,Cij;
  float2 n;

  int i,j,tile,diff,jparticleid;

  __syncthreads();
```

```
for (i = 0, tile = 0; i < BLOCKSIZE; i += NUMTHREADS, tile++)
{
  //load global data into shared memory
  int idx = tile * NUMTHREADS + threadid;
  shX[threadid] = d_x[idx];
  shV[threadid] = d_v[idx];
  shHsml[threadid] = d_hsml[idx];
  shMass[threadid] = d_mass[idx];
  shRho[threadid] = d_rho[idx];
  shP[threadid] = d_p[idx];
  shType[threadid] = d_type[idx];
  //shCell[threadid] = d_cell[idx];

  __syncthreads();

  for (j = 0; j < NUMTHREADS ; j++)
  {
    jparticleid = tile * NUMTHREADS + j;
    diff = gtid - jparticleid;
    switch(diff)
    {
      case 0 : break;
      default : D.x = myX.x-shX[j].x;
      D.y = myX.y-shX[j].y;
      mhsml = (myHsml + shHsml[j])/2.0;

      r = sqrt(D.x*D.x + D.y*D.y);
      if(r<SCALE*mhsml)
      {
        kerneldw(&xdwdx,&ydwdx,r,mhsml,D);

        D = shX[j] - myX;
        V = shV[j] - myV;
        n = D/r;

        Ci = sqrt(GAMMA*B*pow((myRho/RHO0),GAMMA-1)/RHO0);
        Cj = sqrt(GAMMA*B*pow((shRho[j]/RHO0),GAMMA-1)/RHO0);
        Cij = max(Ci,Cj);

        ////////////////////////////////////////////////
        // DENSITY
        ////////////////////////////////////////////////
        mydrhodt+= -shMass[j]*(V.x*xdwdx + V.y*ydwdx);
        mydrhodt+= shMass[j]*(n.x*xdwdx + n.y*ydwdx)*
                   Cij*(shRho[j] - myRho)/shRho[j];
```

```
          ////////////////////////////////////////////////
          // MOMENTUM
          ////////////////////////////////////////////////

          ////////////////////////////////////////////////
          // FI
          ////////////////////////////////////////////////
          K = myP/(myRho*myRho) + shP[j]/(shRho[j]*shRho[j]);
          K*= shMass[j];

          Kxdwdx = K*xdwdx;
          Kydwdx = K*ydwdx;
          mydvdtx+= -Kxdwdx;
          mydvdty+= -Kydwdx;

          ////////////////////////////////////////////////
          // FV
          ////////////////////////////////////////////////
          A = MU*shMass[j]/(3.0f*myRho*shRho[j]);
          C = (n.x*V.x + n.y*V.y)*(n.x*xdwdx + n.y*ydwdx)/r;

          mydvdtx+= A*(7.0*V.x + 5.0*C*n.x);
          mydvdty+= A*(7.0*V.y + 5.0*C*n.y);
        }
        break;
      }
    }
    __syncthreads();
  }

  //coalesce changes to global memory
  d_dvdt[gtid].x = mydvdtx;
  d_dvdt[gtid].y = mydvdty;
  d_drhodt[gtid] = mydrhodt;

}
```

# Appendix C

# The Calculation of Forces for the Texture Memory Implementation

The *main* function calls the following intermediate function *CalcForces* which calculates the thread grid for the kernel *calculateforceskernel* but also binds textures to the sorted device variables. The thread grid is defined by the parameters *numBlocks* and *numThreads*.

```
void CalcForces(float2* d_sorted_x,float2* d_sorted_v,float* d_sorted_mass,
float* d_sorted_hsml,float* d_sorted_rho,float* d_sorted_p,int* d_sorted_type,
uint2* particleHash,uint* cellStart,int numBodies,int numCells,float thetime,
int actualnreal,float* d_drhodt,float2* d_dvdt,float cellsize,float maxH)
{
  cudaBindTexture(0, d_sorted_xTex, d_sorted_x, numBodies*sizeof(float2));
  cudaBindTexture(0, d_sorted_vTex, d_sorted_v, numBodies*sizeof(float2));
  cudaBindTexture(0, d_sorted_massTex, d_sorted_mass, numBodies*sizeof(float));
  cudaBindTexture(0, d_sorted_hsmlTex, d_sorted_hsml, numBodies*sizeof(float));
  cudaBindTexture(0, d_sorted_rhoTex, d_sorted_rho, numBodies*sizeof(float));
  cudaBindTexture(0, d_sorted_pTex, d_sorted_p, numBodies*sizeof(float));
  cudaBindTexture(0, d_sorted_typeTex, d_sorted_type, numBodies*sizeof(int));
  cudaBindTexture(0, particleHashTex, particleHash, numBodies*sizeof(uint2));
  cudaBindTexture(0, cellStartTex, cellStart, numCells*sizeof(uint));

  int numThreads, numBlocks;
  computeGridSize(numBodies, NUMTHREADS, numBlocks, numThreads);

  #ifdef CUDAGETLASTERROR
  printf("\n\nBefore CalcForces %s\n", cudaGetErrorString(cudaGetLastError()));
  #endif

  CalculateForces<<<numBlocks,numThreads>>>
  (particleHash,cellStart,d_drhodt,d_dvdt,cellsize,maxH);

  #ifdef CUDAGETLASTERROR
  printf("\n\nAfter CalcForces %s\n", cudaGetErrorString(cudaGetLastError()));
  #endif

  CUT_CHECK_ERROR("Kernel execution failed");

  cudaUnbindTexture(d_sorted_xTex);
  cudaUnbindTexture(d_sorted_vTex);
  cudaUnbindTexture(d_sorted_massTex);
  cudaUnbindTexture(d_sorted_hsmlTex);
  cudaUnbindTexture(d_sorted_rhoTex);
  cudaUnbindTexture(d_sorted_pTex);
  cudaUnbindTexture(d_sorted_typeTex);
  cudaUnbindTexture(particleHashTex);
  cudaUnbindTexture(cellStartTex);

}
```

This function then calls the kernel *CalculateForces*. Note that not all the textures are passed to the kernel. This kernel will loop over neighbouring cells only and accumulate contributions to rate of change of density and acceleration from particles in each cell by calling the device function *AccumulateForces*.

```
__global__ void CalculateForces(uint2* particleHash, uint* cellStart,
float* d_drhodt,float2* d_dvdt,float cellsize,float maxH)
{
  int  mySortedIndex = __mul24(blockIdx.x,blockDim.x) + threadIdx.x;

  float2 myX = FETCH(d_sorted_x, mySortedIndex);
  float2 myV = FETCH(d_sorted_v, mySortedIndex);
  float myMass = FETCH(d_sorted_mass, mySortedIndex);
  float myHsml = FETCH(d_sorted_hsml, mySortedIndex);
  float myRho = FETCH(d_sorted_rho, mySortedIndex);
  float myP = FETCH(d_sorted_p, mySortedIndex);
  int myType = FETCH(d_sorted_type, mySortedIndex);

  int2  gridPos2;
  int   gridPos2x,gridPos2y;

  float2  mydvdt;
  float   mydvdtx,mydvdty,mydrhodt;

  // get address in grid
  int2  myGridPos = calcGridPos(myX,cellsize);

  mydvdtx = 0.0f;
  mydvdty = 0.0f;
  mydrhodt = 0.0f;
  volatile uint2 sortedData = particleHash[mySortedIndex];
  uint myTrueIndex = sortedData.y;

  // examine only neighbouring cells
  for(int y=LOWCELL; y<HICELL; y++)
  {
    gridPos2y = myGridPos.y + y;
    for(int x=LOWCELL; x<HICELL; x++)
    {
      gridPos2x = myGridPos.x + x;
      {
        gridPos2.x = gridPos2x;
        gridPos2.y = gridPos2y;

        AccumulateForces(gridPos2,mySortedIndex,myX,myV,
                         myHsml,myMass,myRho,myP,myType,
                         particleHash,cellStart,
                         &mydvdtx,&mydvdty,&mydrhodt);
      }
    }
  }

  mydvdt.x = mydvdtx;
  mydvdt.y = mydvdty;

  d_drhodt[myTrueIndex] = mydrhodt;
  d_dvdt[myTrueIndex] = mydvdt;
  d_dvdt[myTrueIndex].y+= -GRAVITY;
}
```

This device function *AccumulateForces* looks for interactions between one particular particle, represented by the thread calling the function, and any particles in the cell with the grid position *gridPos2*, a parameter passed to the function by the calling kernel *CalculateForces*. Note that the textures that were bound in the function *CalcForces* above are referenced in the function below but are not passed down through parameter lists.

```
__device__ void AccumulateForces(int2 gridPos2,int myIndex,float2 myX,
float2 myV,float myHsml,float myMass,float myRho,float myP,int myType,
uint2* particleHash, uint* cellStart,float* mydvdtx,float* mydvdty,float* mydrhodt)
{
  float2 D,V;

  uint cellgridHash = calcGridHash(gridPos2);
  float r, mhsml;

  float xdwdx,ydwdx;
  float K,Kxdwdx,Kydwdx;
  float A,C;
  float Ci,Cj,Cij,rhoj,massj,pj,hsmlj;
  float2 xj,vj,n;

  int  myTrueIndex = FETCH(particleHash, myIndex).y;

  // get start of bucket for this cell
  uint bucketStart = FETCH(cellStart, cellgridHash);
  if (bucketStart == 0xffffffff) return;

  // iterate over particles in this cell
  for(uint i=0; i<MAXPARTICLESPERCELL; i++)
  {
    uint index2 = bucketStart + i;
    uint2 cellData = FETCH(particleHash, index2);
    if (cellData.x != cellgridHash) break;

    // check not colliding with self
    if (index2 != myIndex)
    {
      xj = FETCH(d_sorted_x,index2);
      hsmlj = FETCH(d_sorted_hsml,index2);
      D = myX - xj;
      r = sqrt(D.x*D.x + D.y*D.y);
      mhsml = (myHsml + hsmlj)/2.0;

      if(r<SCALE*mhsml)
      {
        xj = FETCH(d_sorted_x,index2);
        vj = FETCH(d_sorted_v,index2);
        rhoj = FETCH(d_sorted_rho,index2);
        pj = FETCH(d_sorted_p,index2);
        massj = FETCH(d_sorted_mass,index2);
        hsmlj = FETCH(d_sorted_hsml,index2);

        kerneldw(&xdwdx,&ydwdx,r,mhsml,D);

        D = xj - myX;
        V = vj - myV;
        n = D/r;

        Ci = sqrt(GAMMA*B*pow((myRho/RHO0),GAMMA-1)/RHO0);
        Cj = sqrt(GAMMA*B*pow((rhoj/RHO0),GAMMA-1)/RHO0);
        Cij = max(Ci,Cj);
```

```
        ////////////////////////////////////////////////
        // DENSITY
        ////////////////////////////////////////////////
        //Ferrari Riemann
        (*mydrhodt)+= -massj*(V.x*xdwdx + V.y*ydwdx);
        (*mydrhodt)+= massj*(n.x*xdwdx + n.y*ydwdx)*
                      Cij*(rhoj - myRho)/rhoj;

        ////////////////////////////////////////////////
        // MOMENTUM
        ////////////////////////////////////////////////
        ////////////////////////////////////////////////
        // FI
        ////////////////////////////////////////////////
        K = myP/(myRho*myRho) + pj/(rhoj*rhoj);
        K*= massj;

        Kxdwdx = K*xdwdx;
        Kydwdx = K*ydwdx;
        *(mydvdtx)+= -Kxdwdx;
        *(mydvdty)+= -Kydwdx;

        ////////////////////////////////////////////////
        // FV
        ////////////////////////////////////////////////
        A = MU*massj/(3.0f*myRho*rhoj);
        C = (n.x*V.x + n.y*V.y)*(n.x*xdwdx + n.y*ydwdx)/r;
        *(mydvdtx)+= A*(7.0*V.x + 5.0*C*n.x);
        *(mydvdty)+= A*(7.0*V.y + 5.0*C*n.y);
      }
    }
  }
}
```

Also note that the textures are referenced through the function *FETCH* which is defined with

```
#define FETCH(t, i) tex1Dfetch(t##Tex, i)
```

The textures are declared in the same module as the kernel.

```
texture<float2, 1, cudaReadModeElementType> d_xTex;
texture<float2, 1, cudaReadModeElementType> d_vTex;

texture<float, 1, cudaReadModeElementType> d_massTex;
texture<float, 1, cudaReadModeElementType> d_hsmlTex;
texture<float, 1, cudaReadModeElementType> d_pTex;
```

```
texture<float, 1, cudaReadModeElementType> d_cTex;
texture<float, 1, cudaReadModeElementType> d_rhoTex;

texture<int, 1, cudaReadModeElementType> d_typeTex;

texture<float2, 1, cudaReadModeElementType> d_sorted_xTex;
texture<float2, 1, cudaReadModeElementType> d_sorted_vTex;

texture<float, 1, cudaReadModeElementType> d_sorted_massTex;
texture<float, 1, cudaReadModeElementType> d_sorted_hsmlTex;
texture<float, 1, cudaReadModeElementType> d_sorted_pTex;
texture<float, 1, cudaReadModeElementType> d_sorted_cTex;
texture<float, 1, cudaReadModeElementType> d_sorted_rhoTex;

texture<int, 1, cudaReadModeElementType> d_sorted_typeTex;

texture<uint2, 1, cudaReadModeElementType> particleHashTex;
texture<uint, 1, cudaReadModeElementType> cellStartTex;
```

# Bibliography

[1] Harlow FH, Welch JE. Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface. *The Physics of Fluids*, 8(12):2182 2189, 1965.

[2] Mingham CG, Causon DM. High-resolution finite-volume method for shallow water flows. *Journal of Hydraulic Engineering*, 124(6):605–614, 1998.

[3] Qian L, Causon DM, Ingram DM, Mingham CG. Cartesian Cut Cell two-fluid solver for hydraulic flow problems. *Journal of Hydraulic Engineering*, 129:688 – 696, 2003.

[4] Qian L,Causon DM, Mingham CG, Ingram DM. A free-surface capturing method for two fluid flows with moving bodies. *Proceedings of The Royal Society A*, 462:21–42, 2006.

[5] Hirt CW, Nichols BD. Volume of fluid (VOF) method for the dynamics of free boundaries. *Journal of Computational Physics*, 39(1):201 – 225, 1981.

[6] Martin JC, Moyce WJ. Part IV. An experimental study of the collapse of liquid columns on a rigid horizontal plane. *Philosophical Transactions of the Royal Society London*, 244:312 – 324, 1952.

[7] Kleefsman KMT, Fekken G, Veldman AEP, Iwanowski B, Buchner B. A Volume-of-Fluid based simulation method for wave impact problems. *Journal of Computational Physics*, 206:363 – 393, 2005.

[8] Greaves D. Numerical simulation of breaking waves using the volume of fluid method. In *23rd International Workshop on Water Waves and Floating Bodies*, pages 65–68, 2008.

[9] Wang JP, Borthwick AGL, Eatock Taylor R. Finite-volume type VOF method on dynamically adaptive quadtree grids. *International Journal for Numerical Methods in Fluids*, 45(5):485 – 508, 2003.

[10] Greaves D, Borthwick AGL. On the use of adaptive hierarchical meshes for numerical simulation of separated flows. *International Journal for Numerical Methods in Fluids*, 26(3):303 – 322, 1998.

[11] Koshizuka S, Oka Y. Moving-Particle Semi-implicit method for fragmentation of incompressible fluid. *Nuclear Science and Engineering*, 123:421 – 434, 1996.

[12] Archibald S. *Modelling of Extreme Ocean Waves Using High Performance Computing*. PhD thesis, Imperial College London, 2011.

[13] Gratton S. Graphics Card Computing for Cosmology: Cholesky Factorization. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, 2010.

[14] Brandvik T, Pullan G. Acceleration of a two-dimensional Euler solver using commodity graphics hardware. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 221(12):1745 – 1748, 2007.

[15] Brandvik T, Pullan G. Acceleration of a 3D Euler solver using commodity graphics hardware. In *46th AIAA Aerospace Sciences Meeting and Exhibit*, 2008.

[16] Westphal E. Multiparticle collision dynamics on GPU. In *Book of Abstracts SimGPU2011*, 2011.

[17] Griebel M, Zaspel P. Solving the two-phase incompressible Navier-Stokes Equations on massively parallel multi-GPU clusters. In *Proceedings of Parallel CFD 2011*, 2011.

[18] Berentzen I. Astrophysical mesh- and particle-based simulations. In *Book of Abstracts SimGPU2011*, 2011.

[19] Spurzem R, Berczik P, Berentzen I, Nitadori K, Hamada T, Marcus G, Kugel A, Mnner R, Fiestas J, Banerjee R, Klessen R. Astrophysical particle simulations with large custom GPU clusters on three continents . *Computer Science - Research and Development*, 26:145 – 151, 2011.

[20] Gomez-Gesteira M, Rogers BD, Dalrymple RA, Crespo AJC, Narayanaswamy M. User Guide for the SPHysics Code v2.0, 2010.

[21] Lee ES, Moulinec C, Xu R, Violeau D, Laurence D, Stansby P. Comparisons of weakly compressible and truly incompressible algorithms for the SPH mesh free particle method. *Journal of Computational Physics*, 227:8417 – 8436, 2008.

[22] Lee ES, Violeau D, Issa R, Ploix S. Application of weakly compressible and truly incompressible SPH to 3-D water. *Journal of Hydraulic Research*, 48 Extra Issue:50 – 60, 2010.

[23] Liu GR, Liu MB. *Smoothed Particle Hydrodynamics.* World Scientific, 2005.

[24] Monaghan JJ. Smoothed Particle Hydrodynamics. *Reports on Progress in Physics*, 68:1703 – 1759, 2005.

[25] Gingold RA, Monaghan JJ. Kernel estimates as a basis for general particle methods in hydrodynamics. *Journal of Computational Physics*, 46:429 – 453, 1982.

[26] Monaghan JJ, Gingold RA. Shock simulation by the particle method SPH. *Journal of Computational Physics*, 52:374 – 389, 1983.

[27] Monaghan JJ. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30:543 – 574, 1992.

[28] Cleary PW. Modelling confined multi-material heat and mass flows using SPH. *Applied Mathematical Modelling*, 22:981 – 993, 1998.

[29] Gingold RA, Monaghan JJ. Smoothed Particle Hydrodynamics : Theory and application to Non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375 – 389, 1977.

[30] Parshikov AN, Stanislav AM, Loukashenko II, Milekhin IA. Improvements in SPH method by means of interparticle contact algorithm and analysis of perforation tests at moderate projectile velocities. *International Journal of Impact Engineering*, 24(8):779 – 796, 2000.

[31] Robinson M. *Turbulence and Viscous Mixing using Smoothed Particle Hydrodynamics*. PhD thesis, Monash University, 2009.

[32] Sigalotti LDG, Lopez H, Donoso A, Sira E, Klapp J. A shock-capturing SPH scheme based on adaptive kernel estimation. *Journal Computational Physics*, 212:124 – 149, 2006.

[33] Monaghan JJ. Simulating free surface flows with SPH. *Journal of Computational Physics*, 110(2):399 – 406, 1994.

[34] Monaghan JJ, Kos A. Solitary waves on a Cretan beach. *Journal of Waterway, Port, Coastal and Ocean Engineering*, 125:145 – 154, 1999.

[35] Monaghan JJ, Kos A, Issa N. Fluid motion generated by impact. *Journal of Waterway Port, Coastal and Ocean Engineering*, 129(6):250 – 259, 2003.

[36] Monaghan JJ, Kajtar JB. SPH particle boundary forces for arbitrary boundaries. *Computer Physics Communications*, 180:1811 – 1820, 2009.

[37] Morris JP, Fox PJ, Zhu Y. Modeling low Reynolds number incompressible flows using SPH. *Journal of Computational Physics*, 136:214 – 226, 1997.

[38] Dalrymple RA, Knio O. SPH modelling of water waves. In *Proc. Coastal Dynamics*, 2000.

[39] Crespo AJC, Gomez-Gesteira M, Dalrymple RA. Boundary conditions generated by dynamic particles in SPH. *Computers, Materials & Continua*, 5(3):173 – 184, 2007.

[40] Colagrossi A, Landrini M. Numerical simulation of interfacial flows by smoothed particle hydrodynamics. *Journal of Computational Physics*, 191:448 – 475, 2003.

[41] Cummins SJ, Rudman M. An SPH projection method. *Journal of Computational Physics*, 152:584 – 607, 1999.

[42] Ferrari A, Dumbser M, Toro EF, Armanini A. A new 3D parallel SPH scheme for free surface flows. *Computers & Fluids*, 38:1203 – 1217, 2009.

[43] Lo EYM, Shao S. Simulation of near-shore solitary wave mechanics by an imcompressible SPH method. *Applied Ocean Research*, 24:275 – 286, 2002.

[44] Violeau D, Issa R. Numerical modelling of complex turbulent free-surface flows with the SPH method : an overview. *International Journal for Numerical Methods in Fluids*, 53:277 – 304, 2007.

[45] Harada T, Koshizuka S, Kawaguchi Y. Improvement of the boundary conditions in Smoothed Particle Hydrodynamics. *Computer Graphics & Geometry*, 9(3):2 – 15, 2007.

[46] Shepard D. A two dimensional function for irregularly spaced data. In *ACM National Conference*, 1968.

[47] Dilts GA. Moving-LeastSquares-Particle Hydrodynamics I. Consistency and stability. *International Journal for Numerical Methods in Engineering*, 44:1115 – 1155, 1999.

[48] Bonet J, Lok TSL. Variational and momentum preservation aspects of Smoothed Particle Hydrodynamic formulations. *Computer Methods in Applied Mechanics and Engineering*, 180:97 – 115, 1999.

[49] Swegle JW, Hicks DL, Attaway SW. Smoothed particle hydrodynamics stability analysis. *Journal of Computational Physics*, 116:123 – 134, 1995.

[50] Monaghan JJ. SPH without a tensile instability. *Journal of Computational Physics*, 159:290 – 311, 2000.

[51] Monaghan JJ. On the problem of penetration in particle methods. *Journal of Computational Physics*, 82:1 – 15, 1989.

[52] Hughes J, Graham D. Comparison of incompressible and weakly-compressible SPH models for free-surface water flows. *Journal of Hydraulic Research*, 48 Extra Issue:105  117, 2010.

[53] Batchelor GK. *Introduction to fluid dynamics*. Cambridge University Press, 1974.

[54] Oger G, Doring M, Alessandrini B, Ferrant P. Two-dimensional SPH simulations of wedge water entries. *Journal of Computational Physics*, 213:803 – 822, 2006.

[55] NVIDIA. NVIDIA CUDA Programming Guide 2.3.1. 2009.

[56] Wong H. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems Software ISPASS 2010*, 2010.

[57] McCabe C, Causon DM, Mingham CG. GPU accelerated calculations of free surface flows. In *Proceedings of 4th SPHERIC Workshop*, 2009.

[58] Ferrari A. SPH simulation of free surface flow over a sharp-crested weir. *Advances in Water Resources*, 33:270 – 276, 2010.

[59] Nyland L, Harris M, Prins J. *GPU Gems 3*, chapter Fast N body simulation with CUDA. Addison-Wesley, 2007.

[60] Le Grand S. *GPU Gems 3*, chapter Broad-phase collision detection with CUDA. Addison-Wesley, 2007.

[61] Ubbink O. *Numerical prediction of two fluid systems with sharp interfaces*. PhD thesis, Imperial College London, 1997.

[62] Antuono M, Colagrossi A, Marronea S, Molteni D. Free-surface flows solved by means of SPH schemes with numerical diffusive terms. *Computer Physics Communications*, 181:532 – 549, 2010.

[63] Rogers BD, Dalrymple RA, Stansby PK. Simulation of caisson breakwater movement using 2D SPH. *Journal of Hydraulic Research*, 48 Extra Issue:135 – 141, 2010.

[64] Vila JP. On particle weighted methods and Smooth Particle Hydrodynamics. *Mathematical Models and Methods in Applied Science*, 9(2):161 – 209, 1999.

[65] Toro EF, Spruce M, Speares W. Restoration of the contact surface in the HLL Riemann solver. *Shock Waves*, 4:25 – 34, 1994.

[66] Roubtsova V, Kahawita R. The SPH technique applied to free surface flows. *Computers & Fluids*, 35:1359 – 1371, 2006.

[67] Omidvar P, Stansby PK, Rogers BD. Wave body interaction in 2D using smoothed particle hydrodynamics (SPH) with variable particle mass. *International Journal for Numerical Methods in Fluids*, 2011.

[68] Monaghan JJ. SPH and Riemann Solvers. *Journal of Computational Physics*, 136:298–307, 1997.

[69] Parshikov AN. Application of a solution of the Riemann problem in the SPH method. *Computational Mathematics and Mathematical Physics*, 39(7):1173 – 1182, 1999.

[70] Parshikov AN, Medin SA. Smoothed Particle Hydrodynamics using interparticle contact algorithms. *Journal of Computational Physics*, 180(1):358 – 382, 2002.

[71] Inutsuka S. Reformulation of Smoothed Particle Hydrodynamics with Riemann Solver. *Journal of Computational Physics*, 179:238 – 267, 2002.

[72] Colella P, Woodward PR. The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations. *Journal of Computational Physics*, 54:174 – 201, 1984.

[73] van Leer B. Towards the ultimate conservative difference scheme. *Journal of Computational Physics*, 135:229 – 248, 1997.

[74] Cha SH, Whitworth AP. Implementations and tests of Godunov-type particle hydrodynamics. *Mon. Not. R. Astron. Soc.*, 340:73 – 90, 2003.

[75] Molteni D, Bilello C. Riemann solver in SPH. *Memorie della Societa Astronomica Italiana Supplement*, 1:36 – 44, 2003.

[76] Cherfils JM, Blonce L, Pinon G, Rivoalen E. Towards the simulation of wave-body interactions with SPH. In *Proceedings of 4th SPHERIC Workshop*, 2009.

[77] Su SW, Lai MC, Lin CA. An immersed boundary technique for simulating complex flows with rigid boundary. *Computers & Fluids*, 36:313 – 324, 2007.

[78] Zhou ZQ, De Kat JO, Buchner B. A nonlinear 3D approach to simulate green water dynamics on deck. In *Seventh international conference on numerical ship hydrodynamics*, 1999.

[79] Valdez-Balderas D, Dominguez JM, Crespo AJC, Rogers BD. Developing massively parallel SPH simulations on multi-GPU clusters. In *6th International SPHERIC Workshop*, 2011.

[80] Ivings MJ, Causon DM, Toro EF. On Riemann Solvers for compressible liquids. *Intl Journal for Numerical Methods in Fluids*, 28:395 – 418, 1992.