Fluid Dynamics using General Purpose GPU Programming

Gurpal Sian | 140656480

BSc Computer Science with Game Engineering

Project Supervisor: Dr Graham Morgan

Abstract

This paper will describe how algorithms, such as those used for representing the flow of fluids, can be accelerated and optimised using graphics cards and parallel programming methods. The nature of these methods will be for real time purposes such as in video games.

A complete, real-time Fluid Solver implementation will be presented along with benchmarked results showing how the use of parallel programming can greatly improve upon compute time for specific algorithms.

Declaration

"I declare that this dissertation represents my own work except where otherwise stated."

Acknowledgements

I would like to thank the joint efforts of the Game Engineering supervisors, Dr William Blewitt, Dr Graham Morgan and Dr Gary Ushaw, for all of their help and support throughout the course of this project.

Contents

Table of Figures

1. Introduction

1.1 Motivation

Video games are designed to immerse players into a believable world where our laws of physics (usually) apply. It is important that the physics is modelled correctly otherwise the immersion breaks down. Adding real world systems, such as a waterfall, or smoke rising from a fire, into a game means having to calculate, to some degree of accuracy the dynamics of these systems.

Fluid dynamics is one branch of continuum mechanics which aids in representing how fluids behave in nature. Computational algorithms are designed to do this by utilising the processing power of modern computers. The algorithms must perform many calculations repeatedly to generate the correct results. Computational Fluid Dynamics is used when high precision is required for example modelling the turning effect of a ship in water [1]. Thanks to these computations, time and money can be saved since the physics can be modelled very accurately without the need to build real models.

The algorithms used in Computational Fluid Dynamics are generally not well suited to real time applications, since the data sets are incredibly large. The higher the algorithms numerical accuracy, the slower the application will run. On the other hand, the lower the accuracy, the faster the application will perform. When designing airplanes or ships for example, it is obvious that numerical accuracy is paramount, however this is seldom important in video games. In a game, the physics just needs to “look” correct (that is to say, the physics behaves as intended). Games are also real-time applications which means the physics simulations cannot take too long to compute, otherwise the game performance will be diminished.

Fluid flows are usually animated into a game which provides the believability but does not capture the realism of the physics. Since game physics does not require the same amount of precision as found in traditional Computational Fluid Dynamics, the algorithms used in games do not need to be as complex. This means algorithms can be designed to compute believable physics while still maintaining the real-time performance of the game.

Graphics Processing Units (GPUs) are highly specialised hardware designed for very intensive parallel computations [2]. GPUs typically have thousands of cores which means that they can perform many operations simultaneously whereas a normal Central Processing Unit (CPU) in a computer only has up to four cores (meaning up to four threads can be executed at the same time). Since fluid algorithms require calculating the values of large amounts of different data points independently from one another, this kind of computation is best suited for the GPU. The highly parallel nature of the GPU architecture means that, depending on the number of cores on a GPU, it can perform up to thousands of operations in the same time it takes a CPU to perform one.

1.2 Aim and Objectives

The overall aim of the project is To Investigate the Accelerated Computation of Fluid Dynamics Using GPGPU Programming. Through research and testing this paper will describe, step by step, how the GPU can be used to optimise the computation of parallelisable algorithms. We will present a real time fluid simulation which runs on a CPU (single threaded) and a GPU (many threaded) and compare the performance differences between them.

This will be accomplished by five primary objectives:

- Determine an appropriate API (Application Programming Interface) to use for parallel programming
- Determine an appropriate representation to model fluids on a CPU and GPU
- Implement the fluid solver on the CPU and the GPU
- Integrate the solver with a graphical environment to display the simulation on screen and allow user interaction
- Assess the performance differences between the CPU and GPU implementations

Each objective will build upon the previous objectives, thus the order in which they are completed is very important.

To be able to develop programs that run on the GPU, a parallel programming API is needed, the two most popular being CUDA by NVIDIA and OpenCL by the Khronos Group. The API chosen will need to provide an easy way to copy data to and from the GPU while also being easy to program with.

There are many ways to represent fluids using computers. The choice here will be determined by the visual fidelity of the representation and also how parallelisable the fluid solver algorithm is.

The solver will be developed incrementally. This means one function from the solver will be written and then integrated with the graphical environment to ensure the function works as intended. Once the function works correctly, the next function in the solver will be developed and so on. This makes debugging easier since a function will only be added once the previous functions all work correctly.

Finally testing will be done in two phases. Phase one will compare CPU and GPU performances over a large domain providing the bigger picture. Then phase two will look at where the performances overlap between the two implementations using a smaller domain. Results will be obtained via two metrics: overall compute time of the solver and the framerate of the simulation. Objectives one to four will be discussed further in the Methodology (section 3) of the paper while objective five will be part of the Results (section 4) and Evaluation (section 5).

The overall timeline for the project is as follows:
Deadlines

28/10/16 – Ethics approval form	COMPLETE
01/11/16 – Determine API and Fluid approach (Objective 1 and 2)	COMPLETE
04/11/16 – Project Presentation	COMPLETE
02/12/16 – Project Proposal	COMPLETE
09/01/17 – Implement the Fluid Simulation (Objective 3)	COMPLETE
17/02/17 – Integrate Simulation with graphics framework (Objective 4)	COMPLETE
13/03/17 – Carry out performance analysis (Objective 5)	COMPLETE
24/04/17 – Project Poster	COMPLETE
05/05/17 – Dissertation hand in	COMPLETE

The deadlines in green are objective deadlines. Each objective is required to complete the project.

1.3 Paper Structure

Introduction

The Introduction will present the domain of the project along with an explanation of the aim and objectives. This chapter is divided into three subsections:

- Motivation
- Aim and Objectives
- Paper Structure

Background and Literature Review

This chapter aims to convey the state of the art of the domain introduced in chapter one. The subsections are:

- Fluids and their Equations
- Computational Fluid Dynamics
- Fluid Representations
- Real-Time Simulations
- The Graphics Processing Unit
- GPGPU Programming

- GPU Accelerated Algorithms

Methodology

The Methodology is where the design and implementation of the solver will be explained. Since the aim is to compare the GPU performance against the CPU, both solvers will be discussed. Justifications will be made for the use of certain technologies and software designs. The Methodology consists of:

- CPU Implementation
- Graphical Environment
- GPU Implementation

Testing and Results

This chapter will compare the performances of both the CPU and GPU solvers. The metrics used will be discussed along with results of the testing. Subsections include:

- Strategy and Metrics
- Test Phase One
- Test Phase Two

Analysis and Evaluation

The Evaluation will review the results from the previous chapter and also outline how the results reflect the overall project aim. The results will also be reviewed alongside current findings from the state of the art of GPGPU programming. Any anomalous or interesting results will be highlighted. The Evaluation will include:

- Computation Time
- Framerate Analysis
- Optimisations and Extension

Conclusion

The final chapter will summarise the project, aim and objectives and discuss any further work which can be done in the project domain or other relevant areas.

- Overall Summary
- Further Work

2. Background and Literature Review

2.1 Fluids and their Equations

A fluid is any physical material that flows (which means it fills the volume it is contained in) and deforms or changes shape under an applied force [3]. Fluid dynamics is the study of how fluids behave in nature. This allows for the accurate modelling of physical systems to give us a better understanding of the real world. Analysing weather patterns and hazards, monitoring the effects of cooling systems on electrical components, measuring the performance of vehicles or even how blood flows through the human body all involve a detailed grasp of fluid dynamics [4], and through this research, many aspects of science and technology can be improved.

The Navier-Stokes equations are a set of mathematical formulae which describe how viscous fluids move. Viscosity is a property of fluids which determines how easily they deform when shear stress is applied to them [5]. The derivation of these equations stems from applying Newton’s second law of motion to fluids, incorporating fluid stress (viscosity) and finally a pressure term [6].

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t}+\mathbf{u}\cdot\nabla\mathbf{u}\right)=-\nabla p+\nabla\cdot\left(\mu(\nabla\mathbf{u}+(\nabla\mathbf{u})^{T})-\frac{2}{3}\mu(\nabla\cdot\mathbf{u})\mathbf{I}\right)+\mathbf{F}$$

Figure 1: Navier-Stokes equation of momentum [7]

$$\frac{\partial \rho}{\partial t}+\nabla\cdot(\rho\mathbf{u})=0$$

Figure 2: Navier-Stokes continuity equation [7]

The first equation can be simplified by condensing each of the four terms:

$$\rho\frac{\partial \mathbf{u}}{\partial t}=-\nabla p+\nabla\cdot T+\mathbf{F}$$

Figure 3: General form for the Navier-Stokes equation [6]

The first argument denotes the inertial force on each fluid particle. This force is composed of the three terms on the right-hand side of the equation.

The second term is a pressure term. This means that the fluid will not lose volume as it is always pressing against itself.

The third term is a stress term or viscous term. Motion of fluid can be caused by friction and shear stress. This means that moving liquid will cause nearby liquid to begin moving through turbulence. Turbulence is caused by shear stress. In other words, as the viscosity of a fluid changes, so does its ability to flow.

The final term is the force term and describes the external forces which act on every single fluid particle.

The first equation represents how fluid should conserve momentum while the continuity equation governs how fluids should be mass conserving. These two are solved together to determine how a fluid will behave in a given environment. The equations almost always result in a set of nonlinear partial differential equations (PDEs) [6]. PDEs involve rates of change with respect to continuous variables and due to the difficulty in solving them, only simple solutions can be found, for example how fluid moves in a circular pipe. However, when the geometry becomes more complex, solutions become harder to achieve [7].

Depending on the accuracy of the fluid model needed, certain terms can be simplified or required to have more detail, for example given a specific type of fluid, an expression for the stress tensor term (T) must be determined. Also, if a fluid is compressible (meaning it has variable density, such as air), then an equation of state and an equation for the conservation of energy are needed [7]. By contrast, an incompressible fluid, like water, would not require as much detail to model accurately. For incompressible flows, density is assumed to be constant, and thus the continuity equation reduces to:

$$\nabla \cdot (\rho\mathbf{u}) = 0$$

Where ρ (Greek letter rho) is a constant value. In nature, there is no such thing as a truly incompressible flow, since if there was, people would not be able to scream underwater for example. Fluids like water are modelled as incompressible because this gives a good approximation and makes the maths much simpler. Gases such as air which move at speeds lower than Mach 0.3, are also considered incompressible.

2.2 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is the approach to solving fluid dynamics problems using computers, physics and numerical methods. The Navier-Stokes equations were first derived in the 1800s by Claude-Louis Navier, a French engineer and Sir George Gabriel Stokes, a British Scientist. For decades, solving them was done by hand as the need for solutions pre-dated the development of electronic computers (for example in aircraft design). With the emergence of the modern computer came the ability to solve more complex problems and use advanced numerical solutions for fluid dynamics [8].

A numerical method is a way of finding the solution to numerical problem by way of repeating the steps again and again to arrive at approximate answers. Since the answers are approximations, a true exact answer can often not be found, no matter how accurate the method is. Numerical methods are often used in cases where finding the exact answer could be too time consuming or not even possible via a different approach [9].

CFD involves taking a continuous problem and splitting it into discrete sections. This is known as Discretization. By doing this, solving the dynamics of a fluid becomes easier as there are specific time intervals within which calculations will be made and at specific, distinct locations in the fluid. The discretization process forms a grid within which the properties of the fluid are only calculated at the individual gird points. The values in-between can then be interpolated across to achieve values everywhere for the fluid. Once the discrete system has been set up, the PDEs can be solved by a computer since this requires repeating calculations many times. Discretization is done via a numerical method and as such it is these numerical methods which allow approximations to be found for PDEs. The most dominant Numerical Method for Discretization is the Finite Difference Method [10].

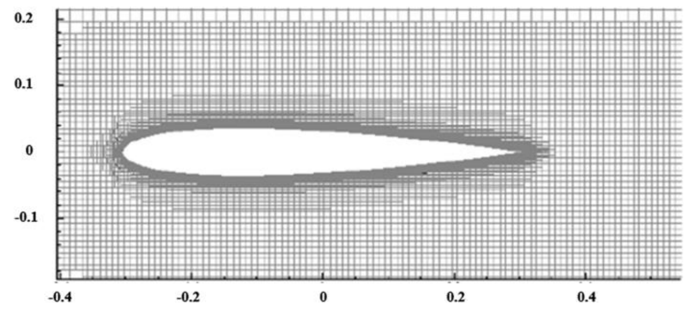


Figure 4: Air flow around wing [11].

This shows how the air flowing around a wing can be discretized into a grid. Variables are calculated at distinct grid points.

Computational Fluid Dynamics requires high numerical accuracy when modelling flows, since, it is crucial that the dynamics of air flow around the wing of a plane be modelled correctly. Due to the precision and the immense number of calculations needed to model fluids, the simulations produced are not suited for real-time applications. Even on supercomputers employed today, it can take up to hours to simulate flows on huge data sets (on the order of tens of millions of data points) [12].

2.3 Fluid Representations

In computer graphics, there are many ways to simulate fluid motion, the two most common being grid based (Eulerian) and particle based (Lagrangian) techniques [13]. The Lagrangian viewpoint defines fluid as discrete points or blobs. Each point has all the properties of the fluid defined such as velocity, mass, density etc. The result is conservation of mass is given for free as all the particles in the fluid have their mass defined. This method is also usually easier to program and tends to run very fast making it a prime choice for real-time applications.

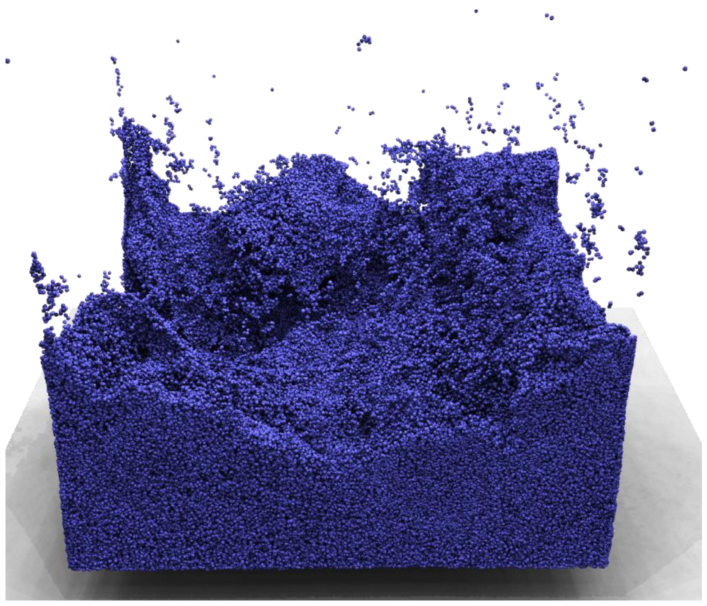


Figure 5: Particle fluid simulation [14].

A particle based simulation. The technique here is known as Smoothed Particle Hydrodynamics (SPH).

The Eulerian viewpoint treats space as a fixed grid and models the fluid as it flows past. The key difference is that the fluid properties are recorded at each cell in the grid instead of being tied directly to the fluid itself. Grid simulations tend to allow for higher numerical accuracy since there aren’t an exponential number of particles to consider however the result is they can run much slower and require more steps to ensure mass is conserved as opposed to particle based counterparts. Finally, due to their higher numerical accuracy, grid techniques often look better than particle simulations.

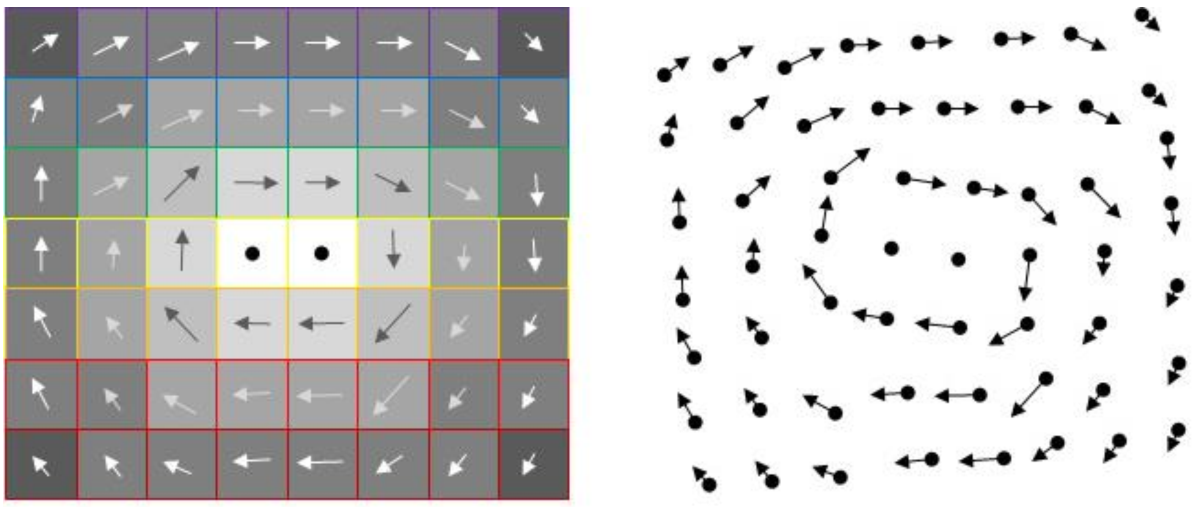


Figure 6: Grid simulation of fluids [15].

Grid simulation. The images show how the simulation tracks the velocity of the fluid as it moves through space. The left grid also has information like temperature (cell colour), density (cell fill) and pressure (arrow colour) stored.

2.4 Real-Time Simulations

2.4.1 Physics in Games

Video games are real-time applications which immerse players into a world with relatively realistic physics. Games employ various ways of representing real world physical phenomena for players to interact with, such as breakable environments or the trajectory of bullets. In contrast, some physical systems may not be represented correctly in a game like smoke from an explosion or water flowing down a river. When a player interacts with such systems, the immersion breaks down as the structures in question do not change behaviour when external forces are applied to them.

These kinds of physics (such as the river example) are usually animated into the scene [16]. Players can see that the water flows downstream, but the second they enter the river, the water does not flow around them or change velocity. Video games however do not require the high level of detail employed in traditional CFD simulations. It is enough for the game to have correct results rather than hyper accurate results. Therefore, real-time algorithms can be designed to provide the correct movements of fluids in games without hurting the performance of the application.

2.4.2 Fluid Animation in Games

Instead of using CFD to model fluids in games, another technique, known as Fluid Animation can be used to produce visually appealing models while still being somewhat accurate to the real world [17]. The water flowing down the river is now modelled using techniques similar to those in CFD and thus can produce animations which resemble real world counterparts. Animators and game designers need no longer sacrifice physical accuracy for visual fidelity. Fluid animation is much better suited to real-time applications like video games since they do not require the rigorous scientific accuracy that CFD requires.

2.4.3 The Limitations of Hardware

The algorithms designed for real-time physics calculations work well for small systems however as mentioned in paper [17], these equations scale poorly, meaning that as the data set size increases, performance of the algorithms decrease, thus hurting real-time performance of the game. [18] also discusses how adding physical complexity to the game adds increased load onto the CPU. Thus, many games still choose to simply animate the physics of certain phenomena into the scene or use less accurate models to ensure smooth framerates. A processor in a computer can only run so fast, for example the Intel i7-4790K quad core CPU can run at 4 GHz (gigahertz) [19] meaning it can process data at a rate of 4 billion cycles a second. CPUs can perform a limited number of instructions per cycle, in the case of the CPU mentioned, it can perform 32 single precision Floating Point Operations per Second (FLOPs) per cycle. With 4 cores that's a theoretical total of 512 Giga FLOPS.

While that figure may be large, it is important to note it is only a theoretical max. Other factors such as how the algorithm uses multiple cores (if at all) or other processes using the CPU will determine the actual throughput of the CPU. It is also important to consider that not everyone who plays video games on their own computers will have the same CPU. Different CPUs have different processing potentials so when designing physics in games, it is easier to reduce the numerical accuracy of the algorithms to allow for a wider audience to play the game with stable performance.

Each core of a CPU can process a different thread meaning the processor can run up to four tasks simultaneously on different data sets. One core will be used for starting the game and will run the main thread of the application. Other cores can then be delegated tasks for calculating the physics of the game, processing the Artificial Intelligence of enemies or sending data to the GPU to render the scene among other things. Adding extra computations in the form of physics calculations increase the stress on the CPU, meaning the time taken to calculate each frame of the game will rise, resulting in lower framerates and poor overall performance of the game.

Multithreading on CPUs has major design challenges. One example would be deadlock – when two threads are executing concurrently, thread one requires a resource from thread two and vice versa, they both wait for each other to finish. Since both are waiting, neither thread can complete its task thus a deadlock occurs. To remedy this, the physics computations can be offloaded onto specialised hardware, freeing up the CPU to run other important tasks. The specialised hardware in question is the Graphics Processing Unit.

2.5 The Graphics Processing Unit

A GPU is designed to render and produce graphics to be displayed on a computer screen. Each pixel in a monitor can have its colour data processed by an individual core on the GPU. A typical GPU can have up to thousands of cores, each with the ability to perform floating point operations (such as vertex calculations found in games). In comparison to the 512 GFLOPS achieved on the aforementioned CPU, a modern GPU can have a theoretical peak compute of 5.63 Terra FLOPS (5630 GFLOPS) [20]. GPU architecture is highly parallel. Each core works independently from the others and there is very little intercommunication between threads, helping to alleviate deadlocks.

Due to the demand for better visual fidelity in games, computer graphics hardware becomes more powerful year upon year.

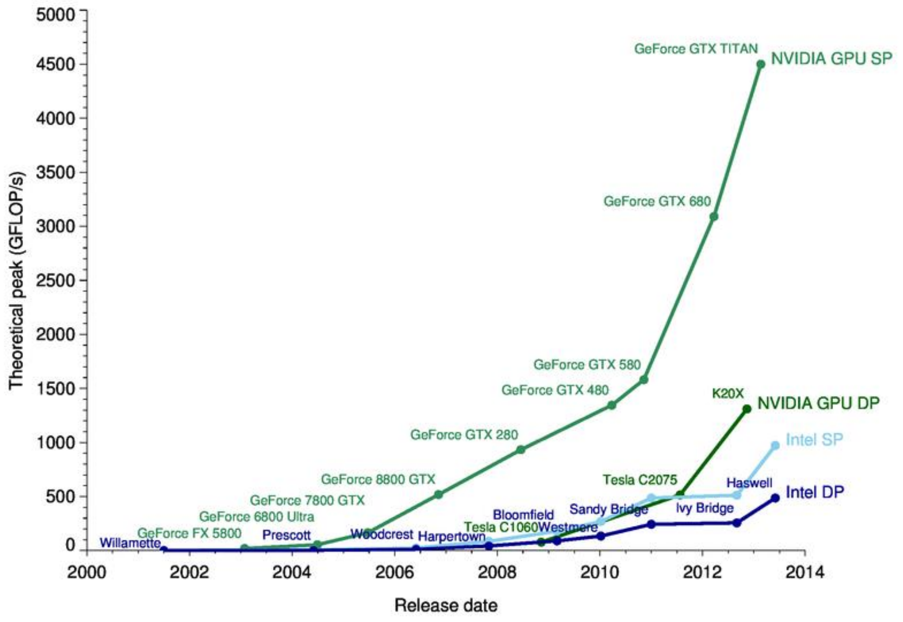


Figure 7: Compute performance of GPUs and CPUs [21].

Chart showing the theoretical peak GFLOP performances of NVIDIA GPUs vs Intel CPUs over time.

The discrepancy shown in figure 8 is since the GPU is designed for compute-intensive, parallel computations which is exactly what graphics rendering is. [2] explains this further by stating that the transistors in a GPU are devoted to data processing rather than traditional data flow or caching like a typical CPU.

Developers are also given the ability to program the hardware themselves to leverage even more processing power from GPUs thanks to specialised languages such as GLSL (OpenGL Shading Language). GLSL allows developers to write short programs called shaders which run exclusively on the GPU. Shaders can be used to enhance graphical scenes in games with techniques such as High Dynamic Range (HDR), Per-Pixel Lighting, Bump Mapping and so on [22]. These shaders are run during the middle stages of the graphics rendering pipeline for video games making them quite exclusive to graphical rendering calculations.

2.6 GPGPU Programming

As GPU technology became more powerful, developers realised that the GPU could be used for much more than just graphics rendering, it could be used to enhance parallel algorithms and programs. New Application Programming Interfaces (APIs) were released which allowed the GPU to do just that. The GPU could now be used to accelerate programs thanks to its massively parallel architecture [23]. The use of a GPU to enhance programs with specialised APIs is known as General Purpose Graphics Processing Unit Programming or GPGPU programming for short. The main APIs used for GPGPU programming are CUDA by NVIDIA and OpenCL by the Khronos Group [24]. As concluded in [24], both APIs can deliver equal performance under fair conditions, however since CUDA is native to NVIDIA graphics cards and the GPU used in this paper is an NVIDIA card, the API we will be using is CUDA.

A modern CUDA enabled GPU is split up into a series of streaming multiprocessors (SM). An SM handles the creation, memory management, scheduling and execution of threads. Threads are grouped in 32s, known as warps. The functions which run on the GPU are known as kernels. These kernels are split into blocks and then given to the GPU to process. The SM will assign blocks to different warps and each warp is managed by the warp scheduler. The GPU is best utilised when blocks can be divided into threads of size 32 as this is the size of each warp. Warps will perform one instruction at a time, so if all 32 threads in the warp agree on their execution path, full efficiency is achieved. Threads can diverge however. When this happens, the warp will process the threads which diverge serially (like on a CPU) and once the threads complete their tasks, they converge and continue along on the shared execution path [2].

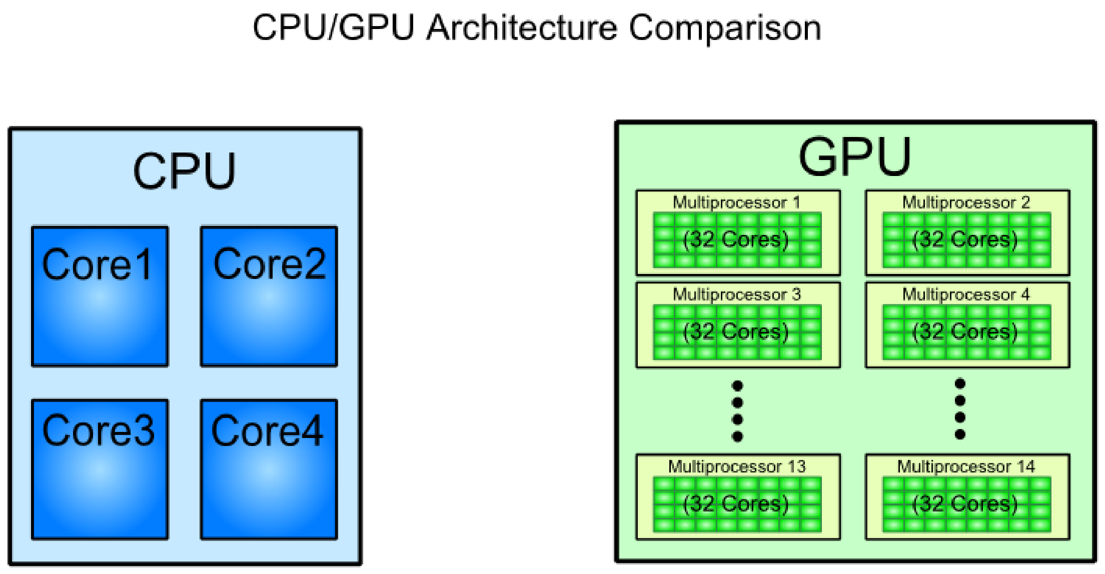


Figure 8: CPU vs GPU architecture [25].

The threads in warps are executed concurrently and independently. SMs are designed to execute hundreds of concurrent threads. GPUs are therefore suited to Single Instruction, Multiple Data (SIMD) operations. The actual architecture employed is known as Single Instruction, Multiple Thread (SIMT), which is similar to SIMD however a key difference is the vector organization of a SIMD approach exposes the SIMD width whereas SIMT instructions define how a single thread will execute. This means that only algorithms which can be performed in parallel will work best on a GPU.

2.7 GPU Accelerated Algorithms

Algorithms which are normally executed sequentially, but take little effort to run concurrently are known as embarrassingly parallel problems. Fluid dynamic algorithms can be shown to be embarrassingly parallel [26], since the calculations are done on individual grid cells or particles, independent from one another.

Existing simulations have been accelerated thanks to the power of GPGPU such as an N-Body Simulation [27]. The paper goes on to show that the CUDA implementation performed over 50 times faster than an optimised serial version. [27] also demonstrates how to write efficient GPGPU code to keep the cores of the GPU busy whilst also ensuring sequential memory access for faster reads. [28] addresses a SPH implementation on the GPU using CUDA. The parallel code allowed for hundreds of more particles to be computed in a shorter amount of time with the GPU thus providing greater detail in the simulations.

GPGPU programming is not limited to the realm of physics simulations however. As GPU compute advances, many researchers and developers are adopting its use in all fields of science and technology. For example, medical imaging, which has a crucial role in a wide variety of medical applications for diagnostics and treatment planning, is a computationally demanding process as the data sets being processed are extremely large and usually in 3 dimensions. [29] explains how GPU compute offers high memory bandwidth, the support for 32-bit floating point arithmetic and high throughput computing among others.

While this paper discusses the benefits of GPU compute, it is important to remember that GPU compute should only be used when algorithms meet two criteria: they are data parallel and they are throughput intensive. Data parallelism allows for many cores to operate on independent data points concurrently while throughput intensive means there is a lot of data to process. The data to be processed must be copied into the GPUs own memory before executing, thus it is more efficient to copy a large amount of data, process it concurrently, then copy it back to the CPU memory, otherwise time is wasted due to memory copying overhead [30]. It is up to the programmer to utilise the GPU properly and maximise throughput by programming kernels to correctly reflect the GPUs parallel architecture.

As mentioned previously, fluid dynamics can be parallelised and are highly throughput intensive, therefore in this paper, a fully interactive, real-time fluid simulation will be presented. Two versions will be shown, one which runs on the CPU and one which runs on the GPU. Their performances will be evaluated with respect to framerate (actual performance) and compute times (raw performance) to show that fluid dynamics can be utilised in video games to create ever more realistic worlds for gamers to explore. The parallel programming API used is CUDA by NVIDIA and the graphical API used to produce the on-screen image is OpenGL, an open source graphics library managed by the Khronos Group. The simulation will be an Eulerian grid based simulation with fluid moving in 2 dimensions. Further work can extend this simulation to 3 dimensions as will be discussed along with other features and applications.

3. Methodology

This chapter is an overview of the development process and implementation of the fluid solver. The methods, technologies, and hardware used will be reviewed along with their advantages/disadvantages justified in context. The structure for this chapter will reflect the development process taken during the project. As such, the CPU implementation will be discussed first then the GPU version will be shown.

3.1 CPU Implementation

3.1.1 Fluid approach

The first task was to determine an appropriate fluid representation. After carrying out research into the fields of Computational Fluid Dynamics and Fluid Animation, an Eulerian grid based approach was chosen as this method provides the most visually appealing simulations as opposed to particle methods [13]. The grid method does however run slower than particle methods but since this paper is comparing the performance of a CPU implementation vs a GPU implementation, the increase in speed gained from the particle method would be irrelevant.

The grid solver used in this project is based off the solver from [16]. This solver is only implemented on the CPU therefore modifications need to be made before the code can run on a GPU, however it serves as a good starting point for beginning the solver. The fluid being simulated is considered to have incompressible flow, which means the density is assumed to be constant even as pressure changes.

Stam notes that simulations in games must be convincing but fast at the same time. To achieve this, the algorithms used should not be too complex since they need to be able to work on a wide variety of machines and PC configurations. The algorithms presented in [26] meet these requirements by being custom tailored versions of the conventional Navier-Stokes equations. Another note to make is the solver proposed does not have strict bounds on its time step, meaning any value for the time step can be used – the algorithms will never “blow up” this solver is considered stable.

3.1.2 Navier-Stokes

To understand Navier-Stokes, the notion of a vector field must be explained. Vector fields map 2D or 3D space to vectors. Since a vector has both a direction it points to and a size, the vector for a given point in space can be thought to show the velocity of a fluid at said point. This makes it easier to determine the path of the fluid as it moves through the field [31].

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \mathbf{F}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa\nabla^2\rho + S$$

.

Figure 9: Navier-Stokes equations used in this solver [16].

The equations used in the solver by Stam. The top equation represents the change in velocity while the bottom equation refers to the change in density with respect to time.

As previously discussed, the top equation is the traditional Navier-Stokes equation for fluid flow in a condensed vector form. The bottom equation represents the density of the fluid. Since lighter objects simply get carried through a velocity field, it is computationally expensive to model every single particle. For this reason, a density value is used to represent the amount of fluid present in each cell of the grid. It should be evident that the two equations above look very similar. This is because the change in density over time can be modelled just as the change in velocity. However, a key difference is the density equation is linear whereas the velocity results in a non-linear problem. For this reason, the density solver was first implemented and the code was then reused with added functionality to solve for velocity.

3.1.3 The Euler Grid

To model fluids using computers, the continuous field must be split into a finite region of space. This region is represented as a 2-dimensional box with identical cells. The fluid is then sampled at each cell centre. Both the velocities and densities of fluid are defined at the centres and are assumed to be constant in each cell.

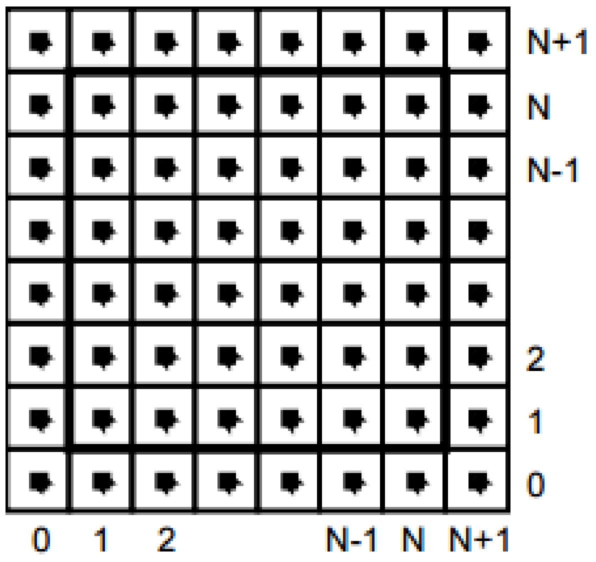


Figure 10: The grid representation used in this solver [16].

As depicted in figure 10, the grid is a box with identical cells. A layer of padding is used around the grid to allow for easier boundary control, with the actual fluid will be defined in cells from (1, 1) to (N, N). To form this grid, a 1 dimensional array is allocated for efficiency purposes whilst also being easier to program for multithreaded tasks. Three arrays are used for the density and velocity (in the case of velocity, three for each component of the vector: u and v). The size of the arrays is (N+2) * (N+2). Adding 2 to the grid length ensures we have the correct padding. There are three arrays for each field as we need one for the current state and one for the previous state of the fluid. The third array is used to allow concurrency when executing the routines.

3.1.4 The Solver

The solver follows a basic pattern: the scene starts with an initial state for both the density and velocity fields, then as time progresses, the fields will change per the environment. This means there will be real-time interaction generated by the user but also means that the fluid will update on its own once forces have been applied. The user will be able to apply forces and add density to the scene using the mouse.

The solver essentially calculates the density and velocity of fluid at specific intervals. The intervals can be given by the fixed variable dt. As previously mentioned, any value for dt can be used with this solver, the simulation will remain stable.

The density solver performs three functions which causes the change at each time step. This change is given by the three terms on the right-hand side of the second equation in figure 9. The first term states that the density will follow the velocity field. The second term says the fluid may diffuse at a certain rate kappa (κ). The final term states that density can increase due to sources. The solver proposed will solve the terms in reverse order. Thus, the density solver will start with an initial density field, then repeatedly solve for the three terms at each time step.

Adding sources is simply done by filling an array with values determined by the user's mouse position and mouse press. The longer the mouse button is pressed, the more source is added to the array and scene. This array is then passed to the solver.

The diffusion function looks at individual cells and spreads the density value across all direct neighbouring cells (top, bottom, left and right). Diffusion occurs at a rate diff when diff is greater than 0. The amount of fluid for a given cell will decrease due to density going into the 4 neighbour cells, but it will also increase due to the flow of density into the current cell. This results in a net difference of:

top_cell + bottom_cell + left_cell + right_cell – (4 * current_cell)

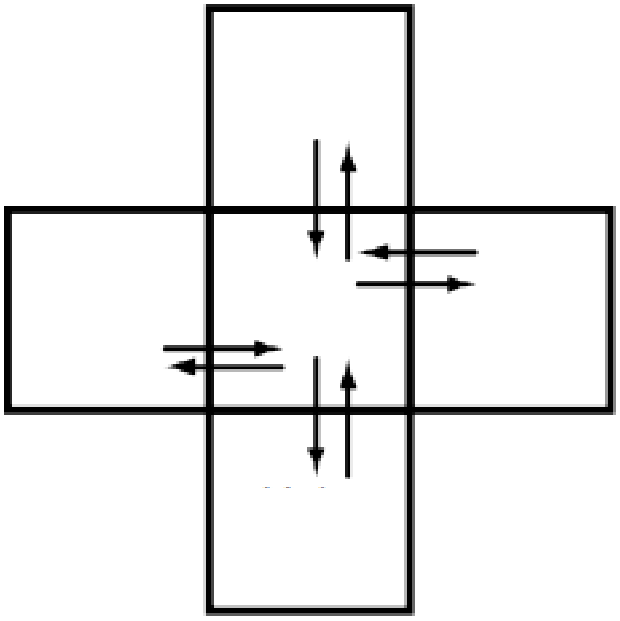


Figure 11: Depicts how the density flows through each cell via diffusion [16].

In [16], it is explained that simply adding the above formula to existing cells to find the new density would not work since for large diffusion rates the solver starts to oscillate, becomes negative then finally diverges becoming unstable. The stable method proposed in [16] calculates the new density, which when diffused backwards in time, yields the previous values already stored in the density array. This method builds up a sparse matrix which needs to be solved via iteration. Each iteration inverses the matrix. After a certain number of repetitions (in this case 20), a suitable approximation is found for the new density. In this instance, Gauss-Seidel relaxation is used for the iteration because of its simplicity.

The final part of the density solver is the advection step. Advection refers to the fluid moving though the velocity field. To correctly determine where the fluid should end up, the density can be modelled as particles which can then be traced through the velocity field. The idea here is similar to that of the diffusion step – we traverse backwards in time to find the particles which end up exactly at the current cell's centre during the current frame. The amount of density the particle carries from its source cell to the current one is found by linearly interpolating the source cell density with its four direct neighbour cell densities.

The advection step is as follows: Begin with two grids for density, one which contains all the values from the previous time step/frame and one which will contain all the new values for the current frame. For each cell in the new grid, trace the cell's centre position backwards in time through the velocity field. At the previous time step, use the old density grid values, linearly interpolate them with their neighbours, and then assign this new density value to the current grid cell in the current frame grid.

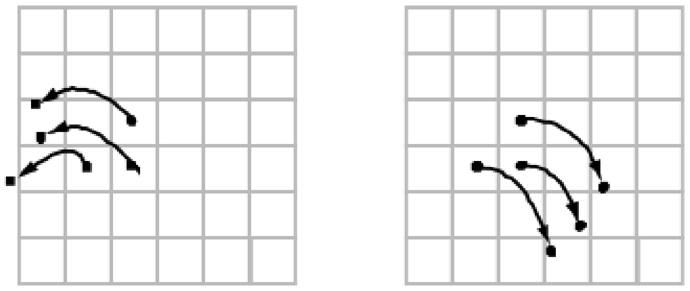


Figure 12: Particles in the grid [16].

The left grid represents how particles could be tracked forwards through time, however this method would depend on the velocity. The method used in practice is shown in the grid on the right. As you can see, the cell centre is traced backwards.

All of these steps combined form the density solver used in this simulation. For convenience they can all be called from a single routine each time step:

```
void dens_step(int N, float* x, float* x0, float* u, float* v, float diff,
float dt, float* dens_newArray)
{
       add_source(N, x, x0, dt);
       SWAP(x0, x);
       diffuse(N, x, x0, diff, dt, dens_newArray);
       SWAP(x0, x);
       advect(N, x, x0, u, v, dt);
}
```

In the parameters, N is the grid length, x and x0 represent the current and previous density grids, u and v are the current u and v values for velocity, diff is the diffusion rate and dt is the time step.

Note that dens_newArray is used to allow for parallel threads to execute this routine. Due to the nature of how this solver works, a third array is needed since new values for density are calculated using updated values of neighbouring cells. In a multithreaded GPU kernel, threads execute at the same time, therefore some threads would be using incorrect values for their neighbouring cells. The third array alleviates this problem making each update in the for loops thread independent. SWAP is a macro which swaps two array pointers.

Next, we move onto the velocity solver, which will evaluate the top equation in figure 9 at every time step. Just as the bottom equation described the changing density with respect to time using the three terms on the right, the same method can be applied to explain the change in velocity over time, it is caused by three factors. These factors are (in the order they appear in the equation): velocity will self-advect, velocity will have viscous diffusion and finally external forces can be applied to the velocity.

The last two factors are very similar to the add source and diffusion steps from the density solver however self-advection may not be so obvious. It simply means that the velocity field moves along itself. In other words, it implies the fluid does not lose volume as it is always pressing against itself (it has internal pressure).

The functions used for the density solver can be reused for the velocity solver which makes this implementation ideal for real-time applications. Not much code is needed for basic simulations. An extra step is required however to ensure the velocity field behaves correctly. This function is the projection function, which forces the fluid to be mass conserving (applying the continuity equation to the solver). When projection is applied to the velocity field, the fluid exhibits swirl-like patterns, known as vortices. Vorticity is the likelihood of a continuous material to rotate. Projection is done using what's known as Hodge Decomposition. Every velocity field is a combination of a mass conserving field (the incompressible, vortex filled field i.e. the one we would like to display) and a gradient field which represents the areas of steepest descent for a height function. Put simply, we can retrieve the mass conserving field by taking the gradient field away from the current velocity field.

To calculate the height field needed, a linear system known as a Poisson equation must be solved. Since this system is sparse like the diffuse system, we can reuse the Gauss-Seidel relaxation technique.

```
void vel_step(int N, float* u, float* v, float* u0, float* v0, float visc,
float dt, float* u_newArray, float* v_newArray)
{
        add_source(N, u, u0, dt);
        add_source(N, v, v0, dt);
        SWAP(u0, u);
        SWAP(v0, v);
        diffuse(N, u, u0, visc, dt, u_newArray);
        diffuse(N, v, v0, visc, dt, v_newArray);


        project(N, u, v, u0, v0, u_newArray);
        SWAP(u0, u);
        SWAP(v0, v);
        advect(N, u, u0, u0, v0, dt);
        advect(N, v, v0, u0, v0, dt);
        project(N, u, v, u0, v0, v_newArray);
}
```

This is the velocity step routine which calls the other functions. Apart from the additional calls to the project function, it is essentially the same as the density step. Function calls need to be doubled since the velocity is a two-component vector. Project is also called twice because the advect function for velocity works best when the velocity field is mass conserving.

The parameter list is practically the same as the density step one however in this case: u and u0 are the arrays for the current and previous horizontal component values for velocity, v and v0 are the current and previous vertical component values for velocity, visc is the fluid viscosity and u_newArray/v_newArray are the third arrays for the velocity to allow for multithreaded executions.

3.2 Graphical Environment

The CPU implementation was written using the C programming language. C is an imperative language which means programs are built up using several procedures or functions (also known as subroutines) which ultimately change the programs state.

Since the density and velocity steps are comprised of similar subroutines, it makes sense to write reusable procedures which can be grouped together to give the desired result, hence why C was chosen. It is also fairly easy to use for small applications such as the simulation in this paper, however extensions can be made by using an object-orientated language such as C++ to allow for objects to hold the simulation state, thus enabling fluid dynamics to be implemented into video games (where C++ is the most common programming language used).

Rendering the simulation is done using OpenGL, an open source API by the Khronos Group, and is the most used graphics library in the video games industry [32]. OpenGL is a cross-platform library which means the applications built using it will work on a range of machines and hardware configurations. Since almost all games are built using OpenGL, it serves as the logical choice to help render the fluid simulation.

Whilst OpenGL has a new modern version (commonly referred to as retained mode), this simulation uses an older version known as immediate mode. Immediate mode uses the fixed function pipeline and is not typically supported anymore (it has since been deprecated and replaced by the programmable graphics pipeline and shaders), however, for this small simulation, immediate mode serves as a quick and easy way to implement a 2-dimensional renderer. This simulation is also written in C which lends itself well to using immediate mode.

OpenGL itself is just a set of functions. It does not provide any way of opening and managing application windows or handling user generated input. To this extent, OpenGL is just a graphics library. This means another library is required to handle window creation and input processing. This simulation uses the GLUT library, which is a lightweight OpenGL window library with basic input handling capabilities.

GLUT is needed because opening an application window on a Windows machine is done differently to opening one on a Mac for example, so libraries like GLUT hide these details behind simple functions which when called, will initialise the window correctly on the target platform, helping portability. GLUT offers the ability to set callbacks for user generated input, which means when a key or mouse button is pressed, an event is triggered, causing a change in the program state. In the case of a mouse button, it would be adding force/source.

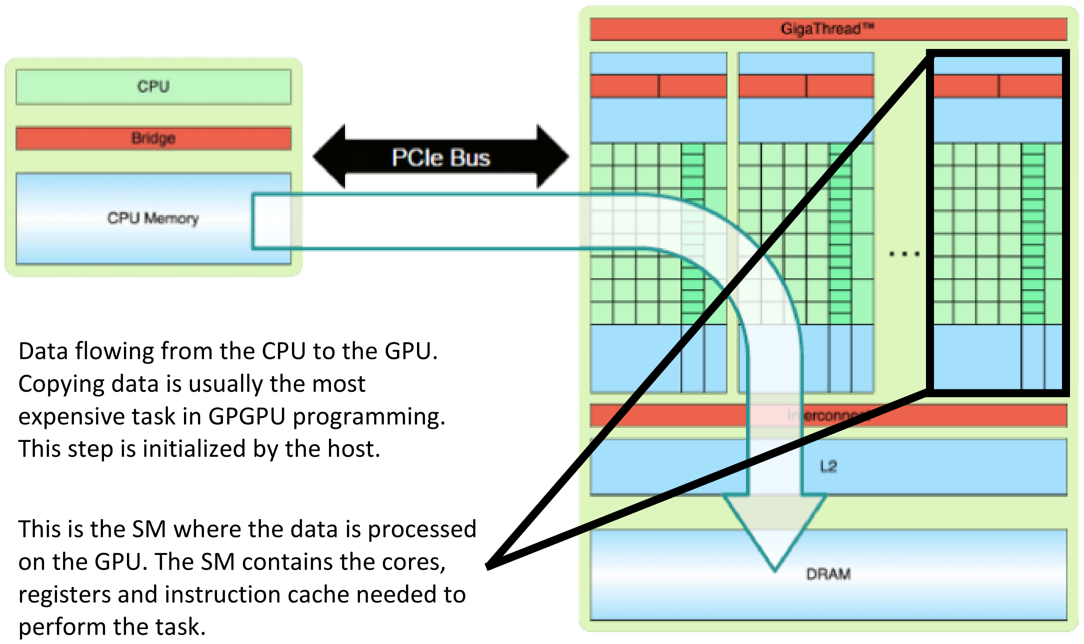
The code for rendering the simulation was provided by Jos Stam alongside his 2003 GDC paper [16]. The code can be found at [33].

3.3 GPU Implementation

3.3.1 GPU Architecture

With the CPU implementation complete, we now move onto the GPU version. To be able to program for the GPU, an understanding of how the architecture works alongside the CPU is needed. In a typical CPU program, code is executed sequentially (unless it is multithreaded, however in this case, threads are still executed one after the other before diverging). CPU programs use system memory. The GPU cannot access system memory, instead having its own on-board pool of memory. Data is copied from the CPU to the GPU, the GPU executes many threads concurrently, and then the data is (usually) copied back to the CPU.

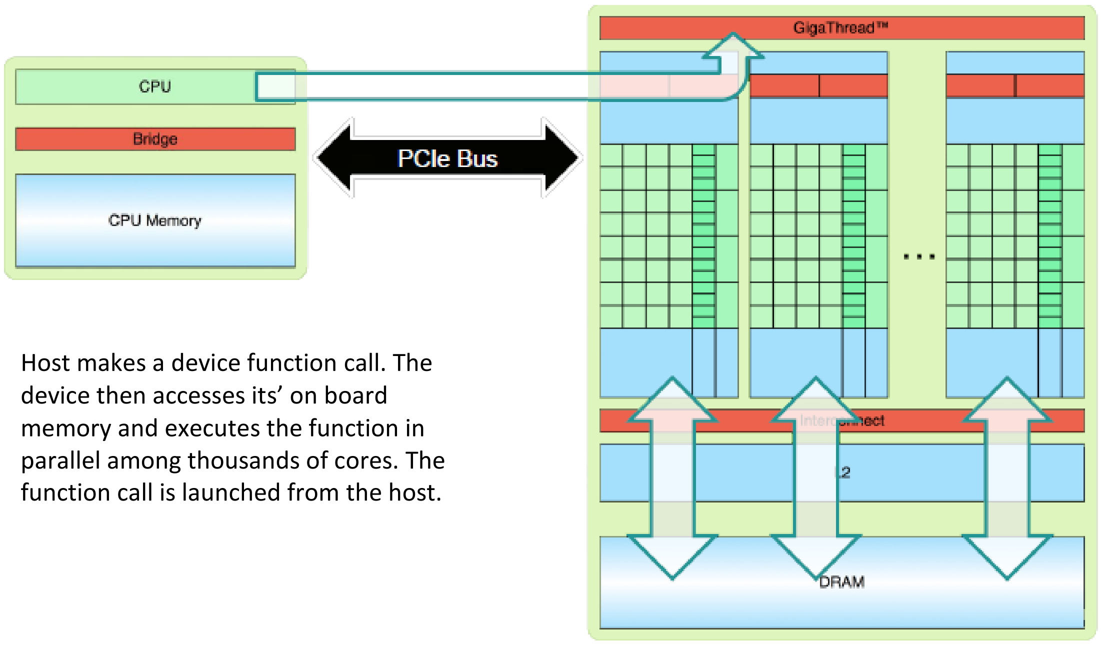
The CPU is known as the host and the GPU is known as the device in the GPGPU programming model. A typical program flow is host code executes serially, a call is made from the host to the device, the device code executes and runs in parallel, and then control is handed back to the host which continues running serially:



Data flowing from the CPU to the GPU. Copying data is usually the most expensive task in GPGPU programming. This step is initialized by the host.

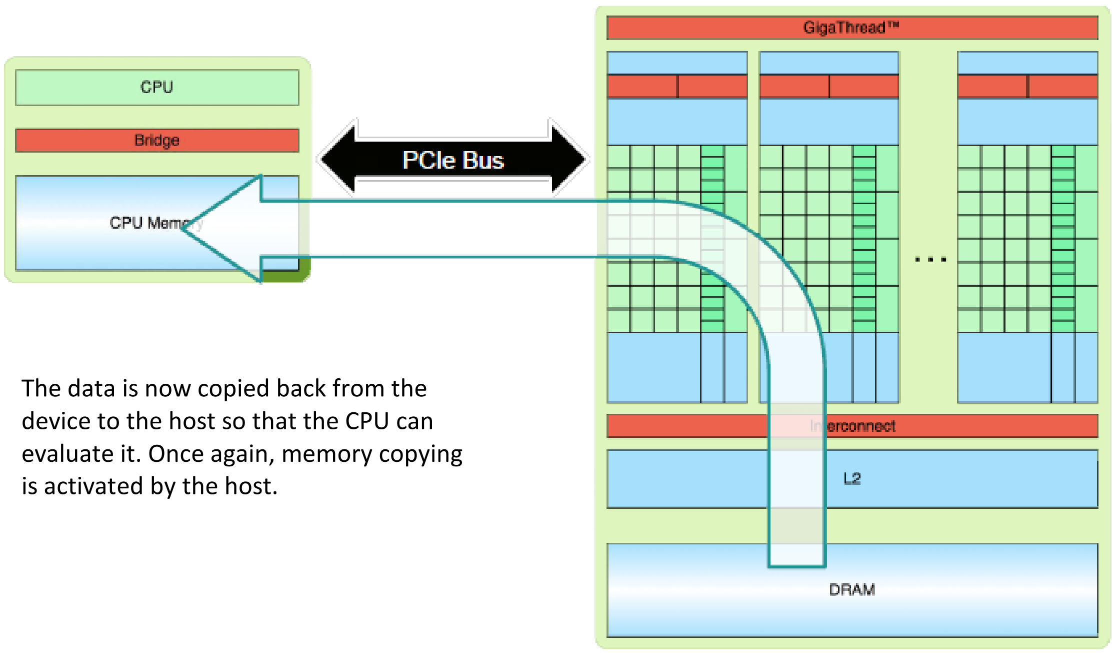
This is the SM where the data is processed on the GPU. The SM contains the cores, registers and instruction cache needed to perform the task.

Figure 13:Copying data to the GPU [34]



Host makes a device function call. The device then accesses its' on board memory and executes the function in parallel among thousands of cores. The function call is launched from the host.

Figure 14: Executing the kernel [34]



The data is now copied back from the device to the host so that the CPU can evaluate it. Once again, memory copying is activated by the host.

Figure 15: Copying the data back to the GPU [34]

3.3.2 CUDA

As previously mentioned, this GPU implementation uses the CUDA API by NVIDIA. CUDA has the advantage of being straightforward to program with, since it uses a C/C++ like language called CUDA C/C++. CUDA C/C++ is essentially the same as the conventional languages except it has some extra functionality and reserved keywords which allow code to be executed on the GPU. This makes it easy to port existing code onto the GPU (provided the CPU code is designed to be executed concurrently). CUDA C/C++ code must be written in files which end with .cu and must also be compiled using the CUDA compiler (NVCC). Any code which is not meant for the GPU is processed using a host C/C++ compiler.

The structure of a CUDA program is very similar to that of a regular C/C++ program, only a few extra steps are required before the GPU can be used to execute functions. To begin, the host first allocates the memory needed for the program in system memory using the malloc keyword. The host now has pointers to system memory but the device has no way of accessing this data. Therefore, the host must also allocate memory on the device using the CUDA runtime API function cudaMalloc. The host can manage all memory manipulation for both the CPU and GPU. Once this is done, the host now has pointers to all the relevant memory locations.

Once the memory has been allocated in the correct places, the host must initialise its memory. In the case of the simulation, all values for the host arrays are set to 0 at the beginning of the program to indicate no data in them. As the simulation executes, the host arrays will be filled with different values.

To initialise device arrays, the host arrays are copied into device memory and stored at the device pointer locations allocated earlier. This is done using the CUDA function cudaMemcpy, which takes as arguments the device side pointer where the memory will be copied to, the host pointer which the memory is being copied from, the amount of memory being copied and finally the direction the memory is being copied. In this case it is from the host to the device (indicated by the CUDA keyword cudaMemcpyHostToDevice. The device can use the device pointers to fetch the arrays from its own memory and process the data.

Now that the GPU has access to the arrays, it can begin processing the data. The host launches the kernel to be executed on the device by calling the device function. A typical kernel call looks like:

```
add_source_gpu<<<blocks, threads>>>(N, dens_device, dens_prev_device, dt);
```

The triple angle brackets represent the execution/kernel configuration. This tells the GPU how many threads should run the kernel in parallel. In CUDA, threads ae arranged in blocks which in turn are arranged in grids. The first parameter in the configuration says how many blocks in the grid should be used while the second parameter indicates how many threads per block should be used. The parameters in the parenthesis are the usual function call parameters. It is important to note that the arrays/pointers being passed into the kernel parameters are the device pointers and not the host pointers.

Kernels ae launched asynchronously which means once they have been called, control returns to the host immediately. To stop the host from continuing, the CUDA function cudaDeviceSynchronize is used to block the host from continuing. This allows the GPU to finish kernel execution before handing back the control to the CPU.

Once the GPU has finished running the kernel, the data must be copied back to the host. This is done using the cudaMemcpy function, only this time, the device and host pointers are swapped, and the direction is now device to host (cudaMemcpyDeviceToHost). At the end of the simulation, we must free all our memory using the free keyword. In a similar fashion, we free device memory using cudaFree with the device pointers as parameters.

For a function/kernel to run on the GPU, it must be prefixed with the CUDA keyword __global__ which tells the compiler that the function must be executed on the device. There is no need to specify that variables inside the kernel belong to the device because they are assumed to reside there by default.

GPU kernels are executed by many threads concurrently. This means the code in the kernel must be written in a way such that each thread performs the same task, but on different data elements. The typical way of processing arrays is with a for loop. In GPGPU programming, this would not work. Since every thread executes the same kernel, each thread would then execute the same for loop, accessing every element in the array.

To execute the kernel correctly, the function must be written in a way such each thread processes a different element. Since the warp scheduler handles which threads are used, we simply need a way for each thread to identify itself. CUDA provides a way to access the thread ID, the block ID and the grid dimension. Using all of these variables combined, each thread can be assigned an ID using: `int i = blockIdx.x*blockDim.x + threadIdx.x;`.

The block ID is the ith block in the grid being used. The block dimension is the number of threads per block. The thread ID is the thread number inside a block. All of these combined retrieve the correct thread number for the entire array. Now that each thread can identify itself, they can process the data in the array concurrently.

Below is an example of the CPU implementation and the GPU implementation for the add source method:

CPU VERSION

```
void add_source(int N, float *current_array, float *previous_array, float dt)
{
	int size = (N+2) * (N+2);
	int grid_length = N + 2;

	for (int i = 0; i < size; i++)
	{
		if (!(i % grid_length == 0) && !(i < grid_length) && !(i % grid_length ==
		(grid_length - 1)) && !(i >((grid_length - 1)*grid_length)))
		{
			current_array[i] = current_array[i] + dt*previous_array[i];
		}
	}
}
```

GPU VERSION

```
__global__ void add_source_gpu(int N, float *current_array, float *previous_array, float
dt)
{
	int i = blockIdx.x*blockDim.x + threadIdx.x;

	int grid_length = N + 2;

	int size = (N + 2)*(N + 2);


	if (i < size)
	{
		if (!(i % grid_length == 0) && !(i < grid_length) && !(i % grid_length ==
		(grid_length - 1)) && !(i >((grid_length - 1)*grid_length)))
		{
			current_array[i] = current_array[i] + dt*previous_array[i];
		}
	}
}
```

As you can see, the CPU version simply iterates over every single element in the array sequentially, then performs the add method. The inner if check excludes the padding cells from the function. The GPU version uses the calculated thread index discussed earlier. The index is checked against the size of the array. This ensures no element is accessed out of bounds. Then, once the padding cell check has been executed, the code for adding the source is identical to the CPU version. The GPU version contains no loop since, each thread will execute the same kernel. Every thread will process consecutive elements in the array in parallel thus the time taken to execute the function is reduced dramatically.

Just like in the CPU implementation, the GPU version executes a step in the solver, swaps array pointers, then executes the next step and so on. The swap is still done on by the host, so the data must be copied back before the swap occurs. Once the arrays are swapped, the data is copied to the device again and the next kernel is launched.

Since there are only a few extensions needed to port existing C code onto the GPU, kernels can be implemented one by one. This helps the development process since if a kernel does not work it is easier to debug what the issue is. This solver was developed using an iterative approach where by each step in the solver was added incrementally. Once the step worked the next function could be made. Each kernel was developed in the same manner as the CPU functions however the projection step had to be split into three separate kernels for the GPU since later parts of the function required updated values from earlier sections.

4. Testing and Results

4.1 Strategy and Metrics

To make the testing as fair as possible, the same hardware was used for both versions of the simulation. The CPU used was an Intel core i7-6700K quad core processor clocked at 4GHz while the GPU used was an NVIDIA GeForce GTX 980TI in the gaming configuration, clocked at 1152 MHz in base mode and 1241 MHz in boost mode. This GPU has 2816 CUDA cores.

Two performance metrics were used during testing: the framerate of the simulation and the time taken to perform the fluid update (which includes both the density and velocity steps). Time taken for the update measure raw compute performance as no other factors are considered here. This is a point-by-point comparison of the CPU compute time vs the GPU compute time. The GPU compute time also includes the memory copying overhead from host to device and vice versa since this is an integral part of GPGPU programming. If this was omitted, then the test would not be a fair comparison.

Framerate is the second performance metric, since raw compute performance is useless if the simulation does not work in real-time. This paper aims to show how fluid simulations can be incorporated into video games and as such, the simulation requires a high framerate to achieve a smooth experience. For the framerate metric, an FPS (frames per second) of 60 or above is considered an excellent framerate in the video games industry [35]. Framerates of 30 or above will be acceptable but this would be the lower end of the spectrum as 30 FPS does not provide the same level of real-time interactivity that 60+ FPS does. Below 30 FPS is considered inadequate for real-time applications.

4.2 Testing: Phase One

Testing was done in two phases. The first testing phase measured the compute time of each implementation. The grid length of the simulation is denoted as N. The program only solves for fluid density and velocity in the cells contained in the grid N^2, however the actual grid length is N + 2 since there is a layer padding around the edges. The actual grid length (N + 2) is one side of the box. This means that the total number of cells for a given grid length would be $(N + 2)^2$.

The length was varied and the results for the compute time were recorded in milliseconds. For ease of readability, N is the independent variable used. The solver automatically adds two to the grid length for the padding, so this does not need to be accounted for when choosing a length, however the results consider the total number of cells in the grid (which includes padding cells). The starting size for N was 32 (total cells including padding = 1,156), being incremented by 32 for each measurement, up to a size of 512 (total cells = 264,196). The results were as follows:

Grid Length (N)	CPU	GPU
0		
32	1.29	2.56
64	5.10	2.69
96	11.52	3.04
128	20.43	3.78
160	31.90	4.51
192	45.67	5.48
224	62.34	6.75
256	81.27	8.25
288	102.46	9.57
320	127.14	11.95
352	153.31	13.37
384	182.80	15.45
416	214.20	17.71
448	245.22	19.00
480	284.75	22.46
512	323.95	25.51

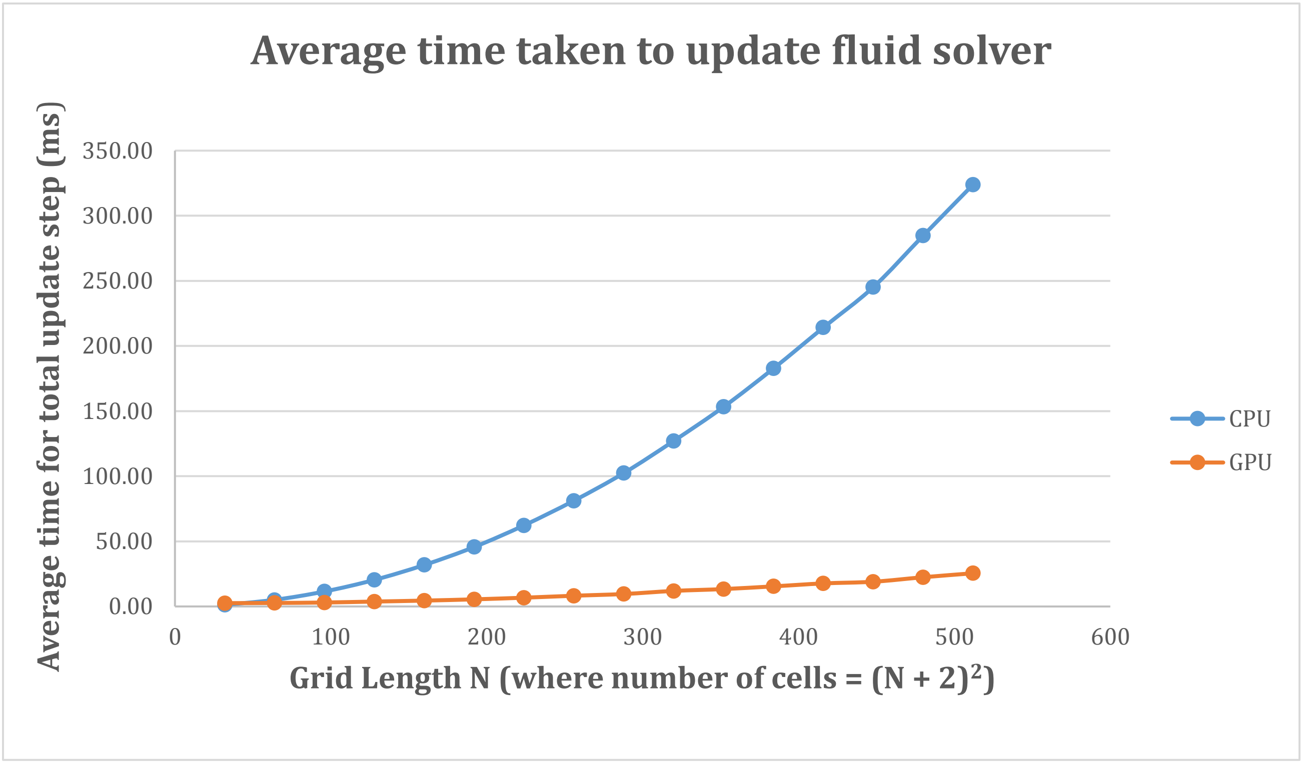


Figure 16: Graph one

As the graph indicates, the larger the grid size becomes, the longer it takes for the CPU to complete the fluid update. This is an exponential increase too which means the time taken increases sharply every time the grid size increases. The GPU however manages to stay at a consistently low time taken up to around 224 for N. After this point, the time taken for the GPU tom update the fluid starts to increase, but ever so slightly, in contrast to the CPU. As the grid size increases, the GPU time taken increases at a steady rate.

Next, the framerate of each version was measured. This measurement considers the time taken to render each frame. This gives a realistic example of how the simulation performs in real-time, which is an accurate representation of how the solver could work in a video game. Framerate is important because if the performance of the game is poor even with reasonable fluid update times, then the algorithm is not successful. The results were:

Grid Length (N)	CPU	GPU
0		
32	717.63	309.70
64	180.08	280.52
96	80.82	256.09
128	46.54	199.51
160	29.83	158.45
192	21.01	128.45
224	15.46	102.05
256	11.87	84.60
288	9.38	70.62
320	7.61	59.21
352	6.29	51.28
384	5.28	43.61
416	4.51	38.52
448	3.89	33.67
480	3.38	29.14
512	2.97	26.35

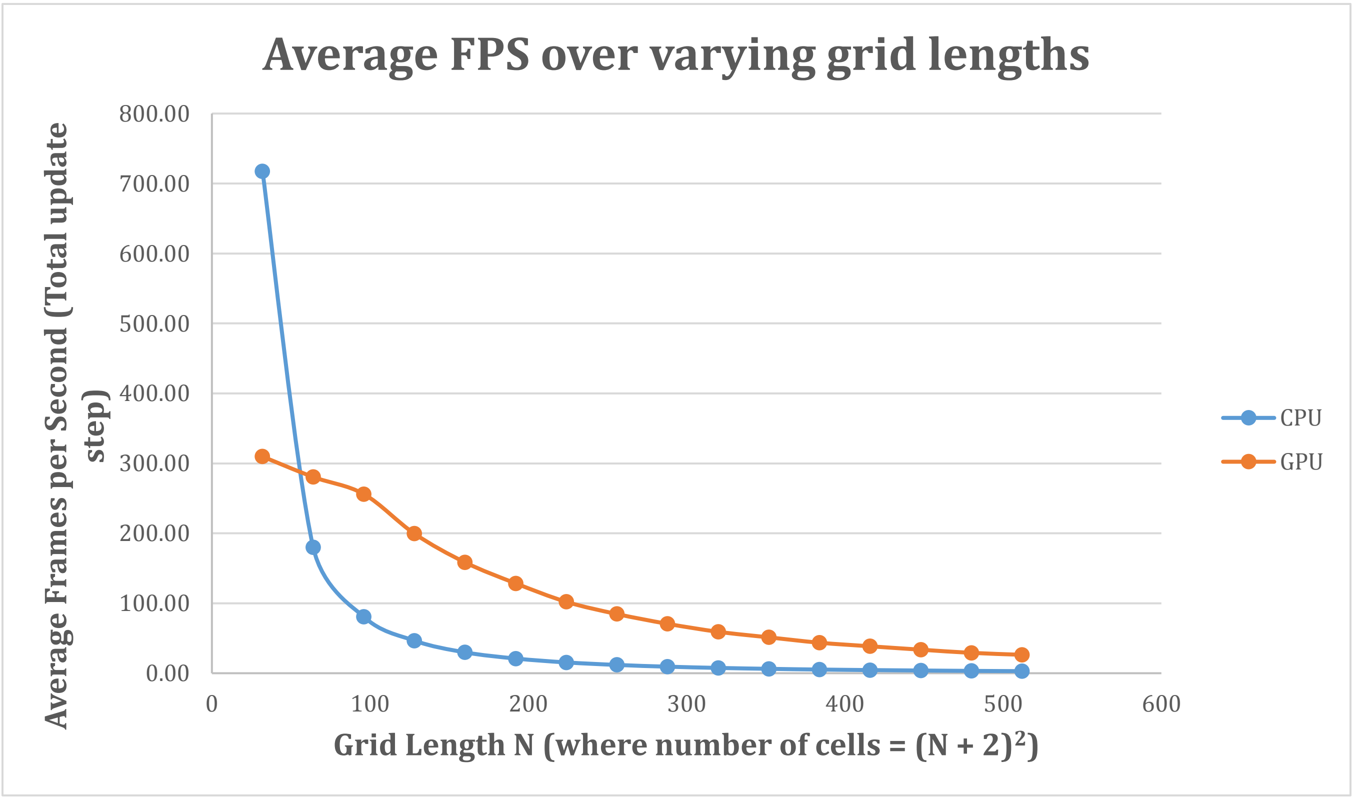


Figure 17: Graph two

Here we can see that the CPU performs significantly better than the GPU for smaller grid sizes. This advantage is quickly diminished as the size of the grid increases however with the CPU having an exponential drop in performance. The CPUs average FPS stays above 60 until around N = 100. At N = 160, the CPUs FPS drops below 30, at which point, interactivity with the simulation becomes worse. The GPU starts off by having a consistent framerate at around 300 - 260 FPS. Then at N = 96, the GPU takes a big hit in performance however this decreases in FPS remains above 60. The GPU managed to stay above 60 FPS until around N = 320. One note to make is the CPU has a much steeper drop off in performance and levels out much earlier in contrast to the GPU which has a small dip in FPS then a very steady decrease.

4.3 Testing: Phase Two

The graphs above makes it unclear what happens as both versions overlap. Therefore, another phase of testing was done with a much smaller range for the grid size. This provided a more in-depth look at what happens as the CPU and GPU version overlap at the beginning.

The starting grid size for phase two was 16, and the final size was 128, being incremented by 8 each time. The results were:

Grid Length (N)	CPU	GPU
16	0.34	2.51
24	0.73	2.53
32	1.29	2.56
40	2.02	2.81
48	2.87	2.69
56	3.90	2.93
64	5.10	2.81
72	6.52	2.77
80	8.16	3.07
88	9.63	3.59
96	11.52	2.93
104	13.55	3.25
112	15.64	3.35
120	17.93	3.40
128	20.39	3.78

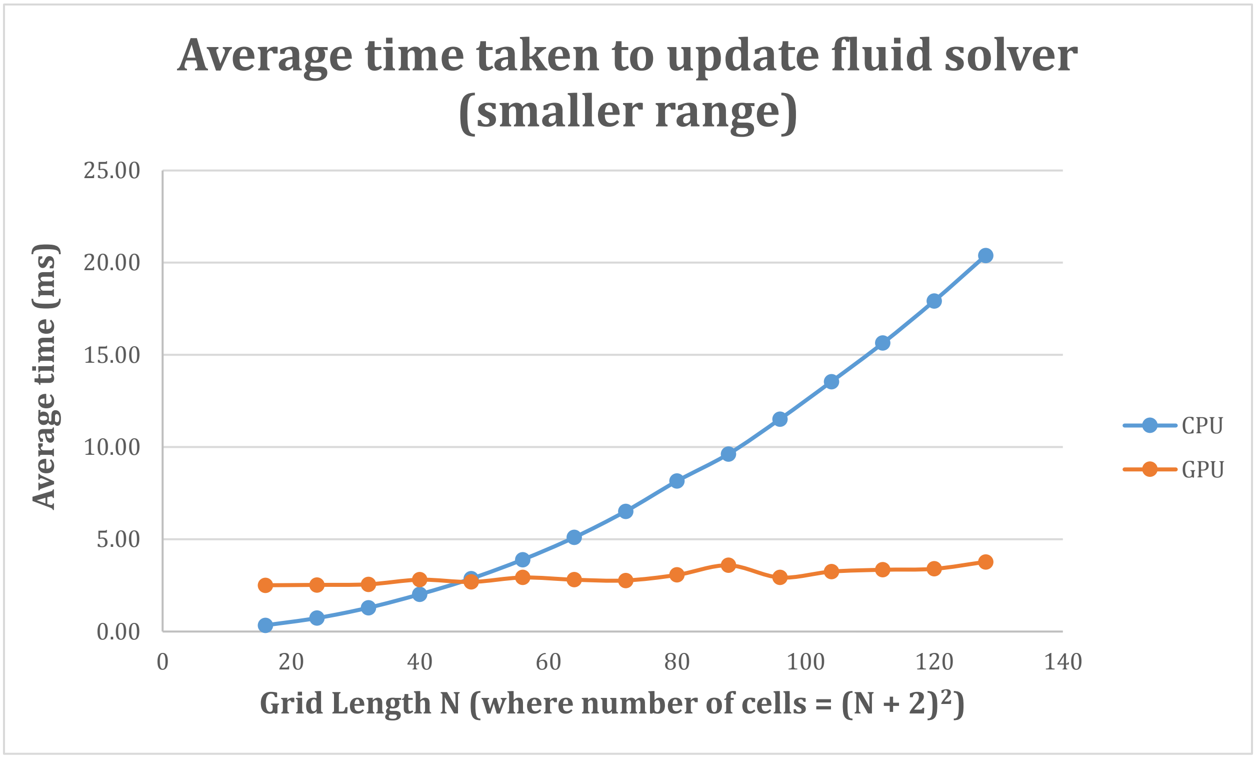


Figure 18: Graph three

A smaller range clearly shows how the performance between the CPU and GPU changes over time. The CPU clearly executes the fluid update much faster for a small period. This is until around N = 48, where the GPU starts to overtake the CPU. The GPU version takes a while to speed up since for small data sets, the speed gained from concurrency is diminished by the time taken to copy all the memory to and from the device. At N = 48, this discrepancy is removed as the CPU time starts to increase rapidly while the GPU time stays quite stagnate, only starting to increase at around N = 112.

Just as with the compute time, a second phase of testing for the framerate was done with a smaller domain. The results for the second FPS testing phase were:

Grid Length (N)	CPU	GPU
16	2367.21	331.89
24	1220.98	331.11
32	717.63	315.68
40	459.03	301.57
48	323.47	303.20
56	240.92	290.60
64	180.08	280.52
72	144.71	279.91
80	116.96	264.80
88	98.57	262.47
96	80.82	256.09
104	70.10	242.69
112	60.47	225.00
120	53.09	216.34
128	46.54	199.51

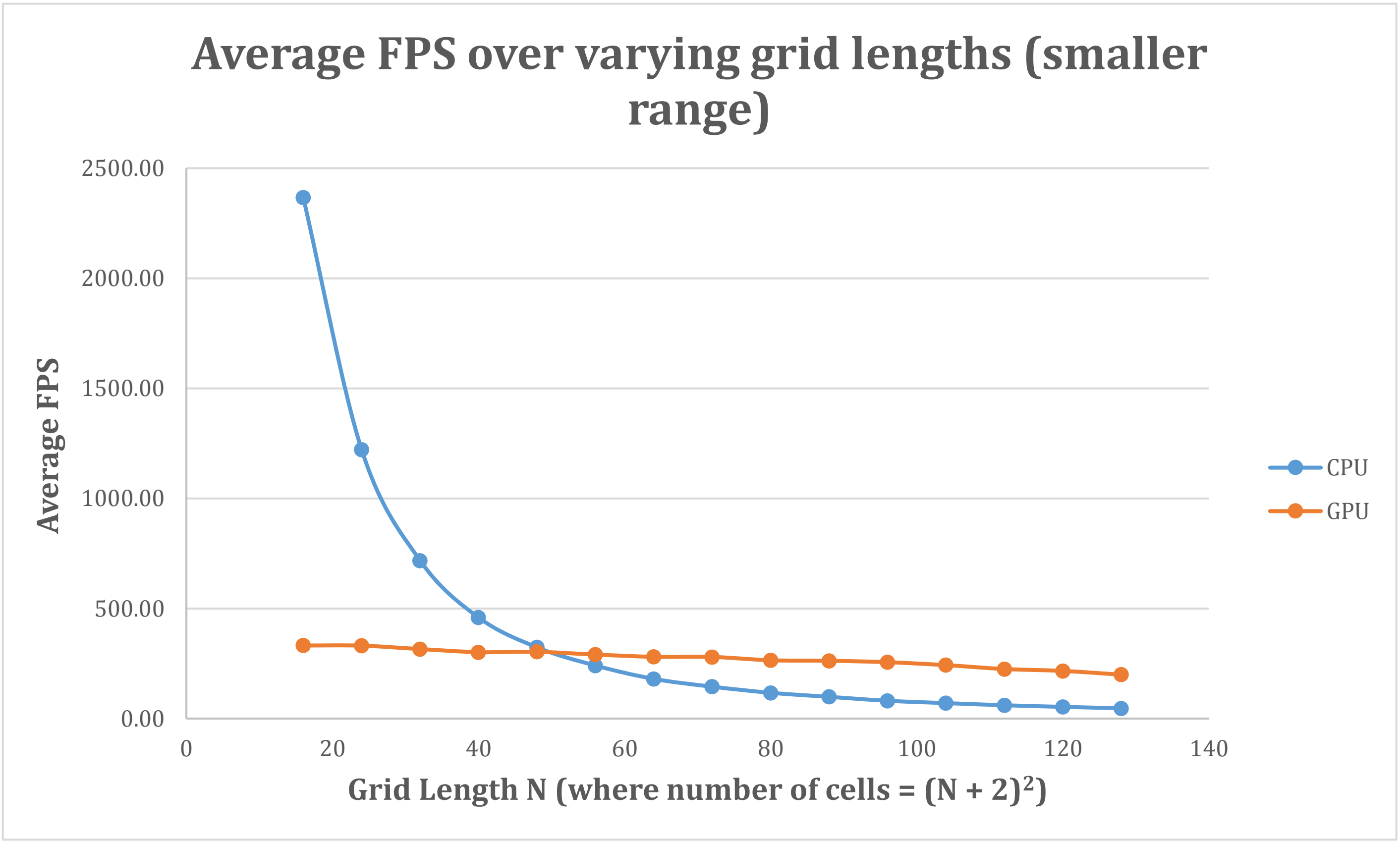


Figure 19: Graph four

As the chart indicates, the CPU has an incredible advantage over the GPU when dealing with small data sets. This is quickly lost though as the size of the grid increases. The GPU implementations FPS remains consistent since it takes the same time to process small data sets using the GPU up to a certain point. At around N = 72, the GPU starts to see a small decrease in performance but this is ever so slight.

5. Analysis and Evaluation

5.1 Computation Time

The results for the compute time show that for small data sets, the CPU out-performs the GPU by a considerable margin. One reason to explain this is the fact that the GPU must copy the data into its own memory, which takes time. For small data sets it is not worth the extra time taken to copy memory since the speed up gained from executing the kernel on the GPU is not significant enough. This point is further clarified by [30] which states that only algorithms which are throughput intensive work well on the GPU, which means the GPU needs to process a large amount of data to be efficient.

The CPU is designed to run very efficiently when performing sequential operations, therefore it makes sense on smaller grids where there is not as much data to operate on, that the CPU performs better. By comparison, the GPU will take the same time to process smaller data sets up to a certain size, since the amount of threads utilised on the GPU will be the same for all small data set sizes. The number of threads parameter in the kernel configuration is always 256, meaning if the total number of cells for a grid is small, any extra blocks will not be used when executing the kernel.

Graph three indicates that the GPU begins to overtake the CPU in terms of compute performance at around $N = 48$ (total number of cells $(N + 2)^2 = 2500$). This coincides with the results from graph one which show the GPU taking over the CPU performance between $N = 32$ and $N = 64$. Around this point, the number of cells in the simulation is almost equal to the number of cores present on the GPU. This could indicate that the GPU starts to utilise almost every core to process the fluid update thus enabling more efficient use of the GPU.

It is important to note that the speed increase gained from the GPU is only up to a point as there are a limited number of cores on a GPU. Beyond a certain data size, the GPU will increase in computation time just as the CPU does, albeit at a much slower rate. As shown in graph one, the CPU compute time increases at an exponential rate, meaning for very large data sets, the CPU would be an impractical choice for modelling real-time physics.

5.2 Framerate Analysis

The CPU version has a framerate of over 2000 frames per second for $N = 16$ whereas the GPU only manages just over 300 frames per second. The lower framerate on the GPU stays at just over 300 FPS for quite a while which could suggest the GPU limits how quickly it processes data for small domains. This also shows how ridiculously fast the

CPU is at processing small amounts of data, as a framerate of over 2000 is incredible (considering games only need to run at 60 FPS to be smooth, a small domain for the fluid size in a real video game would not be used or the FPS would be limited to ensure the GPU isn't wasting time rendering unnecessary frames).

As the grid size is increased, the CPU framerate declines rapidly. Similarly to the compute time of the simulation, the framerates of both versions begins to overlap at around N = 48, further suggesting the GPU optimises throughput when all cores on the chip are being utilised. Framerate is also directly correlated to the update time for the fluid solver, since the faster the solver can update the fluid, the faster the program can render the next frame. Both the compute time and framerate measurements agree that when the number of cells is roughly the same as the number of cores on the GPU, the GPU starts to perform better than the CPU.

The CPU implementation drops below the 30 FPS threshold at around N = 160 (where total cells in simulation is $(N + 2)^2 = 26{,}244$). A grid simulation of 160 cells by 160 cells produces pleasant results however this is at the low end of real-time acceptability (since any FPS below 30 is considered unacceptable). Should the fluid model in question require a larger domain, the CPU would not be able to cope with the increased computation. The GPU manages to stay above 30 FPS until N = 480 (where total cells = 232,324). This domain has an area nine times larger than the CPU limit, meaning the GPU would be able to handle wide, open fluid simulations with no problem. Any larger, and the GPU will start to diminish in visual fidelity as well.

5.3 Optimisations

Optimizations could have been made to the solver to increase performance. For the GPU solver, the pointer swapping is still done on the CPU. This means the data must be copied from the GPU to the CPU just to swap the pointers. Instead, the arrays could be swapped by the GPU, thus saving a lot of overhead, as the data would only need to be copied at the beginning and the end of the velocity and density steps. Extra care would be needed however when swapping arrays on the GPU as the correct pointers must be returned when copying to and from the host and device.

The solver implemented in this paper suffers from being slow due to the numerical accuracy required by the algorithm. This impedes both the CPU and GPU performances. Instead of an Eulerian grid approach, the solver could be developed using a particle method, which would increase the real-time performances of both implementations, at the cost of some visual fidelity. The particle method would also be able to simulate breaking waves, since, each particle is independent of one another.

OpenGL has a retained mode which is where the graphics libraries' data space stores all of the object information for the models to be rendered on screen. Currently, the solver uses immediate mode for rendering. This can slow performance since, the graphics card can only start rendering the fluid once it has received all of the vertex and colour information from the CPU. The CPU constantly sends this data across even if no change to the fluid has occurred, however a better alternative would be to send the data over once, and just update the model when position/velocity is updated. This would speed up rendering as unnecessary draw calls would be omitted.

Below are some screenshots of the CPU and GPU simulations running on the hardware mentioned previously.

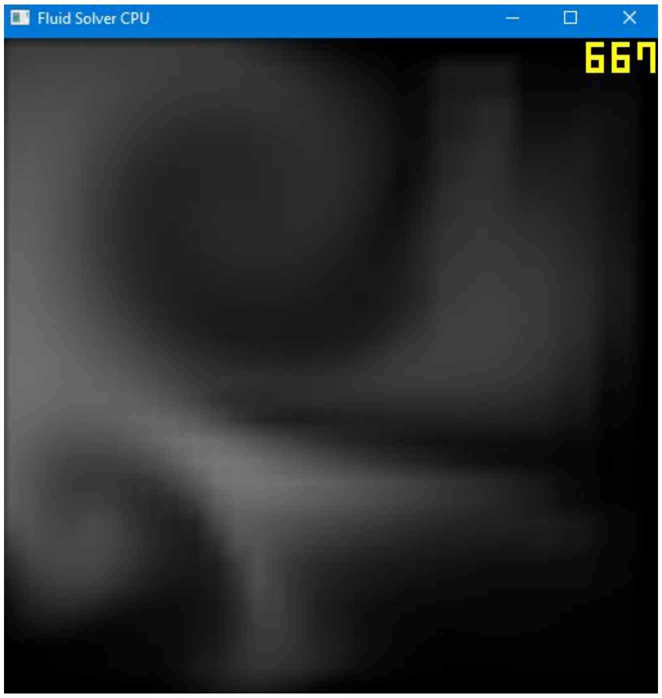


Figure 20: CPU simulation N = 32

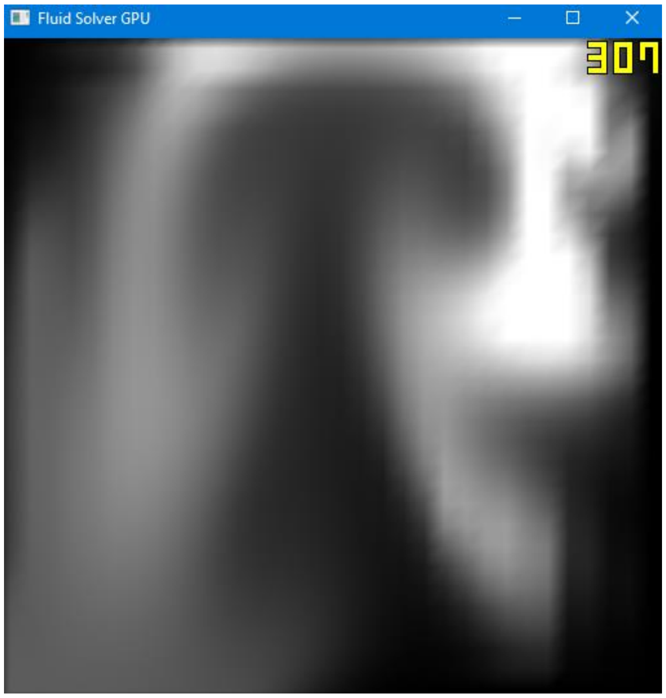


Figure 21: GPU simulation. N = 32.

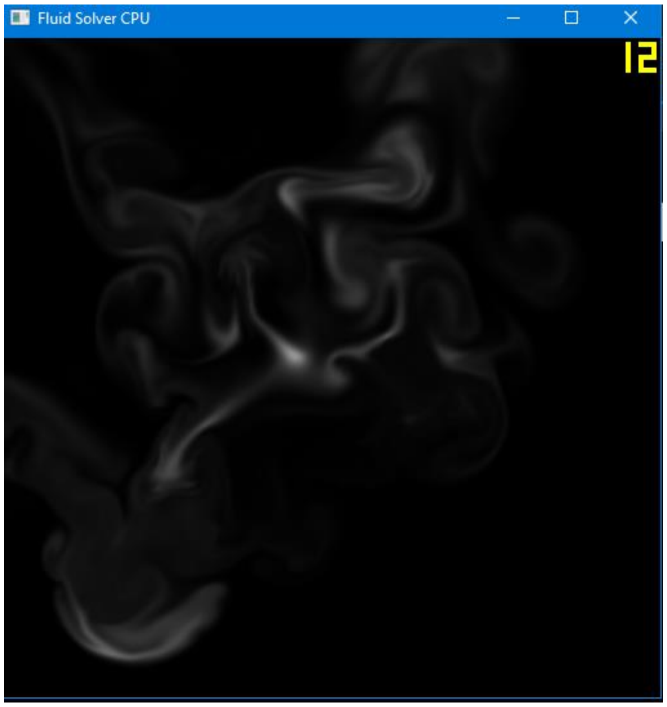


Figure 22: CPU simulation. N = 256.

The swirl like patterns are the result of the projection step, which forces the mass to be conserving.

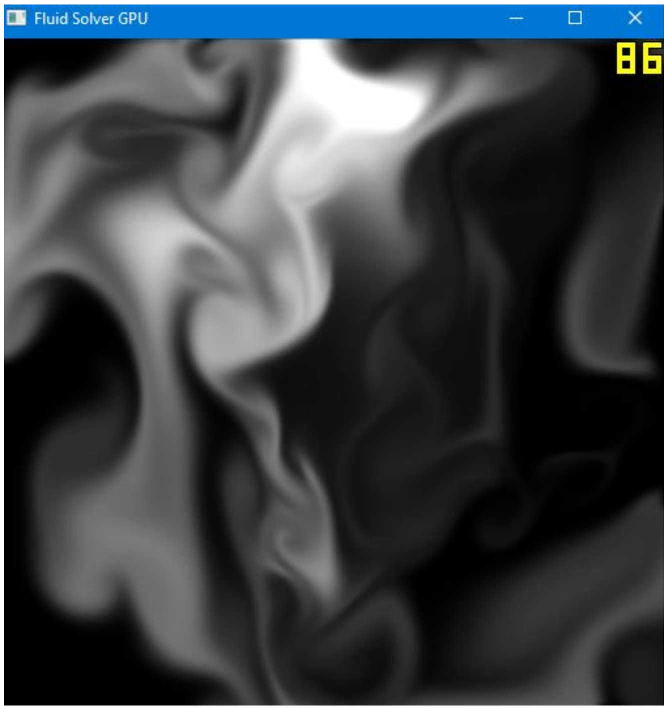


Figure 23: GPU simulation. N = 256.

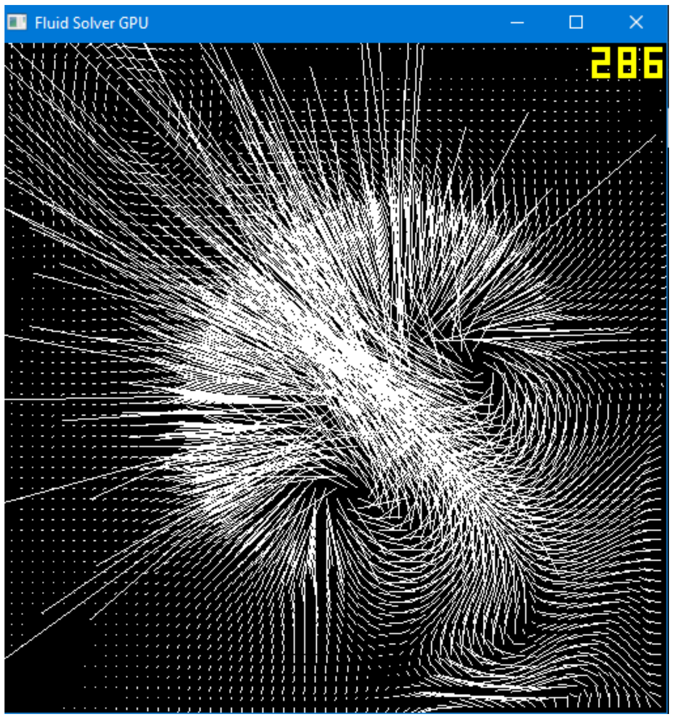


Figure 24: GPU simulation. N = 64.

This mode shows the velocity field. The lines indicate the direction and magnitude (size) of each component of the velocity for each cell.

6. Conclusion

6.1 Overall Summary

The original aim of this project was to Investigate the Accelerated Computation of Fluid Dynamics using GPGPU Programming. This paper has introduced the core concepts of fluid dynamics, real-time physics simulations for video games and GPGPU programming to show how they can be combined to produce visually pleasing and accurate worlds for players to explore. Through extensive research, it has been shown that video game physics can leverage the immense compute power of the GPU, a highly concurrent processing unit, to improve upon already existing fluid animation techniques employed in today's games.

This research is backed up by a complete solver demonstrated in this paper, along with detailed comparisons of the two implementations working. The results have indicated that while the CPU may be well suited for logical tasks and small domains, it pales in comparison to the GPU when it comes to parallel computation (provided the algorithms in question can be executed concurrently).

For the compute time, the maximum speed up achieved was around 12 times on the GPU vs the CPU. This is a substantial increase and proves that GPU compute is a viable option for employing more advanced physics models into video games, especially since the GPU version held up to around 30 FPS, even on the largest grid size. Game designers are no longer restricted when it comes to designing more realistic worlds for games.

6.2 Further Work

While the solver in this paper is in 2-dimensions, there are no restrictions to a 3-dimensional version. An extra array would need to be allocated for the third dimension, however the principles for the update step remain the same. A 3D solver would be more appropriate for the types of games which would employ such a fluid system (for example smoke coming off of a fire).

Another extension to the solver would be adding internal boundaries and rigid bodies. A rigid body is a physical object which cannot be deformed. Examples could be a box inside the solver which the fluid then flows past, or even allowing the user to generate their own internal objects from user generated input. This would be analogous to characters in video games moving through fluid systems or other physical models, since we are interested in the interaction between players and the world.

Neural Networks are systems which model the human brain. Artificial Intelligence in games can be programmed to mimic the users' behaviour or learn how the user plays the game to create more advanced tactics for victory using a neural network. Since the network must be trained via hundreds, to thousands of data points, independent from one another, certain neural network algorithms can be considered an embarrassingly parallel problem, perfect tasks for the GPU to handle [36]. The ability to leverage the massive compute power of the GPU has never been easier and thanks to modern APIs, existing parallel problems can be accelerated tenfold thanks to GPGPU Programming.

7. References

[1] V. Bertram, *Practical Ship Hydrodynamics*. Linacre House, Jordan Hill, Oxford: Butterworth-Heinemann, 2000.

[2] N. Corporation. (2007). *Programming Guide :: CUDA Toolkit Documentation*. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4gDDVEvcg

[3] C. Truesdell and W. Noll, *The Non-Linear Field Theories of Mechanics*, Third Edition ed.: Springer Science & Business Media, 2013.

[4] H. Fernando, M. Gharib, J. Kim, S. Pope, A. Smits, and H. Stone, "RESEARCH IN FLUID DYNAMICS: Meeting National Needs," Division of Fluid Dynamics of the American Physical Society2006.

[5] D. Foster, "Fluid Dynamics – Viscosity," 2015.

[6] A. Gibiansky. (2011). *Fluid Dynamics: The Navier-Stokes Equations*. Available: http://andrew.gibiansky.com/blog/physics/fluid-dynamics-the-navier-stokes-equations/

[7] COMSOL. (2017). *Different Flavors of the Navier-Stokes Equations*. Available: https://www.comsol.com/multiphysics/navier-stokes-equations

[8] D. L. Davidson, "The Role of Computational Fluid Dynamics in Process Industries," *Expanding Frontiers of Engineering,* vol. 32, 2008.

[9] D. o. M. Statistics. *What is a Numerical Method?*

[10] C. Grossmann, H.-G. Roos, and M. Stynes, *Numerical Treatment of Partial Differential Equations*: Springer Science & Business Media, 2007.

[11] J. P. Johnson, G. Iaccarino, K.-H. Chen, and B. Khalighi, "Simulations of High Reynolds Number Air Flow Over the NACA-0012 Airfoil Using the Immersed Boundary Method," 2014.

[12] J. Blazek, *Computational Fluid Dynamics: Principles and Applications*, 3 ed.: Butterworth-Heinemann, 2015.

[13] C. Braley and A. Sandu, "Fluid Simulation For Computer Graphics: A Tutorial in Grid Based and Particle Based Methods."

[14] T.-C. Sun. *Particle-Based Fluid Simulation*. Available: http://web.cse.ohio-state.edu/~wang.3602/courses/cse3541-2014-spring/proj_proposal/Ting-Chun_Sun/index.html

[15] D. M. J. Gourlay. (2014). *Fluid Simulation for Video Games (part 1)*. Available: https://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1

[16] J. Stam, "Real-Time Fluid Dynamics for Games," 2003.

[17] N. Foster and D. Metaxas, "Controlling Fluid Animation " 1997.

[18] B. Þ. Árnason, "Evolution of Physics in Video Games," 2008.

[19] Intel. (2014). *Intel® Core™ i7-4790K Processor*. Available: http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz

[20] R. Shrout. (2015). *The NVIDIA GeForce GTX 980 Ti 6GB Review*. Available: https://www.pcper.com/reviews/Graphics-Cards/NVIDIA-GeForce-GTX-980-Ti-6GB-Review-Matching-TITAN-X-650

[21] M. Galloy. (2013). *CPU vs GPU performance*. Available: http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html

[22] N. Productions™. (2014). *GLSL: An Introduction*. Available: http://nehe.gamedev.net/article/glsl_an_introduction/25007/
[23] NVIDIA. *CUDA Parallel Computing | What is CUDA? | NVIDIA UK*. Available: http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html
[24] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," *International Conference on Parallel Processing,* 2011.
[25] E2MATRIX. (2014). *CPU v/s GPU*. Available: http://www.e2matrix.com/blog/cpu-vs-gpu/
[26] W. D. Henshaw, "Solving Fluid Flow Problems on Moving and Adaptive Overlapping Grids," 2005.
[27] L. Nyland, M. Harris, and J. Prins, *Fast N-Body Simulation with CUDA*: Addison-Wesley Professional 2008.
[28] A. Hérault, G. Bilotta, and R. A. Dalrymple, "SPH on GPU with CUDA," *Journal of Hydraulic Research,* vol. 48, pp. 74-79, 2010.
[29] L. Shi, W. Liu, H. Zhang, Y. Xie, and D. Wang, "A survey of GPU-based medical image computing techniques," *Quantitative Imaging in Medicine and Surgery,* vol. 2, pp. 188–206, 2012.
[30] B. Jeong and G. Abram, "8 Things You Should Know About GPGPU Technology," in *Q&A with TACC Research Scientists*, ed. Texas Advanced Computing Center: The University of Texas at Austin.
[31] S. Dobek, "Fluid Dynamics and the Navier-Stokes Equation," pp. 1-5, 2012.
[32] K. Group. (2016). *OpenGL Overview*. Available: https://www.opengl.org/about/
[33] J. Stam. (2003). *Jos Stam: publications*. Available: http://www.dgp.toronto.edu/~stam/reality/Research/pub.html
[34] A. Tatourian. (2013). *NVIDIA GPU Architecture & CUDA Programming Environment*. Available: https://code.msdn.microsoft.com/windowsdesktop/NVIDIA-GPU-Architecture-45c11e6d
[35] S. Sarkar. (2014). *Why frame rate and resolution matter: A graphics primer*. Available: https://www.polygon.com/2014/6/5/5761780/frame-rate-resolution-graphics-primer-ps4-xbox-one
[36] B. Nemire. (2014). *CUDA Spotlight: GPU-Accelerated Deep Neural Networks*. Available: https://devblogs.nvidia.com/parallelforall/cuda-spotlight-gpu-accelerated-deep-neural-networks/