Technical University of Denmark

Wojciech Pawlak (s091820)

# Final Report

A GPU-accelerated Navier-Stokes Solver using OpenCL

Special Course at GPUlab, Scientific Computing Section, DTU
Supervisor: Allan P. Engsig-Karup Ph.D.

3 August 2012

**DTU Informatics**
Department of Informatics and Mathematical Modelling

# Contents

# List of Figures

# List of Tables

# 1 Introduction

The goal of this report is to document the work done for individual special course "A GPU-accelerated Navier-Stokes Solver using OpenCL". The project took place in GPUlab of Scientific Computing Section of Department of Informatics and Mathematical Modeling (DTU Informatics) at Technical University of Denmark. The project work for this special course took place between late March until early August 2012 and was supervised by Allan P. Engsig-Karup Ph.D. The course was worth 5 ECTS points.

The practical goal of this project was to design and implement a scientific computing application for execution on GPUs. The application was a Partial Differential Equations (PDE) solver for Navier-Stokes equations, which form a basis for most of Computational Fluid Dynamics (CFD) problems. The application was supposed to make us The solution was based on sequential implementation by Griebel et al.[2]. The implemented solution was verified and validated through simulation of standard benchmarking problems in the field like lid-driven cavity problem, flow-over a backward-facing step, flow past an obstacle and others. The performance of implemented PDE solver was analysed for different sizes of two-dimensional grids. As different optimization techniques were applied for improving performance of the PDE solver, solution consists of several versions, varying in the level of optimization. The target platform was a Graphical Processing Unit (GPU). The technology used in implementation was OpenCL, an open standard framework for implementation of programmes that execute across various heterogeneous multicore architectures like CPUs or GPUs. Thus understanding the intrinsics of platform and the approach to write parallel programs using chosen technology was a prerequisite to meet the goals of the project. Moreover, for the project to be feasible familiarization with the application of basic principles of numerical approximation and discretization was necessary.

The report is structured as follows. Section 2 is a survey of current (as of July 2012) state of technologies that can be used for development targeted to GPU architecture. A look into how different vendors implement the OpenCL standard is presented. In addition, a general overview of current achievements in GPU architectures are shown. Section 3 consists of description of necessary theory behind the problem in question. Section shows the Navier-Stokes equations and how are they discretized. The algorithm is described here. In Section 4 design of the application and its implementation is described. It shows how the algorithm is implemented and presents the details about structure of the code. Furthermore, challenges behind implementation are presented. Section 5 consists of the results and performance analysis. The report is concluded with Section 6, where general findings are summarized and possible improvements and future work is presented.

# 2 Survey of GPGPU programming

## 2.1 General trends in GPGPU programming

General-purpose computing on graphics processing units (abbreviated GPGPU) is a technique that utilizes graphics processing units (GPUs). Initially they were used to handle advanced computations in computer graphics in games and visualisations. However, there is a growing trend to extend their usage to perform data-intensive computations that were until not long ago still bound to CPUs only. This means an overall acceleration in variuos general-purpose scientific and engineering applications. In this project, a GPU-based computation-intensive program was implemented, so the platform at hand had to be thoroughly investigated to make most use of it. Thus, this section will focus on this platform mainly and will try to define current efforts in GPGPU programming as of July 2012. The descriptions will be supported with personal experiences gathered during the development process.

Massively parallel, programmable GPU architecture enables superior performance and power efficiency thorugh thousands of smaller, more efficient cores that are mainly intended to be used for parallel computation. Current CPUs consist of a few cores optimized for serial processing. While CPU runs the remainder of the code that mainly controls the flow of the program, GPU allow for order of magnitude faster processing of data through parallelizable data-intensive algorithms. To exploit the high performance of GPUs though, the implementation must express sufficient fine-grained parallelism. Moreover, GPU–CPU data transfer should be minimized and coalesced memory access achieved. A programming model to achieve that has to be provided yielding best performance results possible with a burden of low-level programming. Although the field is relatively young, it progresses extremely fast as already hit the mainstream.

First immediate observation is that there is a still diffusion of non standardized methods in the field, although standardization efforts can be observed. OpenCL is the currently dominant open general-purpose GPU computing language. Dominant proprietary framework is NVIDIA's CUDA framework. One option for writing code to execute on GPU is thus to use directly one of these frameworks. However, for a large scientific Fortran or C code, that was written long before the GPUs even existed, this requires a large effort to rewrite, debug, and maintain a separate version of the code.

Furthermore, in the advent of heterogeneous computing systems of which CPU-GPU combination is an example grow in their complexity. They usually consist of a variety of device architectures like: multi-core CPUs, GPUs, Accelerated Processing Units (APUs) or specialized FPGAs to name a a few. Programs need to be designed to work on multiple platforms and be implemented with parallelism in mind that span over number of devices. Such scale is not addressed in this report though.

Currently, there are two significant industry players in the GPU market: NVIDIA and AMD. This is due to the fact that they are two remaining vendors of GPU architectures nowadays.

As a GPU programming pioneer and 5-years-long-runner NVIDIA has succeeded in providing means to program their proprietary hardware. Most of their efforts go into marketing their CUDA technology and the platforms that support it. From NVIDIA's perspective the main goal now is to bring GPU computing into the post-CUDA age. CUDA C and Fortran are the most widely used programming languages for GPU programming nowadays. However, as the underlying technology is proprietary to NVIDIA and a software model of GPU computing offered is relatively low-level, the use of CUDA today tends to be restricted to computer scientists. The average programmer can be discouraged by the number of new constructs to learn and the lack of abstraction. Researchers are mostly interested in ad-hoc boost in performance of their algorithms usually written in high-level languages. They seek automation in process of porting their codes to GPU platforms.

Meanwhile, AMD chose another approach and is busy implementing and supporting the latest specification of OpenCL standard.

Other companies like Intel try to catch up on them introducing their own specialized hardware - CPUs with embedded Hardware Graphics and technologies and providing implementation and support for the current standards like OpenCL.

## 2.2 OpenCL

OpenCL (Open Computing Language) is an open standard that defined and maintained by Khronos Group and was initially developed by Apple Inc. It is a vendor-independent framework that was from the beginning meant to enable the programmer to develop portable code that can be executed on heterogeneous platforms consisting of CPUs, GPUs and potentially other processing units like APUs or FPGAs. Thanks to that it form a real base in the age of heterogeneous computing. Currently, it has several adopters, among others Intel, AMD, NVIDIA and ARM Holdings.

OpenCL consists of a programming language based on C99 used to write kernels. These are basic units of executable code, C-like functions that are executed on devices that support OpenCL. It also comprise of a set of APIs. Runtime API is used to control kernel execution and manage the scheduling, computation and the memory resources. The

Platform Layer API defines the environment called a context on the platform at hand and creates a hardware abstraction layer over diverse computational resources. Parallel programming in kernels is supported on the level of task-based and data-based parallelism.

A compatible compiler like Visual C++ compiler, gcc, Nvidia's nvcc or Intel's compiler is used to compile a collection of kernels and other supporting functions into an OpenCL Program that is executed on the GPU. This is analogous to a dynamic library in C. An application has a queue that stores kernel execution instances in-order. The execution though might be in-order or out-of-order. The most basic unit of work on an OpenCL device is called a work item.

As OpenCL matures, lots of learning materials can be found in the Internet. In addition, there are numerous courses held at universities all over the world that choose OpenCL as a technology for teaching High Performance Scientific Computing. However, a significant sign of the maturity of some technology can be usually measured in the number of the published books available in the market. OpenCL has a few from which a book by Gaster et al. [1] seemed to be most helpful.

When it comes to the usage of OpenCL in industry, academic institutions, research labs and FPGA vendors are leading implementers. The OpenCL standard homepage[**?**] specifies a number of specialized products that use OpenCL. To name some more popular customer products, latest version of Adobe Photoshop CS6 uses OpenCL to accelerate the image operation features including Blur Gallery, Liquify, and Oil Paint. A different usage is a Winzip 16.5 that uses OpenCL to accelerate extraction and compression of the archives with AES encryption. Yet another software that use OpenCL API are GPU-enhanced functions in Wolfram Mathematica. A tool worth-mentioning is also OpenCL Studio, that integrates OpenCL and OpenGL into a single development environment for high performance computing and visualization. Kishonti Informatics implemented and CLBenchmark for measuring and comparing the processing power of different hardware architectures that use OpenCL 1.1.

There are different OpenCL Desktop Implementations by NVIDIA, AMD and Intel. NVIDIA supports OpenCL through CUDA SDK. CUDA toolchain provides limited support for compilation and profiling OpenCL code. AMD supports the programmers with its AMD APP SDK to allow for development on their platform. AMD APP (Accelerated Parallel Processing) SDK, formerly ATI Stream, is the OpenCL library for AMD graphics cards. It also provides Math Libraries, gDEBugger, APP Profiler and KernelAnalyzer. Intel supports OpenCL through its own Intel SDK for OpenCL Applications 2012 released recently.

### 2.2.1 Current specification

The current specification of the OpenCL API is 1.2 released on 15 November 2011. However, as each of the adopters implement the standard specification on their own, the scope and support for new version varies. What can be said is that all of them guarantee a support for the version 1.1 of standard. However, some of them like AMD competes to adopt new standard as it introduces some valaubale improvements to the OpenCL programming model. In particular, subdevice abstraction provides ways to partition a device into parts that can be treated as seperate device could in 1.1. Partitioning of devices gives more control over assignment of computation to compute units. Furthermore, custom devices and built-in kernels address the problem with embedded platforms like specialized FPGAs or non-programmable hardware with associated firmware (e.g. video encoder/decoders or DSPs). These cannot support OpenCL C directly, so built-in kernels can represent these hardware and firmware capabilities. Development closer to GPU touched the Image format of memory resources.

### 2.2.2 Comparison with CUDA

Comparisons between CUDA and OpenCL are inevitable. Both provide a general-purpose model for data parallelism and low-level access to hardware. Although OpenCL 1.0 was introduced in December 2008, a year and a half after the NVIDIA launched first CUDA, OpenCL still trails CUDA in popularity by a wide margin, especially with regard to HPC. However, it is said that if the OpenCL implementation is correctly tweaked to suit the target architecture, it should not perform worse than CUDA. The key feature of OpenCL is its portability as the memory and execution model are abstracted. That limits the optimization of OpenCL code. In CUDA, programmer can directly use GPU-specific technologies, due to the fact it is limited to NVIDIA hardware. Thanks to that CUDA provides more mature compiler optimisations and execution techniques such as more aggressive pragma unroll addition to loops. In OpenCL, programmer is required to add in the optimisations manually. However, as OpenCL toolkit matures differences between these two technologies will vanish.

Another point is that there is already a number of libraries supported on CUDA like CUBLAS, CUFFT or CUSPARSE. In comparison, OpenCL still lack high quality, open libraries, although there are recent developments such as open-source ViennaCL, AMD's clAmdBlas or clAmdFft libraries (or proprietary AccelerEyes ArrayFire for faster C, C++, Fortran, Python code that uses ArrayFire OpenCL API) that are a symptom that the situation might change in the nearest future. Moreove, ArrayFire made a comparison between two technologies.

### 2.2.3 WebCL

WebCL is a set of JavaScript binding to OpenCL that allows for parallel computation in web applications. JavaScript was never designed to exploit the multithreaded data-parallel computing capabilities available on GPUs. On 17 April 2012 Khronos Group released a WebCL working draft and the standard lack any serious implementation or support.

### 2.2.4 Future developments

Khronos Group Representatives in its presentations talk among other about OpenCL-HLM, high-level programming model that would introduce new language syntax to unify the host and device execution in order to increase usability and possibilities of optimization. At the same time, OpenCL-SPIR that stands for Standard Parallel Intermediate Representation aims to explore ways to provide target back-end for alternative high-level languages. It also says that Long-Term Core Roadmap is to "explore enhanced memory and execution model flexibility to catalyse and expose emerging hardware capabilities". Furthermore, it might be true that OpenCL needs a vendor that will boost its standardization, adaptation and marketing process as NVIDIA does for CUDA. AMD has a chance to take this part as they decided to support an open standard OpenCL instead of introducing their own technology. On one hand NVIDIA and Intel provide software handles to OpenCL for their respective hardware. On the other hand, altough both Intel and NVIDIA have signed on to OpenCL standard committee and so both technically support it, the performance benefits that they offer are pretty poor when compared to AMD's APUs and GPUs. Another downside to OpenCL, similar to the one with OpenGL, is that everyone in the standardization committee has a say in development of a standard. As a consequence that may lead to conflicts and blocking ideas that are another cause of slow progression of OpenCL development in future.

## 2.3 Other GPGPU Technologies

### 2.3.1 CUDA

The current version of NVIDIA's technology is 4.2. CUDA has a rich support from its vendor and is well adapted through out the industry. It has a highly-optimized libraries and rich toolchain. Checking Compute Capability of NVIDIA cards is the way to measure the advance of CUDA platform. Currently latest version is 3.0, which is implemented in latest Kepler GPUs that went into market earlier this year. In Decemeber last year NVIDIA made its nvcc compiler open-source that might be seen as movement towards open standard. CUDA has bindings to all the most popular programming languages.

### 2.3.2 OpenACC

The OpenACC API is a collection of compiler directives in fashion of OpenMP. It is a new parallel programming standard for accelerators developed by Cray, CAPS, Nvidia and PGI. Directives are used to specify loops and regions of code in programming languages like C, C++ or Fortran that are then offloaded by the OpenACC API-enabled compilers and runtimes from the host CPU to the attached accelerator. The resultant code is portable across other accelerators and multicore CPUs. In future, this standard may become a part of the OpenMP standard, as OpenACC was derived from work done within the OpenMP Working Group on Accelerators.

### 2.3.3 C++ AMP

C++ AMP (Accelerated Massive Parallelism) is meant to accelerate the C++ code by taking advantage of data-parallel hardware like GPUs. It has a set of features like multi-dimensional arrays, indexing, memory transfer, and tiling supported by a mathematical function library. Transfer between CPU and GPU can be controlled in OpenCL fashion. It is a library implemented on DirectX 11. Microsoft specified an open specification for implementing data parallelism directly in C++. It is targeted to developers with even no expertise in parallization. The code stays portable. Technology is available in latest version of Visual Studio 2012.

### 2.3.4 DirectCompute

Microsoft DirectCompute is an API that supports GPGPU programming on Windows platform. It is a compute shader, a programmable shader stage that expands Microsoft Direct3D 11, a part of Microsoft DirectX 11 API, beyond graphics programming. DirectCompute shares a range of computational interfaces with OpenCL and CUDA. DirectCompute is programmed by a language which is similar to HLSL that is a DirectX shader language.

### 2.3.5 OpenHMPP

Another example of a set of OpenMP-like directives that preserve legacy codes is OpenHMPP (Hybrid Multicore Parallel Programming), a programming standard for heterogeneous computing. The software is independent from hardwar , thus ready for future architectures. The directive-based model is based on works by CAPS (Compiler and Architecture for Embedded and Superscalar Processors) called HMPP. The main concept are the codelets, that are functions that are executed remotely on hardware. HMPP provides synchronous and asynchronous RPC. Like in OpenCL Memory model there are two address spaces: the host processor one and the HWA memory.

## 2.4 Current advances in GPU Architectures

As mentioned already, today we have two main viable and competitive GPU product lines. NVIDIA and AMD GPUs support a wide range of programming languages. GPUs also make their way to High Performance Computing Centres and supercomputers in academic, research and government labs. As of June 2012, 3 out of 10 Top500 supercomputers in the world were using NVIDIA Tesla Fermi GPUs. In addition, it seems like Tesla Fermi was a significant turning point, when it come to the number of GPU supercomputers in the Top500 list.

### 2.4.1 NVIDIA

#### Fermi

GeForce 400 and 500 series cards were the ones, where completely-rebuilt Fermi architecture was introduced. “The GPU featured up to 512 CUDA cores organized in 16 streaming multiprocessors of 32 cores each. It has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory”. With this iteration NVIDIA started to support Visual Studio and C++.

#### Tesla

The main difference between Fermi-based Tesla cards and the GeForce 500 (also Fermi) series is the unlocked double-precision floating-point performance. This is why these cards are targeted to high performance markets, where floating-point calculations on large scale are performed. It is said 1/2 of peak single-precision floating point performance in Tesla cards compared to 1/8 for GeForce cards. In addition, the Tesla cards have ECC-protected memory and are available in models with higher on-board memory (up to 6GB). So this why Tesla is for workloads where data reliability and overall performance are critical.

#### Kepler

Earlier this year NVIDIA has shipped cards with Kepler architecture. It also ships a family of TESLA GPUs based on the this architecture. Dynamic Parallelism is the most significant development that allows for regions of computation to be dynamically adjusted. It is based on the idea that the calculations are more fain grained in highly computational regions in excess of more coarse calculations in regions of the grid where there is not much computation involved. Moreover, Cuda 5 parallel programming model is planned to be widely available in the third quarter of 2012 on these GPUs.

### 2.4.2 AMD

#### APUs

APU stands for Accelerated Processing Unit and is a processing system with additional processing capabilities to accelrate given types of computations outsie of a CPU. They may embed a GPGPU, an FPGA or other specialized unit. AMD Fusion is a series of AMD's APUs. What might be interesting from perspective of a GPU programmer is that GPU in such chipset can access host CPU memory without going through a device driver as the memory will have a unified memory controller for both CPU and GPU. Current APUs support OpenCL in version 1.1, although AMD announced on its homepage AMD Accelerated Parallel Processing (APP) SDK with OpenCL 1.2 Support targeting APUs. It can be seen in introduction of Hetereogeneous Systems Architecture (HSA) that promises higher facilitation of OpenCL, CPU-GPU seamless cooperation and immediate port to AMD APUs.

#### GPUs

The latest GPUs by AMD are Northern Islands (HD 6xxx) series and Southern Islands (HD 7xxx) series. Some of the former and all the latter support OpenCL's latest specification, what is another argument for that AMD conentrates on support for OpenCL. The achitectures used are VLIW4, VLIW5 and GCN. More detailed overview of these architectures is enclosed in [3]. The most recent of workstation-oriented GPUs are FirePro cards announced recently. Surprisingly they do not support OpenCL even in 1.1 version. However, AMD in general seems to push its OpenCL strategy for GPU computing.

### 2.4.3 Intel

Intel, main CPU vendor, is going to release its Many Integrated Core (MIC) architecture for commercial use later this year. It is codenamed Knights Corner and the branding name for this processor will be Intel Xeon Phi. The project incorporates earlier work on Larrabee many core architecture. It will allow for floating point operations on 512-bit SIMD vector registers. Intel wants to leverage x86 legacy and provide architecture that can utilise existing parallelization software tools, among others OpenCL. It is a direct competitor for NVIDIA Tesla in HPC market. Another of Intel's products is released earlier this year 3rd generation Intel Core with Intel HD Graphics, that can be seen as an example of Intel's APU. It can be programmed with OpenCL through recently published Intel SDK for OpenCL Applications 2012.

## 2.5 Experiences with OpenCL

### 2.5.1 Developing OpenCL Code

Most of the development was done on Intel processor and NVIDIA Geforce card on Windows platform. Microsoft Visual Studio 2010 IDE was used for development.

During the development for GPU platform it is invaluable to have an access to a decent debugger that allows the programmer to debug kernels. During this project there was a the lack of debugging options. Such situation was due to the fact that the choice of debugger is dependant on the platform used. Some combinations will then be not supported as of now, when OpenCL is still in its infancy as a standard. So it was the case during the development process for this project as NVIDIA card was combined with OpenCL technology. Two working options for debugging tried were NVIDIA Nsight (currently in version 4.2) and AMD gDEBugger (currently in version 6.2).

The former supports CUDA code only. The tool is an application based on a client-server architecture. The Visual Studio Plugin serves as a client to the Monitor client process run on the machine. This allows for remote access to GPUs. Moreover, the breakpoints can be set inside the CUDA kernel code so as the current state of local variables, warps and working items will be presented. This valuable data is available through a set of info windows. Tool does not work with OpenCL code though, as it is not possible to set breakpoints in OpenCL code. Tool is distributed as a plugin to Visual Studio 2010.

The later works only with OpenCL code, but supports only the AMD devices. However, it stops on breakpoints on host. Thus it can be partially helpful during the development as it can be configured to stop on specific OpenCL functions and OpenCL errors. It also offers function call history and function properties that gives an overview into the state of paramters that OpenCL API is called with. Finally, there is also an explorer that enables programmer to browse a tree structure of OpenCL constructs used.

On the side, there is an Intel SDK that allows for checking if the code is ready to be run on Intel platform like CPU or new 3rd generation processors with hardware graphics embedded. An offline OpenCL compiler can be used to check the validity of kernel code.

Development on higher level of abstraction is possible through C++ wrapper API. However, this was not attempted in this project. Moreover, OpenCL API is ported in form of wrappers to popular programming languages like Java (JavaCL) or Python (PyOpenCL). Such projects surely will advance a wide adoption of OpenCL standard.

### 2.5.2 Profiling OpenCL Code

Because most of the development process for this project was done on the NVIDIA platform, some of its tools like NVIDIA Visual Profiler 4.2 or NVIDIA Nsight Profiler Visual Studio Edition 2.2 were researched. The latter can only be used with CUDA code, so it was useless for this project. The Visual Profiler was the only program that proved to be helpful as it provided detailed statistics about metrics and events of the application execution. Furthermore, it provides a timeline graph where memory transfers and kernel computations are marked in time. Finally, analysis results provide guidelines that programmer can use to optimize the code. This tool is distributed as part of CUDA Toolkit as of version 4.2. Another helpful resource provided by NVIDIA is its CUDA GPU Occupancy Calculator spreadsheet that allow easy calculation of multiprocessor occupancy of a GPU by a given CUDA kernel. This can be used to optimize OpenCL kernels on NVIDIA platform.

Another profiling tools from the competitor AMD are AMD APP Profiler and AMD KernelAnalyzer. Gaster et al. in their book[1] provide an overview and step-by-step description of how to use this tool. These tools were not used during this project, because the code was not developed on the AMD platform. This is an obvious requirement while using these tools. Tools are available for free in the AMD Developer Zone.

# 3 Navier-Stokes Solver

## 3.1 Computational fluid dynamics

Computational fluid dynamics (CFD) is a branch of fluid mechanics. It uses numerical methods and algorithms to solve and analyse problems that involve flows of fluids. Computationally-intensive calculations are performed on high performance computing platforms to simulate interactions between particles of liquids and gases with surfaces defined by boundary conditions. The more resources computing platforms has the more detailed simulation can be achieved. Until the advent of numerical scientific approach, the theoretical models could only be verified through practical experiments that were time and resource consuming and usually unrepeatable or even infeasible on larger scale. Such numerical simulations are used in many other scientific, engineering and industrial areas.

## 3.2 Discretization in CFD

The equations governing the fluid interaction need to be discretized before simulation, i.e. the results of these equations are only considered at a finite number of selected points. The results of these continuous equations are approximated at these points. From this follows that the more densely discretization points are spaced the more accurate the simulation of a problem is. However, high resolution of space by discrete points is extremely demanding in terms of memory and computation time. This is where GPUs come in handy as they allow for massively parallel computations at many discretization points at one time. Varying from problem to problem, solutions to discrete problems usually require execution of many nested loops and involve time-dependencies, nonlinearites and solution of large linear systems of equations. Various methods for solving those equations that involve multigrid, multilevel and multiscale methods are researched. Adaptive methods help accurately approximate the solution of continuous problem with minimal memory requirements. Furthermore, parallelization involves dividing the problem equally between computing cores by means of domain decomposition.

Many processes take place within different fluids. The interactions between different fluid particles as well as the forces between moving fluids and solid bodies at rest or vice versa need to be considered. A physical property of fluids known as *viscosity* is the source of the occurring forces. It generates frictional forces that act on the fluid initially in motion bringing it eventually to the rest in the absence of external forces. Another property of fluids is called *inertia* and specifies the resistance of any physical object to a change in its state of motion or rest. It is possible to simulate this property when a idealized layered model of fluid is assumed. Flows adhering to this idealization are called *laminar flows* as opposed to *turbulent flows*, which assumes that particles can mix between layers. The relative magnitude of these two properties is measured by dimensionless parameter called Reynolds number. Prandtl number is another parameter that is connect to the behaviour of fluid next to the boundary layers.

## 3.3 Description of Navier-Stokes Equations

Navier-Stokes equations form a fundamental mathematical model for almost all CFD problems. They treat laminar flows of viscous, incompressible fluids. A solution of the equations is called a velocity field that describes velocity of fluid at given point in space and time. For visualisation purposes though trajectories of position of a particles would be needed.

The flow of a fluid in a region $\Omega \subset \mathbb{R}^N$, where $(N = \{2,3\})$ throughout time $t \in [0, t_{end}]$ is characterized by the following quantities:

$$\begin{array}{ll} \vec{u} : \Omega \times [0, t_{end}] \to \mathbb{R}^N & \text{velocity field,} \\ p : \Omega \times [0, t_{end}] \to \mathbb{R} & \text{pressure,} \\ \varrho : \Omega \times [0, t_{end}] \to \mathbb{R} & \text{density.} \end{array}$$

Density changes in incompressible flows are negligible. Therefore the flow is described by a system of partial differential equations which dimensionless form is given by:

$$\begin{aligned} \frac{\partial}{\partial t}\vec{u} + (\vec{u} \cdot \nabla)\vec{u} + \nabla p &= \frac{1}{Re}\Delta\vec{u} + \vec{g} && \text{(momentum equation),} \\ div\,\vec{u} &= 0 && \text{(continuity equation),} \end{aligned} \tag{3.1}$$

where pressure is determined only up to am additive constant. The quantity $Re \in \mathbb{R}$ is the dimensionless *Reynolds number* and $\vec{g} \in \mathbb{R}^N$ denotes body forces such as gravity acting throughout the bulk of the fluid.[2]

In this project the two-dimensional case $(N = 2)$ is considered, so the equations 3.1 rewritten in component form look as follows:

momentum equations:

$$\frac{\partial u}{\partial t} + \frac{\partial p}{\partial x} = \frac{1}{Re}(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}) - \frac{\partial (u^2)}{\partial x} - \frac{\partial uv}{\partial y} + g_x, \tag{3.2a}$$

(3.2b) $$\frac{\partial v}{\partial t} + \frac{\partial p}{\partial y} = \frac{1}{Re}(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}) - \frac{\partial uv}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y;$$

continuity equation:

(3.2c) $$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0,$$

where $\overrightarrow{x} = (x, y)^T$, $\overrightarrow{u} = (u, v)^T$ and $\overrightarrow{g} = (g_x, g_y)^T$.[2] Initially, initial conditions for $u$ and $v$ need to satisfy 3.2c. Fo simplicity, energy equation was not mentioned as Boundary conditions are dependant on modelled problem. Different types are as follows:

1. *No-slip condition:* No fluid penetrates the boundary and the fluid is at rest there.
2. *Free-slip condition:* No fluid penetrates the boundary. There are no frictional losses at the boundary.
3. *Inflow condition:* Both velocity components are given.
4. *Outflow condition:* Neither velocity component changes in the direction normal to the boundary.
5. *Periodic boundary condition:* For periodic problems with a period in one coordinate direction, the computations are restricted to one period interval. The velocities and pressure must then coincide at the left and right boundaries.

This is just the essential minimum. The way these equations are derived and more mathematically detailed descriptions of equations and boundary conditions can be found in [2].

## 3.4 Discretization of Navier-Stokes Equations

Discretization is used in the numerical solution of differential equations to reduce those differential equations to a system of algebraic equations. One of the methods to achieve it is the *finite difference method* (FD). Detailed description on how to derive discretized Navier-Stokes equations may be found in [2].

Discretization in two dimensions is carried out In this project. In two dimensions, the the problem is restricted in size to a rectangular region

$$\Omega := [0, a] \times [0, b] \in \mathbb{R}^2$$

on which a grid is introduced. The gird is divided into $i_{max}$ cells of equal size in the $x$-direction and $j_{max}$ cells in the $y$-direction, resulting in grid lines that are spaced apart at a distance

$$\delta x := \frac{a}{i_{max}} \quad \text{and} \quad \delta y := \frac{b}{j_{max}}.$$

When solving the Navier-Stokes equations, the region $\Omega$ is often discretized using a staggered grid, in which the different unknown variables are not located at the same grid points. Scalar variables like pressure $p$ is located in the cell center. The velocity $u$ is located in the midpoints of the vertical cell edges and the velocity $v$ in the midpoints of the horizontal cell edges (see Figure 3.1).

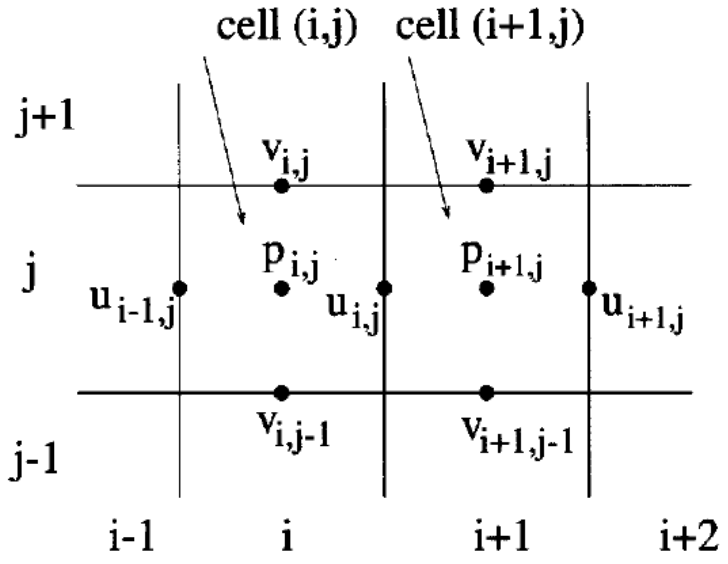

**Figure 3.1: Staggered grid.** Staggered grid for cell $(i, j)$ and $(i + 1, j)$.[2]

As a result, the discrete values of $u$, $v$, and $p$ are located on three separate grids, each shifted by half of a grid spacing to the bottom, to the left, and to the lower left, respectively. Therefore vertical boundaries do not have $v$-values and horizontal boundaries do not have $u$-values. This forces the introduction of extra boundary strips of grid cells (see Figure 3.2). The boundary conditions may be then applied by averaging the nearest grid points and this is done on every side.

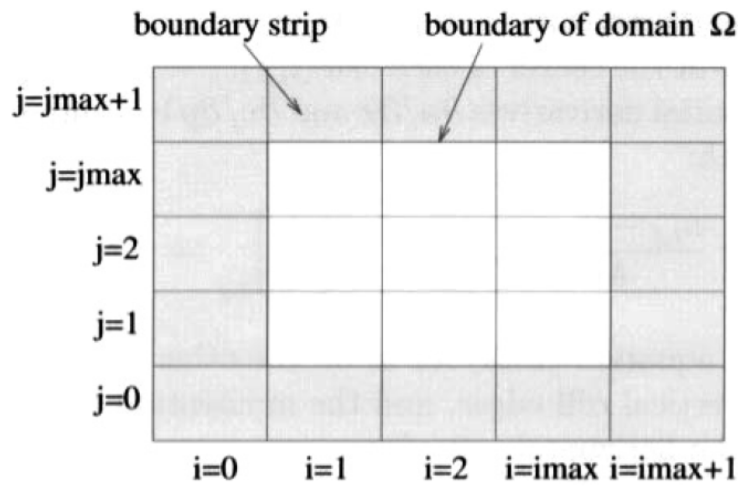

**Figure 3.2: Domain with boundary cells.** Domain with boundary cells.[2]

Such staggered setup of the unknowns prevents possible pressure oscillations which could occur if all three unknown functions $u$, $v$ and $p$ at the same grid points. Another alternatives to staggered grid may be e.g. colocated grid.

Just to illustrate the method, e.g. the continuity equation 3.2c is discretized at the center of each cell $(i, j)$ by replacing the spatial derivatives $\frac{\partial u}{\partial x}$ and $\frac{\partial v}{\partial y}$ by centered differences using half the mesh width:

$$\left[\frac{\partial u}{\partial x}\right]_{i,j} := \frac{u_{i,j} - u_{i-1,j}}{\delta x}, \quad \left[\frac{\partial v}{\partial y}\right]_{i,j} := \frac{v_{i,j} - v_{i,j-1}}{\delta y}. \tag{3.3}$$

For momentum equation 3.2a for $u$ is discretized at the midpoints of the vertical cell edges, and the momentum equation 3.2b for $v$ at the midpoints of the horizontal cell edges and the values required for calculation of some of the terms are shown in Figure 3.3. The disretization process of these equations is dropped here as it may be seen in [2] as the project is fully based on these process. The process may pose some difficulties with some of the terms.

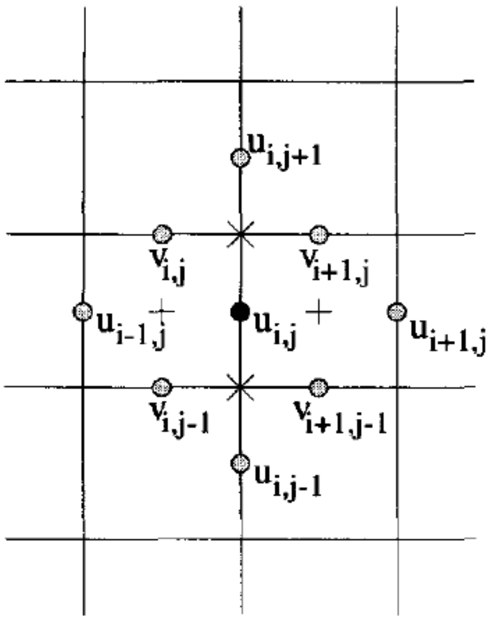

**Figure 3.3: Values required for the discretization of the $u$-momentum equation.** Values required for the discretization of the $u$-momentum equation.[2]

Moreover, following [2], the discretization of the terms of the momentum equation 3.2a for $u$ involves $u$-values on the boundary for $i \in \{1, i_{max} - 1\}$. Moreover, for $j \in \{1, j_{max} - 1\}$, $v$-values lying on the boundary are required as well as additional $u$-values lying outside the domain $\Omega$. Similarly, boundary values of $v$ are required in the discretization of the terms of the momentum equation 3.2b for $v$. In total, we require the values

$$\begin{array}{lll} u_{0,j}, & u_{i_{max},j} & j = 1, \dots, j_{max}, \\ v_{i,0}, & v_{i,j_{max}} & i = 1, \dots, i_{max}, \end{array}$$

on the boundary as well as the values

$$\begin{array}{lll} u_{i,0}, & u_{i,j_{max}+1} & i = 1, \dots, j_{max}, \\ v_{0,j}, & v_{i_{max}+1,j} & j = 1, \dots, j_{max}, \end{array}$$

outside the domain $\Omega$. These velocity values are obtained from a discretization of the boundary conditions of the continuous problem.

These values are differently set depending on the type of boundary condition on given boundary. The description may be once more seen in [2].

What is left to discritize are the time derivatives $\frac{\partial u}{\partial t}$ and $\frac{\partial v}{\partial t}$. The time interval $[0, t_{end}]$ is subdivided into equal subintervals $n\delta t$, $(n+1)\delta t$, where $n = 0, \dots, \frac{t_{end}}{\delta t - 1}$. This means that values $u$, $v$ and $p$ are considered only at times $n\delta t$. The time derivatives are discretized using Euler's method.

## 3.5 The Algorithm

The numerical method to solve the equations through discretization consists of a time-stepping loop. In summary, the $(n+l)$st time step consists of the following parts:

**Step 1:** Compute $F^{(n)}, G^{(n)}$ 3.4 from the velocities $u^{(n)}, v^{(n)}$.

**Step 2:** Solve the Poisson equation 3.6 for the pressure $p^{(n+l)}$.

**Step 3:** Compute the new velocity field $(u^{(n+1)}, v^{(n+1)})^T$ using 3.5 with the pressure values $p^{(n+l)}$ computed in Step 2.

If the time discretization of terms $\dfrac{\partial u}{\partial t}$ and $\dfrac{\partial v}{\partial t}$ in the momentum equations 3.2a and 3.2b is computed, two abbreviations are introduced for simplicity:

$$\begin{aligned} F &:= u^{(n)} + \delta t \left[ \frac{1}{Re}\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{\partial (u^2)}{\partial x} - \frac{\partial uv}{\partial y} + g_x \right], \\ G &:= v^{(n)} + \delta t \left[ \frac{1}{Re}\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{\partial uv}{\partial x} - \frac{\partial (v^2)}{\partial y} + g_y \right]. \end{aligned} \tag{3.4}$$

This gives us the time discretization of the momentum equations 3.2a and 3.2b:

$$\begin{aligned} u^{(n)} &= F^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial x}, \\ v^{(n)} &= G^{(n)} - \delta t \frac{\partial p^{(n+1)}}{\partial y}. \end{aligned} \tag{3.5}$$

Discretization is explicit in the velocities and implicit in the pressure, i.e. the velocity field at time $t_{n+1}$ can be computed once the corresponding pressure is known. Pressure is determined by substituting the relation 3.5 into the continuity equation 3.2c, which after some rearranging, becomes a Poisson equation for pressure $p^{(n+l)}$ a time $t_{n+1}$:

$$\frac{\partial^2 p^{(n+l)}}{\partial x^2} + \frac{\partial^2 p^{(n+l)}}{\partial y^2} = \frac{1}{\delta t}\left(\frac{\partial F^{(n)}}{\partial x} + \frac{\partial G^{(n)}}{\partial y}\right). \tag{3.6}$$

Morover, boundary values for pressure are required in solution of the pressure Poisson equation in Step 2. Furhermore, to obtain the fully discrete momentum equations, the spatial derivatives have to be discretized too.

Fully discrete Poisson equation is obtained along the discretization of the Laplacian and the discrete quantities introduced earlier. Boundary conditions also need to be taken into consideration. The report drops all the details connected with deriving the exact final form of the Poisson equation as [2] describe the process in steps. The result of this derivation is a large, sparse linear system of equations that come form discretization of PDEs. Direct methods as Gaussian elimination are not efficient in terms of computer time or storage, so an iterative method is used. Gauss-Seidel method is a classic example of such methods. Following Griebel et al., the project uses its improved variant called successive overelaxation (SOR) method. The iteration is terminated when the maximal number of steps is reached or when the of the residual is under some absolute tolerance. An $L^2$-norm is used. This method yields some other problems and needs additional fixes during the implementation.

One of the problems is that convergance behaviour deteriorates while spacing between the grid lines is decreased. It results in an increased number of iteration steps to reduce iteration error below a given tolerance. Such behaviour was observed in this project.

The multigrid methods are another class of iteratice methods, where number of iteration steps is independant of the number of unknowns. They solve the problem of increase in the number of iteration steps through performing computation on successively coarser grids obtained by doubling the spacing between the grid lines of each previous grid.

## 3.6 Problems in CFD

$< 1000$

## 3.7 Other attempts of implementing the solver

grid The most common form for a stream to take in GPGPU is a 2D grid because this fits naturally with the rendering model built into GPUs.

## 3.8 Examples of CFD projects using GPUs and OpenCL

A lot of knowledge can be gathered just looking at other projects being implemented using given technology on given platform. As the project The works of Bednarz et al. are worth mentioning. His team implemented Computation Fluid Dynamics solvers with OpenCL. Another similar work was done by Zaspel from research group of Griebel. He was using CUDA. Another project could be

# 4 Design and Implementation

## 4.1 Design

A set of kernels

Modularity

All code should be executed on GPU for valid comparison.

## 4.2 Implementation

### 4.2.1 Developing with OpenCL

A simplest standard OpenCL pipeline would consist of following steps:

1.
2.

find platform find device create context and queue build program write device buffers execute kernel in range read device buffers

Code in project is based on structure from Griebels code[2]. Least as possible was changed in structure as to keep it comparable with the original code.

Code is written in C and OpenCL C.

Two-dimensional arrays in original One-dimensional arrays for OpenCL code. 1D arrays cannot be passed to OpenCL kernels.

Worksize

Implementation consists of sets of kernels as

Griebels cod intrinsics: Memory allocation

Code is portable and works under both on Windows and Unix platforms. Tested on Windows 7 and Linux clusters. Compilers used are nvcc and Visual Studio on Windows and gcc on Linux.

Table Computer specs.

Table

### 4.2.2 Functions

Map

The map operation simply applies the given function (the kernel) to every element in the stream. A simple example is multiplying each value in the stream by a constant (increasing the brightness of an image). The map operation is simple to implement on the GPU. The programmer generates a fragment for each pixel on screen and applies a fragment program to each one. The result stream of the same size is stored in the output buffer.

Reduce

Some computations require calculating a smaller stream (possibly a stream of only 1 element) from a larger stream. This is called a reduction of the stream. Generally a reduction can be accomplished in multiple steps. The results from the prior step are used as the input for the current step and the range over which the operation is applied is reduced until only one stream element remains.

Gather

The fragment processor is able to read textures in a random access fashion, so it can gather information from any grid cell, or multiple grid cells, as desired.

### 4.2.3 Stencil computation

### 4.2.4 Benchmarking code

Code is timed with standard library time.h clock function. No OpenCL timer code because on CPU. Code timed with buffer allocation and memory access and without.

### 4.2.5 Visualisation code

Matlab scripts to visualise code. Tests.

### 4.2.6 Naive kernels

Straightforward port of functions to kernels. Getting rid of for loops. Ensuring that boundaries are not crossed.

### 4.2.7 Shared memory kernels

### 4.2.8 Development

Production code should, however, systematically check the error code returned by each API call and check for failures in kernel launches (or groups of kernel launches in the case of concurrent kernels)

# 5 Performance Analysis

Synchronization between work-items is possible only within workgroups using barriers and memory fences. It is not possible to synchronize workitems that are in different workgroups. This would hinder data parallelism.

Memory management is explicit, so a programmer moves data from host to global memory of device first using OpenCL constructs. Then it can be moved to local memory. To read the results on the host the process has to be reverted.

Workgroup size for a given algorithm should be selecten to be an even multiple of the width of the hardware scheduling units.

## 5.1 Computed results

Theoretic bandwidth vs. Effective bandwidth

Trap of premature optimization

Visual Profiler Recommendations

Bandwidth

Data Transfer Between Host and Device

PCIe

coalescing global memory accesses

compute capability 2.x

Branching and Divergence Avoid different execution paths within the same warp.

vector operations, vector variables

built-in math functions, fast optimized library

Syncrhonization on Global and Local barriers

Synchronization between kernels in the queue. All the kernels are waiting for another to finish to work on nits results. One device - one queue.

CPU code in VIsual studio 2010 basic release build optimizations on both CPU execution and CPU control parts of GPU execution

Asynchronous reading of memory $CL_FALSE$ instead of $CL_TRUE$

allocating memory withou copyting from host pointer

No print options in the loop

Workgroup size

Size of the grid

Kernels are executed asynchronously enquing memory resources asynchronously using events to track the execution status

# 6 Conclusions

## 6.1 General

The project's goal was to get acquainted with doing computations on GPU.

No prior experience with CFD codes and optimizations of numerical methods. If had any could come up with implementation of other numerical method that could be easier to parallelize.

No prior experience with OpenCL (in most parts the same as CUDA, but lack tools)

Lot of problems with constructing even the naive kernel as to get the maximum benefit from parallelizing code on GPU, focus should be first drawn to find ways to parallelize sequential code. Code was thus looked as subblocks of same computations. Most data parallelism should be found.

Problems with bandwidth and minizming transfer between the host an the device, because it was not directly possible to create a kernel from many parts of the algorithm. Examples were researched. Stencil code implementations were found.

To watch the results of minimization in the use of global memory, some version of kernels were implemented. Prefer shared memory access where possible.

Float and double precison.

Other platforms

Different sizes of arrays with regular step

NVIDIA Profiler used

## 6.2 Improvements

Another type of memory object supported by OpenCL could be used and tested, namely images. They are optimized for two-dimensional access patterns. Such access was used in this project as grids of cells are two-dimensional. The challenge would be to learn how to work with this type of memory object as the data stored in it is accessible only through specialized access functions. In comparison, the buffer objects that were used map directly to the standard array representation that programmers are used to when using C programming language.

## 6.3 Future Work

Solution could be simulated on Tegra chip on Android Platform to check the performance of computation on yet another type of platform. This time a mobile GPU would be used. Such

# References

[1] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Elsevier /Morgan Kaufmann, San Francisco, CA, USA, 2011. 2.2, 2.5.2

[2] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical Simulation in Fluid Dynamics: A Practical Introduction*. Siam Monographs on Mathematical Modeling and Computation. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998. 1, 3.3, 3.3, 3.3, 3.4, 3.1, 3.2, 3.4, 3.3, 3.4, 3.5, 4.2.1

[3] Allan Svejstrup Nielsen, Allan Peter Engsig-Karup, and Bernd Dammann. Parallel programming using opencl on modern architectures. Technical report, Technical University of Denmark, 2012. 2.4.2

# List of Appendices

# A Appendix: Source Code

# B Appendix: Platforms and devices

## C Appendix: Visualisations