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.

August 9, 2012

DTU Informatics
Department of Informatics and Mathematical Modelling

Abstract

This report is intended as a summary of project work done for individual special course "A GPU-accelerated Navier-Stokes Solver using OpenCL". It describes the design and implementation of a CFD solver targeted for a GPU platform. The solution was implemented in OpenCL on basis of legacy code, optimized in some extent. The project was tested and analyzed.

Furthermore, essential theoretical details behind the solver are summarized. Report is also a guide over current state of GPGPU programming in general and OpenCL technology in particular. It may serve as brief handbook of good practices as the report consists of description of solutions to challanges that arose during development.

Although NVIDIA was a primary development platform and the report is mainly based on experiences with it, the content can be extended to wider scope of GPGPU programming. The report is concluded with general outcomes, enumeration of possible improvements and proposals for future work.

Contents

List of Figures

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 be based on sequential implementation by Griebel et al.[23]. 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 defining the problem enviorment as well as other factors. 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 presents the Navier-Stokes equations and talk briefly about the numerical method, which is used to discretize them. Furthermore, the algorithm is described here as well as similar

work in the discipline. 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 and good practices are proposed. Section 5 consists of the results and performance analysis backed up graphical data. The good practices are verified here. The report is concluded with Section 6, where general findings are summarized as well as 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 not long ago were 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 through 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 an has already hit the mainstream.

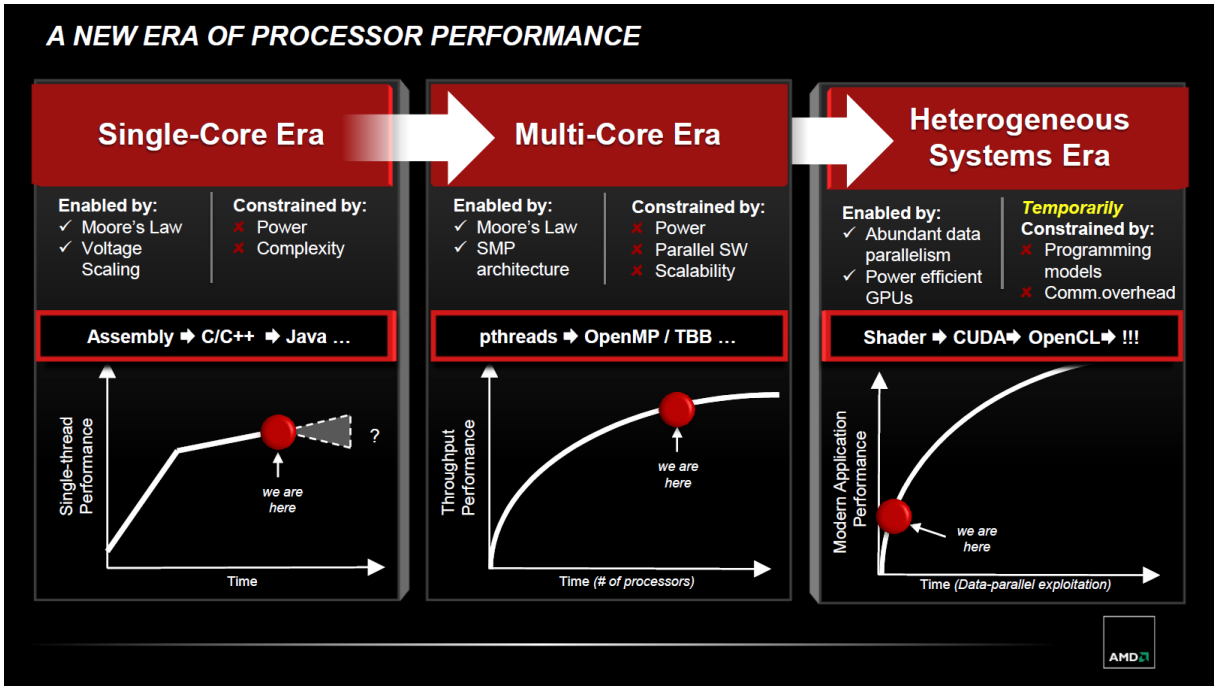

Figure 2.1: A new era of processor performance. A new era of processor performance[21]

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 their proprietary 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. OpenCL is the only technology that can be used with AMD GPU hardware.

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. The execution and memory model can be mapped to wide range of architectures 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.[31]

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.[30]

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. [20] 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[32] specifies a number of specialized products that use OpenCL. To name some more popular customer products, latest version of Adobe Photoshop CS6 and Premiere CS6[3] 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.[5, 6] A tool worth-mentioning is also OpenCL Studio[55], that integrates OpenCL and OpenGL into a single development environment for high performance computing and visualization. Kishonti Informatics implemented and CLBenchmark[25] for measuring and comparing the processing power of different hardware architectures that use OpenCL 1.1.

There are different OpenCL Desktop Implementations by NVIDIA[46], AMD[7] and Intel[29]. NVIDIA supports OpenCL through its normal drivers and CUDA SDK. CUDA toolchain provides limited support for compilation and profiling OpenCL code. On NVIDIA's CUDA-enabled GPUs OpenCL compiles into PTX ISA. Profiling of the signals and instrumentation is possible.

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 valuable improvements to the OpenCL programming model. In particular, subdevice abstraction provides ways to partition a device into parts that can be treated as separate 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.[30, 34]

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 and academic world.[59] 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.[51, 58]

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:

- ViennaCL library for linear algebra routines[56]
- GATLAS (GPU Automatically Tuned Linear Algebra Software)[37]
- AMD's clAmdBlas or clAmdFft libraries[10]
- clpp - OpenCL Data Parallel Primitives Library[17]
- 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. Moreover, ArrayFire made a comparison between two technologies.[2] As of now, most research projects seem to be implemented in CUDA. On the other hand, many projects like Par4All[53] head on in different direction providing the automation of porting the legacy code to GPU technologies. It generates both OpenCL and CUDA code.

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.[35]

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".[30] 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, although 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 possible slow progression of OpenCL development in future, although in this moment development of next version of specification is underway.[21]

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. "The Compute Capability of a device is defined by a major revision number and a minor revision number. Devices with the same major revision number are of the same core architecture. The major revision number is 2 for devices based on the Fermi architecture, and 1 for devices based on the Tesla architecture."[47] In Decemeber last year NVIDIA made its nvcc compiler open-source that might be seen as slight movement towards open standard, which still seems rather probable.[57] 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.[50, 61] 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.[38, 39] It has a set of features like multidimensional 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.[40] 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.[16, 52] The software is independent from hardware , 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.[48, 49] As of June 2012, 3 out of 10 Top500 supercomputers in the world were using NVIDIA Tesla Fermi GPUs.[1] 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.

In general, heterogeneous system architectures will head in direction of architectural integration unify address space between CPU and GPU to achieve fully coherent memory

between them. Physical like in current APUs will mean that CPU and GPUs will be integrated into one chip with a unified memory controller and common manufacturing technology. [21] GPU will act as a co-processor.

2.4.1 NVIDIA

Fermi

GeForce 400 and 500 series cards were the ones, where completely-rebuilt Fermi architecture was introduced.[42] "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.[43] 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.[24, 60]

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 Heterogeneous Systems Architecture (HSA) that promises

higher facilitation of OpenCL, CPU-GPU seamless cooperation and immediate port to AMD APUs.

GPUs

The latest graphic cards by AMD are Northern Islands (HD 6xxx) series and Southern Islands (HD 7xxx) series. Some of the former (HD 63xx, 64xx, and 69xx models) and all the latter support OpenCL's latest 1.2 specification. That is another argument that shows AMD concentrates on supporting OpenCL. The architectures used currently are as follows. The older VLIW5 (5-way VLIW) was used in all models up to HD 6870 in Northern Islands and up to HD76xx in Souther Islands series. More recent VLIW4 (4-way VLIW) architecture was used in HD 69xx models. However, the latest generations of AMD GPUs are already based on new GCN (Graphics Core Next) architecture, where the models HD 77xx-79xx shipped in the beginning of the year are based on it. More detailed overview of all these architectures and the enclosed comparisons may be found in [41]. Moreover, 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.[27] It is codenamed Knights Corner and the branding name for this processor will be Intel Xeon Phi.[54] 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.[28]

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).[8, 45]

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 parameters 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.[33] 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.[45] 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[4] and AMD KernelAnalyzer[9]. Gaster et al. in their book[20] 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 following sections are mainly based on the material presented in Chapter 2 and 3 in [23] as the solution from this book was imitated in this project.

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:

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

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:

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

where pressure is determined only up to am additive constant. The quantity $Re \in \mathbb{R}$ is the dimensionless *Reynolds number* and $\overrightarrow{g} \in \mathbb{R}^N$ denotes body forces such as gravity acting throughout the bulk of the fluid.[23]

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:

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

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

continuity equation:

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

where $\overrightarrow{x} = (x, y)^T$, $\overrightarrow{u} = (u, v)^T$ and $\overrightarrow{g} = (g_x, g_y)^T$.[23] 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 [23].

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 [23].

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 Ω 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).

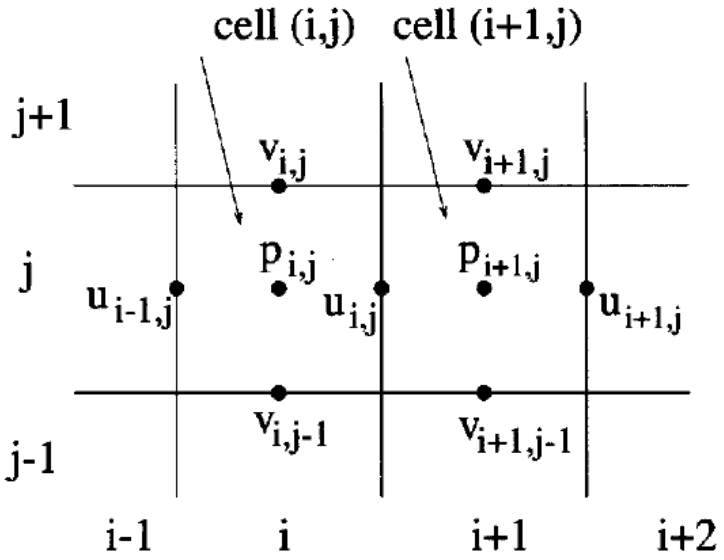

Figure 3.1: Staggered grid. Staggered grid for cell (i, j) and $(i + 1, j)$.[23]

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.

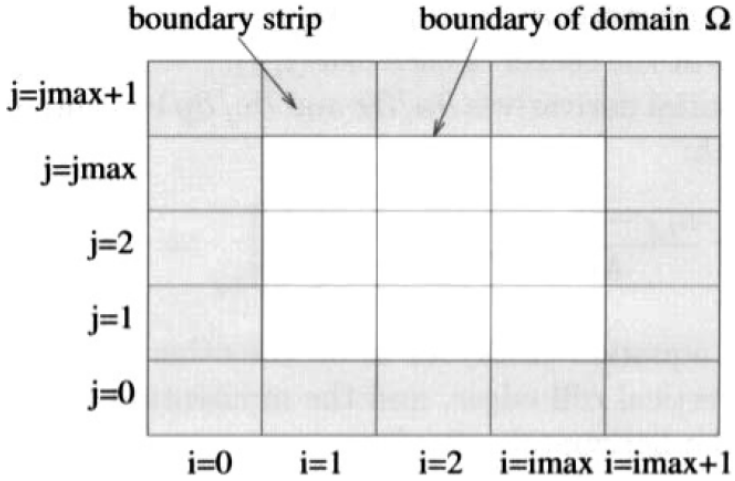

Figure 3.2: Domain with boundary cells. Domain with boundary cells.[23]

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. collocated 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 centred 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 [23] as the project is fully based on these process. The process may pose some difficulties with some of the terms.

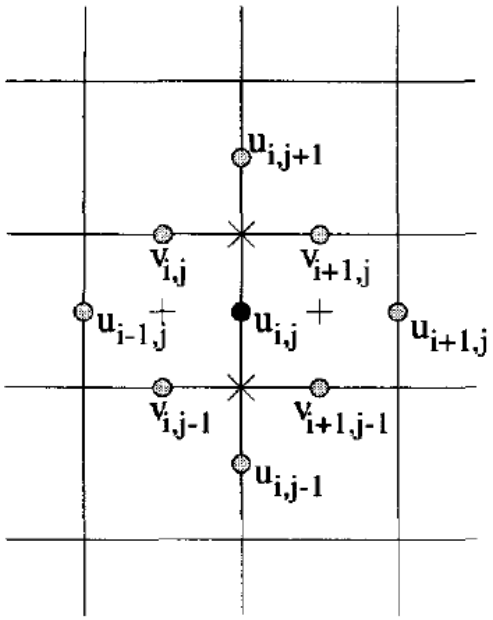

Figure 3.3: Values required for the discretization of the u-momentum equation. Values required for the discretization of the u-momentum equation.[23]

Moreover, following [23], 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 Ω. 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

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

on the boundary as well as the values

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

outside the domain Ω. 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 [23].

What is left to discritize are the time derivatives $\dfrac{\partial u}{\partial t}$ and $\dfrac{\partial v}{\partial t}$. The time interval $[0, t_{end}]$ is subdivided into equal subintervals $n\delta t, (n+1)\delta t$, where $n = 0, \ldots, \dfrac{t_{end}}{\delta t - 1}$. 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 $\frac{\partial u}{\partial t}$ and $\frac{\partial v}{\partial t}$ 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 [23] 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 overrelaxation* (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. In addition, stability conditions need to be introduced to ensure stability of numerical algorithm and avoid generation of oscillations.

One of the problems is that convergence 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 iterative methods, where number of iteration steps is independent 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 Examples of problems

An example of a simple CFD problem that this algorithm could solve is a *lid-driven cavity problem*. A physical configuration for simulation of driven cavity flow is a square contained filled with a fluid. The lid of the container moves at a given constant velocity. This sets the fluid in motion. No-slip conditions are set on three remaining boundaries (see Figure 3.4).

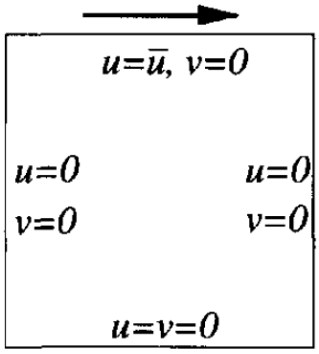

Figure 3.4: Driven cavity flow, problem configuration Driven cavity flow, problem configuration.[23]

Along the upper boundary the velocity u in the x-direction is not set to zero, but given some value. In the beginning of simulation the lid velocity is then set to move, but the

fluid is at rest. The formation of the large primary eddy and the first counter-rotating secondary eddy in the lower right corner can be observed. There is also a counter-rotating eddy in the lower left corner, but it takes more time to develop. Furthermore, different Reynolds number changed the number of eddies created, e.g. $Re = 0$ yields an infinite sequence of eddies and countereddies of exponentially decreasing size given infinite time. However such solution can be only found analytically. The flow in this problem finally reaches steady state. Other problems are those where boundary conditions change over time, so the steady state solution is never reached.

Another type of the problem with a fixed given domain Ω may be a flow over a backward-facing step. A physical configuration is a straight channel that at some point suddenly widens on one of the sides. Different example may be a flow past an obstacle that is immersed on the way of the fluid flow in a channel.

There are alos problems where the shape of the domain that consists of fluid changes over time. A description of *free boundary value problem* as they are called needs to be updated with initial configuration of the domain and the conditions that hold along the free boundaries. The examples of such behaviour in nature might the waves on the ocean or a falling raindrop hitting the surface of water container. It also exists when two or more different fluids mix.

3.7 Examples of CFD projects using GPUs

First of all, work by Bednarz et al. is worth mentioning. His team implemented Computation Fluid Dynamics solvers using OpenCL and simulated them on GPUs. Similarly to this project, they also used staggered grid (3D) and Finite Difference method, but to solve pressure and velocity components through mutual iteratin they used HSMAC (Highly Simplified Marker and Cell) method. Solution si sought with the Newton-Raphson method in stepwise way. They simulated some common CFD cases like Lid Driven Cavity, Natural or Magneto-thermal Convection or implemented BFC (Boundary Fitted Coordinates) for solving Navier-Stokes equations on complex geometries. They claim that such implementation is possible on GPUs using OpenCL and prove it with their results that show speedup in computation.[11, 12]

Another similar work was done by Zaspel from research group of Griebel. This research seems to be a direct continuation of what was done in the earlier project by Griebel et al. that was summarized in [23] which in turn is a basis for this project. Their NaSt3DGPF is a 3D grid-based fluid solver for the non-stationary two-phase incompressible Navier-Stokes equations implemented in CUDA. They claim that he ported the previous solver to GPU. FD solver is based on Chorin's pressure projection approach - the same as in their earlier project. Time discretisations are done using Runge-Kutta 3rd and Adams-Bashforth. Poisson equation is solved using Jacobi-preconditioned conjugate gradient. Complex geometries with different boundary conditions are solved and domain decomposition for multi-GPU MPI parallelization is used. This project and work done by Griebel's research group pose a significant motivation that CFD simulations should be ported to GPU and

optimized. It also shows that an initial choice of numerical method must be made in GPU in mind.[22, 62, 63, 64]

Finally, a Turbostream project that originates from research work by Brandvik and Pullan is yet another project that shows a power of GPU-based CFD solvers. The project's motivation was to simulate the flow of air through turbomachines. The codebase is based on TBLOCK from John Denton. Stencil operation are needed for that, so they constructed an optimized software framework for stencil applications. Sub-block implementation was done after Datta et al.[18, 19] They claim that hand-coding each stencil is not feasible, so kernel generator transforms a high-level definition of a stencil into optimised low-level source code. This is another example of automation in development of GPU-based codes. Code consists of 20 stencil kernels and definitions of those stencils are coded in Python. On NVIDIA GT200 this code yields over 10x speedup. In 2009, they began providing access to the solver to other organisations and they claim that Turbostream is now used for both research and design work by some of the largest turbomachinery manufacturers in the world.[13, 14, 15, 64]

4 Design and Implementation

4.1 Design

The design of the code is based on the implementation by Griebel et al. that is presented in their book[23]. Description of most of the functions is featured in the book, so here only the essential changes to functions will be described.

Numerical methods at most and especially iterative methods map into programming languages in form of loops. As long as consequent iterations are not dependant on one another, programmer can strive for data-parallelism on GPU. Nested loops in numerical methods for 2D problems iterate usually over time and space of the domain. 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. The base code already simulated the code in 2D. As a side note, the allocated memory in Griebel's code was for verbosity allocated in two-dimensional array. Such abstraction of memory is not supported by OpenCL C, so one-dimensional access had to used.

However, boundary conditions apply only to a small subset of cells in the grid. These are checked using if-conditions that are mapped into branches by compiler. In comparison to GPUs, CPUs have many mechanisms like dynamic branch prediction embedded in the chip. This is because a significant part of the silicon is devoted to control units as opposed to a paradigm of GPU architecture, where most of the space is taken by ALU performing computation. Therefore code that is full of nested conditions, but do not expose much, iteration will not expose good performance on GPU. Such code is better left to be executed on CPU.

A set of kernels was implemented matching every function in Griebel's code, so the code in project is based on structure. Griebel et al. in their book claim that they aim for modularity of code. Least as possible was changed in structure as to keep it comparable with the original code.

Griebel's code is completely sequential. However, to increase parallelism that can be used on GPU the algorithm needs to be run on single data units as independently as possible.

4.1.1 General considerations

There can be different types of kernels in terms of how the kernel treats data. The most simple kernel is the map kernel that maps a given function to every data object and executes it. The map operation is straightforward to implement on the GPU as this is a pure data parallelism. On the GPU performance of such a kernel is limited only be the size of workgroup that is mapped to warps or waves that are in turn mapped to a separate sequential multiprocessors or cores. After a fragment of data grid is processed, the result is saved in the output buffer of the same size as the input buffer.

Other type of a kernel is a reduction kernel that produce a result that is considerably smaller than the original input. Generally a reduction can be accomplished in multiple steps. For instance for a resultant output of size 1, 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.

There are also gather kernels that access the input buffer in random fashion gathering values from any grid cell or a multiple of grid cells at a time.

Another type of kernel is a stencil kernel, which access a fixed number of neighbouring cells and computes its value on their basis. This project is mainly based on such kernels.

At higher level, application should assign to each processor the type of work it does best, so serial workloads should stay on host and parallel workloads should be put on device.

For NVIDIA devices of compute capability 2.x and higher, multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using streams to enable enough kernels to execute concurrently.[47]

For two-dimensional arrays like in project, fully coalesced accesses, both the width of the thread block and the width of the array must be a multiple of the warp size “In particular, this means that an array whose width is not a multiple of this size will be accessed much more efficiently if it is actually allocated with a width rounded up to the closest multiple of this size and its rows padded accordingly”[47]

4.1.2 Structure and functions

The original code is a structured around a main time loop that processes consequent steps of simulation as seen on Figure 4.1. The algorithm sweeps over the grid and updates the cells one after another.

Set $t := 0$, $n := 0$
Assign initial values to u, v, p
While $t < t_{\text{end}}$
    Select δt (according to (3.50) if stepsize control is used)
    Set boundary values for u and v
    Compute $F^{(n)}$ and $G^{(n)}$ according to (3.36),(3.37)
    Compute the right-hand side of the pressure equation (3.38)
    Set $it := 0$
    While $it < it_{\max}$ and $\|r^{\text{it}}\| > eps$ (resp., $\|r^{\text{it}}\| > eps\,\|p^0\|$)
        Perform an SOR cycle according to (3.44)
        Compute the residual norm for the pressure equation $\|r^{\text{it}}\|$
        $it := it + 1$
    Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to (3.34),(3.35)
    $t := t + \delta t$
    $n := n + 1$

Figure 4.1: Application's algorithm Application's algorithm as in [23], references invalid

From implementation level, except assignments and condition loops above, every step in algorithm is treated as a separate function.

4.2 Implementing OpenCL on NVIDIA platform

Implementation of OpenCL may vary from platform to platform. Usually the underlying architecture needs to be taken into consideration, when optimizing code for performance. Many techniques are already well known best practices and are proved to yield improvements in performance for many earlier projects. Therefore, they should be taken into consideration in this project as well. Analysis should show what practical improvements they give for this implementation. Many optimizations and best practices as well as the way the execution and memory model may work with devices from different vendors. If this is possible, it should be verified.

4.2.1 OpenCL Execution Model

On NVIDIA platform, OpenCL workitems are executed by CUDA streaming processors (scalar processors or compute elements) and a single workitem is mapped directly to single hardware managed thread. Thread will be blocked if one of the operands is not ready. Reading memory does not block the thread. latency is hidden, because threads are switched, so enough workitem are needed to hide latency. Workgroups are executed on CUDA streaming multiprocessors (compute units) and they do not migrate. However, several concurrent work-groups can reside on one multiprocessor. This is only limited by multiprocessor resources. A whole kernel is launched as an ND Range of workgroups on a sinel CUDA-enabled GPU (compute device). Workgroups are dynamically load-balanced by hardware scheduler.

4.2.2 OpenCL Memory Model on NVIDIA

On NVIDIA platform, each hardware thread has a dedicated `__private` region of registers for stack of variables. Each multiprocessor allocates dedicated storage for `__local` memory and `__constant` caches. For `__constant` memory data is stored in global memory. Workitems running on a multiprocessor can communicate through `__local` shared memory. All workgroups on the device can access `__global` memory that is stored in global memory available to all multiprocessors on device. Atomic operations allow powerful forms of global communication.

4.2.3 OpenCL Synchronization

On a level of workitem, synchronization is achieved through memory system control using `mem_fence()` functions. Independent atomic operations `atom_*()` may also be used. For workgroups synchronization, `barrier()` function is supported as a single instruction fast barrier directly in hardware. Only after the execution of `barrier()`, `mem_fence()`, `read_mem_fence()`, or `write_mem_fence()` are prior writes to global or shared memory of a given thread guaranteed to be visible by other threads. Collective operations like `async_work_group_copy()` leverage the entire multiprocessor. On level of whole ND Range grid, direct hardware support is provided. While using enqueued commands on grid, e.g. `EnqueueNDRange`, an `cl_event` can be used for scheduling which is directly supported in hardware.

4.2.4 Best practices

It is worth mentioning that NVIDIA GPUs have a scalar architecture, so language constructs like vector types can be used for convenience and code readability rather than for performance as these are not going to be mapped into architecture. In general, on NVIDIA architecture a larger number of workitems gives better performance than wide vectors per workitem. Furthermore, performance can be significantly optimized by overlapping memory with hardware computation. This is why high ratio between arithmetic operations and memory transaction is especially valuable. NVIDIA architecture provides many concurrent workitems to facilitate that. Also asynchronous transfers should be investigated as they permit overlapping computation and bulk transfer.

Moreover, an optimization should take advantage of `__local` memory, as it provides access that is hundred times faster than `__global` memory. In addition, workitems can cooperate via this type of memory with much lower overhead using `barrier()` operation with `CLK_LOCAL_MEM_FENCE`. Next step is to maximize the reuse of shared memory through locality management by staging loads and stores.

Next step is to optimize memory access. Locality of `__global` memory access patterns should be assessed. For instance, hardware coalescing of access within 128-byte memory blocks yields huge performance improvement effect. Another approach is to use cached texture memory that in OpenCL are access through Image format. Spatial locality of accesses can be optimized as image reads benefit from processing as 2D blocks. In

this case experiments with different workgroup sizes should be done. Finally, OpenCL should be allowed to allocate memory optimally. This can be achieved through specifying `CL_MEM_ALLOC_HOST_PTR` option while invoking `clCreateBuffer` for device buffers. Then the implementation can optimize alignment and location. Memory can still be access from the host via `clEnqueueMap` if needed.

For transfer/compute separate command queues can be used as they can overlap. This yields best results when transfer and compute time is balanced and is most useful when data has high reuse. Another approach is to use Pinned (non-pageable) memory and to directly pass `ALLOC_HOST` memory to kernel so that GPUs latency hiding is used to ensure maximal bus usage. This approach gives best performance when there is nearly no reuse of data. Then, no events are needed to synchronize between copy and kernel.

Control flow needs to managed. Different execution paths within workgroup are serialized. Different workgroups can execute different code with no impact on performance. Diverging within a workgroup should be avoided, e.g. branch granularity should be a whole multiple of workgroup size.In general used algorithms should be parallel in nature and should not consist of complex control, data structures or excessive message passing.

In terms of the computation, it should be partitioned to such extent as it keeps all the GPU multiprocessors equally busy. This boils down to keeping a number of workgroups and workitems in them high. In particular, number of workgroups should be much bigger then the number of Compute Units. A good estimate is said to be 256-512 workitems per workgroup. At the same time, the resource usage should be kept low enough to allow multiple active workgroups to be processed by single multiprocessor. This is dependant on the number of registers and `__local` memory used by workgroup. Am Occupancy Calculator tool provided by NVIDIA might be useful to get a rough estimate.

Finally, optimized math functions should be used where possible and an impact of compiler options should be investigated. In particular, two should be considered `-cl-mad-enable` that allow using FMADs, computing multiplication-addition operations with less accuracy and `-cl-fast-relaxed-math` that enables many aggressive compiler optimizations for single- and double-precision floating-point arithmetic that may violate the IEEE 754 standard. These options are passed to `clBuildProgram` function that compiles and links a program executable from the OpenCL program source or binary.[44, 47] As a side note, it is a good practice to avoid automatic conversion of doubles to floats by using single-precision floating-point constants, defined with an `f` suffix `3.141592653589793f`, `1.0f`, `0.5f`.

Shared memory holds the parameters or arguments that are passed to kernels at launch, so in kernels with long argument lists, it can be valuable to put some arguments into constant memory (and reference them there) rather than consume shared memory.

4.3 Implementation

Code is written in C on the host and OpenCL C is used to write kernels to be executed on the device. The overall process of setting up the OpenCL environment will be dropped for this project as there are plenty of resources, from original Khronos materials to tutorial in the Internet, that guide a beginner programmer. Moreover, a specification of OpenCL talks about its constructs, programming and memory model. Therefore only essential constructs that are used in implementation are described in the report.

The global worksize of computation on GPU is dependant on specified size of the simulated grid. A human-readable definition file is used to set parameters and create the benchmark problem to be simulated by solver. In Griebel's code the parameters are originally specified in inputfile, but the geometry of the boundaries is hardcoded in the source code. This was left unchanged.

Implementation consists of sets of kernels for every of the functions from original code. Modularity was thus maintained, but the effect that it has on overall performance (loading many small kernels vs. a few complex kernels) needs to be determined in tests.

Griebel's code have some intrinsics. For instance, the original code seems to be written for maximal readability and conformance with the content in the book. Thus ghost cells for boundary conditions are additionally allocated.

Code proved to be portable and works under both on Windows and Unix platforms without any additional changes. The code for the project was first and foremost implemented on Windows 7 platform. The eventual execution on Linux platform was pushed to the test part of implementation. Compilers used are nvcc and Visual Studio on Windows. gcc is used on Linux.

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 from the device local memory on the host the process has to be reverted step by step.

4.3.1 Benchmarking code

Code is timed with standard library time.h clock function. The OpenCL profiling timer is not used in this code because the goal is to measure wall clock time on CPU. Code can be timed with buffer allocation and reading it or without. To compare the performance with CPU that accesses its memory on a constant basis, the memory allocation on the device and then later reading it back should be counted in for a fair comparison.

To visualise results of computation, Matlab scripts were written. In original code, Griebel et all. added a possibility to save particle trajectories for visualisation purposes. This was kept, although it is turned off for performance reasons. It was not ported to GPU.

To implement a naive kernel, an easiest approach is to strive for the straightforward unoptimized port of the function. This usually boils down to understanding that kernel is like a (inner nested) loop's body, so the first thing to do is to e.g. getting rid of for loops. In this moment, the boundaries still should not be not crossed so if-conditions should be added to ensure that.

Implementing shared memory kernels usually does not require any changes in the underlying algorithm. Instead, focus is pushed to the host where the kernel is executed and code of each kernel is tuned by finding optimal configuration for work item/ work groups. This is most often done through straightforward timing experiments and brute-force testing.

Another guideline for optimal OpenCL development is that production code should 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).

No comparison to other project could be done The source code that came closest is by Intel[26]. This one is only slightly similar to the code in project mostly because its implemented in OpenCL and tries to simulate and CFD problem that need solving Navier-Stokes equations.

5 Performance Analysis

5.1 General

For detailed analysis of the kernel performance various conditions should be checked. Previous section proposes optimizations. This one tries to analyse if they are practical to use. Test of OpenCL application should be run across different architectures to check its portability and how code behaves on different machines. The kernels should also scale with problem size. Speedups should be measured between version that execute on host and on device. Finally, as the computation is done on floating point number, comparison between single and double precision might yield interesting results.

5.2 Timing performance

OpenCL calls and kernel executions can be timed using either CPU or GPU timers. In terms of CPU, any CPU timer can be used to measure the elapsed time of an OpenCL call. [44]. However, it is critical to remember that some OpenCL function calls can be non-blocking, i.e. they return control back to the calling CPU thread prior to completing their work. While using OpenCL GPU Timers, each enqueue call optionally returns an event object that uniquely identifies the enqueued command. The event object of a command can be used to measure its execution time. Profiling can be enabled by setting the `CL_QUEUE_PROFILING_ENABLE` flag in properties argument of either `clCreateCommandQueue()` or `clSetCommandQueueProperty()`.[44]

5.3 Overall runtime and performance

On the NVIDIA platform, naive kernels were implemented for stencil functions. Moreover, parts of code were implemented that could have gains in performance thanks to reduction process done and optimized on GPU, e.g. residual computation. Approach used by [36] was imitated and yielded performance optimization for 3 first reduction kernels. No optimized library routine was checked as none was found to be feasible in this setup.

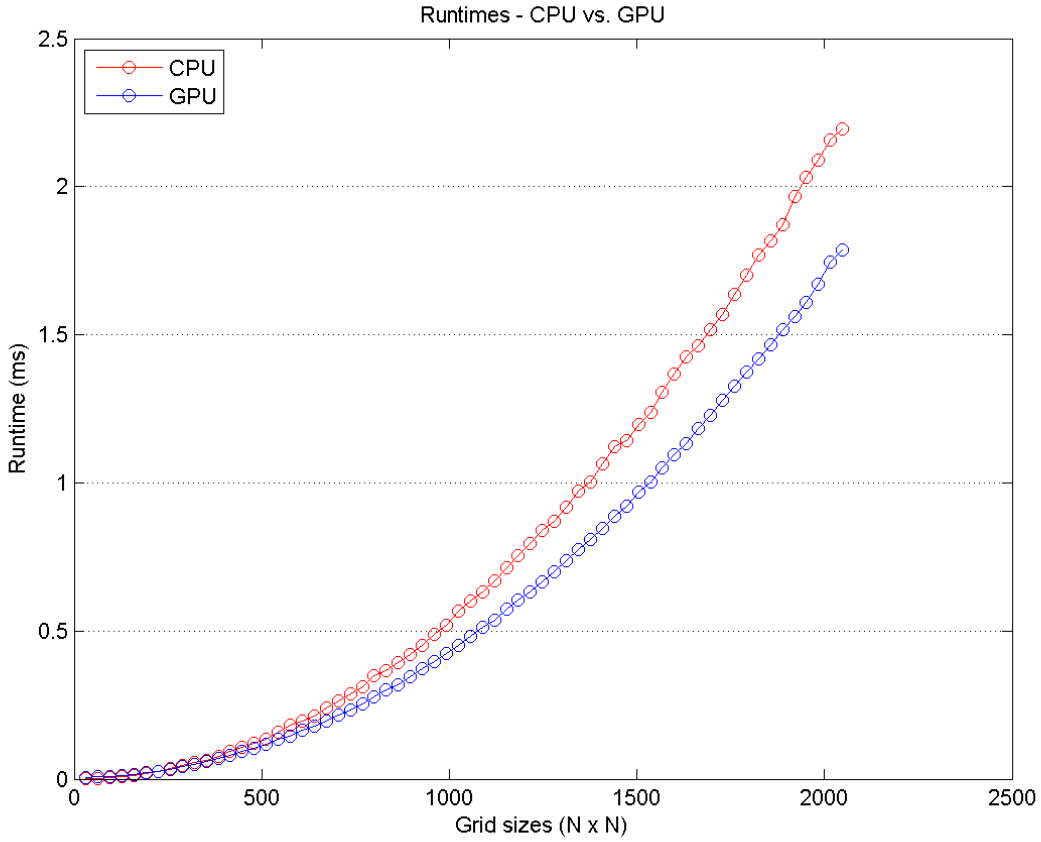

Figure 5.1: Exemplary Runtime - CPU vs. Best GPU Configuration Exemplary Runtime - CPU vs. Best GPU Configuration on NVIDIA 550M

Runtime is counted taking into consideration writing to and reading from the memory on the device.

Performance is measured in GFLOP/s. Kernels vary in the number of floating point operations.

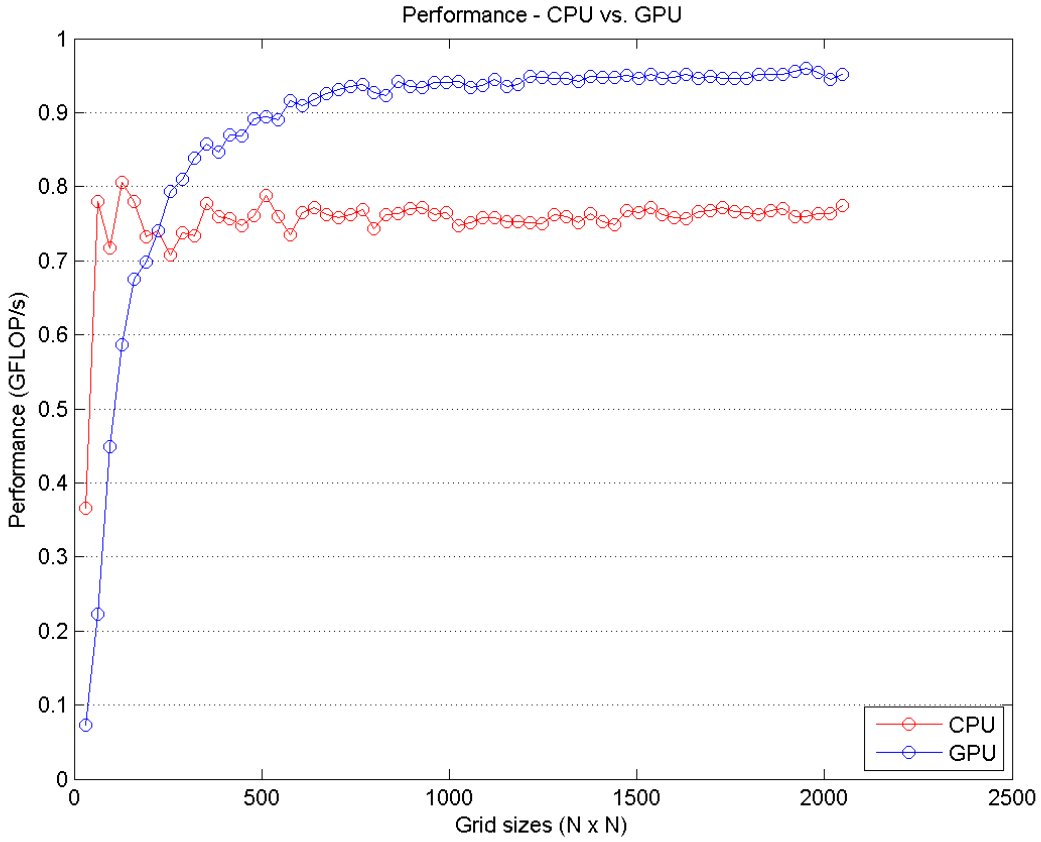

Figure 5.2: Performance - CPU vs. Best GPU Configuration Performance - CPU vs. Best GPU Configuration on NVIDIA 550M

`COMP_TEMP_kernel`	imax*jmax×(21 additions + 20 multiplications)
`COMP_FG_kernel`	imax×jmax×(33 adds + 19 muls) × 2 (F & G)
`COMP_RHS_kernel`	imax×jmax×(3 adds)
`POISSON_p0_kernel`	imax×jmax×(1 add + 1 mul)
`POISSON_2_copy_boundary_kernel`	itermax×imax×jmax ×(1 add + 1 mul)
`POISSON_2_relaxation_kernel`	itermax×imax×jmax ×(5 adds + 4 muls)
`ADAP_UV_kernel`	imax×jmax×(2 adds + 1 mul) × 2 (U & V)

Speedup is a ratio between performance of set kernel on GPU to its original version run on CPU.

While choosing the size of the grid for this project, padding the whole grid with ghost cells for boundary conditions is necessary. This means that the simulated grid in reality is smaller by in width and height.

Versions with and without residual calculation step were tested and proved that this step is a bottleneck. To achieve better speedups the accuracy of final result had to be sacrificed

and a step was dropped for the most optimal results. Computing a residual is the way to measure the convergence in the iterative solver as it is used as a stop criterion. As the distances between gridlines get smaller, the more detailed the discretization is and more iteration are needed. This was observed in this implementation and it hinders the performance of the solution.

Figure 5.3: Performance - Code without Residual Computation vs. Code with Residual Computation GPU Configuration Performance - Code without Residual Computation vs. Code with Residual Computation GPU Configuration on NVIDIA 550M

Furthermore, the relaxation part of the kernel was calculated on the host due to the complications in implementing it on GPU. Once again it was due the fact that a stencil that the SOR method use is tightly bound with the order of computation. The obvious solution once again is to adopt an algorithm that is more suitable for this platform. Essentially all the code should be computed on GPU device only, even if that means running kernels that do not demonstrate any speed-up compared with running them on the host CPU. The data should be transferred to device once in the beginning and next time it should be retrieved from there is when all the computation is over and results should be displayed. In other words, intermediate data structures should be created in device memory, operated on by the device, and destroyed without ever being mapped by the host or copied to host memory.[44] Morover, as each transfer has the overhead

associated with it, many small transfers should be batched into one larger transfer. It performs significantly better than making each transfer separately.[44] Such approach was taken in the project where data is wrote to the device once in the beginning to all the used buffers and then all the reading is than as a batch in the end. An example of intermediate buffer that is used only on the device is a buffer that stores new temperature values in kernel that computes temperature.

In addition, bandwidth between host and device should be kept high all the time. This means, that for computed results the effective bandwidth should be compared with theoretical maximal bandwidth. This way the occupancy of the device can be measured. In this project it was achieved by using a large enough workgroup size so that all the streaming multiprocessors have job to do.

As a main conclusion, it can be claimed that a main prohibiter of performance and its optimization was an algorithm itself. In its basic version as in the Griebel's code it does not expose much data parallelism so suitable for GPUs. In this project, an overall process of porting the implementation to GPU was very quickly inhibited by the number of iterations in Poisson SOR relaxation and thus algorithmic optimization must be sought quickly. Changing the algorithm to suit GPU better was not the main way to achieve the performance optimization for this project, so the results are in general not that optimistic.

Trap of premature optimization should be avoided. The implemented kernel should be always first checked against the sequential code to yield valid results. Moroever, it might be helpful to optimize it in separation from the other kernels used.

5.4 Different workgroup size

The global size was fixed an different workgroup sizes were investigated. In general, a recommendation to use multiples of 16 was followed and proved to be a true guideline in optimization.

Workgroup size for a given algorithm should be selected in such way as to be an even multiple of the width of the hardware scheduling units.

5.5 Synchronous vs. Asynchronous

Iterative methods can achieve convergence faster when they are not synchronized much. This comes in expense of proliferation of read/write data hazards that are obvious performance penalties.

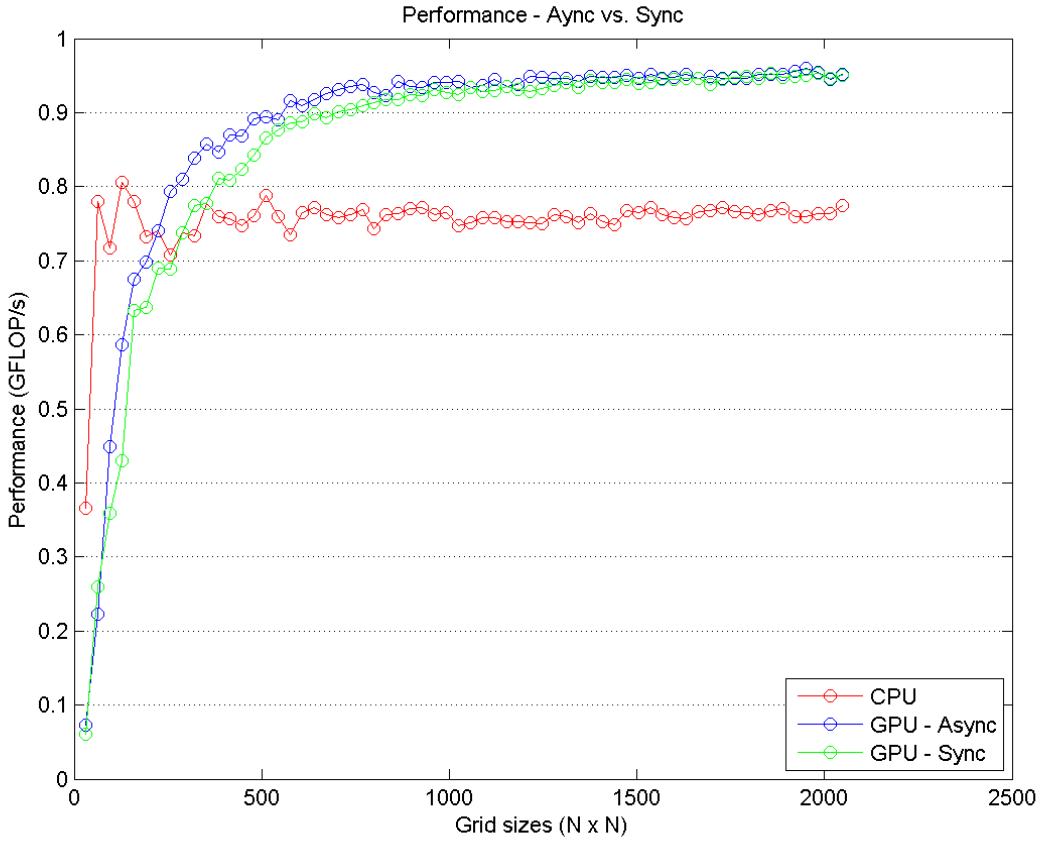

Figure 5.4: Performance - Async vs. Sync GPU Configuration Performance - Async vs. Sync GPU Configuration on NVIDIA 550M

Global synchronization between kernels and other objects in queues is possible using events and `clWaitForEvents()` operation. In general, events are used to track the execution status of API calls done on host. For synchronous version this construct is inserted after every `clEnqueueNDRangeKernel`. It means that host will always synchronize in this place and wait with execution until device will be done with current action in queue, here kernel. In project at hand, consequent kernels are dependant on the order in queue, because they read and write to the same arrays. In general, `clWaitForEvents()` can be inserted after every operation that take events as parameters. A lot of performance is thus lost, because kernels are specified to wait for one another. For SOR iterative method, a kernel could be also implemented so as to have a synchronization barrier in the end of each iteration, but this was not necessary as code was synchronized in parts were code had to be read form device to be partially computed on the host.

Another way of blocking is provided for memory transfer. Here a blocking flag needs to explicitly changed to `CL_TRUE`. The queue want be further processed as long as the whole memory will not be transferred and the operation will return.

An asynchronous version is an extreme version, where the smallest subset of constructs are globally synchronized. It was discovered that these are mostly `clEnqueueReadBuffer()` operations that need to be set a `CL_TRUE` blocking flag, but this is not true in all situations. It was observed that if this operation is enqueued with the non-blocking flag just before the code part where this buffer is used, the code yields erroneous results. This is rather self-explanatory as there is no valid data at the buffer yet at the time it is already read. In all the other situations, the code did not need to be explicitly synchronized, surprisingly also not where kernel were enqueued.

Until recently one device meant one queue, but this is about to change with new architectures that have more schedulers as in GCN or Kepler.

It was also proved to be not connected with what kind of memory is used as it was tried with both normal allocation of memory and using pinned memory. The conclusion is that it is not worth explicitly synchronize the queue using operation on the host. The approach where all the synchronization is turned off and only a minimum subset of synchronization is used to achieve correct results that match the non-optimized results.

5.6 Single vs. Double Precision Floating Point Numbers

Double-precision floating point numbers could be used on NVIDIA platform, because it supported `cl_khr_fp64` extension. Moreover, different build options passed to `clBuildProgram()` can be used to change the behaviour of OpenCL code in terms of floating-point numbers. Namely, following set of options was used at most times:

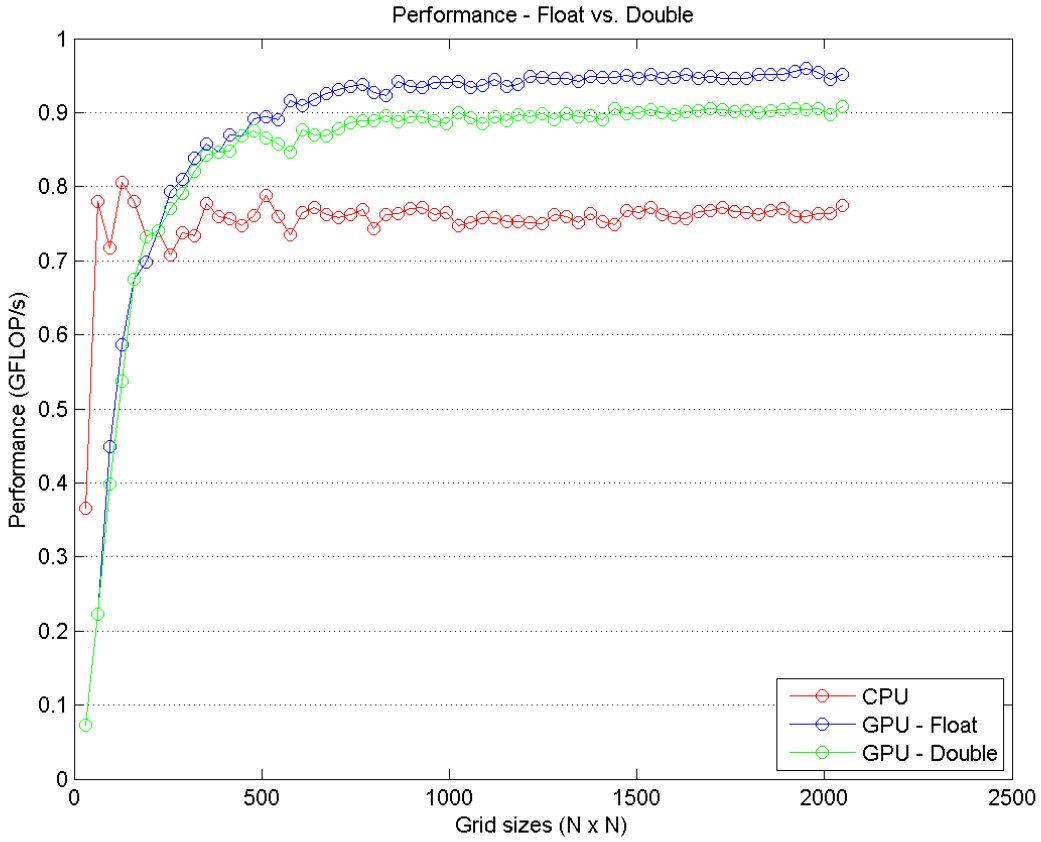

Figure 5.5: Performance - Single vs. Double-Precision GPU Configuration Performance - Single vs. Double-Precision Floating Point Numbers GPU Configuration on NVIDIA 550M

- `-cl-single-precision-constant` proved to save the programmer a burden of changing all the floating point numbers from default double to single-precision representation that was claimed to be an optimization [44]
- `-cl-denorms-are-zero`
- `-cl-strict-aliasing`
- `-cl-mad-enable`
- `-cl-no-signed-zeros`
- `-cl-fast-relaxed-math`

Setting those options and changing to single-precision floating point number representation not surprisingly yielded a boost over the double implementation. However, the differences between values computed on the device an the host could be observed in comparison to

doubles, which cannot be optimized that much yet, but they yield standard-conformant results.

`fabs` was the only mathematical function used in code of kernels and it was optimized by compiler.

5.7 Normal vs. Pinned memory on NVIDIA

NVIDIA OpenCL Best Practices Guide[44] presents the most optimal way to allocate the memory on NVIDIA platform and claims that pinned memory should be used for that. This approach was checked for this project and proved to yield significant increase in performance.

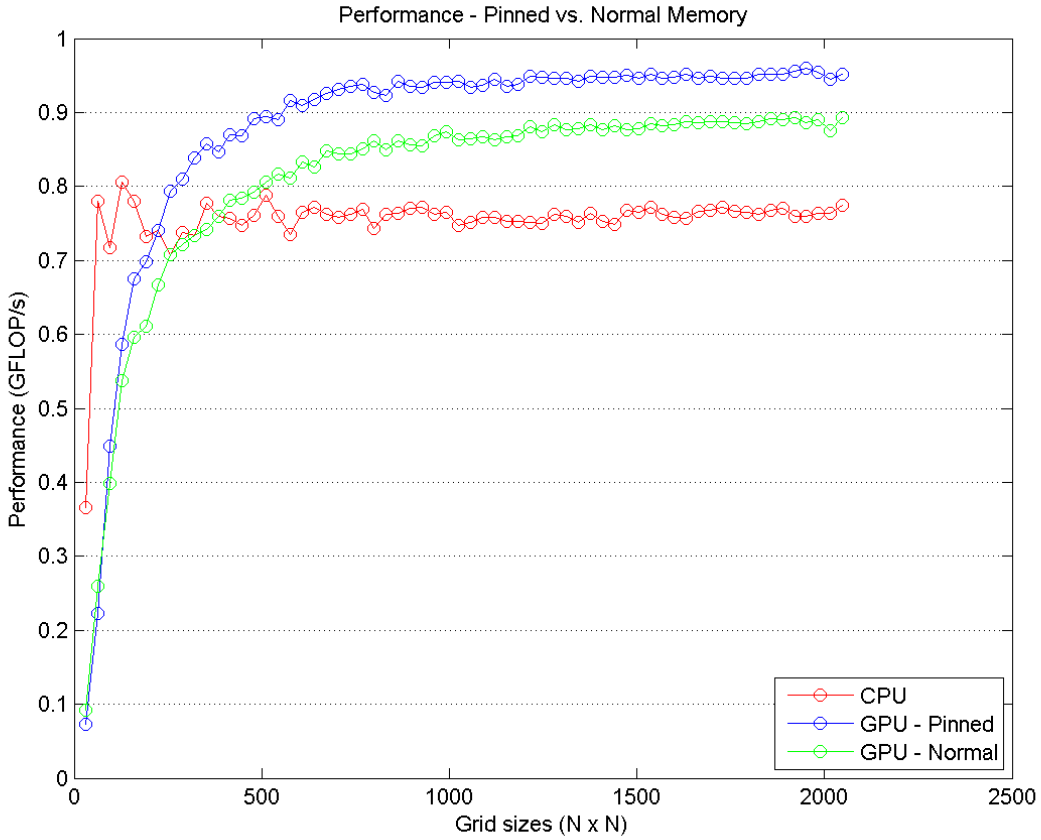

Figure 5.6: Performance - Pinned vs. Normal Memory GPU Configuration Performance - Pinned vs. Normal Memory GPU Configuration on NVIDIA 550M

Pinned memory is allocated in non-paging RAM memory on the machine, so it can be directly accessed by GPU without unnecessary communication with CPU. The way to allocate it without copying from host pointer is to pass `CL_MEM_ALLOC_HOST_PTR` flag in

addition to the type of usage, when the buffer is created using `clCreateBuffer()`. Then the memory is mapped to the buffer using `clEnqueueMapBuffer()`. Afterwards, the buffer memory is passed and retrieved form the device in ordinary way.

In conclusion, this approach proved to give performance boost on NVIDIA platform. It is dependant on the amount of data read into RAM memory.

5.8 Local memory

The attempt to implement optimized kernels using local memory was taken for all kernels that were implemented in naive version. However, this process proved to be not that straightforward as it is matrix multiplication and similar kind of kernels that are thoroughly presented as an example of high priority optimization technique in literature. Without a doubt, reading from global memory should be minimized and reading form it should be coalesced. However, for kernels like these in the project, a couple of buffers is passed to the kernel and computation is dependant on data from different sources. Thus a couple of shared memories have to be maintained, e.g. to compute temperature values for both velocities in velocity field is needed as well as the old temperature. The same holds for computing F and G. To make matters worse, padding must be added to workgroups as these kernels use stencils to compute values. This means that boundary values from neighbouring workgroups need to be padded to workgroup to compute its values correctly without accessing the global memory. This posed a challenge. An approach that could be investigated as an optimization in use of shared memory would be to use a workgroup size that would spread over the whole width of the buffer. Values from global memory would be read row wise as needed row after row, i.e. when the computation would be over with current level another would be read in expense for the previous one that is not needed any more.

On the other hand, the TEMP, FG and POISSON relaxation kernel have a lot of floating-point operations (over 50 for two former ones, 10 for later) that otherwise would need to additionally access global memory. Moreover, stencil with radius 2 need to access four neighbouring cells in average. This is a motivation for pushing the data to workgroup's local memory as these kernels are highly memory bound.

Yet another drawback of ported functions was the abundance of branching they are dependant on. As it is known, different execution paths on the device should be avoided. The function to set boundary conditions is an example of code that not necessarily should be ported to GPU. On the other hand, one of the best practices is to always try to keep all the computation on the device.

As a sidenote, on CUDA platform, a popular way to debug and profile code is to set the verbose flag to the nvcc compiler for PTX information about the utilization of shared memory and registers by threads. This can be done with OpenCL using `-cl-nv-verbose` build option with `clBuildProgram()`.

5.9 Different platforms

Most of development was done on NVIDIA and this one was investigate to the most extent. Programming guide and Best practices acted as a guideline and most optimization was achieved. Intel platform is easier to debug though, because as it is a CPU the source code of kernel can be accessed as it would be the same device (which is apparently true) and thus debugged normally. Moreover, on Intel's platform the code is optimized to the vector SSE/AVX instructions. AMD devices that were available are also based on very similar VLIW (Very Long Instruction Word) vector-based approach on the VLIW4/5 architectures.

5.10 Different problems

Lid-driven cavity problem was simulated at most times. This one is a fixed boundary problem and do not introduce any extraordinary geometry. Furthermore, the boundary conditions are simple.

5.11 Profiling results

The only profiler that was finally used was NVIDIA Visual Profiler available with CUDA SDK. The profiler offered a graphical timeline that could be zoomed. First, profiler helped to discover that the biggest bottleneck of whole implementation was the residual computation, which took the most time despite the fact it was implemented using shared memory. Profiler also showed there is no overlap between memory transfer and computation in kernels that severely hindered performance.

6 Conclusions

6.1 General

The goal of the project was to get acquainted with programming for GPU using OpenCL technology. This project was meant as an introduction to the technology, although some initial experience was already gathered during High Performance Computing course that took place at DTU in January 2012. However, at this course another technology was used. Despite the fact that CUDA has a lot of in common with OpenCL, work on this project showed that assimilation of OpenCL's pipeline may take time. In addition, OpenCL lacks many basic tools like cross-platform debugger or profiler as for now. However, as most of the time and effort in GPU programming is spent on developing and optimizing the kernel code. This proved to be independent from the used technology and yield the best optimization results.

Furthermore, author had no prior experience with CFD codes or algorithmic optimizations of numerical methods. If there was any one could come up with implementation of other numerical method that could be easier to parallelize earlier. Thus, plenty of time was spent on seeking kernel optimizations in place where serious algorithmic optimization should take place as it was with relaxation and residual kernel.

Problems with bandwidth and minimizing transfer between the host and the device were met due to the modularity of the implementation. Solver consists of a number of functions executed in time-stepping loop. For SOR relaxation, the most memory-bound kernels are iterated many time to achieve satisfying results. No single kernel can be created to solve the Navier-Stokes equations. Instead a number of separate kernels need to initialized and then synchronized. The goal is to keep the memory transfer low and try to keep the occupancy of computing cores high. It may be achieved through using local memory and overlap of memory transfers and computations. With just a sequential CPU implementation at hand and no similar projects with open code implemented on GPU to compare to, the task was significantly harder to be solved. In addition, no suitable libraries that could be used for comparison are freely available.

To check what is the result of minimization of use of global memory, a trial was taken to implement the version utilizing shared memory, but no successful implementation was proposed due to the time constraints.

Moreover, computations on single and double precision floating point numbers were investigated. Double-precision is a must in scientific computations, but it pose significant limitation for performance. The GPUs depending on the models deal considerably better or worse with double-precision floating point numbers. GPUs targeted for scientific market like Tesla Fermi architecture have significantly better performance in comparison to GPUs like ordinary Fermi architecture cards meant for fast computation of single-precision floating point numbers sufficient for graphics market.

One of the main motivations to use OpenCL is its portability. However, this technology is only that much competitive as much optimized given platform. In other words, OpenCL may give significant performance boost, but to maximize it an effort to optimize it for every platform separately is needed.

Usually a following effect can be observed on GPUs. With the problem growing in size, speedup on GPU grows due to the ratio of computations to memory transfers grow. In other words, for large grids more time is spent on calculation then on transferring memory from host to device and vice versa. In this project this effect was hindered as a set of kernels was usedn not a single one. Moreover, due to intrinsics of the used iterative numerical method (SOR overrelaxation for solving Poisson pressure equation) the kernels are executed significantly more times for larger grids. This introduce a new layer of problems and new synchronization problems.

The only Profiler used was NVIDIA's Visual Profiler. Optimization guidelines it gives for OpenCL code was found vague. Moreover, it seems to be unable to collect all of the data needed to construct thorough statistics. However, the timeline still helps in development as it shows the execution process of consequent kernels.

6.2 Improvements

Optimize and test each kernel separately

Probably the most obvious improvement for the project would be to test every kernel on its own taking it apart form the whole algorithm and optimizing it to the most extent in seperation. During this project, an implementation that would yield the same correct results as the provided sequential code base was primarily sought. Ensuring that this was satisfied was not straightforward. It may be the case Modularity pose additional difficulties in GPU programming. Instead, code could be written from scratch and own modules around new algorithms would be designed.

Optimize the kernels more

Shared memory kernels could be more thoroughly investigated, i.e. more sizes of rectangular workgroups and their different sizes could be investigated. Only one kernel using local memory was implemented - namely a residua computation kernel. Still the analysis showed that it is better for performance not to use residual all than to use an optimized shared memory kernel. Following the experience gathered in High Performance Computing course,

where a matrix multiplication kernel was implemented, only concrete local work sizes like 4×4, 8×8, 16×16 were investigated. Irregular sizes and different access schemes should get more attention. Every platform varies and differently maps the OpenCL workgroups to its resources and this should be further investigated. OpenCL needs to be customized for every platform separately.

Parallelism on the level of kernels and events

In final implementation kernels were allowed to be enqueued asynchronously and not be synchronized using events. This should lead to overlapping of kernel executions as the next kernel would be started before the previous would finish. As a follow up using two queues on the architecture that enables that could be investigated.

Optimization Options in Kernel Compiler

Kernel compiler that is called through `clBuildProgram` in the host code to build a OpenCL program out of source code with kernels can be provided with compilation options. These gave different outcome, but in general did not make huge difference. However. options like changing the precision of floating point numbers through `-cl-fast-relaxed-math` should have a bigger impact on the accuracy of computation and their speed.

Profile code on AMD platform

AMD APP SDK provides tools for debugging, analysing and profiling code on AMD platform. No experiences in work with them (minor with debugger) were gathered as code was developed and tested mainly on NVIDIA platform. NVIDIA Visual Profiler provided some valuable information, but was kind of vague in proposing solutions.

Benchmark More problems from the book

The original code works with Free Boundary Value problems and the book shows simulations of such problems. These problems are real benchmarks for the implemented solver and prove that solver is valid. Solver can be configured to gather data that later can be used for visualisation of the problems.

Implement Red-Black Gauss-Seidel Kernel

The SOR iteration method (that is a customised form of inherently serial Gauss-Seidel algorithm) proved to pose severe limitations on the parallelization process and running it on GPU due to the fact it is an iterative method. The stencil used in this method is dependant on an order of computations, i.e. some of the cells need to be computed at the time values for given cell are computed. On GPU the order cannot be satisfied and implemented in straightforward manner not say it is impossible. Synchronization on device is only possible within the workgroup and the order of executions of work-item is usually undefined due to the many factors such as following branches. GPUs work best on algorithms that can be mapped to a lot of data at once. A solution to this issiue may be to use Multi-color scheme like red-black Gauss-Seidel. This method allows to reduce dependencies. This method devides single iteration into two and divides the grid into a

red-black checkboard. First sweep is done only on red cells and second sweep is done only on black cells. Naturally, this comes in expense of reduced algorithmic efficiency, because more iterations are necessary to achieve the same accuracy level. Moreover, Gauss-Seidel is usually compared with Jacobi method, which itself offers better parallelization possibilities than normal Gauss-Seidel-based methods.

Use OpenCL C++ Wrapper

Development in C and using OpenCL Platform API in host code might be cumbersome and result in huge codebase. What should be most of the time be optimized is written in OpenCL C in kernel code run on device. Thus it does not matter much how the kernel code is initialized and how the OpenCL Platform API is called on the host. As long as the memory is initialized so as the OpenCL memory objects can be created and all the OpenCL pipeline can be constructed, the host code could be implemented in any language. C++ is a natural choice, but is not still fully supported by OpenCL, e.g. kernels cannot be templated to be called with different parameters determined in runtime. For clean code though C++ wrapper could be used. There are already open-source libraries built on top of it available in the net that simplifies the OpenCL usage further.

Use Vector Types

OpenCL specification 1.1 introduced vector operations and vector variables, but these were not used in this project and thus not investigated, although they could yield promising results on specific device architectures that support vector instructions. SSE/AVX instructions and VLIW architecures come into mind.

Use Image Format

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

3D Grid

When satisfying results would be achieved in two-dimensions, another dimension could be added. This would introduce new challenges, but GPUs are ready to support such grids and there are many successful projects, also mentioned in this report, that simulate similar CFD problems in 3D. This would impose investigation of a wide array of different workgroup sizes.

Multigrid method

A multigrid approach could be implemented as another step of algorithmic parallelization of numerical method used to solve Navier-Stokes equations. This would lead to better efficiency and scalability and the GPUs could be targeted. As doing multigrid requires some more understanding of the numerical discretization methods, this idea was dropped for this project. Essentially though the multigrid method could be implemented in form of a few simple kernels that would perform prolongation, restriction, Jacobi, Gauss-Seidel or SOR smoothening and manage a grid hierarchy.

Distributed calculations

While a size of a simulated grid is increased, a limit of memory resources is met. Form here the only way to increase the size of computed grid, is to decompose the problem and spread it over a number of devices. The results would then be gathered from these devices. Devices could be on the same machine or on the remote machine. Technologies like MPI allow to implement communication between such devices. If the project was sufficently optimized on single GPU, it would interesting to see how it perform on cluster of GPUs.

Mobile platform

Recently GPUs started to be installed on mobile devices like smartphones. Solution could be simulated on Tegra chip on Android Platform to check the performance of computation on yet another type of platform. Such investigation was not much reserached recently and would rather check a general feasibility of such implementation than yield any extraordinary performance.

References

[1] TOP 500, 2012. URL http://www.top500.org/.

[2] Accelereyes, 2012. URL http://blog.accelereyes.com/blog/2012/02/17/opencl_vs_cuda_webinar_recap/.

[3] Adobe, 2012. URL http://blogs.adobe.com/premiereprotraining/2012/05/opencl-and-premiere-pro-cs6.html.

[4] AMD. AMD APP Profiler, 2012. URL http://developer.amd.com/tools/amdappprofiler/pages/default.aspx.

[5] AMD. OpenCL applications, 2012. URL http://developer.amd.com/archive/AppShowcaseArchive/Pages/default.aspx.

[6] AMD. OpenCL APP SDK, 2012. URL http://developer.amd.com/sdks/AMDAPPSDK/samples/showcase/Pages/default.aspx.

[7] AMD. OpenCL Developer Zone, 2012. URL http://developer.amd.com/zones/OpenCLZone/.

[8] AMD. AMD gDEBugger, 2012. URL http://developer.amd.com/tools/gDEBugger/Pages/default.aspx.

[9] AMD. AMD APP KernelAnalyzer, 2012. URL http://developer.amd.com/tools/AMDAPPKernelAnalyzer/Pages/default.aspx.

[10] AMD. AMD Accelerated Parallel Processing Math Libraries (APPML), 2012. URL http://developer.amd.com/libraries/appmathlibs/Pages/default.aspx.

[11] T. Bednarz, C. Caris, and J.A. Taylor. Numerical Simulations in Fluid Dynamics using GPU: a Practical Introduction. Presentation at GPU Technology Conference 2010, 2010. URL http://www.nvidia.com/content/GTC-2010/pdfs/2058_GTC2010.pdf.

[12] T. Bednarz, L. Domanski, and J.A. Taylor. Computational Fluid Dynamics using OpenCL: a Practical Introduction. *19th International Congress on Modelling and Simulation*, 2011.

[13] T. Brandvik and G. Pullan. SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1181 –1188, 2010.

[14] T. Brandvik and G. Pullan. Turbostream, 2012. URL `http://www.turbostream-cfd.com/`.

[15] Tobias Brandvik and Graham Pullan. An Accelerated 3D Navier-Stokes Solver for Flows in Turbomachines. *ASME Conference Proceedings*, 2009(48883):619–629, 2009.

[16] CAPS, 2012. URL `http://www.caps-entreprise.com/fr/page/index.php?id=49&p_p=36`.

[17] clpp OpenCL Data Parallel Primitives Library, 2012. URL `http://code.google.com/p/clpp/`.

[18] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-based Multicore Platforms*. University of California, Berkeley, 2009.

[19] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

[20] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Elsevier /Morgan Kaufmann, San Francisco, CA, USA, 2011. ISBN 9780123877666.

[21] Benedict Gaster, Lee Howes, and Simon McIntosh-Smith. Hipeac 2012 heterogeneous programming – a tutorial, 2012. URL `http://www.heterogeneouscompute.org/?page_id=43`.

[22] Michael Griebel and Peter Zaspel. A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations. *Computer Science - Research and Development*, 25:65–73, 2010. ISSN 1865-2034.

[23] 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. ISBN 9780898713985.

[24] Inside HPC, 2012. URL `http://insidehpc.com/2012/05/15/nvidias-kepler-pushes-parallelism-up-to-eleven/`.

[25] Kishonti Informatics. OpenCL Benchmark, 2012. URL `https://clbenchmark.com/`.

[26] Intel. 3D Fluid Simulation using OpenCL, 2012. URL `http://software.intel.com/en-us/articles/vcsource-samples-3d-fluid-simulation/`.

[27] Intel. Intel MIC, 2012. URL `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`.

[28] Intel. Intel SDK for OpenCL Applications 2012, 2012. URL `http://software.intel.com/en-us/articles/vcsource-tools-opencl-sdk/`.

[29] Intel. OpenCL SDK, 2012. URL http://software.intel.com/en-us/articles/opencl-sdk/.

[30] Khronos. OpenCL overview - November 2011, 2011. URL http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf.

[31] Khronos. OpenCL adopters, 2012. URL http://www.khronos.org/conformance/adopters/conformant-companies.

[32] Khronos. OpenCL Homepage, 2012. URL http://www.khronos.org/opencl/.

[33] Khronos. The OpenCL C++ Wrapper API, 2012. URL http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf.

[34] Khronos, 2012. URL http://www.khronos.org/news/press/khronos-releases-opencl-1.2-specification.

[35] Khronos. WebCL, 2012. URL http://www.khronos.org/webcl/.

[36] Max la Cour Christensen and Klaus Langgren Eskildsen. Parallel Sum Reduction, final report for 02931 Scientific GPU Computing. Technical report, Technical University of Denmark, 2011.

[37] GATLAS Library, 2012. URL http://golem5.org/gatlas/.

[38] Microsoft. C++ AMP : Language and Programming Model — Version 0.9, 2012. URL http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf.

[39] Microsoft. C++ AMP: Language and Programming Model, 2012. URL http://msdn.microsoft.com/en-us/library/hh265136.

[40] Microsoft. DirectCompute, 2012. URL http://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx.

[41] Allan Svejstrup Nielsen, Allan Peter Engsig-Karup, and Bernd Dammann. Parallel Programming using OpenCL on Modern Architectures. Technical report, Technical University of Denmark, 2012.

[42] NVIDIA. Fermi whitepaper, 2009. URL http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[43] NVIDIA. Kepler whitepaper, 2009. URL http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[44] Nvidia. NVIDIA OpenCL Best Practices Guide, 2011. URL http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[45] NVIDIA. NVIDIA Nsight, 2012. URL http://www.nvidia.com/object/nsight.html.

[46] NVIDIA. OpenCL Developer Zone, 2012. URL http://developer.nvidia.com/opencl.

[47] Nvidia. OpenCL Programming Guide Version 4.2, 2012. URL http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf.

[48] NVIDIA, 2012. URL http://blogs.nvidia.com/category/supercomputing/.

[49] NVIDIA, 2012. URL http://blogs.nvidia.com/2012/07/new-top500-list-4x-more-gpu-supercomputers/.

[50] OpenACC, 2012. URL http://openacc.org/.

[51] OpenCLnews, 2012. URL http://openclnews.com/.

[52] OpenHMPP, 2012. URL http://www.openhmpp.org/en/OpenHMPPConsortium.aspx.

[53] Par4All, 2012. URL http://www.par4all.org/.

[54] ITPro Portal, 2012. URL http://www.itproportal.com/2012/08/01/a-close-look-at-xeon-phi-intels-50-core-beast/.

[55] OpenCL Studio, 2012. URL http://www.opencldev.com/.

[56] ViennaCL, 2012. URL http://viennacl.sourceforge.net/.

[57] HPC Wire, 2012. URL http://www.hpcwire.com/hpcwire/2012-01-26/nvidia_releases_upgraded_cuda_compiler,_visual_profiler,_and_npp_library.html.

[58] HPC Wire, 2012. URL http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html.

[59] HPC Wire, 2012. URL http://www.hpcwire.com/hpcwire/2012-07-18/researchers_squeeze_gpu_performance_from_11_big_science_apps.html.

[60] HPC Wire, 2012. URL http://www.hpcwire.com/hpcwire/2012-05-15/nvidia_launches_kepler_into_hpc.html.

[61] HPC Wire, 2012. URL http://www.hpcwire.com/hpcwire/2012-06-20/openacc_group_reports_expanding_support_for_accelerator_programming_standard.html.

[62] Peter Zaspel. Institut für Numerische Simulation - Dipl.-Inform. Peter Zaspel, 2012. URL http://wissrech.ins.uni-bonn.de/people/zaspel.html.

[63] Peter Zaspel and Michael Griebel. A Massively Parallel Two-Phase Solver for Incompressible Fluids on Multi-GPU Clusters. Presentation at GPU Technology Conference 2012, 2012. URL http://developer.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0044-GTC2012-Parallel-Fluids-Clusters.pdf.

[64] Peter Zaspel and Michael Griebel. Solving incompressible two-phase flows on multi-GPU clusters. *Computers and Fluids*, 2012.

A Appendix: Visualisations

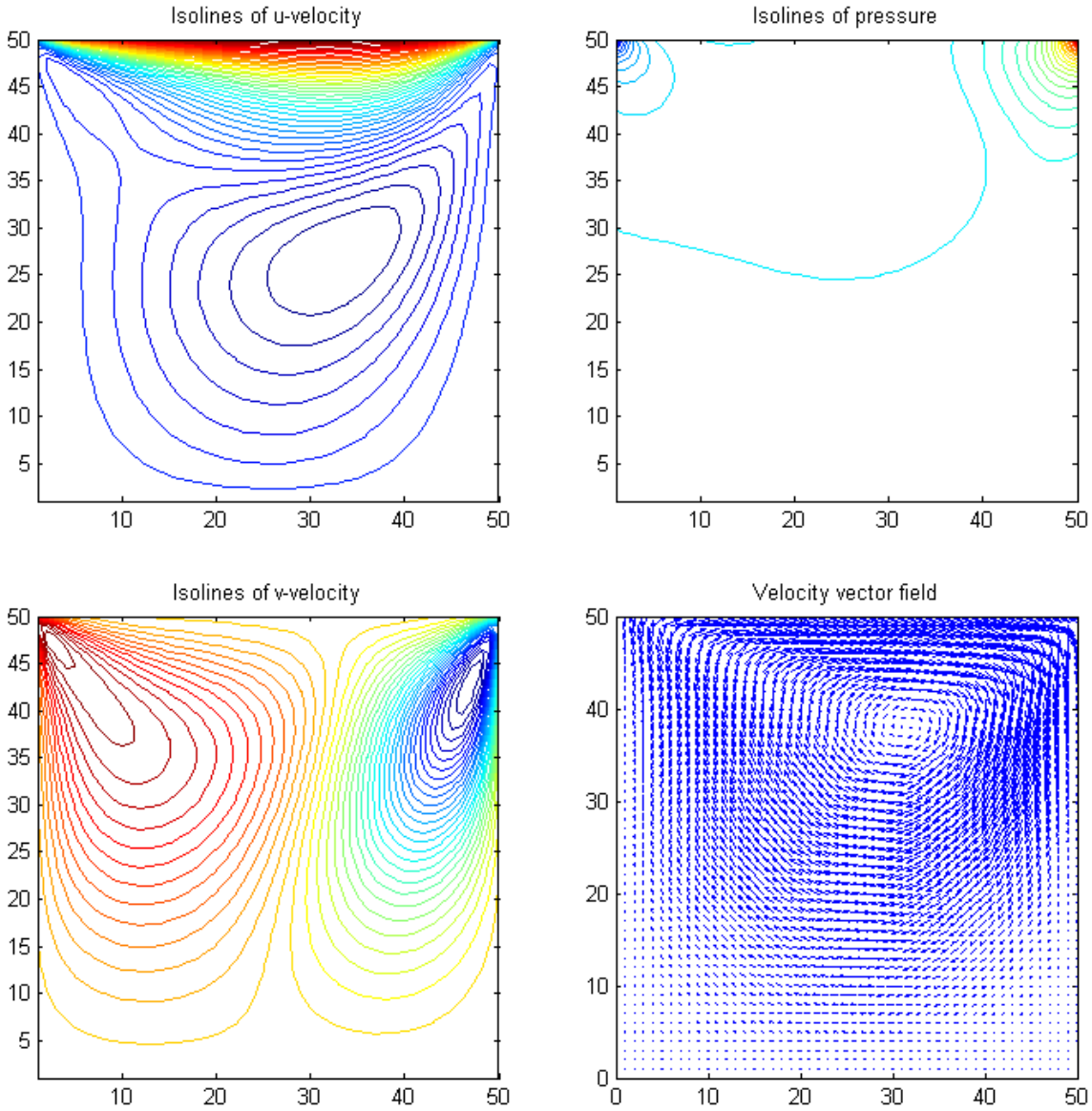

Figure 1: Visualisation of Lid-Driven Cavity Problem $t_{end} = 5s$, $Re = 100$. An example of how data provided from the program can be visualised using Matlab script

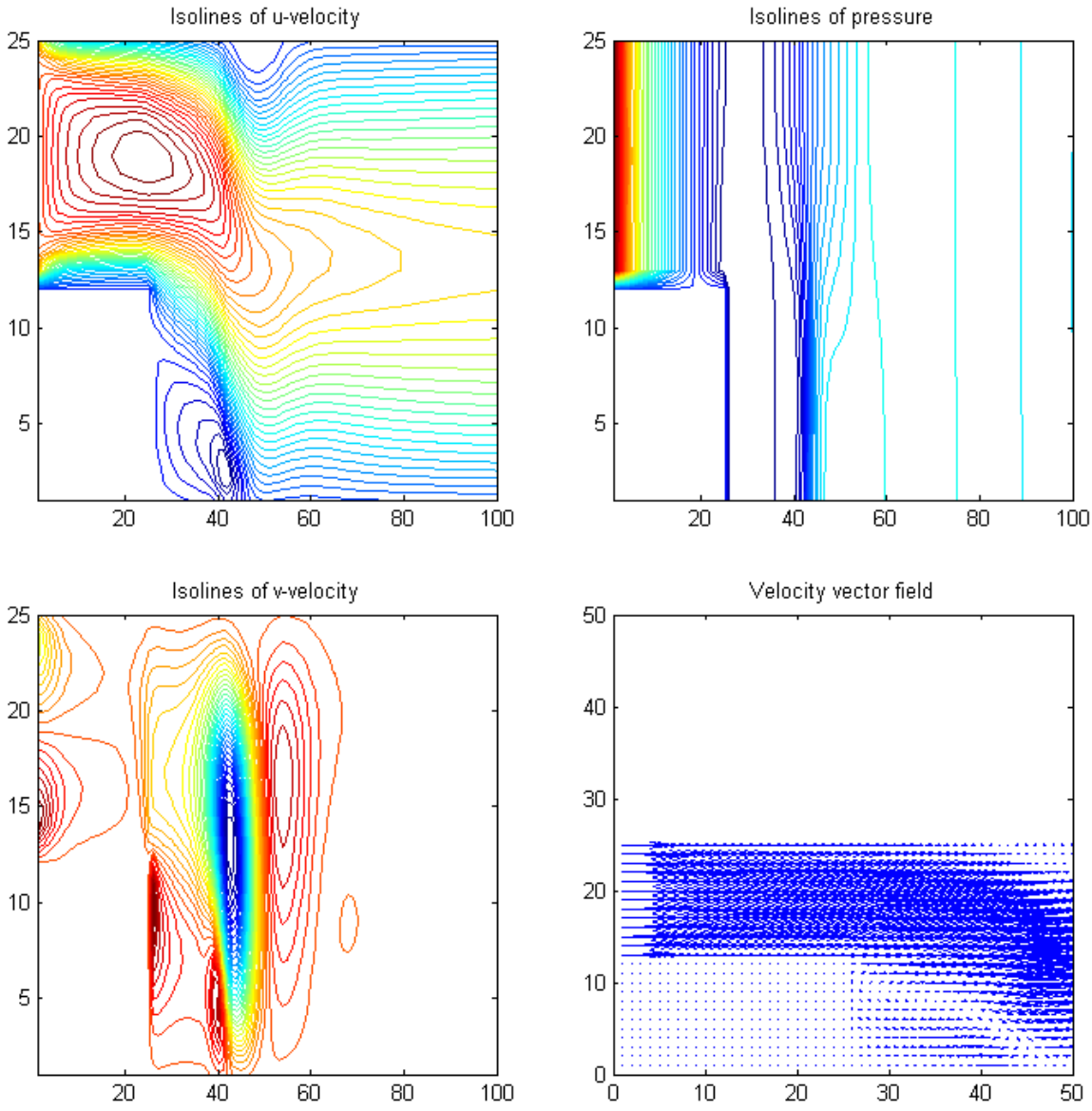

Figure 2: Visualisation of Flow over a Backward-Facing Step Problem $t_{end} = 10s$, $Re = 500$. An example of how data provided from the program can be visualised using Matlab script

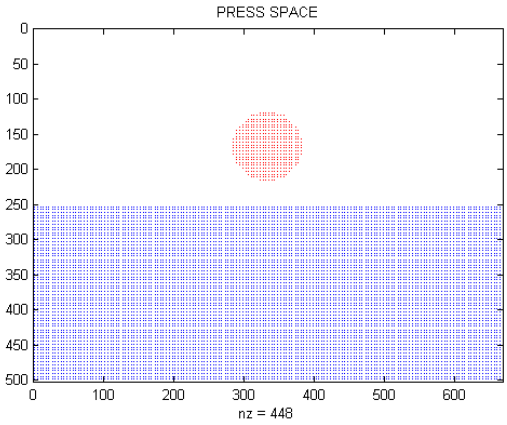

Figure 3: Splash of a liquid drop, time evolution at $Re = 40$

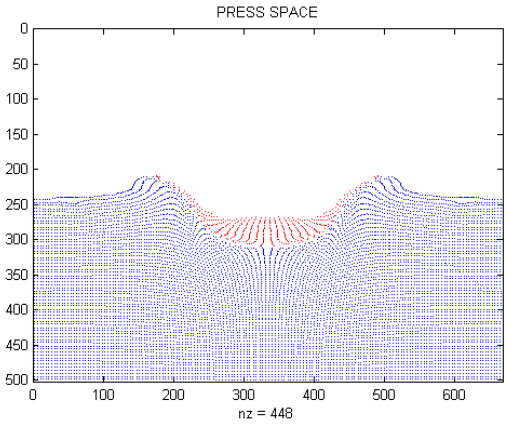

Figure 4: Splash of a liquid drop, time evolution at $Re = 40$

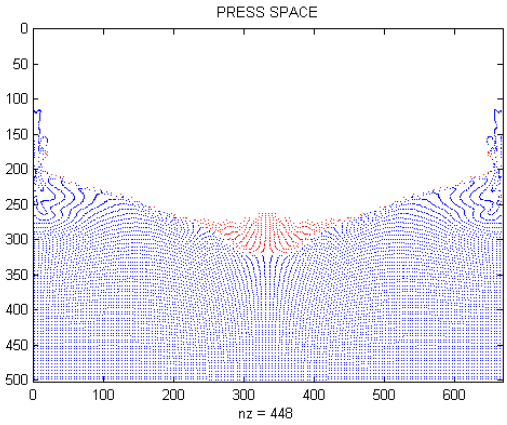

Figure 5: Splash of a liquid drop, time evolution at $Re = 40$

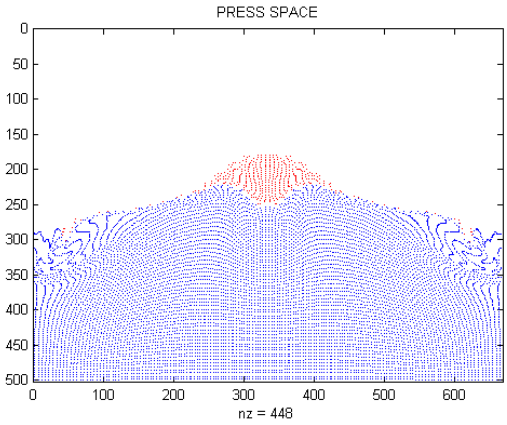

Figure 6: Splash of a liquid drop, time evolution at $Re = 40$