Undergraduate Honours Project preliminary report

Parallelisation and Parameter Tuning of Fluid Simulation on Heterogeneous Computing Devices

Olafs Vandans
Student number: s1139243
Exam number: B021719

Supervisors: Taku Komura and Christophe Dubach

January 2015

1 Introduction

Viscous flow is a widely occurring natural phenomenon that, with the advent of computing, can now be reproduced and studied outside of a laboratory setting.

Computationally simulated fluids are commonly used in a variety of industrial and scientific disciplines. A typical application is special effects in computer games and films, using fluid simulators to generate such phenomena as fire, smoke, water or any other flowing substance. Engineering disciplines have used fluid simulations to study air flow around the wing of an aeroplane and aerodynamic properties of cars, hydrodynamics and many other fields.

These applications require simulation at an appreciably high level of detail, which puts a considerable demand on computing resources. This, along with the data-parallel nature of fluid simulations makes parallelism a promising way to reach acceptable performance.

The aim of my work was to implement a real-time fluid simulation utilising modern parallel hardware and adjusting its parameters during run-time to ensure optimal frame rate. My implementation would have to serve a generic use case in computer graphics, so visual consistency is a higher priority than physical accuracy. This has been successful so far, resulting in a C++ program running a simulation on OpenCL and rendering it with OpenGL. This report describes the work and its intended direction for the following months.

2 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.

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

Here the top equation describes velocity ($\mathbf{u}$) with the parameter ν and external force $\mathbf{f}$, and the bottom equation describes density (ρ) with parameter κ and added density S.

It should be noted that these equations incorporate a simplification that the fluid is incompressible. This prevents such phenomena as wave propagation, making it less suitable for 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 numerous entities according to the same rule set, and fluids are no exception. For computing representation, fluids can be discretised into particles or a constant grid of cells with varying amounts of fluid (densities). The latter scheme will be used in this work, and it yields well to parallelisation on SIMD processors such as GPUs.

3 Parallelisation with OpenCL

OpenCL is a framework for expressing 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 of 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.

3.1 Execution model

The OpenCL runtime can enumerate all compatible devices on a system and send computations to them for execution. It 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 specify data-parallel calculations is to run a kernel over a multi-dimensional range (NDRange) of 1, 2 or 3 dimensions. This range has a further split into same-dimensional workgroups.

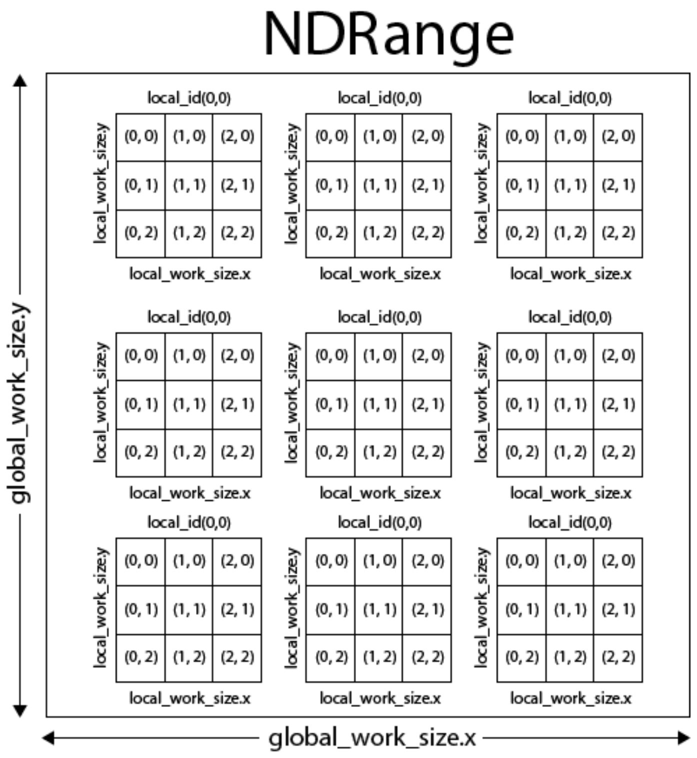

Figure 1: NDRange showing workgroups in 2 dimensions

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.

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.

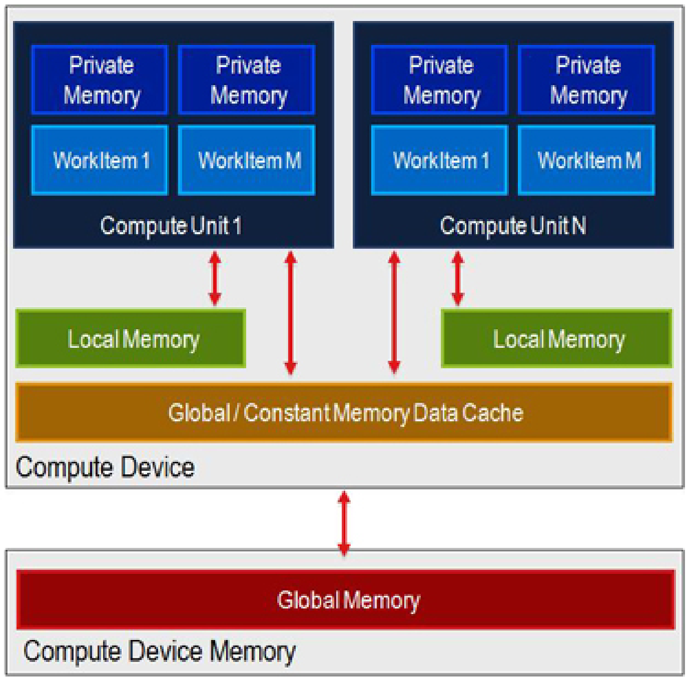

Figure 2: OpenCL memory structure

3.3 Optimisation

Performance optimisations rely mostly on memory manipulation and branching, 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.

4 Reference implementation

Jos Stam's paper [2] was the starting point of my implementation. Here the simulation space is given as a two dimensional grid split in $N \times N$ cells. It is bounded by a single layer of special cells acting as a buffer to prevent leakage of fluid. 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.

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:

- **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. The fluid properties are set by the ν and κ parameters, which determine the viscosity and speed of propagation of velocity changes. More about the stages and intricacies of fluid simulation will be in the final report.

On the implementation level, after the addSource steps, the v_0 and d_0 are used as buffers for storing intermediate results, and need to be cleaned after every step. Modifications to the volume, such as adding material or forces, are done in the host application and passed to the addSource routine. This is the only memory transfer from CPU memory (RAM) to device memory and is potentially a bottleneck.

4.1 Extending to 3D

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

Most of the processes 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. More on this process to follow in the full report.

5 Rendering

An essential part of the project was to visualise the results. For this 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. A separate kernel that raycasts the 3D volume and produces a 2D texture was written.

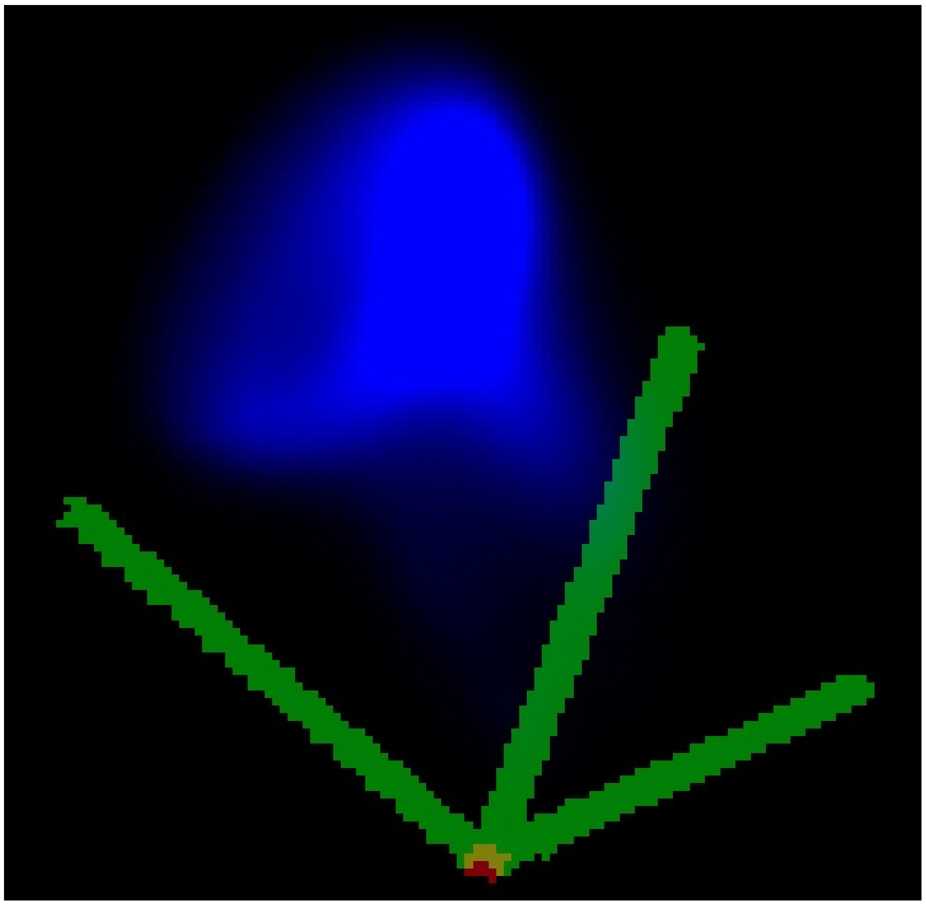

Figure 3: Fluid raycast in 3D space

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. More about this and the whole raycasting operation will be described in the full report.

By this stage the simulation is completed and will not change semantically. The parallelisation and optimisation steps that follow are only performance-targeting changes that conserve the rules of the simulation.

6 Parallelisation

6.1 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. 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.

This task has already been done, though in a slightly different way, by Eric Blanchard [6] and this text will make remarks on his methods.

6.2 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. 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
addSource	$O(n^3)$	density, velocity step
diffuse	$O(Sn^3)$	density, velocity step
advect	$O(n^3)$	density, velocity step
project1	$O(n^3)$	velocity step
project2	$O(Sn^3)$	velocity step
project3	$O(n^3)$	velocity step
setBound	$O(n^2)$	advect, diffuse, project operations

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

Splitting into multiple kernels is also the approach taken in Blanchard [6]. The creation and scheduling overhead should be insignificant compared to the flexibility it gives. This is also good in that there will be no cross-device memory transfers between kernel calls.

7 Parameter Tuning

In the ideal case we could set simulation parameters analytically based on system resources, such as processing unit clock speed, etc. This is unreliable, due in part to unpredictable factors, such as background processes and threading overhead. The program could only make an initial guess, 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 unpredictability makes the case for dynamic tuning in real-time, using a system that takes current framerate as a guideline and makes adjustments perpetually.

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, which checks the frame rate every second and tries to adjust the parameters to ensure a balance between performance and quality. The tuner aims to reach a preset target FPS, which can be specified at launch.

7.1 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 can give 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 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.

Furthermore, if visual quality is of importance, it should be noted that 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 same effect.

7.1.1 The resampling kernel

It was simplest to use bilinear interpolation, widely used in 2D image resizing [3], on the 3D voxel data. The details of this $O(n^3)$ algorithm will be explained in the full report.

7.2 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 determined by a parameter in the application code, which changes how many times these kernels are run. No other modifications are necessary for this value to take effect.

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.

8 Current progress

Currently the implementation runs a parallelised fluid simulation and renders it to texture, which is then rendered on the screen. It is facilitated by a virtual camera, movable by the keyboard to capture the simulation from different angles. Fluid and velocity addition is limited to a single point and also controlled by keyboard.

The implementation uses the OpenCL profiling facility to log start and end times of kernels. An example run on the CPU, which is dynamically autotuned to reach optimum within 200 frames, can be seen below. The stacked histogram plot (Figure 4) shows how much time per frame each simulation stage used.

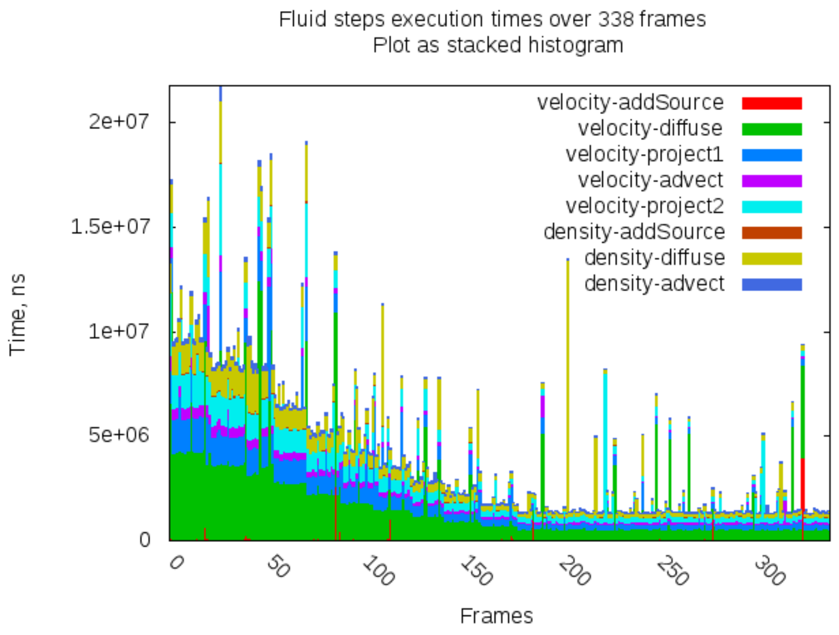

Figure 4: Step execution times during a test run

The graph is noisy, 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.

A normalised version of the same data (Figure 5) reveals the time distribution of all steps 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. This is not observed, however, and more investigation will be carried out to determine why.

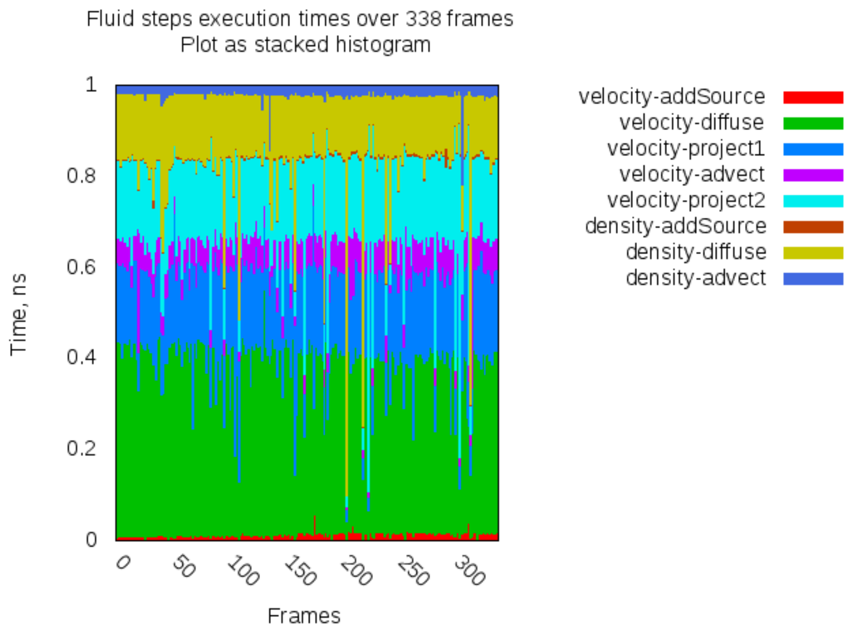

Figure 5: Normalised step execution times during a test run

Overall, the distribution is quite balanced and it is difficult to identify which stage to tweak. There is no one kernel that overshadows others by a significant margin.

8.1 Next steps

Until February 1st To this moment only CPU test runs have been made, but testing on GPU is key to this project. The next steps are to run benchmarks on a powerful GPU to determine how effective this parallelisation has been. The implementation also has an option to run everything sequentially on 1 thread, which is expected to outperform the parallel version on CPUs with a low number of cores. Testing whether it does will also be important.

Until March 1st Currently the implementation uses only a few branching elimination optimisations. It could pay to try some of the more advanced techniques mentioned, as there could be surprises from the yet untested GPUs.

Rest of the time Some valuable insights could be gained by testing this on other device types OpenCL claims compatibility with, such as FPGAs and DSPs. The significance of doing this 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.

To get some practical output from the work, I am planning to have the fluid simulation produce fire. This is more of an engineering problem than a research problem, however.

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)