Computer Science BEng

School of Informatics
University of Edinburgh

Undergraduate Honours Project

Parallelisation and Parameter Tuning of Fluid Simulation on Heterogeneous Computing Devices

Author:
Olafs Vandans
s1139243 / B021719

Supervisors:
Dr. Taku Komura
Dr. Christophe Dubach

April 2, 2015

Abstract

Computational fluid dynamics is an intensive research area with a wide spectrum of applications in engineering, science and graphics. Reproductions of natural phenomena, such as water, fire, smoke and flowing substances, rely on computational models of fluids simulated at increasingly high levels of detail.

To satisfy the rising demand for simulation quality and speed, we need to harness the recent sprawl of parallelism in computing. This project attempts to do so by making a popular fluid simulation method run in real-time. In particular, Jos Stam's Eulerian fluid simulator was implemented and parallelised using OpenCL, and tested on a CPU and a GPU device. Parameters were automatically adjusted during runtime to ensure real-time performance. This had been an unexplored concept for fluid simulations.

The parallel program was benchmarked against an analogous sequential version. The results found that that a speed-up of 63x is readily achievable on a modern GPU. At the same time, OpenCL is still in active development and was a limiting factor in realising some optimisations. This report takes the reader through the construction of the program, design decisions and demonstration of the results.

Acknowledgments

Help from the following people was essential in ensuring the project's completion and quality. I would like to thank:

Taku Komura for providing regular consultations on the graphics and fluid dynamics part, as well as invaluable suggestions about the structure and organisation of this text.

Christophe Dubach for helping me navigate the depths of OpenCL and the GPU architecture peculiarities. A few bug fixes in the implementation are also due to Christophe.

Daniel Holden for leading me on the path to the right rendering solution. Daniel's expertise and the *Corange* game engine gave this project a much necessary foundation, the volume renderer used throughout debugging and live demonstrations.

Contents

1 Introduction

Some of the most stunning visual effects in games and movies are the computer simulated water, fire, smoke and other substances. These vivid recreations of natural phenomena make us truly appreciate the advances of mathematical modelling and computing power. Fluids, or materials that are viscous and that flow, can now be reproduced and studied outside of a laboratory setting - in the virtual world, with a complete control over the environment.

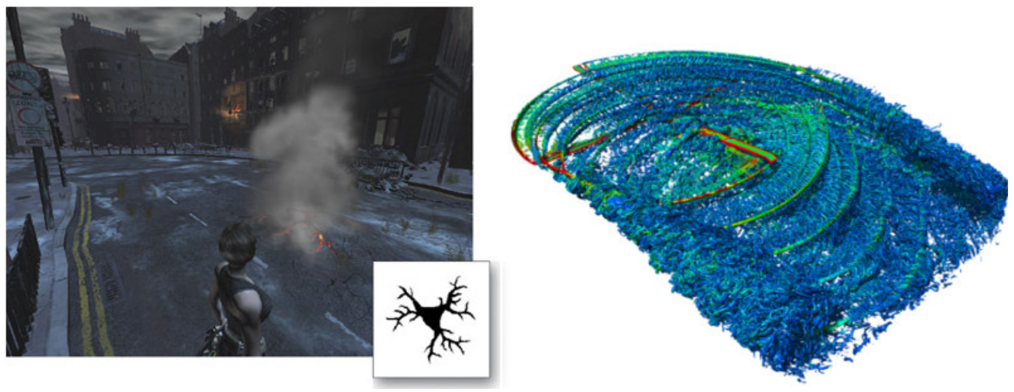

Figure 1: Smoke rising from a crack in a game (left)[14], NASA rotorcraft aerodynamic simulations (right)[22]

Special effects are but one of many applications of computationally generated fluids. They are an integral component of a variety of industrial and scientific disciplines. For instance, hydraulics relies on precise simulations of fluid properties to help design dams, bridges, car suspension systems, study river flow and erosion. Aeronautical engineering has used fluid simulations to study the air flow around the wing of an aeroplane and aerodynamic properties of cars. As of late, some connections have been drawn between fluid mechanics and quantum theory, where it could explain some particle phenomena thought to be purely probabilistic before.[11]

The applications are vast and require simulation at an appreciably high level of detail, which puts a considerable demand on computing resources. It is likely that a laboratory dealing with fluid simulations will use the services of a remote computing cluster. This is where heterogeneous computing becomes relevant - it refers to computation using more than one type of processor for a common task. It could be a combination of CPU and GPU, multiple GPUs, a cluster of different devices, or any other combination of specialised and general purpose processors. Full utilisation of system resources is the goal.

Similarly, parallelism in computing refers to performing a task on one type of processor(s), but split into multiple portions of it running concurrently. Recent general purpose CPUs come with several cores on-chip, able to run multiple threads, and now even lower-end systems often bundle GPUs with hundreds of cores.

Heterogeneous computing together with parallelism offers a promising way to reach acceptable performance for computationally simulated fluids on end-user systems, and will be the essence of this project.

1.1 Project aims

The general goal of my work was to explore ways to address the above performance limitations using state of the art parallelism on modern hardware. The focus is on real-time performance, serving a generic use case in games, simulations and visual content generation. The tangible output will be a program running a fluid simulation in real-time, maintaining a balance between speed and quality.

The basic fluid simulation was implemented according to Jos Stam's renowned paper *Real-Time Fluid Dynamics for Games*[2]. It is widely cited in computer graphics literature and describes what is currently the industry standard in fluid simulation techniques.

OpenCL was chosen for parallelisation - it is a modern portable library for heterogeneous computing and parallelisation. It claims to execute code across numerous platforms, including CPUs, GPUs, DSPs, FPGAs, and support for more is likely to come.

Specific performance goals can be achieved by setting the right simulation parameters. These will have to be adjusted dynamically during run-time, using the frame rate as feedback for decision making. While parallelisation of fluid simulations on general purpose GPUs is a documented practice, automatic tuning to achieve real-time performance has no mention in literature and will be explored in this work.

To summarise, the roadmap of the project is as follows:

- Implement Jos Stam's fluid solver in C++ and extend it to 3D
- Produce a renderer to visualise the results
- Parallelise and optimise the program using OpenCL
- Achieve real-time simulation using dynamic parameter tuning
- Analyse performance on CPU, GPU, possibly other devices

1.2 Report outline

The report will take the reader step-by-step through the process of apprehending the problem and implementing a solution. Wherever possible it will explain the challenges encountered and give reasoning behind the design decisions. The report and each section will have a top-down structure, starting with surface information first, then going down to the detail level.

Background information Section 2 will give an introduction to the main concepts used in this project: OpenCL programming/parallelisation and the mathematics of fluid simulations. It will start off with the theoretical foundation of fluid simulations, the Navier-Stokes equations and their application by Jos Stam. It will then continue with OpenCL programming models and optimisations to give an appreciation for what is to come.

Implementation Section 3 will take a practical direction and describe the actual program that was written, including the method's extension to 3D. Then the rendering component, a volume raycaster, will be introduced. The final subsections will give the compilation and running notes, and an overview of the testing environment.

Parallelisation Section 4 will connect the OpenCL working model introduced previously to the specifics of the implementation. Division into multiple kernels and various optimisations and their unfulfilled expectations will be detailed here, leading to a program that splits the simulation into parallel tasks applicable by OpenCL to any device.

Parameter Tuning Section 5 gives the motivation and design for the simple dynamic tuning component. A run through all considered tunable parameters will be given. The section will end with calibration of the change functions used to gauge the impact of each parameter.

Results and Analysis Section 6 will look back at the completed program and run different benchmarks to explain some anomalies encountered in the preceding section. The results of automatic performance management and impact from different parameters will be examined and presented in performance graphs, revealing how different devices cope with different parameter configurations. The success of the parallelisation will be evaluated in light of an analogous sequential implementation.

Conclusion Section 7 will critique the approaches used and look at alternatives that might have been more progressive. The state of the completed system will be evaluated and considered for future improvements. The report will end with comments on how appropriate the choice of technologies was and how fluid simulations and OpenCL parallelisation fit together in the general landscape of computing.

1.3 Related work

The aforementioned Jos Stam's paper on stable fluids [2] was the starting point of my implementation. It is an influential piece of work that first introduced fluids that are efficient and stable, i.e. not blowing up for certain parameters. Given that the paper's focus is on game development, there is enough room for adjustments in accuracy and performance, which is good for the project's aim.

A competing approach exists for simulating fluids that uses particle interactions, described in *Particle-Based Simulation of Fluids* [17]. This is a completely different paradigm from the one implemented here, but could be explored in the proceedings of this project. More about the two methods is given in section 2.1.

There is an existing conversion of Jos Stam's method to three dimensions, done by Mike Ash [8]. The 3D-extension in this work was done independently and only used Ash's code to verify the results. The end result differs for reasons explained in section 3.1.

The task of converting Stam's approach to OpenCL has already been done, though only in 2D, by Eric Blanchard [6]. This piece of work was a university assignment and does not provide any source code, but it gave the green light to splitting the whole simulation into multiple kernels, which at first seemed like an outrageous idea.

2 Background Information

This section will familiarise the reader with the major theoretical concepts used in the report: fluid simulations and parallelisation. Section 2.1 begins with explaining the mathematical model and characterisation of fluids. Next, it will turn to a detailed walkthrough of the different simulation stages in Jos Stam's solver in section 2.2, scratching the surface of the implementation details as well. Section 2.3 will give a brief overview of parallelisation using OpenCL, its execution and memory model, as well as a general note on optimisation.

2.1 Computational fluid dynamics

A widely used mathematical framework for simulating viscous flow is the Navier-Stokes equations, postulated by Claude-Louis Navier and George Gabriel Stokes in the 19th century. The derivation of these is too involved for this report, but is available online. [1] Essentially, they apply Newtonian physics principles on the smallest spatial units of the fluid to achieve viscosity and flow.

$$\begin{aligned}\frac{\delta \mathbf{u}}{\delta t} &= -(\mathbf{u} \times \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f} \\ \frac{\delta \rho}{\delta t} &= -(\mathbf{u} \times \nabla)\rho + \kappa \nabla^2 \rho + S\end{aligned} \tag{1}$$

The top and bottom equations govern the change of velocity ($\mathbf{u}$) and density (ρ) respectively. It depends on three factors - advection, diffusion and incoming forces - which will be explained shortly. The parameters ν and κ determine the viscosity, and the external inputs are $\mathbf{f}$ and S, used to add forces or material.

This model describes simple, commonly observed substances, such as water and smoke. A fluid simulated this way has the following characteristics:

- Incompressible - the fluid maintains constant volume all through
- Viscous (as opposed to inviscid) - the fluid resists deformation and shear stress
- Laminar (as opposed to turbulent) - the fluid is smooth and does not have any innate turbulences
- Newtonian - the fluid has a constant viscosity and continues to flow upon contact with objects

The fluid being incompressible is a simplification that makes simulation easier, but prevents such phenomena as waves (sound) or compression from pressure. This makes it less suitable for compressible, gaseous substances, but adequate for liquids. For the purposes of visual content generation, however, both ends can be met with acceptable quality.

Simulating any spatial process typically involves manipulating finite elements according to the same rule set. For fluids, there are two exclusive approaches differing in how they represent the fluid:

Lagrangian This method focuses on a single particle's path in the simulation space [17]. Computational cost depends on the amount of fluid and particle granularity, which makes it expensive for large amounts, but efficient for little fluid in a large volume. The simulation and rendering is rather complicated, compared to the Eulerian method. Lagrangian methods are preferable in scientific simulations because of their flexibility - multiple materials, as well as physical obstacles are easily incorporable.

Eulerian This is a more common method used in games and visual effects. It discretises the simulation space into a grid where cells contain varying amounts of fluid (densities). Processing is location-specific and does not discern physical particles. The volume boundaries do not change, which keeps the computational cost constant, but at the same time yields well to parallelisation on SIMD processors such as GPUs. An Eulerian fluid solver will be implemented in this work.

Some methods incorporate other fields, such as pressure and surface tension, or a viscosity field for mixing different fluids. These additional variables are of interest for engineering and scientific applications where every tiny detail is critical, but for this project the basics will suffice.

2.2 Jos Stam's Stable Fluid Solver

This will briefly run over the details of Jos Stam's paper, to be used as a reference for the implementation.

The simulation space is given as a two dimensional grid split into $N \times N$ cells. It is bounded by a single layer of special cells acting as a buffer to prevent leakage of fluid. Cells in this layer mirror the density of their in-bound neighbour and negate its velocity vector to counteract any flow past the border. The fluid itself comprises two components: a scalar field representing the density at a particular cell, and a vector field of velocities for each cell.

The visible part of the fluid is the density field, or a two dimensional array of floating point values showing how much of the substance is in the particular cell. This data is used to render the fluid. The velocity field governs how the material moves around and affects both the density and velocity itself.

2.2.1 State change

At any point in time, the system is in a state represented by the two fields. Advancement to the next state of the simulation takes place in a procedure (step function) that passes the volume through a pipeline of several manipulations. It is logically divided into a velocity and a density step, each doing different primitive operations on the respective components of the volume. These steps just apply the basic rules of the Navier-Stokes equations mentioned previously.

- **Velocity step**
  1. $addSource(\mathbf{v}, \mathbf{v_0})$
  2. $diffuse(\mathbf{v})$
  3. $project(\mathbf{v})$
  4. $advect(\mathbf{v})$
  5. $project(\mathbf{v})$
- **Density step**
  1. $addSource(d, d_0)$
  2. $diffuse(d)$
  3. $advect(d)$

This procedure is parameterised by a time difference parameter (dt) to denote how far in time the new state should be developed. Following is an in-depth explanation of the operations.

addSource This operation simply adds the passed volume data to the current simulation volume. It is used to feed fluid and forces into the volume at start and during the ongoing simulation. Data can just as well be removed (sunken) from certain areas by passing negative values. This operation corresponds to the $\mathbf{f}$ (velocity) and S (density) terms of the Navier-Stokes equations.

diffuse Diffusion can be observed when a droplet of ink is dropped in water - it spreads out slowly and overtakes a large area. On a computational grid this phenomenon can be reproduced as the density value in a cell flowing over into its nearby cells of contact. This exchange happens both ways.

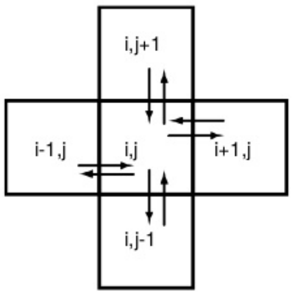

Figure 2: Diffusion exchanges between a cell and its neighbours [2]

It is possible to do this naively by just adding the right amounts to the appropriate cells, but this is unstable and can result in the simulation blowing up. To make it stable, it is possible to solve a linear system of equations to find what densities would lead to the current state, if diffused backwards. This is done using an iterative solver described later in section 2.2.3

Diffusion is represented by the $\nu\nabla^2\mathbf{u}$ (velocity) and $\kappa\nabla^2\rho$ (density) terms in the equation. The constants κ and ν control the amount exchanged per time step, or how viscous the fluid is.

advect Advection moves matter along a velocity field. In this case both the density field, as well as the velocity field itself are affected by advection. The process is similar to diffusion in that it goes backwards along the velocity vector to find the source point. It then linearly interpolates to determine how much to sample from each surrounding cell and transports the mass to the destination cell.

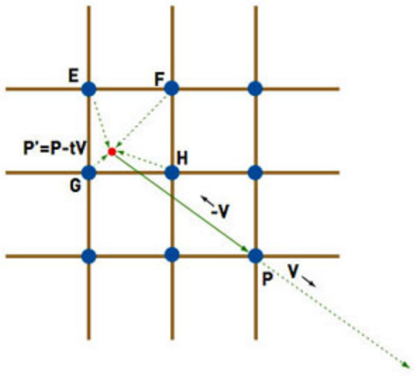

Figure 3: Advection - backtracking along the velocity vector. Blue dots represent cell centers, P receives weighted data from E, F, G, H. [10]

In the equations advection is $-(\mathbf{u}\times\nabla)\mathbf{u}$ and $-(\mathbf{u}\times\nabla)\rho$ for velocity and density respectively.

project The advection step disrupts the mass conserving property of the velocity field - certain areas in the volume may see a net loss or net gain, as the reverse velocity vector will point to arbitrary cells. Some would be missed, and some would be oversampled. To enforce the velocity advection step to be mass conserving, it has to go through a projection phase, which restores the original mass-conserving field. This is done using Hodge decomposition [9], a procedure from pure mathematics to decompose the velocity into a gradient field and an incompressible field.

Practically this means solving another linear system called a Poission equation, again performed using the Gauss-Seidel method.

2.2.2 Adding fluid and velocity

As mentioned above, modifications to the volume are made in the host application and passed to the addSource routine as a field of differences. This implies a memory transfer from CPU memory (RAM) to device memory, which for GPUs goes through the slow PCI bus, which is potentially a bottleneck. In the implementation, after the addSource steps happen, the v_0 and d_0 are used as temporary buffers and need to be cleaned in the end to prevent adding garbage data to the volume.

2.2.3 Gauss-Seidel relaxation

The quality of the simulation depends mainly of the quality of the solver. Gauss-Seidel relaxation is the suggested method, but others are possible as well. It is an iterative method whose accuracy gets better as more iterations are added. This is going to be become a tunable parameter later on.

Mike Ash's [8] solution has the solver separated as a function. It is good for reuse in the density and project phases, and probably clearer to read, but is an unnecessary complication for OpenCL parallelisation, which would require an extra kernel.

Other recommended solver methods include Conjugate Gradient, which is more demanding on space and uses a more complicated data structure, but could be faster. [12]

2.3 Parallelisation with OpenCL

OpenCL is a framework for expressing and running parallel computations on heterogeneous computing systems. It can be seen as an abstraction layer through which an application can access and run computations on any compatible hardware on the system, be it CPU, GPU, DSP or FPGA. This system frees the application programmer from relying on vendor-specific APIs and drivers, similar to how OpenGL provides an abstraction over graphics rendering hardware.

At the application level OpenCL consists of the API for enumerating and using compatible devices, as well as a variant of the C language for writing parallel programs. This language is called OpenCL C and is augmented with constructs for vectorisation and parallelisation.

2.3.1 Execution model

The OpenCL runtime can enumerate all compatible devices on a system and send computations to them for execution. This happens through a particular device's ICD loader (Invisible Client Driver). OpenCL takes care of all the scheduling and memory transfers.

A basic unit of computation is called a kernel. Kernels are written in the said OpenCL C language and invoked through the host application. They are stored as plain text source either in files or hard-coded in the application. When required, kernels are compiled by OpenCL during runtime, just like shaders are compiled by OpenGL.

The way to run data-parallel calculations is to specify a kernel over a multidimensional range (NDRange) of 1, 2 or 3 dimensions. This range has a further split into same-dimensional workgroups.

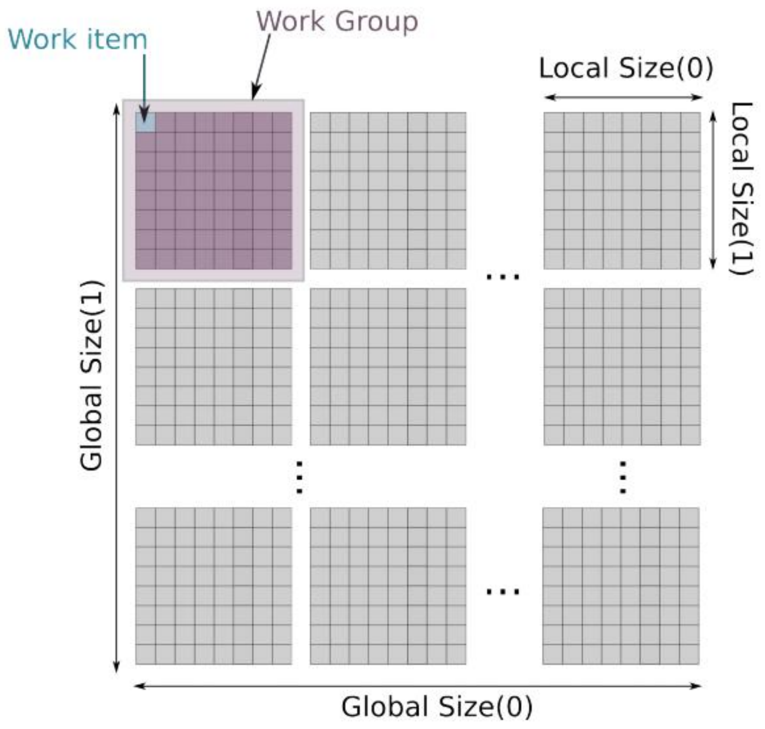

Figure 4: NDRange showing workgroups in 2 dimensions [20]

Another hardware-level grouping of processing elements is warps. Each warp shares an instruction scheduler, meaning that all threads in the warp execute the same instructions in lockstep. This will become important later, on the topic of branch elimination optimisations for the implementation.

As a processing element on a device receives the kernel, it can determine its index in the NDRange using *get_global_id()* and *get_local_id()*. This index is typically used to access the relevant portion of data from the memory. The processing element works with this data and leaves the result in device memory, to be reused later or transferred back to the main memory.

2.3.2 Memory model

GPUs have a specialised memory architecture with different levels of bandwidth and capacity. OpenCL is built to accommodate the GPU specifics by dividing its memory spaces into private, local, global and constant. Optimisations can exploit this division to increase performance. For example, local memory is faster than global, but is only shared among workers in the same workgroup - this could benefit computations that repeatedly access and exchange intermediate data.

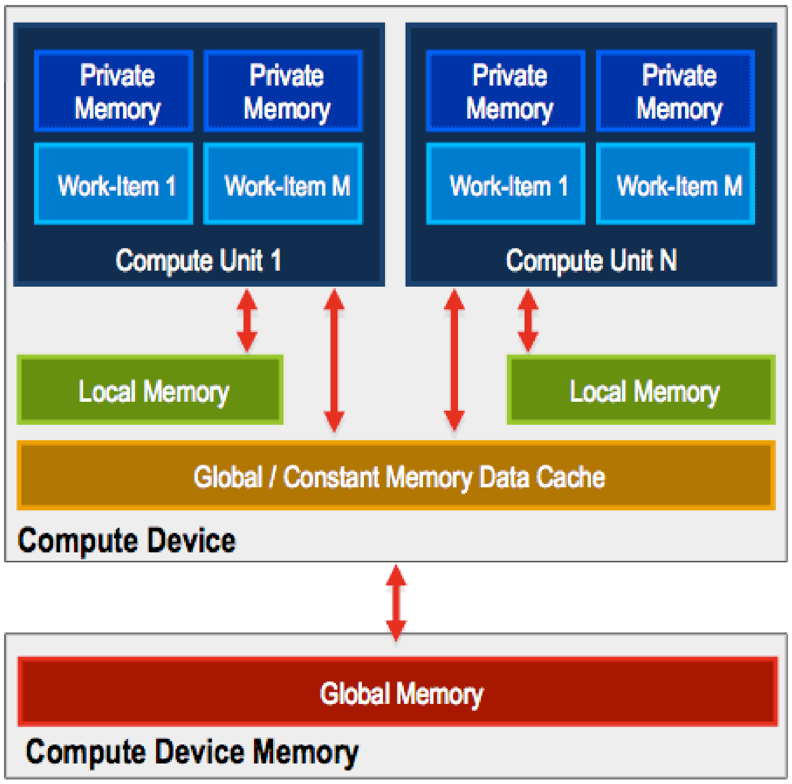

Figure 5: OpenCL memory structure [6]

2.3.3 Optimisation

Performance optimisations rely mostly on smart memory utilisation and keeping code linear, and are architecture dependent, so there is no one-size-fits-all method for generic OpenCL code. At present optimisations can only be tailored to a particular device. This project will attempt to optimise for GPUs.

3 Implementation

Here the basic parallelised program, sans the tuning component, will be described. The implementation of the paper was simple, given the pseudocode, but the extension to 3D (section 3.1) required some more work. Rendering is an important component for such a project, and will be described in section 3.2. In the end a brief overview of the implementation (3.3) with notes on compilation and running (3.4) will be given, and the hardware/software configuration this implementation was tested on in section 3.5.

3.1 Extending to 3D

The steps outlined in the paper [2] were given in two dimensions, so it was necessary to extend them to three dimensions.

Most of the basic fluid operations described in section 2.2 can be identified as performing an action on a cell's neighbours. A basic way to extend this is to consistently reproduce this action to two more neighbours on the nearby cells in the z-direction. This was done for each of the diffusion, advection, projection and addSource steps, and the field's 2D arrays were extended to 3 dimensions. The result is a 3D-capable Eulerian fluid simulator running sequentially at this point.

I believe there is an error in Mike Ash's [8] conversion - his given boundary setting function sets the corners and sides, but forgets to set the edges. This supposedly could cause leakages of the fluid through the edges. This fact makes the two conversions different and not verifiable against each other.

At this point, the 3D-extended fluid simulation runs sequentially (on 1 thread) under OpenCL.

3.2 Rendering

An important part of the project was to visualise the simulation. The goal was not to construct a high quality renderer for end-user applications, but a visual feedback tool for debugging and verifying everything runs properly. The renderer is a modular component that can replaced or turned off, not to impede with performance benchmarks later on.

3.2.1 Raycasting

A volume rendering solution was needed, and, following Daniel Holden's [4] advice, a separate component, a volume raycaster was written. It is loosely based on the GPUTracer [5] project. The presence of OpenCL in the implementation makes it convenient to parallelise this embarrassingly parallel process, so it resides in a separate kernel.

Raycasting sets up a virtual camera plane in 3D space and projects the 3D view on it. This is done by shooting a ray for each pixel and accumulating, in this case, the density values along equally spaced sampling points on the ray. The rays are meant to cross the simulation volume.

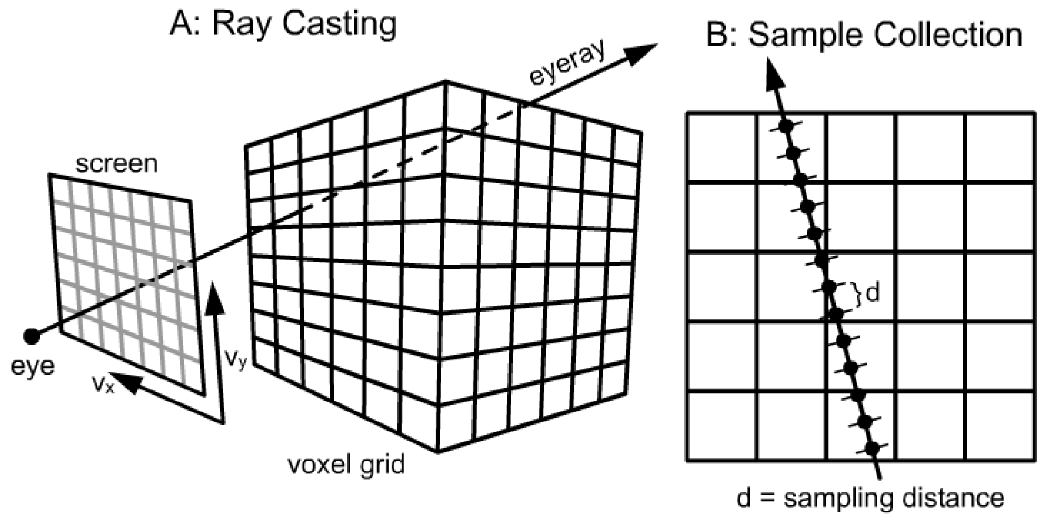

Figure 6: Raycasting process [21]

The result is a 2D texture containing a one-color density trace of the volume. The more density is encountered along the ray, the greater the color intensity of its pixel. Axes are drawn in simulation space, which makes them jaggy on the edges at low resolutions.

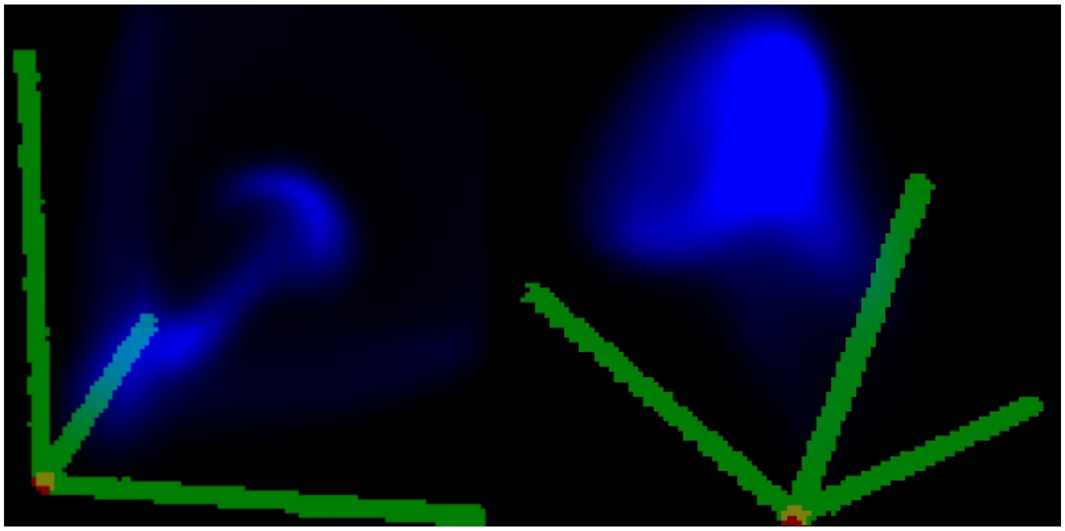

Figure 7: Fluid raycast in 3D space - different angles

At the end of the simulation pipeline, the generated texture is passed back to the CPU for rendering. In the current arrangement, the texture is rendered on screen using OpenGL, but could as well be exported to a file or analysed in memory. A further development to consider would be using OpenCL to OpenGL texture interoperability to do away with the passing back to CPU.

3.2.2 Camera

The position of the virtual plane from the previous section is determined by a virtual camera. The current implementation has a free-roaming camera that allows the user to capture the simulation volume from different angles. Perspective projection was used for added realism.

The camera can be moved around, and its pitch and yaw adjusted by the keyboard.

3.3 Finished implementation

The result is a fully functioning system with a modular architecture. The fluid simulation and rendering components can be replaced with ease. Material and velocity addition is limited to a single point and controlled by the keyboard, as is navigation around the virtual space. The timestep, velocity and density viscosity parameters are constant at 0.5, 0.001 and 0.0005 respectively, giving a vigorous fluid. The data cycle diagram of the application follows.

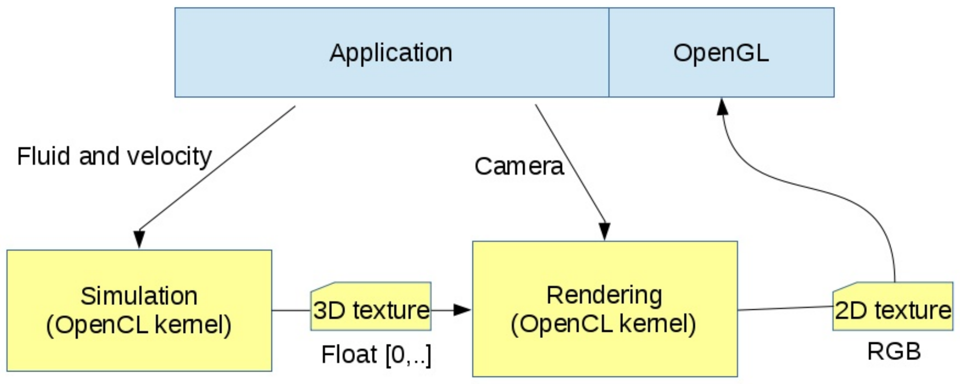

Figure 8: Components and structure of the resultant program

The source code and 64-bit Linux binaries are available at `http://olafs.eu/r/parautofluid.zip`

3.4 Compilation and execution notes

Compilation depends on the OpenCL SDK (software development kit) version present on the system. To compile on OpenCL 1.2, simply call `make`. To build for the older OpenCL 1.1, use the `-Dopencl11` switch, or call `make opencl11`.

The program can be run with the following command line arguments:

`-cpu`	force OpenCL to use the CPU
`-gpu`	force OpenCL to use the GPU
`-sequential`	run the simulation sequentially
`-fps <N>`	set the target frame rate to N
`-norender`	do not render (removes dependency on a graphical environment)
`-nolog`	do not output log files

The simulation will start running upon launch until the program is terminated. To control the program, use the keys:

`W, A, S, D`	navigate the camera around in XZ plane
`Q, E`	move the camera up and down along the Y axis
`arrow keys`	adjust the camera's pitch and yaw
`F, G`	add fluid (F) and velocity (G) to a predetermined point
`+,-`	forcefully change the resolution, overriding the tuning (explained later)
`R`	remove all fluid and velocity / reset simulation

3.5 Testing environment

Tests were performed to measure performance and calibrate the parameter change functions mentioned in section 5.2. The two testing environments will simply be referred as CPU and GPU, and are defined below.

CPU Intel Pentium Dual Core 2.2GHz with 4GB RAM, running on OpenCL 1.2 using the AMD APP SDK v2.9-1.

GPU GeForce GTX 590 with 3GB memory, running on OpenCL 1.1 using the nVidia SDK. Provided by the School of Informatics at The University of Edinburgh.

This choice was limited to what was available, but provides fairly varied environments to test the portability of OpenCL.

The implementation uses the OpenCL profiling functions to log start and end times of the kernels. For the sequential fraction benchmarks in section 6.4.2 the built-in profiling facility was used, toggled by the COMPUTE_PROFILE environment variable.

4 Parallelisation

By this stage the simulation is complete and will not change semantically. What follows are only performance-targeting changes that conserve the rules of the simulation. Now it has to go through the intricate process of parallelisation, i.e. turning a sequential program into one executable with multiple threads, detailed in sections 4.1 to 4.3. The latter will list the considered optimisations, only some of which could be realised.

4.1 Porting to OpenCL

The OpenCL C language is based on the C99 specification of C, making porting from a C-styled language a straightforward process. However, a bigger challenge is to make the code run in parallel, and run efficiently. This had to be done according to the OpenCL execution model, where a single kernel can be run with an N-dimensional range of threads.

4.2 Multidimensional kernels

The individual functions of the simulation pipeline have different runtime complexities stemming from the dimensionality of data the computation addresses. In addition, several of the 3D-natured functions call *setBound*, which is 2D in nature, making parallelisation nontrivial, because a single OpenCL kernel can only be run with a constant NDRange. To remedy this, the simulation step was differentiated into 7 kernels, each performing an isolated N-dimensional computation.

kernel	runtime	used in	reads	writes	calls
addSource	$O(n^3)$	density, velocity step	n^3	n^3	3
diffuse	$O(Sn^3)$	density, velocity step	$7Sn^3$	Sn^3	4
advect	$O(n^3)$	density, velocity step	$11n^3$	n^3	4
project1	$O(n^3)$	velocity step	$6n^3$	$2n^3$	2
project2	$O(Sn^3)$	velocity step	$7Sn^3$	Sn^3	2
project3	$O(n^3)$	velocity step	$9n^3$	$3n^3$	2
setBound	$O(n^2)$	advect, diffuse, project operations	$6n^2$	$6n^2$	$6S + 14$

- S is the number of iterative steps in solving linear equations with Gauss-Seidel.
- n is the resolution (side length of the cube)

Some kernels are called S times externally, but this was still factored in the runtime for clarity. The *reads* and *writes* columns denote how many items (4-byte floats) are transported from the global memory to the kernel's private memory and back. The *calls* column denotes how many times the kernel is called in computing a single step. These data will be used in the performance analysis in section 6.4.

Splitting into multiple kernels is also the approach taken in Blanchard [6]. Compared to the body of computations that needs to be done, kernel creation and scheduling overhead is thought to be insignificant and gives great flexibility. This is also good in that there will be no cross-device memory transfers between kernel calls, as all the memory stays on the device and is reused.

4.3 Optimisations

The multi-dimensional OpenCL kernels split the NDRange into smaller portions of data to be processed in their own thread. This is basic parallelism, and is simply achieved by bringing the bodies of (nested) loops to the top level and using the provided thread *id*s as indices. However, this does not give any guarantees about efficiency, and there are yet more optimizations available for maxing out performance.

A comprehensive guide to OpenCL optimisations that I tried to follow is given by Alan Gray from EPCC [13].

4.3.1 Memory coalescing

GPUs can take advantage of coalesced memory [7] whereby a single memory access causes several consecutive memory words to be brought along with it to the private memory. An application can access this newly available data much faster and increase performance. The addSource kernel does a simple buffer copy, but a single invocation works on 4 sequential elements. This is expected to reduce memory access times by a factor of 4.

Other kernels apart from addSource work on spatial data and access values in all directions from a single voxel. These accesses break coalescing, as the elements could be far away in the sequential memory. The next optimization would mitigate this problem.

4.3.2 Spatial coalescing with Image2D/3D

OpenCL provides higher level Image3D and Image2D objects that achieve spatial coalescing by fetching the whole neighborhood of a cell to the private memory. This makes them highly optimized for image processing and tasks of a spatial nature. However, there are some tedious limitations that need to be considered:

- An Image* object is either read-only or write-only
- Direct indexing is replaced by function calls to read/write data

While the latter only takes away the clean index notation to access elements, the former actually forces the programmer to duplicate function arguments for read-write variables. The fluid simulator implementation used buffers for both reading and writing extensively.

One experiment was to convert just the diffuse kernel from using buffer objects to Image3D, keeping other kernels the same. Once this was properly done, the performance had degraded by more than a half of the original frame-rate. The slowdown likely came from one or both of the following factors:

- Duplication of Image3D parameters into read and write versions defeats some optimisations.
- The other kernels still use buffer objects, forcing a slow memory synchronisation event between the image and the buffers.

To counteract the first, we have to wait for OpenCL 2.0, which promises read-write access to image objects. In fact many OpenCL 2.0 features would have been really helpful for this project.

To address the second point, I would have to convert all the other kernels to use Image3D objects. Unfortunately I hit a wall when trying this, due to the fact that the GPU system ran OpenCL 1.1, where 3D image writes are unsupported, whilst they are available through an AMD extension in v1.2. This would force me to use buffers for writing and Image3D for reading. Given this, there was no point going further with this and the optimisation was dropped.

4.3.3 Linearisation

Code on the GPU is executed in lockstep, meaning that all cores in a warp perform the same instruction at once, on different data. This is why linear code is favorable - as soon as there is branching, GPU processing elements have to execute all the branches and delay the end of the operation.

There are several techniques for eliminating branching, and OpenCL provides some built in functions to facilitate the process. An example case from the advect routine follows:

```
if (x<0.5) x=0.5; if (x>N+0.5) x=N + 0.5;
if (y<0.5) y=0.5; if (y>N+0.5) y=N + 0.5;
if (z<0.5) z=0.5; if (z>N+0.5) z=N + 0.5;
```

This can be recognized as just clamping the value of *x, y, z* to the *(0.5, N+0.5)* range. This can be done linearly with OpenCL's *clamp* function.

```
x = clamp(x, (float)0.5, (float)(N + 0.5));
y = clamp(y, (float)0.5, (float)(N + 0.5));
z = clamp(z, (float)0.5, (float)(N + 0.5));
```

Unfortunately, this is not always possible or introduces more complications. There are trade-offs everywhere, and sometimes branches are the least evil. In the setBound routine I have settled for three *if* branches to run the kernel in SPMD (single-program-multiple-data) fashion. This is to differentiate the treatment of the cube sides, edges and corners when setting bounds. There are many more side-voxels than edge-voxels, and only 8 corner-voxels, so I am hoping most of the similar-kind voxels will land on warps that are pure. This way the warp's instruction scheduler would only need to run one branch for its cores and not cause a delay. The impure warps should be few and far between.

4.3.4 Local reductions

Processing units grouped under the same workgroup share a special layer of memory, the local memory. Being slower than private memory, but faster than global, it can be used to exchange some intermediate data among workgroup cores.

Unfortunately there is no good application of local memory for this project, as each cell is processed independently. It could potentially be useful for the Lagrangian approach where particles would need to share some collision detection information with their neighborhood.

5 Parameter Tuning

By this stage the simulation runs parallelised and with the default parameters (20x20x20 resolution, 20 solver iterations). The goal of the project was to set adequate parameters for the simulation to ensure optimal performance. Sections 5.1 to 5.2 will give rationale for a dynamic tuning system, as opposed to static, as well as the construction of it. After that, the list of possible parameters will be given and their modification during execution explained. This includes the dropped time-step parameter. The latter part, section 5.3 will explain parameter change functions and how they were calibrated from test runs.

5.1 Dynamic vs static tuner

In the ideal case we could set simulation parameters analytically based on system resources, such as core clock speed, memory capacity, bandwidth and caches, etc. But this is unreliable, due in part to unpredictable factors, such as background processes and threading overhead. Another way is for the program to make an initial guess for some static parameter values, as OpenCL is quite rich in its knowledge of the hardware - the available memory, cache size, cache line size and number of cores is obtainable. However, the changing factors may still lead the system away from the optimum.

This makes the case for dynamic tuning in real-time, using feedback from live simulation timings. Such a dynamic tuner will strive to reach a preset target FPS, which can be specified at launch.

5.2 Tuning process and parameters

There are two parameters that are readily changeable: the resolution of the volume and precision of the solvers. These are tuned autonomously by a separate component in the program, the tuner. It receives constant feedback of frame times, timed by the system timer, and accumulates a history of 10 (by default). If their average deviates too much from the pre-set target frame time, a decision to change parameters is made. This component has full control over the simulation and makes adjustments perpetually, giving a live response to external changes, such as a system slowdown due to overload.

To make an intelligent decision, the tuner is provided with parameter change functions, which estimate the influence each parameter has on frame times. A change in one parameter will come synchronised with a balanced change in the other. The construction of these functions is in section 5.3.

There are, of course, endless variations possible to achieve this kind of tuning, but this approach was deemed simple and effective enough for a proof of concept.

5.2.1 Volume resolution

Testing has revealed that the size of the simulation volume has the biggest impact on system resource consumption. A change in the resolution by 1 adds or removes three two-dimensional slices from the volume, giving an expected $O(n^2)$ change in performance.

Implementing this change was a difficult endeavour due to the obvious requirement to ensure continuity of material and flow in the simulation space. Expanding or truncating the volume would not be the expected behaviour, and removing the fluid even less so. After a resolution change, all fluid should remain in the same position and consistency, relative to the boundaries of the volume. One way to guarantee this is to proportionally sample density and velocity from the old volume after a new one has been generated.

This is achieved using a separate kernel for resampling, once again utilising OpenCL's easy-access parallelism. It was simplest to use trilinear interpolation, widely used in image resizing [3], on the 3D voxel data. This $O(n^3)$ algorithm is slightly modified to fit the needs of the simulation - it does not interpolate across the sides of the sampling cube - it simply samples from the voxels at the corners. This would make it miss some elements if the field was resized more than 1 unit at a time, which will not be the case.

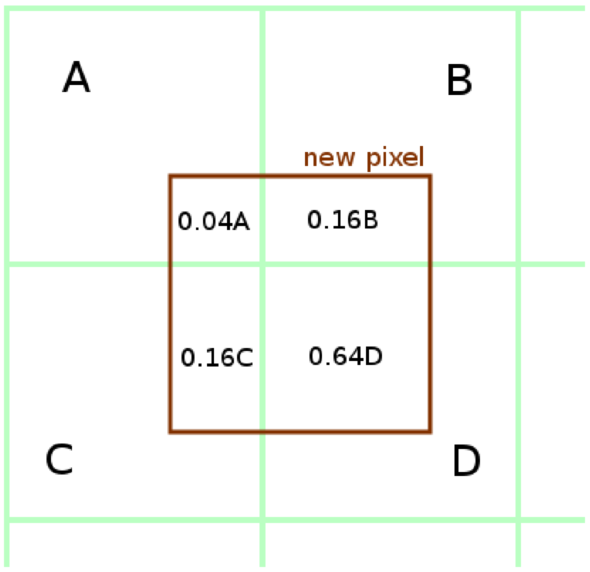

Figure 9: 2D representation of resampling - the new pixel's (brown) corners sample fluid from current pixels

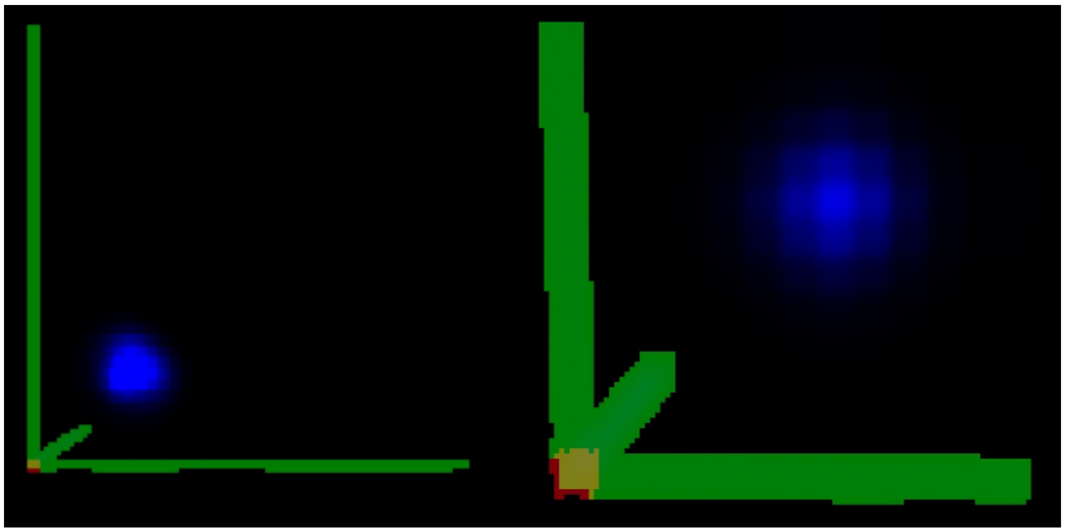

Figure 10: High resolution (30) vs low resolution (10) displaying a blob of fluid

It is arguable whether a change in resolution should be a tunable parameter, because it changes the structure of the simulation volume. Every such change builds a new state as an approximation of the previous state, and resumes simulating from there on. For the purposes of this project such deviations will be allowed.

5.2.2 Precision of the linear solvers

The *diffuse* and *project* stages rely on solving a linear system using an iterative Gauss-Seidel process, where the number of iterations determine the quality of solution. This is controlled by how many times the respective kernel is called from the application code, and is changeable without difficulty - the effects are instantaneous.

While changing the resolution parameter makes a memory-intensive change, the precision parameter influences the amount of processing more. This will be reflected in different devices responding differently to parameter changes.

5.2.3 Time step

The simulation admits a time difference parameter to specify how far in time to construct the new state. The result would be more precise and detailed with a smaller timestep, and less so with a larger one. However, it is possible to gain extra performance by generating a state further in the future and interpolating from the current state, which is cheaper. This could be another parameter, but was not implemented due to a resource constraint - the interpolation would have to happen on the GPU, where the volumes are stored. However, at the same time the next state needs to be computed simultaneously. Two different computations cannot be run at the same time on the GPU.

One way around this would be to stop simulating and pre-compute a batch of interpolated frames before switching back. This kind of interleaving is complicated and needs additional tuning to determine how many frames to interpolate, so was dropped from the list.

5.3 Calibrating the change functions

We need to measure how impactful each of the two parameters is on the end result, the simulation step time. Then we can construct a parameter's *change function*, used to calculate the necessary change in each parameter to achieve a desired effect on the performance. In particular, it is a function $y_{\text{[device]},\text{[parameter]}}(x)$ that takes an integer parameter value and returns the expected frame rendering time for the particular device and parameter.

Several test runs were performed to benchmark how each device type, CPU and GPU, responds to changes in either parameter. The test runs will be referred to by the following labels, denoting the device type and the parameter tested: CPU/resolution, GPU/resolution, CPU/precision, GPU/precision.

The chosen model function for the resolution is $y_{res}(x) = ax^3 + b$. This judgment comes from section 5.2.1 which stated that a change in resolution gives a quadratic change in performance. This defined the derivative of the model function, which therefore should be cubic.

The model function for the precision is linear $y_{prec}(x) = ax + b$, since it only changes how many times some kernels are run.

The test runs will collect frame time data, which will then be used to estimate a and

b using *gnuplot*'s linear regression fitter [23]. To ensure complete coverage, the tests will increment one parameter, starting from a low value, until the step times are reasonably large. The other parameter will be fixed at default values, 20 for resolution and 20 for solver iterations. To minimise noise, the simulation will remain in each state for 20 steps and yield the average step generation time.

CPU/resolution was run for volume resolutions of 5 through 75, when it reached times close to a second.

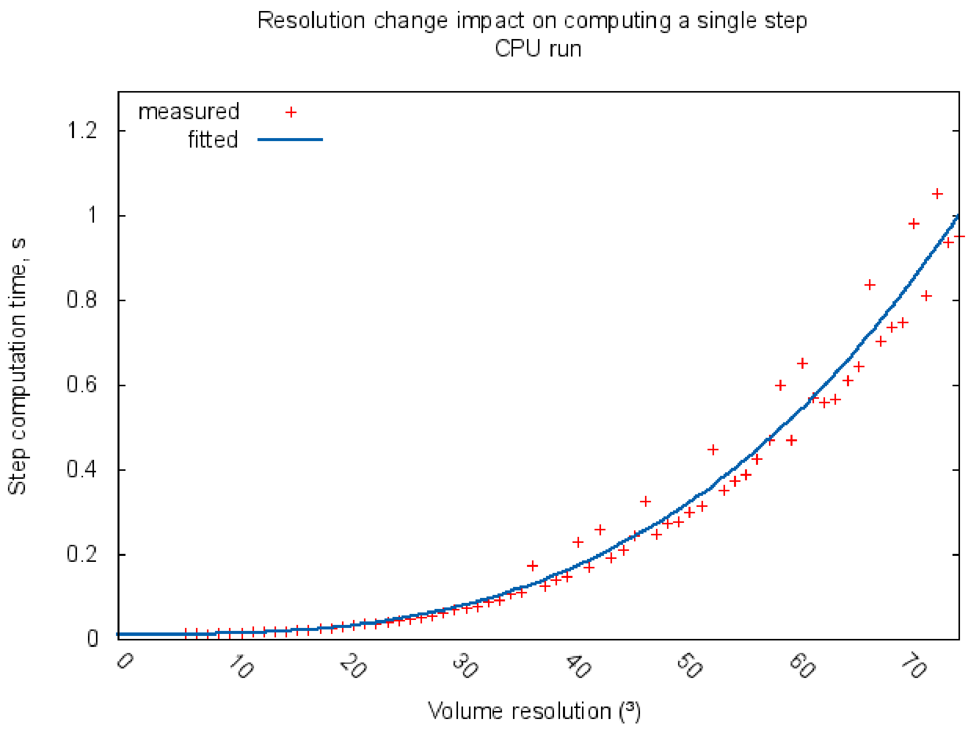

Figure 11: CPU/resolution test run with a best fit line

The graph is noisy due to the fact that the CPU is time-shared by the OS with other processes. They have different priorities, undergo context switches and memory caching operations, which makes the time allowed for one single task unpredictable. This is the reason for some outliers, as seen on the graph.

Best fit parameters were determined to be $a = 0.000002347 \pm 1.8\%$, $b = 0.0112172 \pm 62.6\%$. The standard error for b is very high due to the outliers mentioned above. They push the best fit line above most proper points, which accumulates a large error. Perhaps a good improvement would be to decrease b so that the function matches the points left below the best fit line.

GPU/resolution was run from 5 through 200, when it was still producing 5-6 frames per second.

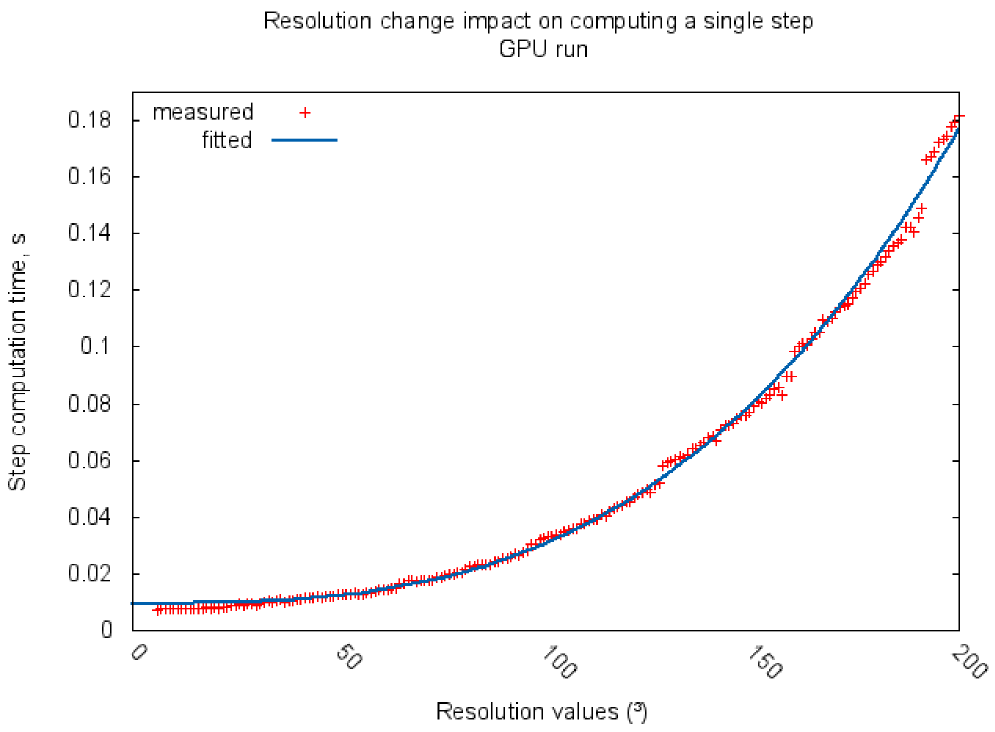

Figure 12: GPU/resolution test run

The best fit line is left out for clarity, as it overlaps the measured points nearly perfectly. The fit is much better than the CPU run's due to less noise and practically no outliers. The parameters are $a = 0.000000021 \pm 0.4\%$, $b = 0.00961385 \pm 2.6\%$.

The GPU is clean from other tasks and therefore much more deterministic, which prevents most noise. There is still a little, though, probably from the scheduling overhead and maybe a little residue of the CPU noise, since kernel launches are still controlled from the CPU.

At higher resolutions there are some interesting discrete performance jumps, which will be discussed in the next section.

CPU/precision tested 1 through 150 solver steps, the frame time model was expected to be linear.

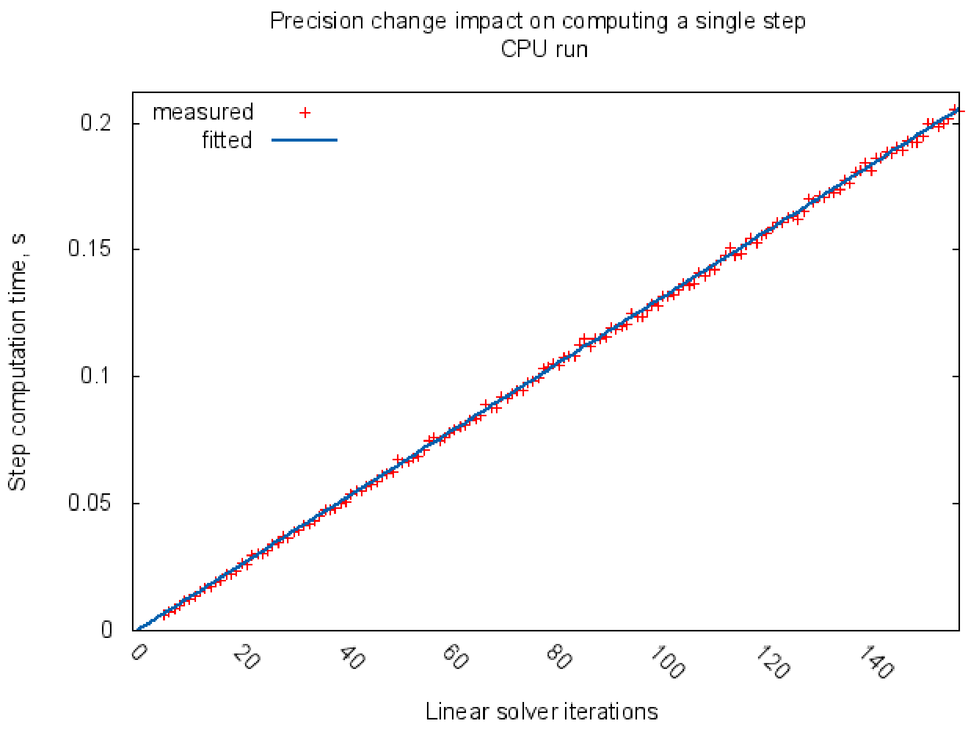

Figure 13: CPU/precision test run with a linear fit

It is indeed linear with a very level growth rate and small errors. The parameters are $a = 0.00129959 \pm 0.2\%$, $b = -0.00157226 \pm 12.5\%$.

The previous explanation for CPU noise should be augmented with the fact that OS scheduling affects memory-intensive calculations much more than processing-intensive ones. It is supported by the known fact that memory is a bottleneck, and background processing tasks make constant memory requests that would slow down the foreground processing of a constantly resizing and reallocating fluid simulation volume. Meanwhile, if we just increase the number of iterations in the diffusion/projection linear solvers, it can cache the working volume and rely on the ALU alone for task completion.

GPU/precision was just as well tested from 1 to 150 steps and showed a very similar pattern as before - a linear progression with minimal noise. For this reason the graph will be omitted, as it is too similar to the above one and does not give any extra information.

The parameters are $a = 0.000376968 \pm 0.2\%$, $b = -0.00059281 \pm 12.7\%$.

Summary The following functions have been generated to gauge performance impact from parameter changes:

- CPU/resolution: $y_{CPU,res} = 0.000002347x^3 + 0.0112172$
- GPU/resolution: $y_{GPU,res} = 0.000000021x^3 + 0.00961385$
- CPU/precision: $y_{CPU,prec} = 0.00129959x - 0.00157226$
- GPU/precision: $y_{GPU,prec} = 0.000376968x - 0.00059281$

Note that these functions are tailored to the particular hardware used in the tests, and the coefficients would be different in other environments. Using these functions as a generic approximation is the best we can do for now. If this is used in an application that allows some time for pre-processing, it could quickly re-run the benchmarks to get appropriate coefficients for the new system.

It is bold to assume that both parameters are independent of each other, which most likely is not the case. This requirement can be relaxed, because we are more interested in the shape of the functions (coefficient a). We want equal contribution to performance improvement from both parameters, and to achieve this the tuner uses the devised functions to evaluate the weighing of both parameters.

6 Results and Analysis

The finished implementation runs a parallelised fluid simulation and renders it to a texture, which is then rendered on the screen using OpenGL. The auto-tuner is persistently changing resolution and simulation precision to reach a target frame rate. This part will look at the results and performance of the auto-tuner in sections 6.1 and 6.2. The test runs from the previous section were interesting in benchmarking two parameters leveraging different system resources - the memory and the CPU - and will be analysed in 6.3. The effects of these test runs, as well as new ones, will be looked at using kernel time distribution graphs. Finally, section 6.4 will have Amdahl's law explain what limits maximum achievable speedup from parallelisation and in light of this project.

6.1 CPU test runs

First, an example run on the CPU, which is dynamically auto-tuned to reach optimum within 200 frames, can be seen below. The stacked histogram plot (Figure 14) shows how much time per frame each simulation stage used.

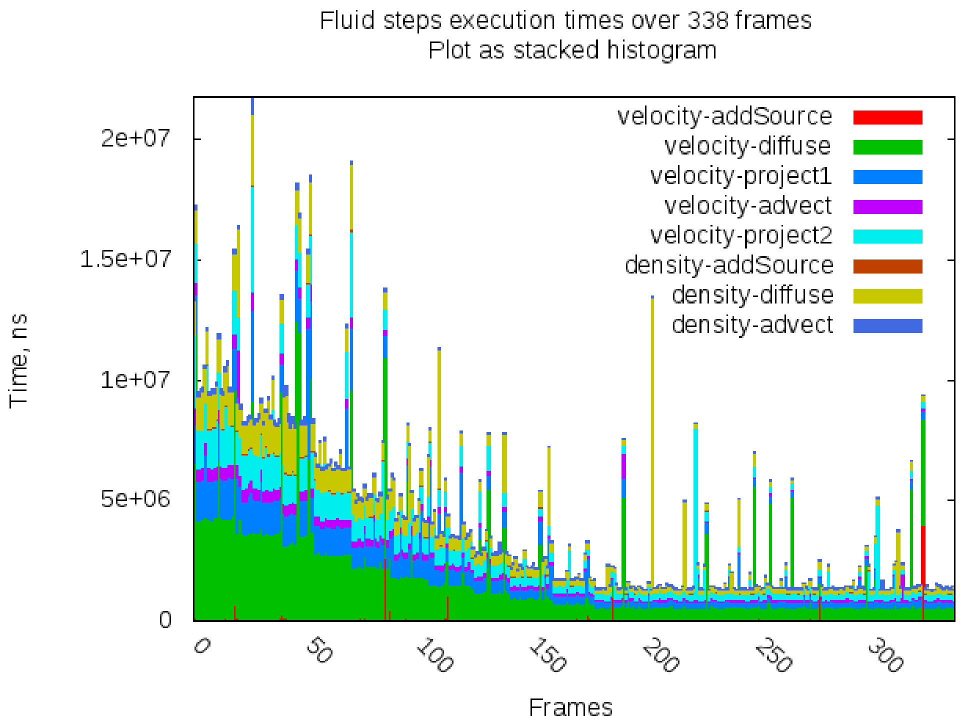

Figure 14: Step execution times during a CPU run

The graph is noisy for reasons explained before, but the general trend is not difficult to see: frame times converge to the target fps of 20, or $5 \times 10^6 ns$ per frame. The initial conditions were too much, which caused the auto-tuner to step in and reduce the parameters to 9 for the resolution and 13 for the solver iterations.

A normalised version of the same data (Figure 15) reveals the time distribution of all stages in a frame. As the tuner changes the parameters, the parts that depend on iterations (*diffuse* and *project*) are expected to diminish in comparison to constant parts.

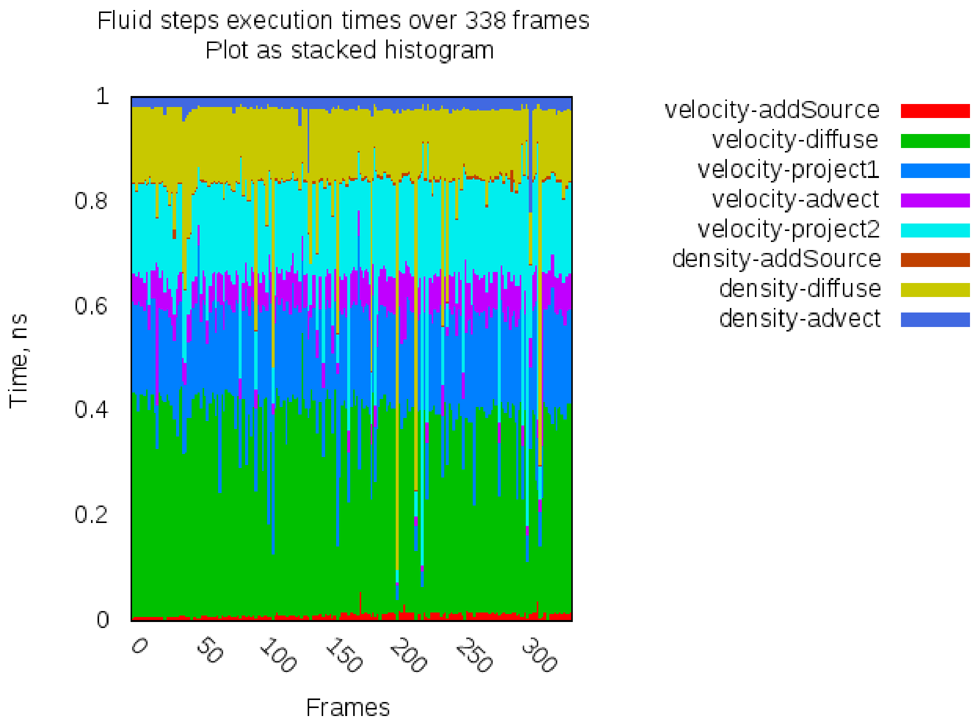

Figure 15: Normalised step execution times during the CPU run

The noise prevents this diminishing from being observed - we can only clearly see a slight growth in the proportion taken by addSource. The GPU test in the following section will provide a clearer picture.

6.2 GPU test runs

GPUs don't have the noise problem, because there is only one task running. The whole process is much more deterministic.

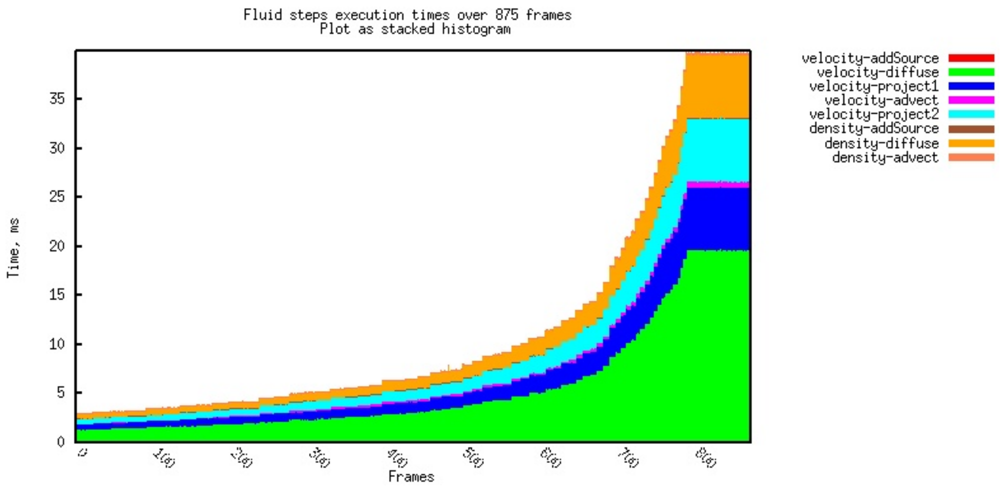

Figure 16: GPU run - 20fps achieved in 875 frames

Here we can clearly see that the iterative operations (project, diffuse) take the most time, as each of them hides many repeated executions of the kernel. The largest part is taken by the velocity-diffuse stage, which is roughly three times the size of other iterative kernels. This corresponds to the three dimensions which are processed during the diffusion of velocity.

A resolution of 82 and solver iterations of 50 are reached, which is a much better state than given by the CPU run.

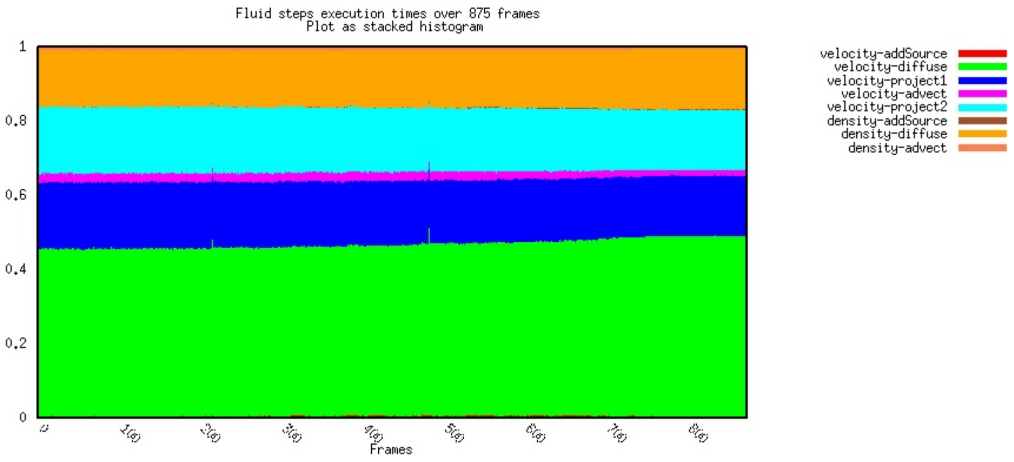

Figure 17: GPU run - normalised

The normalised kernels reveal how the proportion of the iterative stages slowly rises. This hints at the fact that the precision parameter has the potential to overtake resources and should be constrained in practice. For the current configuration, however, there is no such danger.

6.3 Parameter calibration runs

The above measurements arise from an optimally weighted adjustment of both parameters, leaving a combined effect on the end-result. However, this simultaneous tuning hides some interesting correlations exhibited by each parameter separately. The following subsections will revisit the change function calibration test runs from the section before, which acted purely on one parameter only, keeping the other fixed. This will reveal how the devices respond to different computational complexities and memory/CPU utilisation.

CPU/resolution test run had an outliers problem that shifted the change function up. The kernel time distribution shows that the outliers were not caused by random noise bumping the averages, but consistent intervals of low performance, appearing with a clear regularity.

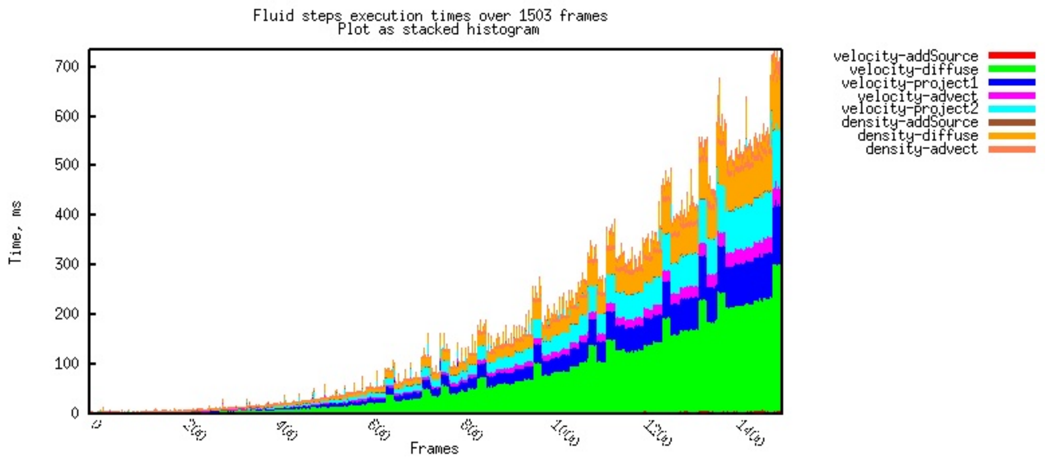

Figure 18: CPU/resolution run - kernel times

Each spike interval is 20 frames wide, corresponding to a certain resolution setting. The intervals don't show a clear pattern and the cause of them is a bit mysterious, but I speculate it to be one of:

- Memory word/cache line misalignment or cache misses at certain resolutions
- Uncalled/random write-back to a further cache level by the memory manager
- OS deliberately throttling or renicing the process priority (unlikely)

An improvement would be to mark these specific resolutions as *bad* and jump over them when tuning. They may, however, change, lie in different spots for different CPUs, or not appear at all, highlighting the unpredictability factor of performance tuning.

The normalised distribution shows a consistent proportion of the frame taken by each kernel, which is expected.

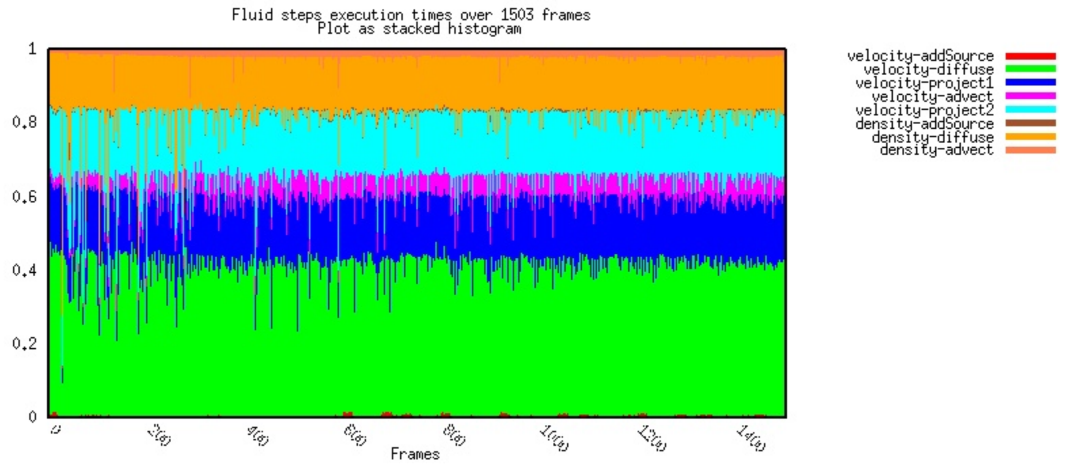

Figure 19: CPU/resolution run - normalised kernel times

Note that the beginning stages (lower resolutions) are visibly noisier than the rest. This is because the workload at lower resolutions is comparable with typical OS background processes, and the simulation process can be easily *jarred* by them. In later stages, however, the simulation takes the bulk of the processing time and background task effects appear negligible.

GPU/resolution test had a great performance leap stemming from the parallelism, but the graph showed some clear jumps in frame times at certain resolutions.

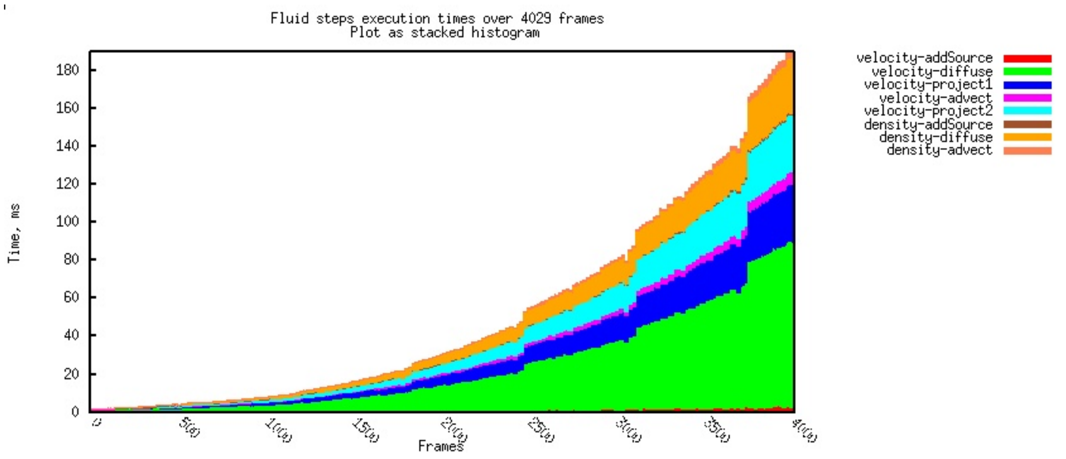

Figure 20: GPU/resolution run - kernel times

The kernel time distribution confirms what was observed before - there is a step-like progression of resolution impact on the frame time. This could happen when:

- The volume is large enough to spill into a slower memory level
- The number of voxels exceeds an integer multiple of the number of cores

To elaborate on the 2nd point, there are 1024 cores on this particular testing GPU. The leaps take place roughly at resolutions of 100, 130, 170, 190. These would correspond to approximately 1000, 2000, 5000, 7000 kernel runs per core. There is again no clear pattern, but, the speculation is that there could be some thresholds for the instruction scheduler. Knowing more about the internals of nVidia GPUs would help to make a more educated guess for this phenomenon.

Another interesting fact is that in each interval there is a slight dip before picking up. This could be coincidental or an even more intricate feature of the GPU memory architecture.

The normalised distribution is smooth and the bounds are horizontal for each kernel, giving no extra information.

CPU/precision and GPU/precision runs had no anomalies and showed linear growth, as expected. The kernel time distribution structure is the same for both, although the CPU versions are noisier and difficult to read. For this reason only the GPU kernel distribution times will be presented.

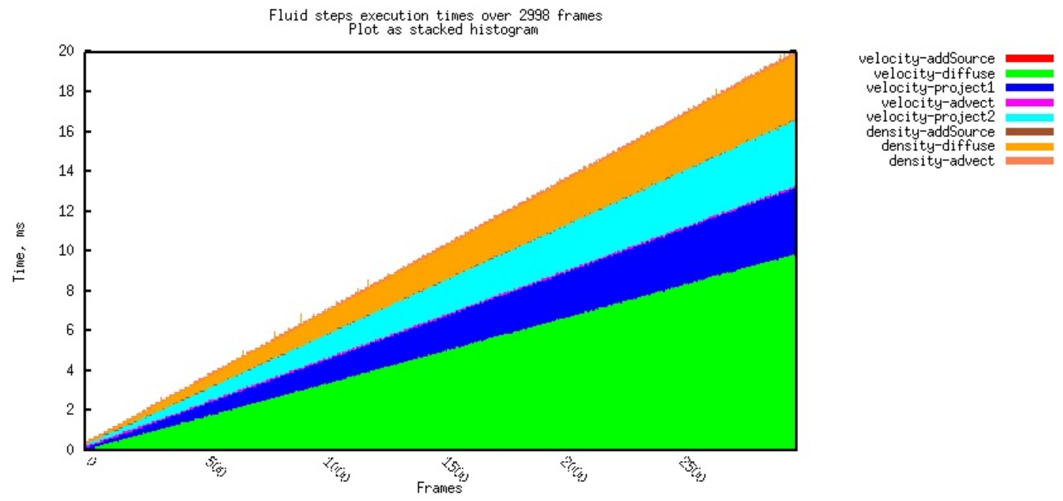

Figure 21: GPU/precision run - kernel times

Four bands emerge, corresponding to the two scalable stages: projection and diffusion. The normalised graph shows that the other kernels, such as advection and addSource, were swallowed quickly by the upscaling.

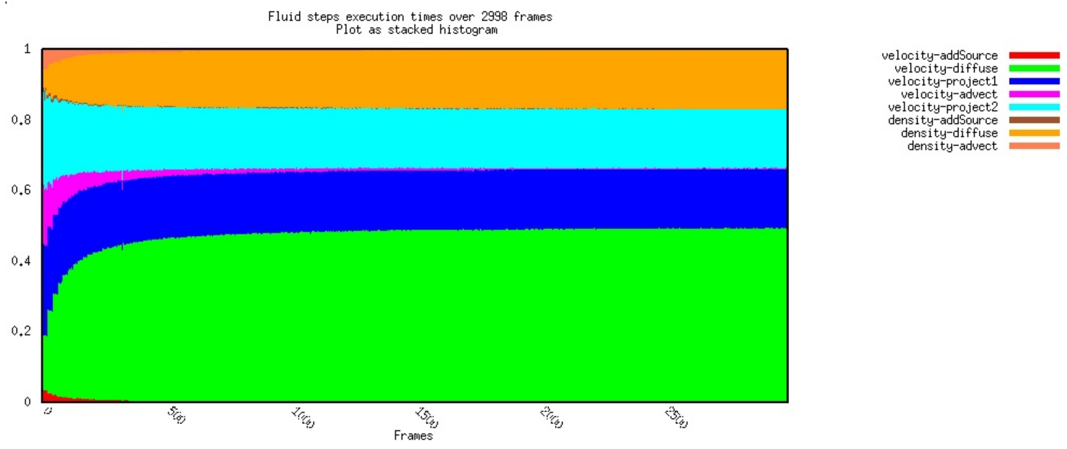

Figure 22: GPU/precision run - normalised kernel times

This kind of scaling by adding iterations turns out to have an overwhelming effect on the GPU usage distribution and should be moderated. Observation suggests that the quality of improvements starts to diminish for precision values beyond 30, therefore it is not necessary to squeeze out that fine margin of improvement. For this reason, in visual applications the tuner should prevent precision values from exceeding 30.

Even with the suggested moderation, the hard truth is that the solvers put the bulk load on the processor. Beyond all optimisations mentioned before, the more intelligent tackle on performance issues would be creating faster, possibly non-iterative solvers or devising new approximation methods.

6.4 Theoretical limits

It would be useful to know if the implementation is achieving the theoretical limits of the GPU, as reported by the manufacturer. The two main metrics that determine GPU performance are the processor FLOPs (floating point operations per second) and the memory bandwidth. For the testing hardware these are reported to be 2488.32 GFLOPS and 327.744 GB/sec of bandwidth respectively [15].

6.4.1 Memory bandwidth and FLOPs

To compute memory bandwidth utilisation, the total number of reads and writes of 4-byte floats were counted. Refer to the table in section 4.2, where the last three columns show the reads, writes and calls for each kernel. These numbers will help estimate the total amount of data travelled from the global to the private memory.

The auto-tuned GPU test run showed that the simulation runs comfortably at resolution n=82 with S=50 solver iterations.

$$\begin{aligned}
\text{perframe}(n, S) &= 6n^3 + 12n^2(6S + 14) + 16n^3 + 16Sn^3 + 24n^3 + 32Sn^3 + 48n^3 \\
\text{persecond}(n, S) &= 4\,\text{bytes} \times 20\,\text{frames} \times \text{peframe}(n, S) \\
\text{persecond}(82, 50) &= 112035825920\,\text{bytes} = 112\text{GB}
\end{aligned}$$

The summative memory transfer per second is 112GB out of 327GB, or 34% of the bandwidth. This is satisfactory, given that this calculation leaves out CPU to GPU memory transfers and a significant portion of the time is spent on actual computations and scheduling.

Figuring out achieved FLOPs is a laborious and error-prone process if counted manually. To assist this there are visual tools, such as *nVidia NSight Visual Studio Edition* and *Visual Profiler* based on *nvprof*, a profiler from the nVidia SDK. Unfortunately these programs work only for CUDA applications and there seem to be no OpenCL alternatives, which forces us to forfeit the FLOPs calculation.

6.4.2 Amdahl's Law

The current implementation juggles with 8 kernels, on top of auto-tuning and possibly rendering - this considerable background load is sequential and deters frame generation times. This effect is formalized by Amdahl's law [16], stating that the maximum amount of parallelism achievable by a system is limited by its sequential fraction. Calculation of the maximum possible speedup can be done using the formula:

$$S_{max} = \frac{N}{(B \times N) + (1 - B)} \tag{2}$$

where

- S_{max} - the maximum possible speedup (times)
- N - number of processors used
- B - sequential fraction of the algorithm

A simple auto-tuning run was performed and timings collected to find out the sequential fraction of the program. Any OpenCL program can be run with the environment variable COMPUTE_PROFILE=1, causing it to output log files during execution. The produced data includes the time spent on the GPU (parallel) and the corresponding sequential CPU time for each kernel run, as well as all the memory transfers between the two.

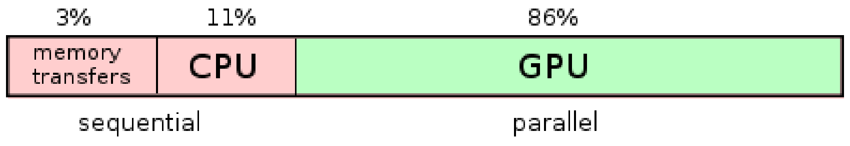

Figure 23: Distribution of CPU and GPU contributions to the simulation

At least 14% of the frame is still processed sequentially. Another such run was performed in the fully sequential mode of operation, where a single GPU core is made to do all the work. This showed the CPU fraction to be a mere 0.06%. Evidently, the multi-kernel scheduling overhead takes a larger portion of sequential time, but the price is worth paying to achieve lower absolute times.

With auto-tuning turned off and parameters fixed at 20 for resolution and 20 solver iterations, let us compare the times taken to render 100 frames:

- Rendering 100 frames in parallel mode: $0.87s$
- Rendering 100 frames in sequential mode: $52.07s$
- Speedup $= \frac{52.07}{0.87} = 63.5\times$

The speedup is indeed great, as expected from a 1024 core GPU. Unfortunately we cannot apply Amdahl's maximum speedup calculation on a chain of several algorithms that our simulation pipeline contains. We can, however, get a naive estimate by allowing some crude assumptions:

- The pipeline is a single algorithm splittable into tasks
- Sequential fraction stays around 0.06% to 14%
- The number of processors is 1024

Two calculations will be done using equation 2 for both mentioned percentages of sequential fractions, to get a general picture of the range of expected speedup. These assumptions should overestimate, because they do not account for the added complexity of kernel scheduling and memory contention.

$$S_{hi} = \frac{1024}{(0.0006 \times 1024) + (1 - 0.0006)} \approx 634\times$$
$$S_{lo} = \frac{1024}{(0.14 \times 1024) + (1 - 0.14)} \approx 7\times$$

The range is large, spanning a whole order of magnitude, which is not great, but the result of 63× does sit comfortably in it. If we could peg the sequential fraction to a certain number, it would be more accurate, but this would require a more involved analysis. On the whole, the potential of parallelisation has been utilised well.

7 Conclusion

This has been a research and engineering challenge turned into exploration of parallelism of different devices and different natured computations. To finish off, an evaluation of the system and where it falls short will be given in section 7.1. The points given here do not fall among the postulated goals of the project, but would have been beneficial for a deeper insight. Finally, an overall picture of the state of the project and parallelisation as an area will be given.

7.1 Evaluation and Future optimisations

The current implementation is by no means perfect and could have been improved further. Here are listed some potential improvements that were within reach.

Rendering The one aspect that could have been done better is the renderer. Currently it is only a density tracer that outputs simple images. The stated aim of the project was to produce a simulation useful for visual applications, but the actual component that produces visuals was given only cursory attention. Still, this was done not to go astray from the primary focus of the project, parallelisation and performance tuning.

Improvements to be considered for the renderer include:

- Calculating whether the ray intersects the simulation box before doing interval sampling. This would speed up the performance and prevent shooting some rays in vain.
- Using a more elaborate color transfer function to highlight different densities of the fluid. One color is rather limiting and hides away the more subtle details of density variations.
- Linear interpolation between the sampling intervals would prevent aliasing. Although no artifacts were observed in the current test runs, they could appear when the fluid is very sparse.

Furthermore, if visual quality is of importance, the change of resolution is not completely smooth, especially on lower resolutions of size 5-20. This could be remedied in the visualisation stage by applying a blur filter or shooting slightly randomised rays to achieve the blur effect.

Optimisations OpenCL 2.0 would have been a great benefit for this project [18]. It promises simultaneous read-and-write access to Image3D objects, which would enable the spatial memory coalescing optimisation. An even more tempting addition is dynamic parallelism, i.e. calling of kernels from kernel code. This would keep the control flow on the GPU and make the simulation program look more like its original C version, not a havoc of kernels constantly queued by the host.

With some more effort, time-step as a tunable parameter could have been viable. The simulation and interpolation phases would just need to be scheduled in a reasonable manner. To leverage full heterogeneity of the system, the interpolation could take place on another GPU or even the CPU, if there was a way to transfer the memory efficiently.

Some valuable insights could be gained by testing on other device types OpenCL claims compatibility with, such as DSPs and FPGAs. For the latter there exists a publication called *An FPGA-Based Floating-Point Jacobi Iterative Solver* [19], which could be used to speed up the current solver process. The larger significance of testing more devices is in exploring how different architectures cope with fluid simulations, possibly making an argument for big vs small caches and few fast or many slow cores, and other architectural idioms.

Tuning The tuning system is basic and simply *pokes in the dark* until the desired performance is reached. The change functions could be used for more informed decision making, aiming to reach the goal in one step. The obvious downside here is that, if it missed the goal due to a miscalibrated function (say, running on a foreign device), it would keep jumping back and forth without reaching it.

7.2 Final remarks

This was an exercise in using current technologies to achieve parallel computation, simulating fluids in particular. The objectives have been reached - a renderable fluid simulation program, which tunes automatically towards a specified frame rate, has been produced using OpenCL. The benchmarks showed a 63× gain in performance over the sequential version, which is considered a success.

The system was made with modularity in mind and could be fitted in a game engine or visualisation program. The implementation runs well and is fairly optimised, although not without some pitfalls along the way. Not all of the GPU benefits could be exploited due to unexpected factors, but they were at least apprehended. While the main goals have been reached, it could pay off to push for some of the more advanced techniques mentioned, and there could be even more surprises from the yet untested processor types.

Testing on a processor with a large number of cores (GPU) was key to this work. However, CPU tests were also performed. As expected, the CPU responds differently to the same computations, showing a clear inferiority in performance. This stems from its background load, different architecture, memory hierarchy and processing element configuration.

On the whole, parallelism as an industry is still in active development. While the OpenCL C as a language is a great leap forward from writing shaders in GPU assembly and reading the frame buffer, the overall feel is still low-level. OpenCL claims to be a portability library, but the program can still fail to compile when different drivers (ICDs) are used. Complications are introduced by SDK vendors rushing ahead of the standard, providing future functionality through extensions. The pitfall is that using them can lock you in with a particular ICD and vendor, as the case with 3D image writes for this project.

Fluid simulations may not play perfectly well with the current state of OpenCL; much of the hassle was managing the 8 kernels from the host code. Each had to have parameters set up, enqueued, possibly waited on and error-checked. This meant that the host's control code for a single frame was 250 lines long and difficult to manage. Then again, fluid simulations are a complex process compared to some less chaotic GPGPU uses, such as vector operations or image filtering. Several interdependent kernels make for a scheduling-heavy application with a considerable CPU overhead and are generally cumbersome.

The future of this project will see a shift from research to engineering challenges as I seek more practical outputs. Particularly, I am planning to have the simulation produce fire for the demonstration and possibly try to fit it in a game engine.

References

[1] Navier-Stokes equations
CFD Online.
28 August 2012
http://www.cfd-online.com/Wiki/Navier-Stokes_equations (20.01.2015)

[2] Real-Time Fluid Dynamics for Games
Proceedings of the Game Developer Conference, March 2003. Jos Stam
http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/GDC03.pdf
(21.01.2015)

[3] Quick image scaling algorithms
Thiadmer Riemersma, ITB CompuPhase, 2001, The Netherlands
September 6, 2001
http://www.compuphase.com/graphic/scale.htm (20.01.2015)

[4] Daniel Holden, programmer and artist, author of Corange
http://www.daniel-holden.com/

[5] GPUTracer
Experimental raytracer and raycaster based on OpenCL
http://code.google.com/p/gputracer/ (21.01.2015)

[6] OpenCL Implementation of Fluid Simulation
Eric Blanchard
COMP599 - Fundamentals of Computer Animation
http://www-etud.iro.umontreal.ca/~blancher/projects/cl_fluids/rapport.pdf (20.01.2015)

[7] Memory CoalescingOpenCL Implementation of Fluid Simulation
Cornell Virtual Workshop
Introduction to GPGPU and CUDA Programming
https://www.cac.cornell.edu/vw/gpu/coalesced.aspx (02.04.2015)

[8] Fluid Simulation for Dummies
Mike Ash, 2006-03-13
https://mikeash.com/pyblog/fluid-simulation-for-dummies.html (24.03.2015)

[9] The Hodge Decomposition
Nikolai Nowaczyk, January 2010
http://math.nikno.de/sites/default/files/hodge.pdf (25.03.2015)

[10] Practical Fluid Dynamics
Mick West, Neversoft co-founder (article on Gamasutra)
http://www.gamasutra.com/view/feature/1549/practical_fluid_dynamics_part_1.php?print=1
(25.03.2015)

[11] Have We Been Interpreting Quantum Mechanics Wrong This Whole Time?
Natalie Wolchover, Quanta Magazine, Science, 06.30.14 (wired.com article)
http://www.wired.com/2014/06/the-new-quantum-reality/ (27.03.2015)

[12] Fluid Simulation Overview
Chrissie C. Cui, The University of Maryland
http://www.cs.umd.edu/class/fall2009/cmsc828v/presentations/Fluid_Sim_Overview.pdf
(27.03.2015)

[13] GPU Performance Optimisation
Alan Gray, EPCC, The University of Edinburgh
www2.epcc.ed.ac.uk/ãlang/GPUHW/GPU_Optimisation.pdf (27.03.2015)

[14] Chapter 30. Real-Time Simulation and Rendering of 3D Fluids
Keenan Crane, Ignacio Llamas, Sarah Tariq; GPU Gems 3
http://http.developer.nvidia.com/GPUGems3/gpugems3_ch30.html (27.03.2015)

[15] nVidia GeForce GTX 590 specifications
Video Card Database on GPUReview
http://www.gpureview.com/geforce-gtx-590-card-648.html (27.03.2015)

[16] Amdahl's Law, notes from a course on Parallel Compting
Aaron Michalove, Washington and Lee University
http://home.wlu.edu/ whaleyt/classes/parallel/topics/amdahl.html (31.03.2015)

[17] Particle-Based Simulation of Fluids
Simon Premoe, Tolga Tasdizen, James Bigler, Aaron Lefohn and Ross T. Whitaker, EUROGRAPHICS 2003
https://www.sci.utah.edu/publications/premoze03/ParticleFluidsHiRes.pdf
(01.04.2015)

[18] Khronos Releases OpenCL 2.0 - updates and additions
The Khronos Group
https://www.khronos.org/news/press/khronos-releases-opencl-2.0 (01.04.2015)

[19] An FPGA-Based Floating-Point Jacobi Iterative Solver
Gerald R. Morris, Viktor K. Prasanna

Department of Electrical Engineering, University of Southern California
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1575859 (01.04.2015)

[20] Enhancing Computational Performance using CPU-GPU Integration
Sukanya.R, Swaathikka.K, Soorya.R
International Journal of Computer Applications
http://research.ijcaonline.org/volume111/number7/pxc3901257.pdf (01.04.2015)

[21] Raycasting to the Polygon Level
Hernantas, Game Development website
http://hernantas.com/2014/05/raycasting-to-the-polygon-level (01.04.2015)

[22] Fine-Tuning Rotorcraft Simulations
Jasim Ahmad and Neal Chaderjian, NASA Ames Research Center
https://www.nasa.gov/content/fine-tuning-rotorcraft-simulations/#.VRx4ZXW1W00
(02.04.2015)

[23] *Fit* command reference, gnuplot manual
http://gnuplot.sourceforge.net/docs_4.2/node82.html (02.04.2015)